FROM COMPILER FOUNDATIONS TO FRONTIER SYSTEMS

Building an ML Compiler from Scratch

Compiler construction, GPU engineering, numerical analysis, dynamic execution, and frontier systems.

https://github.com/StarAtNyte/ml-compiler-from-scratch

Contents

1. Preface: How to Read This Book
2. Foundations: From Programming to Tensor Compilers
3. 1. The UOp: A Graph IR
4. 2. Term Rewriting and the Rewrite Engine
5. 3. Constant folding and symbolic algebra
6. 4. Memory, rendering C, and the runtime
7. 5. Loops and the linearizer
8. 6. Movement ops and rangeify
9. 7. Kernel fusion and fission
10. 8. Automatic differentiation
11. 9. A Tensor library and your first MNIST
12. 10. Fast CPU kernels
13. 11. Fusion, recompute, and flash attention
14. 12. GPU architecture and mapping kernels
15. 13. Shared memory and tensor cores
16. 14. Training a GPT
17. 15. Symbolic Shapes, Inference, and Where to Go Next
18. Appendix A. UOp and .uir Reference
19. Appendix B. Testing, Debugging, and Slop Management
20. Appendix C. Glossary and Quiz Bank
21. Appendix D: Complete source code
22. 20. The Syllabus Contract and Completion Gates
23. 21. Control Flow, SSA, Analysis, and Lowering
24. 22. GPU Engineering: Correctness, Throughput, and Attention
25. 23. Symbolic Shapes, Guards, Mutation, and Graph Capture
26. 24. Memory Planning, Precision, and Distributed Execution
27. 25. From Course Completion to Independent Engineering
28. 26. Numerical Analysis for Compiler and Kernel Engineers

29. 27. Scheduling, Autotuning, and Backend Construction
30. 28. Irregular Workloads and Heterogeneous Accelerators
31. 29. A Frontier-Lab Preparation Map
32. 30. Building a Source Compiler: Contracts, CFGs, and SSA Construction
33. 31. Optimization with Executable-Path Reasoning
34. 32. Executing SSA, Destroying Phis, and Allocating Registers
35. 33. A Dynamic Tensor Compiler with Fusion and Planned Memory
36. 34. Differentiating and Training Through the Dynamic Compiler
37. 35. Register-Tiled GEMM on a Real GPU

Preface: How to Read This Book

You type `loss.backward()` and `optimizer.step()`, and a few milliseconds later your laptop has done a few billion floating point operations in exactly the right order. Somewhere between the Python and the silicon there is a program that turned a description of *what* to compute into instructions for *how* to compute it: which loops, in what order, fused how, tiled how, on how many cores. That program is an ML compiler. This book builds one from nothing: no starter code, no hidden library, about 2,700 lines of Python that generate C and CUDA. By the last chapter it trains a GPT on Shakespeare, and you will understand every line of every kernel it ran.

The complete learning path

New to systems programming? Start with Foundations, whose labs cover programming, C memory, tensor indexing, differentiation, and measurement. Chapters 1–15 then build the reference compiler. After the appendices, Chapters 20–29 develop compiler theory, real GPU optimization, dynamic execution, runtime systems, and research practice. The advanced chapters contain worked explanations and implementation assignments. Their completion gates distinguish study, implementation, correctness, and measured performance; the reference compiler does not already implement every extension.

Chapters 30–34 add two implemented compiler pipelines: source through SSA, optimization and machine backends, and dynamic float32 tensors through autodiff, fusion, planned memory, and training. They include full code, tests, worked traces, and a measured CPU baseline. Chapter 35 adds a real-device register-tiled GEMM with recorded CUDA timings.

The syllabus contract maps every supplied course topic to material and evidence. Use it to track the original course while following the expanded self-study sequence.

What you'll have built

The compiler is called `uopc` (after its one data structure, the *UOp*). By the end of the book it does all of this, and so will yours:

- **Week 1–2:** take a small program written as a graph, simplify it with algebra, and compile it to a C function that `gcc` turns into a shared library you call from Python.
- **Week 3–4:** compile *any* tensor program, with loops, reductions, broadcasting, reshapes and transposes, to correct (if slow) C.
- **Week 5:** split programs into kernels, differentiate them automatically, and train an MLP on MNIST. The reference run reaches 94.06% test accuracy at 14.4 ms per step on a laptop CPU (`book/outputs/mnist.txt`).
- **Week 6:** make CPU kernels fast. A 1024×1024 matrix multiply goes from 5.2 GFLOP/s (naive loops) to 321.4 GFLOP/s through the Tensor API, against 449.4 GFLOP/s for numpy's OpenBLAS (`book/outputs/bench_gemm_1024.txt`). Laptop numbers vary between runs, so expect yours to differ. Flash attention, three ways.
- **Week 7–8:** map kernels onto GPUs: blocks, threads, shared memory, tensor cores, rendered as CUDA.
- **Week 9–10:** train a 112.6k-parameter GPT from scratch at 235 ms per step, and sample from it:

```
step     0  train 4.266  val 3.827    8933 ms/step
step   100  train 2.616  val 2.559     288 ms/step
step   599  train 2.015  val 2.110     235 ms/step

ICGLOUTery man!

For apbeng mantound rear:
```

(Small model, 600 steps, CPU. It has learned that Shakespeare has speaker names in capitals, line breaks, and colons. It has not learned English. Neither had you at that age.)

A first taste

Here is the whole compiler in one breath. Six lines of tensor code:

```
import numpy as np
from uopc import Tensor
x = Tensor(np.arange(6, dtype=np.float32).reshape(2, 3))
y = (x * 2 + 1).relu().sum(1)
print(y.numpy())
```

Run it with `OPT=0 DEBUG=4` (no optimizations, print every generated kernel) and you get this, verbatim:

```
#include <math.h>
#include <stdbool.h>
static inline int floordiv_i32(int a, int b) { int q = a / b; return (q * b != a && ((a < 0) !=
(b < 0))) ? q - 1 : q; }
static inline int floormod_i32(int a, int b) { int r = a % b; return (r != 0 && ((r < 0) != (b <
0))) ? r + b : r; }
void r_2(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 2; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    for (int ridx1 = 0; ridx1 < 3; ridx1++) {
      float val1 = acc0;
      float val2 = data0[((ridx0*3)+ridx1)];
      float alu3 = ((val2*2.0f)+1.0f);
      acc0 = (val1+((0.0f<alu3)?alu3:0.0f));
    }
    float val4 = acc0;
    data1[ridx0] = val4;
  }
}

  r_2                                0.114 ms     0.00 GFLOP/s  (again: 0.022 ms)
realize: 1 kernels, schedule 120.1 ms, run 0.2 ms
[ 9. 27.]
```

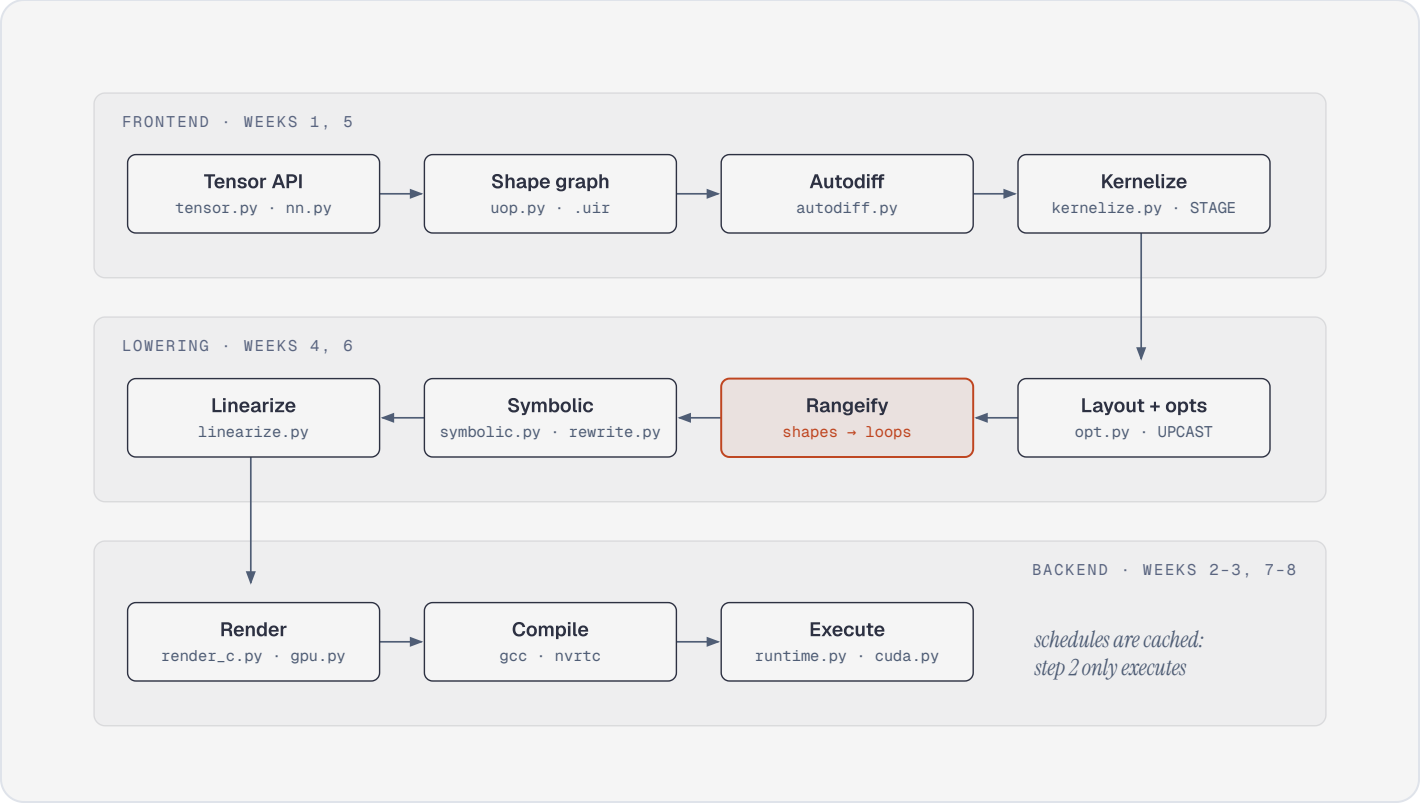
Everything in this book is in that listing somewhere. The multiply, add and ReLU were *fused* into one loop body instead of three passes over memory (Chapter 7). The `sum(1)` became an accumulator `acc0` and an inner loop `ridx1` (Chapter 6). The index `ridx0*3 + ridx1` came from the 2×3 shape being flattened onto a 1-D buffer, then simplified by algebra (Chapter 3). `relu` became `(0.0f<alu3)?alu3:0.0f`, because the compiler has no ReLU, only a comparison and a select (Chapter 1). The loops were opened and closed in the right places by the linearizer (Chapter 5). The C was compiled by `gcc`, cached on disk by a hash of its source, loaded with `ctypes` and called on numpy arrays (Chapter 4). The 120 ms of "schedule" is the Python compiler running for the first time; the second time this program shape is seen, that cost disappears (Chapter 7).

Drop `OPT=0` and the default optimizer notices that the outer loop only has 2 iterations and unrolls it into straight-line code with two accumulators, `acc0` and `acc1`. That is Chapter 10.

> **Key idea:** an ML compiler does not "translate" your program the way a C compiler translates a statement. It *rewrites a graph* over and over, each rewrite moving the description one step closer to loops over

> memory, until what is left can be printed as C.

The map



Read the map in a snake: left to right across the top, right to left across the middle, left to right across the bottom. Each box is one module of `uopc/`, and each is one or two chapters.

Stage	What it does	Module(s)	Lines	Chapter
Tensor API	Lazy tensors: every op just builds graph nodes	`tensor.py`, `nn.py`	326 + 100	9
Shape graph	The IR: hash-consed DAG of UOps, type checking, text format	`uop.py`, `dtype.py`, `uir.py`	319 + 72 + 177	1
Rewriting	Pattern matching and the graph rewrite engine	`rewrite.py`	115	2
Symbolic	Constant folding, algebra, bounds, evaluation by rewriting	`symbolic.py`	295	3 (and 2)
Autodiff	Build the gradient graph from the forward graph	`autodiff.py`	82	8
Kernelize	Cut the graph into kernels, schedule them, cache the plan	`kernelize.py`	207	7
Layout + opts	Upcast, unroll, thread, choose memory layouts	`opt.py`	155	10, 11
Rangeify	Shapes become loops: push indices through movement ops	`rangeify.py`, `composite.py`	191 + 92	6
Linearize	Order the graph into a list with properly nested loops	`linearize.py`	127	5
Render	Print C (or CUDA)	`render_c.py`, `gpu.py`	143 + 170	4, 12, 13
Compile + execute	gcc/NVRTC, load, call	`runtime.py`, `cuda.py`	66 + 90	4, 12

The line counts are real (`wc -l uopc/*.py` gives 2,740 in total, including docstrings). Nothing in the pipeline is large. What makes a compiler hard is not the size of any one pass; it is that every pass trusts every earlier pass, so a small mistake in week 1 surfaces as a wrong number in week 9.

Who this book is for

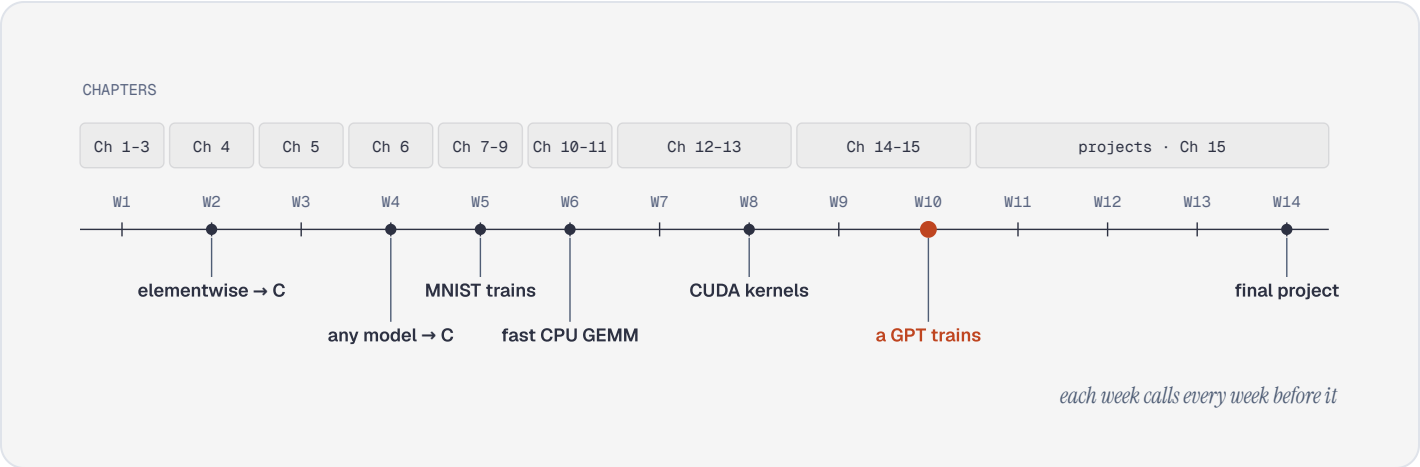
You are a strong programmer. You have written a few thousand lines of C and know what a pointer, a cache line and a segfault are (CMU's 15-213, *Computer Systems: A Programmer's Perspective*, is the reference level). You are comfortable in Python, including classes, closures and `dataclasses`. You have trained a neural network with PyTorch or similar and know roughly what a matrix multiply, a softmax and backpropagation are.

You do *not* need to have taken a compilers course, to know any GPU programming, or to know anything about term rewriting, interval arithmetic, autodiff internals or flash attention. Each of those is taught from first principles when it is needed.

How the book maps onto the course

The book is the companion text for a 14-week, project-based course (`book/SYLLABUS.md`). The course gives a spec each week and tests to pass; the book explains how to meet the spec and why the reference solution looks the way it does.

Week	Topic	Chapters	Milestone
1	UOps, rewriting, symbolic	1, 2, 3	Run programs by rewriting
2	Memory, rendering, runtime	4	Elementwise programs compiled to C
3	Loops	5	GEMM and conv written by hand
4	Movement ops, rangeify	6	Any model compiles to C (slowly)
5	Kernels, autodiff, Tensor	7, 8, 9	MNIST trains
6	Fast CPU kernels, fusion	10, 11	SOTA-competitive CPU GEMM
7–8	GPUs, shared memory, tensor cores	12, 13	CUDA kernels
9–10	Real models	14, 15	A GPT trains
11–14	Projects	15, appendices	Your own extension



The course is cumulative in a way most courses are not. Week 5 does not replace week 1's code; it calls it, thousands of times per training step. The syllabus calls the course "an exercise in slop management", and it means it: if your week-1 rewrite engine is quadratic, your week-9 GPT compiles overnight; if your week-1 hash consing confuses `1` with `1.0`, your week-5 gradients are silently wrong.

Setting up

You need Python 3.11 or newer, a C compiler with OpenMP, and (optionally) an NVIDIA GPU with the CUDA toolkit. The reference machine for every number in this book is an Intel i7-13620H laptop (6 performance + 4 efficiency cores) with Python 3.12.13 and gcc 15.2.0 (`book/outputs/machine.txt`).

```
git clone <your fork of the course repo> ml-compilers && cd ml-compilers
python3 -m venv .venv                # or: uv venv
.venv/bin/pip install -e '.[dev]'    # or: uv pip install -e '.[dev]'
```

The only runtime dependency is numpy; `[dev]` adds pytest (tests), and markdown and pygments (to build this book). The `pyproject.toml` is short enough to read in full:

pyproject.toml

```
[project]
name = "uopc"
version = "0.1.0"
description = "A machine learning compiler built from scratch, one week at a time"
requires-python = ">=3.11"
dependencies = ["numpy"]

[project.optional-dependencies]
dev = ["pytest", "markdown", "pygments"]

[build-system]
requires = ["setuptools>=61"]
build-backend = "setuptools.build_meta"

[tool.setuptools]
packages = ["uopc", "minicc", "tensorjit", "labs", "labs.kernels"]

[tool.setuptools.package-data]
"labs.kernels" = ["*.cuh"]

[tool.pytest.ini_options]
testpaths = ["tests"]
```

Check that your C compiler understands the flags the runtime uses. `uopc/runtime.py` compiles every kernel with `gcc -O3 -march=native -fopenmp -ffp-contract=fast -fassociative-math -fno-signed-zeros -fno-trapping-math` (Chapter 10 explains each one). On Linux, gcc has OpenMP built in. On macOS,

Apple's clang does not support `-fopenmp` out of the box; install gcc from Homebrew and set `CC=gcc-14` (or whatever version you installed).

Then run the test suite. On the reference machine:

```
.....................................................................s...  [100%]
70 passed, 1 skipped in 10.78s
```

The one skipped test is `tests/test_w07_gpu.py::test_cuda_smem_gemm_on_gpu`, which needs a working NVIDIA driver. The reference machine did not have one, so all CUDA in this book is tested by rendering it and by running the same index math through a CPU emulator (Chapter 12). If you have a GPU, that test will run.

A tour of the repository

```
ml-compilers/
├── uopc/            the compiler (one module per stage; see the map)
├── tests/           test_w01_uops.py ... test_w09_models.py: one file per week
├── examples/        mnist_mlp.py, gpt.py, flash_attention.py, bench_gemm.py, datasets.py
├── book/
│   ├── SYLLABUS.md  the course spec
│   ├── src/         this book: 20 markdown files (preface, 15 chapters, appendices A–D), plus
diagrams/
│   ├── outputs/     captured runs quoted in the text (benchmarks, training logs)
│   └── site/        the built HTML
└── tools/
    └── build_book.py   builds book/src -> book/site
```

The package's own docstring is the one-paragraph version of the course:

`uopc/__init__.py`

```python
"""uopc: a machine learning compiler built from scratch, one week at a time.

  week 1   uop, rewrite, symbolic, uir      the IR, rewriting, algebra, the text format
  week 2   render_c, runtime, linearize     memory, C code, compiling and running it
  week 3   linearize                        loops and scopes
  week 4   rangeify, composite              shapes -> loops, PAD/FLIP from primitives
  week 5   kernelize, autodiff, tensor, nn  fusion/fission, gradients, a Tensor library
  week 6   opt                              upcast, unroll, threads, layout
  week 7-8 gpu, cuda                        GPU mapping, CUDA, shared memory, tensor cores
"""
from uopc.uop import UOp, Ops, AxisType, ParamArg, CallArg, RangeArg, sink, range_
from uopc.dtype import dtypes, DType, AddrSpace
from uopc.tensor import Tensor
```

`uopc/`: the compiler

Every module starts with a docstring saying which week it belongs to and what it does. Read those docstrings first when you open a file; they are written to be the summary you would want. Imports only point from later weeks to earlier ones: `uop.py` imports nothing but `dtype.py`; `rewrite.py` imports `uop.py`; `symbolic.py` imports both; `rangeify.py` imports `symbolic.py`; and so on up to `tensor.py`, which imports almost everything. There are no cycles, which means you can build and test the modules in the order the chapters present them.

`tests/`: the oracle

File	Tests	What it checks
`test_w01_uops.py`	12	hash consing, type checking, rewriting, symbolic, evaluation, `.uir`
`test_w02_render.py`	4	loopless C programs over memory
`test_w03_loops.py`	3	hand-written GEMM and conv in the loop IR
`test_w04_rangeify.py`	6	movement ops and reductions against numpy
`test_w05_tensor.py`	5	kernels, gradients (finite differences), the Tensor API
`test_w06_opt.py`	5	optimized kernels still give the same answers
`test_w07_gpu.py`	6	CUDA rendering, the emulator, shared memory, tensor cores
`test_w09_models.py`	3	small end-to-end models

Run one week, one test, or stop at the first failure:

```
.venv/bin/python -m pytest tests/test_w01_uops.py -v
.venv/bin/python -m pytest tests/test_w01_uops.py -k hash_consing
.venv/bin/python -m pytest -x
```

The tests are an *oracle* in the course's sense: they define correct behavior. When you write your own compiler, the strongest thing you can do is make these tests pass against your code, then write more. Appendix B is about how.

`examples/`

`mnist_mlp.py` (week 5), `bench_gemm.py` and `flash_attention.py` (week 6), and `gpt.py` (weeks 9–10) are complete programs that use only the `Tensor` API. `datasets.py` downloads MNIST and Tiny Shakespeare into `~/.cache/uopc/data` the first time you need them.

`book/`

The markdown sources are in `book/src/`: this preface, 15 chapters and four appendices, 20 files in all. Code listings are not copied into the text; they are pulled out of the repository when the book is built, so a listing can never drift from the code it shows. The same mechanism produces two places to read whole files. Each chapter ends with a collapsed "Full code for this chapter" section holding the modules and tests it teaches, and Appendix D (`19-appendix-source.md`) holds the complete source of `uopc/`, `tests/` and `examples/`, one collapsed block per file.

Conventions

Listings

A listing with a label above it, like this one, is the *real code*, extracted from the file named in the label at build time:

`uopc/uop.py · prod`

```
def prod(xs) -> int: return math.prod(xs)
```

The label `uopc/uop.py · prod` means "the definition named `prod` in `uopc/uop.py`". A label like `uopc/uir.py:37-45` means a line range. Listings without a label are illustrations: REPL sessions, wrong versions to contrast with right ones, or generated C. Any output shown in this book was produced by running the code on the reference machine, and pasted unedited (timings will differ on your machine).

The code style is dense: two-space indentation, one-line `if`s, little vertical whitespace. That is a deliberate choice for a book. A function that fits on one screen can be read as a whole, and every chapter's listings would be twice as long in a more conventional style. Your own code does not need to look like this; it needs to be as *simple* as this.

Environment variables

Variable	Values	Effect	Where
`DEBUG`	`1`	one line per `realize`: kernel count, schedule time, run time	`kernelize.py`
	`2`	also one line per kernel: time and GFLOP/s	
	`3`	also run each kernel a second time (warm caches) and report that time	
	`4`	also print the generated source of every newly compiled kernel	
`OPT`	`0`	no kernel optimizations: plain loops	`tensor.py`
	`1` (default)	hand-written heuristics (`HandOpt`)	`opt.py`
	`2`	search: time candidate optimizations and keep the fastest (`SearchOpt`)	`opt.py`
`CC`	`gcc` (default)	the C compiler	`runtime.py`
`CFLAGS`	see above	override all compiler flags (e.g. strict IEEE ordering)	`runtime.py`
`UOPC_CACHE`	`~/.cache/uopc`	where compiled `.so` files are cached	`runtime.py`

Levels are cumulative: `DEBUG=4` prints everything `DEBUG=2` does, plus the source. Note that `DEBUG=4` only prints a kernel the first time it is *compiled* in this process; a cached plan is not printed again.

Words

A *UOp* (micro-op) is one node of the graph. An *op* is its kind (`Ops.ADD`). A *kernel* is one generated C function. *Realize* means "actually compute this tensor now". Appendix C has the full glossary; each chapter defines its terms the first time it uses them.

How each chapter works

Every chapter after this one has the same skeleton:

1. **A hook**: the problem the chapter solves, in one paragraph.
2. **What you'll build** and **Where this fits**: the deliverables, and links to the neighbors.
3. **Concept sections**, each of which explains *why* before *how*, with real code, real output and a diagram wherever structure is easier to see than to read.
4. **Pitfalls**: bugs that actually happened while building the reference compiler. Not hypothetical "be careful with X" warnings: real failures with real symptoms, and the fix. If you are stuck on a bug, read the Pitfalls of the chapter you are in first.
5. **Exercises**, graded ★ (an hour or less), ★★ (an afternoon) and ★★★ (a project). Some are "implement X", some are "predict the output", some are "find the bug". ★ exercises have solutions in collapsed blocks. Try before you peek.
6. **A quiz** in the style of the course's in-person quizzes: short questions that check you understood the idea, not that you memorized the code.
7. **Further reading**: papers, books and source code worth your time.
8. **Full code for this chapter**: the complete modules and tests the chapter teaches, collapsed. It is generated when the book is built, so it always matches the repository. Appendix D has every file.

Three ways to read

- **Taking the course.** Read the chapters for the week *before* the week starts, implement the spec in your own repository, and use the reference code only when you are stuck or when you are done and want to

compare. The exercises marked "implement" are often exactly the assignment.

- **Self-study.** Read in order. For each chapter, re-implement the module from the prose and the tests, then diff your design against `uopc/`. Expect weeks 1, 4 and 5 to take the longest.
- **Reading to understand ML compilers** (you have used XLA, TVM, Triton or `torch.compile` and want to know what is inside). Read Chapters 1, 2, 6, 7 and 10 in order, skim the rest. Chapter 15 compares `uopc` to the production compilers.

Slop management

The syllabus grades style with three questions: *Did you write slop? Can I read your code? Can it be simpler?* Those questions are the most important technical advice in the course, so here is what they mean in practice.

Small. The whole compiler is 2,740 lines. When a module grows past a few hundred lines, something is usually doing two jobs. The UOp, for example, is one class with three fields; there is no `AddUOp`, no `ConstUOp`, no visitor hierarchy. Chapter 1 explains why that uniformity makes everything after it shorter.

Tested. Every week adds a test file, and every test is an end-to-end check against an independent oracle: numpy, finite differences, Python's own arithmetic, or the C program's actual output. A pass that has not been run against an oracle is a pass that is wrong in a way you have not found yet.

Readable. Names say what things are (`call_params`, `inline_call`, `graph_rewrite`). Docstrings say why, not what. Errors are raised where the mistake is made, with a message that names the problem (`Ops.ADD: dtype mismatch dtypes.i32 vs dtypes.f32 (use CAST)`), never three passes later as a `KeyError`.

Failures on record. The file `book/NOTES.md` is a lab notebook of real bugs hit while building `uopc`: hash-cache collisions between `1` and `1.0`, a linearizer that took four designs, an unsimplified index that made attention $O(N^3)$, a fusion decision that recomputed a LayerNorm for every output column. Each one is told in full in the Pitfalls section of the chapter it belongs to, with the numbers. Learning to recognize these failure shapes is most of what experience in compilers is.

Simple beats clever until measured. Every performance trick in Part IV is introduced with a benchmark that shows the problem first. Before that, the rule is: the simplest correct thing.

Acknowledgments and lineage

The design of `uopc` is heavily inspired by tinygrad, whose UOp IR, pattern matcher and "rangeify" lowering are the direct ancestors of the ones in this book. `uopc` is much smaller and makes different trade-offs (it is a teaching compiler; tinygrad runs real workloads on many backends), but if you finish this book, reading tinygrad's source is the natural next step, and Chapter 15 gives a guided route through it. The course syllabus (`book/SYLLABUS.md`) defines the week-by-week spec that the chapters follow.

Warm-up exercises

1. ★ **Run the tests.** Set up the environment and run the whole test suite. Then run `.venv/bin/python -m pytest tests/test_w01_uops.py -v` and read the names of the 12 tests. Which chapters do you expect each to belong to?
2. ★ **Read the C.** Run the "first taste" program from this preface with `DEBUG=4` and then `OPT=0 DEBUG=4`. Find the two differences in the generated C and explain them in one sentence each.
3. ★ **Predict the loops.** Change the program to `(x * 2 + 1).relu().sum(0)` (sum over the other axis). Before running it, predict the loop bounds and the index expression into `data0`. Then check.
4. ★ **The import graph.** Run `wc -l uopc/*.py` and draw the import graph of `uopc/` by reading the `import` lines at the top of each file. Confirm that it has no cycles.

Solutions to ★ exercises

1. Hash consing, type inference and `.uir` belong to Chapter 1; `test_custom_pattern_matcher`, `test_run_stack_program` and `test_call_inlining_and_scoping` to Chapter 2; `test_const_fold_semantics` and the two `test_symbolic_*` tests to Chapter 3; the deep-graph test to Chapters 1 and 2 (both `toposort` and `graph_rewrite` must be iterative).
2. With the default `OPT=1`, the two-iteration outer loop is upcast: it disappears, and its two iterations become two accumulators (`acc0`, `acc1`) updated in one inner loop. The index into `data0` becomes `ridx0` and `ridx0+3` instead of `ridx0*3+ridx1`.
3. The output has 3 elements and the reduction runs over 2 rows, so (with `OPT=0`) the outer loop runs to 3, the inner loop to 2, and the index is `ridx1*3 + ridx0`: the reduction now strides through memory.
4. The spine is `dtype.py` ← `uop.py` ← `rewrite.py` ← `symbolic.py` ← `rangeify.py` ← `kernelize.py` ← `tensor.py` ← `nn.py`, with side branches (`render_c.py` ← `gpu.py`, `composite.py` ← `autodiff.py`,

> `runtime.py` ← `kernelize.py`). Every import points from a later week to an earlier one (or within a week), so there is no cycle. The one surprise is at the bottom of `tensor.py`: it imports `opt.py` *after* the `Tensor` class is defined, to pick the default optimizer.

Further reading

- Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A Programmer's Perspective* (3rd ed., 2015). The prerequisite level for this book, and the best reference for the memory hierarchy chapters.
- Keith D. Cooper and Linda Torczon, *Engineering a Compiler* (2nd ed., 2011). A conventional compilers textbook, useful as a contrast: most of it is about parsing and register allocation, which ML compilers barely need.
- The tinygrad repository, https://github.com/tinygrad/tinygrad. Start with `tinygrad/uop/ops.py` after Chapter 2.
- Tianqi Chen et al., "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning", OSDI 2018. The paper that made "ML compiler" a field; compare its design to `uopc` after Chapter 10.

Foundations: From Programming to Tensor Compilers

Start here if the original course prerequisites are unfamiliar. The course's “no formal prerequisites” means no enrollment requirement; it does not mean no background knowledge. Work through these labs before Chapter 1. Experienced readers may skip a lab after passing its exit check.

This book has two sequences. Chapters 1–15 build the compact `uopc` compiler. Chapters 20–29 extend the learning path into compiler analysis, GPU performance, dynamic execution, runtime engineering, and independent research. Those later labs are assignments to implement, not claims that all their features already exist in `uopc`. The syllabus contract maps the supplied course specification to chapters, tests, and remaining hardware evidence.

F.1 Programming and debugging

If programming itself is new, first learn variables, functions, loops, arrays, classes, exceptions, and tests. Write a command-line calculator, then an interpreter for nested expressions such as `("add", ("mul", 2, 3), 4)`. The interpreter should return 10. Extend it with named variables and report an undefined variable as an error. This is the execution model Chapter 1 later represents as a graph.

Python objects have identity as well as value. Two equal expressions can be separate objects; hash consing deliberately makes structurally identical immutable expressions share an object. Implement an immutable expression class with `(operation, operands)` as its key. Never mutate an object after using its structure as a dictionary key.

Lab: build the interpreter, add ten tests, deliberately introduce a wrong multiplication result, and use a debugger to locate it. Commit the working version with Git, break it, and restore it. **Exit check:** explain identity versus equality, recursion versus iteration, and why a shared graph node must not be modified in place.

F.2 C, memory, and the operating system

A pointer is an address interpreted with an element type. For `float *p`, `p + 3` advances three floats, not three bytes. In a row-major `M × N` matrix, element `(i,j)` lives at `p[i*N+j]`. The array's shape is metadata; the allocation itself is a sequence of bytes.

A local array's lifetime ends when its function returns. A heap allocation persists until released. A shared library exports compiled functions which Python can call through a foreign function interface. The ABI determines how pointers and scalar arguments are passed; mismatching a function signature can corrupt memory even when the source code looks plausible.

Lab: implement vector addition and rectangular matrix multiplication in C. Compile with warnings and debug symbols; run memory checks with address and undefined-behavior sanitizers when supported. Call vector addition from Python using `ctypes`, declaring the argument types. Compare every result with a simple Python reference. Exercise lengths 1, 7, and 257 and matrix dimensions `(3,5,7)`. **Exit check:** draw the memory layout of a 2×3 array and explain why returning a pointer to a local array is invalid. Know how to inspect a crash and separate a compiler error from a runtime error.

F.3 Linear algebra and tensor indexing

Matrix multiplication is a reduction:

```
C[i,j] = sum over k of A[i,k] * B[k,j]
A: (M,K), B: (K,N), C: (M,N)
```

There are approximately `2*M*N*K` floating-point operations when multiply and add count separately. Broadcasting adds repeated logical values without requiring repeated physical storage. A vector of length N broadcast across M rows can use address `j` for every row i: its row stride is zero. A transpose swaps logical axes and their strides; it need not copy the data. A reshape is a view only when the existing layout supports the requested indexing; arbitrary transposes cannot always be flattened into a contiguous view.

Lab: write `offset(index, strides)` and use it to read contiguous, transposed, and broadcast views. Implement a reduction with explicit loops. For `A=[[1,2],[3,4]]` and `B=[[5,6],[7,8]]`, obtain `[[19,22],`

`[43,50]]`. Derive each address rather than calling a matrix library. **Exit check:** distinguish shape, stride, element count, and byte size; derive the gradient shape for a broadcast addition.

F.4 Floating point and differentiation

Floating-point arithmetic rounds after operations. Consequently `(a+b)+c` can differ from `a+(b+c)`. A compiler that reassociates reductions needs an explicit numerical policy. Use both absolute and relative tolerances when comparing results; a fixed relative tolerance is unhelpful near zero.

For a scalar loss `L` and `z=x*y`, reverse mode accumulates `dL/dx += dL/dz*y` and `dL/dy += dL/dz*x`. Accumulation matters because a value can feed several consumers. For `L=x*x+x` at `x=3`, the derivative is 7, not 3 or 4.

Lab: implement scalar reverse-mode autodiff with addition and multiplication. Compare its derivative with `(L(x+h)-L(x-h))/(2*h)` in double precision for several h. Explain why extremely small h causes cancellation. Extend the engine to a sum and a broadcast operation. **Exit check:** derive a linear layer's input and weight gradients and explain why a gradient check should avoid a nondifferentiable ReLU point.

F.5 Measuring a machine

A cache stores recently accessed memory in lines. Contiguous traversal usually reuses each fetched line; a large stride may fetch many bytes to consume one element. Threads share some resources and compete for bandwidth. More threads do not guarantee a speedup.

Lab: time row-first and column-first array traversals in compiled C, consuming the result so the compiler cannot eliminate the loop. Warm up first, repeat measurements, and record the machine, compiler flags, data sizes, and timing distribution. Separate compilation from execution. **Exit check:** predict whether doubling arithmetic while preserving bytes transferred should double runtime for a bandwidth-limited kernel. Explain the prediction using the roofline model in Chapter 10.

F.6 Study order and equipment

Use the following order; move on by demonstrated ability rather than a fixed calendar.

Stage	Material	Required artifact
Foundations	This chapter	C kernels, tensor views, scalar autodiff
Compiler construction	Chapters 1–9	Tested tensor compiler and MNIST training
Performance and models	Chapters 10–15	Benchmark notebook, GPU kernels, GPT training
Compiler depth	Chapter 21	CFG analysis and an LLVM/MLIR lowering experiment
Hardware depth	Chapter 22	Real GPU profiles and a GEMM optimization ladder
Dynamic systems	Chapters 23–24	Guarded execution, memory planner, precision experiments
Numerical and backend depth	Chapters 26–28	Error analysis, schedule search, irregular operations, target port
Implemented internals	Chapters 30–34	Source-to-machine compiler and dynamic tensor runtime
Independent work	Chapters 25 and 29	Reviewed capstone, external codebase change, frontier specialization

A CPU machine is sufficient for the foundations and compiler construction. GPU indexing can be studied with the emulator, but GPU performance and synchronization milestones require real hardware. Multi-device milestones require two devices or processes, with simulated exercises clearly labeled. Do not report an emulator's timing as GPU performance. Record tool versions and GPU architecture because supported instructions and tooling vary.

1. The UOp: A Graph IR

A compiler cannot work on your program directly. Python source is text; a running PyTorch model is a tangle of objects, C++ and CUDA; neither can be *inspected, transformed and printed* in a uniform way. Before a compiler can optimize anything it needs its own representation of programs: a data structure in which "multiply these two things, then add that" is a value you can look at, compare, take apart and rebuild. That data structure is the compiler's *intermediate representation*, its IR, and its design decides how hard every later chapter will be. This chapter designs the IR for the whole book: one tiny class, `UOp`, three fields wide, which by week 10 will describe a GPT.

What you'll build

- `UOp(op, src, arg)`: an immutable graph node, and the `Ops` enum of operations.
- *Hash consing*: building the same node twice returns the same object, so structural equality is `is`.
- *Recursive properties*: `dtype`, `shape` and `addrspace` inferred at construction time; a type checker that rejects malformed graphs with useful errors.
- Week-1 semantics: `CONST`, `STACK`, `INDEX`, ALU ops, `PARAM` and `CALL`.
- Graph utilities: iterative `toposort`, `substitute`.
- The `.uir` text format: a parser and a printer that round-trip exactly.

Where this fits. This is the first chapter of Part I. The Preface shows where the IR sits in the pipeline. Chapter 2 builds the rewrite engine that transforms these graphs (and uses it to *run* them); Chapter 3 teaches it algebra. Every later chapter adds ops to the enum you meet here, but never a second kind of node.

1.1 What an IR is for

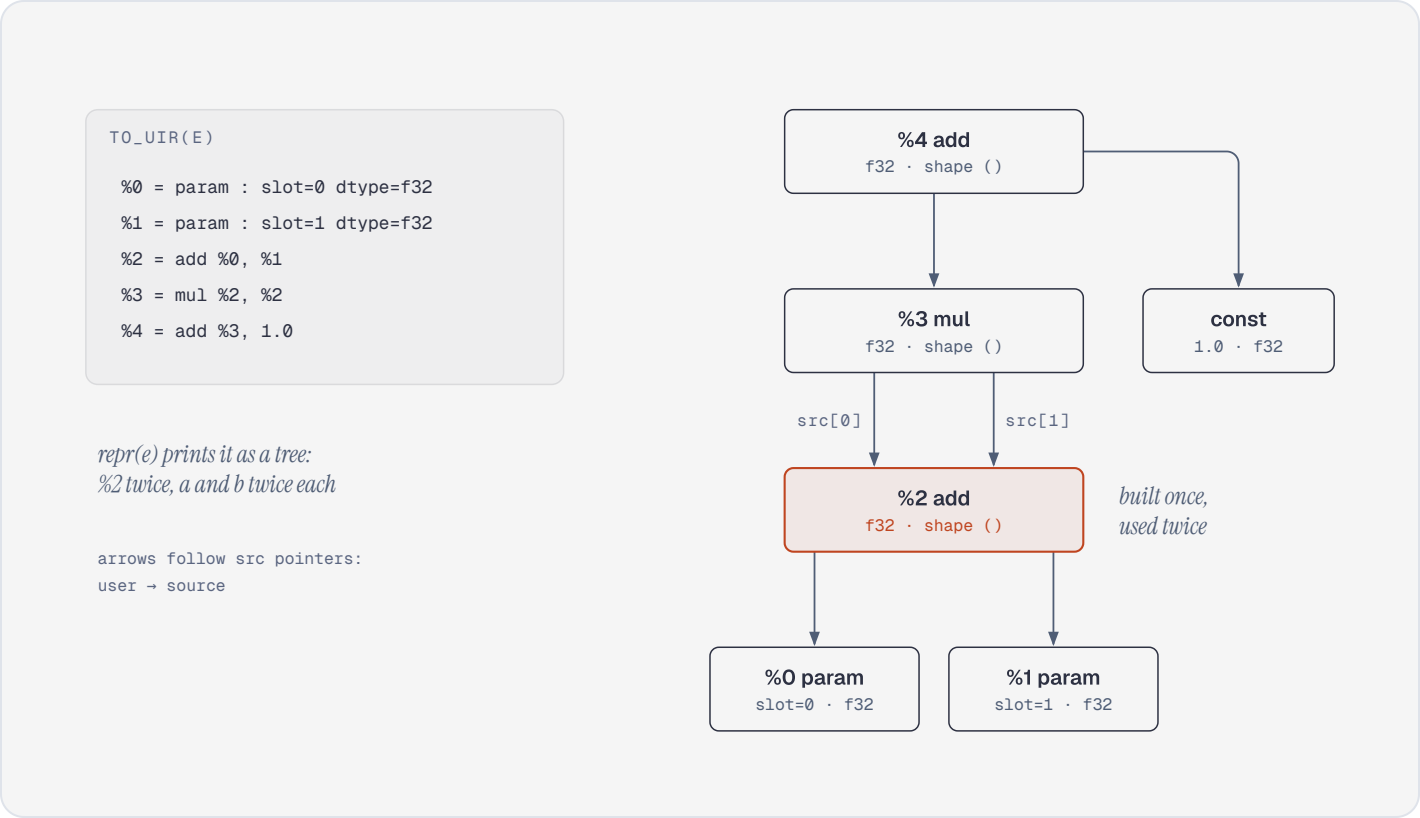
A compiler is a pipeline of translations. In a C compiler, text becomes tokens, tokens become an abstract syntax tree, the tree becomes a lower-level IR (LLVM IR, say), the IR is optimized in dozens of passes, and finally becomes machine code. Each arrow in that chain is easier to write than a single arrow from text to machine code would be, *because each IR is designed to make particular questions easy to answer*:

- An abstract syntax tree answers "what did the programmer write?" It keeps parentheses, variable names and the order of statements.

- A control-flow graph of SSA instructions (LLVM's IR, GCC's GIMPLE) answers "what values flow where, and in which order can things happen?" It forgets names and nesting, and every value is assigned exactly once.
- Machine code answers "what does the CPU do?" and nothing else.

An ML compiler has an unusual source language. The "program" is a computation on tensors that the user assembled at runtime by calling Python functions: `x @ w + b`, `softmax`, `cross_entropy`. There is no text to parse; the frontend (our `Tensor` class, Chapter 9) builds the IR directly as the user calls methods. And the programs are, overwhelmingly, *dataflow*: values are computed from other values, nothing is overwritten, there are no `if` statements that depend on data (a ReLU is a `max`, not a branch) and no loops whose trip count depends on data.

Dataflow programs have a natural representation: a graph whose nodes are operations and whose edges say "this operation consumes that value". Take the expression `(a + b) * (a + b) + 1.0`:



Three properties make this graph a good IR:

1. **It is a DAG, not a tree.** `a + b` is computed once and used twice. A tree (like the AST of the expression as written) would contain two copies of `a + b` and you would need a separate optimization, common subexpression elimination (CSE), to notice they are the same.

2. **It has no order beyond dependencies.** Whether `a` or `b` is "computed first" is not a question the graph asks. That freedom is what lets later passes fuse, reorder, and parallelize.
3. **Every node is a pure value.** A node means "the result of this op on these inputs", always, everywhere. Pure values can be duplicated, shared, moved and compared freely. (Week 2 will add memory and side effects, and we will need explicit ordering edges to keep this property; see Chapter 4.)

Production ML compilers agree: XLA's HLO, TVM's Relay, PyTorch's FX graphs, MLIR's tensor dialects and tinygrad's UOps are all dataflow graphs. Where they differ is in how many *kinds* of nodes they have, and that is our first design decision.

One node type, not fifty

The obvious object-oriented design gives each operation its own class: `Add(lhs, rhs)`, `Const(value)`, `Reshape(x, shape)`, each with its own fields and a visitor method. It feels tidy and type-safe. It is the wrong choice for this compiler, for three reasons:

- **Pattern matching.** Most of the compiler is rewrite rules: "an ADD whose second source is a CONST 0 becomes its first source". If every node has the same shape (an op, a tuple of sources, an argument) one generic matcher handles every rule (Chapter 2). With fifty classes you write fifty matchers, or reflection.
- **Generic traversals.** `toposort`, `substitute`, hashing, printing, pickling and the rewrite engine never need to know what an op means. They work on `(op, src, arg)` and are each a dozen lines.
- **Hash consing.** We are about to make every node unique by its structure. That needs a single key-building function and a single cache.

So a UOp is a triple:

```
UOp(op: Ops, src: tuple[UOp, ...], arg: Any)
```

`op` says what it is, `src` are the inputs (in order), and `arg` holds any non-graph data: the value of a constant, the target dtype of a cast, the new shape of a reshape. That is the entire node. The alternative, a node per op class, is what you find in XLA's C++; tinygrad, which `uopc` follows, uses the triple.

1.2 The `Ops` enum

Here are all the operations the compiler will ever have, grouped by the week they appear:

uopc/uop.py · Ops

```
class Ops(Enum):
  # ---- week 1: values ---------------------------------------------------
  PARAM = auto()      # arg: ParamArg. An input to the enclosing CALL.
  CALL = auto()       # src[0] = body, src[1:] = arguments. arg: CallArg
  CONST = auto()      # arg: python value (bool/int/float). shape ()
  STACK = auto()      # new leading axis; all srcs share one shape
  INDEX = auto()      # src[0] indexed by src[1:]; pops axes off the left
  CAST = auto()       # arg: DType
  RECIP = auto()      # 1/x
  ADD = auto(); MUL = auto(); CMPLT = auto()
  FLOORDIV = auto(); FLOORMOD = auto()
  WHERE = auto()      # (cond, a, b)
  # extra ALU, introduced when we need them (masks, max-reduce, softmax)
  CMPNE = auto(); MAX = auto()
  EXP2 = auto(); LOG2 = auto(); SQRT = auto(); SIN = auto()
  # ---- week 2: memory ---------------------------------------------------
  BUFFER = auto()     # arg: ParamArg. global persistent memory (never in a CALL body)
  ALLOC = auto()      # arg: ParamArg. call-scope storage `float a[10];`
  LOAD = auto()       # (ptr,)
  STORE = auto()      # (ptr, value)
  AFTER = auto()      # (mem, *deps): `mem`, but only once deps have happened
  SINK = auto()       # (*effects): groups side effects, the root of a program
  # ---- week 3: loops ----------------------------------------------------
  RANGE = auto()      # src[0] = end; arg: RangeArg. value is the loop counter
  END = auto()        # (passthrough, *ranges): closes the ranges
  # ---- week 4: movement + reduction -------------------------------------
  RESHAPE = auto(); EXPAND = auto(); PERMUTE = auto(); SHRINK = auto()
  REDUCE = auto()     # arg: (Ops.ADD | Ops.MAX, n) -- reduces the last n axes
  # ---- week 5: kernels --------------------------------------------------
  STAGE = auto()      # materialize src[0] into a temp buffer you can index into
  # ---- weeks 7-8: GPUs --------------------------------------------------
  BARRIER = auto()    # (*deps): __syncthreads()
  WMMA = auto()       # reserved for tensor cores (week 8 lowers GEMMs to WMMA by pattern
instead)

  def __repr__(self): return f"Ops.{self.name}"
  __hash__ = object.__hash__   # members are singletons: identity hashing is correct and much
faster
```

Only the first block matters this week. Notice how *small* it is. There is no subtraction, no negation, no division, no greater-than, no equality, no ReLU. Every one of those is expressible from what is there:

You want	You build	Why it is enough
`x - y`	`x + y * -1`	
`-x`	`x * -1`	
`x / y` (float)	`x * RECIP(y)`	
`x > y`	`CMPLT(y, x)`	swap the operands
`x == y`	`CMPNE(CMPNE(x, y), True)`	"not not-equal"
`relu(x)`	`WHERE(CMPLT(0, x), x, 0)`	
`x // y`, `x % y` (int)	`FLOORDIV`, `FLOORMOD`	primitive: these can't be built from the others

The fewer ops there are, the fewer rules every later pass needs. Every rewrite rule, every renderer case, every gradient rule (Chapter 8) is written once per op. Seven arithmetic ops instead of fifteen halves that work. The cost is that patterns must recognize the *encoded* forms (a rule about subtraction has to look for `x + y*-1`), which Chapter 3 handles.

The week-1 ops, precisely:

Op	src	arg	Result
`CONST`	`()`	a Python `bool`, `int` or `float`	a scalar; dtype from the Python type
`STACK`	`(x0, x1, ..., xn-1)`, all same shape and dtype	`None`	shape `(n, *x0.shape)`
`INDEX`	`(x, i0, ..., ik-1)`, indices are `i32` scalars	`None`	`x` with its first `k` axes removed
`CAST`	`(x,)`	target `DType`	`x` converted, same shape
`RECIP`	`(x,)`, float	`None`	`1/x`
`ADD`, `MUL`	`(a, b)`, same shape and dtype	`None`	same shape and dtype
`CMPLT`	`(a, b)`, same shape and dtype	`None`	`bool`, same shape
`FLOORDIV`, `FLOORMOD`	`(a, b)`, both `i32`	`None`	`i32`, *floor* semantics (Python's, not C's)
`WHERE`	`(cond, a, b)`, `cond` is `bool`	`None`	`a` where `cond` else `b`, elementwise
`PARAM`	`()`	`ParamArg(slot, dtype, shape)`	the `slot`-th input of the enclosing `CALL`
`CALL`	`(body, arg0, ..., argn-1)`	`CallArg(name)`	the body with its PARAMs bound to the args

Why floor division instead of C's truncating division is a subtle and important decision, made in Chapter 3. Python sets that the rest of the compiler uses to talk about groups of ops are defined right after the enum:

uopc/uop.py:59-63

```
UNARY = {Ops.RECIP, Ops.CAST, Ops.EXP2, Ops.LOG2, Ops.SQRT, Ops.SIN}
BINARY = {Ops.ADD, Ops.MUL, Ops.CMPLT, Ops.CMPNE, Ops.FLOORDIV, Ops.FLOORMOD, Ops.MAX}
ALU = UNARY | BINARY | {Ops.WHERE}
COMMUTATIVE = {Ops.ADD, Ops.MUL, Ops.MAX, Ops.CMPNE}
MOVEMENT = {Ops.RESHAPE, Ops.EXPAND, Ops.PERMUTE, Ops.SHRINK}
```

1.3 Data types and address spaces

A dtype is a tiny frozen record: a name, a size in bytes, the C type to print, and the `struct`/numpy format character.

uopc/dtype.py · DType

```python
@dataclass(frozen=True, eq=True)
class DType:
  name: str
  itemsize: int
  ctype: str
  fmt: str | None  # struct / numpy format character

  def __repr__(self): return f"dtypes.{self.name}"
  @property
  def is_float(self) -> bool: return self.name in ("f32", "f16")
  @property
  def is_int(self) -> bool: return self.name == "i32"
```

uopc/dtype.py · dtypes

```python
class dtypes:
  void = DType("void", 0, "void", None)
  bool = DType("bool", 1, "_Bool", "?")
  i32 = DType("i32", 4, "int", "i")
  f32 = DType("f32", 4, "float", "f")
  f16 = DType("f16", 2, "_Float16", "e")
  all = (void, bool, i32, f32, f16)

  @staticmethod
  def from_name(name: str) -> DType:
    for d in dtypes.all:
      if d.name == name: return d
    raise ValueError(f"unknown dtype {name!r}")

  @staticmethod
  def from_py(x) -> DType:
    if isinstance(x, bool): return dtypes.bool
    if isinstance(x, int): return dtypes.i32
    if isinstance(x, float): return dtypes.f32
    raise TypeError(f"no dtype for python value {x!r}")

  @staticmethod
  def min(d: DType):
    return {"bool": False, "i32": -2**31}.get(d.name, -math.inf)

  @staticmethod
  def max(d: DType):
    return {"bool": True, "i32": 2**31 - 1}.get(d.name, math.inf)
```

Only four matter for most of the book: `void` (the type of things that are not values, like a store), `bool`, `i32` and `f32`. `f16` arrives in week 8 for tensor cores. That is deliberately fewer than any real framework. There is no `i64`, so the largest tensor we can index has $2^{31}-1$ elements, and no `f64`, so there are no accidental double-precision slow paths. Adding dtypes later is easy *if* every place that depends on them goes through `DType`'s fields, which is why the renderer asks for `dtype.ctype` instead of switching on names.

A Python literal's dtype is fixed by `from_py`: `bool` before `int` (in Python `True` is an `int`!), then `int`, then `float`. This is why `UOp.const(1)`, `UOp.const(1.0)` and `UOp.const(True)` are three different nodes with three different dtypes, which is about to become a real bug.

The third recursive property, `addrspace`, says *where* a value lives:

`uopc/dtype.py · AddrSpace`

```python
class AddrSpace(Enum):
  """Where a UOp lives. ALU = a plain value in a register. The rest are memory."""
  ALU = auto()      # a value (week 1)
  GLOBAL = auto()   # device memory, a function argument (week 2)
  LOCAL = auto()    # call-scope storage: `float acc[4];` (week 2)
  SHARED = auto()   # GPU workgroup shared memory (week 7)

  def __repr__(self): return f"AddrSpace.{self.name}"
```

All week-1 values are `ALU`: plain values that a C compiler would keep in registers. Memory arrives in week 2 (Chapter 4), shared memory in week 7. We carry the field from day one so the type checker has a place to put the rule "you can only LOAD from memory".

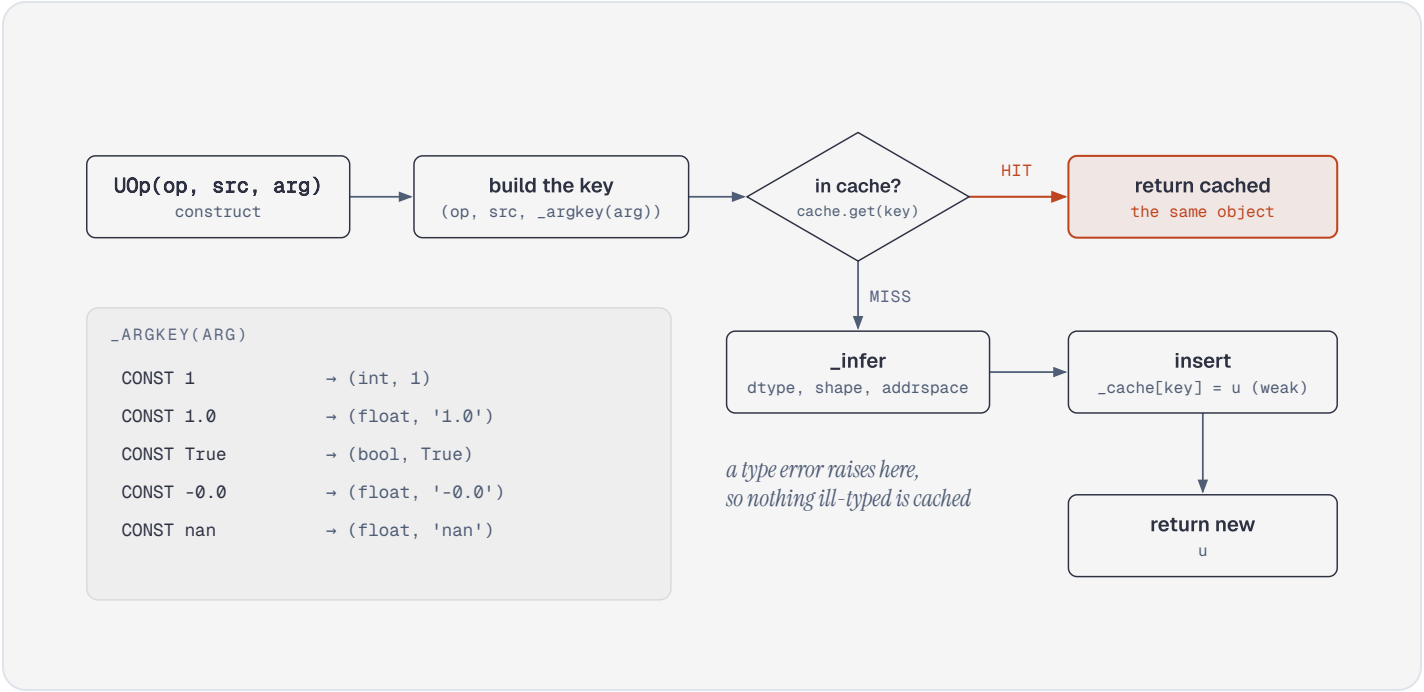
1.4 The UOp class and hash consing

Here is the core of the class:

uopc/uop.py · UOp.__new__

```python
  def __new__(cls, op: Ops, src: tuple[UOp, ...] = (), arg: Any = None):
    key = (op, src, _argkey(arg))
    if (ret := UOp._cache.get(key)) is not None: return ret
    ret = super().__new__(cls)
    ret.op, ret.src, ret.arg = op, src, arg
    ret.dtype, ret.shape, ret.addrspace = _infer(op, src, arg)
    UOp._cache[key] = ret
    return ret
```

Construction is a cache lookup. The key is `(op, src, _argkey(arg))`. If a live UOp with that key exists, you get *that object back*; otherwise a new one is made, its dtype, shape and address space are inferred (and checked), and it is put in the cache. This technique is called **hash consing**, after Lisp's `cons`: it goes back to Ershov's 1958 work on expression evaluation, and functional language implementations have used it to share structure ever since.



The consequence is an invariant that the rest of the compiler leans on everywhere:

> **Key idea:** two UOps are structurally equal if and only if they are the *same Python object*. Equality is `is`, hashing is `id`, and both are O(1) no matter how big the graphs are.

Watch it happen. Two separate constructions of `x + y`:

```
>>> x = UOp(Ops.PARAM, (), ParamArg(0, dtypes.f32))
>>> y = UOp(Ops.PARAM, (), ParamArg(1, dtypes.f32))
>>> a = x + y; b = x + y
>>> a is b, id(a) == id(b)
(True, True)
```

Note what this does *not* require: a recursive comparison. `src` is a tuple of UOps, and since every UOp in it is already unique, comparing two `src` tuples compares a few pointers. Hashing `src` hashes a few pointers. The cache key of a node with a million-node subgraph below it costs the same to build as the key of a leaf. This works because of the order of construction: sources always exist before the node that uses them, so by induction every source is already canonical.

What hash consing buys

Free CSE. Common subexpression elimination is a standard compiler pass: find two computations of the same thing, keep one. With hash consing there is nothing to find; duplicates can't exist. In the DAG above, `s * s` has `s.src[0] is s.src[1]`, and no pass ever had to discover it.

O(1) memoization. Any function of a node can be cached in a `dict` keyed by the node. The rewrite engine (Chapter 2) memoizes its work this way, `simplify` is memoized globally (Chapter 3), and the kernel cache (Chapter 7) keys whole compiled programs on graph structure.

Cheap equality checks in tests and rules. `parse(to_uir(g)) is g` is a complete round-trip test. `simplify(x * 1) is x` checks a rewrite. Rules can say "the same `x` appears twice" by checking identity.

Acyclicity for free. To build a node you must pass its sources, which must already exist. You cannot create a node that is its own ancestor, because it would have to exist before itself. Every UOp graph is a DAG by construction, and nothing ever has to check for cycles.

What it costs

Immutability is mandatory. If you could change a UOp's `src` after construction, its cache key would lie, and every node that had been hash-consed against it would silently change meaning. UOps are treated as immutable: to "modify" a node, build a new one. That is what `replace` does:

uopc/uop.py · UOp.replace

```
def replace(self, **kw) -> UOp:
  return UOp(kw.get("op", self.op), kw.get("src", self.src), kw.get("arg", self.arg))
```

Every construction pays for a lookup. A tuple, a hash and a dict probe per node. That is a few hundred nanoseconds of Python, which is why compile-time performance shows up in later chapters (the GPT graph is large); the payoff (no CSE pass, O(1) everything else) is much larger.

Things that must be different must look different. Two input buffers with the same shape and dtype would be the same node. That is why every `PARAM` and `BUFFER` carries a `slot` number in its arg: it is part of the identity.

Why weak references

The cache is a `weakref.WeakValueDictionary`: it does not keep its values alive. When the last real reference to a UOp disappears, the UOp is garbage collected and its entry vanishes from the cache. Without that, every intermediate node ever created (and a training loop creates millions) would live forever. You can watch the cache shrink:

```
>>> before = len(UOp._cache)
>>> z = UOp.const(12345) + UOp.const(54321)
>>> mid = len(UOp._cache)
>>> del z; gc.collect()
>>> print(before, mid, len(UOp._cache))
12 15 12
```

Three nodes (two constants and the add) were created and three were reclaimed. The constants were kept alive by the ADD's `src` tuple until the ADD itself died. One corollary: you must never key a long-lived dictionary on `id(u)`, because after `u` dies its `id` can be reused by a different UOp. Key on the UOp itself, which holds a strong reference.

Equality and hashing are inherited from `object`

Look at what the class *doesn't* define: `__eq__` and `__hash__`. Python's defaults compare and hash by identity, which, thanks to hash consing, is exactly structural equality. The comment in the source says why this matters:

`uopc/uop.py:119-124`

```
# hash consing means identity *is* structural equality, so we keep object's default
# __eq__/__hash__ (identity, implemented in C). Note __lt__ etc. build UOps instead.
def __reduce__(self): return (UOp, (self.op, self.src, self.arg))
def __repr__(self):
  s = f"UOp({self.op}, arg={self.arg!r}" if self.arg is not None else f"UOp({self.op}"
```

The first version of `uopc` defined `__eq__` and `__hash__` in Python. Both were correct, and compiling the GPT took 163 seconds for 12 steps, because every `dict` lookup in every pass called a Python function. Switching to the C-implemented identity versions (and memoizing `simplify`) took it to 47 seconds. The enum classes do the same: `Ops` sets `__hash__ = object.__hash__`, because Python's `Enum` defines a slower `__hash__` of its own.

`__reduce__` makes pickling (and `copy.deepcopy`) go back through `UOp(...)`, so an unpickled graph is re-hash-consed into the live cache:

```
>>> pickle.loads(pickle.dumps(e)) is e, copy.deepcopy(e) is e
(True, True)
```

1.5 `_argkey`: when Python's equality lies

The cache key cannot be `(op, src, arg)` directly. This was the first real bug in `uopc`. Python says:

```
>>> 1 == 1.0 == True, hash(1), hash(1.0), hash(True)
(True, 1, 1, 1)
>>> 0.0 == -0.0, hash(0.0) == hash(-0.0)
(True, True)
>>> d = {}; d[(Ops.CONST, (), 1)] = "i32 one"
>>> d.get((Ops.CONST, (), True)), d.get((Ops.CONST, (), 1.0))
('i32 one', 'i32 one')
```

A cache keyed on the raw arg merged `CONST 1` (i32), `CONST 1.0` (f32) and `CONST True` (bool) into whichever one was built first. After that, `UOp.const(1.0)` could return a node whose dtype was `i32`, and the type checker would complain about a dtype mismatch in code that was obviously correct, or worse, accept a graph whose constants had the wrong type. Zero has the same problem with sign: `0.0` and `-0.0` are equal and hash equal, but `1/0.0` is `inf` and `1/-0.0` is `-inf`.

The fix makes the type part of the key, and uses `repr` for floats, which distinguishes `-0.0`:

uopc/uop.py · _argkey

```
def _argkey(arg):
  # 1 == 1.0 == True and 0.0 == -0.0 in python; the cache must not merge them.
  if isinstance(arg, float): return (float, repr(arg))
  if isinstance(arg, tuple): return (tuple, tuple(_argkey(a) for a in arg))
  return (type(arg), arg)
```

```
>>> _argkey(1), _argkey(1.0), _argkey(True), _argkey(-0.0), _argkey((1, 2.0))
((<class 'int'>, 1), (<class 'float'>, '1.0'), (<class 'bool'>, True), (<class 'float'>, '-0.0'),
(<class 'tuple'>, ((<class 'int'>, 1), (<class 'float'>, '2.0'))))
>>> UOp.const(1) is UOp.const(1.0), UOp.const(0.0) is UOp.const(-0.0)
(False, False)
```

`repr` also fixes a third, subtler problem: NaN. `nan != nan`, so with raw floats in the key, every `UOp.const(float('nan'))` would miss the cache (dict lookup compares keys with `==` after the identity shortcut) and create a new node. With `repr`, all NaNs share the key `'nan'`:

```
>>> UOp.const(float('nan')) is UOp.const(float('nan'))
True
```

Tuples are recursed into because movement-op args are tuples of numbers. Every other arg type (a `DType`, a `ParamArg`, an `Ops` member) is a frozen dataclass or enum whose equality is already exact.

> **Key idea:** hash consing is only as correct as the equality on its keys. Python's numeric tower deliberately makes `1`, `1.0` and `True` equal; a compiler must not.

1.6 Recursive properties: the type checker

`dtype`, `shape` and `addrspace` are not passed to the constructor. They are *recursive properties*: computed from the op, the arg and the sources' own properties, once, when the node is built. The function that computes them is the type checker of the IR, and every malformed UOp dies inside it:

`uopc/uop.py · _infer`

```
def _infer(op: Ops, src: tuple[UOp, ...], arg) -> tuple[DType, tuple[int, ...], AddrSpace]:
  A = AddrSpace.ALU
  if op is Ops.CONST:
    _check(not src, "CONST has no sources")
    return dtypes.from_py(arg), (), A
  if op is Ops.PARAM: return arg.dtype, arg.shape, A
  if op is Ops.BUFFER:
    _check(len(arg.shape) == 1, "BUFFER is flat")
    return arg.dtype, arg.shape, AddrSpace.GLOBAL
  if op is Ops.ALLOC:
    _check(len(arg.shape) == 1, "ALLOC is flat")
    return arg.dtype, arg.shape, arg.addrspace or AddrSpace.LOCAL
  if op is Ops.CALL:
    params = {u.arg.slot: u for u in call_params(src[0])}
    _check(sorted(params) == list(range(len(src) - 1)), f"CALL needs params 0..{len(src)-2}, body
has {sorted(params)}")
    for i, a in enumerate(src[1:]):
      p = params[i]
      _check(a.shape == p.shape and a.dtype == p.dtype, f"CALL arg {i}: {a.dtype}{a.shape} !=
param {p.dtype}{p.shape}")
    return src[0].dtype, src[0].shape, src[0].addrspace
  if op is Ops.STACK:
    _check(len(src) > 0, "STACK of nothing")
    _check(all(s.shape == src[0].shape and s.dtype == src[0].dtype for s in src), "STACK srcs
must agree in shape and dtype")
    return src[0].dtype, (len(src),) + src[0].shape, A
  if op is Ops.INDEX:
    x, idx = src[0], src[1:]
    _check(len(idx) <= len(x.shape), f"INDEX: {len(idx)} indices into shape {x.shape}")
    _check(all(i.dtype == dtypes.i32 and i.shape == () for i in idx), "INDEX indices are i32
scalars")
    return x.dtype, x.shape[len(idx):], x.addrspace
  if op in ALU:
    if op is Ops.WHERE:
      c, a, b = src
      _check(c.dtype == dtypes.bool, "WHERE condition must be bool")
      _check(a.dtype == b.dtype, f"WHERE branches disagree: {a.dtype} vs {b.dtype}")
      _check(c.shape == a.shape == b.shape, f"WHERE shape mismatch {c.shape} {a.shape}
{b.shape}")
      return a.dtype, a.shape, A
    if op in UNARY:
      _check(len(src) == 1, f"{op} is unary")
      if op is Ops.CAST: return arg, src[0].shape, A
      _check(src[0].dtype.is_float, f"{op} needs a float, got {src[0].dtype}")
      return src[0].dtype, src[0].shape, A
    a, b = src
    _check(a.shape == b.shape, f"{op}: shape mismatch {a.shape} vs {b.shape} (broadcast with
EXPAND)")
    _check(a.dtype == b.dtype, f"{op}: dtype mismatch {a.dtype} vs {b.dtype} (use CAST)")
    if op in (Ops.FLOORDIV, Ops.FLOORMOD): _check(a.dtype == dtypes.i32, f"{op} is integer only")
    return (dtypes.bool if op in (Ops.CMPLT, Ops.CMPNE) else a.dtype), a.shape, A
  if op is Ops.LOAD:
```

```
    _check(src[0].addrspace is not A, "LOAD needs memory")
    return src[0].dtype, src[0].shape, A
  if op is Ops.STORE:
    _check(src[0].addrspace is not A, "STORE needs memory")
    _check(src[0].dtype == src[1].dtype, f"STORE of {src[1].dtype} into {src[0].dtype}")
    return dtypes.void, (), A
  if op is Ops.AFTER: return src[0].dtype, src[0].shape, src[0].addrspace
  if op in (Ops.SINK, Ops.END, Ops.BARRIER): return dtypes.void, (), A
  if op is Ops.RANGE:
    _check(src[0].dtype == dtypes.i32, "RANGE end is i32")
    return dtypes.i32, (), A
  if op is Ops.RESHAPE:
    _check(prod(arg) == prod(src[0].shape), f"RESHAPE {src[0].shape} -> {arg}")
    return src[0].dtype, arg, src[0].addrspace
  if op is Ops.EXPAND:
    s = src[0].shape
    _check(len(arg) == len(s) and all(a == b or a == 1 for a, b in zip(s, arg)), f"EXPAND {s} ->
{arg}")
    return src[0].dtype, arg, src[0].addrspace
  if op is Ops.PERMUTE:
    _check(sorted(arg) == list(range(len(src[0].shape))), f"PERMUTE {arg} of {src[0].shape}")
    return src[0].dtype, tuple(src[0].shape[i] for i in arg), src[0].addrspace
  if op is Ops.SHRINK:
    s = src[0].shape
    _check(len(arg) == len(s) and all(0 <= b <= e <= n for (b, e), n in zip(arg, s)), f"SHRINK
{arg} of {s}")
    return src[0].dtype, tuple(e - b for b, e in arg), src[0].addrspace
  if op is Ops.REDUCE:
    rop, n = arg
    _check(rop in (Ops.ADD, Ops.MAX) and 0 <= n <= len(src[0].shape), f"REDUCE {arg} of
{src[0].shape}")
    return src[0].dtype, src[0].shape[:len(src[0].shape) - n], A
  if op is Ops.STAGE: return src[0].dtype, src[0].shape, A
  if op is Ops.WMMA: return src[2].dtype, src[2].shape, A
  raise UOpError(f"no inference rule for {op}")
```

Read it as a set of typing rules, one per op. In the notation of a type system, the ADD rule is

```
a : (d, s)    b : (d, s)
------------------------
   ADD(a, b) : (d, s)
```

"if both operands have dtype `d` and shape `s`, the sum does too". The code says the same thing, with an error message for each way the premises can fail. CMPLT is identical except that the conclusion is `bool`. FLOORDIV adds the premise `d = i32`. RECIP requires a float. WHERE requires a `bool` condition and agreeing branches. There is no implicit conversion anywhere: `UOp.const(1) + UOp.const(1.0)` is an error, not a silent cast. Implicit promotion is the frontend's business (Chapter 9); the IR stays explicit.

Here is every week-1 error, produced by actually building the bad node:

```
i32 + f32            UOpError: Ops.ADD: dtype mismatch dtypes.i32 vs dtypes.f32 (use CAST)
shape mismatch       UOpError: Ops.ADD: shape mismatch (2,) vs (3,) (broadcast with EXPAND)
ragged stack         UOpError: STACK srcs must agree in shape and dtype
floordiv float       UOpError: Ops.FLOORDIV is integer only
recip int            UOpError: Ops.RECIP needs a float, got dtypes.i32
where cond           UOpError: WHERE condition must be bool
index float          UOpError: INDEX indices are i32 scalars
too many idx         UOpError: INDEX: 2 indices into shape (3,)
call missing param   UOpError: CALL needs params 0..0, body has [0, 1]
call wrong type      UOpError: CALL arg 0: dtypes.f32() != param dtypes.i32()
empty stack          UOpError: STACK of nothing
```

Two messages suggest the fix (`use CAST`, `broadcast with EXPAND`). That costs nothing to write and saves students hours.

Why check at construction time

The alternative is a separate *verifier* pass that walks a finished graph and checks it (MLIR works this way: ops have `verify()` methods that run between passes). Checking at construction is better for us for three reasons:

1. **The stack trace points at the bug.** When a rewrite rule builds an ill-typed node, the exception is raised inside that rule, with the rule's line in the traceback. A verifier would report it after the pass, with no idea which of forty rules produced it.
2. **Ill-typed nodes never exist.** The cache insert is the last line of `__new__`, after `_infer`. A failed construction leaves nothing behind, so no other code can ever observe a malformed UOp.
3. **Properties are computed once.** `u.shape` is a field read, not a recursive computation. Passes ask for shapes constantly.

The cost is that every construction pays for the check. It is a handful of comparisons; it has never shown up in a profile.

Note that the type checker does *not* stop a rewrite rule from replacing a node with a node of a different type. `_infer` checks that each node is well-formed given its sources, not that a replacement means the same thing. Chapter 2 shows a real rule that turned a `bool` into an `i32` and how to catch it.

1.7 Building graphs

Writing `UOp(Ops.ADD, (a, b))` everywhere would be unreadable, so `UOp` overloads Python's operators. This is the sugar, and it is worth reading closely because it fixes how the encoded ops look:

```
uopc/uop.py:166-206
```

```python
  # ---- constructors ----------------------------------------------------
  @staticmethod
  def const(v, dtype: DType | None = None) -> UOp:
    if dtype == dtypes.f16: return UOp(Ops.CONST, (), float(v)).cast(dtypes.f16)   # python has
no f16 values
    if dtype is not None and dtype != dtypes.from_py(v):
      v = {"bool": bool, "i32": int}.get(dtype.name, float)(v)
    return UOp(Ops.CONST, (), v)

  def const_like(self, v) -> UOp:
    """A constant with self's dtype *and shape* (broadcast with RESHAPE + EXPAND)."""
    c = UOp.const(v, self.dtype)
    return c.reshape((1,) * len(self.shape)).expand(self.shape) if self.shape else c
  def _c(self, x) -> UOp: return x if isinstance(x, UOp) else self.const_like(x)
  def alu(self, op: Ops, *src) -> UOp: return UOp(op, (self, *(self._c(s) for s in src)))

  def __add__(self, x): return self.alu(Ops.ADD, x)
  def __radd__(self, x): return self._c(x).alu(Ops.ADD, self)
  def __mul__(self, x): return self.alu(Ops.MUL, x)
  def __rmul__(self, x): return self._c(x).alu(Ops.MUL, self)
  def __neg__(self): return self.alu(Ops.MUL, -1 if self.dtype == dtypes.i32 else -1.0)
  def __sub__(self, x): return self + -self._c(x)
  def __rsub__(self, x): return self._c(x) + -self
  def __floordiv__(self, x): return self.alu(Ops.FLOORDIV, x)
  def __mod__(self, x): return self.alu(Ops.FLOORMOD, x)
  def __lt__(self, x): return self.alu(Ops.CMPLT, x)
  def __gt__(self, x): return self._c(x).alu(Ops.CMPLT, self)
  def ne(self, x): return self.alu(Ops.CMPNE, x)
  def eq(self, x): return self.ne(x).ne(True)
  def maximum(self, x): return self.alu(Ops.MAX, x)
  def recip(self): return UOp(Ops.RECIP, (self,))
  def cast(self, d: DType): return self if d == self.dtype else UOp(Ops.CAST, (self,), d)
  def where(self, a, b):
    a = a if isinstance(a, UOp) else (b.const_like(a) if isinstance(b, UOp) else UOp.const(a))
    b = b if isinstance(b, UOp) else a.const_like(b)
    return UOp(Ops.WHERE, (self, a, b))

  def index(self, *idx) -> UOp:
    return UOp(Ops.INDEX, (self, *(i if isinstance(i, UOp) else UOp.const(int(i)) for i in idx)))
  def load(self) -> UOp: return UOp(Ops.LOAD, (self,))
  def store(self, v) -> UOp: return UOp(Ops.STORE, (self, v))
  def after(self, *deps) -> UOp: return UOp(Ops.AFTER, (self, *deps)) if deps else self
```

Three details matter.

Constants adopt the dtype and shape of the other operand. `x + 1` builds `UOp.const(1, x.dtype)` via `_c` / `const_like`, so `f32 + 1` means `f32 + 1.0`. For a shaped `x`, the constant is also broadcast to `x`'s shape with RESHAPE and EXPAND (week-4 ops, but `const_like` needs them already). The first version

did not broadcast, and `-x` on any shaped UOp failed type checking, because `x * -1.0` multiplied a `(3,)` by a `()`.

There is no subtraction, negation or greater-than. Each is lowered to the canonical form on the spot. Here are the actual graphs, printed with `to_uir` (the format of section 1.10):

```
x - y:
%0 = param : slot=0 dtype=i32
%1 = param : slot=1 dtype=i32
%2 = mul %1, -1
%3 = add %0, %2

x > 3:
%0 = param : slot=0 dtype=i32
%1 = cmplt 3, %0

x.eq(y):
%0 = param : slot=0 dtype=i32
%1 = param : slot=1 dtype=i32
%2 = cmpne %0, %1
%3 = cmpne %2, true

v * 2.0:
%0 = param : slot=0 dtype=f32 shape=(3,)
%1 = reshape 2.0, (1,)
%2 = expand %1, (3,)
%3 = mul %0, %2
```

`<` builds a node; it does not compare. `a < b` returns a `CMPLT` UOp. This is why UOps cannot use Python's default ordering, why `sorted(uops)` is meaningless, and why `__eq__` had to stay identity: if `==` built a CMPNE node, `u in some_list` would always be true (a UOp is truthy). The methods `ne` and `eq` are named methods precisely so that `==` keeps its Python meaning.

1.8 STACK and INDEX: arrays without memory

Week 1 has no memory, no buffers, no loops. How do you write a program over arrays? With two ops that build and take apart *values* that happen to have shape.

`STACK(x0, ..., xn-1)` makes a new leading axis: it stacks n same-shaped values into one value of shape `(n, *x0.shape)`. It is `np.stack(xs, axis=0)`. A 2×3 matrix is a STACK of two STACKs of three scalars:

```
>>> m = stack(stack(1, 2, 3), stack(4, 5, 6))
>>> m.shape, m.dtype
((2, 3), dtypes.i32)
```

`INDEX(x, i0, ..., ik-1)` is the inverse: it pops axes off the *left*. With one index it selects a row; with as many indices as axes it selects a scalar.

```
>>> m.index(UOp.const(1)).shape, m.index(UOp.const(1), UOp.const(2)).shape
((3,), ())
```

The indices are themselves UOps (i32 scalars), so they need not be constants. Indexing by a PARAM is a perfectly good graph whose value is unknown until the PARAM is bound:

```
%0 = stack 1, 2, 3
%1 = stack 4, 5, 6
%2 = stack %0, %1
%3 = param : slot=0 dtype=i32
%4 = index %2, %3, 2
```

Why pop axes from the left and not allow arbitrary axes? Because that is what STACK builds: the leftmost axis is the one STACK added last, so `INDEX(STACK(a, b), i)` is `a` or `b`, a rule Chapter 2 uses to evaluate programs. Arbitrary axis selection is a composition of a PERMUTE (week 4) and a left INDEX.

Elementwise ops work on shaped values exactly as they do on scalars: `ADD` of two `(3,)` values is a `(3,)` value. Shapes must match exactly; there is no implicit broadcasting in the IR (the text format and the Tensor frontend add it as sugar, by inserting RESHAPE and EXPAND).

In week 1, the syllabus restricts STACK's inputs to CONSTs and other STACKs (literals, in effect). Week 4 lifts that restriction, and STACK becomes a general "concatenate along a new axis" op that rangeify has to lower (Chapter 6).

1.9 PARAM and CALL: functions as graphs

A *function* in this IR is just a graph that mentions PARAMs, wrapped in a CALL that supplies their values. `ParamArg` and `CallArg` are frozen dataclasses:

uopc/uop.py · ParamArg

```python
@dataclass(frozen=True)
class ParamArg:
  slot: int
  dtype: DType
  shape: tuple[int, ...] = ()
  addrspace: AddrSpace | None = None   # only used by ALLOC (LOCAL or SHARED)
```

uopc/uop.py · CallArg

```python
@dataclass(frozen=True)
class CallArg:
  name: str = "f"
```

`CALL(body, a0, a1, ...)` means "the value of `body` with PARAM slot `i` replaced by `ai`". The square function applied to 7:

```
%0 = param : slot=0 dtype=i32
%1 = mul %0, %0
%2 = call %1, 7 : name=sq
```

PARAMs are identified by *position* (`slot`), not by name, much like de Bruijn indices in the lambda calculus. That has a consequence worth stating now: since PARAMs are hash consed, **`PARAM slot=0 dtype=i32` is the same object in every function body that uses it**. A nested CALL's body and the enclosing body share the node. This is fine as long as every pass that cares about PARAMs knows which body it is in, and the helper that finds the PARAMs of a body is careful not to look into nested bodies:

uopc/uop.py · call_params

```python
def call_params(body: UOp) -> list[UOp]:
  """PARAMs that belong to this body. A nested CALL's body is its own scope, so we don't look
inside it."""
  out, seen, stack = [], set(), [body]
  while stack:
    u = stack.pop()
    if u in seen: continue
    seen.add(u)
    if u.op is Ops.PARAM: out.append(u)
    stack.extend(u.src[1:] if u.op is Ops.CALL else u.src)
  return out
```

`stack.extend(u.src[1:] if u.op is Ops.CALL else u.src)`: when it reaches a nested CALL, it visits its *arguments* (which belong to the outer scope) but not its *body* (`src[0]`, the inner scope). The type checker uses `call_params` to check a CALL: the body must use exactly slots `0..n-1`, and each argument must match its PARAM's dtype and shape. Chapter 2 tells the story of the bug that made this function necessary.

Why have CALL at all in week 1, when there is nothing yet to call? Partly because the syllabus asks for it (weeks 7–8 use calls as the unit of GPU kernels), and partly because it is the smallest example of *scope*, a concept every later chapter needs: loops in week 3 are scopes, and kernels in week 5 are bodies with parameters.

1.10 Walking the graph

Nearly every pass starts with "visit every node, sources before users". That order is a *topological sort*, and here is the one every pass uses:

uopc/uop.py · UOp.toposort

```
def toposort(self) -> list[UOp]:
  """Every node reachable from self, children before parents. Iterative (graphs get deep)."""
  out, seen, stack = [], set(), [(self, False)]
  while stack:
    u, done = stack.pop()
    if done: out.append(u); continue
    if u in seen: continue
    seen.add(u)
    stack.append((u, True))
    stack.extend((s, False) for s in reversed(u.src) if s not in seen)
  return out
```

It is a depth-first search with an explicit stack instead of recursion. Each node is pushed twice: first as `(u, False)` meaning "expand me", then, after its sources, as `(u, True)` meaning "my sources are done, emit me". The `seen` set ensures each node is expanded once, however many users it has. For our running example the order is:

```
0 PARAM
1 CONST 1.0
2 ADD
3 MUL
```

(for `e = (x + 1.0) * (x + 1.0)`). Sources come out in left-to-right order, so the result is deterministic, which matters: generated C should not change between runs of the same program, or the compile cache (Chapter 4) would miss.

Why not recursion? The obvious recursive DFS is four lines. It also crashes on a chain of 20,000 ADDs, which Python's default recursion limit of 1,000 frames cannot hold. Deep chains are not exotic: unrolled reductions, long training graphs and the test `test_deep_graph_no_recursion_error` all produce them. Raising the recursion limit just moves the crash to the C stack. Every traversal in `uopc` is iterative.

`substitute`: replace nodes everywhere

With a topological order, "replace these nodes, rebuild everything above them" is a single loop:

`uopc/uop.py · UOp.substitute`

```
def substitute(self, mapping: dict[UOp, UOp]) -> UOp:
  """Replace nodes (by identity) everywhere in the graph. Rebuilds only what changed."""
  new: dict[UOp, UOp] = {}
  for u in self.toposort():
    if u in mapping: new[u] = mapping[u]; continue
    src = tuple(new[s] for s in u.src)
    new[u] = u if src == u.src else u.replace(src=src)
  return new[self]
```

Each node is rebuilt from the already-rebuilt versions of its sources, and the `src == u.src` check means unaffected subgraphs are reused, not copied. Because rebuilt nodes go through `UOp.__new__`, the result is automatically hash consed: if a substitution makes two subgraphs identical, they become the same node.

```
>>> to_uir(e.substitute({x: UOp.const(3.0)}))
%0 = add 3.0, 1.0
%1 = mul %0, %0
```

There is a sibling, `UOp.wrap`, that rebuilds bottom-up and wraps selected nodes with a function. It exists because of a real bug in kernel fusion and is explained in Chapter 7.

1.11 The `.uir` text format

Graphs need a text form: to write test programs by hand, to print what a pass did, to exchange programs between students' compilers and the course's oracle. The syllabus specifies one, and its example is the first program of the course:

```
# gemm_256x256.uir
%0 = buffer 65536 : dtype=f32 slot=0
%1 = buffer 65536 : dtype=f32 slot=1
%2 = reshape %0, (256, 256, 1)
%3 = reshape %1, (256, 1, 256)
%4 = mul %2, %3
%5 = reduce %4 : op=add pop=1
```

(Despite its file name, this program does not compute a matrix product. Finding out what it *does* compute is a worked example in Chapter 6.)

Each line defines one node, in SSA style: a name, an op, the sources, and after a colon the attributes that make up the arg. The grammar, from the module docstring:

`uopc/uir.py:11-23`

```
Grammar (one statement per line, `#` starts a comment):

    line     := '%' NAME '=' opname operands? (':' attr*)?
    operands := operand (',' operand)*
    operand  := '%' NAME | literal          literal := int | float | true | false | tuple
    attr     := key '=' value

Integer/float literals in a *source* position become CONSTs (`add %x, 1.0`, `range 10`).
Tuple literals are the arg of movement ops. The last statement is the program.

Two pieces of sugar make hand-written programs readable:
  * binary ops broadcast: size-1 axes are EXPANDed, missing leading axes RESHAPEd in
  * a bare literal adopts the dtype of the other operand (`mul %f32, 2` means 2.0)
```

The parser

`parse` reads line by line, strips comments, and keeps an environment from names to UOps:

`uopc/uir.py · parse`

```python
def parse(text: str) -> UOp:
  env: dict[str, UOp] = {}
  last = None
  for lineno, raw in enumerate(text.splitlines(), 1):
    line = raw.split("#", 1)[0].strip()
    if not line: continue
    try: last = _parse_line(line, env)
    except (ParseError, UOpError, KeyError, ValueError, TypeError) as e:
      raise ParseError(f"line {lineno}: {raw.strip()!r}: {e}") from None
  if last is None: raise ParseError("empty program")
  return last
```

Every error is re-raised with the line number and the offending line, so a typo in a 50-line program is found in seconds. The work of one line happens in `_parse_line`:

uopc/uir.py · _parse_line

```python
def _parse_line(line: str, env: dict[str, UOp]) -> UOp:
  lhs, rhs = (p.strip() for p in line.split("=", 1))
  if not lhs.startswith("%"): raise ParseError("statement must start with %name =")
  body, _, attrs_s = rhs.partition(":")
  opname, _, ops_s = body.strip().partition(" ")
  op = Ops[opname.upper()]
  attrs = {}
  for k, v in re.findall(r"(\w+)=(\([^=]*\)|\S+)", attrs_s):
    attrs[k] = _attr_value(k, v)
  toks = _split_top(ops_s)
  lit = None
  if op in POSITIONAL_ARG:
    if op in (Ops.BUFFER, Ops.ALLOC, Ops.CONST): lit, toks = _literal(toks[0]), toks[1:]
    else: lit, toks = _literal(toks[-1]), toks[:-1]
  src = []
  for t in toks:
    if t.startswith("%"):
      if t[1:] not in env: raise ParseError(f"undefined {t}")
      src.append(env[t[1:]])
    else: src.append(_literal(t))
  env[lhs[1:]] = u = _build(op, src, lit, attrs)
  return u
```

Split `%name = op operands : attrs` into its pieces; parse `key=value` attributes with a regular expression; for ops whose arg is positional (the size of a buffer, the value of a constant, the shape of a reshape) peel that literal off the operand list; look up every `%name` operand in the environment. Then `_build` turns op, sources, literal and attributes into a UOp:

uopc/uir.py · _build

```python
def _build(op: Ops, src: list, lit, attrs: dict) -> UOp:
  # bare literals in source position become CONSTs, typed like their neighbours
  # (a WHERE's condition is bool: its branches decide the literal's type)
  like = next((s for s in (src[1:] if op is Ops.WHERE else src) if isinstance(s, UOp)), None)
  def const(v):
    if op in (Ops.INDEX, Ops.RANGE) or like is None or isinstance(v, bool): return UOp.const(v)
    return UOp.const(v, like.dtype)
  src = [s if isinstance(s, UOp) else const(s) for s in src]
  match op:
    case Ops.BUFFER | Ops.ALLOC:
      return UOp(op, (), ParamArg(attrs.get("slot", 0), attrs.get("dtype", dtypes.f32), (lit,),
attrs.get("space")))
    case Ops.PARAM: return UOp(op, (), ParamArg(attrs["slot"], attrs.get("dtype", dtypes.f32),
tuple(attrs.get("shape", ()))))
    case Ops.CONST: return UOp.const(lit, attrs.get("dtype"))
    case Ops.CALL: return UOp(op, tuple(src), CallArg(attrs.get("name", "f")))
    case Ops.CAST: return src[0].cast(attrs["dtype"])
    case Ops.RANGE: return UOp(op, tuple(src), RangeArg(attrs.get("idx", 0), attrs.get("kind",
AxisType.LOOP)))
    case Ops.REDUCE: return src[0].reduce(attrs.get("op", Ops.ADD), attrs.get("pop", 1))
    case Ops.RESHAPE | Ops.EXPAND | Ops.PERMUTE: return UOp(op, (src[0],), tuple(lit))
    case Ops.SHRINK: return src[0].shrink(lit)
  if op in BINARY:
    a, b = _broadcast(*src)
    return UOp(op, (a, b))
  if op is Ops.WHERE:
    c, a, b = src
    a, b = _broadcast(a, b); c, a = _broadcast(c, a); c, b = _broadcast(c, b)
    return UOp(op, (c, a, b))
  return UOp(op, tuple(src))
```

The `match` dispatches on the few ops whose arg has structure; everything else is `UOp(op, tuple(src))`, with the type checker doing the validation. The parser never checks types itself. It does not need to: if `add` gets an i32 and an f32, `UOp.__new__` raises the same error it raises everywhere else.

Sugar: broadcasting and literal typing

Two conveniences make hand-written programs bearable. First, a bare literal operand adopts the dtype of its neighbor: in `mul %f, 2`, the `2` becomes `2.0`. Second, binary ops broadcast numpy-style: missing leading axes are added with RESHAPE and size-1 axes are EXPANDed.

`uopc/uir.py · _broadcast`

```
def _broadcast(a: UOp, b: UOp) -> tuple[UOp, UOp]:
  if a.shape == b.shape: return a, b
  n = max(len(a.shape), len(b.shape))
  a = a.reshape((1,) * (n - len(a.shape)) + a.shape) if len(a.shape) < n else a
  b = b.reshape((1,) * (n - len(b.shape)) + b.shape) if len(b.shape) < n else b
  out = []
  for x, y in zip(a.shape, b.shape):
    if x != y and 1 not in (x, y): raise ParseError(f"can't broadcast {a.shape} with {b.shape}")
    out.append(max(x, y))
  return (a if a.shape == tuple(out) else a.expand(out)), (b if b.shape == tuple(out) else
b.expand(out))
```

Both are expansions into ordinary UOps; the IR itself still has neither. Here is a program that uses both, and what it means:

```
g = parse("""
%row = stack 1.0, 2.0, 3.0
%m   = stack %row, %row      # shape (2, 3)
%b   = stack 10.0, 20.0, 30.0
%s   = add %m, %b            # (2,3) + (3,): broadcast
%t   = mul %s, 2             # bare 2 adopts f32
""")
```

```
(2, 3) dtypes.f32
%0 = stack 1.0, 2.0, 3.0
%1 = stack %0, %0
%2 = stack 10.0, 20.0, 30.0
%3 = reshape %2, (1, 3)
%4 = expand %3, (2, 3)
%5 = add %1, %4
%6 = reshape 2.0, (1, 1)
%7 = expand %6, (2, 3)
%8 = mul %5, %7
```

Names in the input are arbitrary (`%row`, `%m`); the printer renumbers them.

The printer and round-tripping

`uopc/uir.py · to_uir`

```
def to_uir(root: UOp) -> str:
  names: dict[UOp, str] = {}
  lines = []
  for u in root.toposort():
    if u.op is Ops.CONST and u is not root: continue   # constants are printed inline
    names[u] = f"%{len(names)}"
    ops = [_fmt_lit(s.arg) if s.op is Ops.CONST else names[s] for s in u.src]
    attrs = []
    a = u.arg
    match u.op:
      case Ops.BUFFER | Ops.ALLOC:
        ops = [str(a.shape[0])]
        attrs = [f"dtype={a.dtype.name}", f"slot={a.slot}"] + ([f"space=
{a.addrspace.name.lower()}"] if a.addrspace else [])
      case Ops.PARAM: attrs = [f"slot={a.slot}", f"dtype={a.dtype.name}"] + ([f"shape=
{_fmt_lit(a.shape)}"] if a.shape else [])
      case Ops.CONST: ops = [_fmt_lit(a)]
      case Ops.CALL: attrs = [f"name={a.name}"]
      case Ops.CAST: attrs = [f"dtype={a.name}"]
      case Ops.RANGE: attrs = [f"idx={a.idx}", f"kind={a.kind.name.lower()}"]
      case Ops.REDUCE: attrs = [f"op={a[0].name.lower()}", f"pop={a[1]}"]
      case Ops.RESHAPE | Ops.EXPAND | Ops.PERMUTE | Ops.SHRINK: ops.append(_fmt_lit(a))
    line = f"{names[u]} = {u.op.name.lower()}"
    if ops: line += " " + ", ".join(ops)
    if attrs: line += " : " + " ".join(attrs)
    lines.append(line)
  return "\n".join(lines) + "\n"
```

The printer walks `toposort()` and gives each node the next number. Constants are printed *inline* as literals at their use sites rather than as their own lines, which keeps programs readable (`add %3, 1.0` rather than a separate `%4 = const 1.0`), unless the whole program is a single constant.

The property that ties parser and printer together is the **round trip**: printing a graph and parsing the result gives back the same graph. With hash consing, "the same graph" means the same object, so the test is one line:

```
>>> parse(to_uir(g)) is g
True
```

That `is` checks every node, every arg and every edge of the graph at once. The week-1 test `test_uir_roundtrip_and_syllabus_example` asserts it for the syllabus program. Inline constants make the round trip slightly delicate: a printed literal must parse back to *exactly* the same CONST, including dtype (`2` vs `2.0`), sign of zero (`-0.0`), infinities (`inf`) and bools (`true`). Exercise 6 asks you to find a case where it fails.

Parse errors look like this:

```
ParseError: line 1: '%0 = add %1, 2': undefined %1
ParseError: line 3: '%2 = add %0, %1': can't broadcast (2,) with (3,)
ParseError: line 1: 'x = const 1': statement must start with %name =
```

1.12 What can this language compute?

The syllabus opens week 1 with a lecture on Turing completeness, and it is worth asking what we have built. A week-1 program is a finite DAG of CONSTs, STACKs, INDEXes, ALU ops, PARAMs and CALLs. Then:

- **Every program terminates.** Evaluating a DAG visits each node once. There is no loop, and no recursion: a CALL's body is a graph that already exists when the CALL is built, so a function cannot mention itself (that would be a cycle, and hash-consed graphs cannot have cycles).
- **The amount of work is bounded by the size of the program.** A program with N nodes does at most N operations (plus the work hidden in STACK sizes, which are also written in the program).
- **So it is not Turing complete.** It cannot compute any function whose running time grows faster than its program size: no "loop until convergence", no general recursion. In circuit-complexity terms, a week-1 program is an arithmetic circuit.

Week 3 adds `RANGE`, a loop with a trip count fixed when the loop starts. Bounded loops make the language much more expressive (a 3-line program can now do a billion multiply-adds), but every program still terminates. This is exactly the classic *LOOP language* of Meyer and Ritchie (1967), whose programs compute precisely the primitive recursive functions: everything you would ordinarily call "computable" except what needs unbounded search. The language never gets a `while`.

This is a feature, not a limitation. The things we want to compile (a forward pass, a backward pass, an optimizer step) are fixed-size computations; their trip counts are tensor shapes, known before the computation runs. A language in which every program terminates and every loop bound is known is a language a compiler can analyze completely: it can count FLOPs, compute memory traffic, prove index bounds (Chapter 3) and pick tile sizes. The Turing-complete part (the training loop, data loading, early stopping) stays in Python, the host language, which calls the compiler once per fixed-shape step. Every ML compiler you will meet makes this same split, with more or less dynamism at the boundary (Chapter 15 discusses symbolic shapes, the main exception).

1.13 Putting it together: implementing week 1's IR

If you are building your own compiler, here is the order that works, with the test from `tests/test_w01_uops.py` that checks each step:

1. `DType`, `dtypes`, `AddrSpace`. Then `Ops`, and a `UOp` class with `__new__` that hash conses with an `_argkey`ed key. Test: `test_hash_consing`.
2. `_infer` for CONST, STACK, INDEX, ALU, PARAM, CALL; operator sugar. Test: `test_shape_dtype_inference`, `test_call_type_checks`.
3. Iterative `toposort` and `substitute`.
4. The `.uir` parser and printer. Tests: `test_uir_roundtrip_and_syllabus_example` (this one needs BUFFER, RESHAPE, MUL and REDUCE to *construct*, though nothing evaluates them yet).

Here are the first two tests, which together pin down most of this chapter:

`tests/test_w01_uops.py · test_hash_consing`

```
def test_hash_consing():
  a, b = UOp.const(1) + UOp.const(2), UOp.const(1) + UOp.const(2)
  assert a is b
  assert UOp.const(1) is not UOp.const(1.0) is not UOp.const(True)
  assert UOp.const(0.0) is not UOp.const(-0.0)
```

`tests/test_w01_uops.py · test_shape_dtype_inference`

```
def test_shape_dtype_inference():
  s = stack(stack(1, 2, 3), stack(4, 5, 6))
  assert s.shape == (2, 3) and s.dtype == dtypes.i32
  assert s.index(UOp.const(1)).shape == (3,)
  assert (UOp.const(1) < UOp.const(2)).dtype == dtypes.bool
  with pytest.raises(UOpError): UOp.const(1) + UOp.const(1.0)
  with pytest.raises(UOpError): stack(1, 2) + stack(1, 2, 3)
  with pytest.raises(UOpError): stack(1, 2.0)
```

The remaining week-1 tests (evaluation, rewriting, symbolic) need Chapter 2 and Chapter 3.

Pitfalls

1. Hash-consing collisions from Python equality. (Section 1.5.) A cache keyed on raw args merged `CONST 1`, `CONST 1.0` and `CONST True`, and `0.0` with `-0.0`. The symptom was baffling dtype mismatch errors (`dtypes.i32 vs dtypes.f32`) on lines that built only float constants, and they depended on *which constant had been built first in the process*, so tests passed alone and failed in a suite. Fix: `_argkey` puts the Python type in the key and uses `repr` for floats. Lesson: any time you key a cache on user values, ask what `==` means for them.

2. Recursion depth. A chain of 20,000 ADDs blew Python's recursion limit in a recursive traversal. The symptom is a `RecursionError` deep inside a pass, only on big graphs, i.e. only in week 9. Fix: every traversal (`toposort`, `call_params`, `graph_rewrite`, `inline_call`) uses an explicit stack. The test `test_deep_graph_no_recursion_error` keeps it that way.

3. `const_like` must broadcast. The first `const_like` returned a scalar. `x * 2` worked for scalar `x`, and every shaped program failed at `-x` with `Ops.MUL: shape mismatch (3,) vs ()`. Constants that stand in for "the same value everywhere" must take the shape of the thing they combine with.

4. Printing a DAG as a tree. `repr(u)` prints the graph as a *tree*: shared subgraphs are printed once per use. For `x` squared ten times (`t = t * t`, 11 nodes), the repr is 105,449 characters, and for twenty squarings it would be about 100 million. If your debug print hangs, this is why. Use `to_uir`, which prints each node once:

```
%0 = param : slot=0 dtype=f32
%1 = mul %0, %0
%2 = mul %1, %1
...
%10 = mul %9, %9
```

The same exponential blow-up applies to *any* recursive function on UOps without a memo table, which is the subject of Chapter 2's first experiment.

5. Mutating a UOp. `UOp` has a `__dict__` (passes cache things like bounds and buffers on nodes), so Python will let you write `u.src = (...)` or `u.arg = 3`. Doing so corrupts the cache: the node is still filed under its old key, and every other node that was hash consed to it changes meaning. Never assign to `op`, `src`, `arg` or the inferred properties after construction; build a new node with `replace`.

6. `==` on UOps. Because `<` builds a node, it is tempting to expect `==` to as well, or to think `u == v` compares structure in some special way. It is plain identity, which *is* structural equality here. But `u < v` in an `if` is always true (a UOp object is truthy), so `if a < b:` on UOps is a bug that never raises.

Exercises

1. ★ **Predict the output.** For `x = PARAM(slot=0, f32)`, how many nodes does `(x * x) + (x * x) * (x * x)` contain? What does `to_uir` print? Check your answer.

2. ★ **Subtraction.** Implement `UOp.__sub__` without looking at the source, so that `x - 2` on an i32 PARAM prints as `mul` / `add` lines. Why does `-1` need to be `-1` for i32 and `-1.0` for f32?

3. ★ **Find the bug.** A student's cache uses `key = (op, src, arg)` and adds `and type(arg)` to the key as `(op, src, arg, type(arg))`. Which of the three collisions from section 1.5 does this fix, and which does it not?

4. ★★ **A recursive toposort.** Write the four-line recursive version of `toposort`, check that it produces the same order as the iterative one on random graphs, and find the smallest chain length that crashes it on your machine. Then explain why `sys.setrecursionlimit(10**6)` is not a fix.

5. ★★ **Extend the type checker.** Add a `SUB` op properly: a new enum member, an `_infer` rule, and make `x - y` build it. List every other file in `uopc/` you would have to change for SUB to be a first-class op (renderer, symbolic rules, autodiff, ...). This is the cost section 1.2 was arguing about.

6. ★★ **Find the bug: a round trip that failed.** An earlier version of `_build` typed bare literals like the *first* UOp operand. Build `c = PARAM(0, i32) < PARAM(1, i32)` and `w = c.where(1, 0)` (an i32 WHERE with constant branches). With that version, what dtype would the literal `1` in `where %c, 1, 0` get, why does `parse(to_uir(w)) is w` fail, and why does `where %c, %x, 0` raise a UOpError? Read the current `_build` to see the one-line fix, then write a round-trip property test (random WHEREs with literal and non-literal branches) that would have caught the original.

7. ★★ **Property-based round-trip test.** Write a generator of random well-typed week-1 graphs (random CONSTs of each dtype including `-0.0`, `inf`, `nan`; STACKs; ALU ops; INDEX with constant indices), and assert `parse(to_uir(g)) is g` for 10,000 of them. Count how many distinct bugs it finds in your implementation.

8. ★★ **Memory.** Build a 1,000,000-node chain, record `len(UOp._cache)`, drop the reference, call `gc.collect()`, and check the cache again. Now replace the `WeakValueDictionary` with a plain `dict` in your copy and repeat. Why might a long training run with a plain dict look fine for an hour and then die?

9. ★★★ **Hash consing in C.** Implement the same node type in C: a struct with an op, a source array and an arg union, a global hash table keyed on (op, source pointers, arg bits), and a `mk_node` function. How do you handle `-0.0` and NaN in the arg? How would you free nodes (reference counting is the usual answer; why is it safe here, when it is not safe for general graphs)?

10. ★★★ **What can't it compute?** Prove that every week-1 program without CALL evaluates in at most `N` ALU operations, where `N` is its number of nodes (count STACK elements carefully). Then write a week-1 program of size O(n) that computes `2^(2^n)` mod 2^{32} (hint: squaring). What does this say about "bounded by program size" when constants wrap around?

Solutions to ★ exercises

1. `x * x` is one node, used three times; `(x*x) * (x*x)` is a MUL with both sources the same node; the root is an ADD. With the PARAM that is 4 nodes:

```
%0 = param : slot=0 dtype=f32 %1 = mul %0, %0 %2 = mul %1, %1 %3 = add %1, %2
```

1. `def __sub__(self, x): return self + -self._c(x)` with `__neg__` multiplying by `-1` or `-1.0`. The constant must have the same dtype as `self`, or the MUL fails the type check (`Ops.MUL: dtype mismatch`); `const_like` would also convert it for you.

2. Adding `type(arg)` distinguishes `1`, `1.0` and `True` (different types), but `0.0` and `-0.0` are both `float`, compare equal and hash equal, so they still collide. NaN constants still never hit the cache. And a tuple arg like `(1, 2)` vs `(1.0, 2)` (both `tuple`) still collides; that is why `_argkey` recurses into tuples.

Quiz

1. Name two things hash consing gives a compiler for free, and one thing it forces on it.
2. Why is `UOp(Ops.CONST, (), 1) is UOp(Ops.CONST, (), 1.0)` false, and what would go wrong if it were true?
3. What is the shape of `INDEX(STACK(STACK(1,2,3), STACK(4,5,6)), PARAM(slot=0, i32))`?
4. Why can a UOp graph never contain a cycle? Which line of code guarantees it?
5. Write `x >= y` for i32 UOps using only week-1 ops.

6. Why is `toposort` iterative?
7. In a CALL nested inside another CALL's body, which PARAMs does `call_params` of the outer body return, and why does it skip `src[0]` of the inner CALL but not `src[1:]`?
8. Is the week-1 language Turing complete? Is the week-3 language (with bounded RANGE loops)?

Answers

1. Free: O(1) structural equality (`is`), common subexpression elimination, O(1) memoization keyed by node, acyclicity. Forced: immutability (and anything that must be distinct must differ in its arg, like PARAM slots).
2. `_argkey` includes the Python type, so the keys differ. If they were merged, one node would have one dtype, and asking for an f32 constant could return an i32 node (or vice versa), producing spurious type errors or wrongly typed programs depending on construction order.
3. `(3,)`: one index pops the leading axis of `(2, 3)`.
4. A node's sources must exist before the node is constructed and can never be changed afterward, so a node cannot be its own (transitive) source. The guarantee is that `src` is fixed in `UOp.__new__` (`ret.op, ret.src, ret.arg = op, src, arg`) and never reassigned.
5. `CMPNE(CMPLT(x, y), True)`, i.e. "not (x < y)".
6. Real graphs have dependency chains tens of thousands of nodes deep; a recursive DFS exceeds Python's recursion limit (and raising it risks overflowing the C stack).
7. Only the PARAMs belonging to the outer body: those reachable without entering a nested CALL's body. The inner CALL's arguments (`src[1:]`) are expressions in the *outer* scope, so their PARAMs are the outer function's; its body (`src[0]`) has its own scope.
8. No. Every week-1 program terminates in time bounded by its size. Week 3 adds bounded loops: still total, still not Turing complete (it is the LOOP language, computing primitive recursive functions).

Further reading

- Jean-Christophe Filliâtre and Sylvain Conchon, "Type-Safe Modular Hash-Consing", ML Workshop 2006. A careful treatment of hash consing with weak tables, the same design as `UOp._cache`.
- Andrei P. Ershov, "On programming of arithmetic operations", *Communications of the ACM* 1(8), 1958. The origin of hashing expressions to share common subexpressions.
- Cliff Click and Michael Paleczny, "A Simple Graph-Based Intermediate Representation", IR Workshop
- The "sea of nodes" IR used in HotSpot's C2 compiler: a dataflow graph with hash-consed nodes ("global value numbering" by construction), and explicit control edges where needed.

- Chris Lattner et al., "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation", CGO
- The opposite design point: an extensible IR with many op kinds, regions and verifiers.
- Albert R. Meyer and Dennis M. Ritchie, "The complexity of loop programs", Proc. ACM National Meeting 1967. LOOP programs and the primitive recursive functions.
- tinygrad, `tinygrad/uop/ops.py` (https://github.com/tinygrad/tinygrad): the production ancestor of `UOp`, with many more ops and the same `(op, dtype, src, arg)` core.

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

```
uopc/uop.py · 320 lines

uopc/dtype.py · 72 lines

uopc/uir.py · 178 lines

tests/test_w01_uops.py · 128 lines
```

2. Term Rewriting and the Rewrite Engine

Almost every pass in this compiler has the same shape: *find a piece of the graph that looks like this; replace it with that.* "An ADD of something and zero is that something." "A multiply of two constants is their product." "Indexing into a STACK at a known position is that element." "A CALL is its body with the arguments plugged in." Written one at a time, each of those is a tiny function. Written as a *system*, a set of rules plus an engine that applies them until nothing changes, they are a constant folder, an algebraic simplifier, an interpreter, and later the lowering from shapes to loops. This chapter builds that engine: a pattern language, a matcher, and a bottom-up rewriter that reaches a fixed point without recursion and without ever doing the same work twice. Then it uses the engine to *run* week-1 programs, with no interpreter at all.

What you'll build

- `UPat`: patterns that match UOps by op, dtype, arg and sources, bind names, and handle commutative ops by trying both operand orders.
- `PatternMatcher`: an ordered list of `(pattern, function)` rules, indexed by op.
- `graph_rewrite`: a memoized, iterative, bottom-up rewriter that runs to a fixed point.
- `evaluate`: rules that push ALU ops through STACK, index into STACKs, compute movement ops on constants, and inline CALLs, so that `run(program)` reduces any closed week-1 program to a value.

Where this fits. Chapter 1 gave us the graphs. This chapter gives us the tool for transforming them. Chapter 3 writes the forty-odd algebra rules that the engine runs, and every chapter after that (rangeify, layout, GPU lowering) is mostly new rule sets for the same engine.

2.1 Term rewriting in one page

A **term** is a tree built from function symbols and variables: `add(mul(x, 3), 0)`. A **rewrite rule** is a pair `l → r` of terms where every variable in `r` also appears in `l`:

```
add(x, 0) → x
mul(x, 1) → x
mul(x, 0) → 0
```

A rule *applies* to a term `t` if some subterm of `t` is an instance of `l` (you can substitute terms for `l`'s variables to make it equal); that subterm is called a **redex**. Applying the rule replaces the redex with the correspondingly instantiated `r`. A **term rewriting system** (TRS) is a set of such rules. A term with no redex is in **normal form**.

```
add(mul(add(y, 0), 1), 0)
  → add(mul(y, 1), 0)        by add(x, 0) → x   on the inner add
  → add(y, 0)                by mul(x, 1) → x
  → y                        by add(x, 0) → x
```

Two questions decide whether a TRS is a well-behaved program:

Termination. Does every sequence of rewrites stop? The three rules above obviously terminate: each one makes the term strictly smaller. In general one proves termination by finding a *measure* (term size, or a fancier ordering) that every rule strictly decreases. A single bad rule breaks it: `add(x, y) → add(y, x)` (commutativity) can be applied forever.

Confluence. If a term can be rewritten in two different ways, do the two paths always end at the same normal form? If yes, the *order* in which rules are applied doesn't matter for the final answer, and the TRS defines a function. If no, the answer depends on the strategy. Newman's lemma gives a practical test: a terminating system is confluent iff it is *locally* confluent, which can be checked by looking at every pair of rules whose left sides overlap (the *critical pairs*) and checking that both results can be rewritten to a common term.

For instance, take `mul(x, 0) → 0` and the distribution rule `mul(add(a, b), c) → add(mul(a, c), mul(b, c))`. They overlap on `mul(add(a, b), 0)`. One path gives `0` immediately. The other gives `add(mul(a, 0), mul(b, 0))`, then `add(0, 0)`, then (with a constant folding rule) `0`. The critical pair is joinable: no conflict.

Now a real conflict from `uopc` itself. The constant folder evaluates `add(-0.0, 0.0)` in IEEE arithmetic, which gives `0.0`. The identity rule `add(x, 0.0) → x` gives `-0.0`. Both rules are in the real `symbolic` rule set, and the result depends on which one is tried first:

```
>>> e = UOp.const(-0.0) + UOp.const(0.0)
>>> graph_rewrite(e, symbolic).arg                    # folding is listed first
0.0
>>> rev = PatternMatcher(symbolic.rules[1:] + symbolic.rules[:1])
>>> graph_rewrite(e, rev).arg                         # identity first
-0.0
```

So `symbolic` is not confluent, and we know it: the comment on that rule in `uopc/symbolic.py` says `x+0.0 is not exact for x=-0.0; we accept that`. The practical stance of every production rewriter (LLVM's InstCombine, tinygrad's symbolic, GCC's match.pd) is the same: rule *order* is part of the specification, the engine applies rules in a fixed, deterministic order, and non-confluence is tolerated where the results are equally correct. What is not tolerated is *unsoundness* (a rule that changes the value) or *non-termination* (which the engine detects; see section 2.5).

> **Key idea:** a rewrite system is a program whose instructions are equations. Soundness is each rule's job. Termination and determinism are the engine's job. Confluence is nice to have.

Rewriting graphs, not trees

Our terms are DAGs: `s * s` contains one `s`, not two. Rewriting on shared graphs (*term graph rewriting*) has one big advantage: rewrite a shared subterm once and every user sees the result. On trees, a subterm used `k` times is rewritten `k` times, and nested sharing makes that exponential. Hash consing gives us term graph rewriting for free: if the engine memoizes "node → rewritten node", each distinct node is processed once. Section 2.4 measures the difference.

Strategy

A *strategy* says which redex to rewrite next. The two classic choices are **innermost** (rewrite arguments before the term containing them; this is call-by-value) and **outermost** (try the top first; call-by-need, roughly). `uopc` uses innermost, bottom-up: every source is fully rewritten before its user is looked at. Rules can then assume their operands are already in normal form, which makes them much simpler to write: a rule for `(x + c1) + c2` never needs to worry that `x + c1` might itself still be reducible.

2.2 Patterns: `UPat`

A rule's left-hand side is a *pattern*: a tree that describes a family of UOps. Here is the pattern class:

`uopc/rewrite.py · UPat.__init__`

```
  def __init__(self, op: Ops | tuple[Ops, ...] | set[Ops] | None = None, src=None, name: str | None = None,
               arg: Any = None, dtype: DType | tuple[DType, ...] | None = None, allow_any_len: bool = False):
    self.op = None if op is None else (frozenset([op]) if isinstance(op, Ops) else frozenset(op))
    self.dtype = None if dtype is None else (frozenset([dtype]) if isinstance(dtype, DType) else frozenset(dtype))
    # src: tuple -> match in order. list -> match in any order (for commutative ops).
    self.src, self.name, self.arg, self.allow_any_len = src, name, arg, allow_any_len
```

A `UPat` constrains a UOp on up to five things, and every constraint is optional:

Field	Matches if...	Example
`op`	the UOp's op is in this set	`UPat({Ops.ADD, Ops.MUL})`
`dtype`	the UOp's dtype is in this set	`UPat(dtype=dtypes.i32)`
`arg`	`u.arg == arg`	`UPat(Ops.CONST, arg=0)`
`src`	the sources match these sub-patterns	`UPat(Ops.ADD, src=(p, q))`
`name`	always; binds the UOp to this name	`UPat(name="x")`

A pattern with no constraints at all matches any UOp; with a name it is a *variable*. Two helpers make the common cases short: `UPat.var("x")` (any UOp, bound to `x`) and `UPat.cvar("c")` (any CONST, bound to `c`).

uopc/rewrite.py · UPat.var

```
  @staticmethod
  def var(name: str | None = None, dtype=None) -> UPat: return UPat(name=name, dtype=dtype)
```

uopc/rewrite.py · UPat.cvar

```
  @staticmethod
  def cvar(name: str | None = None, dtype=None) -> UPat: return UPat(Ops.CONST, name=name, dtype=dtype)
```

Matching

uopc/rewrite.py · UPat.match

```
  def match(self, u: UOp, store: dict[str, UOp]) -> list[dict[str, UOp]]:
    """All ways this pattern matches u, each a dict of bindings."""
    if self.op is not None and u.op not in self.op: return []
    if self.dtype is not None and u.dtype not in self.dtype: return []
    if self.arg is not None and u.arg != self.arg: return []
    if self.name is not None:
      if self.name in store and store[self.name] is not u: return []   # same name must bind same
node
      store = {**store, self.name: u}
    if self.src is None: return [store]
    orders = itertools.permutations(self.src) if isinstance(self.src, list) else [self.src]
    out = []
    for pats in orders:
      if len(pats) != len(u.src) and not (self.allow_any_len and len(pats) <= len(u.src)):
continue
      stores = [store]
      for p, s in zip(pats, u.src):
        stores = [m for st in stores for m in p.match(s, st)]
        if not stores: break
      out.extend(stores)
    return out
```

`match` returns a *list* of bindings: every way the pattern matches. Each binding is a dict from names to UOps. An empty list means no match. Three features need explaining.

Consistent names. If a name is already bound, it must be bound to the *same node* (`is`, which by hash consing means structurally equal). So `UPat(Ops.ADD, src=(UPat.var("a"), UPat.var("a")))` matches `x + x` and not `x + y`:

```
>>> same = UPat(Ops.ADD, src=(UPat.var("a"), UPat.var("a")))
>>> same.match(x + y, {}), same.match(x + x, {})
([], [{'a': UOp(Ops.PARAM, arg=ParamArg(slot=0, dtype=dtypes.i32, shape=(), addrspace=None))}])
```

In a tree-based rewriter this check would be a recursive structural comparison. Here it is one pointer comparison, no matter how large the subterm.

Tuple vs list sources. A `tuple` of source patterns matches in order. A `list` matches in *any* order: `match` tries every permutation (`itertools.permutations`). That is how commutative ops are handled. A rule for `x + c` should match both `x + 3` and `3 + x`, and rather than writing it twice, we write the sources as a list:

```
>>> pat = UPat(Ops.ADD, src=[UPat.var("a"), UPat.var("b")])
>>> pat.match(x + y, {})
[{'a': UOp(Ops.PARAM, arg=ParamArg(slot=0, ...)), 'b': UOp(Ops.PARAM, arg=ParamArg(slot=1,
...))},
 {'b': UOp(Ops.PARAM, arg=ParamArg(slot=0, ...)), 'a': UOp(Ops.PARAM, arg=ParamArg(slot=1,
...))}]
```

(output abbreviated) Two bindings, one per order. With a tuple, only the first. Permutations cost `n!` for `n` sources, which is fine because every commutative op is binary.

`allow_any_len`. Normally the number of sources must equal the number of sub-patterns. With `allow_any_len=True` the pattern only constrains a *prefix* of the sources. INDEX has a variable number of indices, and the evaluator uses this to match "INDEX of a STACK with at least one index".

Matching is a *join* over sources: start with the bindings so far, match the first source in each of them, feed every resulting binding to the second source, and so on (`stores = [m for st in stores for m in p.match(s, st)]`). The early `break` stops as soon as no binding survives.

Pattern sugar

Rules are easier to read when the pattern looks like the math. `UPat` overloads arithmetic to build patterns:

`uopc/rewrite.py:128-137`

A non-UPat operand becomes a CONST pattern with that arg, and commutative ops get *list* sources automatically. So `UPat.var("x") + UPat.cvar("c")` is an ADD pattern that matches a constant on either side:

```
>>> c = UPat.var("x") + UPat.cvar("c")
>>> c.src, type(c.src)
([UPat(None, name=x), UPat({Ops.CONST}, name=c)], <class 'list'>)
>>> c.match(x + 3, {}), c.match(UOp.const(3) + x, {})
([{'x': UOp(Ops.PARAM, ...), 'c': UOp(Ops.CONST, arg=3)}], [{'c': UOp(Ops.CONST, arg=3), 'x':
UOp(Ops.PARAM, ...)}])
```

(PARAM reprs abbreviated.)

`UPat.var("x") // 2` is non-commutative, so its sources are a tuple: `(UPat(None, name=x), UPat({Ops.CONST}, name=None))`. The rule `x + 0 → x` from section 2.1 is written, in full:

```
(UPat.var("x") + 0, lambda x: x)
```

2.3 Rules and the `PatternMatcher`

A rule is a pattern and a Python function. When the pattern matches, the function is called with the bindings as keyword arguments, and returns either a replacement UOp or `None`, meaning "on closer inspection, this rule does not apply". The `None` escape hatch is important: many conditions cannot be expressed as patterns ("`c` is positive", "the bounds of `x` fit in `[0, c)`"), so they go in the function.

uopc/rewrite.py · PatternMatcher

```
class PatternMatcher:
  def __init__(self, rules: list[Rule]):
    self.rules = rules
    # index by root op so we only try rules that could possibly match
    self.by_op: dict[Ops, list[Rule]] = {}
    for pat, fn in rules:
      for op in (pat.op if pat.op is not None else Ops):
        self.by_op.setdefault(op, []).append((pat, fn))

  def __add__(self, other: PatternMatcher) -> PatternMatcher: return PatternMatcher(self.rules +
other.rules)

  def rewrite(self, u: UOp, ctx=None) -> UOp | None:
    for pat, fn in self.by_op.get(u.op, ()):
      for binding in pat.match(u, {}):
        kw = {**binding, "ctx": ctx} if ctx is not None else binding
        if (ret := fn(**kw)) is not None and ret is not u: return ret
    return None
```

Order is priority. `rewrite` tries rules in the order given and returns the first non-`None` result. Section 2.1 showed that this order is observable. A rule returning the very node it was given is treated as "no change" (`ret is not u`): with hash consing, a rule that rebuilds an identical node returns the identical object, so this check is exact and costs nothing.

Indexing by op. Trying all 44 rules of `symbolic + evaluate` on every node would be wasteful: a rule whose pattern is an ADD can't match a MUL. The constructor files each rule under every op its root pattern can match (all ops, for a pattern without an op), and `rewrite` only tries that list. For the real rule set:

```
>>> pm = symbolic + evaluate
>>> len(pm.rules)
44
>>> {op.name: len(pm.by_op.get(op, [])) for op in Ops if pm.by_op.get(op)}
{'CALL': 1, 'INDEX': 3, 'CAST': 4, 'RECIP': 3, 'ADD': 14, 'MUL': 10, 'CMPLT': 6, 'FLOORDIV': 8,
 'FLOORMOD': 7, 'WHERE': 7, 'CMPNE': 6, 'MAX': 5, 'EXP2': 3, 'LOG2': 3, 'SQRT': 3, 'SIN': 3,
 'RESHAPE': 1, 'EXPAND': 1, 'PERMUTE': 1, 'SHRINK': 1, 'REDUCE': 1}
```

CONST, PARAM and STACK nodes, which are most leaves, get zero candidates and are skipped immediately. (tinygrad goes further and compiles each pattern into a specialized Python function; that is Exercise 10.)

Composition. `pm1 + pm2` is a new matcher with `pm1`'s rules first. `uopc` builds its rule sets this way: `symbolic = fold + PatternMatcher([...])`, `run` uses `symbolic + evaluate`.

Context. `graph_rewrite(..., ctx=something)` passes `ctx=something` to rule functions as an extra keyword argument. Rangeify (Chapter 6) uses it to carry per-pass state. Note the implementation: when `ctx` is not `None`, *every* rule in the matcher receives it. A rule set that needs context and a rule set that doesn't (like `symbolic`, whose lambdas take only their pattern variables) cannot be mixed in one call; see Pitfall 5.

2.4 `graph_rewrite`: the engine

The job: given a root UOp and a matcher, return the root with every node rewritten to normal form, bottom-up.

First attempt: recursion

The most direct implementation is four lines:

```
def rewrite_rec(u, pm):
  n = u.replace(src=tuple(rewrite_rec(s, pm) for s in u.src))   # normalize sources first
  r = pm.rewrite(n)
  return n if r is None else rewrite_rec(r, pm)                  # a rule fired: normalize the
result
```

It is correct, and it is the right mental model. It has two fatal problems.

It walks trees, not DAGs. Nothing remembers that a node was already rewritten, so a node with `k` users is rewritten `k` times, recursively. For `x` squared `d` times (`t = t * t`, a DAG of `d + 1` nodes) we can count rule lookups:

```
depth  5:   6 nodes   recursive tree walk       63 rule lookups   graph_rewrite   6
depth 10:  11 nodes   recursive tree walk     2047 rule lookups   graph_rewrite  11
depth 15:  16 nodes   recursive tree walk    65535 rule lookups   graph_rewrite  16
depth 20:  21 nodes   recursive tree walk  2097151 rule lookups   graph_rewrite  21
```

`2^(d+1) - 1` versus `d + 1`. Real graphs are full of sharing (every weight is used by the forward and backward pass, every index expression is shared by every load), so this is not academic.

It recurses. On the 20,000-ADD chain from `test_deep_graph_no_recursion_error`:

```
RecursionError: maximum recursion depth exceeded
```

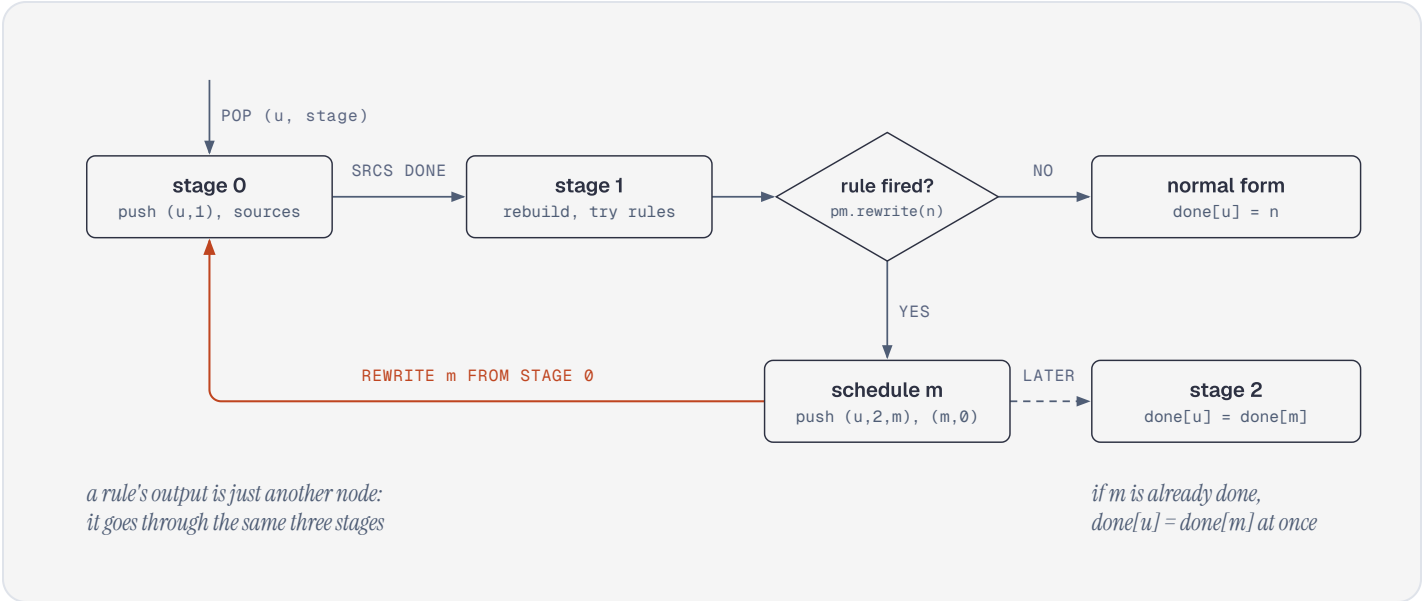
The real engine

`uopc/rewrite.py · graph_rewrite`

```python
def graph_rewrite(root: UOp, pm: PatternMatcher, ctx=None, max_steps: int = 10_000_000) -> UOp:
  """Bottom-up greedy rewrite to a fixed point. Iterative, so deep graphs don't blow the stack.

  Stage 0: schedule children.   Stage 1: children done -> rebuild, try rules.
  Stage 2: a rule fired and we rewrote its result -> record that as our answer.
  """
  done: dict[UOp, UOp] = {}
  stack: list[tuple[UOp, int, UOp | None]] = [(root, 0, None)]
  steps = 0
  while stack:
    u, stage, aux = stack.pop()
    if u in done and stage != 2: continue
    if stage == 0:
      stack.append((u, 1, None))
      stack.extend((s, 0, None) for s in reversed(u.src) if s not in done)
    elif stage == 1:
      src = tuple(done[s] for s in u.src)
      n = u if src == u.src else u.replace(src=src)
      steps += 1
      if steps > max_steps: raise RuntimeError("graph_rewrite did not converge (a rule loops?)")
      if (m := pm.rewrite(n, ctx)) is None: done[u] = n
      elif m in done: done[u] = done[m]
      else: stack.extend([(u, 2, m), (m, 0, None)])
    else:
      done[u] = done[aux]
  return done[root]
```

Two data structures: `done`, the memo table mapping each original node to its normal form, and an explicit `stack` of work items `(node, stage, aux)`. Each node goes through up to three stages:

- **Stage 0, "schedule my sources".** Push myself back as stage 1, then push every source that is not already done as stage 0. Because the stack is LIFO, all the sources (and their sources, and so on) are finished before the stage-1 item comes back up. The `reversed` makes sources come off the stack left to right, so the traversal order matches `toposort`.
- **Stage 1, "my sources are done".** Rebuild the node from its sources' normal forms (`n`, which is just `u` if nothing below changed). Try the rules on `n`.
- No rule applies: `n` is the normal form. Record `done[u] = n`.
- A rule returned `m`, and `m` has already been normalized: `done[u] = done[m]`.
- Otherwise `m` is new and might itself be a redex, or contain new redexes. Push `(u, 2, m)` and then `(m, 0, None)`: normalize `m` completely, then come back.
- **Stage 2, "the thing I rewrote to is done".** `done[u] = done[aux]`.



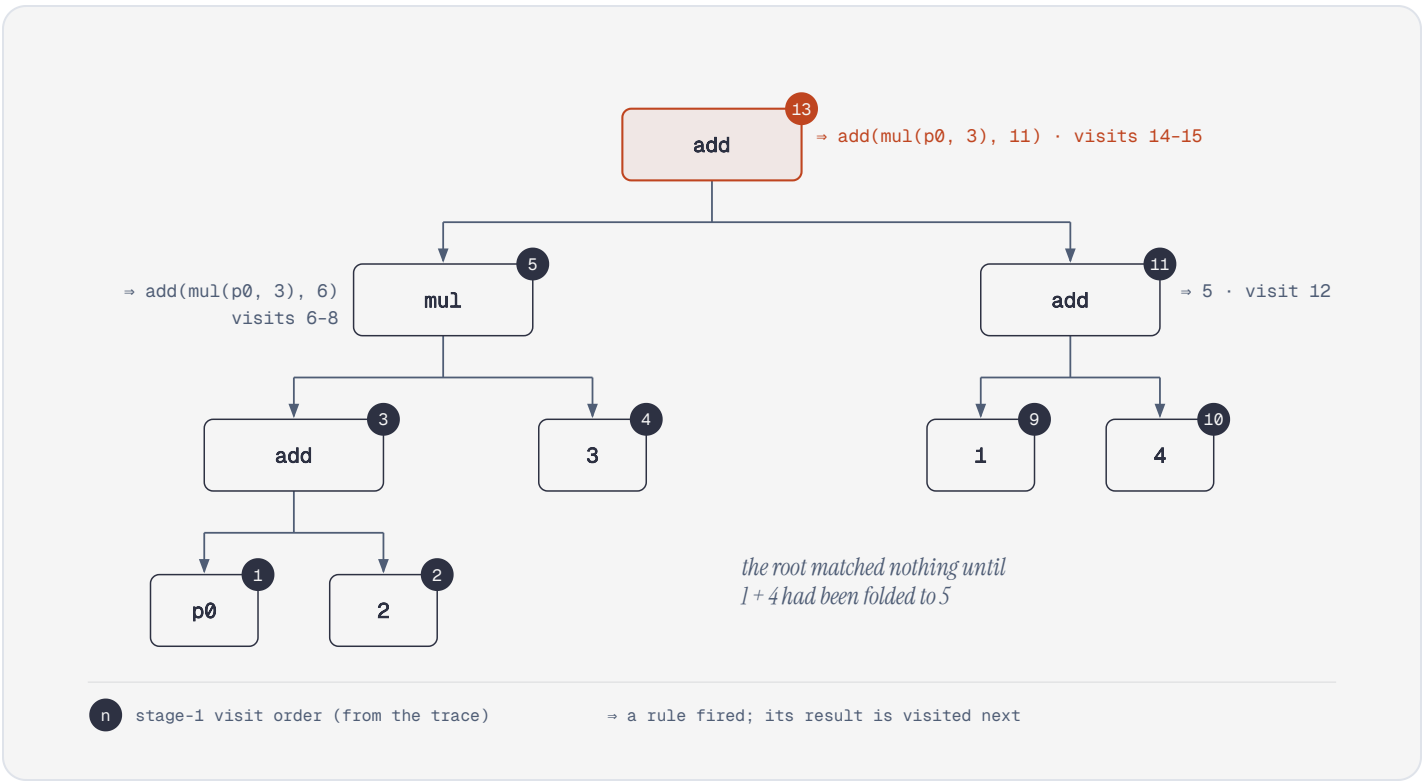
The first line of the loop skips work items for nodes that are already finished: a node with two users is pushed twice but processed once. Stage-2 items are exempt, because they carry the final answer for their node and must always record it.

Watching it run

To see the traversal, subclass `PatternMatcher` so that `rewrite` prints every call. This is a debugging trick worth keeping around:

```
def show(u):
  if u.op is Ops.CONST: return repr(u.arg)
  if u.op is Ops.PARAM: return f"p{u.arg.slot}"
  return f"{u.op.name.lower()}({', '.join(show(s) for s in u.src)})"

class Trace(PatternMatcher):
  def rewrite(self, u, ctx=None):
    r = super().rewrite(u, ctx)
    print(f"  visit {show(u):28s} -> {show(r) if r is not None else '.'}")
    return r

p0 = UOp(Ops.PARAM, (), ParamArg(0, dtypes.i32))
e = (p0 + 2) * 3 + (UOp.const(1) + UOp.const(4))
out = graph_rewrite(e, Trace(symbolic.rules))
```

```
program: add(mul(add(p0, 2), 3), add(1, 4))
  visit p0                          -> .
  visit 2                           -> .
  visit add(p0, 2)                  -> .
  visit 3                           -> .
  visit mul(add(p0, 2), 3)          -> add(mul(p0, 3), 6)
  visit mul(p0, 3)                  -> .
  visit 6                           -> .
  visit add(mul(p0, 3), 6)          -> .
  visit 1                           -> .
  visit 4                           -> .
  visit add(1, 4)                   -> 5
  visit 5                           -> .
  visit add(add(mul(p0, 3), 6), 5)  -> add(mul(p0, 3), 11)
  visit 11                          -> .
  visit add(mul(p0, 3), 11)         -> .
result: add(mul(p0, 3), 11)
```



Read the trace against the algorithm:

- Leaves first, left to right: `p0`, `2`, then their user `add(p0, 2)`, which no rule changes.
- `mul(add(p0, 2), 3)` matches the rule `(x + c1) * c2 → x*c2 + c1*c2` and becomes `add(mul(p0, 3), 6)`. That is a *new* node, so it goes back through stage 0: its new pieces `mul(p0, 3)` and `6` are visited, then the new ADD itself. Nothing fires, so it is the normal form of the old MUL. Note that `p0` and `3` are not visited again: they are already in `done`.

- The right subtree folds `add(1, 4)` to `5`.
- The root is rebuilt from the normal forms of its sources: `add(add(mul(p0, 3), 6), 5)`. That is not the node we started with, and *now* the reassociation rule `(x + c1) + c2 → x + (c1+c2)` matches, giving `add(mul(p0, 3), 11)`. It is re-normalized (only `11` is new) and recorded as the answer for the root.

The root's rule could only fire *after* its sources were rewritten: the original root was `add(mul(...), add(1, 4))`, which matches nothing. This is why bottom-up order matters.

Why results are rewritten again

Stage 2 exists because a rule's output is built by ordinary Python code (`x * c2 + c1.arg * c2.arg`) from ordinary constructors, and nothing guarantees that the pieces it builds are in normal form. Here is a "one-pass" rewriter that applies rules bottom-up but never looks at a rule's result again:

```
def one_pass(root, pm):
  done = {}
  for u in root.toposort():
    n = u.replace(src=tuple(done[s] for s in u.src))
    done[u] = pm.rewrite(n) or n
  return done[root]
```

On `(p0*2 + p1) * 3`, the distribution rule turns the MUL into `(p0*2)*3 + p1*3`, and the one-pass rewriter leaves `(p0*2)*3` unsimplified:

```
one_pass:                          graph_rewrite:
%0 = param : slot=0 dtype=i32      %0 = param : slot=0 dtype=i32
%1 = mul %0, 2                     %1 = mul %0, 6
%2 = mul %1, 3                     %2 = param : slot=1 dtype=i32
%3 = param : slot=1 dtype=i32      %3 = mul %2, 3
%4 = mul %3, 3                     %4 = add %1, %3
%5 = add %2, %4
```

For evaluation (section 2.6) the failure is total: every rule that pushes an operation into a STACK creates new, unevaluated operations. One pass over `a * b + a` with `a = [1, 2, 3]`, `b = [10, 20, 30]` leaves:

```
%0 = mul 1, 10
%1 = add %0, 1
...
%6 = stack %1, %3, %5
```

Memoization, and why it is sound

`done` maps an *original* node to its normal form, and the check `if u in done` skips any node reached a second time. That is only correct if rewriting is a *pure function of the node*: the same node must always rewrite to the same result. Two things make that true. Hash consing means "the same node" is a well-defined, O(1)-checkable notion. And rules are pure functions of their bindings (they do not consult global state that changes during the rewrite). A rule that, say, counted how many times it fired and behaved differently the tenth time would break memoization silently. If a rule needs state, it belongs in `ctx`, and the result is still memoized per call of `graph_rewrite`.

The `elif m in done` shortcut is the memo table paying off again: rewriting `mul(x, 1)` to `x`, where `x` was normalized long ago, finishes in one step.

Termination, detected

If the rules loop, so does the engine. The guard is `max_steps`: one step per stage-1 visit, and an exception after ten million. Here is a commutativity rule, the textbook non-terminating rule:

```
>>> swap = PatternMatcher([(UPat(Ops.ADD, src=(UPat.var("x"), UPat.var("y"))), lambda x, y:
UOp(Ops.ADD, (y, x)))])
>>> graph_rewrite(a + b, swap, max_steps=1000)
RuntimeError: graph_rewrite did not converge (a rule loops?)
```

`a + b` rewrites to `b + a`, which is a new node, which rewrites to `a + b`, which is *not yet in* `done` (it is still waiting in stage 2), and so on. A hard limit is crude, but it turns a hang into an error with a hint, which is the difference between a five-minute bug and a lost afternoon.

How do real rule sets avoid loops? By making every rule decrease *something*. The canonicalization rule in `symbolic` moves constants to the right of commutative ops (`c + y → y + c`), but only when `y` is not itself a constant, so applying it twice is impossible. Reassociation moves constants *outward*, toward each other, where they fold. Chapter 3 discusses termination of the symbolic rules in detail.

Costs

Every node is visited once in stage 0 and once in stage 1, plus one visit per node created by a rule. Each stage-1 visit tries the rules indexed under the node's op, and each try is a pattern match whose cost depends on the pattern's size. For the 44-rule set, a stage-1 visit costs on the order of 100 µs of Python, which is

perfectly adequate for graphs of tens of thousands of nodes and the reason the compiler caches whole compiled plans in Chapter 7 rather than trying to make rewriting faster.

2.5 Writing your own rules

The week-1 test for the engine is a custom matcher with a single rule, strength reduction of `x * 2` into `x + x`:

tests/test_w01_uops.py · test_custom_pattern_matcher

```
def test_custom_pattern_matcher():
  pm = PatternMatcher([(UPat(Ops.MUL, src=[UPat.var("x"), UPat.cvar("c", dtype=dtypes.i32)]),
                        lambda x, c: x + x if c.arg == 2 else None)])
  x = param(0)
  assert graph_rewrite(x * 2, pm) is x + x
```

Three things to notice: the dtype constraint on the constant (`dtype=dtypes.i32`, because `2.0` is a different rule for floats), the list sources (so `2 * x` also matches), and the function returning `None` when `c` is not 2, rather than encoding "arg is 2" in the pattern. (It could: `UPat(Ops.CONST, arg=2, name="c")`. Both are fine.)

A worked example: `where(c, 1, 0)` is a cast

A comparison used as a 0/1 integer often appears as `cond.where(1, 0)`. In C, `(int)cond` is the same thing and cheaper. Here is the rule:

```
b2i = PatternMatcher([(UPat(Ops.WHERE, src=(UPat.var("c"), UPat(Ops.CONST, arg=1),
UPat(Ops.CONST, arg=0))),
                       lambda c: c.cast(dtypes.i32))])
```

```
%0 = param : slot=0 dtype=i32          %0 = param : slot=0 dtype=i32
%1 = param : slot=1 dtype=i32          %1 = param : slot=1 dtype=i32
%2 = cmplt %0, %1                      %2 = cmplt %0, %1
%3 = where %2, 1, 0                    %3 = cast %2 : dtype=i32
```

It works, and it has a bug. Apply it to a *bool* WHERE, `c.where(True, False)`:

```
>>> wb = c.where(True, False)
>>> graph_rewrite(wb, b2i).dtype
dtypes.i32          # was dtypes.bool
```

The pattern's `arg=1` is compared with `==`, and `True == 1`, `False == 0`. The same Python numeric tower that caused the hash-consing bug of Chapter 1 is back, this time in pattern matching: `UPat(Ops.CONST, arg=0)` matches `0`, `0.0`, `-0.0` and `False`. The rewrite silently changed a node's dtype, and since every node above it was rebuilt from the replacement, the type checker would only complain if some user now saw mismatched operands, possibly far away, possibly never. The fix is to constrain the dtype of the node being replaced:

```
UPat(Ops.WHERE, dtype=dtypes.i32, src=(UPat.var("c"), UPat(Ops.CONST, arg=1), UPat(Ops.CONST, arg=0)))
```

> **Key idea:** a rewrite must preserve the dtype and shape of the node it replaces. The engine does not check this. Every rule author must, and a debug assertion in your engine is cheap insurance (Exercise 4).

A checklist for every rule

1. **Is it sound?** For all values of the variables, do both sides compute the same thing, *in the target's arithmetic*? (Int32 wraps; floats round; `x * 0 → 0` is false for `x = inf` or NaN, which is why `symbolic` restricts it to ints.) Test it by evaluating both sides on random inputs.
2. **Does it preserve dtype and shape?**
3. **Does it terminate together with the other rules?** Name the measure it decreases.
4. **Can its pattern be more specific?** Every condition you move from the function into the pattern is checked faster and documents itself.
5. **Is the order right?** If another rule matches the same nodes, which one should win?

Rules with context

A matcher that binds PARAMs to values (a *partial evaluator*) needs to know the values. That is what `ctx` is for:

```
bind = PatternMatcher([(UPat(Ops.PARAM, name="p"),
                        lambda p, ctx: UOp.const(ctx[p.arg.slot], p.dtype) if p.arg.slot in ctx
else None)])
e = (p0 + 2) * p1 + p0 * 0
graph_rewrite(e, bind, ctx={0: 5})
```

```
%0 = add 5, 2
%1 = param : slot=1 dtype=i32
%2 = mul %0, %1
%3 = mul 5, 0
%4 = add %2, %3
```

Combining it with `symbolic` in one call does not work, because `ctx` is passed to *every* rule:

```
TypeError: _fold() got an unexpected keyword argument 'ctx'
```

Run two passes instead: `simplify(graph_rewrite(e, bind, ctx={0: 5}))` gives

```
%0 = param : slot=1 dtype=i32
%1 = mul %0, 7
```

2.6 Running programs by rewriting

Now the payoff. The syllabus says: *the graph rewrite engine should be able to "run" these programs via const folding*. A closed week-1 program (no free PARAMs) is a DAG of constants, STACKs, INDEXes, ALU ops and CALLs. If we have rules that eliminate every non-constant op, the normal form of a program is its value. Rewriting *is* the interpreter.

The rules come in two groups. `fold`, the constant folder, evaluates any scalar ALU op whose sources are all CONSTs, using `exec_alu`, the reference semantics of every ALU op:

`uopc/symbolic.py · _fold`

```
def _fold(x: UOp) -> UOp | None:
  if x.shape != () or not all(s.op is Ops.CONST for s in x.src): return None
  r = exec_alu(x.op, x.dtype, [s.arg for s in x.src])
  return None if r is None else UOp.const(r, x.dtype)
```

`exec_alu` (int32 wraparound, f32 rounding, floor division, C-style casts, refusing to fold undefined behavior) is the subject of Chapter 3. Here we only need to know that for constant scalar inputs it returns the same constant the C program would compute.

The second group, `evaluate`, eliminates everything else:

uopc/symbolic.py · evaluate

```
evaluate = PatternMatcher([
  (UPat({Ops.RESHAPE, Ops.EXPAND, Ops.PERMUTE, Ops.SHRINK, Ops.REDUCE}, name="u"),
_eval_movement),
  (UPat(Ops.INDEX, src=(x,)), lambda x: x),
  (UPat(Ops.INDEX, src=(UPat(Ops.INDEX, name="a"),), allow_any_len=True, name="b"),
   lambda a, b: a.src[0].index(*a.src[1:], *b.src[1:])),
  (UPat(Ops.INDEX, src=(UPat(Ops.STACK, name="s"), UPat.cvar("i")), allow_any_len=True,
name="u"),
   lambda s, i, u: _index_stack(s, i, u.src[2:])),
  (UPat(ALU, name="x"), _push_stack),
  (UPat(Ops.CALL, name="call"), inline_call),
])
```

and `run` is one line:

uopc/symbolic.py · run

```
def run(u: UOp) -> UOp: return graph_rewrite(u, symbolic + evaluate)
```

along with a helper that converts the resulting CONST/STACK tree into Python lists:

uopc/symbolic.py · to_python

```
def to_python(u: UOp):
  """A fully evaluated program as nested python lists."""
  if u.op is Ops.CONST: return u.arg
  if u.op is Ops.STACK: return [to_python(s) for s in u.src]
  raise ValueError(f"program did not evaluate to a value, stuck at {u.op}")
```

Pushing ALU ops through STACK

`fold` only handles scalars. `[1, 2, 3] * [10, 20, 30]` is a MUL of two STACKs. The rule: an elementwise op of STACKs is a STACK of elementwise ops.

```
f(STACK(a0, a1, ...), STACK(b0, b1, ...))  →  STACK(f(a0, b0), f(a1, b1), ...)
```

uopc/symbolic.py · _push_stack

```python
def _push_stack(x: UOp) -> UOp | None:
  # f(STACK(a0, a1), STACK(b0, b1)) -> STACK(f(a0, b0), f(a1, b1))
  if x.shape == () or not any(s.op is Ops.STACK for s in x.src): return None
  return UOp(Ops.STACK, tuple(x.replace(src=tuple(_elem(s, i) for s in x.src)) for i in
range(x.shape[0])))
```

uopc/symbolic.py · _elem

```python
def _elem(s: UOp, i: int) -> UOp: return s.src[i] if s.op is Ops.STACK else s.index(i)
```

The rule fires for any ALU op with a shape and at least one STACK source. Sources that are not STACKs are split with INDEX (`_elem`), so `STACK(...) * PARAM(shape=(3,))` becomes a STACK of `INDEX(param, i) * ...`, which stays symbolic: the rule works even on programs that are not fully known. The rule produces new scalar ALU nodes, which the engine (stage 2) immediately rewrites, and `fold` finishes them. Here is `a * b + a` with `a = [1, 2, 3]`, `b = [10, 20, 30]`, tracing only the rules that fired:

```
%0 = stack 1, 2, 3
%1 = stack 10, 20, 30
%2 = mul %0, %1
%3 = add %2, %0

  mul(stack(1, 2, 3), stack(10, 20, 30))   => stack(mul(1, 10), mul(2, 20), mul(3, 30))
  mul(1, 10)                               => 10
  mul(2, 20)                               => 40
  mul(3, 30)                               => 90
  add(stack(10, 40, 90), stack(1, 2, 3))   => stack(add(10, 1), add(40, 2), add(90, 3))
  add(10, 1)                               => 11
  add(40, 2)                               => 42
  add(90, 3)                               => 93
[11, 42, 93]
```

A WHERE works the same way; its condition is a STACK of bools produced by pushing CMPLT through:

```
  cmplt(stack(1, 2, 3), stack(2, 2, 2))    => stack(cmplt(1, 2), cmplt(2, 2), cmplt(3, 2))
  cmplt(1, 2)                              => True
  cmplt(2, 2)                              => False
  cmplt(3, 2)                              => False
  where(stack(True, False, False), stack(1, 2, 3), stack(10, 20, 30)) => stack(where(True, 1,
10), where(False, 2, 20), where(False, 3, 30))
  where(True, 1, 10)                       => 1
  where(False, 2, 20)                      => 20
  where(False, 3, 30)                      => 30
[1, 20, 30]
```

INDEX of STACK

Three INDEX rules complete the picture. `INDEX(x)` with no indices is `x`. `INDEX(INDEX(x, i), j)` is `INDEX(x, i, j)`, a normal form with one INDEX. And the one that does the work: an INDEX of a STACK at a constant position is that element (indexed further by the remaining indices, if any).

uopc/symbolic.py · _index_stack

```python
def _index_stack(s: UOp, i: UOp, rest: tuple[UOp, ...]) -> UOp | None:
  if not 0 <= i.arg < len(s.src): raise IndexError(f"index {i.arg} out of bounds for STACK of
{len(s.src)}")
  return s.src[i.arg].index(*rest) if rest else s.src[i.arg]
```

```
  index(stack(stack(1, 2, 3), stack(10, 20, 30)), 1) => stack(10, 20, 30)
  index(stack(10, 20, 30), 2)              => 30
30
```

An out-of-range constant index is an error in the program, so the rule *raises* instead of returning `None` (which would leave the INDEX in place and make `to_python` fail later with a less useful message):

```
IndexError: index 5 out of bounds for STACK of 3
```

Movement ops on constant data

`run` can also evaluate RESHAPE, EXPAND, PERMUTE, SHRINK and REDUCE, the week-4 ops, as long as their input is a fully known CONST/STACK tree. The approach is the bluntest one possible: flatten the input to a Python list, compute each output element's source position with ravel/unravel index arithmetic, and rebuild a STACK tree.

uopc/symbolic.py · _eval_movement

```
def _eval_movement(u: UOp) -> UOp | None:
  """Movement ops and REDUCE on fully known values: just compute them."""
  if (lit := _literal(u.src[0])) is None: return None
  flat, ishape = lit
  oshape, n_out = u.shape, prod(u.shape)
  match u.op:
    case Ops.RESHAPE: out = flat
    case Ops.EXPAND: out = [flat[_ravel([0 if s == 1 else x for x, s in zip(_unravel(i, oshape),
ishape)], ishape)] for i in range(n_out)]
    case Ops.PERMUTE:
      inv = [u.arg.index(k) for k in range(len(ishape))]
      out = [flat[_ravel([_unravel(i, oshape)[inv[k]] for k in range(len(ishape))], ishape)] for
i in range(n_out)]
    case Ops.SHRINK: out = [flat[_ravel([x + b for x, (b, _) in zip(_unravel(i, oshape), u.arg)],
ishape)] for i in range(n_out)]
    case Ops.REDUCE:
      rop, k = u.arg
      inner = prod(ishape[len(ishape) - k:])
      out = []
      for i in range(n_out):
        acc = flat[i * inner]
        for v in flat[i * inner + 1:(i + 1) * inner]: acc = exec_alu(rop, u.dtype, [acc, v])
        out.append(acc)
  return _from_flat(out, oshape, u.dtype)
```

The helpers are worth reading, because ravel/unravel is the core of rangeify in Chapter 6, where the same arithmetic is done *symbolically* on loop variables instead of on Python ints:

uopc/symbolic.py · _unravel

```
def _unravel(i: int, shape) -> list[int]:
  out = []
  for s in reversed(shape): out.append(i % s); i //= s
  return out[::-1]
```

uopc/symbolic.py · _ravel

```
def _ravel(idx, shape) -> int:
  i = 0
  for x, s in zip(idx, shape): i = i * s + x
  return i
```

Here is the second week-1 evaluation test, run with a trace. Note how the `.uir` sugar's `mul %1, 2` became a broadcast constant (RESHAPE + EXPAND), which the movement rules evaluate before the MUL can be

pushed through:

```
%0 = stack 1.0, 2.0, 3.0
%1 = stack %0, %0
%2 = reshape 2.0, (1, 1)
%3 = expand %2, (2, 3)
%4 = mul %1, %3
%5 = permute %4, (1, 0)
%6 = reduce %5 : op=add pop=1

  reshape(2.0, (1, 1))
      => stack(stack(2.0))
  expand(stack(stack(2.0)), (2, 3))
      => stack(stack(2.0, 2.0, 2.0), stack(2.0, 2.0, 2.0))
  mul(stack(stack(1.0, 2.0, 3.0), stack(1.0, 2.0, 3.0)), stack(stack(2.0, 2.0, 2.0), stack(2.0,
2.0, 2.0)))
      => stack(mul(stack(1.0, 2.0, 3.0), stack(2.0, 2.0, 2.0)), mul(stack(1.0, 2.0, 3.0),
stack(2.0, 2.0, 2.0)))
  mul(stack(1.0, 2.0, 3.0), stack(2.0, 2.0, 2.0))
      => stack(mul(1.0, 2.0), mul(2.0, 2.0), mul(3.0, 2.0))
  mul(1.0, 2.0)
      => 2.0
  mul(2.0, 2.0)
      => 4.0
  mul(3.0, 2.0)
      => 6.0
  permute(stack(stack(2.0, 4.0, 6.0), stack(2.0, 4.0, 6.0)), (1, 0))
      => stack(stack(2.0, 2.0), stack(4.0, 4.0), stack(6.0, 6.0))
  reduce(stack(stack(2.0, 2.0), stack(4.0, 4.0), stack(6.0, 6.0)), add, 1)
      => stack(4.0, 8.0, 12.0)
[4.0, 8.0, 12.0]
```

Look at the third rule firing. The outer MUL became a STACK of *two* row MULs, but only *one* of them appears in the trace. The two rows are identical, so hash consing made them the same node, and the memo table rewrote it once. This is term graph rewriting doing its job without anyone asking.

CALL inlining is beta reduction

The last evaluation rule inlines function calls. In the lambda calculus, applying a function to an argument is *beta reduction*: `(λx. x * x) 7 → 7 * 7`, the body with the parameter replaced by the argument. A CALL is exactly a lambda application, with PARAM slots in place of variable names:

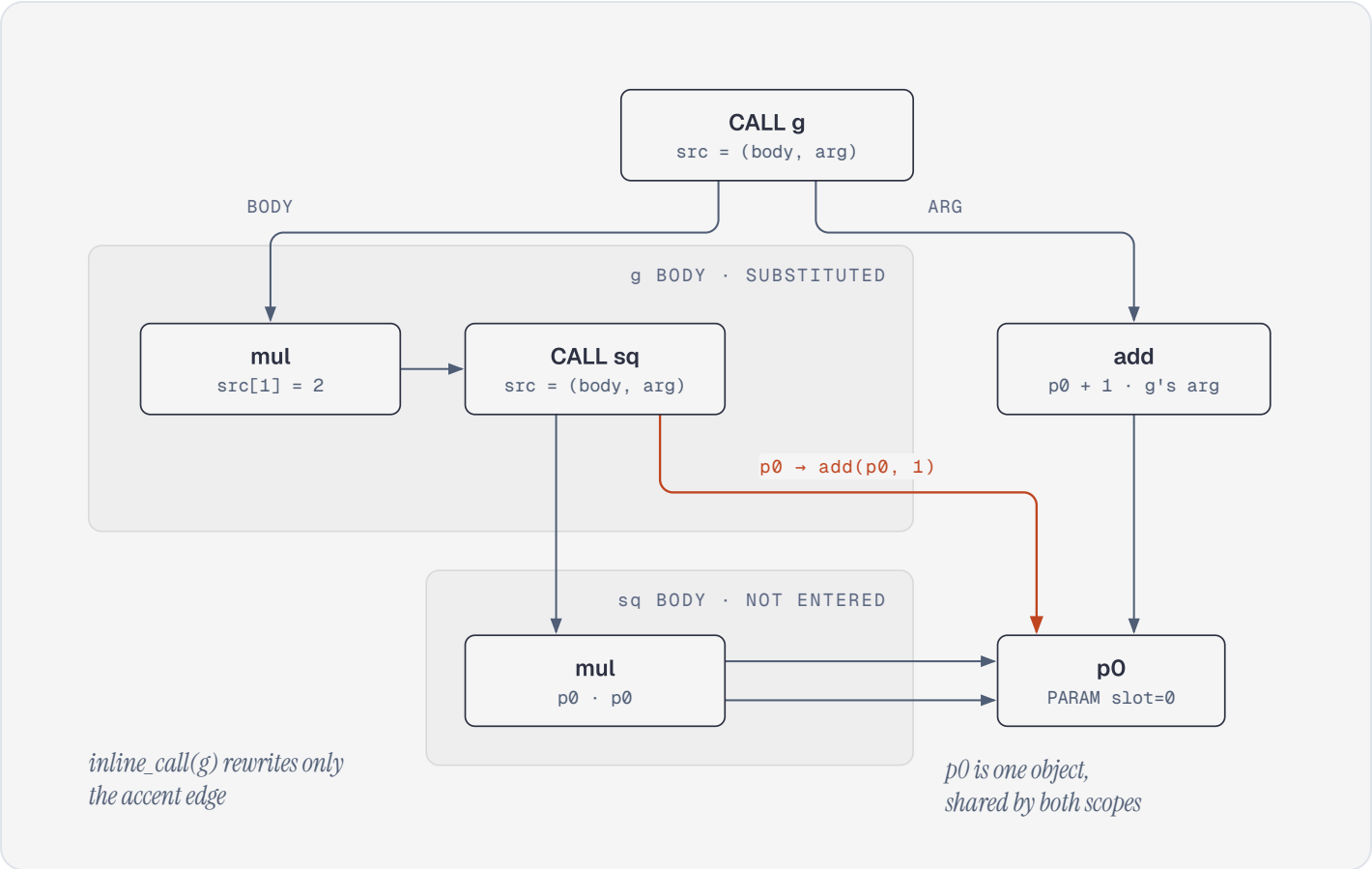
```
%0 = param : slot=0 dtype=i32
%1 = mul %0, %0
%2 = call %1, 7 : name=sq                  run(...) → 49
```

The rule is `(UPat(Ops.CALL, name="call"), inline_call)`:

uopc/symbolic.py · inline_call

```
def inline_call(call: UOp) -> UOp:
  """Beta reduction: replace each PARAM of the body with its argument.
  Scoped: a nested CALL's body (its src[0]) has its own PARAMs, so we never rewrite inside it."""
  args = {p: call.src[1 + p.arg.slot] for p in call_params(call.src[0])}
  order, seen, stack = [], set(), [(call.src[0], False)]
  while stack:
    u, done = stack.pop()
    if done: order.append(u); continue
    if u in seen: continue
    seen.add(u)
    stack.append((u, True))
    stack.extend((s, False) for s in (u.src[1:] if u.op is Ops.CALL else u.src))
  new: dict[UOp, UOp] = {}
  for u in order:
    if u in args: new[u] = args[u]; continue
    src = (u.src[0], *(new[s] for s in u.src[1:])) if u.op is Ops.CALL else tuple(new[s] for s in
u.src)
    new[u] = u if src == u.src else u.replace(src=src)
  return new[call.src[0]]
```

It builds a map from each of the body's PARAMs to the corresponding argument, then rebuilds the body bottom-up, substituting. It is `UOp.substitute` from Chapter 1 with one important difference, in two places: when it meets a nested CALL, it *does not descend into its body* (`u.src[1:] if u.op is Ops.CALL`), and when rebuilding a nested CALL, it keeps `u.src[0]` unchanged. The PARAM-finding helper `call_params` has the same restriction.



The scoping bug

Why the restriction? Recall from Chapter 1 that PARAMs are hash consed: `PARAM slot=0 dtype=i32` is *one object*, shared by every function body that uses slot 0. Consider three nested functions:

```
f(5)  where  f(y) = g(y + 1),  g(z) = sq(z) * 2,  sq(w) = w * w
```

The expected answer is `g(6) = sq(6) * 2 = 72`. As UOps, all three bodies use slot 0, so all three "different" parameters `y`, `z`, `w` are the same node `p0`:

```
f[g[mul(sq[mul(p0, p0)](p0), 2)](add(p0, 1))](5)
```

(written as `name[body](args)`). Now inline `g` with a naive substitution that replaces `p0` everywhere reachable from `g`'s body:

```
def naive_inline(call): return call.src[0].substitute({p0: call.src[1]})
```

```
naive  inline g: mul(sq[mul(add(p0, 1), add(p0, 1))](add(p0, 1)), 2)
scoped inline g: mul(sq[mul(p0, p0)](add(p0, 1)), 2)
```

The naive version reached into `sq`'s body and replaced *its* parameter, which belongs to a different scope. `sq`'s body is now `(p0+1)*(p0+1)` where `p0` is `sq`'s own parameter, so `sq(z)` computes `(z+1)²`. The program still type checks (the body still uses slot 0), and it silently computes the wrong answer:

```
run: 72   by hand: g(6) = sq(6)*2 = 72
naive result: 98
```

`98 = sq'(7) * 2 = 7 * 7 * 2`: the argument was incremented twice. In other cases the type checker does catch it: if the argument of the outer call is a constant, the naive substitution removes the inner body's PARAM entirely and rebuilding the inner CALL fails with `UOpError: CALL needs params 0..0, body has []`. That is the lucky case.

This is the classic *variable capture* problem of substitution in the lambda calculus, in a slightly unusual form. Textbook capture happens when a substituted term mentions a variable that a binder inside the body also binds; the cure is to rename the inner binder (α-conversion). Here the "variables" are positional slots and hash consing has merged all the same-numbered slots, so renaming is not available. The cure is to respect scopes: a CALL's body is a binder, and substitution must not cross it.

Note that `run` would get the right answer even with the naive inliner, because bottom-up rewriting reaches the innermost CALL first: `sq(p0)` is inlined into `g`'s body (as `p0 * p0`, which happens to be correct since its argument is `p0` itself) before `g` is inlined. But `inline_call` is a library function, and any pass that calls it directly on an outer CALL (a top-down inliner, a pass that inlines only some functions) would hit the bug. Correctness should not depend on the caller's traversal order.

Interpreter versus compiler

`run` is a complete interpreter for week-1 programs, written in about 80 lines of rules. How good an interpreter is it?

```
n=   10  run()    17.35 ms   first [0, 12, 26]
n=  100  run()    47.65 ms   first [0, 102, 206]
n= 1000  run()   141.11 ms   first [0, 1002, 2006]
```

(`a * b + a` for vectors of length `n`.) About 140 µs per element: every element of every intermediate value is a hash-consed node, every operation is a pattern match. numpy does all 1000 elements in 1.76 µs (`python -m timeit`). Nobody should run models this way.

But the interpreter has two jobs where speed is irrelevant. First, it is an **oracle**: a second, independent implementation of the semantics. When the C backend of Chapter 4 produces an answer, `run` can check it on small inputs; the course's graders use exactly this kind of reference evaluator. Second, it shows that the *same machinery* that optimizes a program can execute it. The line between "simplify" and "evaluate" is just how much is known: `symbolic` rewrites `x * 1` to `x` because the rule is true for every `x`; `evaluate` rewrites `2 * 3` to `6` because everything is known. A program with some PARAMs bound and some not is *partially evaluated* by the same rules (section 2.5's `bind` example). Partial evaluation, taken seriously, is a way of deriving compilers from interpreters (the Futamura projections); this book takes the more conventional route of generating C, but keep the idea in mind when Chapter 6 turns shape computations into loops using these same rules.

Pitfalls

1. CALL scoping (NOTES 2). Section 2.6 in full. Because PARAMs are hash consed, the PARAM of an inner function body is the same object as the PARAM of the outer one, and naive substitution while inlining the outer call rewrote the inner body too. Symptoms: either a `UOpError` about missing params when rebuilding the inner CALL (lucky), or a silently wrong value (98 instead of 72). Fix: `call_params` and `inline_call` never descend into a nested CALL's `src[0]`. Bottom-up rewriting also happens to inline inner calls first, which hid the bug in `run` until a test built nested calls by hand.

2. Recursion depth (NOTES 3). The first `graph_rewrite` was the four-line recursive version. A 20,000-node ADD chain raised `RecursionError`. Fix: the explicit stack with three stages. `test_deep_graph_no_recursion_error` is the regression test.

3. Forgetting the memo table. A recursive rewriter without `done` is exponential in the depth of sharing (`2097151` rule lookups instead of `21` for a 21-node graph). It works on every small test and hangs on the first real model. Memoize on the node; hash consing makes it O(1).

4. Patterns compare args with `==`. `UPat(Ops.CONST, arg=0)` matches `0`, `0.0`, `-0.0` and `False`; `arg=1` matches `True`. A rule meant for i32 constants fires on bools and changes a node's dtype (section 2.5's `where(c, 1, 0)` example). Add a `dtype=` constraint whenever a rule's correctness depends on the type, and check that replacements preserve dtype and shape.

5. `ctx` goes to every rule. When `graph_rewrite` is called with a context, every rule function in the matcher receives a `ctx=` keyword argument. Combining a context-using rule set with `symbolic` fails with `TypeError: _fold() got an unexpected keyword argument 'ctx'`. Either run separate passes, or give context-free rules a `ctx=None` parameter.

6. Rules that undo each other. `(a + b) * c → a*c + b*c` (distribute) and `a*c + b*c → (a+b) * c` (factor) are each sound and together loop forever. The engine reports `graph_rewrite did not converge (a rule loops?)` after `max_steps`. When it does, print the last few rewrites (a `Trace` matcher) and look for a cycle. The fix is always to pick one direction as canonical.

7. Order-dependent results. Rule order is priority. Moving a rule (or concatenating matchers in the other order) can change results, as `-0.0 + 0.0` showed. Treat the order of a rule list as part of its meaning, and put the most specific rules first.

Exercises

1. ★ **Predict the trace.** Using the `Trace` matcher from section 2.4 with `symbolic.rules`, predict every line printed for `graph_rewrite((p0 * 1) + (p0 * 1), Trace(symbolic.rules))` where `p0` is an i32 PARAM. Remember hash consing. Then run it.

2. ★ **Your first rule.** Write a matcher with one rule that rewrites `MAX(x, x) → x` and a test for it. (`symbolic` already has this rule; write it yourself, then compare.) Why is the source pattern a tuple and not a list, even though MAX is commutative?

3. ★ **Find the bug.** A student writes the rule `(UPat.var("x") * 0, lambda x: x.const_like(0))` for all dtypes. Give a float value of `x` for which it is unsound, and explain how `symbolic` avoids the problem.

4. ★★ **Type-preserving rewrites.** Add an optional check to your `graph_rewrite`: when a rule returns `m` for node `n`, assert `m.dtype == n.dtype and m.shape == n.shape`, with an error message naming the rule (hint: `PatternMatcher.rewrite` knows which rule fired). Run the whole test suite with the check on. Then catch the `where(c, 1, 0)` bug from section 2.5 with it.

5. ★★ **Top-down rewriting.** Write `graph_rewrite_topdown`, which tries rules on a node *before* rewriting its sources (and again after). Find a rule set and a program where it produces a different normal form than bottom-up, and one where it is much faster (hint: a rule that discards a large subtree, like `WHERE(True, x, y) → x`).

6. ★★ **Soundness by testing.** Write a function `check_rule(pat, fn, n=1000)` that generates random i32 inputs, builds random instances of the pattern, applies the rule, and compares `run(before)` and `run(after)` after binding PARAMs to random constants. Run it on every rule in `symbolic` (you will need to synthesize instances from patterns). Did you find any rule that is unsound for some int32 values? (Think about overflow.)

7. ★★ **Inlining by hand.** Implement `inline_call` yourself, then write a test with three levels of nesting where the outer argument mentions the outer PARAM (like section 2.6's `f`, `g`, `sq`), that fails for the naive version and passes for yours. Call `inline_call` directly on the middle CALL, not via `run`.

8. ★★ **A rewrite tracer.** Add a `DEBUG_REWRITE=1` environment variable to your engine that prints each rule that fires as `rule-name: before -> after` using `to_uir`-style one-line output. Use it to explain every step of `simplify(((x * 4) + 8) // 4)` for an i32 PARAM `x`.

9. ★★★ **Equality saturation.** A greedy rewriter commits to the first rule that fires, which is why order matters. *Equality saturation* (the `egg` library) instead applies all rules in all ways, recording equivalences in an e-graph, and extracts the best term at the end. Implement a tiny e-graph (union-find over hash-consed nodes, congruence closure by rebuilding) for the rules `x+0→x`, `x*1→x`, `x*2→x+x`, `(a+b)+c→a+(b+c)`, commutativity, and extract the smallest term. Show that commutativity, which loops in `graph_rewrite`, is harmless here.

10. ★★★ **Compile your patterns.** `UPat.match` interprets the pattern tree on every call. Write a function that turns a `UPat` into Python source code for a specialized matcher (nested `if`s on `u.op`, `u.src[0].op`, ..., with the permutations unrolled), `exec` it, and use it in `PatternMatcher`. Measure `simplify` on a large index expression before and after. (tinygrad compiles its patterns too; find its pattern compiler in `tinygrad/uop/` after you've tried.)

Solutions to ★ exercises

1. `p0 * 1` is built once (hash consing), so the root is `add(m, m)` with `m = mul(p0, 1)`:

```
visit p0 -> . visit 1 -> . visit mul(p0, 1) -> p0 visit add(p0, p0) -> mul(p0, 2) visit 2 -> .
visit mul(p0, 2) -> .
```

The identity rule turns `m` into `p0`, which is already done, so it finishes immediately. The root is rebuilt as `add(p0, p0)` and the rule `xi + xi → x * 2` fires. The second use of `m` is never visited: it is the same node, already in `done`.

1. `python pm = PatternMatcher([(UPat(Ops.MAX, src=(UPat.var("x"), UPat.var("x"))), lambda x: x)]) p = UOp(Ops.PARAM, (), ParamArg(0, dtypes.f32)) assert graph_rewrite(p.maximum(p), pm) is p` Both

sub-patterns are the same variable, so both orders bind the same thing; a list would just try the same match twice.

2. `x = inf` (or NaN): `inf * 0` is NaN in IEEE arithmetic, not 0. `symbolic` restricts the rule to integer and bool dtypes: `(UPat.var("x", dtype=ints) * 0, ...)`, and `exec_alu` returns NaN for `inf * 0` when folding constants.

Quiz

1. What is a normal form? What two properties of a rewrite system together guarantee that every term has exactly one?
2. Why does `graph_rewrite` rewrite a node's sources before trying rules on the node?
3. What does stage 2 of `graph_rewrite` do, and what goes wrong without it?
4. A rule returns a node that is structurally identical to its input. What does `PatternMatcher.rewrite` do, and why is the check O(1)?
5. Why is `done` (the memo table) correct? What property must every rule have for it to be?
6. What does `UPat(Ops.ADD, src=[UPat.var("x"), UPat.cvar("c")])` match that `UPat(Ops.ADD, src=(UPat.var("x"), UPat.cvar("c")))` does not?
7. In beta reduction of a CALL, why must substitution not enter a nested CALL's body? Why doesn't α-renaming (the usual fix in the lambda calculus) apply here?
8. `run` evaluates `[1,2,3] * [10,20,30]`. Name the two rules that fire, in order.

Answers

1. A term with no redex (no rule applies anywhere). Termination (some normal form is always reached) and confluence (all rewrite paths reach the same one).
2. So that rules see sources in normal form (simpler rules), and because many rules can only match after their sources are rewritten (the root of `(p0+2)*3 + (1+4)` matched nothing until `1+4` became `5`).
3. It records the normal form of a rule's *result* as the answer for the original node, after the result has been fully rewritten. Without it (one-pass rewriting), the new nodes a rule builds are never simplified: `(p0*2)*3` stays unfolded, and evaluation leaves `mul(1, 10)` nodes unevaluated.
4. It treats it as "no change" (`ret is not u`) and tries the next rule. With hash consing a structurally identical node *is* the same object, so the check is a pointer comparison.

5. Rewriting must be a pure function of the node: same node in, same normal form out. Rules must not depend on mutable global state; hash consing makes "same node" well-defined.
6. `3 + x` (constant on the left). A list source tries both orders.
7. PARAMs are hash consed, so the inner body's slot-0 PARAM is the same object as the outer body's; substituting into the inner body changes a different function's parameter. There are no names to rename (slots are positional, and all same-slot PARAMs are one node), so scoping must be respected structurally instead.
8. The STACK push-through rule (`_push_stack`) turns the MUL into a STACK of three scalar MULs, then `fold` (constant folding via `exec_alu`) evaluates each scalar MUL.

Further reading

- Franz Baader and Tobias Nipkow, *Term Rewriting and All That*, Cambridge University Press, 1998. The standard textbook: termination orderings, confluence, critical pairs, Newman's lemma.
- Nachum Dershowitz and Jean-Pierre Jouannaud, "Rewrite Systems", in *Handbook of Theoretical Computer Science*, Vol. B, 1990. A dense survey of the theory.
- Eelco Visser, "A Survey of Rewriting Strategies in Program Transformation Systems", *Electronic Notes in Theoretical Computer Science* 57, 2001. Innermost, outermost, and programmable strategies in practical systems.
- Max Willsey et al., "egg: Fast and Extensible Equality Saturation", POPL 2021. The alternative to greedy rewriting (Exercise 9).
- Benjamin C. Pierce, *Types and Programming Languages*, MIT Press, 2002, chapters 5–6: the lambda calculus, substitution, variable capture, and de Bruijn indices.
- Neil D. Jones, Carsten K. Gomard and Peter Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall, 1993. Interpreters, specializers, and the Futamura projections.
- tinygrad, `tinygrad/uop/ops.py` (https://github.com/tinygrad/tinygrad): the production `UPat` / `PatternMatcher` / `graph_rewrite`, from which `uopc`'s are simplified.

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

`uopc/rewrite.py · 115 lines`

`uopc/symbolic.py · 302 lines`

`tests/test_w01_uops.py · 128 lines`

3. Constant folding and symbolic algebra

When rangeify (Chapter 6) turns a reshape into loops, it writes index expressions such as `((i*8 + j)//8) % 4`. That is just `i` with extra steps, and a C compiler will not always spot it: integer division and modulo are expensive, and gcc has to prove a lot before it can delete them. Our own compiler knows more than gcc does. It knows that `i` runs from 0 to 3 and `j` from 0 to 7, because it made those loops. This chapter teaches the compiler to use that knowledge. We build three things: a precise *reference semantics* for every ALU op, a *bounds analysis* that computes a range `[vmin, vmax]` for every integer expression, and a *symbolic simplifier*, a rule set for the rewrite engine of Chapter 2 that does high-school algebra on the IR. Together they turn `((i*8+j)//8)%4` into `i`, decide `i < 4` is always true, and pick between C's fast `/` and a slower floor-division helper. As a side effect they also fold constant programs all the way down to a value, which is what the course means by "running a program by rewriting".

What you'll build

- `exec_alu`: the single source of truth for what `ADD`, `MUL`, `FLOORDIV`, `CAST` and friends compute, bit-exact with the C we will generate (int32 wraparound, float32 rounding, floor division).
- `fold`: one rewrite rule that evaluates ALU ops on constants.
- `bounds` / `vmin` / `vmax`: interval arithmetic over the IR, memoized on each node.
- `symbolic`: about forty rewrite rules for canonical form, identities, reassociation, distribution, division and modulo, recombination and comparisons, plus a catch-all that turns any expression whose bounds pin it to one value into that value.
- `simplify`: a memoized entry point that the rest of the compiler calls thousands of times.

Where this fits. Chapter 1 built the UOp and Chapter 2 built `UPat`, `PatternMatcher` and `graph_rewrite`. This chapter is the first big *client* of the rewrite engine. Chapter 4 uses `vmin` to choose how to render division, and Chapter 6 calls `simplify` on every index it creates.

3.1 Why a compiler needs algebra

Start with the end product. Here is a kernel that copies a 4x8 array through a reshape to a flat 32-element view and back. Rangeify's output index is mathematically `i*8+j`, but syntactically it is a tangle. We render it twice, once raw and once after `simplify`:

```python
from uopc.uop import UOp, Ops, ParamArg, range_, sink
from uopc.dtype import dtypes
from uopc.symbolic import simplify
from uopc.render_c import render_c
def buf(slot, n, dtype=dtypes.f32): return UOp(Ops.BUFFER, (), ParamArg(slot, dtype, (n,)))
A, O = buf(0, 32), buf(1, 32)
i, j = range_(4, 0), range_(8, 1)
flat = i * 8 + j
idx = ((flat // 8) % 4) * 8 + flat % 8          # what a reshape (4,8)->(32,)->(4,8) produces
print(render_c(sink(O.index(flat).store(A.index(idx).load()).end(i, j))))
print(render_c(sink(O.index(flat).store(A.index(simplify(idx)).load()).end(i, j))))
```

The two function bodies (the shared prelude of helper functions is omitted from both):

```c
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    for (int ridx1 = 0; ridx1 < 8; ridx1++) {
      int alu1 = ((ridx0*8)+ridx1);
      float val2 = data0[((((alu1/8)%4)*8)+(alu1%8))];
      data1[alu1] = val2;
    }
  }
}
```

```c
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    for (int ridx1 = 0; ridx1 < 8; ridx1++) {
      int alu1 = ((ridx0*8)+ridx1);
      float val2 = data0[alu1];
      data1[alu1] = val2;
    }
  }
}
```

On a toy this is cosmetic, since gcc can strength-reduce division by 8. On real kernels the stakes are higher. Division by a non-power-of-two costs tens of cycles, and an index that *mentions* a loop variable it does not really depend on changes where the compiler places loops and accumulators. Pitfall 12 in the development notes, retold in Chapter 6, is a 250x slowdown caused by exactly that. Algebra is how the compiler finds out what an expression *means*, as opposed to how it was written.

There is a second, older reason. The week-1 milestone asks you to "run" closed programs (only `CONST`, `STACK`, `INDEX` and ALU ops) with nothing but rewriting. Chapter 2 built the `evaluate` rules that push ALU ops through `STACK` and resolve `INDEX`. Once those rules expose a scalar op on two constants,

something has to *compute* it. That something is constant folding, and it needs a precise answer to "what does `ADD` compute?"

3.2 Reference semantics: `exec_alu`

A compiler that folds `2 + 3` into `5` is making a promise: the program with the fold behaves exactly like the program without it. Without the fold, the addition happens in compiled C on int32 or float32 values. So the folder must compute what *C on this machine* would compute. Python's arithmetic is not that: Python ints are unbounded, Python floats are 64-bit, and Python's `//` floors where C's `/` truncates.

`exec_alu` is the one place where the IR's arithmetic is defined:

`uopc/symbolic.py · exec_alu`

```python
def exec_alu(op: Ops, dtype, vals: list):
  # inputs first: an f32 CONST may hold a double (16777217.0) that C rounds when it parses the literal
  if op is not Ops.CAST and dtype.is_float: vals = [truncate(dtype, v) if isinstance(v, float) else v for v in vals]
  if op is Ops.WHERE and dtype.is_float: vals = [vals[0]] + [truncate(dtype, v) for v in vals[1:]]
  match op:
    case Ops.ADD: r = vals[0] + vals[1]
    case Ops.MUL: r = vals[0] * vals[1] if not (dtype.is_float and any(math.isinf(v) for v in vals) and 0 in vals) else math.nan
    case Ops.MAX: r = max(vals[0], vals[1])
    case Ops.CMPLT: return vals[0] < vals[1]
    case Ops.CMPNE: return vals[0] != vals[1]
    case Ops.FLOORDIV:
      if vals[1] == 0 or (dtype == dtypes.i32 and vals[0] == -2**31 and vals[1] == -1): return None   # C traps here
      r = vals[0] // vals[1]
    case Ops.FLOORMOD: r = vals[0] % vals[1] if vals[1] != 0 else None
    case Ops.WHERE: r = vals[1] if vals[0] else vals[2]
    case Ops.RECIP: r = 1.0 / vals[0] if vals[0] != 0 else math.copysign(math.inf, vals[0])
    case Ops.EXP2: r = _safe(math.pow, 2.0, vals[0]) if vals[0] < 1024 else math.inf
    case Ops.LOG2: r = math.log2(vals[0]) if vals[0] > 0 else (-math.inf if vals[0] == 0 else math.nan)
    case Ops.SQRT: r = math.sqrt(vals[0]) if vals[0] >= 0 else math.nan
    case Ops.SIN: r = math.sin(vals[0]) if math.isfinite(vals[0]) else math.nan
    case Ops.CAST:
      v = vals[0]
      if dtype == dtypes.i32:
        if not math.isfinite(v): return None          # undefined behavior in C: refuse to fold
        r = int(v)                                    # C truncates toward zero
      elif dtype == dtypes.bool: return bool(v)
      else: r = float(v)
    case _: return None
  return None if r is None else truncate(dtype, r)
```

Every result passes through `truncate`, which brings a Python value into the exact value set of the dtype:

`uopc/dtype.py · truncate`

```python
def truncate(d: DType, x):
  """Bring a python value into the exact value set of dtype d (used by const folding)."""
  if d == dtypes.bool: return bool(x)
  if d == dtypes.i32: return ((int(x) + 2**31) % 2**32) - 2**31   # two's complement wraparound
  if d == dtypes.f32: return struct.unpack("f", struct.pack("f", float(x)))[0] if
math.isfinite(x) else float(x)
  if d == dtypes.f16: return struct.unpack("e", struct.pack("e", float(x)))[0] if
math.isfinite(x) and abs(x) < 65520 else float(x)
  raise TypeError(f"can't truncate to {d}")
```

Read the two functions together, op by op, and compare with C:

Op	`exec_alu`	C we generate	Notes
`ADD`, `MUL` on i32	Python op, then wrap to 32 bits	`a+b`, `a*b`	Signed overflow is formally UB in C; see Pitfalls
`ADD`, `MUL` on f32	f64 op, then round to f32	`a+b` in `float`	One rounding of the exact result, as IEEE requires
`MUL` f32 `inf*0`	`nan`	`nan`	Python's `inf*0` is also nan, but the guard makes it explicit
`FLOORDIV`, `FLOORMOD`	Python `//`, `%` (floor)	`/`, `%` or helpers	Division by zero refuses to fold (`None`)
`CMPLT`, `CMPNE`	Python `<`, `!=`	`<`, `!=`	Result is `bool`, not truncated
`WHERE`	`b if c else d`	`c?b:d`	
`RECIP`	`1/x`, signed `inf` for 0	`1.0f/x`	`1/-0.0 = -inf` in IEEE; Python would raise
`EXP2`, `LOG2`, `SQRT`, `SIN`	`math` functions with IEEE edge cases	`exp2f`, `log2f`, `sqrtf`, `sinf`	Python raises where IEEE returns inf or nan
`CAST` to i32	`int(v)`, truncation toward 0	`(int)v`	inf/nan to int is UB: refuse to fold
`CAST` to bool	`bool(v)`	`(_Bool)v`	

Three design decisions are buried in that table.

Return `None` rather than guess. `exec_alu` returns `None` for division by zero and for casting a non-finite float to int. `None` means "rule does not apply", so the folder simply leaves the op alone and the runtime will do whatever the hardware does. Folding an undefined operation into a defined value would make the program behave differently depending on whether the optimizer ran, which is the worst kind of bug to chase.

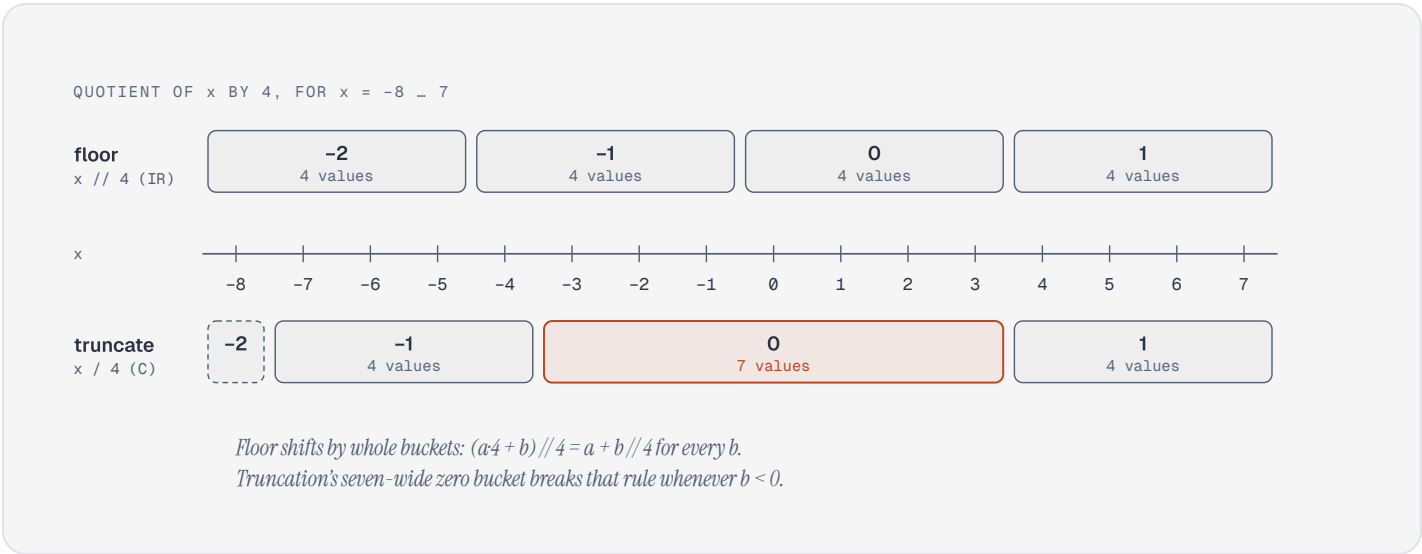
Round the result, every time. Python computes `0.1 + 0.2` in double precision. C on `float` operands computes it in single precision. `truncate(f32, x)` rounds through `struct.pack("f", x)`, which is how we get the exact float32 result. For `+`, `*` and `/`, computing in double and then rounding to single gives the correctly rounded single result: a double has more than twice the significand bits of a float (53 vs 24), so double rounding cannot happen for one operation on float inputs.

Floor, not truncation. The IR's `FLOORDIV` and `FLOORMOD` are Python's operators: the quotient rounds toward minus infinity and the remainder has the sign of the divisor. C's `/` rounds toward zero. That choice costs the renderer some work (Chapter 4 emits a helper when it cannot prove the operands are non-negative), and in exchange makes the division rules in section 3.5 true without side conditions. Here is the semantics in action:

```
>>> from uopc.uop import Ops
>>> from uopc.dtype import dtypes
>>> from uopc.symbolic import exec_alu
>>> exec_alu(Ops.FLOORDIV, dtypes.i32, [-7, 2]), exec_alu(Ops.FLOORMOD, dtypes.i32, [-7, 2])
(-4, 1)
>>> exec_alu(Ops.ADD, dtypes.i32, [2**31 - 1, 1])
-2147483648
>>> exec_alu(Ops.MUL, dtypes.i32, [65536, 65536])
0
>>> exec_alu(Ops.ADD, dtypes.f32, [0.1, 0.2]), 0.1 + 0.2
(0.30000001192092896, 0.30000000000000004)
>>> exec_alu(Ops.ADD, dtypes.f32, [16777216.0, 1.0]), 16777216.0 + 1.0
(16777216.0, 16777217.0)
>>> exec_alu(Ops.CAST, dtypes.i32, [-2.7]), exec_alu(Ops.CAST, dtypes.i32, [float("inf")])
(-2, None)
>>> exec_alu(Ops.MUL, dtypes.f32, [float("inf"), 0.0])
nan
>>> exec_alu(Ops.RECIP, dtypes.f32, [-0.0]), exec_alu(Ops.FLOORDIV, dtypes.i32, [5, 0])
(-inf, None)
>>> exec_alu(Ops.CMPLT, dtypes.bool, [3, 4])
True
```

`16777216.0` is 2^{24}, the point where float32 runs out of integers: the next float above it is `16777218.0`, so adding 1 rounds back down. Python's double happily says `16777217.0`. A folder that skipped `truncate` would compute a number the generated C program can never produce.

The picture to keep in your head for floor versus truncation is a number line cut into buckets. Floor division by 4 cuts it into buckets of exactly four integers each, everywhere. Truncating division has one oversized bucket: everything from -3 to 3 lands on 0.



Checking the semantics against the machine

A reference semantics is only worth something if it matches the hardware, so test it against the hardware. The script below (a scratch script, not part of the repo) builds a loopless kernel per op with the tools of Chapter 4, runs it on 1000 random operand pairs, and compares every output with `exec_alu`:

```python
import random, numpy as np
from uopc.uop import UOp, Ops, ParamArg, sink
from uopc.dtype import dtypes
from uopc.symbolic import exec_alu
from uopc.runtime import run_c, np_dtype

def buf(slot, n, dtype): return UOp(Ops.BUFFER, (), ParamArg(slot, dtype, (n,)))

N = 1000
random.seed(0)
cases = [(Ops.ADD, dtypes.i32), (Ops.MUL, dtypes.i32), (Ops.FLOORDIV, dtypes.i32),
         (Ops.FLOORMOD, dtypes.i32), (Ops.ADD, dtypes.f32), (Ops.MUL, dtypes.f32)]
for op, dt in cases:
  a, b = buf(0, N, dt), buf(1, N, dt)
  out = buf(2, N, dt)
  s = sink(*[out.index(i).store(a.index(i).load().alu(op, b.index(i).load())) for i in range(N)])
  if dt == dtypes.i32:
    A = np.array([random.randint(-2**31, 2**31 - 1) for _ in range(N)], np.int32)
    B = np.array([random.choice([random.randint(-9, 9) or 1, random.randint(-2**31, 2**31 - 1)])
for _ in range(N)], np.int32)
  else:
    A = np.array([random.uniform(-1e6, 1e6) for _ in range(N)], np.float32)
    B = np.array([random.uniform(-1e6, 1e6) for _ in range(N)], np.float32)
  O = np.zeros(N, np_dtype(dt))
  run_c(s, {0: A, 1: B, 2: O})
  ref = [exec_alu(op, dt, [x.item(), y.item()]) for x, y in zip(A, B)]
  bad = sum(r != o.item() for r, o in zip(ref, O))
  print(f"{op.name:9s} {dt.name}: {bad} mismatches / {N}")
```

```
ADD       i32: 0 mismatches / 1000
MUL       i32: 0 mismatches / 1000
FLOORDIV  i32: 0 mismatches / 1000
FLOORMOD  i32: 0 mismatches / 1000
ADD       f32: 0 mismatches / 1000
MUL       f32: 0 mismatches / 1000
```

The integer operands are chosen so that most additions and multiplications overflow, so this also checks the wraparound. This kind of *differential test*, where two implementations that should agree are run on random inputs, is the cheapest strong test a compiler writer has. Appendix B returns to it.

3.3 Constant folding

With `exec_alu` in hand, constant folding is one rule:

uopc/symbolic.py · _fold

```
def _fold(x: UOp) -> UOp | None:
  if x.shape != () or not all(s.op is Ops.CONST for s in x.src): return None
  r = exec_alu(x.op, x.dtype, [s.arg for s in x.src])
  return None if r is None else UOp.const(r, x.dtype)
```

uopc/symbolic.py · fold

```
fold = PatternMatcher([(UPat(ALU, name="x"), _fold)])
```

The pattern `UPat(ALU, name="x")` matches any ALU op (the `PatternMatcher` indexes it under every ALU op, see Chapter 2). The function checks that every source is a `CONST`, evaluates, and builds the result with `UOp.const(r, x.dtype)`. Because UOps are hash consed, the result *is* the canonical node for that constant, so folding `2+3` and writing `5` by hand give the very same object.

Two details matter:

- **Only scalars fold.** A shaped op like `STACK(1,2) + STACK(3,4)` is not handled here. The `evaluate` rules of Chapter 2 first push the ADD inside the STACK (`STACK(1+3, 2+4)`), and then this rule folds the scalar pieces. Keeping the folder scalar keeps it trivially correct.
- **Folding can refuse.** If `exec_alu` returns `None`, `_fold` returns `None` and the node stays. `graph_rewrite` treats `None` as "no rule fired" and moves on.

Since `graph_rewrite` is bottom-up and re-rewrites every rule result, folding the leaves of a deep constant expression produces new constants, which make their parents foldable, and so on up to the root. That is all the machinery needed to evaluate `(2+3)*(4+5)`, or the 20,000-deep chain in `test_deep_graph_no_recursion_error`.

3.4 Bounds analysis: interval arithmetic

Folding only helps when the inputs are literally constant. Most index expressions contain loop variables, which are not constant, but they are *bounded*: a `RANGE` with end 4 takes the values 0, 1, 2, 3 and nothing else. If we know a range for every subexpression we can prove facts like "this division by 8 is always 0" or "this comparison is always true".

The idea

Represent what we know about an integer expression `e` as an interval `[lo, hi]` with the guarantee `lo <= e <= hi` for every possible execution. Then define, for each operation, how intervals combine. This is *interval arithmetic*, the simplest instance of what program analysis calls *abstract interpretation*: run the program on abstract values (here, intervals) instead of concrete ones.

The one property that matters is **soundness**: the true value is always inside the interval. Tightness is nice but optional. `[-2147483648, 2147483647]` is always a correct answer for an i32 expression, it just says nothing. The rules:

- **Constant** `c`: `[c, c]`.
- **Range** with end `n`: `[0, n-1]`.
- **Addition** is monotone in both arguments, so the extremes of the sum come from the extremes of the inputs: `[a0, a1] + [b0, b1] = [a0+b0, a1+b1]`.
- **Multiplication** is not monotone when signs are mixed (`-3 * -3` is large), but it is *bilinear*, so its extremes over a box lie at the corners. Compute all four products `a0*b0, a0*b1, a1*b0, a1*b1` and take min and max.
- **Max** is monotone: `[max(a0,b0), max(a1,b1)]`.
- **Floor division by a positive constant** `c` is monotone (non-decreasing): `[a0//c, a1//c]`.
- **Floor modulo by a positive constant** `c` always lands in `[0, c-1]`. If the input already lies in `[0, c-1]` the modulo is the identity, so the input's interval is tighter and we return it.
- **Comparison** `a < b` is a boolean. Its interval is over `{False < True}`: if `a1 < b0` it is always true, `[True, True]`; if `a0 >= b1` always false; otherwise `[False, True]`, unknown.
- **WHERE** takes the true branch's interval if the condition is always true, the false branch's if always false, and the union (hull) otherwise.
- **CAST from bool** to int maps `False/True` to `0/1`.

Everything else (floats, shaped values, ops we have no rule for) gets the full range of its dtype.

The code

uopc/symbolic.py · _bounds

```
def _bounds(u: UOp) -> tuple:
  d = u.dtype
  full = (dtypes.min(d), dtypes.max(d))
  if u.op is Ops.CONST: return (u.arg, u.arg)
  if u.op is Ops.RANGE: return (0, max(0, vmax(u.src[0]) - 1))
  if d.is_float or u.shape != (): return full
  if u.op is Ops.CAST and u.src[0].dtype == dtypes.bool: return (int(vmin(u.src[0])),
int(vmax(u.src[0])))
  if u.op is Ops.WHERE:
    (c0, c1), (a0, a1), (b0, b1) = bounds(u.src[0]), bounds(u.src[1]), bounds(u.src[2])
    if c0: return (a0, a1)
    if not c1: return (b0, b1)
    return (min(a0, b0), max(a1, b1))
  if u.op not in ALU or len(u.src) != 2: return full
  (a0, a1), (b0, b1) = bounds(u.src[0]), bounds(u.src[1])
  match u.op:
    case Ops.ADD: return (a0 + b0, a1 + b1)
    case Ops.MUL:
      if any(math.isinf(x) for x in (a0, a1, b0, b1)): return full
      ps = [a0 * b0, a0 * b1, a1 * b0, a1 * b1]
      return (min(ps), max(ps))
    case Ops.MAX: return (max(a0, b0), max(a1, b1))
    case Ops.FLOORDIV if b0 == b1 and b0 > 0: return (a0 // b0, a1 // b0)
    case Ops.FLOORMOD if b0 == b1 and b0 > 0:
      return (a0, a1) if 0 <= a0 and a1 < b0 else (0, b0 - 1)   # floor mod by c>0 is always in
[0, c)
    case Ops.CMPLT: return (True, True) if a1 < b0 else (False, False) if a0 >= b1 else (False,
True)
    case Ops.CMPNE: return (True, True) if a1 < b0 or b1 < a0 else (False, False) if a0 == a1 ==
b0 == b1 else (False, True)
  return full
```

The rules above are almost line for line. Notice the special cases: `RANGE` uses `max(0, vmax(end) - 1)` so that a zero-trip loop still has a well-formed interval, and `MUL` gives up if a bound is infinite. That check is defensive: float expressions return the full range before reaching it, and `0 * inf` would be nan.

The public entry point adds two things:

uopc/symbolic.py · bounds

```python
def bounds(u: UOp) -> tuple:
  if (b := u.__dict__.get("_bounds")) is not None: return b
  b = _bounds(u)
  if u.dtype == dtypes.i32 and (b[0] < dtypes.min(u.dtype) or b[1] > dtypes.max(u.dtype)):
    b = (dtypes.min(u.dtype), dtypes.max(u.dtype))   # might wrap around: give up
  u.__dict__["_bounds"] = b
  return b
```

Memoization on the node. UOps are immutable and hash consed (see Chapter 1), so the bounds of a node never change and every structurally equal expression is the same object. The result can be cached directly on the node, in its `__dict__`. Without the cache, `bounds` of a deep expression would recompute shared subexpressions exponentially many times.

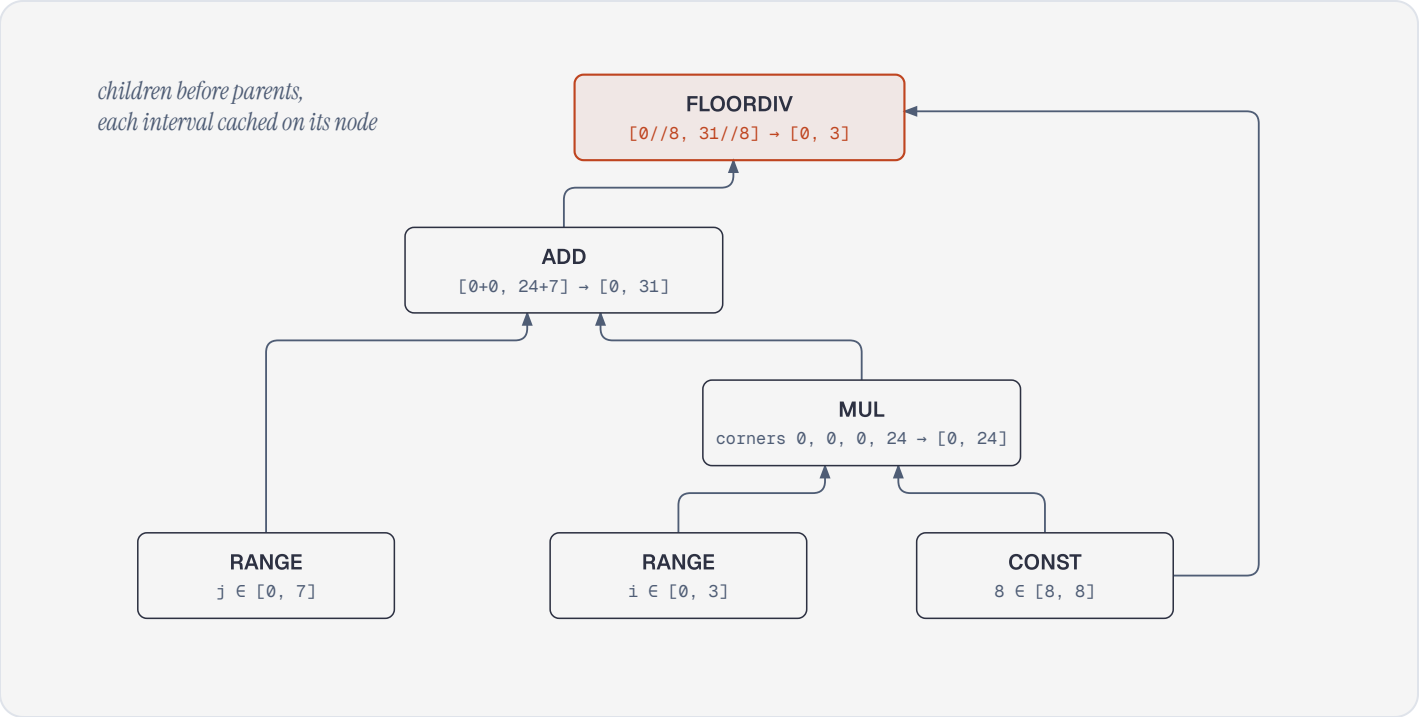
Overflow means "give up". Our intervals are computed with unbounded Python ints. If an i32 interval escapes `[-2^31, 2^31-1]`, the real computation *might* wrap around, and then the true value could be anywhere. Returning the full range is the only sound answer. This one line is why everything else in the analysis can ignore wraparound.

```
>>> from uopc.uop import range_
>>> from uopc.symbolic import vmin, vmax, bounds
>>> i, j = range_(4, 0), range_(8, 1)
>>> bounds(i), bounds(j)
((0, 3), (0, 7))
>>> bounds(i * 8), bounds(i * 8 + j)
((0, 24), (0, 31))
>>> bounds((i * 8 + j) // 8), bounds((i * 8 + j) % 8)
((0, 3), (0, 7))
>>> bounds((i + -2) * (j + -3))
(-8, 6)
>>> bounds((i + -2) // 3), bounds((i + -2) % 3)
((-1, 0), (0, 2))
>>> bounds(i < 4), bounds(i < 2)
((True, True), (False, True))
>>> bounds(i + i * -1)
(-3, 3)
```

Check one by hand. `(i-2)*(j-3)` has factors in `[-2, 1]` and `[-3, 4]`. The corners are `6, -8, -3, 4`, so `[-8, 6]`. Both extremes are attained (`i=0, j=7` and `i=0, j=0`), so the interval is exact here.

The last line shows the classic weakness, called the *dependency problem*. `i - i` is always 0, but interval arithmetic treats the two occurrences of `i` as independent variables, each anywhere in `[0, 3]`, and concludes `[-3, 3]`. The interval is sound, just loose. The cure is to simplify algebraically *first* (`x + x*-1 -> 0` is one of our rules) and ask for bounds afterwards. Rules and bounds work as a team.

The figure shows bounds flowing up the expression tree of `(i*8 + j) // 8`. Each node's interval is computed from its children's, exactly the order in which `graph_rewrite` visits them.



What bounds are used for

Bounds analysis shows up all over the compiler:

1. **Deciding comparisons.** `i < 4` with `i` in `[0, 3]` is `True`. Masks and guards that rangeify generates for padding collapse when they are always true.
2. **Deleting division and modulo.** If `0 <= x < c` then `x // c == 0` and `x % c == x`.
3. **Collapsing anything constant.** If `vmin(e) == vmax(e)`, `e` is a constant (section 3.5).
4. **Choosing C operators.** The renderer emits plain `/` and `%` only if `vmin(numerator) >= 0`, otherwise a floor-division helper (Chapter 4).
5. **Placing loops.** Bounds-driven simplification removes spurious mentions of loop variables, which changes where the linearizer (Chapter 5) puts code.

3.5 The symbolic rule set

Now the algebra. `symbolic` is `fold` plus one more `PatternMatcher`. We read it in groups. The pattern variables are module-level `UPat`s, reused by many rules:

uopc/symbolic.py:138-141

```
def _lt_mul(x: UOp, c1: UOp, c2: UOp) -> UOp | None:
  # x*c1 < c2  <=>  x < ceil(c2/c1)  for c1 > 0
  return x < -(-c2.arg // c1.arg) if c1.arg > 0 else None
```

`xi` matches only i32 values. That one restriction carries a lot of weight, as we will see: most of algebra is only true for integers.

To show what each group does we use a ten-line printer and a helper that prints an expression next to its simplified form. Neither is part of `uopc`, they are just for this chapter (`x` and `y` are i32 `PARAM`s, `f` is an f32 `PARAM`, `i` and `j` are ranges with ends 4 and 8):

```
from uopc.uop import UOp, Ops, ParamArg, range_
from uopc.dtype import dtypes
from uopc.symbolic import simplify

SYM = {Ops.ADD: "+", Ops.MUL: "*", Ops.FLOORDIV: "//", Ops.FLOORMOD: "%", Ops.CMPLT: "<",
Ops.CMPNE: "!="}
def fmt(u: UOp) -> str:
  if u.op is Ops.CONST: return repr(u.arg)
  if u.op is Ops.RANGE: return "ijklmn"[u.arg.idx]
  if u.op is Ops.PARAM: return "xyf"[u.arg.slot]
  if u.op in SYM: return f"({fmt(u.src[0])}{SYM[u.op]}{fmt(u.src[1])})"
  if u.op is Ops.MAX: return f"max({fmt(u.src[0])}, {fmt(u.src[1])})"
  if u.op is Ops.WHERE: return f"({fmt(u.src[0])} ? {fmt(u.src[1])} : {fmt(u.src[2])})"
  return f"{u.op.name.lower()}({', '.join(fmt(s) for s in u.src)})"

x = UOp(Ops.PARAM, (), ParamArg(0, dtypes.i32)); y = UOp(Ops.PARAM, (), ParamArg(1, dtypes.i32))
f = UOp(Ops.PARAM, (), ParamArg(2, dtypes.f32))
i, j = range_(4, 0), range_(8, 1)
def show(e): print(f"{fmt(e):34s} -> {fmt(simplify(e))}")
```

Canonical form: constants on the right

uopc/symbolic.py:143-146

```
def _i32(v: int) -> int: return truncate(dtypes.i32, v)   # int32 arithmetic wraps, so folded
constants must too

x, y, z = UPat.var("x"), UPat.var("y"), UPat.var("z")
c1, c2 = UPat.cvar("c1"), UPat.cvar("c2")
```

`3 + x` and `x + 3` are the same value. If both forms could occur, every rule that wants "something plus a constant" would need two patterns, and rules that match two levels deep would need four. Instead, the first rule picks a *canonical form*: in commutative ops, the constant goes on the right. Every later rule can then assume `x + c`, never `c + x`.

`UPat` can match commutative ops in any order (a `list` src tries all permutations, see Chapter 2), and the sugar `x + 0` builds exactly such a list pattern. So the canonical form is not strictly needed for correctness. It is needed for *meeting*: two constants can only fold if some rule brings them next to each other, and that is much easier when every constant is known to sit on the right. The guard `y.op is not Ops.CONST` stops the rule from swapping two constants forever (that case belongs to `fold`).

Identities

`uopc/symbolic.py:147-161`

```
ints = (dtypes.i32, dtypes.bool)
xi = UPat.var("x", dtype=dtypes.i32)

symbolic = fold + PatternMatcher([
  # canonical form: constants on the right of commutative ops
  (UPat((Ops.ADD, Ops.MUL, Ops.MAX, Ops.CMPNE), src=(UPat(Ops.CONST, name="c"), UPat(name="y")),
name="u"),
   lambda u, c, y: u.replace(src=(y, c)) if y.op is not Ops.CONST else None),
  # identities
  (x + 0, lambda x: x), (x + 0.0, lambda x: x),   # x+0.0 is not exact for x=-0.0; we accept that
  (x * 1, lambda x: x), (x * 1.0, lambda x: x),
  (UPat.var("x", dtype=ints) * 0, lambda x: x.const_like(0)),
  (x // 1, lambda x: x), (x % 1, lambda x: x.const_like(0)),
  (UPat(Ops.MAX, src=(x, x)), lambda x: x),
  (UPat(Ops.WHERE, src=(UPat(), x, x)), lambda x: x),
  (UPat(Ops.WHERE, src=(UPat.cvar("c"), x, y)), lambda c, x, y: x if c.arg else y),
```

These are the rules you learned before algebra: `x+0 = x`, `x*1 = x`, `x*0 = 0`, `x//1 = x`, `x%1 = 0`, `max(x,x) = x`, a `WHERE` with equal branches is that branch, a `WHERE` with a constant condition is one of its branches, and a `WHERE` nested inside a `WHERE` on the same condition can skip the inner test. A cast to the same dtype is a no-op. `b != False` is `b`. `x < x` and `x != x` are false *for integers*. `x + x` becomes `x*2`,

and `x + x*-1` (which is how `x - x` is spelled, since `__sub__` builds `x + (-x)` and `-x` is `x * -1`) becomes 0.

```
(3+x)                          -> (x+3)
(2*x)                          -> (x*2)
(x*1)                          -> x
(x*0)                          -> 0
(f*0.0)                        -> (f*0.0)
(f+0.0)                        -> f
(x//1)                         -> x
(x%1)                          -> 0
((x<y) ? x : x)                -> x
((x<y) ? ((x<y) ? x : 5) : y)  -> ((x<y) ? x : y)
(x+x)                          -> (x*2)
(x+(x*-1))                     -> 0
```

Look at the floats. `f*0.0` does *not* simplify, because in IEEE arithmetic `inf*0 = nan`, `nan*0 = nan` and `-1.0*0.0 = -0.0`. The rule `x*0 -> 0` is guarded by `dtype=ints`. In the same spirit, `x < x` is only false for integers: `nan < nan` is false too, but `nan != nan` is *true*, so `x != x -> False` would be wrong for floats.

`f + 0.0 -> f` *does* fire, and it is not quite exact: `-0.0 + 0.0` is `+0.0` in IEEE, so the rewrite turns a `+0.0` result into `-0.0` when `f` is `-0.0`. The comment in the source says "we accept that". Our compiler also tells gcc `-fno-signed-zeros` (Chapter 4), so the C side makes the same trade. This is a policy choice, and every ML compiler makes it. Write such choices down.

Reassociation, and why only for integers

`uopc/symbolic.py:162-165`

```
  (UPat(Ops.WHERE, src=(z, UPat(Ops.WHERE, src=(z, x, UPat())), y)), lambda z, x, y: z.where(x,
y)),
  (UPat(Ops.WHERE, src=(z, x, UPat(Ops.WHERE, src=(z, UPat(), y)))), lambda z, x, y: z.where(x,
y)),
  (UPat(Ops.CAST, src=(x,), name="c"), lambda c, x: x if c.dtype == x.dtype else None),
  (UPat(Ops.CMPNE, src=(UPat.var("x", dtype=dtypes.bool), UPat(Ops.CONST, arg=False))), lambda x:
x),
```

Pull constants outward until they meet: `(x+3)+4 -> x+7`, `(x*4)*3 -> x*12`, and `(x+3)*4 -> x*4 + 12`. For i32 these are true even with wraparound. Integers modulo 2^{32} form a ring, and associativity and distributivity hold in every ring.

Float addition is not associative. With f32 semantics from `exec_alu`:

```
>>> A = lambda a, b: exec_alu(Ops.ADD, dtypes.f32, [a, b])
>>> A(A(1.0, 1e8), -1e8), A(1.0, A(1e8, -1e8))
(0.0, 1.0)
```

Near 10^8 the spacing between float32 values is 8, so `1.0 + 1e8` rounds back to `1e8` and the 1 is lost. Reassociating would change the answer from 0 to 1. Hence `xi`, not `x`, in every reassociation rule:

```
((x+3)+4)                        -> (x+7)
((x*4)*3)                        -> (x*12)
((x+3)*4)                        -> ((x*4)+12)
((x+1)+y)                        -> ((x+y)+1)
((x+y)*4)                        -> ((x*4)+(y*4))
((x*3)+(x*5))                    -> (x*8)
((f+1.0)+2.0)                    -> ((f+1.0)+2.0)
```

(ML compilers do reassociate floats when they vectorize reductions, but they do it deliberately in one place, with a flag: see `-fassociative-math` in Chapter 10. The simplifier is not that place.)

Distribution and collecting like terms

`uopc/symbolic.py:166-170`

```
  (UPat((Ops.CMPLT, Ops.CMPNE), src=(xi, xi)), lambda x: UOp.const(False)),
  (xi + xi, lambda x: x * 2),
  (xi + xi * -1, lambda x: x.const_like(0)),
  # reassociate constants outward, so that they can meet and fold
  ((xi + c1) + c2, lambda x, c1, c2: x + _i32(c1.arg + c2.arg)),   # ints only: float addition
isn't associative
```

The first rule distributes a constant over a sum: `(a+b)*c -> a*c + b*c`. Usually distributing makes expressions *bigger*, so why do it? Because the division and modulo rules below look at the individual terms of a sum. `((x+y)*4) // 4` can only be seen as "a multiple of 4, divided by 4" after the product is spread over the terms. Development note 14 records the cost of not having this rule: GPT attention kernels kept chains like `((x/4096)%4)` in their inner loops (see Chapter 6, Pitfalls).

The second rule floats a constant out of a sum, `(x+c) + y -> (x+y) + c`, so that the constant reaches the outermost addition, where it can meet another constant. The last two collect like terms: `x*3 + x*5 -> x*8` and `x*3 + x -> x*4`.

Division and modulo

uopc/symbolic.py:171-178

```
  ((xi * c1) * c2, lambda x, c1, c2: x * _i32(c1.arg * c2.arg)),
  ((xi + c1) * c2, lambda x, c1, c2: x * c2 + _i32(c1.arg * c2.arg)),
  # distribute constants over sums so div/mod can see every term: (a + b) * c -> a*c + b*c
  (UPat(Ops.MUL, src=(UPat(Ops.ADD, src=(xi, UPat.var("y")), name="s"), UPat.cvar("c"))), lambda
x, y, s, c: x * c + y * c),
  (UPat(Ops.ADD, src=[xi + c1, UPat.var("y")]), lambda x, c1, y: (x + y) + c1 if y.op is not
Ops.CONST else None),
  (xi * c1 + xi * c2, lambda x, c1, c2: x * _i32(c1.arg + c2.arg)),
  (xi * c1 + xi, lambda x, c1: x * _i32(c1.arg + 1)),
  # division and modulo (floor semantics make these unconditional)
```

This is the heart of index simplification. The first three rules handle nested division and modulo by constants:

- `(x // c1) // c2 = x // (c1*c2)` for positive `c1, c2`. Write `x = q*c1*c2 + r` with `0 <= r < c1*c2`. Then `x // c1 = q*c2 + r // c1`, where `0 <= r // c1 < c2`, and dividing that by `c2` gives `q`.
- `(x % c1) % c2 = x % c2` when `c2` divides `c1`. Taking the remainder by `c1` subtracts a multiple of `c1`, which is a multiple of `c2`, and that does not change the remainder mod `c2`.
- `(x % c1) // c2 = (x // c2) % (c1 // c2)` when `c2` divides `c1`. This is "the middle digit": think of `x` in mixed radix with digit sizes `c2` and `c1/c2`. This one is exercise 6.

The next two call `_div_sum`, which splits a sum into terms, and the last two use bounds to delete division outright. The helpers:

uopc/symbolic.py · _divides

```
def _divides(u: UOp, c: int) -> UOp | None:
  """u / c if every term of u is a multiple of c, else None."""
  if u.op is Ops.CONST: return u.const_like(u.arg // c) if u.arg % c == 0 else None
  if u.op is Ops.MUL and u.src[1].op is Ops.CONST:
    return u.src[0] * (u.src[1].arg // c) if u.src[1].arg % c == 0 else None
  if u.op is Ops.ADD:
    if (a := _divides(u.src[0], c)) is not None and (b := _divides(u.src[1], c)) is not None:
return a + b
  return None
```

uopc/symbolic.py · _split_terms

```python
def _split_terms(u: UOp) -> list[UOp]:
  return _split_terms(u.src[0]) + _split_terms(u.src[1]) if u.op is Ops.ADD else [u]
```

uopc/symbolic.py · _div_sum

```python
def _div_sum(x: UOp, c: UOp, mod: bool) -> UOp | None:
  """(a*c + b) // c == a + b//c  and  (a*c + b) % c == b % c. Always true for *floor*
division."""
  if c.arg <= 0: return None
  terms = _split_terms(x)
  quot, rest = [], []
  for t in terms: (quot if _divides(t, c.arg) is not None else rest).append(t)
  if not quot: return None
  r = sum(rest[1:], rest[0]) if rest else x.const_like(0)
  if mod: return r % c
  return sum((_divides(t, c.arg) for t in quot[1:]), _divides(quot[0], c.arg)) + r // c
```

`_divides(u, c)` returns `u/c` if every term of `u` is visibly a multiple of `c` (a constant multiple, or a product with a constant multiple), and `None` otherwise. `_div_sum` partitions the terms of `x` into those divisible by `c` (`quot`) and the rest. The rules it implements are:

```
(a*c + b) // c  =  a + b // c
(a*c + b) %  c  =  b % c
```

Why these are always true for floor division. Floor division is defined by `x = (x // c)*c + x % c` with `0 <= x % c < c` (for `c > 0`). Let `q = b // c` and `r = b % c`, so `b = q*c + r` with `0 <= r < c`. Then `a*c + b = (a + q)*c + r`, and since `0 <= r < c`, uniqueness of floor division says `(a*c + b) // c = a + q` and `(a*c + b) % c = r`. Nothing was assumed about the sign of `a` or `b`.

Why they are false for truncating division. Take `a = 1, b = -1, c = 4`. Then `a*c + b = 3`, and `3 / 4 = 0` in C. But `a + b/c = 1 + (-1/4) = 1 + 0 = 1`, because C truncates `-0.25` to `0`. The rule is only true for truncation when `b >= 0`, so a compiler with C semantics would need a bounds check before every application. Look at the number-line figure again: truncation's oversized bucket around zero is exactly what breaks "shift by a whole bucket". This is development note 4: floor semantics make the IR's algebra *unconditional*, and the renderer pays a small price instead.

```
((x*4)//4)                        -> x
(((x*6)+4)%2)                     -> 0
(((x*6)+4)//2)                    -> ((x*3)+2)
(((x*4)+y)//4)                    -> (x+(y//4))
(((x*4)+y)%4)                     -> (y%4)
(((i*8)+j)//2)                    -> ((i*4)+(j//2))
((x//4)//2)                       -> (x//8)
((x%12)%4)                        -> (x%4)
((x%12)//4)                       -> ((x//4)%3)
(j//8)                            -> 0
(j%8)                             -> j
```

The last two lines are the bounds-based rules: `j` lies in `[0, 7]`, so `j // 8 = 0` and `j % 8 = j`. Unlike `_div_sum`, these need bounds, and they are what makes index simplification work on loop variables.

Recombination

`uopc/symbolic.py:179-181`

```
  ((xi // c1) // c2, lambda x, c1, c2: x // (c1.arg * c2.arg) if c1.arg > 0 and c2.arg > 0 and
c1.arg * c2.arg < 2**31 else None),
  ((xi % c1) % c2, lambda x, c1, c2: x % c2 if c2.arg > 0 and c1.arg % c2.arg == 0 else None),
  ((xi % c1) // c2, lambda x, c1, c2: (x // c2) % (c1.arg // c2.arg) if c2.arg > 0 and c1.arg %
c2.arg == 0 else None),
```

The definition of floor division, read backwards: `(x//c)*c + x%c = x`. Unraveling a flat index into digits and raveling it back produces exactly this shape, so it appears whenever a reshape round-trips. The second rule handles the same pattern with an extra term on the left. The `list` src means the two addends may come in either order.

```
(((x//4)*4)+(x%4))                -> x
(((((i*8)+j)//8)*8)+(((i*8)+j)%8)) -> ((i*8)+j)
```

Comparisons

`uopc/symbolic.py:182-184`

```
  (UPat(Ops.FLOORDIV, src=(xi, UPat.cvar("c"))), lambda x, c: _div_sum(x, c, mod=False)),
  (UPat(Ops.FLOORMOD, src=(xi, UPat.cvar("c"))), lambda x, c: _div_sum(x, c, mod=True)),
  (UPat(Ops.FLOORDIV, src=(xi, UPat.cvar("c"))), lambda x, c: x.const_like(0) if 0 <= vmin(x) and
vmax(x) < c.arg else None),
```

uopc/symbolic.py · _lt_mul

```
def _lt_mul(x: UOp, c1: UOp, c2: UOp) -> UOp | None:
  # x*c1 < c2  <=>  x < ceil(c2/c1)  for c1 > 0
  return x < -(-c2.arg // c1.arg) if c1.arg > 0 else None
```

Move constants across a comparison: `x + c1 < c2` is `x < c2 - c1` (true over the integers; we are again trusting there is no overflow). For multiplication, `x*c1 < c2` with `c1 > 0` is `x < ceil(c2/c1)`. Proof: `x*c1 < c2` holds iff `x < c2/c1` over the reals, and for an integer `x` that is iff `x < ceil(c2/c1)`. The expression `-(-c2 // c1)` is the standard ceiling-division idiom with floor division. Negative `c1` would flip the inequality, so `_lt_mul` declines.

```
((x+3)<10)                        -> (x<7)
((x*4)<10)                        -> (x<3)
((x*-4)<10)                       -> ((x*-4)<10)
(i<4)                             -> True
((i+j)<11)                        -> True
(i!=5)                            -> True
cast((i<5))                       -> 1
```

The last four did not come from comparison rules at all. They came from the final rule.

The catch-all: `_const_bounds`

uopc/symbolic.py:185-187

```
  (UPat(Ops.FLOORMOD, src=(xi, UPat.cvar("c"))), lambda x, c: x if 0 <= vmin(x) and vmax(x) <
c.arg else None),
  # recombine: (x//c)*c + x%c == x
  (UPat(Ops.ADD, src=[(xi // c1) * c1, xi % c1]), lambda x, c1: x),
```

uopc/symbolic.py · _const_bounds

```python
def _const_bounds(x: UOp) -> UOp | None:
  # an int/bool expression whose bounds pin it to one value *is* that value
  if x.shape == () and not x.dtype.is_float and x.op in ALU and (b := bounds(x))[0] == b[1]:
    return UOp.const(b[0], x.dtype)
  return None
```

Any integer or boolean scalar ALU expression whose interval has collapsed to a point *is* that point. `i + j < 11` has `i + j` in `[0, 10]`, so the comparison is `[True, True]`, so it is `True`. This one rule subsumes dozens of special cases. It also makes the analysis compose: once a subexpression becomes a constant, the rules above it can fire.

It is last on purpose. `PatternMatcher.rewrite` tries rules in list order and returns the first result that is not `None`. Cheap structural rules go first, and the comparatively expensive bounds query only runs when nothing else applied.

A rewrite, step by step

How do the rules cooperate? Wrap the matcher to print every rule that fires (a scratch subclass, not in the repo) and simplify `((i*8 + j)//8) % 4`:

```python
from uopc.rewrite import PatternMatcher, graph_rewrite
from uopc.symbolic import symbolic

class Traced(PatternMatcher):
  def rewrite(self, u, ctx=None):
    r = super().rewrite(u, ctx)
    if r is not None: print(f"  {fmt(u):28s} => {fmt(r)}")
    return r

e = ((i * 8 + j) // 8) % 4
print(fmt(e))
print("=", fmt(graph_rewrite(e, Traced(symbolic.rules))))
```

```
((((i*8)+j)//8)%4)
  (((i*8)+j)//8)               => ((i*1)+(j//8))
  (i*1)                        => i
  (j//8)                       => 0
  (i+0)                        => i
  (i%4)                        => i
= i
```

Bottom-up, `i*8+j` has nothing to do. At `//8`, `_div_sum` sees the term `i*8` divisible by 8 and the remainder `j`, and produces `i*1 + j//8`. That result is itself rewritten (that is the "re-rewrite" step of `graph_rewrite`): `i*1 -> i` by an identity, `j//8 -> 0` by bounds, `i+0 -> i`. Finally `i % 4 -> i` because `i` lies in `[0, 3]`. No single rule knows how to simplify the whole expression, but five small rules, each obviously correct, do it together.

The same machinery inverts any mixed-radix ravel. Take a `(2, 3, 4)` array with indices `i, j, k`, ravel them into a flat index, and pull each digit back out, then reshape the array to `(4, 6)`:

```
i, j, k = range_(2, 0), range_(3, 1), range_(4, 2)
flat = (i * 3 + j) * 4 + k
for name, e in [("flat", flat), ("k", flat % 4), ("j", (flat // 4) % 3), ("i", flat // 12),
                ("row of (4,6)", flat // 6), ("col of (4,6)", flat % 6)]:
  print(f"{name:12s} {fmt(e):30s} -> {fmt(simplify(e))}")
```

```
flat         ((((i*3)+j)*4)+k)              -> (((i*12)+(j*4))+k)
k            (((((i*3)+j)*4)+k)%4)          -> k
j            ((((((i*3)+j)*4)+k)//4)%3)     -> j
i            (((((i*3)+j)*4)+k)//12)        -> i
row of (4,6) (((((i*3)+j)*4)+k)//6)         -> ((i*2)+(((j*4)+k)//6))
col of (4,6) (((((i*3)+j)*4)+k)%6)          -> (((j*4)+k)%6)
```

The digits come back exactly. The reshape gives the exact answer, `i*2 + (j*4+k)//6` for the row and `(j*4+k)%6` for the column, which no rule should simplify further since `j*4+k` really does straddle rows of 6.

3.6 Termination and confluence

A term rewriting system (Chapter 2) has two classic worries: does it stop (*termination*), and does the answer depend on which rule fires first (*confluence*)?

Termination. Most of our rules shrink the expression or keep its size and push constants rightward and outward. You can make this precise with a *measure*, a number that strictly decreases with every rewrite: node count first, then the number of constants that are not in canonical position. The distribution rule is the troublemaker, since `(a+b)*c -> a*c + b*c` grows the term. It still terminates because it pushes multiplications strictly *below* additions, and no rule pushes them back up.

Add the reverse rule, `a*c + b*c -> (a+b)*c` ("factoring"), and that argument dies:

```
xi, yi, c = UPat.var("x", dtype=dtypes.i32), UPat.var("y", dtype=dtypes.i32), UPat.cvar("c")
factor = PatternMatcher([(UPat(Ops.ADD, src=[xi * c, yi * c]), lambda x, y, c: (x + y) * c)])
graph_rewrite(x * 4 + y * 4, symbolic + factor, max_steps=100_000)
```

```
RuntimeError: graph_rewrite did not converge (a rule loops?)
```

Distribution and factoring undo each other forever. `graph_rewrite` counts steps and raises instead of hanging, which turns a silent infinite loop into a clear error. When you add a rule, ask what it makes smaller.

Confluence. Our system is not confluent in general. Different rule orders can reach different normal forms that are equal as functions but not as trees, for instance `x*4 + y*4` and `(x+y)*4` if both forms could be reached. We do not need confluence for *correctness*, since every rule is an equivalence, so any normal form computes the right value. We need it for *quality* and for caching: hash consing and the kernel cache of Chapter 7 work best when equal expressions end up as the same tree. The canonical-form rule and the fixed rule order buy most of that in practice.

3.7 Memoized `simplify`

The rest of the compiler does not call `graph_rewrite(u, symbolic)` directly. It calls `simplify`:

uopc/symbolic.py · simplify

```
def simplify(u: UOp) -> UOp:
  """graph_rewrite(u, symbolic), memoized: rewriting is a pure function of a (hash-consed)
node."""
  if (r := _simplified.get(u)) is not None: return r
  if len(_simplified) > 200_000: _simplified.clear()
  r = _simplified[u] = graph_rewrite(u, symbolic)
  _simplified[r] = r
  return r
```

Rewriting is a pure function of the input node, and nodes are hash consed, so the result can be cached in a dict keyed by node. The result also maps to itself, since a normal form simplifies to itself. The cache is cleared wholesale when it grows past 200,000 entries, a crude but adequate bound on memory.

Why bother? Rangeify simplifies every index it builds, and the same subexpressions recur across thousands of nodes and many kernels. Development note 22 records that compiling the GPT graph for 12 steps took 163 s before two fixes, identity-based `__eq__` / `__hash__` and memoizing `simplify`, and 47 s after. A micro-benchmark: ravel six loop indices into one flat index, unravel it again, and combine the digits into a 46-node expression.

```
import time
from uopc.uop import range_
from uopc.symbolic import simplify
from uopc.rewrite import graph_rewrite
from uopc.symbolic import symbolic
rs = [range_(n, i) for i, n in enumerate((2, 3, 4, 5, 6, 7))]
flat = 0
for r, n in zip(rs, (2, 3, 4, 5, 6, 7)): flat = flat * n + r       # ravel 6 indices
idxs = []
for n in (7, 6, 5, 4, 3, 2): idxs.append(flat % n); flat = flat // n   # unravel again
e = idxs[0]
for t in idxs[1:]: e = e * 10 + t
for name, f in [("graph_rewrite", lambda: graph_rewrite(e, symbolic)), ("simplify #1", lambda:
simplify(e)), ("simplify #2", lambda: simplify(e))]:
  t0 = time.perf_counter(); r = f(); print(f"{name:14s} {(time.perf_counter()-t0)*1e3:8.3f} ms
nodes {len(e.toposort())} -> {len(r.toposort())}")
```

```
graph_rewrite     8.400 ms  nodes 46 -> 27
simplify #1       7.329 ms  nodes 46 -> 27
simplify #2       0.003 ms  nodes 46 -> 27
```

(The first call to `simplify` is a little faster than the raw `graph_rewrite` before it only because the bounds caches on the nodes are already warm.)

3.8 Testing an algebra

Every rule above is "obviously correct", and so was every buggy rule ever written. The good test for a simplifier is a *property test*: generate random expressions, simplify them, and check on random inputs that the simplified expression computes the same value. While we are at it, check that `bounds` is sound, that is, every concrete value lies inside the computed interval. A scratch version:

```python
import random, time
from uopc.uop import UOp, Ops, range_
from uopc.symbolic import simplify, vmin, vmax

def ev(u, env):
  if u.op is Ops.CONST: return u.arg
  if u.op is Ops.RANGE: return env[u]
  a, b = (ev(s, env) for s in u.src)
  return {Ops.ADD: a + b, Ops.MUL: a * b, Ops.FLOORDIV: a // b if b else 0, Ops.FLOORMOD: a % b
if b else 0,
          Ops.CMPLT: a < b, Ops.MAX: max(a, b)}[u.op]

rs = [range_(n, i) for i, n in enumerate((3, 5, 8, 16))]
def gen(d):
  if d == 0 or random.random() < 0.25: return random.choice(rs) if random.random() < 0.6 else
UOp.const(random.randint(-6, 6))
  op = random.choice(["+", "*", "//", "%", "+", "*"])
  a = gen(d - 1)
  if op in ("//", "%"): return a // random.randint(1, 9) if op == "//" else a % random.randint(1,
9)
  b = gen(d - 1) if op == "+" else UOp.const(random.randint(-4, 6))
  return a + b if op == "+" else a * b

random.seed(1)
n_exprs, wrong, unsound, smaller = 3000, 0, 0, 0
t0 = time.perf_counter()
for _ in range(n_exprs):
  e = gen(5); s = simplify(e)
  smaller += len(s.toposort()) < len(e.toposort())
  for _ in range(20):
    env = {r: random.randrange(r.src[0].arg) for r in rs}
    v = ev(e, env)
    wrong += v != ev(s, env)
    unsound += not (vmin(e) <= v <= vmax(e))
print(f"{n_exprs} random expressions, {n_exprs*20} evaluations: {wrong} wrong, {unsound} out of
bounds, "
      f"{smaller} got smaller ({time.perf_counter()-t0:.1f} s)")
```

```
3000 random expressions, 60000 evaluations: 0 wrong, 0 out of bounds, 1880 got smaller (2.7 s)
```

Three seconds of runtime checks every rule in combination with every other rule, which no hand-written test can match. Note what the generator *avoids*: divisors are positive constants and values stay small, so nothing overflows. That is the domain the rules are designed for (see Pitfalls). A generator that produced huge constants would find counterexamples, and that is a feature: it tells you exactly where your assumptions are. Exercise 5 asks you to build a stronger version.

3.9 Relation to computer algebra systems

What we built is a tiny *computer algebra system* (CAS). Real ones such as Mathematica, SymPy and Maxima face the same questions on a larger scale:

- **Normal forms.** A CAS tries to put expressions in a canonical form such that two expressions are equal iff their normal forms are identical. For polynomials over the integers this is achievable: expand everything and sort the monomials. Once you add floor division and modulo, no simple complete normal form exists. Our rules are a *heuristic* simplifier: sound, but not complete.
- **Decision procedures.** Deciding "is this expression always equal to that one" for integer arithmetic with `+`, constant multiplication, comparisons and division by constants is *Presburger arithmetic*, which is decidable but expensive. SMT solvers such as Z3 decide it routinely. Polyhedral compilers (isl, used in Polly and PPCG) use exactly this theory for loop analysis. We trade completeness for speed: `simplify` must run thousands of times per compile.
- **Incompleteness, concretely.** Here are true identities our rules miss:

```
((x*2)//4)                          -> ((x*2)//4)
max(i, 0)                           -> max(i, 0)
((x+1)*(x+1))                       -> ((x+1)*(x+1))
```

`(x*2)//4` is `x//2` (divide numerator and divisor by their gcd). `max(i, 0)` is `i` since `vmin(i) = 0`. `(x+1)*(x+1)` is not wrong as written, but there is no product-of-sums expansion. Each is a reasonable rule to add if a real kernel needs it (exercises 2 and 7). The discipline is: add rules when a real kernel needs them, test each with the property test, and keep the set small enough to reason about termination.

tinygrad's `symbolic.py` is the industrial-strength version of this file and a good next read. It has the same structure (folding, bounds, div/mod rules, a `_min_max` per node) and many more rules.

3.10 The payoff in rangeify

The chain from this chapter to performance is direct. Chapter 6 lowers movement ops by pushing index expressions through them. A `RESHAPE` from `(N, N)` to `(N*N,)` and back produces `((i*N + j)//N) % N` for the row, which is `i` after `simplify`, but syntactically it mentions `j`, and the reduce lowering places an

accumulator in the loops its index *mentions*. Without simplification, fused attention at N=512 took 1787 ms instead of 7 ms (development note 12; see Chapter 6, Pitfalls, and Chapter 11).

> **Key idea:** a simplifier is not about pretty output. It decides what an expression *depends on*, and dependencies decide loop structure. One wrong dependency can cost a factor of N.

Pitfalls

Floor versus truncation (note 4). The IR's `FLOORDIV` floors and C's `/` truncates. If your renderer emits `/` for every `FLOORDIV`, everything works until an index goes negative (a convolution with padding, a shifted window). Then you read the wrong element, silently. The fix is in the renderer: `/` only when `vmin(numerator) >= 0` and the divisor is a positive constant, otherwise `floordiv_i32`. The test `test_floordiv_rendering_negative` exists for this.

Folding in the wrong precision (note 5). Computing `16777216.0 + 1.0` in Python's double gives `16777217.0`, a value no float32 program can produce. Every folded result must be rounded to its dtype.

Rounding the result but not the inputs. A sharper version of the same bug survived in `uopc` until a chapter author found it while writing this section. `UOp.const(16777217.0)` stores the double `16777217.0` as the arg of an f32 constant. C rounds the literal `16777217.0f` to `16777216.0f` at compile time, but the first `exec_alu` added the unrounded double. With that version:

```
out = UOp(Ops.BUFFER, (), ParamArg(0, dtypes.f32, (1,)))
e = UOp.const(16777217.0) + UOp.const(2.0)
print("folded:", simplify(e).arg)
O = np.zeros(1, np.float32); run_c(sink(out.index(0).store(e)), {0: O}); print("C:", O[0])
```

```
folded: 16777220.0
C: 1.6777218e+07
```

The folder said `16777217 + 2 = 16777220` (round the exact `16777219` to even), while the C program computes `16777216 + 2 = 16777218`. The fix is one line: `truncate` the inputs as well as the output. The current `exec_alu` does exactly that (its first line), so the same snippet now prints `folded: 16777218.0`,

and `test_folding_matches_c_semantics` keeps it that way. (Rounding constants at creation would also work, but would change every float literal the renderer prints.)

Casting inf or nan to int (note 6). In C this is undefined behavior. A folder that "helpfully" turns `(int)INFINITY` into `INT_MAX`, or into whatever Python's `int()` raises, makes your program's behavior depend on whether optimization ran. `exec_alu` returns `None` and leaves the cast to runtime.

Float reassociation (note 7). `(x+1.0)+2.0 -> x+3.0` looks harmless and is not exact. Every reassociation and distribution rule in `symbolic` is restricted to `xi` (i32). The one float identity we keep, `x+0.0 -> x`, is wrong for `x = -0.0`, and the code says so in a comment.

Distribution is load-bearing (note 14). Without `(a+b)*c -> a*c + b*c`, `_div_sum` cannot see the terms of a product of a sum, and `((x+y)*4096 + z) // 4096` style chains survive into inner loops.

Constants outside the dtype. The folder wraps results to 32 bits, but the first reassociation rules built new constants with plain Python arithmetic. `(x*65536)*65536` became `x*4294967296`, an i32 `CONST` whose value does not fit in an i32. The C literal `4294967296` silently has type `long`, so the C arithmetic was done in 64 bits. Index expressions never get near this range, which is why nothing had broken. The fix wraps every constant the rules create with `_i32` (a `truncate` to int32): because int32 arithmetic is arithmetic mod 2^{32}, `x*65536*65536` *is* `x*0`, and `simplify` now returns `0`.

The rules assume no overflow. `(a*c + b)//c = a + b//c` is a theorem about integers, not about int32. If `a*c` wraps, the two sides differ. `bounds` gives up on intervals that might wrap, but `_div_sum` does not consult bounds. Index arithmetic in this course stays far below 2^{31}, and GPU compilers make the same assumption. Know that you are making it.

`INT_MIN // -1`. `exec_alu` computes `-2147483648 // -1 = 2147483648`, which wraps to `-2147483648`. In C on x86 the same division traps:

```
#include <stdio.h>
int main(){ volatile int a=-2147483647-1, b=-1; printf("%d\n", a/b); }
```

```
$ bash -c 'gcc -O2 m.c -o m && ./m'
/bin/bash: line 1: 1062784 Floating point exception   (core dumped) bash -c 'gcc -O2 m.c -o m &&
./m'
```

Signed overflow and this case are undefined behavior in C, which `exec_alu` models as wraparound. The index arithmetic the compiler generates never divides by a negative constant, so it never hits this. Still, a

folder must never be *more defined* than the target: `exec_alu` now returns `None` for this one case and leaves the trap to run time.

Recompute on every call (note 22). Calling `graph_rewrite(u, symbolic)` directly from rangeify recomputes the same simplifications for every kernel. Memoize on the hash-consed node.

UPat `arg` matching uses `==`. The pattern `x + 0` stores `arg=0`, and `UPat.match` compares with `u.arg != self.arg`. Since `0 == 0.0 == False` in Python, `x + 0` also matches `x + 0.0` and `x + False`. It is harmless for these identities (all three are identities), but it is the same trap as hash-consing collision note 1 in Chapter 1. If you write a rule whose meaning depends on the constant's type, also match on `dtype`.

Exercises

1. ★ **Predict the output.** Without running anything, write down `simplify(e)` for each `e`, with `x` an i32 `PARAM` and `i = range_(4, 0)`, `j = range_(8, 1)`: (a) `(x*6 + 4)//2`, (b) `(x % 12)//4`, (c) `((i*8 + j)//2) % 4`, (d) `(i*8 + j) < 32`, (e) `(x + 5)*3 + -15`. *Hint: apply `_div_sum` by hand; use bounds for the last steps.*

2. ★ **Add a rule.** `max(i, 0)` is `i` when `vmin(i) >= 0`, but `symbolic` leaves it alone. Write a `PatternMatcher` with one rule for `MAX(x, CONST c)` that returns `x` when `vmin(x) >= c`, and check that `graph_rewrite(i.maximum(0), symbolic + yours)` returns `i`.

3. ★★ **Tighter bounds.** `_bounds` returns the full range for `FLOORDIV` by a negative constant. Derive the interval for `[a0, a1] // c` with `c < 0` (is the function increasing or decreasing?) and for `MAX` with a `WHERE` operand. Add both and run the property test of section 3.8 to check soundness.

4. ★★ **Find the bugs.** In your *own* compiler (the reference one has been fixed, see Pitfalls): (a) run `simplify((x*65536)*65536)` and look at the constant's value. What C code would it produce? (b) Fold `UOp.const(16777217.0) + UOp.const(2.0)` and compare with the C program that computes it. Fix both with `truncate`, and add a regression test for each (compare `test_folding_matches_c_semantics`). *Hint: every place that builds a `CONST` from arithmetic on `.arg` values is suspect.*

5. ★★ **Property-test your simplifier.** Extend the fuzzer of section 3.8 to generate `WHERE`, `CMPLT`, `MAX` and negative divisors, and to check that `simplify(simplify(e)) is simplify(e)` (normal forms are fixed points). Run it on your own week-1 implementation.

6. ★★ **Prove it.** Show that `(x % c1) // c2 = (x // c2) % (c1 // c2)` for every integer `x` when `c2 > 0` and `c2` divides `c1`. *Hint: write* `x = q*c1 + r` *and* `r = s*c2 + t`.

7. ★★★ **GCD division.** Generalize `_div_sum` so that `(x*2)//4 -> x//2` and `(x*6 + 4)//8 -> (x*3 + 2)//4`. *Hint: divide every term and the divisor by the gcd of all the constant factors. When is this valid for floor division? (Always, if every term is a multiple of the gcd.)*

8. ★★ **Floats.** For each of `x*0.0`, `x - x`, `x < x`, `x != x` and `(x + 1.0) - 1.0` (with `x` an f32), give a value of `x` for which the "obvious" integer simplification is wrong.

9. ★★★ **Termination measure.** Define a measure on expressions that strictly decreases under every rule in `symbolic` except canonicalization, and argue separately that canonicalization can fire at most once per node. Then explain exactly why adding factoring (section 3.6) breaks your measure.

10. ★ **Fold the edge cases.** For each input, predict what `exec_alu` returns, then run it: `(Ops.RECIP, f32, [0.0])`, `(Ops.LOG2, f32, [0.0])`, `(Ops.SQRT, f32, [-1.0])`, `(Ops.CAST, bool, [0.5])`, `(Ops.FLOORMOD, i32, [7, -2])`.

Solutions to ★ exercises

1. (a) `(x*3)+2`: both terms are divisible by 2. (b) `(x//4)%3`, the middle digit rule. (c) `((i*8)+j)//2` becomes `(i*4)+(j//2)`, then `% 4` keeps the remainder of the sum mod 4. `i*4` is a multiple of 4, so it is dropped and `(j//2)%4` remains. `j//2` lies in `[0, 3]`, so the modulo is the identity: the answer is `j//2`. (d) `True`, since `i*8+j` lies in `[0, 31]`. (e) `(x+5)*3` becomes `x*3 + 15`, then `+ -15` reassociates to `x*3 + 0`, then `x*3`.

2.

```
>>> from uopc.rewrite import UPat, PatternMatcher, graph_rewrite
>>> from uopc.symbolic import symbolic, vmin
>>> mx = PatternMatcher([(UPat(Ops.MAX, src=(UPat.var("x"), UPat.cvar("c"))), lambda x, c: x if
vmin(x) >= c.arg else None)])
>>> fmt(graph_rewrite(i.maximum(0), symbolic + mx)), fmt(graph_rewrite((i + -2).maximum(0),
symbolic + mx))
('i', 'max((i+-2), 0)')
```

The second expression, `max(i-2, 0)`, is correctly left alone: `i-2` can be negative.

10. `inf` (IEEE: `1/+0 = +inf`), `-inf`, `nan`, `True`, and `-1` (floor modulo takes the sign of the divisor: `7 = -4*-2 + -1`).

Quiz

1. Why does `exec_alu` return `None` for `CAST` of `inf` to i32 instead of some fixed value?
2. Give integers `a, b, c` for which `(a*c + b)/c != a + b/c` with C's truncating division.
3. What interval does `bounds` compute for `(i + -2) * (i + -2)` with `i = range_(4, 0)`? Is it tight? Why not?
4. Why is `x*0 -> 0` restricted to integer dtypes?
5. Why is `_const_bounds` the *last* rule in `symbolic`?
6. What does `bounds` return for an i32 expression whose computed interval exceeds 2^{31}-1, and why is that the only sound choice?
7. `simplify` caches results in a dict keyed by node. What property of UOps makes this correct?
8. Name two places outside `symbolic.py` where `vmin` / `vmax` change what code the compiler emits.

Answers

1. Casting a non-finite float to int is undefined behavior in C. Any fixed value would make the folded program disagree with the unfolded one on some machine. Refusing keeps both identical.
2. `a = 1, b = -1, c = 4`: `3/4 = 0` but `1 + (-1/4) = 1 + 0 = 1`.
3. The factor is in `[-2, 1]`, and the corners give `4, -2, -2, 1`, so `[-2, 4]`. The true range of a square is `[0, 4]`. The dependency problem: interval arithmetic treats the two factors as independent.
4. Floats: `inf*0` and `nan*0` are nan, and `-1.0*0.0` is `-0.0`.
5. It is the most expensive rule (it queries bounds) and the most general. Cheaper structural rules should get the first chance, and `PatternMatcher` returns the first rule that fires.
6. The full i32 range. The real computation may wrap around, after which the value can be anything in the type, so no tighter interval is guaranteed.
7. Hash consing makes structurally equal nodes the same object, and nodes are immutable. So the simplification of a node is a pure function of its identity.
8. The C renderer (`/` versus `floordiv_i32`, Chapter 4), and rangeify / the linearizer, through simplified indices that no longer mention spurious loop variables (Chapters 5 and 6).

Further reading

- tinygrad, `tinygrad/uop/symbolic.py` and `tinygrad/uop/ops.py` (`vmin` / `vmax`, `_min_max`): the production version of everything in this chapter.
- Ramon E. Moore, *Interval Analysis* (1966), and Moore, Kearfott and Cloud, *Introduction to Interval Analysis* (SIAM, 2009).
- Patrick Cousot and Radhia Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", POPL 1977.
- Franz Baader and Tobias Nipkow, *Term Rewriting and All That* (Cambridge, 1998): termination orders, confluence, critical pairs.
- David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", ACM Computing Surveys, 1991.
- Raymond T. Boute, "The Euclidean Definition of the Functions div and mod", ACM TOPLAS, 1992: floor, truncation and Euclidean division compared.
- Henry S. Warren, *Hacker's Delight*, 2nd ed., chapter 9 (integer division) and chapter 10 (division by constants).
- Leonardo de Moura and Nikolaj Bjørner, "Z3: An Efficient SMT Solver", TACAS 2008. Try proving your rewrite rules with Z3's Python API.

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

```
uopc/symbolic.py · 302 lines

tests/test_w01_uops.py · 128 lines
```

4. Memory, rendering C, and the runtime

Everything so far has been a *value*: constants, stacks of constants, arithmetic on them. Values are easy to reason about because they never change. A real kernel is not like that. It receives pointers to arrays that someone else allocated, reads from them, writes results into other arrays, and maybe keeps a scratch variable that it overwrites as it goes. This chapter adds memory to the IR, which means facing the one question a graph IR cannot dodge: *in what order do things happen?* Then we cross the boundary to the real machine. We print the graph as a C function, compile it with gcc into a shared library, load it into the Python process, and call it on numpy arrays. At the end of the chapter you have a working compiler for loopless programs. The next chapter adds loops.

What you'll build

- Six memory ops: `BUFFER`, `ALLOC`, `INDEX` as an address, `LOAD`, `STORE`, `AFTER`, plus `SINK` as the root of a program with side effects.
- The `addrspace` recursive property, which turns "you can't store to a value" into a type error.
- `CRenderer` (`uopc/render_c.py`): one UOp becomes at most one line of C, with single-use pure expressions inlined, constants printed exactly, and floor division rendered correctly.
- The runtime (`uopc/runtime.py`): gcc with the right flags, a content-addressed cache of `.so` files, and calls through `ctypes` with numpy buffers.

Where this fits. Chapter 3 gave us `vmin`, which the renderer uses to decide how to print division. Chapter 5 adds `RANGE` / `END` and the full linearizer. This chapter only needs the loopless special case of linearization, which is a topological sort.

4.1 Values versus memory

Recall the two promises a UOp makes (Chapter 1). It is *immutable*: its op, sources and arg never change. And it is *hash consed*: building the same structure twice gives the same object. Those promises are what make `simplify` memoizable and common-subexpression elimination free. They describe *values*, things like `3`, `x+1` and `a[i]*2`. In compiler terms the graph is in *SSA form* (static single assignment): every value is defined exactly once, by the node that computes it, and every use refers to that definition directly.

Memory breaks both promises. An array has *identity*: two arrays with the same contents are still different arrays. It also has *time*: `tmp[0]` before a store and `tmp[0]` after it are different values, even though the

expression that reads them is spelled the same. A pure DAG has neither notion. It has no "before", only "uses".

You cannot avoid memory, since kernels exist to read and write arrays. So the IR keeps two worlds and makes the crossing between them explicit:

- **Values** live in registers (conceptually). `addrspace` is `ALU`.
- **Memory** is a place that values can be loaded from and stored into. `addrspace` is `GLOBAL` (arrays passed in by the caller), `LOCAL` (scratch storage that lives for one call), or later `SHARED` (GPU shared memory, Chapter 13).

uopc/dtype.py · AddrSpace

```python
class AddrSpace(Enum):
  """Where a UOp lives. ALU = a plain value in a register. The rest are memory."""
  ALU = auto()      # a value (week 1)
  GLOBAL = auto()   # device memory, a function argument (week 2)
  LOCAL = auto()    # call-scope storage: `float acc[4];` (week 2)
  SHARED = auto()   # GPU workgroup shared memory (week 7)

  def __repr__(self): return f"AddrSpace.{self.name}"
```

`addrspace` is a recursive property like `dtype` and `shape`. It is computed by `_infer` when the node is built, and the type checker uses it to reject nonsense such as loading from a number.

4.2 The memory ops

Six new ops, each small:

Op	Sources	Arg	dtype, shape, addrspace	Meaning	C
`BUFFER`	none	`ParamArg(slot, dtype, (n,))`	`dtype`, `(n,)`, `GLOBAL`	Caller's array, argument number `slot`	`float* data0`
`ALLOC`	none	`ParamArg(slot, dtype, (n,), space)`	`dtype`, `(n,)`, `LOCAL`	Call-scope scratch storage	`float acc0[4];`
`INDEX`	`(mem, i)`	none	`dtype`, `()`, mem's	The *address* of element `i`	`data0[i]`
`LOAD`	`(addr,)`	none	`dtype`, `()`, `ALU`	Read the value at an address	`float val1 = data0[i];`
`STORE`	`(addr, value)`	none	`void`, `()`, `ALU`	Write a value to an address	`data1[i] = v;`
`AFTER`	`(mem, *deps)`	none	same as `mem`	`mem`, but only once `deps` have happened	nothing
`SINK`	`(*effects)`	none	`void`	The root: these effects are the program	nothing

Memory is always *flat*. A 2-D matrix is a 1-D buffer plus an index expression `i*N + j` that you write yourself (and that rangeify writes for you in Chapter 6). That keeps the memory model trivial and puts all layout logic in index arithmetic, where the simplifier of Chapter 3 can see it.

`BUFFER` and `ALLOC` take a `ParamArg`, the same record used for `PARAM`:

uopc/uop.py · ParamArg

```
@dataclass(frozen=True)
class ParamArg:
  slot: int
  dtype: DType
  shape: tuple[int, ...] = ()
  addrspace: AddrSpace | None = None   # only used by ALLOC (LOCAL or SHARED)
```

The `slot` of a `BUFFER` is its position in the C function's argument list. The `slot` of an `ALLOC` is just a number to make its name unique (`acc0`, `acc1`). The type rules for the memory ops are a few lines of

`_infer`:

uopc/uop.py:235-240

```
  if op is Ops.PARAM: return arg.dtype, arg.shape, A
  if op is Ops.BUFFER:
    _check(len(arg.shape) == 1, "BUFFER is flat")
    return arg.dtype, arg.shape, AddrSpace.GLOBAL
  if op is Ops.ALLOC:
    _check(len(arg.shape) == 1, "ALLOC is flat")
```

uopc/uop.py:252-256

```
    return src[0].dtype, (len(src),) + src[0].shape, A
  if op is Ops.INDEX:
    x, idx = src[0], src[1:]
    _check(len(idx) <= len(x.shape), f"INDEX: {len(idx)} indices into shape {x.shape}")
    _check(all(i.dtype == dtypes.i32 and i.shape == () for i in idx), "INDEX indices are i32
scalars")
```

uopc/uop.py:274-282

```
    return (dtypes.bool if op in (Ops.CMPLT, Ops.CMPNE) else a.dtype), a.shape, A
  if op is Ops.LOAD:
    _check(src[0].addrspace is not A, "LOAD needs memory")
    return src[0].dtype, src[0].shape, A
  if op is Ops.STORE:
    _check(src[0].addrspace is not A, "STORE needs memory")
    _check(src[0].dtype == src[1].dtype, f"STORE of {src[1].dtype} into {src[0].dtype}")
    return dtypes.void, (), A
  if op is Ops.AFTER: return src[0].dtype, src[0].shape, src[0].addrspace
```

Read them as a type system. `INDEX` *preserves* the address space of what it indexes: indexing a `GLOBAL` buffer gives a `GLOBAL` address, not a value. `LOAD` requires its source to be memory and produces an `ALU` value. `STORE` requires memory, checks that the stored value has the buffer's dtype, and produces `void`, which is nothing you can compute with. `AFTER` is transparent: it has the type of its first source.

The builder methods on `UOp` are one-liners:

uopc/uop.py:201-206

```
  def index(self, *idx) -> UOp:
    return UOp(Ops.INDEX, (self, *(i if isinstance(i, UOp) else UOp.const(int(i)) for i in idx)))
  def load(self) -> UOp: return UOp(Ops.LOAD, (self,))
  def store(self, v) -> UOp: return UOp(Ops.STORE, (self, v))
  def after(self, *deps) -> UOp: return UOp(Ops.AFTER, (self, *deps)) if deps else self
```

A short session shows the properties flowing through:

```
>>> from uopc.uop import UOp, Ops, ParamArg, sink
>>> from uopc.dtype import dtypes
>>> from uopc.render_c import render_c
>>> def buf(slot, n, dtype=dtypes.f32): return UOp(Ops.BUFFER, (), ParamArg(slot, dtype, (n,)))
>>> a, b, c = buf(0, 4), buf(1, 4), buf(2, 4)
>>> a, a.dtype, a.shape, a.addrspace
(UOp(Ops.BUFFER, arg=ParamArg(slot=0, dtype=dtypes.f32, shape=(4,), addrspace=None)), dtypes.f32,
(4,), AddrSpace.GLOBAL)
>>> p = a.index(2)
>>> p.dtype, p.shape, p.addrspace
(dtypes.f32, (), AddrSpace.GLOBAL)
>>> v = p.load()
>>> v.dtype, v.shape, v.addrspace
(dtypes.f32, (), AddrSpace.ALU)
>>> st = c.index(2).store(v * 2.0)
>>> st.dtype, st.shape, st.addrspace
(dtypes.void, (), AddrSpace.ALU)
>>> a.index(2) is p, a.index(2).load() is v
(True, True)
```

The last line is the crux of this chapter. Hash consing still applies to memory ops, so "load `a[2]`" written twice is *one* node. For a buffer that nobody writes, that is exactly right: reading an input twice gives the same value, and computing it once is free CSE. For a buffer that the program *does* write, it is exactly wrong, and we need a way to say "the load after the store".

`buf` is the same helper as `tests/helpers.py` provides; we use it throughout:

`tests/helpers.py · buf`

```
def buf(slot, n, dtype=dtypes.f32): return UOp(Ops.BUFFER, (), ParamArg(slot, dtype, (n,)))
```

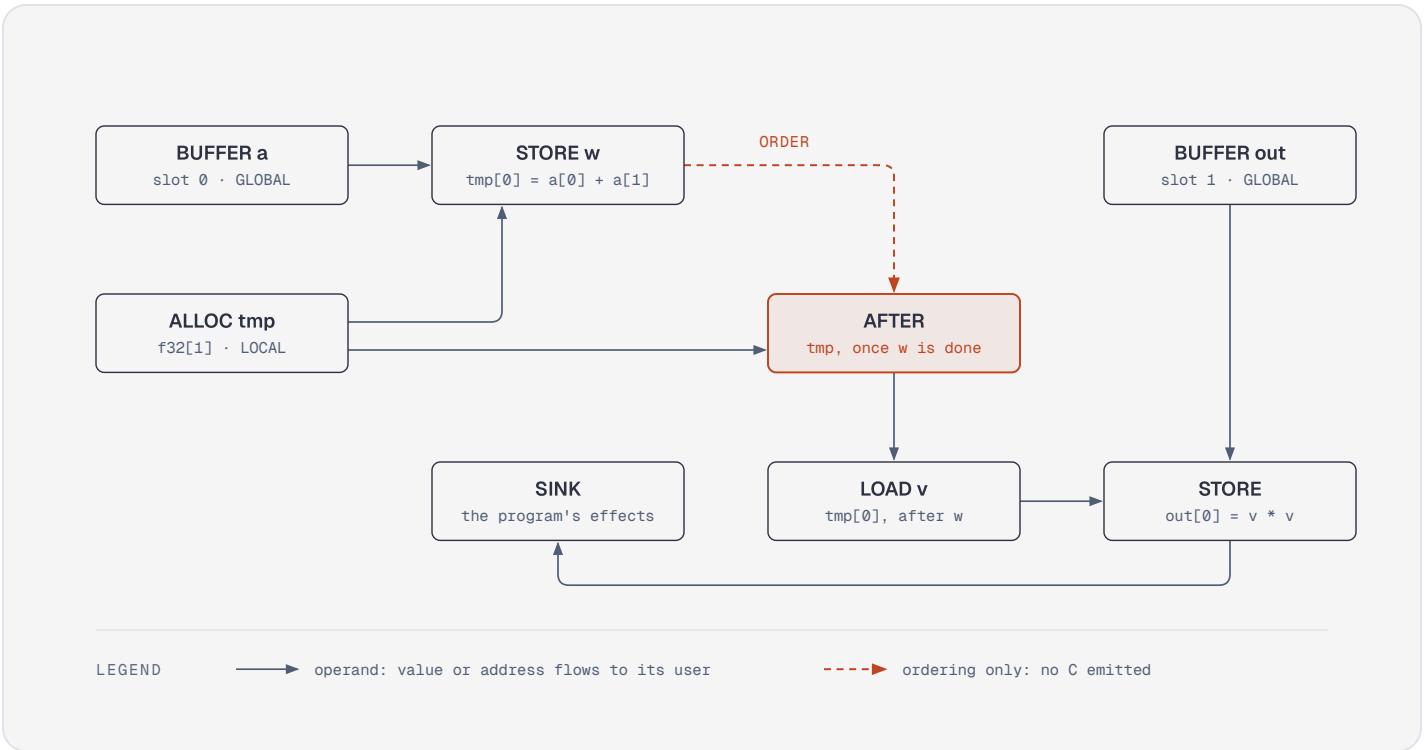
4.3 AFTER: order in a graph that has none

Here is the smallest program where order matters, from the week-2 tests. It writes `a[0] + a[1]` into a one-element scratch buffer, then reads it back and squares it:

tests/test_w02_render.py · test_alloc_and_after

```
def test_alloc_and_after():
  # tmp[0] = a[0] + a[1]; out[0] = tmp[0] * tmp[0]  -- AFTER orders the read after the write
  a, out = buf(0, 2), buf(1, 1)
  tmp = UOp(Ops.ALLOC, (), ParamArg(0, dtypes.f32, (1,)))
  w = tmp.index(0).store(a.index(0).load() + a.index(1).load())
  v = tmp.after(w).index(0).load()
  O = np.zeros(1, np.float32)
  run_c(sink(out.index(0).store(v * v)), {0: np.array([1, 2], np.float32), 1: O})
  assert O[0] == 9
```

The key line is `v = tmp.after(w).index(0).load()`. `tmp.after(w)` is a new node, `AFTER(tmp, w)`. It *is* `tmp`: same dtype, shape and address space, and the renderer prints it as `tmp`. But it has an extra source, the store `w`, and in a DAG an extra source is a dependency. Anything built on `tmp.after(w)` can only run after `w`. And because `AFTER(tmp, w)` is a different node from `tmp`, the load built on it is a different node from a load built on plain `tmp`. Hash consing can no longer merge "before" and "after".



Here is the program after linearization, one line per node, with sources as `%n` references. The printer is a scratch helper, not part of `uopc`, and worth having in your own compiler from day one:

```
from uopc.uop import ParamArg, RangeArg
from uopc.linearize import linearize
def listing(lin):
  n = {u: i for i, u in enumerate(lin)}
  for i, u in enumerate(lin):
    a = u.arg
    if isinstance(a, ParamArg): a = f"slot={a.slot} {a.dtype.name}{list(a.shape)}"
    elif isinstance(a, RangeArg): a = f"idx={a.idx}"
    elif a is None: a = ""
    print(f"%{i:<3d}{u.op.name:7s}{' '.join(f'%{n[s]}' for s in u.src):16s}{a}")
listing(linearize(sink(out.index(0).store(v * v))))
```

```
%0  BUFFER                slot=1 f32[1]
%1  CONST                 0
%2  INDEX  %0 %1
%3  ALLOC                 slot=0 f32[1]
%4  INDEX  %3 %1
%5  BUFFER                slot=0 f32[2]
%6  INDEX  %5 %1
%7  LOAD   %6
%8  CONST                 1
%9  INDEX  %5 %8
%10 LOAD   %9
%11 ADD    %7 %10
%12 STORE  %4 %11
%13 AFTER  %3 %12
%14 INDEX  %13 %1
%15 LOAD   %14
%16 MUL    %15 %15
%17 STORE  %2 %16
%18 SINK   %17
```

and as C:

```
void kernel(float* restrict data0, float* restrict data1) {
  float acc0;
  float val1 = data0[0];
  float val2 = data0[1];
  acc0 = (val1+val2);
  float val3 = acc0;
  data1[0] = (val3*val3);
}
```

`AFTER` (`%13`) produced no C at all. Its whole job was to put `%12` before `%15` in the linear order.

What happens without it

Delete `.after(w)` and load from `tmp` directly:

```
v_bad = tmp.index(0).load()
print(render_c(sink(out.index(0).store(v_bad * v_bad))))
```

```
void kernel(float* restrict data1) {
  float acc0;
  float val1 = acc0;
  data1[0] = (val1*val1);
}
```

The store `w` vanished, and with it every read of `data0`. Nothing reachable from the `SINK` depends on `w`, so `w` is not part of the program. What remains reads an uninitialized variable, which is undefined behavior in C. It "works" or fails depending on what happened to be on the stack. The function signature even lost its first parameter, since `data0` is no longer used. (`run_c` computes the argument list from the same graph, so it still calls the function correctly.)

This is the general rule of the memory model, and it is worth stating as an invariant:

> **Key idea:** every read of memory that the program itself writes must go through an `AFTER` that names the write. Reads of memory that the program never writes (inputs) need no `AFTER`, and get CSE for free.

Why a dependency edge, and not something heavier

Compilers have several ways to put side effects into a graph. LLVM keeps instructions in ordered basic blocks, so order is simply list position, and builds *MemorySSA* on top to recover which store a load can see. Sea-of-nodes compilers (HotSpot C2, V8's TurboFan) thread an explicit *memory state* value through every load and store, so a store consumes the old memory state and produces a new one. JAX threads *effect tokens* through ordered side effects. All of these turn "happens before" into data flow, so that the usual graph algorithms respect it.

`AFTER` is the lightest version of that idea: a node that means "this memory, once these effects have happened". It costs one node per ordering constraint and nothing in the generated code. It is also exactly the tool we need for accumulators in Chapter 5, where `acc.after(init, k)` means "the accumulator, after it was initialized, inside loop `k`". tinygrad uses the same op with the same name.

SINK

A program with stores has no single "result value". Its result is a set of effects. `SINK` gathers them: `sink(store1, store2, ...)` is the root of the graph, its dtype is `void`, and the linearizer and renderer start from it. Anything not reachable from the `SINK` is dead code by definition, as the missing store above showed.

4.4 A loopless program, end to end

The first week-2 test computes `c = a*b + 1` on four elements, one store per element:

tests/test_w02_render.py · test_elementwise_loopless

```
def test_elementwise_loopless():
  a, b, c = buf(0, 4), buf(1, 4), buf(2, 4)
  s = sink(*[c.index(i).store(a.index(i).load() * b.index(i).load() + 1.0) for i in range(4)])
  A, B, C = np.arange(4, dtype=np.float32), np.full(4, 2, np.float32), np.zeros(4, np.float32)
  run_c(s, {0: A, 1: B, 2: C})
  np.testing.assert_allclose(C, A * B + 1)
```

`run_c` renders the `SINK`, compiles the C, and calls it with the numpy arrays. Here is the C it produces (we pass `name="elementwise"` to `render_c`; the prelude of helper functions is omitted):

```
void elementwise(float* restrict data0, float* restrict data1, float* restrict data2) {
  float val1 = data0[0];
  float val2 = data1[0];
  data2[0] = ((val1*val2)+1.0f);
  float val3 = data0[1];
  float val4 = data1[1];
  data2[1] = ((val3*val4)+1.0f);
  float val5 = data0[2];
  float val6 = data1[2];
  data2[2] = ((val5*val6)+1.0f);
  float val7 = data0[3];
  float val8 = data1[3];
  data2[3] = ((val7*val8)+1.0f);
}
```

The rest of this chapter takes that pipeline apart: how the nodes got into this order (section 4.5), how each became a line (4.6), how the C became machine code callable from Python (4.7 and 4.8), and which mistakes the type system rejected along the way (4.9).

4.5 Ordering the graph

C is a list of statements. The graph is a DAG. *Linearizing* means choosing a list order in which every node comes after all of its sources: a topological order. Any topological order is a correct program for a loopless graph, because every dependency, including every ordering dependency added by `AFTER`, is an edge.

`UOp.toposort` (Chapter 1) already produces one: an iterative depth-first search that emits a node after its children. Run on the elementwise program it visits the first store's operands, emits them, emits the store, then moves on to the second store. That is why the C above is grouped by element.

The real `linearize` in `uopc/linearize.py` does more, because loops impose constraints that a plain topological order ignores: a node that depends on loop variable `k` must sit *inside* loop `k`, and a node that does not depend on `k` should sit outside it. For a loopless program all those constraints are empty and `linearize` reduces to a topological order with ties broken by toposort position. The algorithm in full is the subject of Chapter 5.

4.6 The C renderer

The renderer walks the linear order once and produces one C function. Here is the main loop:

`uopc/render_c.py · CRenderer.render`

```
  def render(self, sink: UOp, name: str = "kernel") -> str:
    lin = linearize(sink)
    uses: dict[UOp, int] = {}
    for u in lin:
      for s in u.src: uses[s] = uses.get(s, 0) + 1
    r: dict[UOp, str] = {}
    lines: list[str] = []
    depth = 1
    bufs = sorted({u for u in lin if u.op is Ops.BUFFER}, key=lambda u: u.arg.slot)
    counter = 0
    def var(prefix: str) -> str:
      nonlocal counter
      counter += 1
      return f"{prefix}{counter}"
    def line(s: str): lines.append("  " * depth + s)

    for u in lin:
      op = u.op
      if op is Ops.CONST: r[u] = cconst(u.arg, u.dtype)
      elif op is Ops.BUFFER: r[u] = f"data{u.arg.slot}"
      elif op is Ops.ALLOC:
        r[u] = f"{'smem' if u.addrspace is AddrSpace.SHARED else 'acc'}{u.arg.slot}"
        pre = self.shared_prefix if u.addrspace is AddrSpace.SHARED else ""
        if self._scalar(u): line(f"{ctype(u.dtype)} {r[u]};")     # a size-1 accumulator is just
a variable
        else: line(f"{pre}{ctype(u.dtype)} {r[u]}[{u.arg.shape[0]}];")
      elif op is Ops.AFTER: r[u] = r[u.src[0]]
      elif op is Ops.INDEX:
        base = u.src[0]
        while base.op is Ops.AFTER: base = base.src[0]
        r[u] = r[u.src[0]] if base.op is Ops.ALLOC and self._scalar(base) else f"{r[u.src[0]]}
[{r[u.src[1]]}]"
      elif op is Ops.LOAD:
        r[u] = var("val")
        line(f"{ctype(u.dtype)} {r[u]} = {r[u.src[0]]};")
      elif op is Ops.STORE: line(f"{r[u.src[0]]} = {r[u.src[1]]};")
      elif op is Ops.RANGE:
        r[u] = self.range_name(u)
        for l in self.render_range(u, r[u], r[u.src[0]]): line(l)
        if self.is_loop(u): depth += 1
      elif op is Ops.END:
        for rng in u.src[1:]:
          if self.is_loop(rng):
            depth -= 1
            line("}")
      elif op is Ops.BARRIER: line(self.barrier + ";")
      elif op is Ops.SINK: pass
      elif op is Ops.WHERE:
        expr = f"({r[u.src[0]]}?{r[u.src[1]]}:{r[u.src[2]]})"
        self._assign(u, expr, uses, r, line, var)
      elif op in ALU:
        if len(u.src) == 1: expr = _unop(op, r[u.src[0]], u.dtype, u.arg)
```

```
        else: expr = _binop(op, r[u.src[0]], r[u.src[1]], u.src[0], u.src[1], u.dtype)
        self._assign(u, expr, uses, r, line, var)
      elif op is Ops.WMMA: self.render_wmma(u, r, line)
      else: raise NotImplementedError(f"C renderer can't handle {op}")

    params = ", ".join(f"{self.buffer_prefix}{ctype(b.dtype)}* {self.restrict} data{b.arg.slot}"
for b in bufs)
    return self.header(sink, lin) + f"{self.kernel_prefix} {name}({params}) {{\n" +
"\n".join(lines) + "\n}\n"
```

It keeps a dict `r` from each UOp to the C *expression* that denotes it, and a list of output lines. Each op either *defines a name* (a line like `float val1 = data0[0];`), or *is an expression* that later lines paste into themselves (`r[u] = "(val1*val2)"`), or *emits a statement* (stores, loop headers). The cases below go through the decisions in that loop.

Names

UOp	Name	Example
`BUFFER` slot `n`	`data{n}`	`data0`
`ALLOC` slot `n`	`acc{n}` (or `smem{n}` for shared)	`acc0`
`LOAD`	`val{counter}`	`val3`
multi-use ALU	`alu{counter}`	`alu2`
`RANGE` idx `n`	`ridx{n}`	`ridx1`

Names come from a single counter, so they are unique within the function and *deterministic*: the same graph always produces the same text, character for character. That matters more than it looks. The runtime caches compiled code by a hash of the source text (section 4.8), so any nondeterminism in naming, such as iterating over a Python `set` of nodes, would make identical kernels miss the cache.

Expressions and statements: what gets inlined

Pure ALU ops used once become expressions and are pasted into their user. Ops used more than once get a named variable. The decision is in `_assign`:

```
uopc/render_c.py · CRenderer._assign
```

```python
def _assign(self, u, expr, uses, r, line, var):
  if uses.get(u, 0) <= 1: r[u] = expr   # used once: inline it
  else:
    r[u] = var("alu")
    line(f"{ctype(u.dtype)} {r[u]} = {expr};")
```

Why not inline everything? Because the graph is a DAG, and a DAG can be exponentially smaller than the tree you get by expanding it. Consider `v = v*v + v` applied four times:

```python
a, out = buf(0, 1), buf(1, 1)
v = a.index(0).load()
for _ in range(4): v = v * v + v
print(render_c(sink(out.index(0).store(v)), "chain"))
```

```c
void chain(float* restrict data0, float* restrict data1) {
  float val1 = data0[0];
  float alu2 = ((val1*val1)+val1);
  float alu3 = ((alu2*alu2)+alu2);
  float alu4 = ((alu3*alu3)+alu3);
  data1[0] = ((alu4*alu4)+alu4);
}
```

Every intermediate value is used three times. Fully inlined, the final expression would contain $3^4 = 81$ copies of `val1`, and twenty levels would be 3.5 billion. gcc would eventually CSE it back, after we spent gigabytes printing it. Naming every value used more than once keeps the C linear in the size of the graph. Inlining the single-use ones keeps it readable.

Loads are never inlined, even when used once: a `LOAD` always becomes `float valN = ...;`. The reason is ordering. The line for a load sits at the load's position in the linear order, which the linearizer chose to respect every `AFTER`. If we inlined the load into its user, the read would move to wherever the user is, possibly past a store to the same address. Naming the load pins the read to its place. (gcc is free to reorder it again, but only where the C semantics allow, and `restrict` tells it exactly where that is.)

`CONST`, `INDEX` and `AFTER` produce expressions and never lines: a constant is a literal, an index is `data0[expr]`, and `AFTER` is just its first source's expression.

Constants

`uopc/render_c.py · cconst`

```python
def cconst(v, d: DType) -> str:
  if d == dtypes.bool: return "1" if v else "0"
  if d == dtypes.i32: return str(v) if v != -2**31 else "(-2147483647-1)"
  if math.isnan(v): return "NAN"
  if math.isinf(v): return "INFINITY" if v > 0 else "-INFINITY"
  s = repr(float(v))
  return s + "f" if d == dtypes.f32 else f"(({d.ctype}){s}f)"
```

```
>>> from uopc.render_c import cconst
>>> [cconst(v, dtypes.f32) for v in (1.0, 0.1, -0.0, 1e-45, float("inf"), float("nan"),
3.4028234663852886e38)]
['1.0f', '0.1f', '-0.0f', '1e-45f', 'INFINITY', 'NAN', '3.4028234663852886e+38f']
>>> [cconst(v, dtypes.i32) for v in (0, -5, 2**31 - 1, -2**31)], cconst(True, dtypes.bool)
(['0', '-5', '2147483647', '(-2147483647-1)'], '1')
```

Each case is there for a reason:

- **The `f` suffix.** In C, `1.0` is a `double`. `x * 1.0` with `float x` promotes `x` to double, multiplies in double precision, and rounds back. That is slower, defeats single-precision vectorization, and is not the f32 semantics that `exec_alu` promises. `1.0f` keeps the arithmetic in float.
- **`repr` for the digits.** Python's `repr` of a float is the shortest decimal string that round-trips to the same double. Any value that is exactly representable in float32 (which a correctly folded f32 constant is) therefore prints as a decimal that C parses back to exactly that float. `%.6g` style formatting would silently lose bits.
- **`INFINITY` and `NAN`** come from `<math.h>`. `repr` would print `inf`, which is not C.
- **`(-2147483647-1)`.** C has no negative integer literals. `-2147483648` is the unary minus applied to `2147483648`, which does not fit in an `int`, so it has type `long` and the expression is a `long`. The standard idiom (it is how `<limits.h>` defines `INT_MIN`) builds the value inside `int` range.

Division

The prelude defines two helpers:

`uopc/render_c.py · PRELUDE`

```
PRELUDE = """#include <math.h>
#include <stdbool.h>
static inline int floordiv_i32(int a, int b) { int q = a / b; return (q * b != a && ((a < 0) !=
(b < 0))) ? q - 1 : q; }
static inline int floormod_i32(int a, int b) { int r = a % b; return (r != 0 && ((r < 0) != (b <
0))) ? r + b : r; }
"""
```

and `_binop` chooses between them and C's operators:

uopc/render_c.py · _binop

```
def _binop(op: Ops, a: str, b: str, ua: UOp, ub: UOp, d: DType) -> str:
  match op:
    case Ops.ADD: return f"({a}+{b})"
    case Ops.MUL: return f"({a}*{b})"
    case Ops.CMPLT: return f"({a}<{b})"
    case Ops.CMPNE: return f"({a}!={b})"
    case Ops.MAX: return f"(({a}>{b})?{a}:{b})"
    case Ops.FLOORDIV:
      # C division truncates; it equals floor division when both sides are non-negative
      if vmin(ua) >= 0 and ub.op is Ops.CONST and ub.arg > 0: return f"({a}/{b})"
      return f"floordiv_i32({a},{b})"
    case Ops.FLOORMOD:
      if vmin(ua) >= 0 and ub.op is Ops.CONST and ub.arg > 0: return f"({a}%{b})"
      return f"floormod_i32({a},{b})"
  raise NotImplementedError(op)
```

C's `a / b` truncates toward zero. Floor division differs exactly when the division is inexact and the operands have opposite signs, and in that case the floor is one less than the truncated quotient. That is `floordiv_i32`. For the remainder, C's `%` has the sign of the dividend while floor modulo has the sign of the divisor. They differ when the remainder is nonzero and the signs disagree, and then adding `b` fixes it. That is `floormod_i32`.

The fast path is the common case. If Chapter 3's bounds analysis proves the numerator is non-negative and the divisor is a positive constant, truncation and floor agree, so the renderer emits plain `/` and `%`, which gcc turns into shifts and masks for powers of two and into multiply-by-reciprocal sequences otherwise. Index arithmetic almost always takes this path, because loop variables start at 0. The test `test_floordiv_rendering_negative` exercises the slow path, with operands loaded from memory whose bounds are unknown:

tests/test_w02_render.py · test_floordiv_rendering_negative

```python
def test_floordiv_rendering_negative():
  a, out = buf(0, 2, dtypes.i32), buf(1, 2, dtypes.i32)
  x = a.index(0).load()
  s = sink(out.index(0).store(x // a.index(1).load()), out.index(1).store(x % a.index(1).load()))
  O = np.zeros(2, np.int32)
  run_c(s, {0: np.array([-7, 2], np.int32), 1: O})
  assert list(O) == [-4, 1]
  assert "floordiv_i32" in render_c(s)
```

```
void divmod(int* restrict data0, int* restrict data1) {
  int val1 = data0[0];
  int val2 = data0[1];
  data1[0] = floordiv_i32(val1,val2);
  data1[1] = floormod_i32(val1,val2);
}
```

The helpers are `static inline`, so gcc inlines them, and in the fast path they cost nothing because they are not called.

Casts, select and math functions

`uopc/render_c.py · _unop`

```python
def _unop(op: Ops, a: str, d: DType, arg) -> str:
  if op is Ops.CAST: return f"(({arg.ctype})({a}))"
  if op is Ops.RECIP: return f"(1.0f/{a})" if d == dtypes.f32 else f"(({d.ctype})1.0f/{a})"
  return f"{UNARY_FN[op]}{'f' if d == dtypes.f32 else ''}({a})"
```

Casts are C casts, `RECIP` is `1.0f/x`, and the transcendental ops call `<math.h>`'s float versions (`exp2f`, `log2f`, `sqrtf`, `sinf`). Calling `exp2` on a float would compute in double. `WHERE` becomes the ternary operator. The `test_casts_and_where` program renders as:

```
void casts(float* restrict data0, int* restrict data1) {
  float val1 = data0[0];
  data1[0] = ((val1<0.0f)?-1:((int)(val1)));
  float val2 = data0[1];
  data1[1] = ((val2<0.0f)?-1:((int)(val2)));
  float val3 = data0[2];
  data1[2] = ((val3<0.0f)?-1:((int)(val3)));
}
```

`(int)(3.7f)` is `3`: C's cast truncates toward zero, matching `exec_alu`'s `int(v)`.

Scratch memory

An `ALLOC` of one element is rendered as a plain scalar variable (`float acc0;`), and an `INDEX` into it renders as just `acc0`. Larger allocations become arrays:

uopc/render_c.py · CRenderer._scalar

```
  def _scalar(self, alloc: UOp) -> bool: return alloc.arg.shape == (1,) and alloc.addrspace is
AddrSpace.LOCAL
```

The scalar case is not cosmetic. A local array is memory as far as C is concerned, and although gcc can often promote small arrays to registers, a scalar is guaranteed to be eligible for a register. Accumulators in Chapter 5 are size-1 allocs that get updated on every iteration of a hot loop, and they must live in registers. For a 4-element scratch array written by four stores, `after(*ws)` names all four writes as dependencies:

```
t4 = UOp(Ops.ALLOC, (), ParamArg(1, dtypes.f32, (4,)))
ws = [t4.index(k).store(a.index(k % 2).load() * float(k)) for k in range(4)]
t4a = t4.after(*ws)
print(render_c(sink(out.index(0).store(t4a.index(0).load() + t4a.index(3).load()))))
```

```
void kernel(float* restrict data0, float* restrict data1) {
  float acc1[4];
  float val1 = data0[0];
  acc1[0] = (val1*0.0f);
  float val2 = data0[1];
  acc1[1] = (val2*1.0f);
  acc1[2] = (val1*2.0f);
  acc1[3] = (val2*3.0f);
  float val3 = acc1[0];
  float val4 = acc1[3];
  data1[0] = (val3+val4);
}
```

Notice `val1` used for both `acc1[0]` and `acc1[2]`. `a.index(0).load()` and `a.index(2 % 2).load()` are the same hash-consed node, so the input is read once: CSE for free, correct because nobody writes `data0`. Notice also `val1*0.0f`: nothing simplified this graph before rendering, and the simplifier would not have removed it anyway, since `x*0.0` is not `0.0` for floats.

The signature, and `restrict`

The function takes one pointer per `BUFFER` reachable from the `SINK`, sorted by slot: `void kernel(float* restrict data0, float* restrict data1)`. The `restrict` qualifier is a promise to the C compiler that, for the lifetime of the call, the memory reached through `data0` is not reached through any other pointer. Without it, gcc must assume that a store to `data1[i]` might change `data0[j]`, so it has to reload after every store and cannot vectorize freely. With it, loads can be hoisted, reordered and combined. In return, *we* must never pass the same array as two different buffers. Our IR has no way to express aliasing buffers, so the promise is easy to keep, but it is a promise. Breaking it is undefined behavior.

4.7 The C ABI in one page

To call the compiled function from Python we need to agree with it, at the machine level, on how arguments are passed. That agreement is the *ABI* (application binary interface). On x86-64 Linux it is the System V AMD64 ABI. The parts we use:

1. **Integer and pointer arguments** go in registers `rdi, rsi, rdx, rcx, r8, r9`, in that order (more go on the stack). Our kernels take only pointers, one per buffer.
2. **A pointer is just an address**, 64 bits. The callee does not know the array's length, its dtype, or whether it is contiguous. `data0[3]` on a `float*` means "the 4 bytes at address `data0 + 12`".
3. **Return values** go in `rax` (or `xmm0` for floats). Our kernels return `void`.

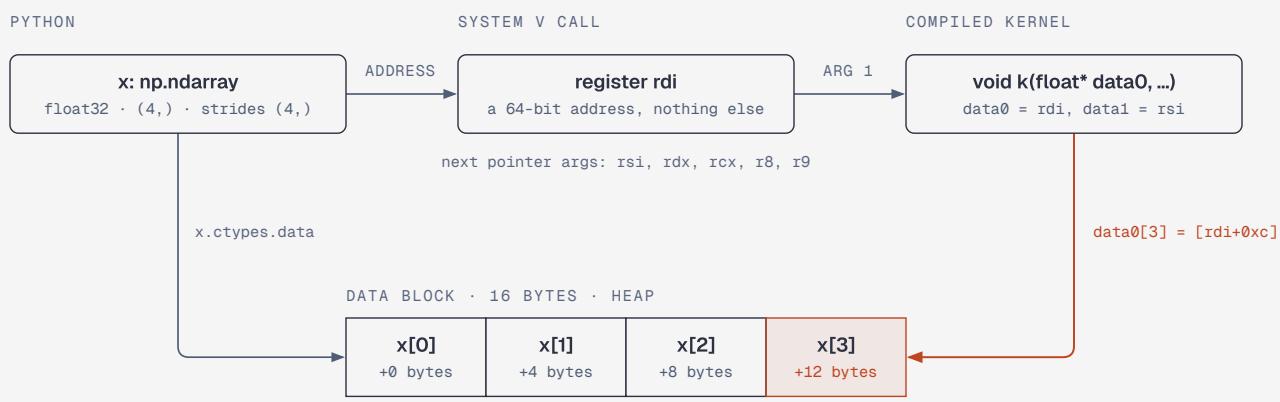
You can see all three in the machine code gcc produced for a tiny kernel:

```
void k(float* restrict data0, float* restrict data1) {
  float val1 = data0[0];
  float val2 = data0[3];
  data1[1] = ((val1*2.0f)+val2);
}
```

```
    1100:   endbr64
    1104:   vmovss xmm0,DWORD PTR [rdi]
    1108:   vmovss xmm1,DWORD PTR [rdi+0xc]
    110d:   vfmadd132ss xmm0,xmm1,DWORD PTR [rip+0xeea]        # 2000 <_fini+0xee4>
    1116:   vmovss DWORD PTR [rsi+0x4],xmm0
    111b:   ret
```

`data0` arrived in `rdi` and `data1` in `rsi`. `data0[3]` is `[rdi+0xc]` (3 × 4 bytes), and `data1[1]` is `[rsi+0x4]`. The multiply and add became a single fused multiply-add, `vfmadd132ss`, because we compile with `-ffp-contract=fast` (section 4.8). The constant `2.0f` lives in a read-only data section, addressed relative to the instruction pointer (`rip`), which is what `-fPIC` asks for. (`objdump -d -M intel` on the cached `.so`; `nm -D` lists the exported symbol: `0000000000001100 T k`.)

On the Python side, `ctypes` does the marshalling. A numpy array is a Python object with a header (dtype, shape, strides) and a pointer to a data block. `arr.ctypes.data` is that pointer as an integer, and `ctypes.c_void_p(...)` passes it as a 64-bit pointer argument. The header is left behind: the C function sees only the address.



dtype, shape and strides stay behind in Python: the kernel trusts that the address points at enough contiguous float32 values, and nothing checks that it does.

Four consequences follow, and the runtime handles only the first:

- **Contiguity.** A numpy view like `x[::2]` has a stride of 8 bytes, but the C code assumes 4. The runtime checks `c_contiguous` and refuses.
- **Dtype.** Nothing checks that a `float*` parameter receives float32 data. Pass a float64 array and the kernel reinterprets its bytes (Pitfalls).
- **Length.** Nothing checks that the array is long enough. Reading past the end is undefined behavior.
- **Lifetime.** The pointer is valid only while the numpy array is alive and not resized. Our calls are synchronous, so the array outlives the call. An asynchronous runtime (GPUs, Chapter 12) has to keep references explicitly.

4.8 The runtime

`uopc/runtime.py` is 66 lines. It has three jobs: compile C to a shared object, cache that by content, and call it.

Compiler flags

`uopc/runtime.py · CFLAGS`

```
CFLAGS = os.environ.get("CFLAGS", "-O3 -march=native -fopenmp -ffp-contract=fast -fassociative-math "
                                  "-fno-signed-zeros -fno-trapping-math").split()
```

Flag	Effect	Why
`-O3`	Full optimization, including auto-vectorization	We want gcc's best code; we only generate simple C
`-march=native`	Use every instruction this CPU has (AVX2, FMA, ...)	Kernels run on the machine that compiled them
`-fopenmp`	Enable `#pragma omp`	`THREAD` axes in Chapter 10
`-ffp-contract=fast`	Allow `a*b+c` to become one FMA instruction	Faster, and more accurate (one rounding instead of two)
`-fassociative-math`	Allow reordering float additions	Required to vectorize float reductions (note 20)
`-fno-signed-zeros`	Treat `-0.0` like `+0.0`	Required by `-fassociative-math`; matches `x+0.0 -> x` in Chapter 3
`-fno-trapping-math`	Assume float ops never trap	Required by `-fassociative-math`
`-shared -fPIC`	Build a position-independent shared library	So it can be loaded into the running Python process
`-lm`	Link the math library	`exp2f`, `sqrtf`, ...

Notice what is *not* there: `-ffast-math`. That flag also assumes there are no NaNs and no infinities, which would break `max` reductions initialized to `-INFINITY` and any code that relies on `nan` propagating. `-fassociative-math` and its two prerequisites are the part of fast-math that we actually want. The FMA contraction means the C program and `exec_alu` can differ in the last bit for `a*b+c`. Folding happens on constants, which gcc would fold the same way, so this does not bite in practice, but it is another place where "bit-exact" has an asterisk. `CFLAGS` can be overridden from the environment.

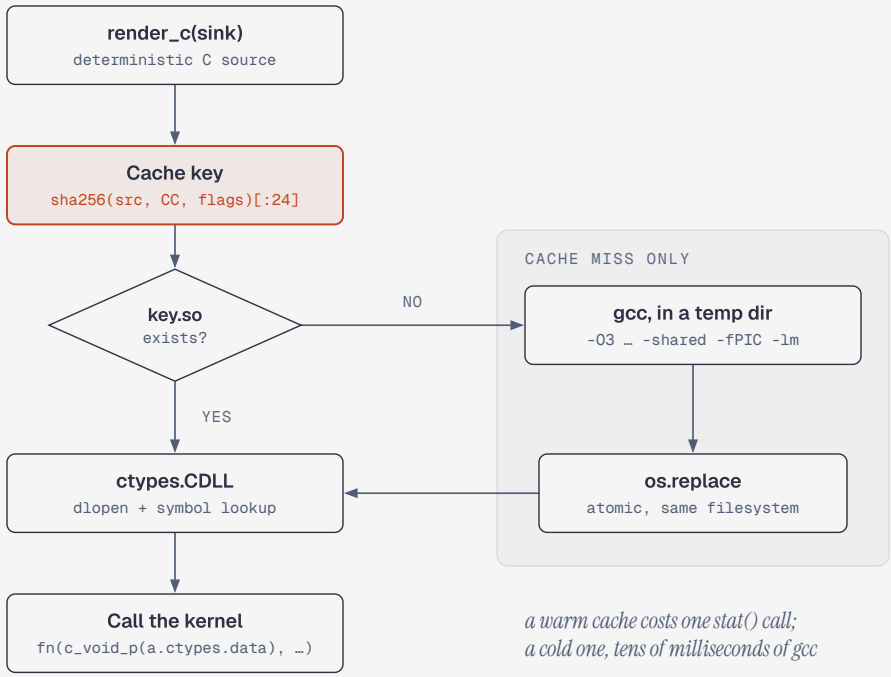
Compiling, with a content-addressed cache

`uopc/runtime.py · compile_c`

```
def compile_c(src: str, flags: list[str] | None = None) -> Path:
  flags = CFLAGS if flags is None else flags
  key = hashlib.sha256((src + "\0" + " ".join([CC, *flags])).encode()).hexdigest()[:24]
  CACHE.mkdir(parents=True, exist_ok=True)
  so = CACHE / f"{key}.so"
  if so.exists(): return so
  with tempfile.TemporaryDirectory(dir=CACHE) as td:   # same filesystem, so os.replace is atomic
    c = Path(td) / "k.c"
    c.write_text(src)
    tmp = Path(td) / "k.so"
    p = subprocess.run([CC, *flags, "-shared", "-fPIC", "-o", str(tmp), str(c), "-lm"],
capture_output=True, text=True)
    if p.returncode != 0:
      numbered = "\n".join(f"{i+1:4d} {l}" for i, l in enumerate(src.splitlines()))
      raise CompileError(f"{CC} failed:\n{p.stderr}\n--- source ---\n{numbered}")
    os.replace(tmp, so)   # atomic: concurrent compiles of the same kernel are fine
  return so
```

The cache key is a SHA-256 of the source text, the compiler, and the flags. The `.so` is stored under `~/.cache/uopc/<key>.so`. The design is *content addressing*: the name of the artifact is a function of everything that determines it, so there is no invalidation logic at all. Change one character of the kernel, or one flag, and the key changes. The same kernel compiled by two processes, or on two runs a week apart, gets the same key. The cost is that the renderer must be deterministic (section 4.6).

Atomicity matters because several processes (pytest workers, say) may compile the same kernel at the same time. Each compiles into its own temporary directory and then `os.replace`s the result into place. `os.replace` is atomic on POSIX *within one filesystem*, so a reader sees either no file or a complete one, never a half-written `.so`. The temporary directory is created *inside* the cache directory for exactly that reason. The first version used the system temp dir, and `os.replace` from `/tmp` (a tmpfs) to `~/.cache` (the disk) failed: it was a cross-filesystem rename (note 10).



A compile error is reported with gcc's message and a numbered listing of the generated source, since the generated source is the thing you need to read:

```
gcc failed:
~/.cache/uopc/tmpmugz9v_0/k.c: In function ‘k’:
~/.cache/uopc/tmpmugz9v_0/k.c:2:14: error: expected ‘;’ before ‘}’ token
    2 |   a[0] = 1.0f
      |              ^
      |              ;
    3 | }
      | ~

--- source ---
   1 void k(float* a) {
   2   a[0] = 1.0f
   3 }
```

(The cache path is shortened to `~/.cache/uopc`; the run used a scratch cache directory.)

Calling

```
uopc/runtime.py · Program
```

```python
class Program:
  """A compiled C function taking one pointer per buffer slot."""
  def __init__(self, src: str, name: str = "kernel", flags: list[str] | None = None):
    self.src, self.name = src, name
    self.lib = ctypes.CDLL(str(compile_c(src, flags)))
    self.fn = getattr(self.lib, name)
    self.fn.restype = None

  def __call__(self, *bufs: np.ndarray):
    for b in bufs:
      if not b.flags.c_contiguous: raise ValueError("buffers must be contiguous")
    self.fn(*(ctypes.c_void_p(b.ctypes.data) for b in bufs))
```

`ctypes.CDLL` calls `dlopen` on the shared object, which maps it into the process. `getattr(lib, name)` looks up the exported symbol. Setting `restype = None` tells ctypes the function returns `void` (the default would read `rax` as an `int`). No `argtypes` are declared. Instead every argument is wrapped in `c_void_p`, which ctypes passes as a 64-bit pointer. That is everything the System V ABI needs for a function of `n` pointers.

A session, against an empty cache:

```
>>> import time, numpy as np
>>> from uopc.runtime import compile_c, Program
>>> src = "void axpy(float* restrict x, float* restrict y) { for (int i = 0; i < 4; i++) y[i] =
2.0f*x[i] + y[i]; }\n"
>>> t0 = time.perf_counter(); so = compile_c(src); print(f"{(time.perf_counter()-t0)*1e3:.1f}
ms", so.name)
60.1 ms e50750a3fe3c54975390ccb4.so
>>> t0 = time.perf_counter(); so = compile_c(src); print(f"{(time.perf_counter()-t0)*1e3:.1f}
ms", so.name)
0.1 ms e50750a3fe3c54975390ccb4.so
>>> p = Program(src, "axpy")
>>> x, y = np.arange(4, dtype=np.float32), np.ones(4, np.float32)
>>> p(x, y); y
array([1., 3., 5., 7.], dtype=float32)
>>> p(x[::2], y[:2])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "uopc/runtime.py", line 55, in __call__
    if not b.flags.c_contiguous: raise ValueError("buffers must be contiguous")
                                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: buffers must be contiguous
```

A cold compile is tens of milliseconds (gcc has to start, parse, optimize, link), and a warm one is a `stat` call. For a GPT training run that compiles a few hundred kernels, the difference between the first run and every

later run is the difference between seconds and nothing.

Gluing it together

uopc/runtime.py · run_c

```
def run_c(sink, buffers: dict[int, np.ndarray], name: str = "kernel") -> str:
  """Render, compile and run a SINK of stores. buffers maps slot -> numpy array. Returns the C
source."""
  from uopc.render_c import render_c
  from uopc.uop import Ops
  src = render_c(sink, name)
  bufs = sorted({u for u in sink.toposort() if u.op is Ops.BUFFER}, key=lambda u: u.arg.slot)
  for b in bufs:
    a = buffers[b.arg.slot]
    if a.dtype != np_dtype(b.dtype) or a.size < b.arg.shape[0]:
      raise TypeError(f"buffer slot {b.arg.slot}: kernel wants {b.arg.shape[0]} x {b.dtype.name},
got {a.size} x {a.dtype}")
  Program(src, name)(*(buffers[b.arg.slot] for b in bufs))
  return src
```

`run_c` is the week-2 entry point: render, find the buffer slots in the same order as the renderer's signature (sorted by slot, reachable from the `SINK`), compile, and call. Both sides compute the parameter list from the same graph with the same sort, so they cannot disagree. Return the source, so tests can inspect it.

4.9 Type systems for shape, dtype and addrspace

The syllabus devotes a lecture to "type systems for shape, dtype, addrspace". By now we have all three, and it is worth being precise about what each one catches. All three are checked at construction time, in `_infer` (Chapter 1), so a malformed node never exists. The error points at the line of Python that built it, not at some later pass that tripped over it. Tracebacks are trimmed to their last line here:

```
>>> from uopc.uop import UOpError
>>> a, ai = buf(0, 4), buf(1, 4, dtypes.i32)
>>> (UOp.const(1.0) + 2.0).load()
uopc.uop.UOpError: LOAD needs memory
>>> ai.index(0).store(UOp.const(1.0))
uopc.uop.UOpError: STORE of dtypes.f32 into dtypes.i32
>>> a.index(UOp.const(1.0))
uopc.uop.UOpError: INDEX indices are i32 scalars
>>> a.index(0).load() + ai.index(0).load()
uopc.uop.UOpError: Ops.ADD: dtype mismatch dtypes.f32 vs dtypes.i32 (use CAST)
>>> a.index(0).load() // a.index(1).load()
uopc.uop.UOpError: Ops.FLOORDIV is integer only
>>> UOp(Ops.BUFFER, (), ParamArg(0, dtypes.f32, (2, 2)))
uopc.uop.UOpError: BUFFER is flat
>>> a.index(0, 1)
uopc.uop.UOpError: INDEX: 2 indices into shape (4,)
>>> a.index(7).load()
UOp(Ops.LOAD, src=(UOp(Ops.INDEX, src=(UOp(Ops.BUFFER, arg=ParamArg(slot=0, dtype=dtypes.f32,
shape=(4,), addrspace=None)), UOp(Ops.CONST, arg=7),)),))
```

Property	Catches	Example
dtype	Mixing ints and floats without a cast; storing the wrong type; float indices; integer-only ops on floats	`STORE of dtypes.f32 into dtypes.i32`
shape	Elementwise ops on mismatched shapes; too many indices; non-flat memory	`INDEX: 2 indices into shape (4,)`
addrspace	Loading from or storing to a value; computing with an address as if it were a value	`LOAD needs memory`

The last line of the session is the important negative result: `a.index(7)` on a 4-element buffer is accepted. Our types describe *what kind* of thing a node is, not *which values* it can take. Several real bugs are invisible to them:

- **Out-of-bounds indices.** The index is an expression with bounds, and Chapter 3 computes them. A render-time check `0 <= vmin(idx)` and `vmax(idx) < n` would catch most out-of-bounds accesses statically (exercise 4). A type system that tracks value ranges is a *refinement type* system, and that is roughly what this check is.
- **Reading before writing.** The missing-`AFTER` bug of section 4.3 type-checks perfectly. Detecting it needs a dataflow analysis: is there a path from the `ALLOC` to the `LOAD` that does not pass through a store?
- **Aliasing.** Passing one numpy array as two `restrict` buffers is invisible to the IR.
- **Runtime marshalling.** The numpy array's dtype and length are not checked against the `BUFFER`.

Each check you add catches bugs earlier and costs code. The course grades "can it be simpler?", so add them when a bug teaches you that you need them, and write the bug down.

4.10 One renderer, several targets

`CRenderer` is a class, not a function, because the CUDA backend of Chapter 12 is a subclass. The differences between C dialects are mostly a handful of strings, so they are class attributes: `kernel_prefix` (`"void"` here, `extern "C" __global__ void __launch_bounds__(...)` for CUDA), `buffer_prefix`, `shared_prefix` for GPU shared memory, `barrier` for `__syncthreads()`, and `restrict` (CUDA spells it `__restrict__`). The methods `header`, `range_name`, `render_range` and `is_loop` are the structural hooks. Everything else, the walk over the linear order, naming, inlining, constants and division, is shared. When you write a renderer for another C-like target (OpenCL, Metal, HIP, or plain C with different intrinsics), start by listing which of these strings and hooks change. If the list is short, subclass. If you find yourself overriding `render` itself, the abstraction is in the wrong place.

4.11 Implementing week 2

A build order in which every step can be tested on its own:

1. **Memory ops in the IR.** `BUFFER`, `ALLOC`, `INDEX`, `LOAD`, `STORE`, `AFTER`, `SINK` in `Ops`, and their rules in `_infer`, including `addrspace`. Test the error messages of section 4.9 first: they are the cheapest tests you will ever write.
2. **A topological order.** For loopless programs, `toposort` from the `SINK` is enough. Print it with `%n` references, as in section 4.3, and check by eye that every store comes before the `AFTER` that names it.
3. **The renderer, one op at a time.** Start with `BUFFER`, `CONST`, `INDEX`, `LOAD`, `STORE` and `ADD`. Compile the output by hand with `gcc -c` before writing any runtime code. Then add `MUL`, comparisons, `WHERE`, casts, `RECIP` and the math functions, then `ALLOC` and `AFTER`, then the floor-division helpers with the `vmin` fast path.
4. **The runtime.** `compile_c` without the cache first, then add content addressing, then the atomic rename. Call with ctypes and compare with numpy.

5. **Differential tests.** Run random operands through every ALU op, compiled versus `exec_alu`, as in Chapter 3. This is the test that finds the bugs you did not think of.
6. **The week-2 tests.** `tests/test_w02_render.py`: elementwise, `ALLOC` with `AFTER`, negative floor division, casts and `WHERE`.

Keep the renderer boring. It should be a direct transcription of the linear order, with every decision about *what* to compute made earlier, in the graph. Optimizations belong in rewrites, where they can be tested on the IR, not in string manipulation.

Pitfalls

Forgetting `AFTER`. Loading from scratch memory without `.after(store)` gives a load that does not depend on the store, so the store is dead and the load reads garbage (section 4.3). The C compiles, the test may even pass by luck, and gcc with `-Wall` may or may not warn about the uninitialized variable. The fix is the invariant: a read of anything this program writes goes through `AFTER` naming the write.

Rendering `FLOORDIV` as `/` (note 4). It is correct for non-negative operands only. If you take the shortcut without the `vmin` check, everything passes until the first negative index (padding, shifted windows, any `x - c`). `test_floordiv_rendering_negative` is the regression test.

Float literals without `f`. `x * 0.5` with `float x` is computed in double. It is correct-ish, slower, blocks single-precision vectorization, and differs from `exec_alu`'s f32 semantics in rounding. Always suffix.

`-2147483648` is not an int. In C, it is `-(2147483648)`, and `2147483648` is a `long`. Use `(-2147483647-1)`.

Wrong dtype through ctypes. The runtime does not check numpy dtypes. Passing a float64 array to a `float*` kernel reinterprets the bytes:

```
>>> y64 = np.ones(4)
>>> p(x, y64); y64
array([3.84000000e+02, 1.14688016e+05, 1.00000000e+00, 1.00000000e+00])
```

The kernel wrote four floats (16 bytes) over the first two doubles. Nothing crashed. That listing calls a `Program` directly, which takes raw arrays and cannot know better. `run_c` knows each `BUFFER`'s `ParamArg`, and after this pitfall was found it checks dtype and length before calling:

```
TypeError: buffer slot 0: kernel wants 4 x f32, got 4 x float64
```

(`test_run_c_checks_buffer_dtypes`.) Exercise 5 asks you to add the same check to your runtime.

Cross-filesystem `os.replace` (note 10). Compiling in `/tmp` and renaming into `~/.cache` fails when they are different filesystems, and even where a fallback copy works, it is not atomic. Create the temporary directory inside the cache directory.

Nondeterministic rendering. If the order of lines or the choice of names depends on Python `set` or `dict` iteration over objects hashed by `id`, two runs produce different source text for the same graph, and the content-addressed cache misses every time. Iterate in linearized order and number names with a counter.

Inlining everything. Rendering every node as an expression produces C whose size is exponential in the depth of shared subexpressions. Name anything used more than once.

Inlining loads. Moving a load into its user can move it past a store to the same address. Name every load at its position in the linear order.

Exercises

1. ★ **Predict the C.** Without running it, write the C that `render_c` produces for `sink(out.index(0).store((a.index(0).load() + 1.0) * (a.index(0).load() + 1.0)))` with `a`, `out` as 4-element f32 buffers. Which nodes get names? *Hint: count uses after hash consing.*

2. ★ **Find the bug.** A student writes a two-element dot product with a scratch accumulator: `acc = ALLOC f32[1]`, `w0 = acc[0].store(a[0]*b[0])`, `w1 = acc[0].store(acc[0].load() + a[1]*b[1])`, `out[0].store(acc.after(w1)[0].load())`. The result is wrong. Why, and what is the one-word fix?

3. ★★ **Dot product by hand.** Write a loopless program that computes the dot product of two 8-element vectors into `out[0]`, using a one-element `ALLOC` updated by a chain of eight stores. Check it against numpy with `run_c`, and look at the C.

4. ★★ **Static bounds checking.** Add a check to your renderer: for every `INDEX` into a `BUFFER` or `ALLOC` of size `n`, raise if `vmin(idx) < 0` or `vmax(idx) >= n`. Which test programs from weeks 2 and 3 does it accept? Construct a program it wrongly rejects. (*Hint: the dependency problem of Chapter 3.*)

5. ★★ **Safer calls.** Make your runtime check each numpy array's dtype and size against the corresponding `BUFFER`'s `ParamArg` before calling (as `run_c` does). What does it cost per call? The kernelizer's hot path (`realize`) skips the check because it allocates every array itself: is that a sound argument?

6. ★★ **Why loads are named.** Construct a graph (with `AFTER`) where inlining a single-use `LOAD` into its user would read the wrong value. *Hint: you need a store between the load's position and its user's.*

7. ★★ **Disassemble.** Compile `test_elementwise_loopless`'s kernel, find the `.so` in the cache, and read it with `objdump -d -M intel`. Did gcc vectorize the four elements into one `vmulps`/`vaddps`? Remove `restrict` from the source by hand, recompile with `compile_c`, and compare.

8. ★★★ **A second backend.** Write `render_py(sink)` that emits a Python function operating on numpy arrays (one statement per UOp, same naming scheme), and use it as an oracle: run every week-2 test program through both backends and compare outputs. What do you have to do about floor division? About f32 rounding?

9. ★ **Constants.** For each value, predict `cconst(v, dtypes.f32)`: `2.5`, `1e10`, `-float("inf")`, `0.30000001192092896`. Why is `repr` safe for the last one?

Solutions to ★ exercises

1. `a.index(0).load()` is one node, used by both `+` nodes, but the two `+ 1.0` nodes are *also* the same node (same op, same sources), so the multiply's two sources are one node used twice. The C is:

```
void kernel(float* restrict data0, float* restrict data1) {
  float val1 = data0[0];
  float alu2 = (val1+1.0f);
  data1[0] = (alu2*alu2);
}
```

2. `w1` loads `acc[0]` without `AFTER(w0)`, so its load does not depend on `w0`, nothing depends on `w0`, and `w0` is dead. `w1` reads an uninitialized accumulator. The rendered C says so plainly (note that `data0[0]` and `data1[0]` are never read):

```
void kernel(float* restrict data0, float* restrict data1, float* restrict data2) {
  float acc0;
  float val1 = acc0;
  float val2 = data0[1];
  float val3 = data1[1];
  acc0 = (val1+(val2*val3));
  float val4 = acc0;
  data2[0] = val4;
}
```

Fix: load through `acc.after(w0)` inside `w1`.

9. `2.5f`, `10000000000.0f`, `-INFINITY`, `0.30000001192092896f`. The last value is exactly a float32 (it is `f32(0.3)`), and `repr` prints the shortest decimal that identifies the double. That decimal is closer to this float than to any other float, so C's float parser returns exactly this float.

Quiz

1. Why does hash consing make `AFTER` necessary?
2. What does `AFTER(mem, dep)` render as in C?
3. Why is a single-use `LOAD` given a variable name, while a single-use `ADD` is inlined?
4. When does the renderer emit plain `/` for `FLOORDIV`?
5. What does `restrict` promise, and who must keep the promise?
6. In which register does the second buffer pointer arrive on x86-64 Linux?
7. Why is the compiled-kernel cache keyed on the source text and the flags, and not on, say, the UOp graph?
8. Name two bugs that type-check fine in our IR.

Answers

1. Hash consing merges structurally equal nodes, so "load `tmp[0]`" before and after a store would be the same node. `AFTER(tmp, store)` is a different node, so loads built on it are different nodes, and the store becomes a dependency.
2. Nothing. It renders as its first source; its only effect is on the linear order.
3. A load must happen at its position in the linear order, which respects stores; inlining could move it past a store. A pure ALU op has no such constraint, and inlining it keeps the C readable.

4. When bounds analysis proves the numerator non-negative (`vmin >= 0`) and the divisor is a positive constant. Then truncation and floor agree.
5. That memory reached through that pointer is not reached through any other pointer during the call. The caller (our runtime) must keep it; the compiler exploits it.
6. `rsi` (the first is in `rdi`).
7. The source text plus compiler and flags fully determine the `.so`, and that key is stable across processes and runs. A graph key would have to account for renderer changes too. (A graph-level cache exists as well, one level up, in Chapter 7.)
8. Any two of: out-of-bounds constant or computed indices; reading scratch memory before writing it (missing `AFTER`); passing aliased buffers; passing a numpy array of the wrong dtype or length.

Further reading

- Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed.: chapter 3 (machine-level programs, x86-64 calling conventions) and chapter 7 (linking, shared libraries, position-independent code).
- Michael Matz, Jan Hubička, Andreas Jaeger, Mark Mitchell (eds.), *System V Application Binary Interface, AMD64 Architecture Processor Supplement*.
- Ulrich Drepper, "How To Write Shared Libraries" (2011).
- The Python documentation for `ctypes`, and numpy's documentation for `ndarray.ctypes`.
- The GCC manual, "Options That Control Optimization": `-ffast-math` and its component flags.
- ISO/IEC 9899 (the C standard), §6.7.3.1 "Formal definition of restrict".
- Cliff Click and Michael Paleczny, "A Simple Graph-Based Intermediate Representation", IR '95 (sea of nodes, memory as a value).
- LLVM documentation, "MemorySSA".
- tinygrad, `tinygrad/renderer/cstyle.py` (the C-style renderer this chapter's design follows) and `tinygrad/runtime/ops_cpu.py`.

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

`uopc/render_c.py · 143 lines`

`uopc/runtime.py · 70 lines`

`tests/test_w02_render.py · 45 lines`

`tests/helpers.py · 15 lines`

5. Loops and the linearizer

A 1024x1024 matrix multiply is a billion multiply-adds. Written in the loopless language of Chapter 4 it would be a billion `STORE` nodes and a C file measured in gigabytes. Every real kernel is a loop nest, so the IR needs loops. Adding them to the IR is easy: two ops, `RANGE` and `END`. Getting them back *out* of the IR is not. The graph says which values depend on which loop variables, but it does not say which loop is nested inside which, which statements go inside a loop and which go before it, or which of two sibling loops runs first. The *linearizer* has to decide all of that, and decide it correctly for every program. It took four designs to get right. This chapter writes GEMM and convolution by hand in the loop IR, then builds the linearizer, including the three designs that failed and the exact way each one failed.

What you'll build

- `RANGE` and `END`: loop variables as ordinary i32 values, and a node that closes loops.
- Hand-written GEMM and 1-D convolution in the loop IR, with accumulators built from `ALLOC` + `AFTER`.
- `scopes`: the set of loops each node must sit inside.
- `loop_targets`: scopes closed under "encloses", which turns them into paths in a loop tree.
- `linearize`: a list scheduler with one priority heap per scope, which opens and closes loops, orders sibling loops by their data dependencies, and places accumulators so each thread gets its own.

Where this fits. Chapter 4 linearized loopless programs with a topological sort and rendered them to C. This chapter replaces that sort with the real `linearize` from `uopc/linearize.py`, and the renderer gains `for` loops. Chapter 6 will generate exactly these loop programs automatically from shapes and movement ops, so everything here is also what rangeify's output looks like.

5.1 RANGE and END

`uopc/uop.py · RangeArg`

```
@dataclass(frozen=True, order=True)
class RangeArg:
  idx: int                          # loops nest in increasing idx order
  kind: AxisType = AxisType.LOOP
```

uopc/uop.py · range_

```
def range_(end: int | UOp, idx: int, kind: AxisType = AxisType.LOOP) -> UOp:
  return UOp(Ops.RANGE, (end if isinstance(end, UOp) else UOp.const(int(end)),), RangeArg(idx,
kind))
```

`RANGE` has one source, the loop's end (an i32 value, usually a constant), and an arg `RangeArg(idx, kind)`. Its *value* is the loop counter: a `RANGE` with end 4 is an i32 that takes the values 0, 1, 2, 3. That is the whole trick. A loop variable is a value like any other, so `i*8 + j` is ordinary arithmetic, the simplifier of Chapter 3 works on it unchanged, and bounds analysis knows `i` lies in `[0, end-1]`.

`idx` gives each loop an identity and a nesting order: **loops nest in increasing `idx` order**. If loops 0 and 1 are ever open at the same time, loop 1 is inside loop 0. `kind` is an `AxisType`. In this chapter every loop is a plain `LOOP`, plus one `THREAD` in section 5.7. The others are for later chapters:

uopc/uop.py · AxisType

```
class AxisType(Enum):
  LOOP = auto()      # a plain serial for-loop
  REDUCE = auto()    # a for-loop that a reduction accumulates over
  UPCAST = auto()    # fully unrolled (week 6). uopc unrolls *during* rangeify (Opts.upcast), so
no RANGE of
                     # this kind is ever built; the name stays for the course's vocabulary
  THREAD = auto()    # parallel across CPU threads (week 6)
  GLOBAL = auto()    # GPU blockIdx (week 7)
  LOCAL = auto()     # GPU threadIdx (week 7)
  def __repr__(self): return f"AxisType.{self.name}"
  __hash__ = object.__hash__
```

`END` closes loops. Its sources are `(passthrough, *ranges)`: `x.end(i, j)` is a node that means "`x`, with loops `i` and `j` finished". It has dtype `void`, and it is how a loop's body becomes a dependency of whatever comes after the loop. A store inside a loop over `k` is *one* node in the graph, but it executes `K` times. `store.end(k)` is the event "all `K` of them have happened".

Because UOps are hash consed, a `RANGE` *is* its `(end, idx, kind)`. `range_(8, 0)` called twice gives the same loop. That is what you want when you mean the same loop, and a bug when you meant two (see Pitfalls).

Here is the first test of week 3 that uses loops, a 2-D elementwise kernel that adds the first element of each row to every element of the row:

tests/test_w03_loops.py · test_loop_invariant_code_is_hoisted

```python
def test_loop_invariant_code_is_hoisted():
  N = 8
  A, C = buf(0, N * N), buf(1, N * N)
  i, j = range_(N, 0), range_(N, 1)
  row = A.index(i * N).load()            # depends only on i
  s = sink(C.index(i * N + j).store(row + A.index(i * N + j).load()).end(i, j))
  lin = linearize(s)
  pos = {u: n for n, u in enumerate(lin)}
  assert pos[row] < pos[j]                # emitted before the j loop opens
```

```c
void add_row_head(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 8; ridx0++) {
    int alu1 = (ridx0*8);
    float val2 = data0[alu1];
    for (int ridx1 = 0; ridx1 < 8; ridx1++) {
      int alu3 = (alu1+ridx1);
      float val4 = data0[alu3];
      data1[alu3] = (val2+val4);
    }
  }
}
```

The renderer turns a `RANGE` into a `for` header and an `END` into closing braces (one per range it closes). The interesting part is *where* everything else went. `row = A[i*N]` depends on `i` only, so it was computed once per row, outside the `j` loop. `i*8` is computed once per row too and reused by both indices. Nobody asked for this hoisting. It falls out of the linearizer's rules, and the test checks it.

5.2 Scope: which loops a node must sit inside

A node has to be inside loop `r` if it uses the value of `r`, directly or through any chain of sources. It can leave loop `r` once something has `END`ed `r`. That gives three rules, the *scope* of a node:

```
scope(RANGE r)       = scope(end) | {r}
scope(END x, *rs)    = scope(x) - rs
scope(anything else) = union of scope(srcs)
```

`uopc/linearize.py · scopes`

```python
def scopes(sink: UOp) -> dict[UOp, frozenset[UOp]]:
  sc: dict[UOp, frozenset[UOp]] = {}
  for u in sink.toposort():
    s = frozenset().union(*(sc[x] for x in u.src))
    if u.op is Ops.RANGE: s = s | {u}
    if u.op is Ops.END: s = s - set(u.src[1:])
    sc[u] = s
  return sc
```

One pass over the topological order computes every scope, since a node's scope depends only on its sources'. For the row-add kernel: `i` has scope `{i}`, `i*8` has `{i}`, the load `A[i*8]` has `{i}`, `i*8 + j` has `{i, j}`, and the final store has `{i, j}`. The `END(store, i, j)` has scope `{}`, and so does the `SINK`.

Scope is a *lower bound* on placement: a node with scope `{i}` must be inside loop `i`. Could it go deeper, inside `j` too? It would compute the same value, just eight times as often. So the natural rule is "as shallow as possible": place each node exactly inside the loops of its scope. That is what made `A[i*8]` land outside the `j` loop. Section 5.5 shows that this natural rule is not quite right, and why.

5.3 Writing GEMM by hand

`C = A @ B` with `A` of shape `(N, K)`, `B` of shape `(K, M)`:

```
for i in range(N):
  for j in range(M):
    acc = 0
    for k in range(K):
      acc = acc + A[i*K + k] * B[k*M + j]
    C[i*M + j] = acc
```

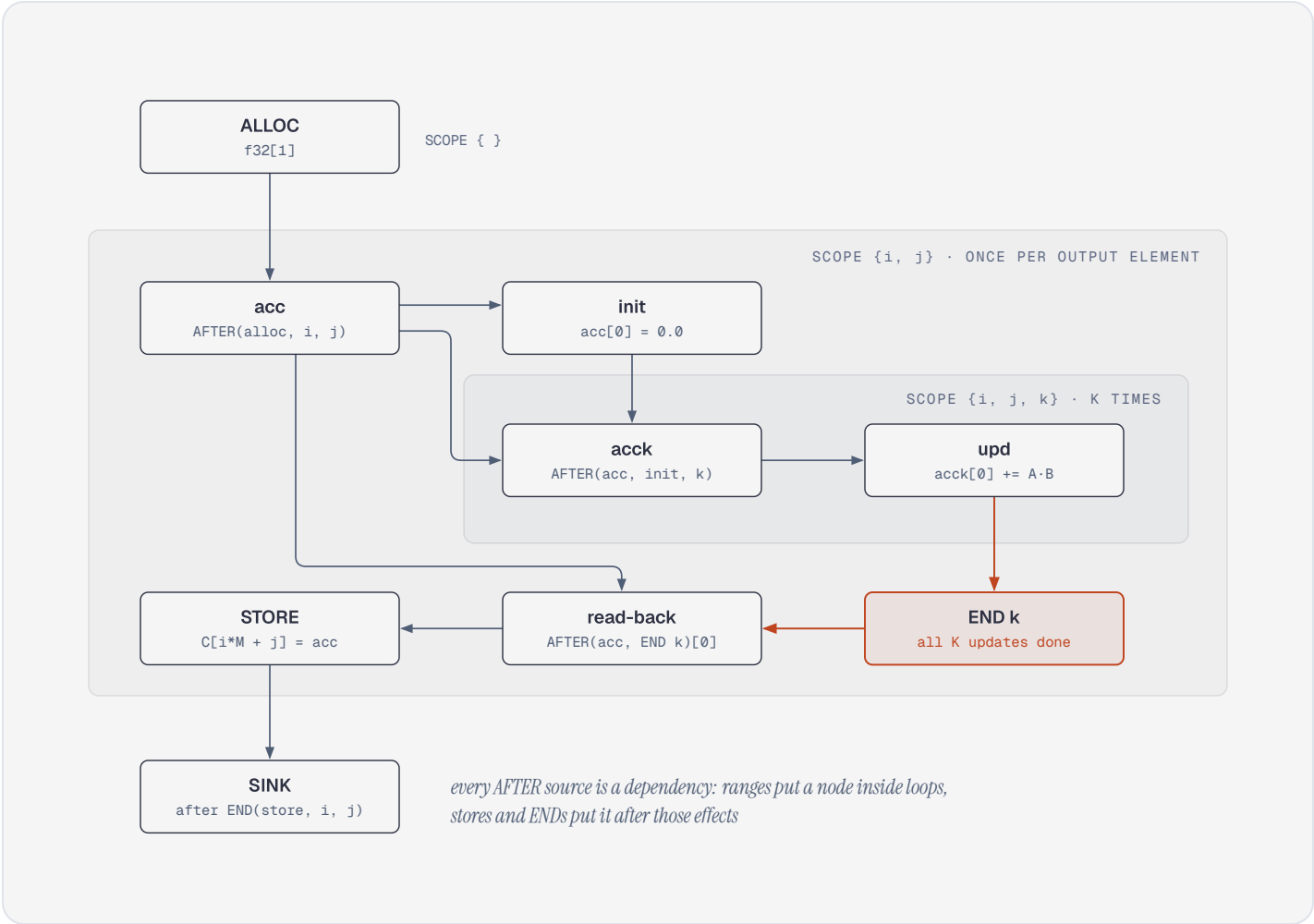
The accumulator is the interesting part. It is mutable state that is reset for every `(i, j)`, updated `K` times, and read once after the `k` loop. In our IR mutable state is memory, so `acc` is a one-element `ALLOC`, and every "after" in that description becomes an `AFTER`. Here is the week-3 test's GEMM:

`tests/test_w03_loops.py · gemm`

```
def gemm(N, M, K):
  A, B, C = buf(0, N * K), buf(1, K * M), buf(2, N * M)
  i, j, k = range_(N, 0), range_(M, 1), range_(K, 2)
  acc = UOp(Ops.ALLOC, (), ParamArg(0, dtypes.f32, (1,))).after(i, j)
  init = acc.index(0).store(UOp.const(0.0))
  acck = acc.after(init, k)
  upd = acck.index(0).store(acck.index(0).load() + A.index(i * K + k).load() * B.index(k * M +
j).load())
  return sink(C.index(i * M + j).store(acc.after(upd.end(k)).index(0).load()).end(i, j))
```

Line by line:

1. `acc = ALLOC(f32[1]).after(i, j)`. The accumulator, *inside loops* `i` *and* `j`. `AFTER` with ranges as dependencies is how you put memory into a scope: `scope(acc) = {i, j}`, so everything built on `acc` happens per `(i, j)`.
2. `init = acc[0].store(0.0)`. The reset. Its scope is `{i, j}`, so it runs once per output element, before anything that depends on it.
3. `acck = acc.after(init, k)`. The accumulator, after the reset, inside loop `k`. Scope `{i, j, k}`.
4. `upd = acck[0].store(acck[0].load() + A[i*K + k] * B[k*M + j])`. The update. It loads and stores through `acck`, so it is ordered after `init` and sits inside `k`.
5. `upd.end(k)`. The event "all `K` updates have happened". Scope `{i, j}`.
6. `acc.after(upd.end(k))[0].load()`. The final value, read after the `k` loop. Scope `{i, j}`.
7. `C[i*M + j].store(...)`, then `.end(i, j)` closes the outer loops, and `sink(...)` makes it a program.



The test runs it against numpy and checks that there are exactly three loops:

tests/test_w03_loops.py · test_gemm_loops

```
def test_gemm_loops():
  N, M, K = 17, 9, 33
  a, b = np.random.randn(N, K).astype(np.float32), np.random.randn(K, M).astype(np.float32)
  c = np.zeros((N, M), np.float32)
  src = run_c(gemm(N, M, K), {0: a, 1: b, 2: c})
  np.testing.assert_allclose(c, a @ b, atol=1e-4)
  assert src.count("for (") == 3
```

Here is the C for `gemm(4, 3, 5)`:

```
void gemm(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    for (int ridx1 = 0; ridx1 < 3; ridx1++) {
      float acc0;
      acc0 = 0.0f;
      for (int ridx2 = 0; ridx2 < 5; ridx2++) {
        float val1 = acc0;
        float val2 = data0[((ridx0*5)+ridx2)];
        float val3 = data1[((ridx2*3)+ridx1)];
        acc0 = (val1+(val2*val3));
      }
      float val4 = acc0;
      data2[((ridx0*3)+ridx1)] = val4;
    }
  }
}
```

Every line traces back to a node. `float acc0;` is the `ALLOC`, declared inside `j` (section 5.7 explains why there). `acc0 = 0.0f;` is `init`. The four lines inside `ridx2` are the update: load, two input loads, store. `float val4 = acc0;` is the load through `acc.after(upd.end(k))`, after the closing brace that `END(k)` produced. The accumulator is a scalar, so gcc keeps it in a register for the whole `k` loop.

The convolution test has the same shape, with one output loop instead of two and a sliding input index `o + k`:

tests/test_w03_loops.py · test_conv1d_loops

```
def test_conv1d_loops():
  # out[o] = sum_k x[o + k] * w[k]
  n, kw = 20, 5
  X, W, O = buf(0, n), buf(1, kw), buf(2, n - kw + 1)
  o, k = range_(n - kw + 1, 0), range_(kw, 1)
  acc = UOp(Ops.ALLOC, (), ParamArg(0, dtypes.f32, (1,))).after(o)
  init = acc.index(0).store(UOp.const(0.0))
  acck = acc.after(init, k)
  upd = acck.index(0).store(acck.index(0).load() + X.index(o + k).load() * W.index(k).load())
  s = sink(O.index(o).store(acc.after(upd.end(k)).index(0).load()).end(o))
  x, w, out = np.random.randn(n).astype(np.float32), np.random.randn(kw).astype(np.float32),
np.zeros(n - kw + 1, np.float32)
  run_c(s, {0: x, 1: w, 2: out})
  np.testing.assert_allclose(out, np.correlate(x, w, "valid"), atol=1e-5)
```

```
void conv1d(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 16; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    for (int ridx1 = 0; ridx1 < 5; ridx1++) {
      float val1 = acc0;
      float val2 = data0[(ridx0+ridx1)];
      float val3 = data1[ridx1];
      acc0 = (val1+(val2*val3));
    }
    float val4 = acc0;
    data2[ridx0] = val4;
  }
}
```

Once you have written one reduction in this style, every reduction is the same five lines: allocate in the output loops, initialize, accumulate in the reduce loop, end it, read back. Chapter 6's `REDUCE` lowering generates exactly these five lines.

5.4 What the linearizer must do

Given a `SINK`, produce a list of UOps such that:

1. **Topological:** every node comes after its sources (this includes every `AFTER` ordering edge).
2. **Scoped:** every node is emitted while all loops of its scope are open.
3. **Nested:** loops open and close like a stack. A `RANGE` pushes, an `END` pops the innermost ones.
4. **Shallow:** a node is not put inside loops it does not need, or the work is multiplied for nothing (and an accumulator reset placed too deep destroys the reduction).
5. **Ordered siblings:** if two loops are both inside `i` and neither inside the other, the one whose results the other reads must run first.

Constraint 2 says "at least these loops". Constraint 4 says "at most these loops". If you could always satisfy both with equality, i.e. "emit each node with exactly its scope open", linearizing would be easy. GEMM shows you can't. Here is every node of `gemm(4, 3, 5)` in final order, with its scope and the *target* that the final linearizer actually places it at (the table comes from a scratch script that calls `scopes` and `loop_targets` and prints sources as `%n`):

```
%0  BUFFER                scope={}       target={}
%1  CONST                 scope={}       target={}
%2  CONST                 scope={}       target={}
%3  CONST                 scope={}       target={}
%4  CONST                 scope={}       target={}
%5  CONST                 scope={}       target={}
%6  BUFFER                scope={}       target={}
%7  BUFFER                scope={}       target={}
%8  RANGE  %1             scope={i}      target={}
%9  MUL    %8 %2          scope={i}      target={i}
%10 MUL    %8 %5          scope={i}      target={i}
%11 RANGE  %2             scope={j}      target={i}
%12 ADD    %9 %11         scope={i,j}    target={i,j}
%13 INDEX  %0 %12         scope={i,j}    target={i,j}
%14 ALLOC                 scope={}       target={}
%15 AFTER  %14 %8 %11     scope={i,j}    target={i,j}
%16 INDEX  %15 %3         scope={i,j}    target={i,j}
%17 STORE  %16 %4         scope={i,j}    target={i,j}
%18 RANGE  %5             scope={k}      target={i,j}
%19 AFTER  %15 %17 %18    scope={i,j,k}  target={i,j,k}
%20 INDEX  %19 %3         scope={i,j,k}  target={i,j,k}
%21 LOAD   %20            scope={i,j,k}  target={i,j,k}
%22 ADD    %10 %18        scope={i,k}    target={i,j,k}
%23 INDEX  %6 %22         scope={i,k}    target={i,j,k}
%24 LOAD   %23            scope={i,k}    target={i,j,k}
%25 MUL    %18 %2         scope={k}      target={i,j,k}
%26 ADD    %25 %11        scope={j,k}    target={i,j,k}
%27 INDEX  %7 %26         scope={j,k}    target={i,j,k}
%28 LOAD   %27            scope={j,k}    target={i,j,k}
%29 MUL    %24 %28        scope={i,j,k}  target={i,j,k}
%30 ADD    %21 %29        scope={i,j,k}  target={i,j,k}
%31 STORE  %20 %30        scope={i,j,k}  target={i,j,k}
%32 END    %31 %18        scope={i,j}    target={i,j}
%33 AFTER  %15 %32        scope={i,j}    target={i,j}
%34 INDEX  %33 %3         scope={i,j}    target={i,j}
%35 LOAD   %34            scope={i,j}    target={i,j}
%36 STORE  %13 %35        scope={i,j}    target={i,j}
%37 END    %36 %8 %11     scope={}       target={}
%38 SINK   %37            scope={}       target={}
```

Look at `%24`, the load `A[i*K + k]`. Its scope is `{i, k}`: it does not depend on `j`. But loop `k` only exists inside loop `j` (the reduction runs once per output element), so there is no moment when `i` and `k` are open and `j` is not. The load *must* be placed in `{i, j, k}`, one level deeper than its scope. The same holds for `k*3` (`%25`, scope `{k}`) and `B[k*3 + j]` (`%28`, scope `{j, k}`). Scope is a set of loops. What we need is a *position in the nest*, and those are not the same thing. Note also `%18`, the `RANGE k` itself: its scope is `{k}`, but it must be opened while `i` and `j` are open. The difference between the scope and the target columns is the whole story of this chapter.

5.5 Four designs, three failures

This section retells development note 8. The failed designs are worth studying: each is the first thing a good programmer would try, and each fails on a program from this course. To make the failures concrete, here is a deliberately simple scheduler (a scratch file, not in the repo) that emits, at each step, the first ready node that is "legal here", preferring plain nodes, then opening loops, then closing them. Its notion of "legal" is a parameter:

```
from uopc.uop import Ops
from uopc.linearize import scopes

def naive_linearize(sink, fits):
  """First tries at a scheduler. Walk the toposort and emit the first ready node that is legal
here."""
  topo, sc = sink.toposort(), scopes(sink)
  out, stack, done = [], [], set()
  def legal(u):
    if u.op is Ops.RANGE: return sc[u] - {u} <= set(stack)                        # its end is
computable here
    if u.op is Ops.END: return list(u.src[1:]) == stack[len(stack) - len(u.src) + 1:]  # closes
the innermost loops
    return fits(sc[u], set(stack))
  while len(out) < len(topo):
    ready = [u for u in topo if u not in done and all(s in done for s in u.src) and legal(u)]
    # plain nodes first, then open a loop, then close one
    u = min(ready, key=lambda u: (u.op is Ops.RANGE, u.op is Ops.END), default=None)
    if u is None:
      stuck = [(x.op.name, sorted(r.arg.idx for r in sc[x])) for x in topo if x not in done and
all(s in done for s in x.src)]
      raise RuntimeError(f"deadlock: open loops {[r.arg.idx for r in stack]}, ready {stuck}")
    out.append(u); done.add(u)
    if u.op is Ops.RANGE: stack.append(u)
    if u.op is Ops.END: del stack[len(stack) - len(u.src) + 1:]
  return out

exact = lambda scope, open_loops: scope == open_loops    # attempt (a)
subset = lambda scope, open_loops: scope <= open_loops   # attempt (b)
```

Attempt (a): emit a node when its scope equals the open loops

This is "as shallow as possible" taken literally. It fails on GEMM exactly as the table above predicts:

```
--- attempt (a): scope == open loops, on GEMM
deadlock: open loops [0, 1, 2], ready [('ADD', [0, 2]), ('MUL', [2])]
```

With loops `i, j, k` open, the index `i*5 + k` (scope `{i, k}`) and `k*3` (scope `{k}`) are ready but never legal: their scope never equals the stack. Nothing else can run until they do, so the scheduler is stuck.

Attempt (b): emit a node when its scope is a subset of the open loops

The obvious fix: "inside at least these loops". GEMM now linearizes and runs. But legality alone no longer determines placement. A node with scope `{i}` is legal at depth `{i}`, `{i, j}` and `{i, j, k}`, and *which* it gets depends on what the scheduler happens to try first. Correctness now rests on scheduling luck.

Here is where the luck runs out. Subtract each row's sum from the row: `out[i, j] = x[i, j] - sum_k x[i, k]`. The reduction loop `k` and the output loop `j` are *siblings* inside `i`. The program, with `j` given the smaller `idx`:

```
def sub_rowsum(N, M):
  X, OUT = buf(0, N * M), buf(1, N * M)
  i, j, k = range_(N, 0), range_(M, 1), range_(M, 2)
  acc = UOp(Ops.ALLOC, (), ParamArg(0, dtypes.f32, (1,))).after(i)
  init = acc.index(0).store(UOp.const(0.0))
  acck = acc.after(init, k)
  upd = acck.index(0).store(acck.index(0).load() + X.index(i * M + k).load())
  total = acc.after(upd.end(k)).index(0).load()
  return sink(OUT.index(i * M + j).store(X.index(i * M + j).load() - total).end(i, j))
```

Attempt (b) opens `j` first (it is ready, and nothing says not to), then finds that the store inside `j` needs `total`, which needs the `k` loop, which is legal to open inside `j`. On a 2x3 matrix of ones:

```
void sub_rowsum_b(float* restrict data0, float* restrict data1) {
  float acc0;
  for (int ridx0 = 0; ridx0 < 2; ridx0++) {
    int alu1 = (ridx0*3);
    acc0 = 0.0f;
    for (int ridx1 = 0; ridx1 < 3; ridx1++) {
      int alu2 = (alu1+ridx1);
      float val3 = data0[alu2];
      for (int ridx2 = 0; ridx2 < 3; ridx2++) {
        float val4 = acc0;
        float val5 = data0[(alu1+ridx2)];
        acc0 = (val4+val5);
      }
      float val6 = acc0;
      data1[alu2] = (val3+(val6*-1.0f));
    }
  }
}
```

```
attempt (b): [[-2.0, -5.0, -8.0], [-2.0, -5.0, -8.0]]
uopc:        [[-2.0, -2.0, -2.0], [-2.0, -2.0, -2.0]]
```

Every row sum should be 3, so every output should be `1 - 3 = -2`. Attempt (b) nests the reduction inside the output loop, so the accumulator, reset once per row, keeps accumulating across `j`: the sums come out 3, 6, 9. (With `max` instead of `sum` the bug would hide, since `max` is idempotent. That is how bugs like this survive testing.) Even if the reset were moved inside `j`, the program would recompute the row sum `M` times, turning O(N·M) work into O(N·M²).

Attempt (c): order sibling loops by data dependency

The fix for (b) seemed clear: decide loop order up front. If any node that needs loop `A` depends on any node that needs loop `B`, then `B` must be opened (and finished) before `A`. In `sub_rowsum` the output store (needs `j`) depends on the update (needs `k`), so `k` comes first. Good.

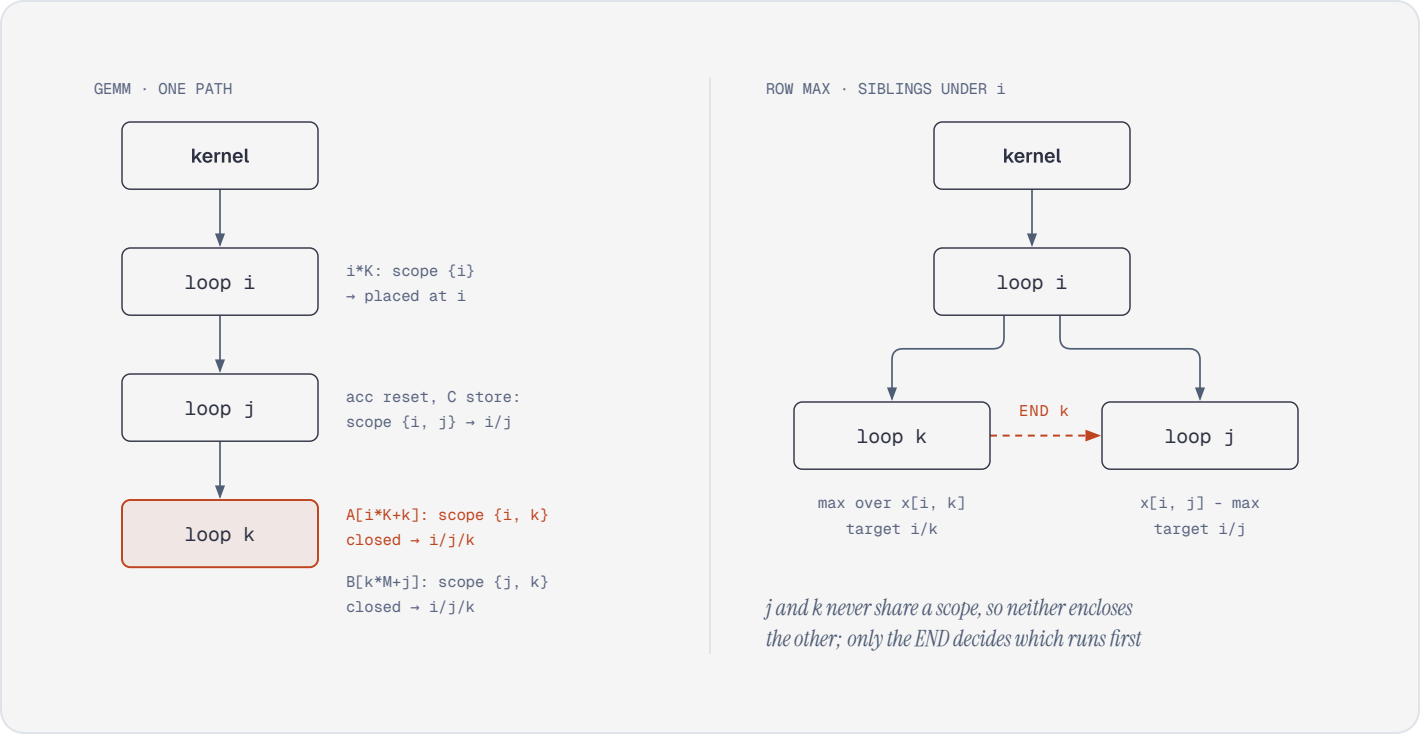
But "depends on any node that needs `B`" is far too broad. In GEMM, the load `B[k*M + j]` depends on `k*M`, a pure function of `k`, and the update depends on `A[i*K + k]`. By this rule, everything inside the `i` loop that touches `k` "depends on loop `k`", and `k` was forced outermost, deadlocking or producing nonsense nests. Data dependency on a loop *variable* says a node is *inside* the loop. Only a dependency on the loop's `END`, its finished results, says the loop must come *before*.

Attempt (d): scopes as paths in the loop tree

The final design separates the two questions that (a) to (c) kept mixing up.

Nesting comes from scopes. If two ranges `a < b` (by `idx`) ever appear together in some scope, then `b` is nested inside `a`, and that holds *everywhere*: `b` is a single loop, it has one position in the nest, and that position is inside `a`. So every node that uses `b` must also sit inside `a`, whether or not it mentions `a`. Define `enc(b)`, the loops enclosing `b`, as every smaller range that `b` shares any scope with, and *close* each scope under `enc`. The closed scope, sorted by `idx`, is the node's *target*: a path from the root of the loop tree, and therefore exactly the stack of open loops when the node should run.

For GEMM, scope `{i, j, k}` appears (the update), so `enc(j) = {i}` and `enc(k) = {i, j}`. The load `A[i*K + k]` has scope `{i, k}`, which closes to `{i, j, k}`, the target in the table. In `sub_rowsum`, `j` and `k` never share a scope, so neither encloses the other: they are siblings in the tree, both children of `i`.



Sibling order comes only from `END`s. When the scheduler must open one of several sibling loops, it looks for pending nodes that need loop `A` and that (transitively, through pending nodes) read the `END` of a sibling candidate `B`. Those block `A`: `B` goes first. Reading `k`'s *variable* never blocks anything. Reading `k`'s *results*, which is only possible through `END(k)`, does.

With nesting fixed by the tree and siblings ordered by `END`s, "emit a node exactly when the open loops equal its target" is now always possible. That is attempt (a)'s rule again, applied to targets instead of scopes. The deadlock is gone because targets are, by construction, positions that exist in the nest.

5.6 The final linearizer

The module docstring is a summary of everything above:

uopc/linearize.py:1-25

```
"""Weeks 2-3: turning a graph into a straight-line list of instructions with loops.

A graph has no order; C does. Linearizing means picking a topological order
that also respects *scope*: the set of loops a node must sit inside.

    scope(RANGE r)       = scope(end) | {r}
    scope(END x, *rs)    = scope(x) - rs
    scope(anything else) = union of scope(srcs)

Loops nest in increasing RangeArg.idx order. If two ranges a < b ever appear in
one scope, b is nested inside a, so *every* node that uses b must also sit inside a.
Closing scopes under "encloses" turns each scope into a path in the loop tree:
a sorted tuple that is exactly the stack of open loops when the node runs.

Then we list-schedule:
  1. emit a ready node whose scope is exactly the open loops
  2. else open a loop that pending nodes need next (sibling loops whose END
     another sibling's nodes read must go first)
  3. else close the innermost loop
ALLOCs are declared in the deepest scope shared by all their users (so each CPU
thread gets its own private accumulator).
"""
from __future__ import annotations
import heapq
from uopc.uop import UOp, Ops
```

Targets

uopc/linearize.py · _key

```
def _key(s) -> tuple[UOp, ...]: return tuple(sorted(s, key=lambda r: r.arg))
```

uopc/linearize.py · loop_targets

```python
def loop_targets(topo: list[UOp], sink: UOp) -> dict[UOp, tuple[UOp, ...]]:
  sc = scopes(sink)
  # enclosing loops of each range: every smaller range it ever shares a scope with
  enc: dict[UOp, set[UOp]] = {}
  for s in set(sc.values()):
    t = _key(s)
    for j, r in enumerate(t): enc.setdefault(r, set()).update(t[:j])
  def close(s):
    out, todo = set(s), list(s)
    while todo:
      for a in enc.get(todo.pop(), ()):
        if a not in out: out.add(a); todo.append(a)
    return _key(out)
  target = {u: close(sc[u]) for u in topo}
  for u in topo:
    if u.op is Ops.RANGE: target[u] = close(sc[u] - {u} | enc.get(u, set()))
  return target
```

`enc` is built from the distinct scopes: for each scope, sorted by `idx`, each range is enclosed by all the ranges before it in that scope. `close` is a transitive closure (if `k` is inside `j` and `j` is inside `i`, then `k` is inside `i`) with an explicit worklist. Every node's target is its closed scope.

`RANGE` nodes are special: the target of loop `r` is where `r` is *opened*, which is its parent's path, so it is the closure of its enclosing loops (plus its end's scope, if the end is itself loop-dependent), without `r` itself. In the GEMM table, `RANGE k` has target `{i, j}`: open `k` when `i` and `j` are the open loops.

The scheduler

`uopc/linearize.py · linearize`

```
def linearize(sink: UOp) -> list[UOp]:
  topo = sink.toposort()
  order = {u: i for i, u in enumerate(topo)}
  target = loop_targets(topo, sink)
  users: dict[UOp, list[UOp]] = {u: [] for u in topo}
  for u in topo:
    for s in set(u.src): users[s].append(u)
  for u in topo:
    if u.op is Ops.ALLOC and u.addrspace is AddrSpace.SHARED: target[u] = ()   # shared memory:
whole block, top of kernel
    elif u.op is Ops.ALLOC and users[u]:
      ts = [target[x] for x in users[u]]
      n = 0
      while all(len(t) > n and t[n] is ts[0][n] for t in ts): n += 1
      target[u] = ts[0][:n]

  waiting = {u: len(set(u.src)) for u in topo}
  ready: set[UOp] = set()                          # ready RANGE / END nodes
  heaps: dict[tuple, list[tuple[int, UOp]]] = {}   # ready everything-else, bucketed by scope
  pending, stack, out = set(topo), [], []

  def make_ready(u: UOp):
    if u.op in (Ops.RANGE, Ops.END): ready.add(u)
    else: heapq.heappush(heaps.setdefault(target[u], []), (order[u], u))

  def emit(u: UOp):
    out.append(u); pending.discard(u); ready.discard(u)
    for x in users[u]:
      waiting[x] -= 1
      if waiting[x] == 0: make_ready(x)

  for u in topo:
    if waiting[u] == 0: make_ready(u)
  while pending:
    cur = tuple(stack)
    # 1. anything that belongs exactly here
    if heaps.get(cur):
      emit(heapq.heappop(heaps[cur])[1]); continue
    # 2. open a loop that pending nodes need next
    need = {u: target[u][len(cur)] for u in pending
            if u.op is not Ops.RANGE and len(target[u]) > len(cur) and target[u][:len(cur)] ==
cur}
    cands = {r for r in need.values() if r in ready and target[r] == cur}
    if cands:
      # if nodes needing loop A read (through an END) the results of sibling loop B, open B first
      ended: dict[UOp, set[UOp]] = {}
      blocked: set[UOp] = set()
      for u in topo:
        if u not in pending: continue
        e = set(u.src[1:]) & cands if u.op is Ops.END else set()
        for x in u.src:
          if x in pending: e |= ended.get(x, set())
```

```
        ended[u] = e
        if u in need and e - {need[u]}: blocked.add(need[u])
      r = min(cands - blocked or cands, key=lambda r: r.arg)
      emit(r); stack.append(r); continue
    # 3. close the innermost loop(s)
    ends = [u for u in ready if u.op is Ops.END and target[u] == cur[:len(cur) - (len(u.src) -
1)]
            and set(u.src[1:]) == set(cur[len(cur) - (len(u.src) - 1):])]
    if ends:
      e = min(ends, key=order.__getitem__)
      emit(e); del stack[len(stack) - (len(e.src) - 1):]; continue
    stuck = sorted(pending, key=order.__getitem__)[:5]
    raise LinearizeError(f"can't schedule; open loops {[r.arg.idx for r in cur]}, stuck at "
                         f"{[(u.op, [r.arg.idx for r in target[u]]) for u in stuck]}")
  if stack: raise LinearizeError(f"RANGE without END: loops {[r.arg.idx for r in stack]} are
never closed")
  return out
```

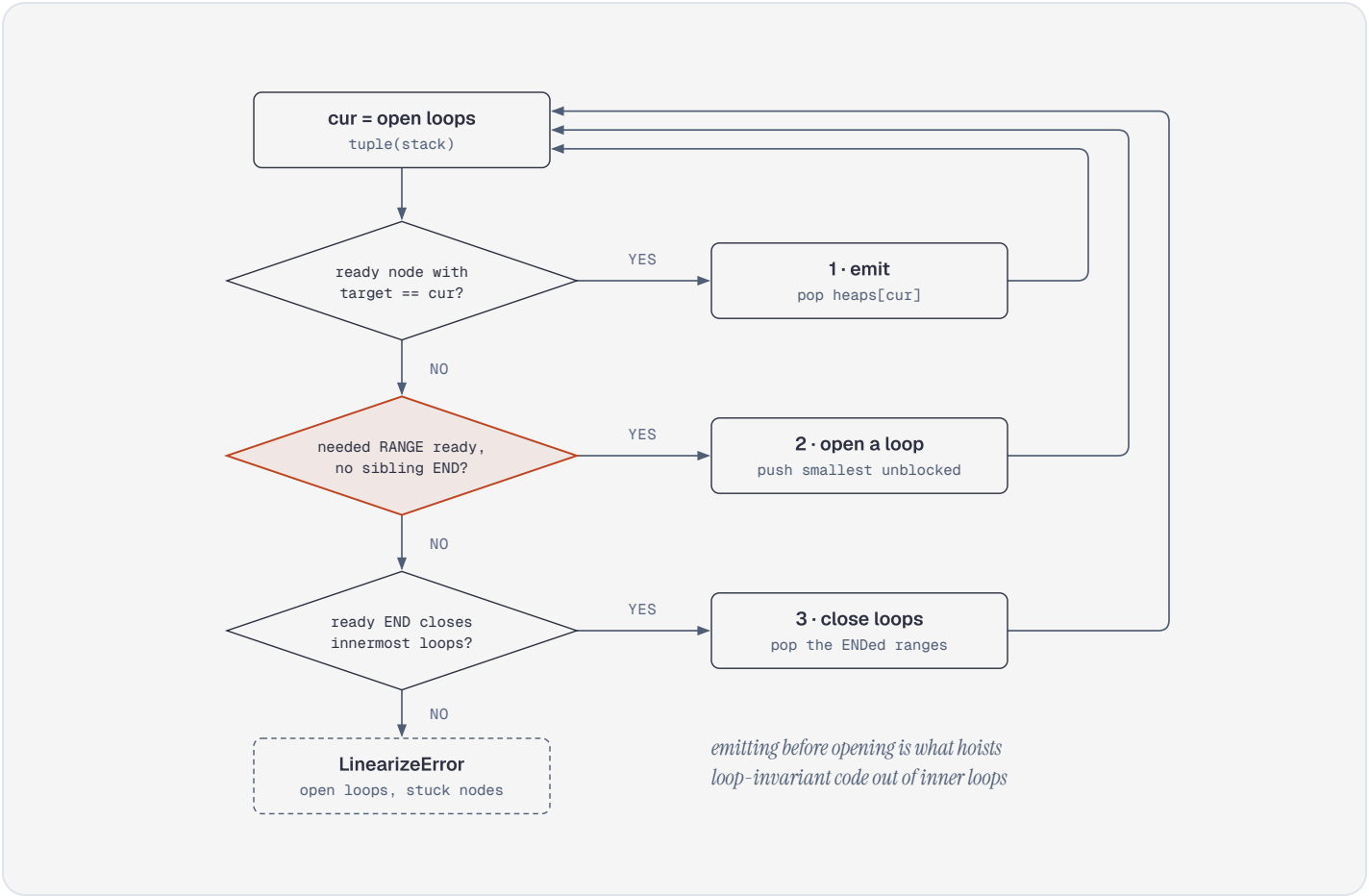
This is *list scheduling*, the standard algorithm compilers use to order instructions in a basic block: keep a set of *ready* nodes (all sources emitted), repeatedly pick one by priority, emit it, and update readiness of its users. The `waiting` dict counts unemitted distinct sources per node, so readiness updates are O(1) per edge. Two refinements make it a *loop* scheduler.

One heap per target. Ready ordinary nodes go into `heaps[target]`, a min-heap ordered by toposort position. The heap for the current stack of open loops holds exactly the nodes that may run *now*. Toposort order as the priority keeps the output close to the order in which the program was written, which keeps the C readable and deterministic. Ready `RANGE`s and `END`s go into a separate set, since they change the stack.

Three steps, in priority order. Each iteration looks at the current stack `cur`:

1. **Emit.** If anything is ready at exactly `cur`, emit the first one. Doing this before opening loops is what makes hoisting happen: everything that can run at the current depth runs before the scheduler goes deeper.
2. **Open.** Otherwise, find which loop the pending work needs next. For every pending non-`RANGE` node whose target extends `cur`, `need[u]` is the next loop on its path. Candidates are those that are ready and whose own target is `cur` (they open here). Then the sibling rule: walk the pending nodes in topological order, propagating `ended[u]`, the set of candidate loops whose `END` `u` transitively reads. A node that needs loop `A` but has read the `END` of a different candidate `B` blocks `A`. Open the unblocked candidate with the smallest `idx`.
3. **Close.** Otherwise, if a ready `END` closes exactly the innermost loops of the stack (and its own target is what remains), emit it and pop.

If none applies, the program cannot be scheduled, and the error lists the open loops and the first few stuck nodes with their targets.



Watching it run

The renderer sees only the final list, but the list records every decision. A scratch script replays `linearize(gemm(4, 3, 5))`, classifying each node by the step that must have produced it (a `RANGE` is an *open*, an `END` a *close*, anything else an *emit*) and tracking the stack. Constants and buffers, all emitted at the start with an empty stack, are left out:

```
2 open   %8  RANGE  %1            open after: (i)
1 emit   %9  MUL    %8 %2         open after: (i)
1 emit   %10 MUL    %8 %5         open after: (i)
2 open   %11 RANGE  %2            open after: (i, j)
1 emit   %12 ADD    %9 %11        open after: (i, j)
1 emit   %13 INDEX  %0 %12        open after: (i, j)
1 emit   %14 ALLOC                open after: (i, j)
1 emit   %15 AFTER  %14 %8 %11    open after: (i, j)
1 emit   %16 INDEX  %15 %3        open after: (i, j)
1 emit   %17 STORE  %16 %4        open after: (i, j)
2 open   %18 RANGE  %5            open after: (i, j, k)
1 emit   %19 AFTER  %15 %17 %18   open after: (i, j, k)
1 emit   %20 INDEX  %19 %3        open after: (i, j, k)
1 emit   %21 LOAD   %20           open after: (i, j, k)
1 emit   %22 ADD    %10 %18       open after: (i, j, k)
1 emit   %23 INDEX  %6 %22        open after: (i, j, k)
1 emit   %24 LOAD   %23           open after: (i, j, k)
1 emit   %25 MUL    %18 %2        open after: (i, j, k)
1 emit   %26 ADD    %25 %11       open after: (i, j, k)
1 emit   %27 INDEX  %7 %26        open after: (i, j, k)
1 emit   %28 LOAD   %27           open after: (i, j, k)
1 emit   %29 MUL    %24 %28       open after: (i, j, k)
1 emit   %30 ADD    %21 %29       open after: (i, j, k)
1 emit   %31 STORE  %20 %30       open after: (i, j, k)
3 close  %32 END    %31 %18       open after: (i, j)
1 emit   %33 AFTER  %15 %32       open after: (i, j)
1 emit   %34 INDEX  %33 %3        open after: (i, j)
1 emit   %35 LOAD   %34           open after: (i, j)
1 emit   %36 STORE  %13 %35       open after: (i, j)
3 close  %37 END    %36 %8 %11    open after: ()
1 emit   %38 SINK   %37           open after: ()
```

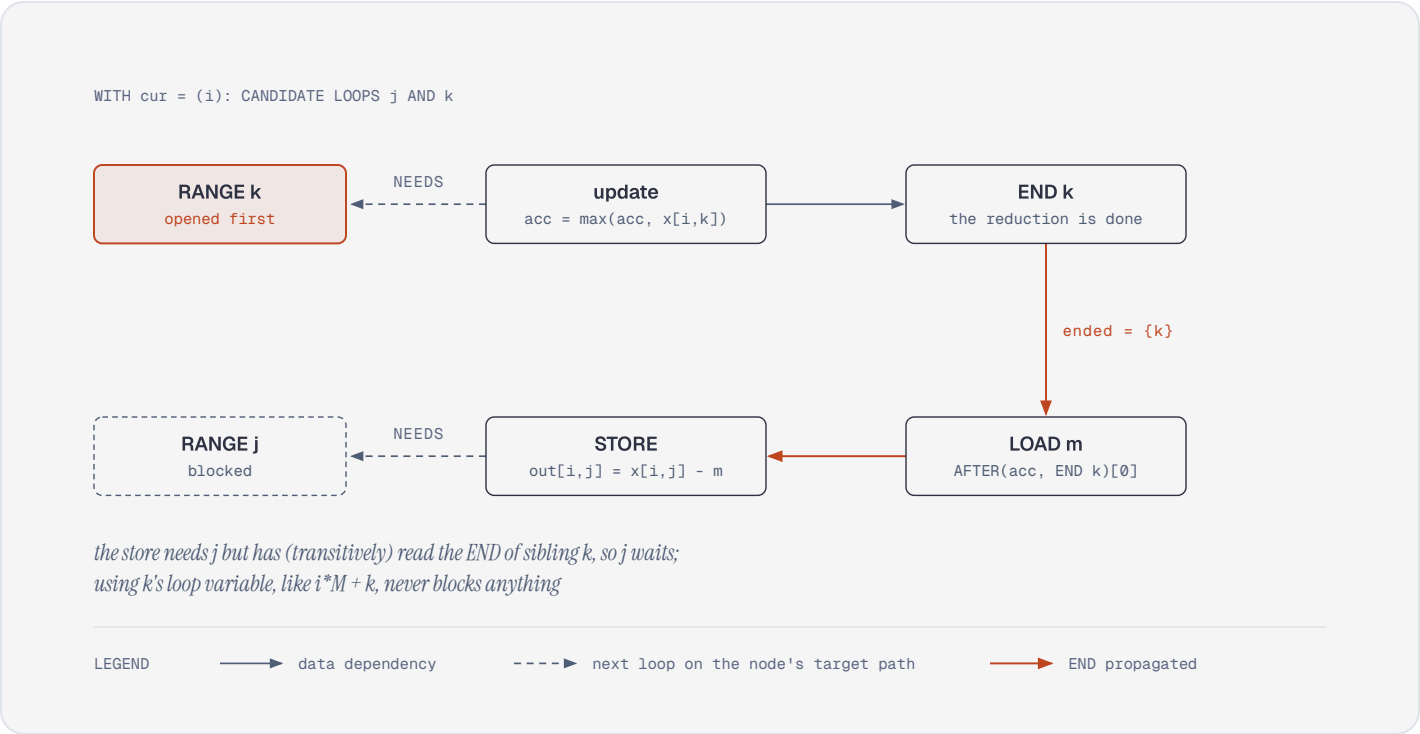
Read it as a conversation between the three steps. With only `i` open, step 1 finds `i*3` and `i*5` (`%9`, `%10`) and emits them; those are the hoisted row offsets. Nothing else has target `(i)`, so step 2 opens `j`, the only loop anything needs next. Inside `(i, j)`, step 1 drains everything with that target: the output address, the `ALLOC` (placed here by the common-prefix rule), the accumulator view and the reset store. Only when that heap is empty does step 2 open `k`. That ordering is what puts `acc0 = 0.0f;` before the `k` loop and not inside it. Inside `(i, j, k)` the body runs in toposort order. The update store makes `END(k)` ready, step 1 has nothing left at `(i, j, k)`, and step 3 closes `k`. Back at `(i, j)`, the read-back and the store to `C` become ready and are emitted, then `END(i, j)` closes two loops at once, and the `SINK` is last.

Here is the sibling rule on the row-max version of the program above, `out[i, j] = x[i, j] - max_k x[i, k]`. Again `j` (idx 1) has a smaller `idx` than `k` (idx 2), so "smallest `idx` first" alone would open `j`:

```python
def sub_rowmax(N, M):
  X, OUT = buf(0, N * M), buf(1, N * M)
  i, j, k = range_(N, 0), range_(M, 1), range_(M, 2)     # j (output) has a *smaller* idx than k
(reduce)
  acc = UOp(Ops.ALLOC, (), ParamArg(0, dtypes.f32, (1,))).after(i)
  init = acc.index(0).store(UOp.const(-math.inf))
  acck = acc.after(init, k)
  upd = acck.index(0).store(acck.index(0).load().maximum(X.index(i * M + k).load()))
  m = acc.after(upd.end(k)).index(0).load()            # the row max, readable only after the k
loop
  return sink(OUT.index(i * M + j).store(X.index(i * M + j).load() - m).end(i, j))
```

```c
void sub_rowmax(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    int alu1 = (ridx0*4);
    float acc0;
    acc0 = -INFINITY;
    for (int ridx2 = 0; ridx2 < 4; ridx2++) {
      float val2 = acc0;
      float val3 = data0[(alu1+ridx2)];
      acc0 = ((val2>val3)?val2:val3);
    }
    float val4 = acc0;
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      int alu5 = (alu1+ridx1);
      float val6 = data0[alu5];
      data1[alu5] = (val6+(val4*-1.0f));
    }
  }
}
```

With `i` open, both `j` and `k` are candidates. The final store needs `j` and reads `m`, which reads `END(..., k)`, so `ended[store] = {k}` and `j` is blocked. The scheduler opens `k`, runs the reduction, closes it, emits the read-back `val4 = acc0` (target `{i}`, step 1), and only then opens `j`. The test against numpy passes with error `0.0`. Rangeify produces this structure for every softmax and layer norm, and Chapter 11 depends on it.



Cost

Each node enters and leaves a heap once: O(n log n). Step 2 scans the pending nodes, O(n), but it runs only when a loop is about to be opened, which happens once per `RANGE`, and kernels have few loops. For the kernels in this course linearization takes milliseconds. It is not where compile time goes (Chapter 14 profiles that).

5.7 Where accumulators live, and threads

An `ALLOC` has no sources, so its scope is `{}`, and by the rules so far it would be declared at the top of the function. For a single-threaded program that is fine: the reset `acc0 = 0.0f;` sits in the right place regardless of where the variable was declared. With threads it is a bug. The linearizer therefore overrides the `ALLOC`'s target: it is declared in the *deepest scope common to all its users*, the longest common prefix of their targets (development note 9). In GEMM the only user of the `ALLOC` is `acc = ALLOC.after(i, j)`, with target `{i, j}`, so `float acc0;` appears inside the `j` loop.

Why it matters: in Chapter 10 the outer loop becomes a `THREAD` axis, rendered as an OpenMP parallel loop. Mark `i` as `AxisType.THREAD` in the GEMM and the C becomes:

```
void gemm_omp(float* restrict data0, float* restrict data1, float* restrict data2) {
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 64; ridx0++) {
    int alu1 = (ridx0*64);
    for (int ridx1 = 0; ridx1 < 64; ridx1++) {
      float acc0;
      acc0 = 0.0f;
      for (int ridx2 = 0; ridx2 < 64; ridx2++) {
        float val2 = acc0;
        float val3 = data0[(alu1+ridx2)];
        float val4 = data1[((ridx2*64)+ridx1)];
        acc0 = (val2+(val3*val4));
      }
      float val5 = acc0;
      data2[(alu1+ridx1)] = val5;
    }
  }
}
```

In OpenMP, variables declared inside the parallel loop's body are private to each thread, and variables declared outside it are shared. Declared inside the `j` loop, every thread has its own `acc0`. Declared at function scope, all 16 threads would update one variable at once: a data race. A scratch experiment moves the declaration to function scope by editing the generated C, and runs both versions on a 256x256x256 GEMM:

```
-O3 -march=native -fopenmp   inside j loop   max error 4.96e-05
-O3 -march=native -fopenmp   function scope  max error 4.96e-05
-O0 -fopenmp                 inside j loop   max error 4.96e-05
-O0 -fopenmp                 function scope  max error 85.8
```

At `-O3` the race is invisible, because gcc keeps the accumulator in a register for the whole `k` loop and the shared variable is barely touched. At `-O0` every access goes to memory and the threads trample each other. A data race is undefined behavior whether or not today's optimizer hides it, so the declaration has to be in the right place by construction, not by luck.

(GPU shared memory is the opposite case: an `ALLOC` in `SHARED` space must be visible to the whole thread block, so the linearizer pins it to the empty scope, the top of the kernel. See Chapter 13 and development note 24.)

5.8 Loop-invariant code motion, for free

Classic compilers have a pass called *loop-invariant code motion* (LICM): find computations inside a loop whose operands do not change across iterations and move them before the loop. Our linearizer never runs such a pass, yet it hoists. Placement by target *is* LICM: a node is emitted at the shallowest depth its target allows, and step 1 runs before step 2, so everything that can run before a loop opens does.

Predict the C for this kernel before reading it:

```
X, O = buf(0, 4 * 8), buf(1, 4 * 8)
i, j = range_(4, 0), range_(8, 1)
s = sink(O.index(i * 8 + j).store(X.index(i * 8 + j).load() * X.index(j).load() +
X.index(i).load()).end(i, j))
```

```c
void predict(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    for (int ridx1 = 0; ridx1 < 8; ridx1++) {
      int alu2 = ((ridx0*8)+ridx1);
      float val3 = data0[alu2];
      float val4 = data0[ridx1];
      data1[alu2] = ((val3*val4)+val1);
    }
  }
}
```

`X[i]` (scope `{i}`) is hoisted out of `j`. `X[j]` (scope `{j}`) is not hoisted out of `i`, even though it does not depend on `i`: `j` is nested inside `i`, so the closure puts it at `{i, j}`. Getting it out of the `i` loop would mean computing all of `X[0..7]` first into a temporary, which is a *different program* (more memory, one more loop), not a placement decision. That is the job of the optimizer in Chapter 10, not the linearizer. The same limit explains why `A[i*K + k]` in GEMM is reloaded for every `j`.

Two things make this cheap trick robust. Hash consing means a value used in several places is one node, placed once. And the simplifier of Chapter 3 removes spurious dependencies before the linearizer sees them: an index like `((i*8 + j)//8)` that *mentions* `j` but equals `i` would otherwise pin a node, or an accumulator, one loop too deep (development note 12, Chapter 6).

5.9 Rendering loops

The renderer from Chapter 4 needs only three additions for loops, all visible in `CRenderer.render`: a `RANGE` prints a loop header and increases the indentation depth, an `END` prints one closing brace per range it closes, and every name in the loop body is declared inside the braces, which C scopes correctly for free. The header itself is a hook:

uopc/render_c.py · CRenderer.render_range

```
def render_range(self, u: UOp, name: str, end: str) -> list[str]:
  pragma = ["#pragma omp parallel for"] if u.arg.kind is AxisType.THREAD else []
  return pragma + [f"for (int {name} = 0; {name} < {end}; {name}++) {{"]
```

uopc/render_c.py · CRenderer.is_loop

```
def is_loop(self, u: UOp) -> bool:
  """On a CPU every range is a for loop -- GPU ranges too, which makes this an emulator for GPU
kernels."""
  return True
```

A `THREAD` range gets an OpenMP pragma. Everything else is a plain counted loop with the range's end rendered as an expression, so a symbolic end (Chapter 15) needs no change here. `is_loop` exists for the GPU renderer of Chapter 12: there, a `GLOBAL` or `LOCAL` range is not a loop at all but a hardware index (`blockIdx.x`, `threadIdx.x`), so it opens no braces and its `END` closes none. On the CPU every range is a loop, including GPU-kind ranges, which is how the C backend doubles as an emulator for GPU kernels.

Because the linearizer guarantees proper nesting, the renderer never has to think about structure. It prints the list in order and keeps a depth counter. Every structural decision was made in one place.

5.10 Why a flat graph, and not structured loops?

There is an obvious alternative design. MLIR's `scf.for`, LLVM's loop nests, Halide's loop trees and TVM's TIR all represent a loop as a *region*: an op that owns a nested body, the way a `for` statement owns its block. With regions, nesting is explicit and there is nothing to linearize. Why does this course (and tinygrad) use a flat graph with `RANGE` and `END` instead?

Rewrites see through loops. In a flat graph `i*8 + j` is an ordinary expression whose leaves happen to be `RANGE`s. The simplifier, CSE through hash consing, and every pattern rule work across loop boundaries with no special cases. With regions, an expression inside a loop body and the same expression outside it are different objects in different blocks, and "hoist this out of the loop" is a transformation someone has to write.

Placement is derived, not stored. In a region-based IR every rewrite that changes what an expression depends on must also decide where the result lives. In the flat graph, rangeify (Chapter 6) and the optimizer (Chapter 10) can rewrite indices, split ranges and unroll loops without ever thinking about placement. The linearizer recomputes the best placement at the end.

The cost is this chapter. Structure has to be recovered, which needs scopes, the closure under "encloses", the sibling rule and the `ALLOC` placement rule, plus error reporting for programs that have no valid nesting. Region-based IRs pay instead with verifier rules ("a value defined inside a region cannot be used outside it") and with passes that move code between regions. Neither choice is free. The flat graph fits a compiler whose main activity is rewriting index expressions, which is what an ML compiler mostly does.

5.11 Implementing week 3

A build order that keeps every step testable:

1. **RANGE and END in `_infer`.** `RANGE` is i32 with an i32 end, `END` is void. Add `range_` and `.end()`. Bounds of a `RANGE` go into `_bounds` (Chapter 3).
2. **`scopes`.** Write it and print the scope column for GEMM. It must match the table in section 5.4.
3. **Targets.** Write `enc` and `close`. Print the target column. Every target must be a prefix-closed path (sorted by `idx`), and `A[i*K + k]` must be at `(i, j, k)`.
4. **A scheduler without siblings.** Steps 1, 2 (smallest `idx`, no blocking) and 3. The row-add and GEMM kernels and both week-2 tests (which have no loops) should pass. Check the hoisting test.
5. **`ALLOC` placement.** The deepest common prefix of the users' targets. GEMM's `float acc0;` must be inside `j`.
6. **The sibling rule.** Write `sub_rowsum` with `j` given the smaller `idx` and watch it fail, then add the `ended` propagation until it passes. Test with `sum`, not `max`.
7. **Errors.** Make every way of getting stuck raise `LinearizeError` with the open loops and the stuck nodes.
8. **The renderer.** Loop headers, braces and indentation. Compile and run against numpy, always.

At each step, print the linearized list next to the generated C. Nearly every linearizer bug is visible in one of those two printouts before it is visible in a wrong number.

Pitfalls

Exact scope matching deadlocks (note 8a). A node's scope is a set; its position is a path. `A[i*K + k]` has scope `{i, k}`, and no moment in GEMM has exactly `{i, k}` open. Close scopes under "encloses" first.

Subset matching relies on luck (note 8b). "Scope is a subset of the open loops" makes placement depend on scheduler tie-breaking. It nested a reduction inside its sibling and gave `[-2, -5, -8]` for `[-2, -2, -2]` (section 5.5).

Ordering siblings by any data dependency (note 8c). Reading a loop's *variable* means being inside it. Only reading through its `END` means coming after it. Mixing the two forced `k` outermost because of `k*N`.

Accumulators declared too shallow (note 9). An `ALLOC` at function scope is shared by every OpenMP thread. The race may be invisible at `-O3` and catastrophic at `-O0` (section 5.7). Declare in the deepest scope common to the users.

Accumulator in the wrong scope. Write `acc = ALLOC(...).after(i)` instead of `.after(i, j)` in GEMM, and the reset runs once per row, not once per output element:

```
void gemm_bad(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 2; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    for (int ridx1 = 0; ridx1 < 3; ridx1++) {
      for (int ridx2 = 0; ridx2 < 4; ridx2++) {
        float val1 = acc0;
        float val2 = data0[((ridx0*4)+ridx2)];
        float val3 = data1[((ridx2*3)+ridx1)];
        acc0 = (val1+(val2*val3));
      }
      float val4 = acc0;
      data2[((ridx0*3)+ridx1)] = val4;
    }
  }
}
```

```
[[ 4.  8. 12.]
 [ 4.  8. 12.]]
```

With all-ones inputs of shape `(2, 4) @ (4, 3)` every entry should be 4. The linearizer did exactly what it was told: the reset depends only on `i`. The `AFTER` ranges on an accumulator *are* its reset frequency.

Forgetting `END`. A store inside a loop that is never `END`ed leaves the loop open. The `SINK` then sits inside the loop, the linearizer finishes with a loop still on its stack, and the renderer emits a function with a missing brace:

```
void noend(float* restrict data0, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 8; ridx0++) {
    float val1 = data0[ridx0];
    data2[ridx0] = val1;
}
```

gcc reported `error: expected declaration or statement at end of input`: the first sign of the bug was a C syntax error, far from its cause. `linearize` now checks that its loop stack is empty when it finishes:

```
LinearizeError: RANGE without END: loops [0] are never closed
```

(`test_missing_end_is_an_error`.)

One RANGE where you meant two. Two sibling loops written as `range_(8, 0)` twice are the *same* hash-consed node. The first `END` closes it, and the second `END` can never be scheduled:

```
LinearizeError: can't schedule; open loops [], stuck at [(Ops.END, []), (Ops.SINK, [])]
```

Give every distinct loop its own `idx`. With `range_(8, 0)` and `range_(8, 1)` the same program renders as two sibling loops.

Ending only some of the loops. `store.end(i)` when the store's scope is `{i, j}` leaves `j` open with no `END` for it, and the scheduler stops:

```
LinearizeError: can't schedule; open loops [0, 1], stuck at [(Ops.END, [0, 1]), (Ops.SINK, [0,
1])]
```

Max hides accumulator bugs. A `max` reduction that accidentally accumulates across extra iterations often still gives the right answer, because `max` is idempotent. Test reductions with `sum` too.

Exercises

1. ★ **Transpose by hand.** Write a loop program that stores the transpose of an `N x M` buffer into an `M x N` buffer, run it with `run_c` for `N=3, M=5`, and compare with numpy.

2. ★ **Predict the loops.** For `out[i*8 + j] = x[i*8 + j] * w[j] + b[i]` with `i` of 4 and `j` of 8 (idx 0 and 1), which loads are inside the `j` loop? Then check against the `predict` kernel in section 5.8.

3. ★★ **Conv2d.** Write a single-channel valid 2-D convolution `out[y, x] = sum_{ky, kx} img[y+ky, x+kx] * w[ky, kx]` in the loop IR with four ranges, test it against a numpy reference, and read the C. Which loads are hoisted, and where is the accumulator declared?

4. ★★ **Row softmax by hand.** Write `softmax` over the rows of an `N x M` matrix as three sibling loops inside the row loop: a max reduction, a sum of `exp(x - max)`, and the normalized output. Use `EXP2` with `exp(x) = exp2(x * log2(e))`. Check that the generated C has the loops in the right order whatever `idx` values you give the three sibling ranges.

5. ★★ **Find the bug.** A student's GEMM has no `init`: the update is `acck[0].store(acck[0].load() + A[i*K+k] * B[k*M+j])` with `acck = ALLOC.after(i, j, k)`, and the read-back is `acc.after(upd.end(k))[0].load()`. Predict the generated C. Why might the test still pass on some runs?

6. ★★ **GEMM with k in the middle.** Write the `i, k, j` loop order: for each `i` and `k`, broadcast `A[i, k]` and update a whole row of `C`. You need a row accumulator of `M` elements (an `ALLOC` of size `M`) or direct updates of `C` through `AFTER`. Compare the generated C with the `i, j, k` version.

7. ★★★ **Implement it.** Implement `scopes`, `loop_targets` and `linearize` yourself from the rules in sections 5.2 to 5.6, without looking at `uopc/linearize.py`, until `tests/test_w03_loops.py`, the week-2 tests and the `sub_rowsum` / `sub_rowmax` programs pass with your linearizer.

8. ★★★ **Better errors.** `linearize` now names the loops left open by a forgotten `END`. Extend the message to name the *nodes* that sit in those loops, then detect the "same `RANGE` closed twice" case and suggest using distinct `idx` values.

9. ★★ **Where is the accumulator?** Rewrite GEMM so that the `ALLOC` has two users: `acc.after(i, j)` for the reset and the read-back, and a separate `acc.after(i, j, init, k)` for the update. Compute by hand the targets of both users and their longest common prefix, and predict where `float acc0;` is declared. Then make `i` a `THREAD` axis and check the C.

Solutions to ★ exercises

1.

```
N, M = 3, 5
A, B = buf(0, N * M), buf(1, M * N)
i, j = range_(N, 0), range_(M, 1)
s = sink(B.index(j * N + i).store(A.index(i * M + j).load()).end(i, j))
a = np.arange(N * M, dtype=np.float32).reshape(N, M); b = np.zeros((M, N), np.float32)
print(run_c(s, {0: a, 1: b}, "transpose")); print((b == a.T).all())
```

```
void transpose(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    for (int ridx1 = 0; ridx1 < 5; ridx1++) {
      float val1 = data0[((ridx0*5)+ridx1)];
      data1[((ridx1*3)+ridx0)] = val1;
    }
  }
}
```

and `True`. (Reads are contiguous and writes are strided. Chapter 10 discusses when that matters.)

2. `x[i*8 + j]` and `w[j]` are inside `j`. `b[i]` has scope `{i}` and is hoisted out of `j`. `w[j]` does not depend on `i` but is still inside `i`, because loop `j` is nested inside `i`. The `predict` kernel has exactly this shape, with `data0` playing all three roles.

Quiz

1. What is the scope of `END(store, k)` if `store` has scope `{i, j, k}`?
2. Why does `A[i*K + k]` in GEMM end up inside the `j` loop although it does not depend on `j`?
3. Two ranges never appear in a common scope. What is their relationship in the loop tree?
4. In the final linearizer, what is the *only* kind of dependency that decides which of two sibling loops runs first?

5. Why does step 1 (emit at the current depth) come before step 2 (open a loop)? What optimization does that ordering give you?
6. Where is an `ALLOC` of `LOCAL` space declared, and why does it matter for OpenMP?
7. What happens if you write `range_(8, 0)` for two loops that should be siblings?
8. Why can't the linearizer hoist `X[j]` out of the `i` loop in section 5.8's kernel?

Answers

1. `{i, j}`: `END` removes the ranges it closes.
2. Loop `k` is nested inside `j` (they share the update's scope `{i, j, k}`), so every node using `k` sits inside `j`. Closing the scope `{i, k}` under "encloses" gives `{i, j, k}`.
3. Neither encloses the other; they are siblings (or in unrelated subtrees).
4. A pending node that needs one loop and reads, through an `END`, the results of the other.
5. Everything that can run at the current depth runs before going deeper, which is loop-invariant code motion for free (and it emits accumulator resets before the loops that use them).
6. In the deepest scope common to all its users (the longest common prefix of their targets). Variables declared inside an OpenMP parallel loop are private per thread; declared outside, they are shared and race.
7. Hash consing makes them one `RANGE` node, so they are one loop. The second `END` can never be scheduled and linearization fails.
8. `j` is nested inside `i`, so any node using `j` is inside `i`. Computing `X[0..7]` once before the `i` loop would need a temporary buffer and an extra loop, which is a change to the program, not a placement.

Further reading

- Keith D. Cooper and Linda Torczon, *Engineering a Compiler*, 2nd ed., chapter 12 (instruction scheduling, list scheduling) and chapter 10 (loop-invariant code motion).
- Steven S. Muchnick, *Advanced Compiler Design and Implementation* (1997), chapter 17 (code scheduling).
- Randy Allen and Ken Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach* (2001): loop nests, dependence, and why legal loop orders are a dependence question.
- Jonathan Ragan-Kelley et al., "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines", PLDI 2013: separating *what* is computed from *where* in the loop nest it is computed.

- OpenMP Architecture Review Board, *OpenMP Application Programming Interface* specification, "Data-sharing attribute rules".
- tinygrad, `tinygrad/codegen/` (the linearizer and the `RANGE` / `END` design this chapter follows).

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

```
uopc/linearize.py · 128 lines

tests/test_w03_loops.py · 54 lines
```

6. Movement ops and rangeify

By the end of Chapter 5 you can write a matrix multiply in the loop IR: three `RANGE`s, an `ALLOC` for the accumulator, a handful of `AFTER`s to order the stores, and a lot of index arithmetic like `ridx0*256 + ridx2`. It works. It is also miserable. Nobody wants to write `for` loops for every `softmax` in a transformer, and nobody can write them correctly by hand for long. Tensor programmers think in *shapes*: "reshape this to `(N, 1, K)`, broadcast it against that, multiply, and sum the last axis." This chapter adds that vocabulary to the IR (four movement ops, a `REDUCE`, a general `STACK`) and then writes the compiler pass that removes it again: **rangeify** turns a graph of shaped tensors into a graph of scalars inside loops. After this chapter, "compile any model to C (slow, but correct)" (the syllabus milestone for weeks 3 and 4) is done.

What you'll build

- The movement ops `RESHAPE`, `EXPAND`, `PERMUTE`, `SHRINK`, their type rules, and the *index map* each one stands for.
- `REDUCE` (pop the last *n* axes) and `STACK` with arbitrary inputs.
- `ravel` / `unravel`, the two functions that convert between multi-dimensional and flat indices.
- `rangeify(u, idx)`: "what is the value of `u` at index `idx`?", answered by pushing the index down the graph, one rule per op.
- The `REDUCE` lowering: a fresh loop, an accumulator, its initialization and read-back.
- `lower(root, out)`: a whole shaped graph becomes a `SINK` of stores you can render and run.
- GEMM and convolution written with nothing but movement ops, and the composite ops `PAD` and `FLIP` built from primitives in `uopc/composite.py`.

Where this fits. Rangeify sits between the shape world of the frontend (the Tensor library of Chapter 9, which is a thin layer over the ops in this chapter) and the loop world of Chapter 5. Its output goes through the symbolic simplifier of Chapter 3, the linearizer, and the C renderer. In Chapter 7 we stop lowering the whole graph as one loop nest and cut it into kernels; in Chapter 10 the `Opts` that `lower()` accepts start to matter.

> **A note on listings.** Every generated C function in this chapter was produced by `render_c` on the output of `lower()`. The renderer always emits the same four-line prelude (two includes and the `floordiv_i32` / `floormod_i32` helpers from Chapter 4); it is omitted from the listings to save space. Everything else is verbatim.

6.1 Shapes are functions, not memory

Start with the question every array library has to answer: *what is a tensor?* The obvious answer is "a block of memory plus a shape". A `float[12]` buffer with shape `(3, 4)` stores element `(i, j)` at offset `4*i + j`. This is **row-major** order: the last axis is contiguous, and the **stride** of axis `k` is the product of the sizes after it, here `(4, 1)`.

A more useful answer for a compiler is: *a tensor of shape* `s` *is a function from index tuples in* `s` *to values.* The buffer above is the function

```
x(i, j) = data[4*i + j]        for 0 <= i < 3, 0 <= j < 4
```

Under this view, a **movement op** does not move any data. It builds a new function by composing the old one with an **index map**. Transposing `x` gives `t(i, j) = x(j, i)`. Taking rows 1..2 gives `s(i, j) = x(i + 1, j)`. Broadcasting a row vector `b` of shape `(1, 4)` to `(3, 4)` gives `e(i, j) = b(0, j)`. None of these needs a single byte of new memory; each is a cheap rewrite of the index before it reaches the buffer.

NumPy implements the same idea with *strides*: a view is `(base pointer, shape, strides, offset)` and a transpose just swaps two strides. Strides are fast (one multiply-add per axis) but not closed under composition: `x.T.reshape(12)` cannot be expressed as any single stride vector, so NumPy silently copies. Index maps are closed under composition, because a composition of functions is a function. The price is that some compositions contain `//` and `%`, which the symbolic simplifier then has to clean up.

Representation	Transpose	Broadcast	Slice	Reshape of a transposed view
Strides (NumPy, early tinygrad `View`)	swap strides	stride 0	offset + shape	**not expressible**, copy
Index maps (this book)	permute indices	index -> 0	index + begin	`unravel(ravel(idx))`, always works

> **Key idea:** a shaped graph describes *what* every element is, as a composition of index maps ending at buffers and constants. Lowering it to loops is the act of evaluating that composition symbolically, for a symbolic index made of loop counters.

6.2 The four movement ops

`uopc` has exactly four primitive movement ops. Each has an argument (`arg`), a type rule that computes its output shape from its input shape, and an index map. Here are the type rules, straight from the IR's type checker in `uopc/uop.py` (Chapter 1 built `_infer`; this is the week-4 part):

uopc/uop.py:286-303

```
    return dtypes.i32, (), A
  if op is Ops.RESHAPE:
    _check(prod(arg) == prod(src[0].shape), f"RESHAPE {src[0].shape} -> {arg}")
    return src[0].dtype, arg, src[0].addrspace
  if op is Ops.EXPAND:
    s = src[0].shape
    _check(len(arg) == len(s) and all(a == b or a == 1 for a, b in zip(s, arg)), f"EXPAND {s} ->
{arg}")
    return src[0].dtype, arg, src[0].addrspace
  if op is Ops.PERMUTE:
    _check(sorted(arg) == list(range(len(src[0].shape))), f"PERMUTE {arg} of {src[0].shape}")
    return src[0].dtype, tuple(src[0].shape[i] for i in arg), src[0].addrspace
  if op is Ops.SHRINK:
    s = src[0].shape
    _check(len(arg) == len(s) and all(0 <= b <= e <= n for (b, e), n in zip(arg, s)), f"SHRINK
{arg} of {s}")
    return src[0].dtype, tuple(e - b for b, e in arg), src[0].addrspace
  if op is Ops.REDUCE:
    rop, n = arg
    _check(rop in (Ops.ADD, Ops.MAX) and 0 <= n <= len(src[0].shape), f"REDUCE {arg} of
{src[0].shape}")
```

Every malformed movement op dies at construction time with a message saying what was wrong. That is the same "the type checker is the first test" philosophy as in Chapter 1, and it catches most shape bugs in the frontend long before anything is lowered.

RESHAPE

`x.reshape(s)` reinterprets the elements of `x`, in row-major order, as a tensor of shape `s`. The only requirement is that the element count is the same. To find the source element of output index `idx`, compute its position in row-major order (**ravel**) and convert that position back into an index of the source shape (**unravel**):

```
reshape(x, s)[idx] = x[unravel(ravel(idx, s), x.shape)]
```

Example: `x` has shape `(3, 4)`; `y = x.reshape((2, 6))`. Then `y[1, 2]` is flat position `1*6 + 2 = 8`, which in shape `(3, 4)` is `(8 // 4, 8 % 4) = (2, 0)`. So `y[1, 2] = x[2, 0]`.

EXPAND

`x.expand(s)` repeats size-1 axes. It requires the same rank, and each axis either keeps its size or grows from 1. The index map sends the index on an expanded axis to 0:

```
expand(x, s)[i0, ..., in] = x[j0, ..., jn]    where jk = 0 if x.shape[k] == 1 else ik
```

`EXPAND` does not add axes. To broadcast a vector of shape `(4,)` to `(3, 4)` you first `RESHAPE` it to `(1, 4)` and then `EXPAND`. Keeping the two jobs separate keeps each op's rule trivial.

PERMUTE

`x.permute(p)` reorders axes: output axis `k` is input axis `p[k]`, so the output shape is `(x.shape[p[0]], x.shape[p[1]], ...)`. To evaluate it we need the *inverse* direction: given an output index, which input index? Output coordinate `k` is input coordinate `p[k]`, so

```
permute(x, p)[idx] = x[inv]    where inv[p[k]] = idx[k] for every k
```

For a 2D transpose `p = (1, 0)` the permutation is its own inverse, which is exactly why bugs in this rule survive 2D tests (see Pitfalls).

SHRINK

`x.shrink(((b0, e0), (b1, e1), ...))` keeps the half-open window `[bk, ek)` on every axis. The output shape is `(e0 - b0, e1 - b1, ...)` and the index map is an offset:

```
shrink(x, arg)[idx] = x[i0 + b0, i1 + b1, ...]
```

`SHRINK` is slicing with step 1. There is no strided slice primitive; `Tensor._stride2d` in Chapter 9 builds `x[::2]` from `PAD`, `RESHAPE` and `SHRINK`.

6.3 Broadcasting is EXPAND plus RESHAPE

NumPy's broadcasting rule: align shapes on the right, prepend 1s to the shorter one, and on every axis the sizes must be equal or one of them must be 1; the result takes the larger size. In `uopc`, broadcasting is not an op at all. It is sugar that inserts `RESHAPE` (to prepend 1s) and `EXPAND` (to grow 1s). The `.uir` parser does it for binary ops:

uopc/uir.py · _broadcast

```
def _broadcast(a: UOp, b: UOp) -> tuple[UOp, UOp]:
  if a.shape == b.shape: return a, b
  n = max(len(a.shape), len(b.shape))
  a = a.reshape((1,) * (n - len(a.shape)) + a.shape) if len(a.shape) < n else a
  b = b.reshape((1,) * (n - len(b.shape)) + b.shape) if len(b.shape) < n else b
  out = []
  for x, y in zip(a.shape, b.shape):
    if x != y and 1 not in (x, y): raise ParseError(f"can't broadcast {a.shape} with {b.shape}")
    out.append(max(x, y))
  return (a if a.shape == tuple(out) else a.expand(out)), (b if b.shape == tuple(out) else
b.expand(out))
```

The Tensor frontend has the same logic in `_broadcast_shape` and `Tensor.broadcast_to`. Because the ALU type rule demands identical shapes (`ADD: shape mismatch (3, 4) vs () (broadcast with EXPAND)`), every broadcast is explicit in the graph. That is a feature: rangeify never has to think about broadcasting, only about `EXPAND`.

6.4 REDUCE: pop the last n axes

`x.reduce(op, n)` combines the last `n` axes with `op` (`ADD` or `MAX`) and removes them. Shape `(2, 3, 4)` reduced with `n = 2` becomes `(2,)`; with `n = 0` it is the identity; with `n = len(shape)` it becomes a scalar of shape `()`.

Why only the *last* axes? Because it is a canonical form. Any reduction over any set of axes is a `PERMUTE` that moves those axes to the end, followed by a `REDUCE`. The Tensor library does exactly that:

uopc/tensor.py · Tensor._reduce

```
  def _reduce(self, op: Ops, axis=None, keepdim=False) -> Tensor:
    axes = list(range(self.ndim)) if axis is None else [a % self.ndim for a in (axis if
isinstance(axis, (tuple, list)) else [axis])]
    keep = [i for i in range(self.ndim) if i not in axes]
    t = self.permute(keep + axes)
    r = Tensor(t.uop.reduce(op, len(axes)))
    if keepdim: r = r.reshape(tuple(1 if i in axes else s for i, s in enumerate(self.shape)))
    return r
```

One op with one argument is easier to lower, to differentiate (Chapter 8) and to pattern-match than a family of reductions over arbitrary axis sets. The permute costs nothing: it is an index map, and it disappears into the index arithmetic of the load.

A reduction needs an **identity element** to initialize its accumulator: 0 for `ADD`, and the smallest representable value for `MAX` (`-inf` for floats, `INT_MIN` for `i32`).

uopc/rangeify.py · identity

```
def identity(op: Ops, d: DType):
  if op is Ops.ADD: return 0.0 if d.is_float else 0
  return -float("inf") if d.is_float else dtypes.min(d)
```

6.5 STACK, generalized

In week 1, `STACK` could only stack constants (so the const folder could run programs). Now it can stack any same-shaped, same-dtype values: `STACK(a, b, c)` has shape `(3, *a.shape)`, and element `[k, *rest]` is element `rest` of the `k`-th source. `INDEX(x, i)` is its inverse: it pops the leading axis by selecting entry `i`. Together they make the IR able to build tensors out of pieces, which is how `PAD` will be built below.

6.6 Rangeify: computing one element

Now the main event. We have a shaped graph and want a loop program. The idea is to stop thinking about whole tensors and ask a much smaller question:

> **Key idea:** *what is the value of node* `u` *at index* `idx`*?* If `idx` is made of loop counters, the answer is a scalar expression that, placed inside those loops, computes every element.

For the root of shape `(M, N)`, create two loops `i in [0, M)` and `j in [0, N)` and ask for the root's value at `(i, j)`. Every op answers by asking its sources the same question at some *transformed* index. The index flows down the graph, from the root toward the leaves, transformed by each movement op's index map, until it hits a `BUFFER` (which answers with a `LOAD`) or a `CONST` (which answers with itself). Here is the whole pass:

`uopc/rangeify.py · rangeify`

```python
def rangeify(u: UOp, idx: tuple[UOp, ...], ctx: Ctx) -> UOp:
  """The scalar value of u at index idx."""
  key = (u, idx)
  if (r := ctx.memo.get(key)) is not None: return r
  assert len(idx) == len(u.shape), f"{u.op}: {len(idx)} indices for shape {u.shape}"
  op = u.op
  if op is Ops.CONST: r = u
  elif op is Ops.BUFFER: r = u.index(idx[0]).load()
  elif op is Ops.STAGE: r = rangeify(u.src[0], idx, ctx)      # no kernelizer: compute it inline
  elif op in ALU: r = u.replace(src=tuple(rangeify(s, idx, ctx) for s in u.src))
  elif op is Ops.INDEX:
    iv = tuple(rangeify(i, (), ctx) for i in u.src[1:])
    r = rangeify(u.src[0], iv + idx, ctx)
  elif op is Ops.STACK:
    i, rest = idx[0], idx[1:]
    r = rangeify(u.src[-1], rest, ctx)
    for k in range(len(u.src) - 2, -1, -1):
      r = (i < k + 1).where(rangeify(u.src[k], rest, ctx), r)
  elif op is Ops.RESHAPE:
    # simplify now: ((i*N + j)//N) % N mentions j but *is* i. Dependence analysis
    # (which loops a reduce must sit in) and the memo both need the truth.
    r = rangeify(u.src[0], tuple(simplify(i) for i in unravel(ravel(idx, u.shape),
u.src[0].shape)), ctx)
  elif op is Ops.EXPAND:
    r = rangeify(u.src[0], tuple(UOp.const(0) if s == 1 and n != 1 else i for i, s, n in zip(idx,
u.src[0].shape, u.shape)), ctx)
  elif op is Ops.PERMUTE:
    inv = [0] * len(idx)
    for k, p in enumerate(u.arg): inv[p] = idx[k]
    r = rangeify(u.src[0], tuple(inv), ctx)
  elif op is Ops.SHRINK: r = rangeify(u.src[0], tuple(i + b if b else i for i, (b, _) in zip(idx,
u.arg)), ctx)
  elif op is Ops.REDUCE: r = _reduce(u, idx, ctx)
  else: raise NotImplementedError(f"rangeify: {op}")
  ctx.memo[key] = r
  return r
```

Go through it branch by branch:

- **CONST**: a constant is the same at every index.
- **BUFFER**: buffers are flat (`BUFFER is flat` in the type rule), so `idx` has exactly one component. The value is `LOAD(INDEX(buf, i))`, the same address-then-load pair you used in Chapter 4.
- **ALU**: elementwise ops have no index map. `f(a, b)` at `idx` is `f(a[idx], b[idx])`. Note `u.replace(src=...)`: the op and its arg are reused, only the sources change, and the result has shape `()` because its sources do.
- **INDEX**: `x.index(i0, ...)` pops leading axes, so `x.index(i)[idx] = x[i, *idx]`. The index values themselves are scalars, rangeified at the empty index `()`.

- **STACK**: `STACK(a, b, c)[k, *rest]` is "`a[rest]` if `k < 1`, else `b[rest]` if `k < 2`, else `c[rest]`", built as a chain of `WHERE`s from the last source backward.
- **RESHAPE**, **EXPAND**, **PERMUTE**, **SHRINK**: exactly the four index maps from the previous section. The `if b else i` in `SHRINK` avoids building `i + 0` nodes for untouched axes.
- **STAGE** is the kernelizer's materialization marker (Chapter 7). Without a kernelizer there is nothing to materialize into, so it is transparent.
- **REDUCE** is the only op that creates new loops. It gets its own section.

Watching an index flow

Here is a small chain: a `(3, 4)` matrix, transposed, rows 1 and 2 of the transpose kept, times a broadcast vector of length 3.

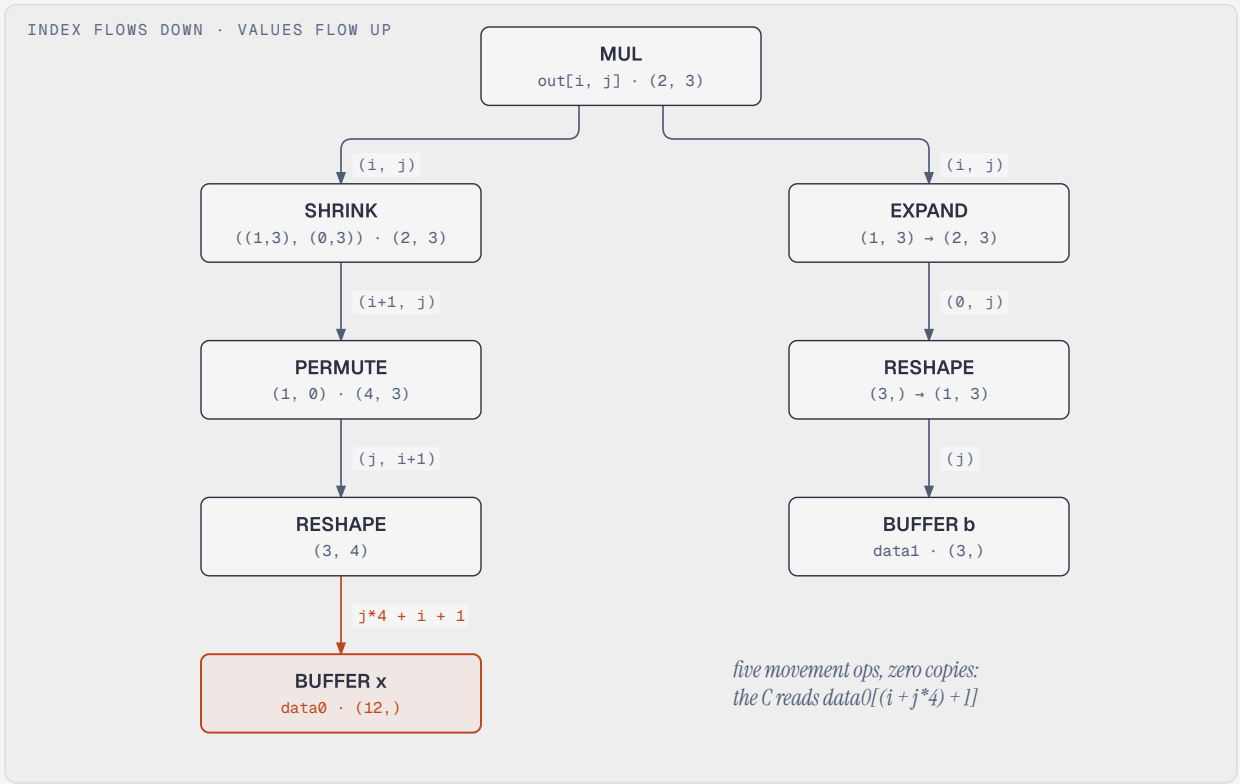
```
x = buf(0, 12).reshape((3, 4))                    # a 3x4 matrix
y = x.permute((1, 0)).shrink(((1, 3), (0, 3)))    # rows 1..2 of x^T: shape (2, 3)
b = buf(1, 3).reshape((1, 3)).expand((2, 3))
root = y * b
```

I wrapped `rangeify` in a tracer that prints each call before it recurses (the tracer is a scratch script, not repo code; `i` and `j` are `RANGE 0` and `RANGE 1`):

```
MUL      shape=(2, 3)   idx=(i, j)
  SHRINK   shape=(2, 3)   idx=(i, j)
    PERMUTE  shape=(4, 3)   idx=((i+1), j)
      RESHAPE  shape=(3, 4)   idx=(j, (i+1))
        BUFFER   shape=(12,)    idx=(((i+(j*4))+1))
  EXPAND   shape=(2, 3)   idx=(i, j)
    RESHAPE  shape=(1, 3)   idx=(0, j)
      BUFFER   shape=(3,)     idx=(j)
```

Read it top to bottom: `SHRINK` adds the begin offset 1 to axis 0; `PERMUTE` swaps the two coordinates; `RESHAPE` ravels `(j, i+1)` in shape `(3, 4)` into `j*4 + i + 1` (already simplified, see below). On the other branch, `EXPAND` zeroes the broadcast axis and the `RESHAPE` from `(3,)` to `(1, 3)` drops it. The generated C is the whole story in two loads:

```c
void view_chain(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 2; ridx0++) {
    for (int ridx1 = 0; ridx1 < 3; ridx1++) {
      float val1 = data0[((ridx0+(ridx1*4))+1)];
      float val2 = data1[ridx1];
      data2[((ridx0*3)+ridx1)] = (val1*val2);
    }
  }
}
```



Five movement ops, zero copies, and the C compiler sees ordinary strided loads. This is the payoff of the function view: movement ops cost nothing at run time *if* the index arithmetic simplifies.

ravel and unravel

`RESHAPE` is the only rule that needs real arithmetic. Two helpers do the work:

```
uopc/rangeify.py · ravel
```

```python
def ravel(idx: tuple[UOp, ...], shape: tuple[int, ...]) -> UOp:
  flat = UOp.const(0)
  for i, s in zip(idx, shape): flat = flat * s + i
  return flat
```

uopc/rangeify.py · unravel

```python
def unravel(flat: UOp, shape: tuple[int, ...]) -> tuple[UOp, ...]:
  out, stride = [], 1
  for s in reversed(shape):
    out.append((flat // stride) % s if s != 1 else UOp.const(0))
    stride *= s
  return tuple(reversed(out))
```

`ravel` is Horner's rule: for shape `(s0, s1, s2)`, `((0*s0 + i0)*s1 + i1)*s2 + i2`. `unravel` goes the other way, peeling off the last axis first: coordinate `k` is `(flat // stride_k) % s_k`, where `stride_k` is the product of the sizes after axis `k`. A size-1 axis always has coordinate 0, so it gets a `CONST 0` instead of a pointless `% 1`.

Both build *UOp expressions*, not numbers: the flat index is a graph of `ADD`, `MUL`, `FLOORDIV`, `FLOORMOD` over `RANGE`s. Here is what they produce for a `(4, 6)` index, printed with a small formatting helper (`i` is `RANGE 0`, `j` is `RANGE 1`):

```
ravel((i, j), (4, 6))      = ((((0*4)+i)*6)+j)
simplified                 = ((i*6)+j)
unravel(.., (4, 6))        = ['((((((0*4)+i)*6)+j)//6)%4)', '((((((0*4)+i)*6)+j)//1)%6)']
simplified                 = ['i', 'j']
unravel(.., (2, 12)) simpl = ['(((i*6)+j)//12)', '(((i*6)+j)%12)']
unravel(.., (24, 1)) simpl = ['((i*6)+j)', '0']
```

Raveling and unraveling in the *same* shape is the identity, and the symbolic rules of Chapter 3 prove it: `((i*6 + j) // 6) % 4` becomes `(i + j//6) % 4` by the floor-division rule `(a*c + b)//c = a + b//c`, then `j//6 = 0` because bounds analysis knows `0 <= j < 6`, then `i % 4 = i` because `0 <= i < 4`. Reshaping into a genuinely different shape, like `(2, 12)`, leaves real `//` and `%` in the index. Those are the cost of the function view, and they are cheap.

Memoization: rangeify is a DAG walk

The graph is a DAG, not a tree: `x * x` uses `x` twice, and `softmax` uses its input three times. Without memoization, rangeify would walk a shared subgraph once per path to it, and a chain like `a = x + x; b = a`

`+ a; c = b + b; ...` would cost 2^n calls. `ctx.memo` caches the answer per `(node, index)` pair. Because UOps are hash consed (Chapter 1), two index expressions that are structurally equal are the *same Python object*, so the tuple `idx` is a cheap, exact dictionary key. Hash consing also gives free common-subexpression elimination of the *output*: two paths that compute `LOAD(INDEX(buf0, i*4 + j))` produce one node, which the renderer emits once.

The memo is keyed on the index as well as the node. The same node evaluated at two different indices (say `y[i]` and `y[i+1]`) is two different scalar computations.

6.7 Lowering REDUCE: accumulators and loops

Every other op maps an index to an index. `REDUCE` is different: element `idx` of `x.reduce(ADD, n)` is a *sum over* all values of the popped axes. In loop language that is a new loop, an accumulator, an initialization before the loop, an update inside it, and a read after it. Chapter 5 showed how to write that by hand with `ALLOC`, `AFTER`, `STORE` and `END`. `_reduce` writes it automatically:

`uopc/rangeify.py · _reduce`

```
def _reduce(u: UOp, idx: tuple[UOp, ...], ctx: Ctx) -> UOp:
  rop, n = u.arg
  x = u.src[0]
  rshape = x.shape[len(x.shape) - n:]
  # unrolled copies of the same reduce (they differ only by constants in idx) share one loop
  rkey = (u, frozenset(ranges_in(*idx)))
  if (parts := ctx.reduce_ranges.get(rkey)) is None:
    parts = [_axis_parts(ctx, s, ctx.opts.unroll if k == n - 1 else 1, AxisType.REDUCE) for k, s
in enumerate(rshape)]
    ctx.reduce_ranges[rkey] = parts
  real = [r for r, _ in parts if r.op is Ops.RANGE]
  vals = [rangeify(x, idx + c, ctx) for c in _combos(parts)]
  while len(vals) > 1:                                    # unrolled reduce axis: a balanced tree
    vals = [UOp(rop, (vals[i], vals[i + 1])) if i + 1 < len(vals) else vals[i] for i in range(0,
len(vals), 2)]
  if not real: return vals[0]
  acc = ctx.alloc(u.dtype)
  # the accumulator lives inside the loops the output index uses...
  acc0 = acc.after(*ranges_in(*idx))
  init = acc0.index(0).store(UOp.const(identity(rop, u.dtype), u.dtype))
  # ...is re-read every iteration of the reduce loops...
  accl = acc0.after(init, *real)
  upd = accl.index(0).store(UOp(rop, (accl.index(0).load(), vals[0])))
  # ...and is read once more after the loops end
  return acc0.after(upd.end(*real)).index(0).load()
```

Ignore `_axis_parts`, `_combos` and the balanced tree for now. With default options (no unrolling, which is a week-6 feature, see Chapter 10) they create exactly one `REDUCE`-kind `RANGE` per popped axis (or `CONST 0` for a size-1 axis), and `vals` has one element: the source's value at `idx + (r0, r1, ...)`. Then the accumulator is built in five steps:

1. `acc = ctx.alloc(dtype)`: a fresh one-element `ALLOC`, rendered as a local `float acc0;`.
2. `acc0 = acc.after(*ranges_in(*idx))`: *this* accumulator lives inside every loop the output index mentions. An `AFTER` whose dependencies are `RANGE`s is how the IR says "a fresh instance per iteration of these loops". The linearizer reads it as the scope of the `ALLOC`.
3. `init = acc0.index(0).store(identity)`: write 0 (or `-inf`) into it.
4. `accl = acc0.after(init, *real)`: the accumulator *after* the init and *inside* the reduce loops. Load it, combine with the new value, store it back: `upd`.
5. `acc0.after(upd.end(*real)).index(0).load()`: after the reduce loops have ended, load the result. This is the scalar value that `REDUCE` returns to its consumer.

Why all the `AFTER`s? The IR is a pure graph with no statement order. Two `LOAD`s of `acc0[0]` with no ordering edge are the same node (hash consing!), so "the accumulator before the loop" and "the accumulator

after the loop" *must* be different nodes. `AFTER` makes them different and tells the linearizer which effects must happen first. This is the memory-ordering discipline of Chapter 4, applied mechanically.

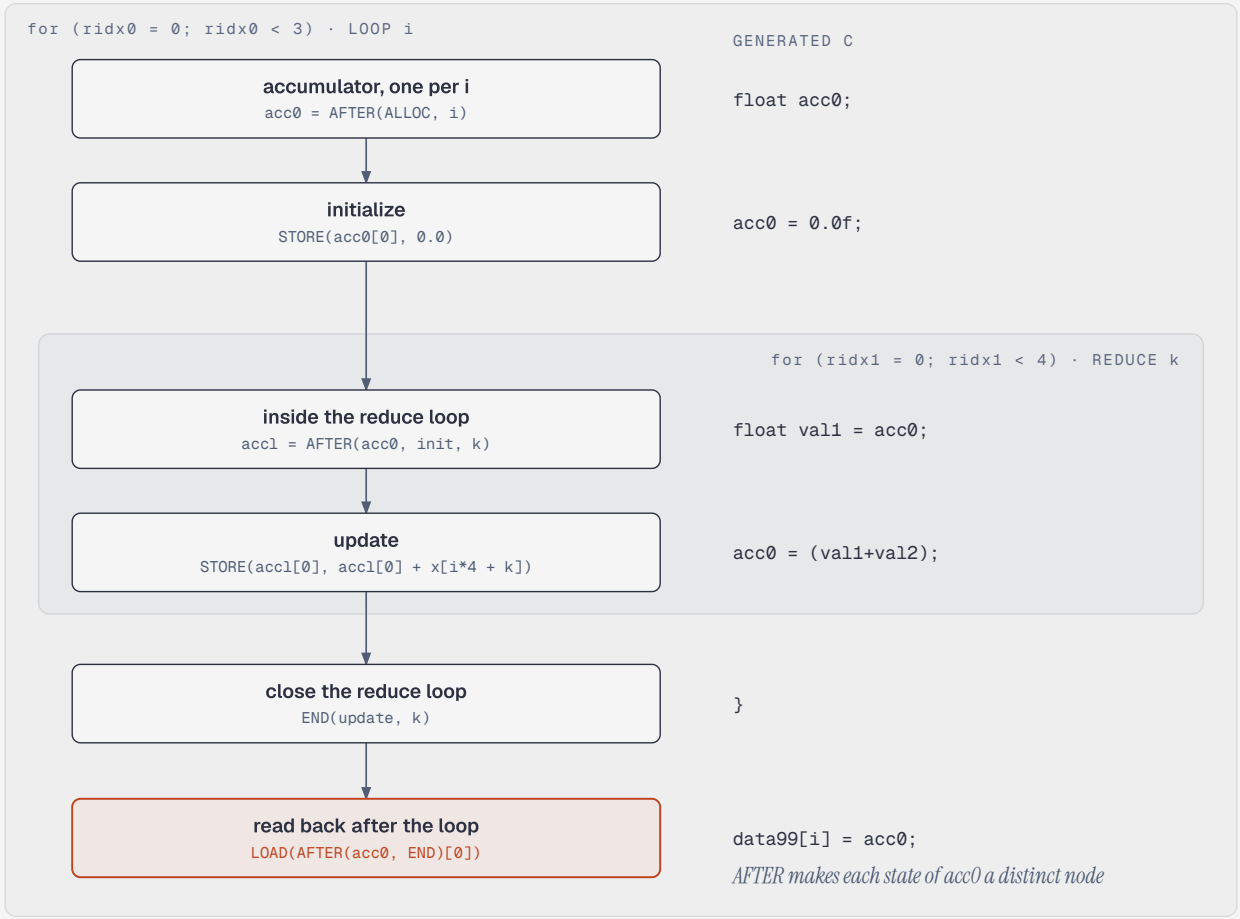
Here is a row sum, `x.reduce(ADD, 1)` on a `(3, 4)` matrix, as `.uir` after `lower()`:

```
%0 = buffer 3 : dtype=f32 slot=1
%1 = range 3 : idx=0 kind=loop
%2 = index %0, %1
%3 = alloc 1 : dtype=f32 slot=0
%4 = after %3, %1
%5 = index %4, 0
%6 = store %5, 0.0
%7 = range 4 : idx=1 kind=reduce
%8 = after %4, %6, %7
%9 = index %8, 0
%10 = load %9
%11 = buffer 12 : dtype=f32 slot=0
%12 = mul %1, 4
%13 = add %12, %7
%14 = index %11, %13
%15 = load %14
%16 = add %10, %15
%17 = store %9, %16
%18 = end %17, %7
%19 = after %4, %18
%20 = index %19, 0
%21 = load %20
%22 = store %2, %21
%23 = end %22, %1
%24 = sink %23
```

and as C:

```c
void rowsum(float* restrict data0, float* restrict data99) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      float val1 = acc0;
      float val2 = data0[((ridx0*4)+ridx1)];
      acc0 = (val1+val2);
    }
    float val3 = acc0;
    data99[ridx0] = val3;
  }
}
```

Match them up: `%4` (`after %3, %1`) is the `float acc0;` inside the `ridx0` loop; `%6` is `acc0 = 0.0f`; `%8` .. `%17` is the loop body; `%18` closes the reduce loop; `%19` .. `%21` is the read after it.



Where the accumulator lives decides the loop structure

Step 2 is the most consequential line in the file. The accumulator is placed in the loops that `idx` *mentions*, found syntactically by `ranges_in`:

`uopc/rangeify.py · ranges_in`

```
def ranges_in(*us: UOp) -> list[UOp]:
  return sorted({r for u in us for r in u.toposort() if r.op is Ops.RANGE}, key=lambda r: r.arg)
```

Consider `x - max(x, axis=1)` broadcast back over the row, the first half of a softmax:

```
x = buf(0, 12).reshape((3, 4))
m = x.reduce(Ops.MAX, 1).reshape((3, 1)).expand((3, 4))
root = x - m
```

The output loops are `i` (rows) and `j` (columns). The index reaching the `REDUCE` went through `EXPAND` (column index replaced by 0) and `RESHAPE` (`(i, 0)` in `(3, 1)` to `(i,)` in `(3,)`), so it is just `(i,)`. The accumulator lives in loop `i` only, and the linearizer makes the max loop a *sibling* of the `j` loop instead of nesting it inside:

```
void sub_max(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    int alu1 = (ridx0*4);
    float acc0;
    acc0 = -INFINITY;
    for (int ridx2 = 0; ridx2 < 4; ridx2++) {
      float val2 = acc0;
      float val3 = data0[(alu1+ridx2)];
      acc0 = ((val2>val3)?val2:val3);
    }
    float val4 = acc0;
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      int alu5 = (alu1+ridx1);
      float val6 = data0[alu5];
      data1[alu5] = (val6+(val4*-1.0f));
    }
  }
}
```

The max is computed once per row, not once per element. Note the range numbering: the output's `j` is `RANGE 1` (`ridx1`), the reduce's is `RANGE 2`, but the reduce loop comes first because the `j` loop reads its result through an `END`. That ordering decision is the linearizer's (Chapter 5, the "sibling loops" story in NOTES 8c); rangeify only has to produce the right graph.

The flip side: `ranges_in` is syntactic. If the index *mentions* `j` without depending on it, the accumulator lands inside the `j` loop and the reduce is recomputed for every `j`. That is exactly what happened in a real bug, and it is why `RESHAPE` simplifies its indices immediately (see "Index simplification at RESHAPE" below).

Unrolled copies share one loop

`ctx.reduce_ranges` caches the reduce loops per `(REDUCE node, set of ranges in idx)`. Its purpose is week 6's register tiling: when an output axis is unrolled by 4, rangeify asks for the same `REDUCE` at four

indices that differ only by constants, and those four accumulators must share *one* reduce loop (that is the whole point of a register tile). It has a pleasant side effect even without unrolling. Here two shifted views of the same row sum share a loop:

```
x = buf(0, 20).reshape((4, 5))
y = x.reduce(Ops.ADD, 1)
z = y.shrink(((0, 3),)) + y.shrink(((1, 4),))        # s[i] + s[i+1]
```

```
void neighbors(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    float acc1;
    acc1 = 0.0f;
    for (int ridx1 = 0; ridx1 < 5; ridx1++) {
      float val1 = acc0;
      int alu2 = ((ridx0*5)+ridx1);
      float val3 = data0[alu2];
      acc0 = (val1+val3);
      float val4 = acc1;
      float val5 = data0[(alu2+5)];
      acc1 = (val4+val5);
    }
    float val6 = acc0;
    float val7 = acc1;
    data1[ridx0] = (val6+val7);
  }
}
```

Two accumulators, one loop. Both copies `END` the same `RANGE`; `merge_ends` (below) merges those two `END`s into one.

6.8 lower(): from a shaped root to a SINK

`rangeify` answers "the value at `idx`". `lower` wraps it into a complete program: create the output loops, ask for the root's value, store it into the output buffer at the raveled index, close the loops, simplify.

`uopc/rangeify.py · lower`

```
def lower(root: UOp, out: UOp, ctx: Ctx | None = None) -> UOp:
  """Lower one shaped value into a SINK that stores it, element by element, into BUFFER `out`."""
  ctx = ctx or Ctx()
  o = ctx.opts
  if o.gpu:
    # each output axis gets a GLOBAL range and (optionally) a LOCAL one; neither becomes a loop
on a GPU
    gl = [_gpu_axis(ctx, s, o.local.get(k, 1), o.upcast.get(k, 1)) for k, s in
enumerate(root.shape)]
    plist = gl
  else:
    # the THREAD axis gets the lowest range index, so it is the outermost loop
    order = ([o.thread] if o.thread is not None else []) + [k for k in range(len(root.shape)) if
k != o.thread]
    tiled = {k: t for k, t in o.tile.items() if 1 < t < root.shape[k] and root.shape[k] % t == 0
and t % o.upcast.get(k, 1) == 0}
    parts: dict[int, tuple[UOp, int]] = {}
    # tiled axes: first every block loop (outer), then the loops inside a block. A block of
outputs reuses
    # the same rows of each operand, so they stay in cache while the block is computed
    outer = {k: ctx.rng(root.shape[k] // t, AxisType.THREAD if k == o.thread else AxisType.LOOP)
for k, t in tiled.items()}
    for k in order:
      f = o.upcast.get(k, 1)
      if k in tiled:
        inner = tiled[k] // f
        parts[k] = (outer[k] * inner + ctx.rng(inner) if inner > 1 else outer[k], f)
      else:
        parts[k] = _axis_parts(ctx, root.shape[k], f, AxisType.THREAD if k == o.thread else
AxisType.LOOP)
    plist = [parts[k] for k in range(len(root.shape))]
  stores = []
  for idx in _combos(plist):
    val = rangeify(root, idx, ctx)
    stores.append(out.index(ravel(idx, root.shape) if root.shape else UOp.const(0)).store(val))
  loops = ranges_in(*(b for b, _ in plist))
  body = stores[0] if len(stores) == 1 else UOp(Ops.SINK, tuple(stores))
  return simplify(merge_ends(sink(body.end(*loops))))
```

On the CPU path with default `Opts`, `_axis_parts` returns one `LOOP` range per output axis of size greater than 1 and `CONST 0` for size-1 axes (a loop of one iteration is not a loop). `_combos` then yields a single index tuple. The rest is short:

- the store address is `ravel(idx, root.shape)`: outputs are always contiguous row-major;
- `ranges_in(...)` collects the output loops, and `body.end(*loops)` closes them;
- `merge_ends` merges duplicate `END`s produced by unrolled copies (one `END` per loop, over a `SINK` of their bodies);

- `simplify` runs the symbolic rules over the whole program once more.

uopc/rangeify.py · merge_ends

```
def merge_ends(root: UOp) -> UOp:
  """Unrolled copies each END the same shared loop. Merge them into one END over a SINK."""
  while True:
    groups: dict[tuple, list[UOp]] = {}
    for u in root.toposort():
      if u.op is Ops.END: groups.setdefault(u.src[1:], []).append(u)
    dup = next((g for g in groups.values() if len(g) > 1), None)
    if dup is None: return root
    merged = UOp(Ops.END, (UOp(Ops.SINK, tuple(e.src[0] for e in dup)), *dup[0].src[1:]))
    root = root.substitute({e: merged for e in dup})
```

The `gpu` branch and the `THREAD` / `UPCAST` handling are for Chapter 10 and Chapter 12. Notice that they only change *which* ranges exist and how an output index is assembled from them (`outer * factor + c`). The rangeify rules themselves never change. That separation (what to compute versus how to map it onto the machine) is the central design decision of the compiler, borrowed from Halide.

The test helper that every week-4 test uses is `lower` plus the runtime:

tests/helpers.py · run_shaped

```
def run_shaped(root: UOp, inputs: list[np.ndarray]) -> np.ndarray:
  """Lower a shaped graph whose BUFFERs are slots 0..len(inputs)-1, run it, return the result."""
  n = max(1, prod(root.shape))
  out = buf(len(inputs), n, root.dtype)
  res = np.zeros(n, np_dtype(root.dtype))
  run_c(lower(root, out), {**dict(enumerate(inputs)), len(inputs): res})
  return res.reshape(root.shape)
```

6.9 Index simplification at RESHAPE

Look again at the `RESHAPE` branch of `rangeify`:

```
r = rangeify(u.src[0], tuple(simplify(i) for i in unravel(ravel(idx, u.shape), u.src[0].shape)),
ctx)
```

The `simplify` call is not cosmetic. `lower` simplifies everything at the end anyway, so the C would have the same arithmetic either way. What changes is the *loop structure*, because `_reduce` places accumulators according to the ranges an index mentions. To see it, take the `sub_max` example and add a flatten/unflatten pair at the top, the kind of reshape round trip Tensor code produces all the time (`x.reshape(-1)` then back):

```
x = buf(0, 16).reshape((4, 4))
rowmax = x.reduce(Ops.MAX, 1).reshape((4, 1)).expand((4, 4))
y = (x - rowmax).reshape((16,)).reshape((4, 4))
```

The index reaching the `REDUCE` is, unsimplified, `((i*4 + j) // 4) % 4`: mathematically `i`, syntactically mentioning `j`. I monkeypatched `simplify` to the identity during rangeify (and ran the real one afterwards, so the arithmetic is cleaned up but the structure is frozen):

```
void no_simplify(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    int alu1 = (ridx0*4);
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      int alu2 = (alu1+ridx1);
      float val3 = data0[alu2];
      float acc0;
      acc0 = -INFINITY;
      for (int ridx2 = 0; ridx2 < 4; ridx2++) {
        float val4 = acc0;
        float val5 = data0[(alu1+ridx2)];
        acc0 = ((val4>val5)?val4:val5);
      }
      float val6 = acc0;
      data1[alu2] = (val3+(val6*-1.0f));
    }
  }
}
```

The row max is now recomputed for every column: O(N^3) work for an O(N^2) problem. With the real code:

```
void with_simplify(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    int alu1 = (ridx0*4);
    float acc0;
    acc0 = -INFINITY;
    for (int ridx2 = 0; ridx2 < 4; ridx2++) {
      float val2 = acc0;
      float val3 = data0[(alu1+ridx2)];
      acc0 = ((val2>val3)?val2:val3);
    }
    float val4 = acc0;
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      int alu5 = (alu1+ridx1);
      float val6 = data0[alu5];
      data1[alu5] = (val6+(val4*-1.0f));
    }
  }
}
```

The same mistake in the real compiler made fused attention at N=512 take 1787 ms instead of 7 ms (NOTES 12; the story is told in full in Chapter 11). The memo benefits too: `(u, (i,))` and `(u, (((i*4+j)//4)%4,))` are different keys for the same value.

> **Key idea:** in a compiler, *syntactic* facts (which variables an expression mentions) drive structural decisions (which loop a computation lives in). Keep expressions in normal form at the moment you inspect them, not only at the end.

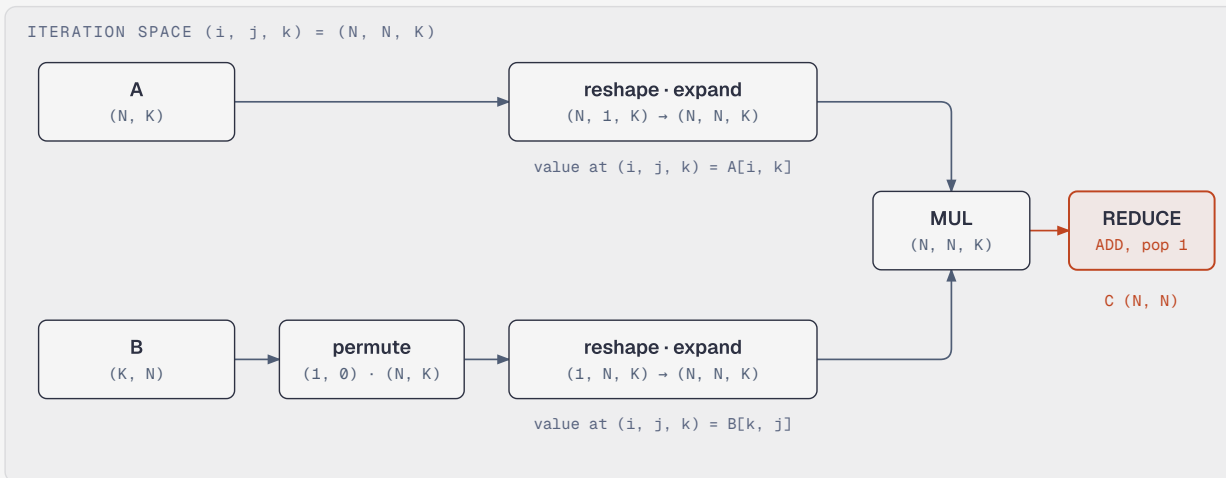
6.10 GEMM from movement ops

Now the payoff: matrix multiplication, written without a single loop. Start from the math,

```
C[i, j] = sum_k A[i, k] * B[k, j]
```

and think about the three-dimensional **iteration space** `(i, j, k)`, of shape `(N, N, K)`. The summand `A[i, k] * B[k, j]` is a function on that cube. Build each factor as a tensor of shape `(N, N, K)`:

- `A` has shape `(N, K)`. Reshape to `(N, 1, K)` (insert a `j` axis of size 1) and expand to `(N, N, K)`: the value at `(i, j, k)` is `A[i, k]` for every `j`.
- `B` has shape `(K, N)`. Permute to `(N, K)` (now indexed `[j, k]`), reshape to `(1, N, K)`, expand: the value at `(i, j, k)` is `B[k, j]` for every `i`.
- Multiply elementwise and `REDUCE(ADD, 1)` pops `k`, leaving `(N, N)`.



the syllabus reshapes A to (N, N, 1) instead: the value at (i, j, k) is A[i, j], so the sum over k is A · rowsum(B), not a matrix product

The week-4 test does exactly this in `.uir` (the parser's broadcasting sugar inserts the `EXPAND`s):

tests/test_w04_rangeify.py · test_gemm_from_movement_ops

```
def test_gemm_from_movement_ops():
  N = 16
  a, b = np.random.randn(N, N).astype(np.float32), np.random.randn(N, N).astype(np.float32)
  g = parse(f"""
  %0 = buffer {N*N} : dtype=f32 slot=0
  %1 = buffer {N*N} : dtype=f32 slot=1
  %2 = reshape %0, ({N}, 1, {N})
  %3 = reshape %1, ({N}, {N})
  %4 = permute %3, (1, 0)
  %5 = reshape %4, (1, {N}, {N})
  %6 = mul %2, %5
  %7 = reduce %6 : op=add pop=1""")
  np.testing.assert_allclose(run_shaped(g, [a.ravel(), b.ravel()]), a @ b, atol=1e-4)
```

With `N = 4`, the generated C is the textbook triple loop, with the accumulator in the `j` loop and `A`'s row offset hoisted:

```
void gemm(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    int alu1 = (ridx0*4);
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      float acc0;
      acc0 = 0.0f;
      for (int ridx2 = 0; ridx2 < 4; ridx2++) {
        float val2 = acc0;
        float val3 = data0[(alu1+ridx2)];
        float val4 = data1[((ridx2*4)+ridx1)];
        acc0 = (val2+(val3*val4));
      }
      float val5 = acc0;
      data2[(alu1+ridx1)] = val5;
    }
  }
}
```

Two loads per multiply-add, and `B` is read down a column (stride 4). That stride is a performance disaster at N=1024 (about 2–5 GFLOP/s depending on the run, NOTES 19), and fixing it is Chapter 10's job. Correctness first.

The Tensor library's `matmul` is the same construction with the reduction axis last (`(..., M, 1, K) * (..., 1, N, K)`), which also handles batches for free, because extra leading axes just ride along. See Chapter 9.

The syllabus GEMM puzzle

The course syllabus opens with this program, called `gemm_256x256.uir`:

```
%0 = buffer 65536 : dtype=f32 slot=0
%1 = buffer 65536 : dtype=f32 slot=1
%2 = reshape %0, (256, 256, 1)
%3 = reshape %1, (256, 1, 256)
%4 = mul %2, %3
%5 = reduce %4 : op=add pop=1
```

Before reading on, work out what it computes with index maps. The broadcast shape is `(256, 256, 256)`; call its axes `(i, j, k)`.

- `%2 = A.reshape(256, 256, 1)` expanded: the value at `(i, j, k)` is `A[i, j]` (the size-1 axis is `k`, so `k` is dropped).
- `%3 = B.reshape(256, 1, 256)` expanded: the value at `(i, j, k)` is `B[i, k]` (the size-1 axis is `j`).
- Reducing `k`: `C[i, j] = sum_k A[i, j] * B[i, k] = A[i, j] * (sum_k B[i, k])`.

That is `A * rowsum(B)`, an elementwise product with a broadcast row sum. It is not a GEMM (NOTES 11). The generated C (N=4) agrees: `A`'s load `val3` sits *outside* the `k` loop because its index does not mention `k`, and the loop only sums a row of `B`:

```
void syllabus(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    int alu1 = (ridx0*4);
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      int alu2 = (alu1+ridx1);
      float acc0;
      acc0 = 0.0f;
      float val3 = data0[alu2];
      for (int ridx2 = 0; ridx2 < 4; ridx2++) {
        float val4 = acc0;
        float val5 = data1[(alu1+ridx2)];
        acc0 = (val4+(val3*val5));
      }
      float val6 = acc0;
      data2[alu2] = val6;
    }
  }
}
```

The repository keeps a test that pins this down, so nobody "fixes" the oracle to match a wrong belief:

`tests/test_w04_rangeify.py · test_syllabus_gemm_is_not_a_gemm`

```
def test_syllabus_gemm_is_not_a_gemm():
  # the example in the syllabus computes A * rowsum(B): a good reminder to test everything
  N = 8
  a, b = np.random.randn(N, N).astype(np.float32), np.random.randn(N, N).astype(np.float32)
  g = parse(f"""
  %0 = buffer {N*N} : dtype=f32 slot=0
  %1 = buffer {N*N} : dtype=f32 slot=1
  %2 = reshape %0, ({N}, {N}, 1)
  %3 = reshape %1, ({N}, 1, {N})
  %4 = mul %2, %3
  %5 = reduce %4 : op=add pop=1""")
  np.testing.assert_allclose(run_shaped(g, [a.ravel(), b.ravel()]), a * b.sum(1, keepdims=True),
atol=1e-4)
```

The correct version reshapes `A` to `(N, 1, K)` and `B` *transposed* to `(1, N, K)`. (The docstring of `uopc/uir.py` uses `(256, 1, 256)` and `(1, 256, 256)` without the transpose, which computes `A @ B^T`, a perfectly good GEMM if `B` is stored transposed.) The lesson generalizes: write the index map down. "Reshape and multiply" looks right whatever the axes are.

6.11 Convolution from movement ops

A 1D valid convolution (strictly, a cross-correlation, which is what ML frameworks call convolution) with a length-`K` kernel:

```
out[i] = sum_k x[i + k] * w[k]        for 0 <= i < L - K + 1
```

The factor `x[i + k]` is a "sliding window". Each fixed `k` is a `SHRINK` of `x` (the window `[k, k + Lo)`), so a `STACK` of `K` shrinks gives a `(K, Lo)` tensor `win[k, i] = x[i + k]`:

```
L, K = 8, 3; Lo = L - K + 1
x, w = buf(0, L), buf(1, K)
win = UOp(Ops.STACK, tuple(x.shrink(((k, k + Lo),)) for k in range(K)))   # (K, Lo): win[k, i] =
x[i+k]
cols = win.permute((1, 0))                                                # (Lo, K)
out = (cols * w.reshape((1, K)).expand((Lo, K))).reduce(Ops.ADD, 1)        # (Lo,)
```

This is **im2col** in miniature: build the matrix of windows, then it is a matrix-vector product.

```
void conv1d(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 6; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    float val1 = data0[ridx0];
    float val2 = data0[(ridx0+1)];
    float val3 = data0[(ridx0+2)];
    for (int ridx1 = 0; ridx1 < 3; ridx1++) {
      float val4 = acc0;
      float val5 = data1[ridx1];
      acc0 = (val4+(((ridx1<1)?val1:((ridx1<2)?val2:val3))*val5));
    }
    float val6 = acc0;
    data2[ridx0] = val6;
  }
}
```

The `STACK` became a `WHERE` chain selecting among three loads, all hoisted out of the `k` loop. On `x = 0..7` and `w = [1, 10, 100]` it prints `[210. 321. 432. 543. 654. 765.]`, the same as `np.convolve(x, w[::-1], "valid")`. `Tensor.conv2d` (Chapter 9) is the 2D version: `KH*KW` shrinks, a `STACK`, a permute, and a matmul.

6.12 Composite ops: PAD and FLIP

The syllabus says PAD and FLIP are *composite*: not primitives, but built from the primitives. That is a design choice worth arguing about. tinygrad has both as primitive movement ops, because a primitive `PAD` lowers to a single bounds check. Making them composite keeps rangeify and autodiff smaller (four movement rules instead of six) and is a superb exercise in index maps. The cost is uglier generated code, as you will see.

Concatenation

Everything starts with concatenating two same-shaped tensors along an axis: `STACK` them (new leading axis of size 2), move that axis next to `axis`, and merge the two with a reshape.

`uopc/composite.py · concat2`

```python
def concat2(a: UOp, b: UOp, axis: int) -> UOp:
  """Concatenate two same-shaped tensors along `axis`: STACK, move the new axis next to `axis`, merge."""
  assert a.shape == b.shape
  s = UOp(Ops.STACK, (a, b))                        # (2, *shape)
  s = move_axis(s, 0, axis)                         # (..., 2, n, ...)
  shape = list(a.shape)
  shape[axis] *= 2
  return s.reshape(tuple(shape))
```

Check the index map for 1D: `s = STACK(a, b)` has shape `(2, n)`, `s[t, i]` is `a[i]` if `t = 0` else `b[i]`. Reshape to `(2n,)`: element `p` is `s[p // n, p % n]`, which is `a[p]` for `p < n` and `b[p - n]` for `p >= n`. Concatenation.

PAD

Padding with zeros after the data is "concatenate with a zero tensor of the same shape, then shrink away the excess". Padding before is the mirror image. If the padding is larger than the axis, do it in rounds, doubling each time:

uopc/composite.py · zeros_like

```python
def zeros_like(x: UOp) -> UOp:
  z = UOp.const(0, x.dtype)
  return z.reshape((1,) * len(x.shape)).expand(x.shape) if x.shape else z
```

uopc/composite.py · pad_axis

```python
def pad_axis(x: UOp, axis: int, before: int, after: int) -> UOp:
  """Zero-pad one axis by doubling with zeros and shrinking back."""
  while after > 0:
    n = x.shape[axis]
    x = _shrink_axis(concat2(x, zeros_like(x), axis), axis, 0, n + min(after, n))
    after -= min(after, n)
  while before > 0:
    n = x.shape[axis]
    x = _shrink_axis(concat2(zeros_like(x), x, axis), axis, n - min(before, n), 2 * n)
    before -= min(before, n)
  return x
```

uopc/composite.py · pad

```
def pad(x: UOp, arg: tuple[tuple[int, int], ...]) -> UOp:
  for axis, (b, a) in enumerate(arg): x = pad_axis(x, axis, b, a)
  return x
```

`pad_axis(v, 0, 1, 2)` on a length-3 vector builds this graph:

```
%0 = reshape 0.0, (1,)
%1 = expand %0, (5,)
%2 = buffer 3 : dtype=f32 slot=0
%3 = expand %0, (3,)
%4 = stack %2, %3
%5 = reshape %4, (6,)
%6 = shrink %5, ((0, 5),)
%7 = stack %1, %6
%8 = reshape %7, (10,)
%9 = shrink %8, ((4, 10),)
```

and rangeify plus the simplifier turn it into a bounds-checked select:

```
void pad1d(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 6; ridx0++) {
    int alu1 = (ridx0+4);
    int alu2 = (alu1%5);
    float val3 = data0[(alu2%3)];
    data1[ridx0] = (((alu1/5)<1)?0.0f:(((alu2/3)<1)?val3:0.0f));
  }
}
```

Two observations. First, the `/` and `%` here are C's truncating operators, which the renderer emits only because bounds analysis proved the operands non-negative (NOTES 4). Second, the load `data0[(alu2%3)]` happens *unconditionally*, even for padding positions. It is safe because the index map of a `STACK` branch is always in bounds for that branch's shape (here `% 3`), so a speculative load never reads outside the buffer. A primitive `PAD` would emit `(0 <= i-1 && i-1 < 3) ? x[i-1] : 0`, which is shorter. Exercise 9 asks you to build it.

FLIP, and the diagonal trick

Reversing an axis cannot be written with `SHRINK` and `RESHAPE` alone, since none of them maps `i` to `n - 1 - i`. The composite builds it as a matrix-vector product with the **anti-identity** `M[i, j] = [i + j == n - 1]`:

```
y[i] = sum_j x[j] * M[i, j] = x[n - 1 - i]
```

How do you build `M` from movement ops? With a classic trick. Let `u` be the one-hot vector of length `2n - 1` with a 1 at position `n - 1` (a 1 padded with `n - 1` zeros on each side). Expand it to `n + 1` rows, flatten, keep the first `n * 2n` elements, and view them as `(n, 2n)`. Element `(i, j)` of that view is at flat position `2n*i + j`, which comes from row-length-`(2n-1)` data, so it reads

```
u[(2n*i + j) mod (2n - 1)] = u[(i + j) mod (2n - 1)]       because 2n = 1 (mod 2n - 1)
                           = u[i + j]                       because i + j <= 2n - 2 for j < n
```

and `u[i + j] = 1` exactly when `i + j = n - 1`. Shrinking to the first `n` columns gives `M`. Taking a row-stride that is one longer than the data shears the data by one position per row; that is the "diagonal trick".

uopc/composite.py · anti_eye

```python
def anti_eye(n: int, dtype) -> UOp:
  """M[i, j] = (i + j == n - 1), from one-hot + the expand/reshape/shrink "diagonal trick".

  u is one-hot at n-1, length 2n-1. Expand u to (n+1, 2n-1), flatten, keep n*2n
  elements, view as (n, 2n): element (i, j) is u[(2n*i + j) mod (2n-1)] = u[i + j].
  """
  if n == 1: return UOp.const(1, dtype).reshape((1, 1))
  one = UOp.const(1, dtype).reshape((1,))
  u = pad_axis(one, 0, n - 1, n - 1)                              # (2n-1,)
  m = u.reshape((1, 2 * n - 1)).expand((n + 1, 2 * n - 1)).reshape(((n + 1) * (2 * n - 1),))
  m = m.shrink(((0, n * 2 * n),)).reshape((n, 2 * n))
  return m.shrink(((0, n), (0, n)))
```

uopc/composite.py · flip_axis

```python
def flip_axis(x: UOp, axis: int) -> UOp:
  """Reverse one axis: y[i] = sum_j x[j] * [i + j == n-1]. O(n^2), but only primitives."""
  n = x.shape[axis]
  if n == 1: return x
  xl = move_axis(x, axis, len(x.shape) - 1)                       # (..., n)
  lead = xl.shape[:-1]
  xe = xl.reshape(lead + (1, n)).expand(lead + (n, n))
  m = anti_eye(n, x.dtype).reshape((1,) * len(lead) + (n, n)).expand(lead + (n, n))
  y = (xe * m).reduce(Ops.ADD, 1)                                 # (..., n)
  return move_axis(y, len(x.shape) - 1, axis)
```

`anti_eye(4)` evaluated by the week-1 constant folder (`run` + `to_python`) is

```
[[0.0, 0.0, 0.0, 1.0], [0.0, 0.0, 1.0, 0.0], [0.0, 1.0, 0.0, 0.0], [1.0, 0.0, 0.0, 0.0]]
```

and flipping a length-3 vector generates:

```
void flip3(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    for (int ridx1 = 0; ridx1 < 3; ridx1++) {
      float val1 = acc0;
      float val2 = data0[ridx1];
      int alu3 = ((((ridx0*6)+ridx1)%5)+1);
      int alu4 = (alu3%3);
      acc0 = (val1+(val2*(((alu3/3)<1)?0.0f:(((alu4/2)<1)?(((alu4%2)<1)?1.0f:0.0f):0.0f))));
    }
    float val5 = acc0;
    data1[ridx0] = val5;
  }
}
```

Correct, and O(n^2): the anti-identity is computed with index arithmetic for every `(i, j)` and multiplied into a sum. For a course whose goal is "correct first, fast later", that is acceptable, and it keeps the primitive set small. Flipping is rare in models (it appears in the gradient of a convolution with respect to its input, which the autodiff of Chapter 8 never needs because it differentiates the im2col construction instead). Exercise 5 asks for an O(n) composite.

The week-4 tests check `PAD` and `FLIP` against NumPy, including padding larger than the axis:

tests/test_w04_rangeify.py · test_pad_flip_composites

```
def test_pad_flip_composites():
  x = np.random.randn(3, 4).astype(np.float32)
  u = buf(0, 12).reshape((3, 4))
  np.testing.assert_allclose(run_shaped(pad(u, ((1, 5), (2, 0))), [x.ravel()]), np.pad(x, ((1,
5), (2, 0))))
  np.testing.assert_allclose(run_shaped(flip_axis(u, 1), [x.ravel()]), x[:, ::-1])
  np.testing.assert_allclose(run_shaped(flip_axis(u, 0), [x.ravel()]), x[::-1])
  assert to_python(run(anti_eye(3, u.dtype))) == [[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

6.13 Testing rangeify: random movement chains

Every rule of rangeify is a few characters long, which means a typo in any of them produces plausible but wrong indices. Hand-written tests catch the cases you thought of. The best test in the week-4 file instead generates random chains of movement ops and compares against NumPy, which implements the same semantics with strides:

tests/test_w04_rangeify.py · test_random_movement_chains

```
@pytest.mark.parametrize("seed", range(5))
def test_random_movement_chains(seed):
  rng = np.random.default_rng(seed)
  x = rng.standard_normal(24).astype(np.float32)
  u, ref = buf(0, 24).reshape((2, 3, 4)), x.reshape(2, 3, 4)
  for _ in range(6):
    k = rng.integers(4)
    if k == 0:
      p = tuple(rng.permutation(len(ref.shape))); u, ref = u.permute(p), ref.transpose(p)
    elif k == 1:
      arg = tuple((int(b), int(rng.integers(b + 1, s + 1))) for s in ref.shape for b in
[rng.integers(0, s)])
      u, ref = u.shrink(arg), ref[tuple(slice(b, e) for b, e in arg)]
    elif k == 2:
      u, ref = u.reshape((1,) + u.shape), ref.reshape((1,) + ref.shape)
      u, ref = u.expand((2,) + u.shape[1:]), np.broadcast_to(ref, (2,) + ref.shape[1:])
    else:
      u, ref = u.reshape((int(np.prod(ref.shape)),)), ref.reshape(-1)
  np.testing.assert_allclose(run_shaped(u, [x]), ref)
```

Six random ops on a 24-element tensor, five seeds, about thirty lines. It exercises permutations of rank 3 and 4 (which catch the inverse-permutation bug that 2D transposes hide), shrinks of expanded tensors, and reshapes of non-contiguous views, which is precisely where strides would need a copy and index maps produce `//` and `%`. Appendix B (Chapter 17) discusses this kind of differential fuzzing in general.

Pitfalls

1. The syllabus GEMM is not a GEMM (NOTES 11). `reshape(A, (N, N, 1)) * reshape(B, (N, 1, N))` reduced over the last axis computes `A * rowsum(B)`. It type-checks, it runs, and it produces numbers of a

plausible magnitude. Only an oracle catches it. Derive the index map of every test program by hand once, and test against NumPy, never against your own expectations.

2. Unsimplified indices change the loop structure (NOTES 12). Rangeify of a `RESHAPE` produces `((i*N + j)//N) % N`, which mentions `j` but is `i`. Because the reduce lowering places accumulators in the loops an index mentions, row reductions got nested inside the column loop: O(N^3). Fused attention at N=512 took 1787 ms; simplifying at each `RESHAPE` made it 7 ms. The C above shows the before and after. The symptom to look for in generated code: an `acc = -INFINITY` *inside* a loop whose counter its reduction never uses.

3. `const_like` must broadcast (NOTES 13). `-x` is `x * -1`. If the constant is created as a scalar `CONST`, the type checker rejects the multiply:

```
UOpError: Ops.MUL: shape mismatch (3, 4) vs () (broadcast with EXPAND)
```

`UOp.const_like` therefore returns `CONST.reshape((1, 1)).expand((3, 4))` for a `(3, 4)` node. Rangeify turns that back into the scalar constant for free (`EXPAND` maps the index to 0, `RESHAPE` of `(1, 1)` drops it, `CONST` ignores the index), so the broadcast costs nothing.

4. Distribution is needed before div/mod can simplify (NOTES 14). A merge-then-split chain produces `(((i*4 + j)*4 + k) // 16) % 4`. The div rule `(a*c + b)//c = a + b//c` needs to see the `i*16` term, which only exists after distributing `(i*4 + j)*4` to `i*16 + j*4`. With the full rule set the expression simplifies to `i`. With the distribution rule filtered out of `symbolic`:

```
full rules:           i
without distribution: (((((i*4)+j)*4)+k)//16)
```

The leftover mentions `j` and `k`, which triggers Pitfall 2 all over again. Without this rule, GPT attention kernels had `((x/4096)%4)` chains in their inner loops.

5. The inverse permutation. The rule is `inv[p[k]] = idx[k]`. The tempting `inv[k] = idx[p[k]]` applies the permutation instead of its inverse. For every permutation of two axes, and for any permutation that is its own inverse, the two agree, so a test suite of transposes passes. A rank-3 rotation like `(1, 2, 0)` exposes it. This is why the random-chain test uses rank-3 and rank-4 shapes.

6. Speculative loads must stay in bounds. Composite `PAD` evaluates both branches of its `WHERE` and loads unconditionally. That is only safe because every `STACK` branch's index stays in that branch's shape. If you add a primitive `PAD` whose out-of-range index is computed as `i - 1`, the load must be guarded, or you read `data0[-1]`. C will not tell you.

Exercises

1. ★ **Predict the index.** `x` is a `(3, 4)` buffer. Write down the load index (in terms of the loop counter `r`) for `x.permute((1, 0)).reshape((12,))`, then check with `lower` + `render_c`.

2. ★ **Unravel by hand.** Compute `unravel(17, (2, 3, 4))` using the formula `(flat // stride) % size`. Then compute `ravel` of your answer.

3. ★★ **Predict the loop structure.** What C does `x.reshape((1, 3, 4)).expand((2, 3, 4)).reduce(Ops.ADD, 2)` generate (shape `(2,)`)? How many times is the sum computed, and why? *Hint:* which ranges does the index reaching the `REDUCE` mention?

4. ★★ **2D convolution.** Write a `(H, W)` by `(KH, KW)` valid convolution as a `.uir` program or with UOp builders, using only `SHRINK`, `STACK`, `PERMUTE`, `RESHAPE`, `EXPAND`, `MUL` and `REDUCE`. Test it against `scipy.signal.correlate2d(x, w, "valid")` or a NumPy loop.

5. ★★ **An O(n) FLIP.** `flip_axis` is O(n^2). Write a composite that reverses a length-`n` vector with a `STACK` of `n` one-element `SHRINK`s, and compare the generated C with `flip3` above. *Hint:* `x.shrink(((n-1-k, n-k),))` has shape `(1,)`.

6. ★★ **Find the bug.** A student's `PERMUTE` rule is `r = rangeify(u.src[0], tuple(idx[p] for p in u.arg), ctx)`. Which of these does it get wrong: `x.permute((1, 0))`, `x.permute((0, 2, 1))`, `x.permute((2, 0, 1))`? Prove it with a NumPy example.

7. ★★ **Max-pooling with movement ops.** Write 2x2 max pooling of an `(H, W)` image (`H`, `W` even) as one `RESHAPE`, one `PERMUTE` and one `REDUCE(MAX, 2)`. Compare with `Tensor.max_pool2d` in `uopc/tensor.py`, which uses `max((3, 5))`.

8. ★★ **The syllabus GEMM, fixed.** Rewrite `gemm_256x256.uir` so that it computes `A @ B` with `A` and `B` both stored row-major. Run it through `run_shaped` at `N = 256` against NumPy.

9. ★★★ **Primitive PAD.** Add `Ops.PAD` (arg: `((before, after), ...)`) with a type rule and a rangeify rule that produces `valid.where(src[idx - before], 0)` with a guarded index. Compare the C for `pad(x, ((1, 1), (0, 2)))` with the composite's. Then decide, with numbers, whether the course was right to make it composite.

10. ★★★ **Views as strides.** Write `as_strided(chain) -> (shape, strides, offset) | None` that decides whether a chain of movement ops over a buffer can be expressed as a single strided view (NumPy-style),

returning `None` when a `RESHAPE` makes that impossible. Use `_last_axis_stride` in `uopc/opt.py` for inspiration: it evaluates the index at 0 and 1. Fuzz it against `test_random_movement_chains`.

11. ★★★ **Bounds on unrolled reduce sharing.** `_reduce` keys its loop cache on `(node, frozenset(ranges_in(*idx)))`. Construct a graph where two *different* index tuples mentioning the same ranges ask for the same `REDUCE` (as in `neighbors`) and argue why sharing the loop is still correct. Then find a case where sharing would be wrong if the cache were keyed on the node alone.

Solutions to ★ exercises

1. `reshape` ravels `r` in shape `(12,)` (just `r`) and unravels it in the transposed shape `(4, 3)`: `(r // 3, r % 3)`. `PERMUTE (1, 0)` swaps them: `(r % 3, r // 3)` in `x`'s shape `(3, 4)`. The buffer index is `(r % 3) * 4 + r // 3`. The generated C:

```
void flat_t(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 12; ridx0++) {
    float val1 = data0[(((ridx0%3)*4)+(ridx0/3))];
    data1[ridx0] = val1;
  }
}
```

2. Strides of `(2, 3, 4)` are `(12, 4, 1)`. `17 // 12 % 2 = 1`, `17 // 4 % 3 = 4 % 3 = 1`, `17 // 1 % 4 = 1`. So `(1, 1, 1)`, and `ravel` gives `((0*2 + 1)*3 + 1)*4 + 1 = 17`.

Quiz

1. Why can't `EXPAND` add a new axis? What do you use instead?
2. `x` has shape `(2, 3, 4)`. What are the output shape and the index map of `x.permute((2, 0, 1))`?
3. Why does `rangeify` memoize on `(node, idx)` and not on `node` alone?
4. What decides which loops a `REDUCE`'s accumulator lives in?
5. Why does the reduce lowering need three different `AFTER` nodes on the same `ALLOC`?
6. Give the index map of `REDUCE`, and say why only the *last* axes are reduced.
7. What does the syllabus's `gemm_256x256.uir` compute?
8. Why is the unconditional load in the composite `PAD` safe?

Answers

1. Its type rule requires equal rank, and each axis keeps its size or grows from 1. Insert the axis with a `RESHAPE` to size 1 first, then `EXPAND` it. One job per op keeps each index map trivial.
2. Shape `(4, 2, 3)`. Output axis `k` is input axis `p[k]`, so `out[a, b, c] = x[b, c, a]` (`inv[2] = a`, `inv[0] = b`, `inv[1] = c`).
3. The same node evaluated at different indices is a different scalar computation (`y[i]` vs `y[i+1]`). Keying on the index too makes sharing exact; hash consing makes the key cheap.
4. The ranges that the index reaching the `REDUCE` *mentions* (`ranges_in(*idx)`), which is why indices must be simplified before that point.
5. The accumulator before the loop (with the init store), inside the loop (after the init, per reduce iteration), and after the loop (after the `END`) are three different memory states. In a pure hash-consed graph, loads of the same address with the same dependencies are one node, so the states must be distinct nodes, and the dependencies tell the linearizer the order.
6. It does not have one: `REDUCE` creates new loops over the popped axes and accumulates. Last axes only is a canonical form; any axis set is a `PERMUTE` away, and one simple op is easier to lower, differentiate and pattern-match.
7. `C[i, j] = A[i, j] * sum_k B[i, k]`, an elementwise product with a broadcast row sum.
8. The select picks the branch by position, but the load's index is computed through the `STACK` branch's own index map, which always lands inside that branch's shape (`% 3` in the example), so it never leaves the buffer even when its value is discarded.

Further reading

- J. Ragan-Kelley et al., "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines", PLDI 2013. The algorithm/schedule split that `lower(root, out, Ctx(opts=...))` imitates.
- C. R. Harris et al., "Array programming with NumPy", *Nature* 585, 2020. Strides, views, and broadcasting, explained by the people who designed them.
- The NumPy documentation on broadcasting (`numpy.org/doc/stable/user/basics.broadcasting.html`).
- tinygrad (`github.com/tinygrad/tinygrad`): `tinygrad/shape/` (ShapeTracker and View, the strides-plus-masks representation) and, in recent versions, the rangeify-based scheduler. Reading it after this chapter is the best way to see what a production version adds.
- T. Chen et al., "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning", OSDI 2018, for tensor expressions as index-space lambdas (`te.compute`).

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

`uopc/rangeify.py · 201 lines`

`uopc/composite.py · 92 lines`

`tests/test_w04_rangeify.py · 77 lines`

7. Kernel fusion and fission

Chapter 6 ended with a compiler that can lower any shape graph into one loop nest. "Any" is the problem. Ask it for `(A @ B) @ C` with three 128x128 matrices and it produces a correct C function that takes **367 ms**. The same computation split into two loop nests, with the intermediate `A @ B` written to a temporary buffer in between, takes **1.1 ms**. Nothing about the arithmetic changed; only *where the loop boundaries are*. This chapter is about choosing those boundaries: which parts of the graph become one kernel (**fusion**), where the graph is cut and an intermediate is written to memory (**fission**), and how the resulting list of kernels is compiled once, cached, and executed over and over by a training loop.

What you'll build

- The rule that decides where a graph is cut: **materialization points** (roots, `STAGE`, `REDUCE`).
- `make_plan`: cut the graph, replace each cut by a placeholder buffer, renumber buffers locally so identical kernels render identical C, and lower each piece with `rangeify`.
- The `Plan`: a list of compiled kernels plus a description of which buffers each one reads and writes, holding no UOps at all.
- Buffers owned by UOps, freed by the garbage collector.
- `realize()`, with a **structural JIT cache** keyed on the shape of the graph, not its data.
- `stage_broadcast_compute`, a one-rule cost model for the fusion/recompute trade-off, and `fuse_reduce`, the switch that lets reductions nest.

Where this fits. The kernelizer sits between the shape graph (built by the Tensor library of Chapter 9 and extended by the autodiff of Chapter 8) and rangeify. It calls `lower()` from Chapter 6 once per kernel and the renderer and runtime from Chapter 4 once per *distinct* kernel. The optimizer of Chapter 10 plugs into it through the `opt` argument.

Unless stated otherwise, the listings in this chapter were produced with `OPT=0` (no week-6 optimizer, so the C stays readable) and with the renderer's four-line prelude removed.

7.1 Why one giant loop nest is wrong

Here is `(A @ B) @ C` built with the movement ops of Chapter 6 and lowered as a single kernel (`N = 4` to keep the listing short):

```python
def mm(a, b):                                    # (n, k) @ (k, m), movement ops only
  n, k = a.shape; _, m = b.shape
  ae = a.reshape((n, 1, k)).expand((n, m, k))
  be = b.permute((1, 0)).reshape((1, m, k)).expand((n, m, k))
  return (ae * be).reduce(Ops.ADD, 1)
A, B, C = (buf(s, 16).reshape((4, 4)) for s in range(3))
root = mm(mm(A, B), C)
```

```
void abc(float* restrict data0, float* restrict data1, float* restrict data2, float* restrict
data3) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    int alu1 = (ridx0*4);
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      float acc1;
      acc1 = 0.0f;
      for (int ridx2 = 0; ridx2 < 4; ridx2++) {
        float val2 = acc1;
        float acc0;
        acc0 = 0.0f;
        float val3 = data2[((ridx2*4)+ridx1)];
        for (int ridx3 = 0; ridx3 < 4; ridx3++) {
          float val4 = acc0;
          float val5 = data0[(alu1+ridx3)];
          float val6 = data1[((ridx3*4)+ridx2)];
          acc0 = (val4+(val5*val6));
        }
        float val7 = acc0;
        acc1 = (val2+(val7*val3));
      }
      float val8 = acc1;
      data3[(alu1+ridx1)] = val8;
    }
  }
}
```

Read it with Chapter 6's rule in mind: an accumulator lives in the loops its index mentions. The inner product `(A @ B)[i, k]` is needed by the outer reduction at index `(i, k)`, where `k` (`ridx2`) is the *outer reduce's* loop counter. So the inner accumulator `acc0` lives inside the `ridx2` loop, which lives inside the `j` loop (`ridx1`). Every element `(A @ B)[i, k]` is recomputed once for every `j`: four nested loops, O(N^4) work for an O(N^3) problem. At `N = 128`:

```
one loop nest :    367.4 ms
two kernels   :      1.1 ms
max |diff| vs numpy: 0.00018310547 0.00018310547
```

Both answers are right. One is over 300x slower.

The general principle: **a value that is consumed through an `EXPAND` is recomputed once per copy, unless something stores it.** `EXPAND` is free as an index map precisely because it does not remember anything; every broadcast copy re-derives its value from scratch. For a load from a buffer, re-deriving is a single load and costs nothing extra. For a reduction (an entire loop) or a long elementwise chain, it costs a multiple of the work.

Not every reduction under an `EXPAND` is a problem. In `x - max(x)` (Chapter 6, `sub_max`), the max's index mentions only the row loop, so the linearizer computes it once per row in a sibling loop. Trouble starts when the consumer of a reduction is itself inside a loop the producer does not need, and the most common such consumer is *another reduction*: matmul chains, attention (`softmax(QK^T) V`), a LayerNorm feeding a linear layer.

> **Key idea:** fusion saves memory traffic, fission saves recomputation. A kernel boundary costs one write and one read of the intermediate; not having one costs (copies) x (work per element). A kernelizer is a policy for choosing between them.

What fusion buys: counting bytes

The other side of the trade is memory traffic, and it is worth putting numbers on it. Take softmax of a `512 x 512` float32 matrix: 1 MiB of input. The Tensor library builds it from seven primitive operations: a max-reduce, a subtract (an add of a negated, broadcast max), a multiply by `log2(e)`, an `EXP2`, a sum-reduce, a reciprocal, and a multiply. Three ways to run it:

Strategy	Kernels	Main-memory traffic (approx.)
one kernel per primitive op (an eager, unfused library)	7	six full-size ops read 1 MiB each and the four full-size elementwise ones write 1 MiB each: ~10 MiB
cut at every `REDUCE` (the `uopc` default)	3	max reads 1 MiB, sum reads 1 MiB, output reads 1 MiB and writes 1 MiB: ~4 MiB
everything fused (`fuse_reduce=True`)	1	one 2 KiB row at a time stays in L1 across the three passes: ~1 MiB read, 1 MiB written

The row-wise reductions write only 2 KiB each, which is why cutting at them is nearly free. A memory-bound kernel's time is proportional to its traffic, so the difference between the first and last row is roughly 5x

in run time, before any other optimization. This is the arithmetic behind every "fused softmax" and "fused LayerNorm" kernel in production libraries.

A vocabulary of fusions

Compiler papers name fusions by the kinds of ops on either side of the edge. In `uopc` terms:

Producer -> consumer	Example	What `uopc` does
elementwise -> elementwise	`relu(x * 2 + 1)`	always fused (loop fusion)
elementwise -> reduce	`sum(exp(x - m))`	fused into the reduction's inner loop ("input fusion")
reduce -> elementwise	`x / sum(x)`	cut; the consumer reads the reduction's temp ("epilogue" fusion only with `fuse_reduce`)
reduce -> reduce through `EXPAND`	`(A @ B) @ C`, `softmax(QK^T) V`	cut (fusing them is the recompute disaster above)
elementwise -> `EXPAND` -> reduce	LayerNorm feeding a matmul	fused unless the recompute is expensive (`stage_broadcast_compute`)
movement ops anywhere	`x.T`, `x[1:3]`, `pad`	always fused; they are index maps, never kernels

The last row deserves emphasis. A movement op never becomes a kernel by itself unless it is a root (the user asked for a transposed copy) or the week-6 layout pass decides a copy pays for itself (Chapter 10). Views are free; that was the point of Chapter 6.

7.2 Materialization points

The kernelizer's policy is deliberately simple. A node is **materialized** (computed by its own kernel and written to a buffer) if it is:

- a **root**: something the user asked for, unless it already *is* a contiguous buffer;
- a `STAGE`: an explicit request for a temporary buffer (from the user via `Tensor.contiguous()`, or inserted by a rule, see below);

- a **`REDUCE`**, unless `fuse_reduce` is on.

Everything between materialization points is fused into the kernel of the nearest materialized consumer.

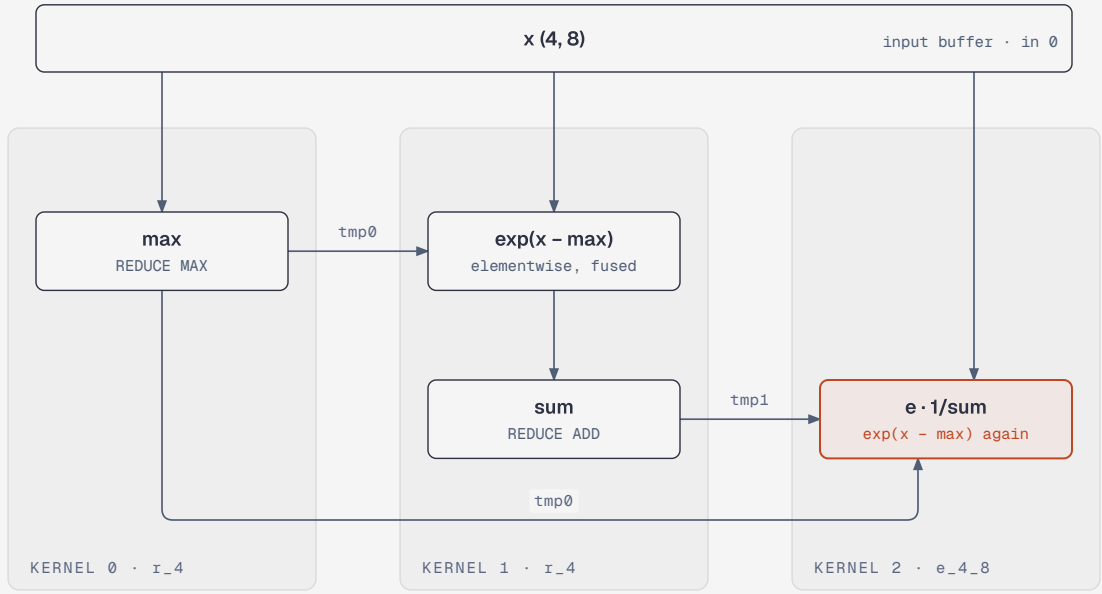
uopc/kernelize.py · _materialize_points

```
def _materialize_points(roots: list[UOp], fuse_reduce: bool) -> list[UOp]:
  topo = UOp(Ops.SINK, tuple(roots)).toposort()
  m = []
  for u in topo:
    if u.op is Ops.SINK: continue
    if u.op is Ops.STAGE or (u.op is Ops.REDUCE and not fuse_reduce) or (u in roots and not
is_contiguous_buffer(u)):
      m.append(u)
  return m
```

Cutting at *every* `REDUCE` is a blunt rule, and it is exactly the right default. A reduction's output is smaller than its input by the size of the reduced axes, so writing it costs little, and it guarantees that no reduction is ever nested inside another reduction's loop. Elementwise chains, on the other hand, are always fused: `((a * 2 + 1).relu() * 3)` becomes one kernel with one load and one store per element, instead of four kernels with four round trips through memory. With `DEBUG=4`:

```
void e_2_4(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 2; ridx0++) {
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      int alu1 = ((ridx0*4)+ridx1);
      float val2 = data0[alu1];
      float alu3 = ((val2*2.0f)+1.0f);
      data1[alu1] = (((0.0f<alu3)?alu3:0.0f)*3.0f);
    }
  }
}
```

Softmax is the standard example of a cut graph. `softmax(x) = exp(x - max(x)) / sum(exp(x - max(x)))` has two reductions, so it becomes three kernels: the max, the sum, and the elementwise output.



each REDUCE ends a kernel; the cheap exp(x − max) is recomputed in kernel 2 rather than stored

Note what the cut does *not* prevent: `exp(x - max)` is computed twice, once inside the sum kernel and once inside the output kernel. That is a recomputation too, but a constant factor of 2 on a cheap elementwise op, which is far cheaper than writing and re-reading an `(N, N)` temporary. Fusion is a policy, not a law.

7.3 make_plan: from a graph to kernels

Here is the kernelizer in full. It takes the roots, returns a `Plan` and the list of input `BUFFER`s the plan refers to:

`uopc/kernelize.py · make_plan`

```python
def make_plan(roots: list[UOp], fuse_reduce: bool = False, opt=None) -> tuple[Plan, list[UOp]]:
  """Kernelize. Returns the plan and the canonical list of input BUFFERs it refers to as ("in",
i)."""
  inputs = [u for u in UOp(Ops.SINK, tuple(roots)).toposort() if u.op is Ops.BUFFER]
  in_idx = {u: i for i, u in enumerate(inputs)}
  roots = [graph_rewrite(r, strip_detach) for r in roots]
  # fuse_reduce asks for "never write the big intermediate", so don't stage broadcasts either
  if not fuse_reduce: roots = stage_broadcast_compute(roots)
  if opt is not None: roots = opt.layout(roots)
  points = _materialize_points(roots, fuse_reduce)
  # every materialization point gets a placeholder buffer; consumers read it through a reshape
  ref: dict[UOp, tuple[str, int]] = {}
  temps: list[tuple[DType, int]] = []
  outs: list = []
  place: dict[UOp, UOp] = {}
  for k, u in enumerate(points):
    if u in roots: continue
    ref[u] = ("tmp", len(temps))
    temps.append((u.dtype, max(1, prod(u.shape))))
  for r, u in enumerate(roots):
    if is_contiguous_buffer(u): outs.append(("in", in_idx[base_buffer(u)]))
    else:
      ref[u] = ("out", r)
      outs.append((u.dtype, max(1, prod(u.shape))))
  for u in points:
    ph = UOp(Ops.BUFFER, (), ParamArg(-1 - len(place), u.dtype, (max(1, prod(u.shape)),)))   #
negative slots: placeholders
    place[u] = ph.reshape(u.shape) if u.shape != ph.shape else ph
  ph_ref = {base_buffer(place[u]): ref[u] for u in points}

  kernels = []
  for u in points:
    body = u.src[0] if u.op is Ops.STAGE else u
    body = body.substitute({p: v for p, v in place.items() if p is not u})
    # renumber buffers 0..n-1 locally (output last) so identical kernels render identical C
    bufs = [b for b in body.toposort() if b.op is Ops.BUFFER]
    local = {b: UOp(Ops.BUFFER, (), ParamArg(i, b.dtype, b.arg.shape)) for i, b in
enumerate(bufs)}
    out = UOp(Ops.BUFFER, (), ParamArg(len(bufs), u.dtype, (max(1, prod(u.shape)),)))
    body = body.substitute(local)
    kname = _kernel_name(body)
    if opt is not None: ast = opt.lower(body, out)
    else: ast = lower(body, out)
    from uopc.render_c import render_c
    src = render_c(ast, kname)
    refs = [ph_ref[b] if b.arg.slot < 0 else ("in", in_idx[b]) for b in bufs] + [ref[u]]
    # simplification can delete a read (x*0 -> 0): the C signature then has fewer parameters,
    # so bind exactly the slots that survived, or every later argument shifts by one
    alive = sorted({b.arg.slot for b in ast.toposort() if b.op is Ops.BUFFER})
    refs = [refs[s] for s in alive]
```

```
    kernels.append(Kernel(kname, src, refs, flops=_flops(body)))
  return Plan(kernels, temps, outs), inputs
```

Walk through it with softmax on a `(4, 8)` input.

1. Inputs. Every `BUFFER` in the graph, in topological order, gets an input number: `("in", 0)`, `("in", 1)`, ... The plan never refers to a specific buffer object, only to "input number `i`". This is what lets one plan run on different data.

2. Rewrites. First `strip_detach` removes `STAGE(arg="detach")` markers, which `Tensor.detach()` inserts only to stop gradients (Chapter 8); they must not become kernels. Then `stage_broadcast_compute` (below) may wrap some nodes in `STAGE` (unless `fuse_reduce` is on), and the optional `opt.layout` from Chapter 10 may add more. Each returns new roots.

uopc/kernelize.py · strip_detach

```
strip_detach = PatternMatcher([(UPat(Ops.STAGE, arg="detach", src=(UPat.var("x"),)), lambda x:
x)])
```

3. Points. `_materialize_points` lists the cut nodes in topological order, so producers come before consumers. That order *is* the schedule.

4. References. Every point that is not a root gets a temporary, `("tmp", j)`, with its dtype and element count. Every root gets an output, `("out", r)`, unless it is already a contiguous buffer, in which case its "output" is an alias of the input (`("in", i)`).

5. Placeholders. Each point `u` gets a placeholder `BUFFER` with a *negative* slot (real buffers have positive slots, so they cannot collide), reshaped to `u`'s shape. A consumer that used `u` will now read the placeholder instead.

6. Kernel bodies. For each point, take its expression (the `STAGE`'s source, or the node itself) and substitute every *other* point by its placeholder. The body now reads only buffers. It is a complete, self-contained shape graph: exactly what `lower()` accepts.

7. Local renumbering. The body's buffers are renumbered `0, 1, ..., n-1` in topological order, and the output gets slot `n`. `refs` remembers, for each local slot, what it really is: an input, a temp, or an output.

8. Lowering. `lower()` (or `opt.lower()`) rangeifies the body into a `SINK`, `render_c` turns it into C, and the kernel is recorded with its name, source, refs and a FLOP estimate. One last detail: the refs are filtered to the buffer slots that still appear in the lowered program (`alive`). Simplification can delete a read entirely,

and the rendered C function then has fewer parameters than the body had buffers. Pitfall 7 is the bug this line fixes.

For softmax, the plan is:

```
r_4 [('in', 0), ('tmp', 0)] 32
r_4 [('in', 0), ('tmp', 0), ('tmp', 1)] 132
e_4_8 [('in', 0), ('tmp', 0), ('tmp', 1), ('out', 0)] 136
temps: [(dtypes.f32, 4), (dtypes.f32, 4)]
outs: [(dtypes.f32, 32)]
inputs: [UOp(Ops.BUFFER, arg=ParamArg(slot=1, dtype=dtypes.f32, shape=(32,), addrspace=None))]
```

Read the refs as function signatures. Kernel 0 reads input 0 and writes temp 0 (the row max). Kernel 1 reads input 0 and temp 0 and writes temp 1 (the row sum of `exp(x - max)`). Kernel 2 reads all three and writes output 0. The generated code:

```
void r_4(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float acc0;
    acc0 = -INFINITY;
    for (int ridx1 = 0; ridx1 < 8; ridx1++) {
      float val1 = acc0;
      float val2 = data0[((ridx0*8)+ridx1)];
      acc0 = ((val1>val2)?val1:val2);
    }
    float val3 = acc0;
    data1[ridx0] = val3;
  }
}
```

```
void r_4(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    float val1 = data1[ridx0];
    for (int ridx1 = 0; ridx1 < 8; ridx1++) {
      float val2 = acc0;
      float val3 = data0[((ridx0*8)+ridx1)];
      acc0 = (val2+exp2f(((val3+(val1*-1.0f))*1.4426950408889634f)));
    }
    float val4 = acc0;
    data2[ridx0] = val4;
  }
}
```

```
void e_4_8(float* restrict data0, float* restrict data1, float* restrict data2, float* restrict
data3) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data1[ridx0];
    float val2 = data2[ridx0];
    for (int ridx1 = 0; ridx1 < 8; ridx1++) {
      int alu3 = ((ridx0*8)+ridx1);
      float val4 = data0[alu3];
      data3[alu3] = (exp2f(((val4+(val1*-1.0f))*1.4426950408889634f))*(1.0f/val2));
    }
  }
}
```

A few things to notice. `exp` is `exp2(x * log2(e))`, because the IR's only exponential is `EXP2` (Chapter 9). The placeholder reads (`data1[ridx0]` of the max, a `(4,)` buffer reshaped to `(4, 1)` and expanded to `(4, 8)`) became loads hoisted out of the column loop: to rangeify, a placeholder is just another buffer behind a chain of movement ops. And each kernel's parameters are numbered from `data0`, whatever the global buffer numbers were.

Kernel names and FLOPs

Kernels are named in tinygrad's style: `r_` if the body contains a reduction, `e_` if it is purely elementwise, followed by the output shape. Names are for humans reading `DEBUG` output and profiles; nothing depends on them.

uopc/kernelize.py · _kernel_name

```
def _kernel_name(root: UOp) -> str:
  red = any(u.op is Ops.REDUCE for u in root.toposort())
  return ("r_" if red else "e_") + "_".join(map(str, root.shape or (1,)))
```

uopc/kernelize.py · _flops

```
def _flops(u: UOp) -> int:
  """Rough FLOP count of a shaped kernel body: one per ALU element computed."""
  from uopc.uop import ALU
  return sum(prod(x.shape or (1,)) for x in u.toposort() if x.op in ALU) + \
         sum(prod(x.src[0].shape) for x in u.toposort() if x.op is Ops.REDUCE)
```

The FLOP count is a rough "one per ALU element plus one per reduced element", good enough for the GFLOP/s column of `DEBUG=2` and for comparing kernels with each other. Chapter 10 counts FLOPs more

carefully.

7.4 Identical kernels, identical C

Step 7, the local renumbering, looks like bookkeeping. It is an optimization with a large payoff. Two kernels that do the same thing to different buffers, say the row sums of two different `(16, 16)` matrices, produce the same body up to buffer identity. After renumbering, the bodies are *the same UOp* (hash consing again), lower to the same `SINK`, and render to byte-identical C:

```
a: 1  b: 2
r_16 [('in', 0), ('out', 0)]
r_16 [('in', 1), ('out', 1)]
same C source: True
void r_16(float* restrict data0, float* restrict data1) {
```

The runtime of Chapter 4 caches compiled shared objects by a hash of the source and flags, so the second kernel costs no compilation. In a transformer, every layer has the same shapes: twelve layers produce twelve copies of each kernel and one compilation. The kernel name intentionally carries no buffer numbers, for the same reason.

7.5 The Plan

The output of kernelization is a `Plan`:

uopc/kernelize.py · Kernel

```python
@dataclass
class Kernel:
  name: str
  src: str
  refs: list[tuple[str, int]]      # one per local slot: ("in", i) | ("tmp", j) | ("out", r)
  program: Program | None = None
  flops: int = 0
```

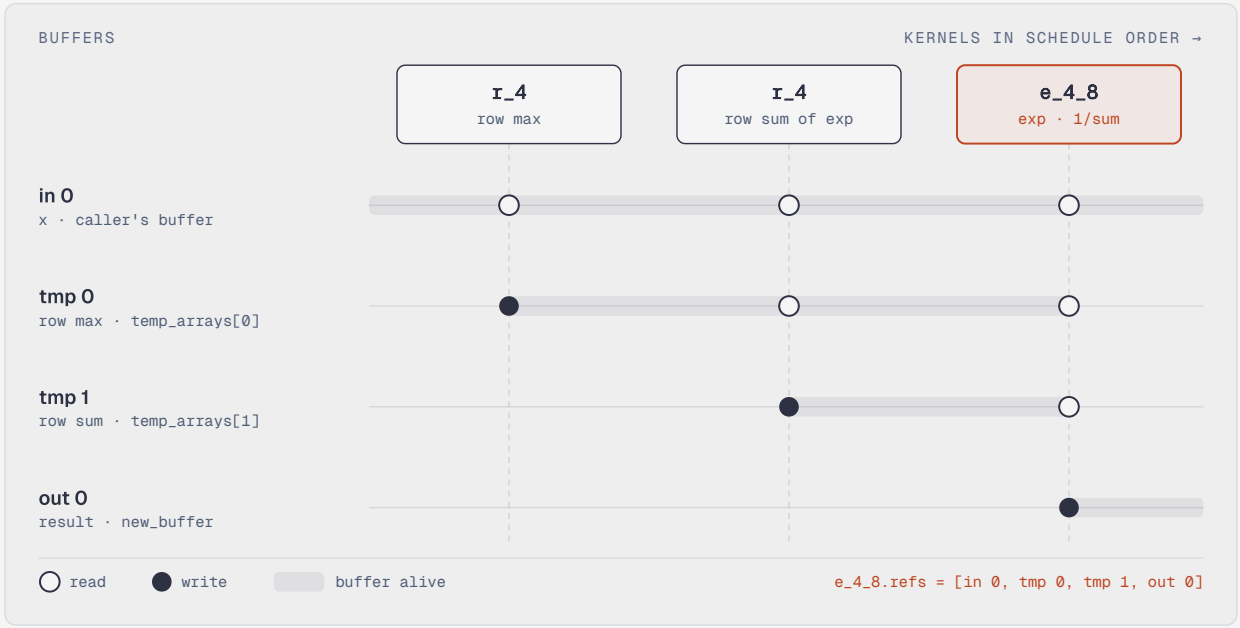
uopc/kernelize.py · Plan

```python
@dataclass
class Plan:
  kernels: list[Kernel]
  temps: list[tuple[DType, int]]
  outs: list[tuple[str, int] | tuple[DType, int]]   # per root: ("in", i) alias, or (dtype, size) fresh
  temp_arrays: list | None = None   # allocated on first run, then reused: no page faults on every step
```

Two design decisions are visible here.

First, **a plan contains no UOps**: kernel sources are strings, buffers are referred to by `("in" | "tmp" | "out", number)`. A plan can therefore be cached forever without keeping any tensor alive, and it can run on any set of inputs with the right shapes.

Second, **temporaries belong to the cached plan**: `realize` allocates `temp_arrays` the first time a plan runs and reuses them on every later run. Temps never escape (only outputs are handed back, in fresh `new_buffer`s), so reuse is safe as long as runs of one plan do not overlap, which they cannot in a single-threaded Python loop. An earlier version allocated them with `np.empty` on every run; that is simpler, but every step then paid page faults on freshly mapped memory, which the `DEBUG=3` output below makes visible.



7.6 Buffers belong to UOps

Where does the data for "input 0" live? In the `BUFFER` UOp itself:

uopc/kernelize.py · new_buffer

```
def new_buffer(dtype: DType, size: int, data: np.ndarray | None = None) -> UOp:
  global _next_slot
  _next_slot += 1
  u = UOp(Ops.BUFFER, (), ParamArg(_next_slot, dtype, (max(size, 1),)))
  if data is not None: u.__dict__["np"] = np.ascontiguousarray(data.reshape(-1),
dtype=np_dtype(dtype))
  return u
```

uopc/kernelize.py · buffer_data

```
def buffer_data(u: UOp) -> np.ndarray:
  if "np" not in u.__dict__: u.__dict__["np"] = np.zeros(u.arg.shape[0], np_dtype(u.dtype))
  return u.__dict__["np"]
```

uopc/kernelize.py · base_buffer

```
def base_buffer(u: UOp) -> UOp:
  while u.op is Ops.RESHAPE: u = u.src[0]
  return u
```

`new_buffer` hands out a fresh, positive slot number per buffer (so two buffers with the same shape are different UOps, and hash consing never merges them) and stores the NumPy array in the UOp's `__dict__`. Recall from Chapter 1 that `UOp._cache` is a `WeakValueDictionary`: it does not keep nodes alive. When the last `Tensor` that refers (directly or through a graph) to a `BUFFER` UOp goes away, Python frees the UOp, its `__dict__`, and the array. Memory management for the whole system is the Python garbage collector. There is no `free` anywhere in `uopc`.

The consequence for a training loop is pleasant: after a step, `Tensor.uop` of each parameter is rebound to its new output buffer, the old graph becomes unreachable, and every intermediate buffer of the previous step is freed.

7.7 realize(): running a plan

`realize` is the function the Tensor library calls when you ask for data (`.numpy()`, `.item()`, `realize_all`):

uopc/kernelize.py · realize

```
def realize(roots: list[UOp], fuse_reduce: bool = False, opt=None) -> list[UOp]:
  """Compute each root. Returns, per root, a BUFFER UOp holding the result."""
  uniq = list(dict.fromkeys(roots))
  t0 = time.perf_counter()
  key = (_structure_key(uniq), fuse_reduce, opt)
  inputs = [u for u in UOp(Ops.SINK, tuple(uniq)).toposort() if u.op is Ops.BUFFER]
  if (plan := _plans.get(key)) is None:
    plan, _ = make_plan(uniq, fuse_reduce, opt)
    for k in plan.kernels:
      k.program = Program(k.src, k.name)
      if DEBUG >= 4: print(k.src)
    _plans[key] = plan
  t1 = time.perf_counter()
  if plan.temp_arrays is None: plan.temp_arrays = [np.empty(n, np_dtype(d)) for d, n in
plan.temps]
  temps = plan.temp_arrays
  outs = [inputs[o[1]] if o[0] == "in" else new_buffer(o[0], o[1]) for o in plan.outs]
  def get(rf):
    kind, i = rf
    return buffer_data(inputs[i]) if kind == "in" else temps[i] if kind == "tmp" else
buffer_data(outs[i])
  for k in plan.kernels:
    ts = time.perf_counter()
    k.program(*(get(r) for r in k.refs))
    if DEBUG >= 2:
      dt = time.perf_counter() - ts
      extra = ""
      if DEBUG >= 3:   # run again: the second run has warm caches and no first-touch page faults
        ts = time.perf_counter(); k.program(*(get(r) for r in k.refs)); extra = f"  (again:
{(time.perf_counter()-ts)*1e3:.3f} ms)"
      print(f"  {k.name:30s} {dt*1e3:8.3f} ms  {k.flops/max(dt, 1e-9)/1e9:7.2f} GFLOP/s{extra}")
  if DEBUG >= 1: print(f"realize: {len(plan.kernels)} kernels, schedule {(t1-t0)*1e3:.1f} ms, run
{(time.perf_counter()-t1)*1e3:.1f} ms")
  by_root = dict(zip(uniq, outs))
  return [by_root[r] for r in roots]
```

It deduplicates the roots, looks the graph's *structure* up in `_plans`, builds and compiles a plan on a miss, allocates temporaries and outputs, and calls every kernel with the right arrays. The `DEBUG` levels print progressively more:

DEBUG	Prints
1	one line per `realize`: number of kernels, schedule time (Python side, including compiles), run time
2	plus one line per kernel: name, time, GFLOP/s
3	plus a second, warm run of each kernel (no first-touch page faults, warm caches)
4	plus the C source of each kernel *when it is compiled* (cache misses only)

Softmax of a `512 x 512` matrix, computed twice, with `DEBUG=2`:

```
  r_512                            0.680 ms     0.39 GFLOP/s
  r_512                            1.498 ms     0.70 GFLOP/s
  e_512_512                        2.116 ms     0.50 GFLOP/s
realize: 3 kernels, schedule 238.1 ms, run 4.4 ms
[1.0000002 1.        0.9999997]
  r_512                            0.634 ms     0.41 GFLOP/s
  r_512                            1.427 ms     0.74 GFLOP/s
  e_512_512                        1.511 ms     0.69 GFLOP/s
realize: 3 kernels, schedule 0.1 ms, run 3.6 ms
[1.0000002 1.        0.9999997]
```

The first call spends 238 ms building the plan (rangeify, simplify, render, three `gcc` invocations; these runs used an empty `UOPC_CACHE` directory, so nothing was precompiled). The second spends 0.1 ms: the plan was found in the cache, and its temporaries were already allocated. The kernels themselves are slow (well under 1 GFLOP/s, against a machine peak in the hundreds; that is Chapter 10's job).

And with `DEBUG=3`, on a graph with an explicit `STAGE`:

```
  e_256_256                        0.523 ms     0.38 GFLOP/s  (again: 0.353 ms)
  r_256                            0.034 ms     1.93 GFLOP/s  (again: 0.023 ms)
realize: 2 kernels, schedule 153.1 ms, run 1.1 ms
```

The first run of `e_256_256` is about 50% slower than the second because it writes to a freshly allocated temporary: the first touch of every 4 KB page is a page fault. Benchmark with warm runs.

7.8 The structural JIT cache

A training loop builds a new graph every step: new parameter buffers, new data, same model. Running rangeify, simplify, and the renderer for every step would cost seconds per step for a GPT (the first compile of the GPT graph takes ~9 s even after optimization; NOTES 22). The cache key must therefore identify graphs that are *the same program on different data*:

uopc/kernelize.py · _structure_key

```
def _structure_key(roots: list[UOp]) -> tuple:
  ids: dict[UOp, int] = {}
  key = []
  for u in UOp(Ops.SINK, tuple(roots)).toposort():
    ids[u] = len(ids)
    arg = ("buf", u.dtype.name, u.arg.shape) if u.op is Ops.BUFFER else _argkey(u.arg)
    key.append((u.op, arg, tuple(ids[s] for s in u.src)))
  return tuple(key)
```

It is the graph in topological order, with each node replaced by `(op, arg, source positions)`, so the key captures the exact DAG shape including sharing. The one exception is `BUFFER`: its arg is replaced by `(dtype, size)`, dropping the slot number. Two graphs that differ only in *which* buffers they read have the same key, and since the plan refers to inputs by topological position, the cached plan is correct for both. That is the whole JIT: a dictionary.

Everything else in the arg stays in the key, including the value of every `CONST`. That is correct (a different constant is a different program) and it is a trap.

Python numbers are structure

AdamW's bias corrections are `1 / (1 - b1**t)`, a Python float that changes every step. Multiply a tensor by it and it becomes a `CONST` in the graph, hence part of the structure key, hence a cache miss every step (NOTES 17). Here is the effect, at `DEBUG=1`:

```python
for t in range(1, 4):
  corr = 1 / (1 - 0.9 ** t)
  (w * corr).realize()
  print(f"step {t}: corr={corr:.4f}  cached plans={len(_plans)}")
for t in range(1, 4):
  corr = Tensor(np.array([1 / (1 - 0.9 ** t)], np.float32))
  (w * corr).realize()
  print(f"step {t}: cached plans={len(_plans)}")
```

```
-- bias correction as a Python float (baked into the graph as a CONST)
realize: 1 kernels, schedule 85.1 ms, run 0.1 ms
step 1: corr=10.0000  cached plans=1
realize: 1 kernels, schedule 78.1 ms, run 0.1 ms
step 2: corr=5.2632  cached plans=2
realize: 1 kernels, schedule 76.9 ms, run 0.1 ms
step 3: corr=3.6900  cached plans=3
-- bias correction as a 1-element tensor (a BUFFER: data, not structure)
realize: 1 kernels, schedule 74.2 ms, run 0.1 ms
step 1: cached plans=4
realize: 1 kernels, schedule 0.1 ms, run 0.1 ms
step 2: cached plans=4
realize: 1 kernels, schedule 0.1 ms, run 0.0 ms
step 3: cached plans=4
```

Each Python-float step pays ~80 ms of compilation (with an empty on-disk `.so` cache) for a 0.1 ms kernel. In a real model with hundreds of kernels that is the difference between a fast training loop and a compiler benchmark. The fix in `uopc/nn.py` is to pass the changing scalars as one-element tensors:

uopc/nn.py · AdamW.step

```python
  def step(self, *extra: Tensor):
    self.t += 1
    # bias corrections are python floats baked into the graph as constants: each step's
    # graph differs only in two CONSTs, so we pass them as tensors to keep the JIT cache warm
    c1 = Tensor([self.lr / (1 - self.b1 ** self.t)])
    c2 = Tensor([1 / (1 - self.b2 ** self.t)])
    news = []
    for p, m, v in zip(self.params, self.m, self.v):
      g = p.grad
      m.assign(m * self.b1 + g * (1 - self.b1))
      v.assign(v * self.b2 + g.square() * (1 - self.b2))
      upd = m * c1.reshape(*([1] * p.ndim)) / ((v * c2.reshape(*([1] * p.ndim))).sqrt() +
self.eps)
      news.append(p.assign(p.detach() * (1 - self.lr * self.wd) - upd))
    Tensor.realize_all(*news, *self.m, *self.v, *extra)
```

The same reasoning explains why the GPT sampler pads its context to a fixed length (NOTES 18):

```
ctx = ctx + [0] * (a.ctx - len(ctx))                    # fixed shape -> one compiled graph
```

A shape is structure too. Feeding contexts of length 1, 2, 3, ... would compile a new plan for every generated token. Padding to the maximum length and reading out the logits at the right position keeps one plan. Chapter 15 shows the real fix: symbolic shapes.

> **Key idea:** the JIT cache splits every value into *structure* (ops, shapes, constants) and *data* (buffer contents). Anything that changes between steps must be data.

7.9 STAGE: asking for a buffer

`STAGE` is the IR's fission knob. `UOp(Ops.STAGE, (x,))` means "materialize `x` into a temporary and let consumers index into it". It has no semantics beyond identity: rangeify without a kernelizer simply computes through it (Chapter 6), and its gradient is the identity (Chapter 8). The Tensor library exposes it as `contiguous()`:

uopc/tensor.py · Tensor.contiguous

```
def contiguous(self) -> Tensor:
  """Ask for a materialization point: the kernelizer will write this into a temp buffer."""
  return Tensor(UOp(Ops.STAGE, (self.uop,)))
```

`(a.exp() * 2).contiguous().sum(1)` is two kernels: an `e_256_256` that writes `exp(a) * 2` to a temp, and an `r_256` that sums it (the `DEBUG=3` output above). Without `.contiguous()` it would be one fused `r_256`. Which is faster depends on the consumer, and the next section is about the one case where the compiler should decide on its own.

7.10 The recompute trap: stage_broadcast_compute

Cutting at every `REDUCE` prevents reductions from being recomputed. It does nothing about *elementwise* chains that are broadcast into a reduction. The canonical case is a LayerNorm feeding a linear layer:

```
x = Tensor.randn(128, 256); W = Tensor.randn(256, 256)
m = x.mean(-1, keepdim=True)
y = ((x - m) * ((x - m).square().mean(-1, keepdim=True) + 1e-5).rsqrt()) @ W
```

The normalized activation `xn = (x - m) * rsqrt(var + eps)` has shape `(128, 256)`. The matmul expands it to `(128, 256, 256)` (one copy per output column) and reduces. Since `xn` is elementwise, it is fused into the matmul kernel and recomputed for every output column:

```
void r_128_256(float* restrict data0, float* restrict data1, float* restrict data2, float*
restrict data3, float* restrict data4) {
  for (int ridx0 = 0; ridx0 < 128; ridx0++) {
    int alu1 = (ridx0*256);
    float val2 = data1[ridx0];
    float val3 = data2[ridx0];
    for (int ridx1 = 0; ridx1 < 256; ridx1++) {
      float acc0;
      acc0 = 0.0f;
      for (int ridx2 = 0; ridx2 < 256; ridx2++) {
        float val4 = acc0;
        float val5 = data0[(alu1+ridx2)];
        float val6 = data3[((ridx2*256)+ridx1)];
        acc0 = (val4+(((val5+((val2*0.00390625f)*-1.0f))*(1.0f/sqrtf(((val3*0.00390625f)+1e-
05f))))*val6));
      }
      float val7 = acc0;
      data4[(alu1+ridx1)] = val7;
    }
  }
}
```

The inner loop does the LayerNorm arithmetic (and a `sqrtf` and a division, which `gcc` may or may not hoist) for every multiply-add of the matmul. In a GPT forward/backward step, this pattern cost **374 ms of kernel time; staging the broadcast computations brought it to 125 ms** (NOTES 15).

The fix is a rewrite that runs before cutting:

`uopc/kernelize.py · stage_broadcast_compute`

```python
def stage_broadcast_compute(roots: list[UOp], factor: int = 4) -> list[UOp]:
  """Fusing is only free if nothing gets recomputed. A computed tensor that is EXPANDed by
  `factor` or more (e.g. a LayerNorm output feeding a matmul) would be recomputed for every
  copy, so we materialize it first. This is the fusion/recompute trade-off, as one rule."""
  from uopc.uop import MOVEMENT, ALU
  todo: set[UOp] = set()
  for u in UOp(Ops.SINK, tuple(roots)).toposort():
    if u.op is not Ops.EXPAND or prod(u.shape) < factor * prod(u.src[0].shape): continue
    base = u.src[0]
    while base.op in MOVEMENT: base = base.src[0]
    if base.op in ALU and base.op is not Ops.CAST and base.shape and _fused_work(base) >= 4096:
todo.add(base)
  return UOp.wrap(roots, todo, lambda n: UOp(Ops.STAGE, (n,))) if todo else roots
```

uopc/kernelize.py · _fused_work

```python
def _fused_work(u: UOp) -> int:
  """ALU ops * elements that recomputing u would cost (stopping at materialized nodes)."""
  from uopc.uop import ALU
  n, seen, stack = 0, set(), [u]
  while stack:
    x = stack.pop()
    if x in seen or x.op in (Ops.BUFFER, Ops.REDUCE, Ops.STAGE): continue
    seen.add(x)
    if x.op in ALU: n += 1
    stack.extend(x.src)
  return n * prod(u.shape)
```

The rule, in words: find every `EXPAND` that multiplies the element count by at least `factor` (4). Walk down through movement ops to the computed value being broadcast. If it is a real computation (an ALU op other than a bare `CAST`, not a scalar) and recomputing it is expensive (`_fused_work`, the number of ALU ops in the fused expression times its element count, is at least 4096), wrap it in a `STAGE`. On the LayerNorm example:

```
STAGEs inserted: 1  shape: (128, 256)  fused work: 262144
kernels with staging:    ['r_128', 'r_128', 'e_128_256', 'r_128_256']
kernels without staging: ['r_128', 'r_128', 'r_128_256']
```

With staging, the LayerNorm output is written once by its own `e_128_256` kernel and the matmul kernel becomes a plain GEMM:

```
void e_128_256(float* restrict data0, float* restrict data1, float* restrict data2, float*
restrict data3) {
  for (int ridx0 = 0; ridx0 < 128; ridx0++) {
    float val1 = data1[ridx0];
    float val2 = data2[ridx0];
    for (int ridx1 = 0; ridx1 < 256; ridx1++) {
      int alu3 = ((ridx0*256)+ridx1);
      float val4 = data0[alu3];
      data3[alu3] = ((val4+((val1*0.00390625f)*-1.0f))*(1.0f/sqrtf(((val2*0.00390625f)+1e-
05f))));
    }
  }
}
```

```
void r_128_256(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 128; ridx0++) {
    int alu1 = (ridx0*256);
    for (int ridx1 = 0; ridx1 < 256; ridx1++) {
      float acc0;
      acc0 = 0.0f;
      for (int ridx2 = 0; ridx2 < 256; ridx2++) {
        float val2 = acc0;
        float val3 = data0[(alu1+ridx2)];
        float val4 = data1[((ridx2*256)+ridx1)];
        acc0 = (val2+(val3*val4));
      }
      float val5 = acc0;
      data2[(alu1+ridx1)] = val5;
    }
  }
}
```

How much does it buy? It depends on what else limits the kernel, which is a lesson about cost models. I timed the whole plan (best of 5 runs) both ways:

```
OPT=0:  with staging: 1.99 ms   without: 1.91 ms
OPT=1:  with staging: 0.58 ms   without: 1.35 ms
```

At `OPT=0` the naive GEMM is bound by its stride-256 walk down `W`'s columns: the CPU waits on memory either way, the recompute hides in the shadow of the cache misses, and the extra kernel makes the staged version slightly slower. With the week-6 optimizer (layout pass, register tiling, threads), the GEMM becomes compute-bound and the recompute is pure overhead: staging is 2.3x faster. The week-6 kernel list gains a transposing copy of `W` (`e_1_256_256`) from the layout pass, explained in Chapter 10. Optimizations interact; measure the configuration you ship.

Why the thresholds

Both thresholds exist because staging has a cost: an extra kernel launch, a write and a read of the intermediate, and a temporary allocation.

- `factor = 4`: an `EXPAND` that merely doubles a tensor recomputes each element twice. Writing and re-reading it is about as expensive. Fourfold reuse or more makes staging a clear win.
- `_fused_work >= 4096`: softmax's `1 / sum` is a tiny `(rows, 1)` computation broadcast over the row. The first version of the rule had no work threshold, staged it, and added **two useless kernels** to every softmax (NOTES 15). Recomputing a reciprocal per element is cheaper than a kernel launch.
- `CAST` is excluded because a bare type conversion is as cheap as the load it wraps.

This is a cost model in miniature: an estimate of recompute work, compared to a fixed threshold. It is crude and it is right often enough. Production compilers (XLA, Inductor, tinygrad) have more elaborate versions of exactly this trade-off.

Wrapping bottom-up

`stage_broadcast_compute` collects a set of nodes to wrap, then rebuilds the graph with `UOp.wrap`, not with the more obvious `substitute`. The difference matters when one target contains another (NOTES 16):

uopc/uop.py · UOp.wrap

```
@staticmethod
def wrap(roots: list[UOp], targets: set[UOp], fn) -> list[UOp]:
  """Rebuild the graph bottom-up, replacing each node t in targets by fn(t'), where t' is t
  with *its* sources already rebuilt. (substitute() would wrap the original, unrewritten t.)"""
  new: dict[UOp, UOp] = {}
  for u in UOp(Ops.SINK, tuple(roots)).toposort():
    src = tuple(new[s] for s in u.src)
    n = u if src == u.src else u.replace(src=src)
    new[u] = fn(n) if u in targets else n
  return [new[r] for r in roots]
```

`root.substitute({inner: STAGE(inner), outer: STAGE(outer)})` replaces `outer` by a `STAGE` of the *original* `outer`, which still contains the original, unstaged `inner`; the staging of `inner` is lost. `wrap` rebuilds bottom-up and wraps the *rebuilt* node, so nested targets compose:

```
inner = (x * 2.0 + 1.0)            # wants staging
outer = UOp(Ops.EXP2, (inner,)) * inner       # also wants staging, and contains inner
root = outer.reduce(Ops.ADD, 1)
```

```
substitute: STAGE nodes = 1
wrap:       STAGE nodes = 2
```

7.11 fuse_reduce: letting reductions nest

The default cuts at every `REDUCE`. With `fuse_reduce=True`, reductions are not materialization points; they fuse into their consumers, and rangeify's accumulator placement decides the loop structure. `fuse_reduce` also turns off `stage_broadcast_compute`: the switch means "never write the big intermediate", and staging a broadcast computation is exactly such a write. For softmax, fusing is a strict improvement: one kernel instead of three, no temporaries, and the row's max and sum computed in sibling loops:

```
void r_4_8(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    int alu1 = (ridx0*8);
    float acc0;
    acc0 = -INFINITY;
    float acc1;
    acc1 = 0.0f;
    for (int ridx2 = 0; ridx2 < 8; ridx2++) {
      float val2 = acc0;
      float val3 = data0[(alu1+ridx2)];
      acc0 = ((val2>val3)?val2:val3);
    }
    float val4 = acc0;
    float alu5 = (val4*-1.0f);
    for (int ridx3 = 0; ridx3 < 8; ridx3++) {
      float val6 = acc1;
      float val7 = data0[(alu1+ridx3)];
      acc1 = (val6+exp2f(((val7+alu5)*1.4426950408889634f)));
    }
    float val8 = acc1;
    for (int ridx1 = 0; ridx1 < 8; ridx1++) {
      int alu9 = (alu1+ridx1);
      float val10 = data0[alu9];
      data1[alu9] = (exp2f(((val10+alu5)*1.4426950408889634f))*(1.0f/val8));
    }
  }
}
```

For `(A @ B) @ C` it is the 367 ms disaster from the start of the chapter. So `fuse_reduce` is a global switch (`Tensor.fuse_reduce`), off by default, and it pays off for one important case: attention, where fusing the softmax reductions into the attention kernel and keeping the score row in registers is the first step toward flash attention (Chapter 11). A smarter kernelizer would decide per reduction; Exercise 8 asks you to write that rule.

7.12 The whole step, end to end

Putting Chapters 6, 7 and 8 together on a tiny model (a linear layer, ReLU, mean of squares) and kernelizing the loss *and* the gradient of the weight in one plan:

```
X, W = Tensor.randn(32, 64), Tensor.randn(64, 10, requires_grad=True)
loss = (X @ W).relu().square().mean()
loss.backward([W])
plan, _ = make_plan([loss.uop, W.grad.uop])
```

```
['r_32_10', 'r_1', 'e_1', 'r_10_64', 'e_64_10']
```

Five kernels: the forward matmul `X @ W`; the sum of squares; the scale by `1/320` for the mean; the gradient matmul `X^T @ dL/dh` (with the ReLU and square derivatives fused into its inner loop); and a transposing copy of that result into `W`'s `(64, 10)` layout. The forward matmul is computed **once** and read by both the loss and the gradient kernels. That happened for free: the gradient graph refers to the very same `REDUCE` node as the forward graph (hash consing), so it is one materialization point with one temp. Chapter 8 explains how that gradient graph is built.

7.13 Building it yourself: an order of work

If you are implementing week 5 from this chapter, this order keeps every step testable:

1. **One kernel per root, no cuts.** `make_plan` with `_materialize_points` returning only the roots is just `lower()` per root plus bookkeeping. Test: the week-4 tests, through the new path.
2. **Cut at `REDUCE`.** Add placeholders and substitution. Test: softmax has 3 kernels (`test_kernelize_fuses_elementwise_and_splits_reductions`) and matches NumPy; `(A @ B) @ C` at N=128 runs in milliseconds, not hundreds of them.
3. **Local renumbering.** Test: two same-shaped reductions of different buffers render identical C.
4. **The Plan, without UOps.** Make `realize` run a plan against a list of input arrays. Test: build the plan once, run it on two different inputs, compare both against NumPy.
5. **The structural cache.** Test: the second `realize` of a same-shaped graph reports a sub-millisecond schedule time at `DEBUG=1`.
6. **`STAGE` and `contiguous()`**, then `stage_broadcast_compute`. Test: the LayerNorm-into-matmul graph gains one kernel, and the softmax graph does not.
7. **The training loop** of Chapter 9: `test_training_reduces_loss`.

At each step, keep `DEBUG=2` output from a fixed small model in a text file and diff it as you go: kernel counts and names are the cheapest regression test a kernelizer has.

Pitfalls

1. The fusion/recompute trap (NOTES 15). Fusing everything between reductions sounds free, but an elementwise result that is `EXPAND`ed into a reduce is recomputed for every output column. A GPT step spent 374 ms in kernels; `stage_broadcast_compute` brought it to 125 ms. Staging is not free either: the first version staged softmax's tiny `1/sum` and added two useless kernels. Only stage when the recompute is expensive (`_fused_work >= 4096`).

2. `substitute` vs. bottom-up wrap (NOTES 16). `graph.substitute({x: STAGE(x)})` wraps the *original* `x`, so anything inside `x` that should also be staged is not. When rewrite targets can nest, rebuild bottom-up and transform the rebuilt node (`UOp.wrap`).

3. JIT cache misses from Python floats (NOTES 17). A changing Python scalar becomes a `CONST`, a `CONST` is structure, and every step recompiles. Symptom: `DEBUG=1` shows `schedule` times of hundreds of milliseconds on every step instead of fractions of a millisecond. Fix: pass changing scalars as one-element tensors.

4. Changing shapes recompile too (NOTES 18). Sampling from a GPT with a growing context compiles a new plan per token. Pad to a fixed shape (or implement symbolic shapes, Chapter 15).

5. `DEBUG=4` prints nothing on a cache hit. Sources are printed when a plan is compiled. If you run the same graph twice to "see the C", the second run is silent. This is a feature, but it has confused everyone at least once. Clear `kernelize._plans` or change a shape.

6. Measuring a fusion decision in the wrong configuration. The LayerNorm staging looked like a slight *loss* at `OPT=0` (1.99 ms vs 1.91 ms) and is a 2.3x win at `OPT=1`. A memory-bound kernel hides recomputation behind its cache misses; once the optimizer makes it compute-bound, the recompute is the bottleneck. Tune fusion heuristics with the optimizer on, on the shapes you care about.

7. A deleted read shifted every kernel argument (found and fixed). `(t * 0)` for an integer tensor `t` simplifies to the constant 0, so the lowered kernel no longer reads `t`, and `render_c` emits a function with *one* parameter, the output. `refs`, however, was still computed from the body's buffers before lowering: `[("in", 0), ("out", 0)]`. The runtime passed `t`'s array as the first argument, which the kernel took to be

its output, and `(Tensor([1, 2, 3]) * 0)` wrote its zeros *into the input*. The fix binds exactly the slots that survive lowering. Now:

```
e_3 [('out', 0)]
void e_3(int* restrict data1) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    data1[ridx0] = 0;
  }
}

[0, 0, 0] [1, 2, 3]
```

Note the parameter is still called `data1`: the renderer names parameters by slot, so the gap in the numbering is the visible trace of the deleted read. The regression test is `test_deleted_reads_do_not_shift_kernel_arguments`:

tests/test_w05_tensor.py · test_deleted_reads_do_not_shift_kernel_arguments

```python
def test_deleted_reads_do_not_shift_kernel_arguments():
  t = Tensor(np.array([1, 2, 3], np.int32))
  assert (t * 0).numpy().tolist() == [0, 0, 0]          # simplifies to a constant: t is no longer read...
  assert t.numpy().tolist() == [1, 2, 3]                # ...and must not be overwritten
```

The general lesson: any two components that must agree on a calling convention (here, the list of buffers) should derive it from the same object, the final program, not from an earlier stage that a later optimization can invalidate.

Exercises

1. ★ **Count kernels.** Without running anything, predict the number of kernels (default settings) for: `a + b`, `(a + b).sum()`, `a.softmax(-1)`, `a.var(-1)`, `(a @ b).relu() @ c`. Then check with `make_plan` and `len(plan.kernels)`.

2. ★ **Read a plan.** For `x.var(-1)` on a `(4, 8)` input, print the plan's refs and temps and describe in one sentence what each kernel computes.

3. ★★ **Measure the recompute.** Build `(A @ B) @ C` with `fuse_reduce=True` via `make_plan(..., fuse_reduce=True)` and time it against the default at N = 32, 64, 128. Fit the ratio to a power of N.

4. ★★ **A cache-miss detector.** Add a `DEBUG` mode (in your own copy) that, on a cache miss, finds the most similar cached key and prints the first position where they differ. Use it to find the `CONST` that changes in a training step that keeps recompiling.

5. ★★ **Find the bug.** A student renumbers kernel buffers by *slot* order instead of topological order: `bufs = sorted({...}, key=lambda b: b.arg.slot)`. Do identical kernels still render identical C? Construct a counterexample. *Hint:* `new_buffer` hands out slots in creation order.

6. ★★ **Placeholder slots.** Why are placeholder slots negative? What would break if they reused `0, 1, 2, ...`?

7. ★★ **Buffer reuse.** Each temporary of a plan gets its own array. Write a pass over the plan that computes each temp's live range (first and last kernel that touches it) and assigns temps with disjoint live ranges to the same storage. How much memory does it save on the softmax plan? On the five-kernel training step above?

8. ★★★ **Per-reduction fusion.** Replace the global `fuse_reduce` switch with a rule: fuse a `REDUCE` into its consumer unless the consumer would evaluate it inside a reduce loop it does not depend on. *Hint:* you can detect that from shapes alone: is there an `EXPAND` between the `REDUCE` and another `REDUCE` that consumes it? Check that softmax fuses to one kernel and `(A @ B) @ C` stays at two.

9. ★★★ **A better cost model.** `stage_broadcast_compute` ignores the cost of the extra write and read. Model each choice in bytes moved plus ALU ops, with a machine balance parameter (FLOPs per byte, from Chapter 10's roofline), and stage only when the model says so. Compare with the fixed thresholds on the LayerNorm example at `OPT=0` and `OPT=1`.

Solutions to ★ exercises

1. `a + b`: 1 (elementwise, fused). `(a + b).sum()`: 1 (the add fuses into the reduction, and the reduction is the root). `a.softmax(-1)`: 3 (max, sum, output). `a.var(-1)`: the mean is a `REDUCE`, the squared deviations are summed by a second `REDUCE`, and the root multiplies that sum by `1/n`, so 3 (`r`, `r`, `e`). `(a @ b).relu() @ c`: 3, which surprises most people. The first matmul is a `REDUCE` (one kernel) and the second is the root (one kernel), but the ReLU in between is an elementwise computation broadcast 64-fold into the second matmul, and its fused work (2 ALU ops x 4096 elements = 8192) passes the 4096 threshold, so `stage_broadcast_compute` gives it its own `e_64_64` kernel. The real output for 64x64 inputs at `OPT=0`: `['r_64_64', 'e_64_64', 'r_64_64']`. Check each with `len(make_plan([t.uop])[0].kernels)`; at `OPT=1`, the layout pass may add transposing copies (Chapter 10).

2. Kernel 0 (`r_4`) reads the input, writes the row sums (temp 0). Kernel 1 (`r_4`) reads the input and temp 0, writes the row sums of squared deviations from the mean (temp 1). Kernel 2 (`e_4`) reads temp 1 and scales it by `1/8` into the output.

Quiz

1. Name the three kinds of materialization points.
2. Why is a reduction's output cheap to materialize, compared to its input?
3. What does a kernel's `refs` list contain, and why does the `Plan` contain no UOps?
4. Two kernels compute row sums of two different matrices of the same shape. Why do they produce byte-identical C, and why does that matter?
5. What is in `_structure_key` for a `BUFFER`? For a `CONST`?
6. Why did AdamW recompile every step, and what was the fix?
7. When does `stage_broadcast_compute` insert a `STAGE`?
8. Who frees the memory of an intermediate buffer, and when?

Answers

1. Roots (unless already contiguous buffers), `STAGE` nodes, and `REDUCE` nodes (unless `fuse_reduce`).
2. It is smaller by the product of the reduced axes: a row sum of an `(N, N)` matrix writes `N` values after reading `N^2`.
3. One entry per local buffer slot: `("in", i)`, `("tmp", j)`, or `("out", r)`. Without UOps, a cached plan keeps no tensors alive and applies to any inputs with the same structure.
4. Local renumbering in topological order makes their bodies the same hash-consed UOp, so they lower and render identically; the runtime's content-addressed cache then compiles the source once.
5. `("buf", dtype name, shape)`: the slot (identity) is dropped. A `CONST` keeps its value (via `_argkey`), so a different constant is a different key.
6. Its bias corrections were Python floats, so they became `CONST`s that changed every step, changing the structure key. Passing them as one-element tensors makes them data.
7. When an `EXPAND` grows a tensor by at least 4x and the computed (non-`CAST`, non-scalar ALU) value under it has fused work (ALU ops x elements) of at least 4096.
8. The Python garbage collector, when the last reference to the `BUFFER` UOp disappears: the array lives in the UOp's `__dict__`, and `UOp._cache` holds only weak references.

Further reading

- T. Chen et al., "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning", OSDI 2018, section 3 on operator fusion (injective, reduction, and "complex-out-fusable" operators).
- The XLA documentation's description of operator fusion (`openxla.org/xla`), and the "Operation Semantics" page for the HLO ops it fuses.
- J. Ansel et al., "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation", ASPLOS 2024. TorchInductor's scheduler makes the same fuse-or-materialize decisions, with a much larger cost model.
- J. Ragan-Kelley et al., "Halide", PLDI 2013, on the three-way trade-off between parallelism, locality, and redundant recomputation. `compute_root` vs. `compute_inline` is exactly `STAGE` vs. fusion.
- tinygrad's scheduler (`tinygrad/engine/schedule.py` in older releases, the rangeify-based scheduler in newer ones): kernel naming, `DEBUG` levels, and the structural cache in this chapter all follow tinygrad's conventions.

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

uopc/kernelize.py · 220 lines

tests/test_w05_tensor.py · 111 lines

8. Automatic differentiation

Training a neural network means nudging every parameter in the direction that lowers the loss. For that you need the gradient: one partial derivative of a scalar loss with respect to each of (potentially) millions of parameters, every step. Deriving those by hand is out of the question, and estimating them numerically costs one or two extra evaluations of the whole model *per parameter*. **Reverse-mode automatic differentiation** computes all of them for a small constant multiple of the cost of computing the loss once. In `uopc` it is a remarkably short graph transformation: given the shape graph of the loss, build a second shape graph that computes the gradient. That gradient graph is then kernelized, rangeified and compiled like any other graph, by the machinery of the previous two chapters. This chapter derives the algorithm, then derives every one of its rules.

What you'll build

- `gradient(root, targets)`: reverse-mode AD over a DAG of UOps, in 13 lines.
- `_rule(u, g)`: a local derivative rule for every op that can appear in a pure shape graph, each derived from first principles: ALU ops, the four movement ops, `REDUCE` (sum and max, with ties), `STACK` and `INDEX`.
- `Tensor.backward`, which hangs the gradients on the parameters as lazy tensors.
- Gradient checking with finite differences, and second derivatives for free.

Where this fits. Autodiff runs on the shape graph, *before* the kernelizer (Chapter 7) and rangeify (Chapter 6). It uses the composite ops `PAD` and `sum_to` from `uopc/composite.py` (Chapter 6). Chapter 9 builds a training loop on top of `backward`, and Chapter 14 trains a GPT with it.

8.1 Three ways to get a derivative

Take `f(x) = sum(x * y + x)` for vectors `x` and `y`. There are three ways to get `df/dx`.

Symbolic differentiation applies calculus rules to the *expression* and produces a new expression: `df/dx_i = y_i + 1`. This is what a computer algebra system does. Applied naively to a program, it suffers from expression swell: the derivative of a product of `n` factors has `n` terms each with `n - 1` factors, and shared subexpressions get duplicated.

Numerical differentiation treats `f` as a black box: `df/dx_i ~ (f(x + eps e_i) - f(x - eps e_i)) / (2 eps)`. It is easy and indispensable for *testing* (see "Gradient checking" below), but it costs two evaluations of `f` per input element, and its accuracy is limited by the tension between truncation error (eps too large) and floating-point cancellation (eps too small).

Automatic differentiation works on the *program*, not the expression: it applies the chain rule to the sequence (or graph) of primitive operations that computes `f`, reusing intermediate values. Its result is exact up to floating-point rounding, and its cost is a constant multiple of the cost of `f`. Because UOp graphs are hash consed, our "symbolic" output graph never duplicates a shared subexpression either: AD on a DAG gets the benefits of both.

8.2 Forward mode and reverse mode

Think of the program as a composition of functions, `f = f_3 . f_2 . f_1`, with `x -> h1 -> h2 -> L`. The chain rule says its Jacobian is a product of the Jacobians of the pieces:

```
J_f = J_3 J_2 J_1          (J_k is the Jacobian of f_k at its input)
```

You never want to build these matrices: a `(1000,)`-to-`(1000,)` elementwise op has a 10^6-entry Jacobian that is diagonal. You want products of them with vectors, and the order in which you multiply decides the cost.

- **Forward mode** multiplies right to left, starting from a direction `v` in input space: `J_3 (J_2 (J_1 v))`. Each pass gives one *column* of `J_f`, the derivative of every output with respect to one input direction. Getting the full gradient of a scalar loss with respect to `n` parameters takes `n` passes. (Dual numbers, `a + b*eps` with `eps^2 = 0`, are forward mode.)
- **Reverse mode** multiplies left to right, starting from a covector `w` in output space: `((w^T J_3) J_2) J_1`. Each pass gives one *row*: the derivative of one output with respect to every input. For a scalar loss, `w = 1` and one pass gives the entire gradient.

A loss is a scalar and a model has millions of parameters, so ML uses reverse mode. The price is memory: the backward pass multiplies by `J_1` *last*, and `J_1` depends on `x` (for `MUL`, the derivative with respect to one factor is the other factor), so every intermediate value needed by a local derivative must be kept around until the backward pass reaches it. In our lazy graph world, "kept around" just means "the gradient graph refers to the forward node", and the kernelizer decides whether that node becomes a buffer (Chapter 7).

> **Key idea:** reverse mode propagates `dL/d(node)`, the sensitivity of the one scalar output to each intermediate value, from the output back to the inputs. Each op only needs to know how to turn the sensitivity of its output into sensitivities of its inputs: a vector-Jacobian product.

8.3 The chain rule on a DAG

Programs are not chains; they are DAGs. In `x * y + x`, `x` is used twice. The multivariate chain rule handles fan-out: if `L` depends on `x` through several uses `u_1, ..., u_k`, then

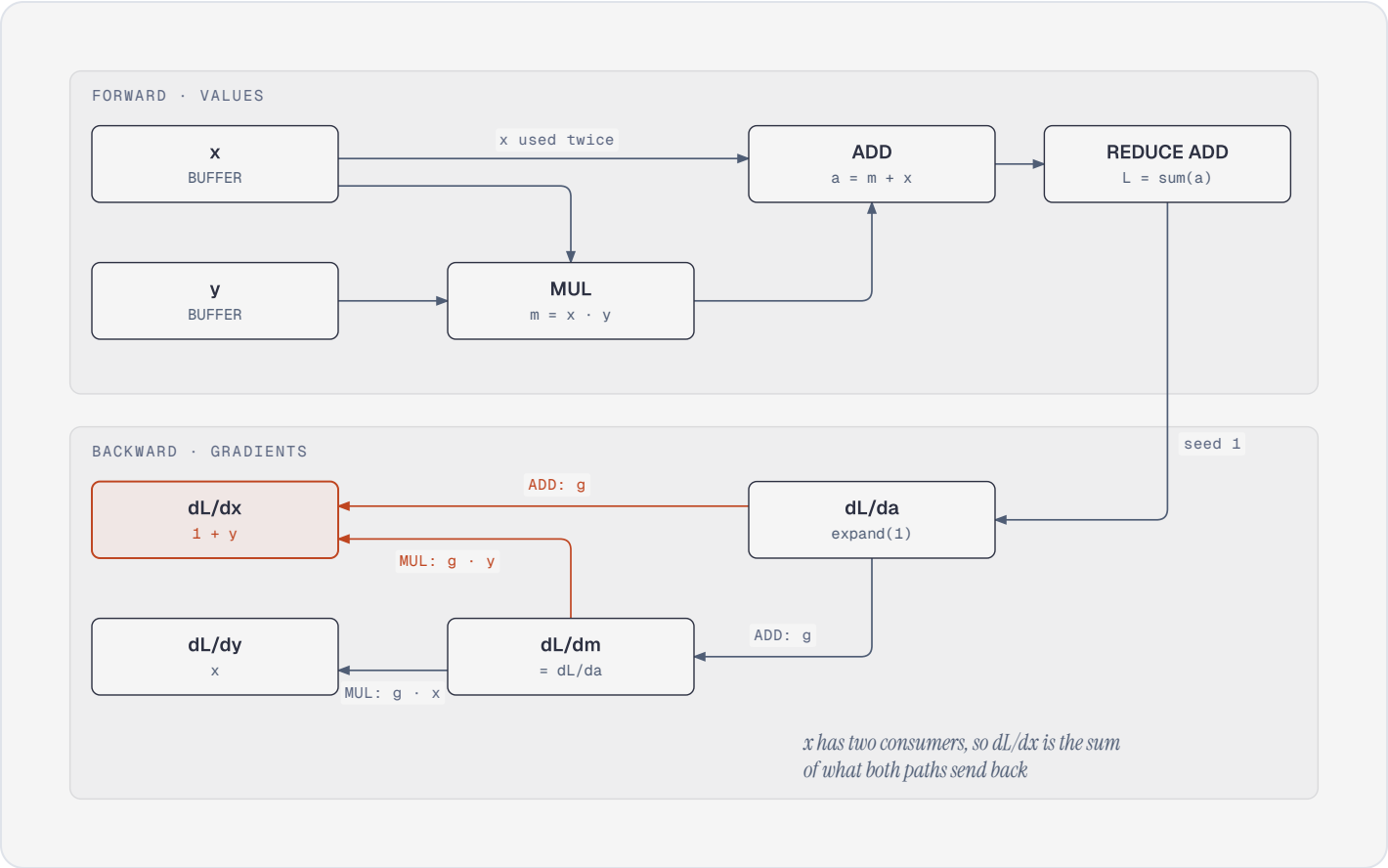
```
dL/dx = sum over uses k of  (dL/du_k) * (du_k/dx)
```

Contributions from different paths *add up*. Reverse-mode AD on a DAG is therefore:

1. Seed `dL/dL = 1`.
2. Visit nodes in reverse topological order, so that when a node is visited every one of its consumers has already been visited, and its gradient is complete.
3. For each node `u` with gradient `g`, use `u`'s local rule to compute a contribution for each source, and *add* it to that source's running gradient.

For `L = sum(x * y + x)`:

```
forward:   m = x * y      a = m + x      L = sum(a)
backward:  dL/da = expand(1)                        REDUCE(ADD) -> EXPAND
           dL/dm += dL/da ;   dL/dx += dL/da        ADD passes g to both sources
           dL/dx += dL/dm * y ; dL/dy += dL/dm * x  MUL: each source gets g times the other
result:    dL/dx = 1 + y ,  dL/dy = x
```



8.4 Why only pure shape graphs

The syllabus says autodiff works "only on pure shape graphs (not STORE, CALL, AFTER, RANGE, END)". The reasons are worth spelling out.

- `STORE` and `AFTER` are effects and orderings on memory. The derivative of "write `v` to address `p`" is not a value; differentiating through memory requires recording what was overwritten (a "tape"), which is what imperative AD systems like PyTorch's autograd do at run time.
- `RANGE` and `END` are loops with an iteration count. Differentiating a loop in reverse means running it backward, which needs every iteration's state (or recomputation, "checkpointing"). Compilers like Enzyme do this on LLVM IR, at considerable complexity.
- `CALL` could be inlined first; the shape graph never contains one.

A shape graph has none of those problems. Every node is a pure function of its sources, applied to whole tensors, so every local derivative is again a pure function on whole tensors, expressible in the same ops. That is why autodiff runs *before* rangeify: after rangeify, the graph is full of loops and stores. This placement is the same as in JAX (which differentiates `jaxpr`s, a pure array IR, before XLA lowers them) and tinygrad.

8.5 gradient(): the algorithm

Here is the whole driver:

uopc/autodiff.py · gradient

```
def gradient(root: UOp, targets: list[UOp]) -> list[UOp]:
  topo = root.toposort()
  # only walk nodes that lie on a path to some target
  on_path = set(targets)
  for u in topo:
    if any(x in on_path for x in u.src): on_path.add(u)
  grads: dict[UOp, UOp] = {root: _full(root, 1.0)}
  for u in reversed(topo):
    if u not in grads or u not in on_path or u in targets and not u.src: continue
    for x, gx in zip(u.src, _rule(u, grads[u])):
      if gx is None or x not in on_path: continue
      grads[x] = grads[x] + gx if x in grads else gx
  return [grads.get(t, zeros_like(t)) for t in targets]
```

Line by line:

- `topo = root.toposort()` lists every node the loss depends on, children before parents.
- `on_path` is the set of nodes that depend on *some target*. A node not on the path cannot carry gradient to a target, so its rule is never applied. In a training step this prunes the entire data-preparation part of the graph: there is no point computing a gradient for the input images. It is computed in one forward sweep: a node is on the path if any source is.
- `grads` starts with the seed: a tensor of ones shaped like the root (for a scalar loss, `CONST 1.0`). `_full` builds a broadcast constant, like `const_like` (NOTES 13: the constant must have the node's shape, or the ALU type rule rejects it).
- The loop walks `reversed(topo)`, parents before children. It skips nodes that have no gradient (nothing on the path consumed them), nodes off the path, and leaf targets (a `BUFFER` has no sources to pass gradient to).
- For each source, `_rule` returns a gradient contribution or `None` ("this input is not differentiable", like the condition of a `WHERE`). Contributions *add*: `grads[x] + gx` is an `ADD` UOp, which is the fan-out sum of the chain rule.
- Finally, one gradient per target. A target the loss does not depend on gets zeros of its shape, not an error (`gradient(f, [z])` for an unrelated `z` returns a `(4,)`-shaped `EXPAND` of 0.0).

Note what this function does *not* do: it computes no numbers. Everything it builds is a UOp graph. Here is the gradient of `sum(x * y + x)` with respect to `x`, printed as `.uir`:

```
%0 = reshape 1.0, (1,)
%1 = expand %0, (3,)
%2 = buffer 3 : dtype=f32 slot=1
%3 = mul %1, %2
%4 = add %1, %3
```

`%1` is `dL/da`, the seed broadcast back over the reduced axis. `%3` is the `MUL` rule's contribution `g * y`, and `%4` adds the `ADD` rule's contribution `g`. Lowered, it is one line of C:

```
void grad_x(float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    float val1 = data1[ridx0];
    data2[ridx0] = (val1+1.0f);
  }
}
```

The simplifier removed the multiplication by one. Note the parameter list: the gradient with respect to `x` does not read `x` at all, so the kernel takes only `y` (`data1`) and the output. On `x = [1, 2, 3]`, `y = [10, 20, 30]`:

```
dloss/dx = [11. 21. 31.]  expected y+1 = [11. 21. 31.]
dloss/dy = [1. 2. 3.]  expected x = [1. 2. 3.]
```

Sharing between forward and backward

The gradient graph refers to forward nodes by identity: `dL/dy = g * x` contains the very `x` node of the forward graph, and the gradient of a matmul contains the forward activations. When the loss and the gradients are kernelized together, any forward node that both need is one node, materialized at most once. Chapter 7 showed it on a training step: the forward `X @ W` is one kernel whose output is read by both the loss kernel and the gradient kernel. No "saved tensors" machinery is needed; hash consing is the tape.

8.6 The rules

Every rule maps `(u, g)`, a node and the gradient of the loss with respect to it (same shape as `u`), to one gradient per source (same shape as that source), or `None`.

```
uopc/autodiff.py · _rule
```

```
def _rule(u: UOp, g: UOp) -> tuple[UOp | None, ...]:
  s = u.src
  match u.op:
    case Ops.ADD: return (g, g)
    case Ops.MUL: return (g * s[1], g * s[0])
    case Ops.MAX:
      a, b = s
      half = g * _full(g, 0.5)
      zero = zeros_like(g)
      return ((b < a).where(g, (a < b).where(zero, half)), (a < b).where(g, (b < a).where(zero,
half)))
    case Ops.RECIP: return (-(g * u * u),)
    case Ops.EXP2: return (g * u * _full(u, LN2),)
    case Ops.LOG2: return (g * s[0].recip() * _full(u, 1 / LN2),)
    case Ops.SQRT: return (g * u.recip() * _full(u, 0.5),)
    case Ops.SIN: return (g * UOp(Ops.SIN, (s[0] + _full(u, math.pi / 2),)),)
    case Ops.CAST: return (g.cast(s[0].dtype) if s[0].dtype.is_float else None,)
    case Ops.WHERE: return (None, s[0].where(g, zeros_like(g)), s[0].where(zeros_like(g), g))
    case Ops.CMPLT | Ops.CMPNE | Ops.FLOORDIV | Ops.FLOORMOD: return (None,) * len(s)
    case Ops.RESHAPE: return (g.reshape(s[0].shape),)
    case Ops.EXPAND: return (sum_to(g, s[0].shape),)
    case Ops.PERMUTE:
      inv = [0] * len(u.arg)
      for i, p in enumerate(u.arg): inv[p] = i
      return (g.permute(inv),)
    case Ops.SHRINK: return (pad(g, tuple((b, n - e) for (b, e), n in zip(u.arg, s[0].shape))),)
    case Ops.STACK: return tuple(g.index(i) for i in range(len(s)))
    case Ops.STAGE: return (None,) if u.arg == "detach" else (g,)
    case Ops.REDUCE:
      rop, n = u.arg
      x = s[0]
      ge = g.reshape(u.shape + (1,) * n).expand(x.shape)
      if rop is Ops.ADD: return (ge,)
      mx = u.reshape(u.shape + (1,) * n).expand(x.shape)
      mask = x.eq(mx).cast(x.dtype)                      # ties share the gradient
      cnt = mask.reduce(Ops.ADD, n).reshape(u.shape + (1,) * n).expand(x.shape)
      return (ge * mask * cnt.recip(),)
    case Ops.INDEX:
      x, idx = s[0], s[1:]
      if not all(i.op is Ops.CONST for i in idx): raise NotImplementedError("gradient of a data-
dependent INDEX (use a one-hot matmul)")
      gx = g
      for i, n in zip(reversed(idx), reversed(x.shape[:len(idx)])):
        z = zeros_like(gx)
        gx = UOp(Ops.STACK, tuple(gx if k == i.arg else z for k in range(n)))
      return (gx,) + (None,) * len(idx)
    case Ops.CONST | Ops.BUFFER | Ops.PARAM: return ()
  raise NotImplementedError(f"no gradient rule for {u.op} (autodiff only works on pure shape
graphs)")
```

The helper `_full(like, v)` is a constant `v` broadcast to `like`'s shape:

uopc/autodiff.py · _full

```python
def _full(like: UOp, v: float) -> UOp:
  c = UOp.const(v, like.dtype)
  return c.reshape((1,) * len(like.shape)).expand(like.shape) if like.shape else c
```

Below, each rule is derived. Notation: `u` is the node's output, `a`, `b` (or `x`) its sources, `g` is `dL/du` and the job is `dL/da = g * du/da` for elementwise ops (a vector-Jacobian product with a diagonal Jacobian), or the appropriate generalization for ops that move elements around.

ALU ops

ADD. `u = a + b`, `du/da = du/db = 1`. Both sources get `g`. Addition is a gradient *distributor*.

MUL. `u = a * b`, `du/da = b`, `du/db = a`. So `(g * b, g * a)`: each source gets the gradient times the *other* factor. This is the rule that makes reverse mode keep forward values alive. For `x * x`, both contributions go to `x` and add up to `2 g x`, which is the derivative of `x^2`: fan-out summation at work.

RECIP. `u = 1/x`, `du/dx = -1/x^2 = -u^2`. The rule reuses the forward output `u` instead of recomputing `1/x`: `-(g * u * u)`.

EXP2. `u = 2^x = e^(x ln 2)`, so `du/dx = ln 2 * 2^x = u * ln 2`. Again the output is reused.

LOG2. `u = ln(x) / ln 2`, so `du/dx = 1 / (x ln 2)`: `g * (1/x) * (1/ln 2)`.

SQRT. `u = x^(1/2)`, `du/dx = 1 / (2 sqrt(x)) = 0.5 / u`.

SIN. `du/dx = cos(x)`. There is no `COS` op, so it is written `sin(x + pi/2)`, the same trick `Tensor.cos` uses.

The Tensor library builds everything else from these. `exp(x) = exp2(x * log2(e))` gets its derivative from the chain rule through `MUL` and `EXP2`: `exp2(x log2 e) * ln 2 * log2(e) = exp(x)`. `tanh`, `sigmoid`, `gelu`, `log_softmax`: none of them has a rule of its own. Their gradients are assembled from the primitive rules by `gradient()`, and they are tested numerically (below).

CAST. Converting between float types is the identity up to rounding, so the gradient is `g` converted back: `g.cast(s[0].dtype)`. Casting *from* an integer or bool has no derivative (the input is discrete), so the rule returns `None`.

WHERE. `u = c ? a : b`. For each element, `u` is either `a` or `b`, so the gradient goes to whichever was selected: `dL/da = c ? g : 0`, `dL/db = c ? 0 : g`. The condition is a bool and gets `None`.

CMPLT, CMPNE, FLOORDIV, FLOORMOD. Comparisons produce bools; integer division produces integers. Piecewise constant functions have derivative zero wherever they have one, and the derivative is undefined at the jumps. Returning `None` for every source says "no gradient flows here", which is the useful convention.

MAX, and the problem of ties

`u = max(a, b)` is differentiable wherever `a != b`: the gradient goes entirely to the larger input. At `a == b` the function has a kink and no derivative; any convex combination of the two one-sided derivatives is a valid *subgradient*. The rule picks the symmetric one, half each:

```
dL/da = g     if a > b,   0 if a < b,   g/2 if a == b
```

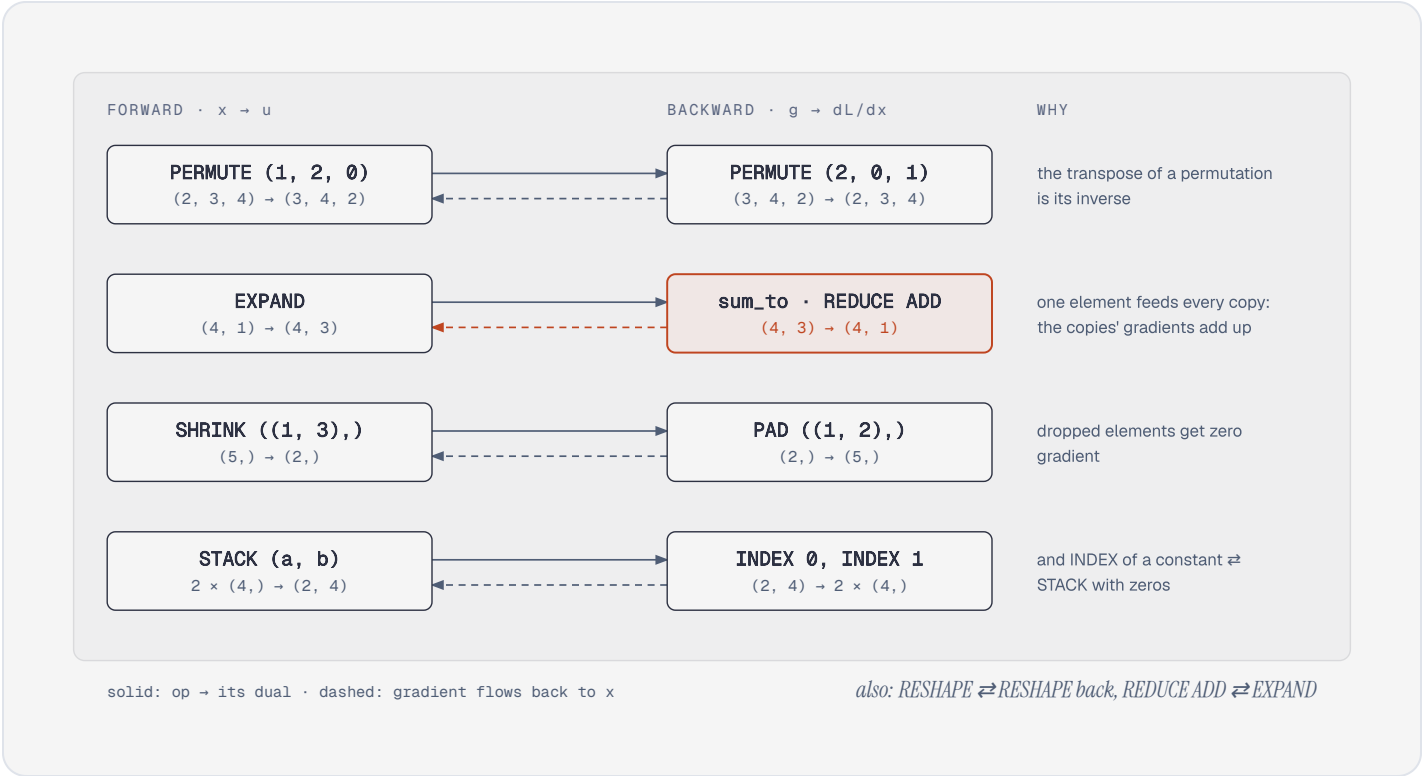
That is what the nested `WHERE`s implement. The choice is not cosmetic: it means `abs(x) = max(x, -x)` has gradient `g/2 - g/2 = 0` at `x = 0`, the conventional subgradient, while an asymmetric rule would give `+g` or `-g`. Two definitions of the same mathematical function can still differ at the kinks:

```
d|x|/dx  at [-2, 0, 3]: [-1.  0.  1.]
relu'    at [-2, 0, 3]: [0. 0. 1.]
max(x,0)' at [-2, 0, 3]: [0.  0.5 1. ]
```

`Tensor.relu` is `(x > 0).where(x, 0)`, so its derivative at 0 comes from the `WHERE` rule (the condition is false, gradient 0). `x.maximum(0)` goes through the `MAX` rule and splits the tie. Both are "correct". With randomly initialized float weights, exact ties are rare in practice, but they are not rare in tests that use integers, or after a ReLU has produced many exact zeros. Pick one definition, use it everywhere, and know what it does at the kink.

Movement ops: transposes of index maps

Every movement op is *linear*, and better: each output element is a copy of exactly one input element (or several outputs are copies of the same input, for `EXPAND`). For such a map, the vector-Jacobian product is "send each output's gradient back to the input element it was copied from, and add up the gradients of all outputs that copied the same input". Each rule is the dual of its forward op.



RESHAPE. A reshape is a bijection between element positions (same row-major order). The gradient is reshaped back: `g.reshape(s[0].shape)`.

PERMUTE. `u = x.permute(p)`; output axis `k` is input axis `p[k]`. The gradient has `u`'s axes and must be brought back to `x`'s: input axis `m` is output axis `k` where `p[k] = m`. That is the inverse permutation `inv[p[k]] = k`, so the rule is `g.permute(inv)`. (Same inverse as in rangeify's `PERMUTE` rule, for the same reason: a permutation's Jacobian is a permutation matrix, whose transpose is its inverse.)

EXPAND. `u[i, j] = x[i, 0]` for every `j`: one input element feeds every output along the expanded axis. By the fan-out rule, its gradient is the *sum* of their gradients: `dL/dx[i, 0] = sum_j g[i, j]`. So the gradient of a broadcast is a reduction, written as `sum_to`:

`uopc/composite.py · sum_to`

```
def sum_to(g: UOp, shape: tuple[int, ...]) -> UOp:
  """Reverse of EXPAND: sum g over the axes where `shape` has 1 and g doesn't."""
  axes = [i for i, (s, t) in enumerate(zip(g.shape, shape)) if s != t]
  if not axes: return g
  keep = [i for i in range(len(shape)) if i not in axes]
  return g.permute(keep + axes).reduce(Ops.ADD, len(axes)).reshape(shape)
```

`sum_to` finds the axes where the target shape has 1 and `g` does not, permutes them to the end, reduces them (recall `REDUCE` pops the last axes), and reshapes back to the size-1 form. This is the rule that turns a

bias added to every row into a column sum in the backward pass, and the one that turns the broadcast inside a matmul into the *other* matmul of the backward pass.

SHRINK. `u = x[b:e]` (per axis). Elements inside the window are copied, elements outside are dropped. The gradient of a dropped element is 0, so the gradient is `g` placed back at offset `b` inside a zero tensor of `x`'s shape: `pad(g, (b, n - e))`. SHRINK and PAD are each other's duals. Since `pad` is the composite from Chapter 6, the gradient graph of a slice contains `STACK`s, reshapes and shrinks:

```
%0 = reshape 0.0, (1,)
%1 = expand %0, (4,)
%2 = reshape 1.0, (1,)
%3 = expand %2, (2,)
%4 = expand %0, (2,)
%5 = stack %3, %4
%6 = reshape %5, (4,)
%7 = stack %1, %6
%8 = reshape %7, (8,)
%9 = shrink %8, ((3, 8),)
```

That is `d sum(v[1:3]) / dv` for a length-5 `v`, and it lowers to a single bounds-checked select. Note that the gradient doesn't read `v` at all (the slice is linear), so the kernel has no input:

```
void grad_shrink(float* restrict data0) {
  for (int ridx0 = 0; ridx0 < 5; ridx0++) {
    int alu1 = (ridx0+3);
    data0[ridx0] = (((alu1/4)<1)?0.0f:((((alu1/2)%2)<1)?1.0f:0.0f));
  }
}
```

```
d sum(v[1:3])/dv = [0. 1. 1. 0. 0.]
```

This is also why autodiff never needs `FLIP`: the gradient of a convolution built from shrinks (im2col, Chapter 6) is a sum of pads.

REDUCE

Sum. `u[i] = sum_k x[i, k]`, so `du[i]/dx[i, k] = 1`: every element of the reduced group gets the group's gradient. `REDUCE(ADD)` and `EXPAND` are duals, mirror images of the previous rule. In code: reshape `g` from `u.shape` to `u.shape + (1,)*n` and expand to `x.shape`.

Max. `u[i] = max_k x[i, k]`. Away from ties, the gradient goes to the arg-max element only. With ties, the rule splits it evenly among all maximal elements, the same convention as elementwise `MAX`:

```
mask[i, k] = (x[i, k] == u[i])          as a float
cnt[i]     = sum_k mask[i, k]
dL/dx      = g[i] * mask[i, k] / cnt[i]
```

The code builds exactly that: `mx` is `u` broadcast back, `mask` compares, `cnt` is a sum-reduce of the mask broadcast back, and the result is `ge * mask * (1 / cnt)`. On `x = [1, 3, 3]`:

```
d max(x)/dx at [1. 3. 3.] = [0.  0.5 0.5]
```

and for the elementwise version with ties at index 1:

```
d sum(max(x,y)) / dx = [1.  0.5 0. ]  / dy = [0.  0.5 1. ]
```

(For comparison, PyTorch's `torch.amax` documents the same even split; `torch.max(dim=...)` sends the gradient to the single returned index.)

Note the cost: the backward of a max-reduce contains *another* reduction (the count), so it becomes an extra kernel. Softmax uses `max` only for numerical stability and its gradient through the max is mathematically zero (softmax is invariant to shifting its input), but the graph does not know that and computes it anyway.

STACK and INDEX

STACK. `u = STACK(a_0, ..., a_{n-1})`, so `u[k] = a_k`. Source `k`'s gradient is slice `k` of `g`: `g.index(k)`.

INDEX with constant indices. `u = x.index(c)` selects `x[c]`. The dual of selecting one slice is placing the gradient into a zero tensor at that slice, which is a `STACK` with `g` at position `c` and zeros elsewhere. For several indices the code does that one axis at a time, innermost first. The indices themselves are integers and get `None`.

INDEX with data-dependent indices (`x.index(i)` where `i` is computed from data, like an embedding lookup with token ids) is not supported:

```
NotImplementedError: gradient of a data-dependent INDEX (use a one-hot matmul)
```

The dual of a gather is a *scatter-add* (`dL/dx[i] += g` for the selected `i`, accumulating when the same index is selected twice), and scatter-add needs an op this IR does not have. The workaround in the error message is how the GPT of Chapter 14 does embeddings: multiply a one-hot matrix by the embedding table. A one-hot matmul is a gather written with ops we can differentiate, at the cost of `vocab` times more arithmetic. (Exercise 9 asks for a real scatter.)

uopc/nn.py · Embedding

```python
class Embedding:
  """A one-hot matmul: differentiable without a scatter op."""
  def __init__(self, vocab: int, dim: int):
    self.vocab, self.weight = vocab, Tensor.randn(vocab, dim) * 0.02
    self.weight = self.weight.realize(); self.weight.requires_grad = True
  def __call__(self, idx: Tensor) -> Tensor: return idx.one_hot(self.vocab) @ self.weight
```

The one-hot matrix is built by comparison (`idx.unsqueeze(-1).eq(arange(n))`), whose gradient is `None`, and cast to float, so the only gradient path is through the matmul into `weight`: the `EXPAND` rule's `sum_to` adds up the gradients of every row that selected the same token, which is exactly the scatter-add, computed densely.

STAGE. Materialization is the identity function, so the gradient passes through: `(g,)`. With one exception: a `STAGE` whose arg is `"detach"` returns `(None,)`. That is how `Tensor.detach()` is implemented:

uopc/tensor.py · Tensor.detach

```python
  def detach(self) -> Tensor:
    """Same value, but gradients stop here. (Kernelize strips the marker, so it costs nothing to
run.)"""
    return Tensor(UOp(Ops.STAGE, (self.uop,), "detach"))
```

The marker means "same value, but treat it as a constant when differentiating". It costs nothing at run time: the kernelizer's `strip_detach` removes it before planning (Chapter 7), so it never becomes a kernel. Detaching is how you write a stop-gradient, for example in a target that should not be trained toward. With `w = [1, 2]`, `sum(w * w.detach())` treats the second factor as a constant, so its gradient is `w` rather than `2w`:

```
grad with detach:    [1. 2.]
```

CONST, BUFFER, PARAM are leaves: no sources, nothing to return.

Anything else (`STORE`, `RANGE`, ...) raises `NotImplementedError("no gradient rule for ... (autodiff only works on pure shape graphs)")`.

8.7 Worked examples: the gradients you will actually train with

The rules are local. It is worth checking, twice, that composing them gives the textbook formulas for the two gradients that dominate every training run.

Matrix multiplication

The Tensor library computes `Y = X @ W` for `X` of shape `(M, K)` and `W` of shape `(K, N)` as

```
Xe = X.reshape(M, 1, K).expand(M, N, K)
We = W.T.reshape(1, N, K).expand(M, N, K)          (W.T is PERMUTE (1, 0))
Y  = (Xe * We).reduce(ADD, 1)                       shape (M, N)
```

Push a gradient `G = dL/dY` of shape `(M, N)` back through it with the rules:

1. `REDUCE(ADD)`: `G` is reshaped to `(M, N, 1)` and expanded to `(M, N, K)`. Call it `Ge`; its value at `(i, j, k)` is `G[i, j]`.
2. `MUL`: the `Xe` side gets `Ge * We`, the `We` side gets `Ge * Xe`.
3. `EXPAND` on the `X` side: `sum_to` sums `Ge * We` over axis 1 (the `N` axis that `X` was broadcast along): `dL/dX[i, k] = sum_j G[i, j] * W[k, j]`. That is `G @ W^T`.
4. `EXPAND` on the `W` side sums `Ge * Xe` over axis 0 (the `M` axis): `sum_i G[i, j] * X[i, k]`, of shape `(N, K)`. The `RESHAPE` and inverse `PERMUTE` rules bring it back to `(K, N)`: `dL/dW = X^T @ G`.

Two matmuls, each the same size as the forward one, both produced by the `EXPAND` rule: the broadcast that turned a matrix product into an elementwise product turns, in the backward pass, into the reduction of the

other matrix product. Nobody wrote a "matmul backward". This is also the `r_10_64` kernel of the training step in Chapter 7 (`X^T @ G`, of shape `(10, 64)` before the final transposing copy).

Softmax cross-entropy

For logits `z` (one row) and a one-hot label `y`, `Tensor.cross_entropy` computes `L = -sum_c y_c log_softmax(z)_c` and averages over the batch, with `log_softmax(z) = (z - m) - log(sum_c exp(z_c - m))` and `m = max(z)`. By hand: the derivative of `log(sum exp)` with respect to `z_c` is `softmax(z)_c`, the `m` terms cancel (shifting all logits by a constant changes nothing), so

```
dL/dz_c = (softmax(z)_c - y_c) / B
```

for a batch of `B` rows. The autodiff graph knows none of this. It applies `MUL`, `ADD`, `LOG2`, `EXP2`, `REDUCE(ADD)`, `REDUCE(MAX)` (with its mask and count) and `EXPAND` rules one by one, and the result must agree:

```
logits = np.array([[2.0, 1.0, 0.1], [0.5, 2.5, 0.3]], np.float32)
labels = np.array([0, 1], np.int32)
t = Tensor(logits, requires_grad=True)
t.cross_entropy(Tensor(labels)).backward([t])
```

```
autodiff:
 [[-0.17049944  0.12121649  0.04928295]
 [ 0.05430186 -0.09876046  0.0444586 ]]
(softmax - onehot) / B:
 [[-0.17049944  0.12121648  0.04928295]
 [ 0.05430187 -0.09876049  0.04445861]]
```

Equal to the last float32 digit or so. The max's contribution is present in the graph (it is not simplified away) and sums to zero numerically, which is the "mathematically zero" gradient through the max mentioned above.

8.8 From gradient() to Tensor.backward

The Tensor library's `backward` is a loop over `gradient`'s results:

uopc/tensor.py · Tensor.backward

```
def backward(self, params: list[Tensor]) -> Tensor:
  """Set p.grad = d(self)/d(p) for each p. Gradients are lazy Tensors like everything else."""
  assert self.shape == (), "backward needs a scalar"
  root = getattr(self, "_graph", self.uop)
  for p, g in zip(params, gradient(root, [p.uop for p in params])):
    p.grad = Tensor(g) if p.grad is None else p.grad + Tensor(g)
  return self
```

Three details matter to users. The loss must be a scalar (the seed `1.0` only makes sense for one output). The gradients are lazy `Tensor`s: nothing is computed until the optimizer realizes them, together with the parameter updates, in one `realize_all` (one plan, one cache entry). And gradients *accumulate*: a second `backward` adds to `p.grad`, which is why optimizers have `zero_grad`.

The `getattr(self, "_graph", self.uop)` line handles a subtle interaction with the kernelizer. When a tensor is realized (by `.item()`, `.numpy()`, or an optimizer's `realize_all`), its `uop` is rebound to the output `BUFFER`, and a buffer is a leaf with no gradient path to anything. Printing the loss before calling `backward` would therefore silently produce zero gradients. `realize_all` keeps the pre-realization graph in `t._graph`, and `backward` differentiates that:

```
loss = (w * w).sum()
print("loss.item() =", loss.item())
loss.backward([w]); print("grad after .item():", w.grad.numpy())
```

```
loss.item() = 5.0
grad after .item(): [2. 4.]
```

The price is that the gradient graph refers to the forward *graph*, not to the buffer `.item()` produced, so the forward work the gradients need is computed again when they are realized (in the same plan as the updates, so it is still one realization per step). Realizing loss and gradients together, as the optimizers do, avoids the repeat.

8.9 Gradient checking

Every rule above is one line, which means every rule is one typo away from a plausible wrong answer. The oracle is numerical differentiation with **central differences**:

```
df/dx_i ~ (f(x + eps e_i) - f(x - eps e_i)) / (2 eps)
```

Its truncation error is O(eps^2) (the even terms of the Taylor expansion cancel), against O(eps) for the one-sided difference. The week-5 tests compute it in float64 with NumPy, as an independent implementation of the same function:

tests/test_w05_tensor.py · numgrad

```python
def numgrad(f, x, eps=1e-3):
  g = np.zeros_like(x, dtype=np.float64)
  for i in np.ndindex(x.shape):
    xp, xm = x.astype(np.float64), x.astype(np.float64)
    xp[i] += eps; xm[i] -= eps
    g[i] = (f(xp) - f(xm)) / (2 * eps)
  return g
```

tests/test_w05_tensor.py · test_gradients

```python
@pytest.mark.parametrize("name,fn,npfn", [
  ("tanh_sum", lambda t: t.tanh().sum(), lambda x: np.tanh(x).sum()),
  ("softmax_xent", lambda t: t.log_softmax(-1)[:, 1].sum(), lambda x: (x -
np.log(np.exp(x).sum(-1, keepdims=True)))[:, 1].sum()),
  ("max", lambda t: (t.max(-1) * t.sum(0).max()).sum(), lambda x: (x.max(-1) *
x.sum(0).max()).sum()),
  ("pad_shrink", lambda t: t.pad(((1, 1), (0, 2)))[1:3, 1:].square().sum(),
   lambda x: (np.pad(x, ((1, 1), (0, 2)))[1:3, 1:] ** 2).sum()),
  ("sqrt_recip", lambda t: (t.square() + 1).sqrt().reciprocal().sum(), lambda x: (1 / np.sqrt(x
** 2 + 1)).sum()),
  ("matmul", lambda t: (t @ t.T).sin().sum(), lambda x: np.sin(x @ x.T).sum()),
])
def test_gradients(name, fn, npfn):
  x = r(3, 4)
  t = Tensor(x, requires_grad=True)
  fn(t).backward([t])
  np.testing.assert_allclose(t.grad.numpy(), numgrad(npfn, x), rtol=2e-2, atol=2e-3)
```

Three points of craft. The reference function `npfn` is plain NumPy, not `uopc`, so a bug in a shared component cannot cancel itself out. The differences are taken in float64, so `eps = 1e-3` gives truncation errors around 1e-6 without cancellation trouble; `uopc` computes in float32, hence the tolerances `rtol=2e-2, atol=2e-3`. And the test functions are chosen to cover rules: `tanh` (EXP2, RECIP, MUL), `log_softmax` with a slice (MAX-reduce, SHRINK, LOG2, EXPAND), a max of sums (both reductions), pad-then-shrink (SHRINK and the composite PAD), `sqrt` and `recip`, and `sin` of a matmul (PERMUTE, EXPAND, REDUCE).

Here is the same check for a graph that exercises `STACK` and `INDEX` (via `t.stack_with(t * t)[1]`):

```
x = np.random.default_rng(0).standard_normal((2, 3)).astype(np.float32)
t = Tensor(x, requires_grad=True)
loss = (t.stack_with(t * t)[1].sum(0) * Tensor(np.array([1., 2., 3.], np.float32))).sum()
loss.backward([t])
```

```
autodiff:
 [[ 0.25146043 -0.52841944  3.842536  ]
 [ 0.20980023 -2.1426775   2.1695704 ]]
finite differences:
 [[ 0.25146043 -0.52841944  3.842536  ]
 [ 0.20980023 -2.1426775   2.1695704 ]]
```

Gradient checks should avoid kinks: a finite difference across `max(a, b)` at `a == b` averages the two one-sided slopes and agrees with the tie rule only by coincidence. Random float inputs avoid exact ties with probability 1.

8.10 Higher-order derivatives

`gradient` takes a graph and returns a graph. Nothing stops you from differentiating its output:

```
x = buf(0, 4)
f = (x * x * x).reduce(Ops.ADD, 1)                  # sum(x^3)
(g,) = gradient(f, [x])                             # 3x^2, a shaped graph like any other
(h,) = gradient(g.reduce(Ops.ADD, 1), [x])          # d/dx sum(3x^2) = 6x
```

```
g = [ 3. 12. 27. 48.]
h = [ 6. 12. 18. 24.]
```

```
void second_derivative(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    float alu2 = (val1*2.0f);
    data1[ridx0] = (((alu2+alu2)+val1)+val1);
  }
}
```

`(2x + 2x) + x + x = 6x`: the second derivative arrives as a sum of the contributions of every path, just as the chain rule on a DAG says, and float reassociation (which the simplifier refuses to do; NOTES 7) leaves it in that form. Because the gradient graph uses only ops that have rules, every gradient graph is differentiable again. Systems with a hand-written backward kernel per op (as in many early frameworks) have to provide a second set of rules for double backward; graph-to-graph AD gets it for free.

8.11 What gradients cost

A rule of thumb: the backward pass costs about twice the forward. Each `MUL` in the forward becomes two in the backward (one per source). Each matmul (an `EXPAND` - `MUL` - `REDUCE`) becomes two matmuls in the backward: the `EXPAND`s turn into reductions via `sum_to`, and the `REDUCE` turns into an `EXPAND`. And every elementwise op in the forward pass leaves a trail of elementwise ops in the backward pass, which the kernelizer fuses into the neighboring matmul kernels. In the training step of Chapter 7, the ReLU's backward (`WHERE` of a comparison) and the square's backward (`g * x + g * x`) ended up inside the inner loop of the weight-gradient matmul:

```
        _Bool alu3 = (0.0f<val2);
        float alu4 = ((alu3?val2:0.0f)*0.0031250000465661287f);
        float val5 = data1[((ridx2*64)+ridx1)];
        acc0 = (val1+((alu3?(alu4+alu4):0.0f)*val5));
```

(`0.0031250000465661287` is `1/320` rounded to float32 by the constant folder, NOTES 5; the forward kernel printed the same constant as `0.003125f`.) Making this fast is the subject of Part IV.

8.12 Building it yourself: an order of work

1. Write `gradient` with only `ADD`, `MUL` and the leaves, and test `sum(x * y + x)` against the hand-derived `1 + y` and `x`. Include a function that uses an input twice.
2. Write `numgrad` (in float64 NumPy) *before* adding more rules. From here on, every new rule gets a gradient-check test the moment it is written.
3. Add the unary ALU rules (`RECIP`, `EXP2`, `LOG2`, `SQRT`, `SIN`), `CAST` and `WHERE`; test `tanh`, `sigmoid`, `relu`.
4. Add `REDUCE(ADD)` and `EXPAND` together (they are duals; one without the other cannot be tested on anything interesting), then `RESHAPE` and `PERMUTE`. Test a matmul and a non-self-inverse permutation.
5. Add `SHRINK` (needs `pad`), `STACK`, constant `INDEX`, then `REDUCE(MAX)` and `MAX` with ties. Test `log_softmax` with slicing, and a dedicated tie test (`max` of `[1, 3, 3]`).
6. Hook it into `Tensor.backward`, write SGD, and watch a loss go down (`test_training_reduces_loss`).

Pitfalls

1. Fan-out must accumulate. The single most common bug in a first AD implementation is `grads[x] = gx` instead of `grads[x] = grads[x] + gx`. Everything that uses a value once still works; `x * x` gets gradient `x` instead of `2x`. Test a function where some input is used twice.

2. The visiting order. Rules must be applied in reverse topological order. A naive recursive "propagate to my sources as soon as I get a gradient" visits a shared node before all its consumers have contributed, and propagates an incomplete gradient (or propagates twice). The `reversed(topo)` loop makes the order a non-issue, and it is iterative, so deep graphs do not hit the recursion limit (NOTES 3).

3. Constants must have the node's shape (NOTES 13). The seed and the constants inside rules (`_full(u, LN2)`, the `0.5` in `MAX`) must be broadcast to the node's shape, or the ALU type rule rejects the graph with `shape mismatch ... vs ()`. `_full` exists for this.

4. Kinks are conventions. `relu` and `maximum(x, 0)` differ at 0 (gradient 0 vs 0.5). Neither is wrong. Tests with integer-valued inputs hit kinks constantly; use random floats for gradient checks, and pin the convention in a dedicated test.

5. Data-dependent indexing has no gradient here. Embedding lookups written with `INDEX` of token ids fail with `NotImplementedError`. Use a one-hot matmul, or add a scatter op (Exercise 9).

6. Python floats in the gradient graph. Learning rates, bias corrections and loss scales that change per step become `CONST`s in the update graph and miss the JIT cache every step (NOTES 17, Chapter 7). The gradient graph is built fresh every step; keep its *structure* identical.

7. Gradients accumulate across `backward` calls. `Tensor.backward` adds to an existing `p.grad`. Forgetting `opt.zero_grad()` doubles the effective learning rate on the second step, and the graph grows by one `ADD` per step, so the structural cache misses every step too.

8. Realizing the loss destroyed its graph (found and fixed). `Tensor.realize` rebinds `uop` to the output buffer. `loss.item(); loss.backward(params)` used to differentiate a `BUFFER` and return zeros for every parameter, without any error. Logging the loss before the backward pass is the most natural thing in the world, so this would have bitten everyone. `realize_all` now keeps `t._graph` and `backward` uses it; `test_item_before_backward_and_detach` pins the behavior.

Exercises

1. ★ **Derive by hand.** Compute `dL/dx` for `L = sum(exp2(x) * x)` by hand, then build it with `gradient` and compare on `x = [0, 1, 2]`.

2. ★ **Predict the graph.** What does `gradient(x.reshape((2, 3)).permute((1, 0)).reduce(Ops.ADD, 2), [x])` return, for a `(6,)` buffer `x`? Predict the ops in the result, then print it with `to_uir`.

3. ★★ **Forward mode.** Implement `jvp(root, target, tangent)`: forward-mode AD that returns the directional derivative of `root` in direction `tangent`. Write the forward rule for `ADD`, `MUL`, `EXP2`, `RESHAPE`, `EXPAND` and `REDUCE`. Check `jvp` against `gradient` via `sum(gradient * tangent) == jvp`.

4. ★★ **Find the bug.** A student's `PERMUTE` rule is `return (g.permute(u.arg),)`. Which tests in `test_gradients` catch it? Write the smallest test that does.

5. ★★ **The softmax gradient.** Derive `d softmax(x)_i / dx_j = s_i (delta_ij - s_j)` by hand. Then compare the size (node count) of the gradient graph `gradient()` builds for `sum(softmax(x) * w)` with a hand-written rule `g_x = s * (g - sum(g * s))`. Is the max-reduce's gradient in there? Why is it mathematically zero?

6. ★★ **Ties in MIN.** `Tensor.min` is `-((-x).max())`. What gradient does `x.min()` give at `x = [2, 1, 1]`? Predict, then check.

7. ★★ **Gradient checking with float32 only.** Change `numgrad` to evaluate the function with `uopc` (float32) instead of NumPy (float64). Find the `eps` that minimizes the error for `tanh_sum`, and explain the U-shaped error curve.

8. ★★★ **Checkpointing.** The forward activations kept alive for the backward pass dominate training memory. Design a `checkpoint(f)` for the Tensor API that recomputes `f`'s forward during the backward pass instead of keeping its outputs. *Hint:* in this lazy, hash-consed world, what does "keeping" even mean? Look at what the kernelizer materializes.

9. ★★★ **Scatter-add.** Design an op `SCATTER_ADD(x_shape, idx, g)` (or a rangeify-level lowering) that makes the gradient of a data-dependent `INDEX` possible, give its type rule and its rangeify rule, and explain why it needs either atomics or a reduction over the index axis.

Solutions to ★ exercises

1. `d/dx (2^x x) = 2^x ln2 x + 2^x = 2^x (x ln 2 + 1)`. At `x = [0, 1, 2]`: `[1, 2(ln 2 + 1), 4(2 ln 2 + 1)] = [1, 3.386, 9.545]`. In the graph, `MUL` sends `g * x` to the `EXP2` node and `g * exp2(x)` to `x`; the `EXP2` rule turns its gradient into `g * x * exp2(x) * ln 2`, and the two contributions to `x` are added.

2. The `REDUCE(ADD, 2)` rule produces `RESHAPE` of the seed to `(1, 1)` and an `EXPAND` to `(3, 2)`; the `PERMUTE` rule permutes that back with the inverse `(1, 0)` to `(2, 3)`; the `RESHAPE` rule reshapes to `(6,)`. So: a constant 1.0, reshape, expand, permute, reshape. All ones, as it must be, since every element of `x` contributes once to the total sum.

Quiz

1. Why does ML use reverse mode rather than forward mode?
2. State the chain rule for a node with two consumers. Where in `gradient()` does it appear?
3. Why is `gradient()` applied to shape graphs and not to rangeified loop programs?
4. What is the gradient rule for `EXPAND`, and why?
5. What is the gradient rule for `SHRINK`?
6. How does the `REDUCE(MAX)` rule handle ties, and what does `max(x, -x)` give at `x = 0`?
7. Why can `gradient` be applied to its own output?

8. Why is the gradient of a data-dependent `INDEX` not supported, and what is the workaround?

Answers

1. The loss is one scalar and the parameters are many. Reverse mode gets the whole gradient (one row of the Jacobian) in one backward pass; forward mode needs one pass per parameter.
2. `dL/dx = dL/du_1 * du_1/dx + dL/du_2 * du_2/dx`: contributions add. In `grads[x] = grads[x] + gx if x in grads else gx`.
3. Shape graphs are pure functions of whole tensors, so every local derivative is again a pure tensor expression in the same ops. Loop programs contain stores, orderings and loops, whose derivatives need a tape or reversed loops.
4. `sum_to(g, x.shape)`: sum the gradient over the broadcast axes, because one input element feeds every copy and the fan-out contributions add up.
5. `pad(g, ((b, n - e), ...))`: put the gradient back at its window inside zeros, since dropped elements have zero gradient.
6. It splits the gradient evenly among all maximal elements (`mask / count`); the elementwise `MAX` rule gives half to each side of a tie. So `max(x, -x)` (that is, `abs`) has gradient `g/2 - g/2 = 0` at 0.
7. Its output is an ordinary shape graph built only from ops that have rules.
8. Its dual is a scatter-add, which the IR cannot express. Use a one-hot matrix multiply instead.

Further reading

- A. G. Baydin, B. A. Pearlmutter, A. A. Radul, J. M. Siskind, "Automatic Differentiation in Machine Learning: a Survey", JMLR 18, 2018. The best single overview of forward/reverse mode.
- A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed., SIAM, 2008. The reference text, including checkpointing.
- The JAX documentation, "The Autodiff Cookbook" (JVPs, VJPs, and higher-order derivatives as function transformations on a pure IR).
- A. Paszke et al., "Automatic differentiation in PyTorch", NIPS 2017 Autodiff Workshop, for the tape-based, run-time alternative.
- W. Moses and V. Churavy, "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients" (Enzyme), NeurIPS 2020: AD on LLVM IR, loops, stores and all.
- tinygrad's `gradient.py` (in recent versions): the same `(node, grad) -> source grads` pattern matcher design, including the movement-op duals.

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

`uopc/autodiff.py · 82 lines`

`tests/test_w05_tensor.py · 111 lines`

9. A Tensor library and your first MNIST

You can now compile any shape graph into C, cut it into kernels, and differentiate it. But nobody wants to *write* shape graphs. Typing `A.reshape((M, 1, K)).expand((M, N, K))` every time you mean `A @ B` is how compiler people punish themselves, not how machine-learning people work. This chapter builds the thin layer that turns the compiler into something that feels like PyTorch: a `Tensor` class with broadcasting, reductions over any axes, matrix multiplication, convolution, softmax and cross-entropy; a handful of layers; two optimizers. Then we use it to train a classifier on MNIST, the course's week-5 milestone, and look closely at the 32 C kernels that make up one training step. None of this needs a new compiler feature. Every operation in this chapter is *a few lines of graph building*, and that is the point: if your IR is right, the frontend is small.

What you'll build

- `uopc/tensor.py`: a lazy `Tensor` that wraps a shaped UOp graph, with NumPy-style broadcasting, movement ops, reductions over arbitrary axes, `matmul`, `conv2d`, `softmax`, `log_softmax`, `argmax`, `one_hot` and `cross_entropy`.
- `uopc/nn.py`: `Linear`, `Conv2d`, `LayerNorm`, `Embedding`, a `parameters()` walker, and the `SGD` and `AdamW` optimizers.
- `examples/mnist_mlp.py`: a 784-128-10 MLP that trains to 94% test accuracy in about nine seconds.

Where this fits. Chapter 7 taught the compiler to split a graph into kernels and Chapter 8 taught it to differentiate one. This chapter is the user-facing API on top of both. Chapter 10 then makes the kernels it produces fast.

9.1 Why a lazy Tensor

A NumPy array is *eager*: `a + b` computes and stores a new array immediately. An eager design is simple, but it hides the program from the compiler. If `y = (x * 2 + 1).relu().sum(1)` ran eagerly, it would be four separate loops with three temporary arrays, and no later pass could fuse them, because by the time `sum` runs the temporaries already exist.

Our `Tensor` is *lazy*. It is a thin wrapper around a UOp, and each operator builds a bigger UOp graph. Nothing runs until somebody asks for numbers (`.numpy()`, `.item()`, or `.realize()`). At that moment the

whole pending graph goes through the pipeline from Chapter 7: kernelize, rangeify, render, compile (or fetch from cache), run.

Laziness buys three things, each of which this chapter relies on:

1. **Fusion.** The compiler sees the whole expression and decides where to cut it into kernels.
2. **Autodiff on graphs.** `gradient()` from Chapter 8 needs the forward graph as data. An eager library would have to record a tape; a lazy one already *is* the tape.
3. **Whole-step JIT caching.** A training step builds the same graph shape every iteration, so its compiled plan can be reused (`_structure_key` in `kernelize.py`).

The price is that errors and costs show up late. A shape mismatch is caught when you build the graph (the UOp type checker runs at construction), but a slow kernel only shows up at `realize()`.

Here is a session, run with `DEBUG=1` so that every `realize` prints one line:

```
>>> x = Tensor(np.arange(6, dtype=np.float32).reshape(2, 3))
>>> y = (x * 2 + 1).relu().sum(1)
>>> y
<Tensor (2,) f32>
>>> y.uop.op, y.uop.src[0].op, y.uop.src[0].arg
Ops.REDUCE Ops.WHERE None
>>> y.numpy()
realize: 1 kernels, schedule 6.1 ms, run 0.1 ms
[ 9. 27.]
>>> y.uop.op, y.uop.arg
Ops.BUFFER ParamArg(slot=2, dtype=dtypes.f32, shape=(2,), addrspace=None)
```

Before `.numpy()`, `y` is a REDUCE over a WHERE (that is `relu`) over arithmetic over a BUFFER. Building it ran no C at all. After `.numpy()` the tensor has been *rebound*: `y.uop` is now a BUFFER that holds the result, so asking again costs nothing, and any graph built on `y` later starts from data rather than recomputing the whole chain.

9.2 The Tensor class

The class starts with a constructor that accepts either a UOp (the internal path, used by every operator) or anything NumPy can turn into an array (the user path):

uopc/tensor.py · Tensor.__init__

```
  def __init__(self, data, dtype: DType | None = None, requires_grad: bool = False):
    if isinstance(data, UOp): self.uop = data
    else:
      arr = np.asarray(data)
      if dtype is None: dtype = dtypes.f32 if arr.dtype.kind == "f" else dtypes.bool if
arr.dtype.kind == "b" else dtypes.i32
      arr = arr.astype(np_dtype(dtype))
      buf = new_buffer(dtype, arr.size, arr)
      self.uop = buf.reshape(arr.shape) if arr.shape != buf.shape else buf
    self.requires_grad = requires_grad
    self.grad: Tensor | None = None
```

User data is copied into a fresh BUFFER UOp (`new_buffer` in `kernelize.py` attaches the NumPy array to the UOp itself, so the memory lives exactly as long as the node does; see Chapter 7). BUFFERs are flat, so a 2-D array becomes `BUFFER(6).reshape((2, 3))`. dtypes are inferred the obvious way: floats become `f32`, bools `bool`, everything else `i32`.

Realization is where the frontend hands over to the compiler:

uopc/tensor.py · Tensor.realize_all

```
  @staticmethod
  def realize_all(*ts: Tensor):
    todo = [t for t in ts if t.uop.op is not Ops.BUFFER and not (t.uop.op is Ops.RESHAPE and
base_buffer(t.uop).op is Ops.BUFFER and prod(t.shape) == base_buffer(t.uop).arg.shape[0])]
    if not todo: return
    bufs = _realize([t.uop for t in todo], Tensor.fuse_reduce, Tensor.opt)
    for t, b in zip(todo, bufs):
      t._graph = t.uop   # keep the graph: loss.item() before loss.backward() must still have
gradients
      t.uop = b.reshape(t.shape) if b.shape != t.shape else b
```

Three details matter. First, tensors that are *already* data (a BUFFER, or a reshape of a whole BUFFER) are skipped; realizing them would launch a useless copy kernel. Second, `realize_all` realizes many tensors in *one* call. That is how an optimizer step updates every parameter, every moment estimate and the loss in a single schedule: one structural cache lookup, one batch of kernels. Realizing them one by one would work, but it would compute shared subexpressions (the gradients) once per tensor. Third, before rebinding a tensor to its result BUFFER, `realize_all` keeps the graph it came from in `t._graph`, so that `loss.item()` followed by `loss.backward(...)` still has a graph to differentiate (the Pitfalls tell how the first version got this wrong).

`assign` is the only "mutation" in the library, and it is not really one:

uopc/tensor.py · Tensor.assign

```
  def assign(self, x: Tensor) -> Tensor:
    """Make this tensor (e.g. a parameter) hold x's value. Functional: rebinds to a new
buffer."""
    assert x.shape == self.shape and x.dtype == self.dtype, f"assign {x.shape} to {self.shape}"
    self.uop = x.uop
    return self
```

It rebinds the Python object to a new graph. Buffers are never written in place by user code; a parameter update builds `p - lr * g` and then makes `p` *point at* that expression. When the expression is realized, `p.uop` becomes the new BUFFER and the old one is garbage-collected along with its UOp. This functional style keeps autodiff sound (no node ever changes its value) and lets the kernelizer reason purely about graphs.

9.3 Broadcasting

Elementwise ops in the IR are strict: `ADD` demands that both operands have the same shape. The type checker from Chapter 1 says so, loudly. Users expect NumPy's *broadcasting* instead: `(2, 3) + (3,)` adds the vector to every row. The rules are:

1. Line up the shapes from the right; pad the shorter one with leading 1s.
2. In each position the sizes must be equal, or one of them must be 1.
3. The result takes the larger size in each position.

uopc/tensor.py · _broadcast_shape

```
def _broadcast_shape(*shapes):
  n = max(len(s) for s in shapes)
  shapes = [(1,) * (n - len(s)) + tuple(s) for s in shapes]
  out = []
  for dims in zip(*shapes):
    d = max(dims)
    if any(x not in (1, d) for x in dims): raise ValueError(f"can't broadcast {shapes}")
    out.append(d)
  return tuple(out)
```

Turning a shape into a broadcast shape takes exactly two primitive movement ops: a RESHAPE to add the leading 1s, and an EXPAND to stretch the 1s:

uopc/tensor.py · Tensor.broadcast_to

```
  def broadcast_to(self, shape) -> Tensor:
    shape = tuple(shape)
    if self.shape == shape: return self
    u = self.uop
    if len(u.shape) < len(shape): u = u.reshape((1,) * (len(shape) - len(u.shape)) + u.shape)
    return Tensor(u.expand(shape) if u.shape != shape else u)
```

and every binary operator funnels through one helper:

uopc/tensor.py · Tensor._binop

```
  def _binop(self, op: Ops, x, reverse=False) -> Tensor:
    # numpy-style promotion for python scalars: int tensor (op) 0.5 is a float computation
    if isinstance(x, float) and not self.dtype.is_float: return self.float()._binop(op, x,
reverse)
    x = self._const(x)
    a, b = (x, self) if reverse else (self, x)
    shape = _broadcast_shape(a.shape, b.shape)
    return Tensor(UOp(op, (a.broadcast_to(shape).uop, b.broadcast_to(shape).uop)))
```

So `a + b` with `a` of shape `(2, 3)` and `b` of shape `(3,)` becomes `ADD(a, EXPAND(RESHAPE(b, (1, 3)), (2, 3)))`. In the REPL:

```
>>> c = a + b
>>> c.uop.op, [s.op for s in c.uop.src]
Ops.ADD [Ops.RESHAPE, Ops.EXPAND]
>>> c.uop.src[1].op, c.uop.src[1].shape, c.uop.src[1].src[0].op, c.uop.src[1].src[0].shape
Ops.EXPAND (2, 3) Ops.RESHAPE (1, 3)
```

(`a` itself shows up as a RESHAPE because it is a flat BUFFER viewed as `(2, 3)`.) Does the EXPAND cost anything? Not a byte. Rangeify (Chapter 6) turns `EXPAND` into "use index 0 on the stretched axis", so the generated C simply never looks at `ridx0` when it reads `b`:

```
void e_2_3(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 2; ridx0++) {
    for (int ridx1 = 0; ridx1 < 3; ridx1++) {
      int alu1 = ((ridx0*3)+ridx1);
      float val2 = data0[alu1];
      float val3 = data1[ridx1];
      data2[alu1] = (val2+val3);
    }
  }
}
```

> **Key idea:** broadcasting is free because it is *indexing*, not copying. A broadcast operand is read through an index expression that ignores the broadcast loop. The same fact is what makes matmul, one-hot and argmax cheap to express below.

The rest of the elementwise vocabulary

With `_binop` in hand, the arithmetic operators are one-liners. Several are *derived* rather than primitive, because the IR deliberately has a small ALU (ADD, MUL, MAX, CMPLT, CMPNE, RECIP, EXP2, LOG2, SQRT, SIN, WHERE, CAST):

`uopc/tensor.py:129-153`

```
  def __add__(self, x): return self._binop(Ops.ADD, x)
  def __radd__(self, x): return self._binop(Ops.ADD, x, True)
  def __mul__(self, x): return self._binop(Ops.MUL, x)
  def __rmul__(self, x): return self._binop(Ops.MUL, x, True)
  def __neg__(self): return self * -1
  def __sub__(self, x): return self + -self._const(x)
  def __rsub__(self, x): return self._const(x) + -self
  def __truediv__(self, x): return self * self._const(x).reciprocal()
  def __rtruediv__(self, x): return self._const(x) * self.reciprocal()
  def __lt__(self, x): return self._binop(Ops.CMPLT, x)
  def __gt__(self, x): return self._binop(Ops.CMPLT, x, True)
  def __ge__(self, x): return (self < x).logical_not()
  def __le__(self, x): return (self > x).logical_not()
  def eq(self, x): return self._binop(Ops.CMPNE, x).logical_not()
  def ne(self, x): return self._binop(Ops.CMPNE, x)
  def logical_not(self): return self._binop(Ops.CMPNE, True)
  def maximum(self, x): return self._binop(Ops.MAX, x)
  def minimum(self, x): return -((-self).maximum(-self._const(x)))
  def where(self, a, b) -> Tensor:
    a = a if isinstance(a, Tensor) else Tensor.full((), a)
    b = b if isinstance(b, Tensor) else Tensor.full((), b, a.dtype)
    shape = _broadcast_shape(self.shape, a.shape, b.shape)
    return Tensor(UOp(Ops.WHERE, (self.broadcast_to(shape).uop, a.broadcast_to(shape).uop,
b.broadcast_to(shape).uop)))
  def cast(self, d: DType) -> Tensor: return Tensor(self.uop.cast(d))
  def float(self) -> Tensor: return self.cast(dtypes.f32)
```

The first line of `_binop` (shown above) is NumPy-style type promotion: an integer tensor combined with a Python `float` is cast to `f32` first, so `Tensor([1, 2, 3]) * 0.5` is `[0.5, 1.0, 1.5]` (see the Pitfalls for what happened without it). Subtraction is addition of a negation, division is multiplication by a reciprocal, `>` is `<` with the operands swapped, `==` is "not `!=`". Why not add SUB and DIV to the IR? Because every op you add must be handled by the renderer, the symbolic simplifier, the bounds analysis *and* autodiff. Deriving `a - b` as `a + b * -1` costs nothing at runtime (gcc sees `x + y*-1.0f` and emits a subtraction) and saves four rules in four files.

Transcendentals follow the same philosophy:

`uopc/tensor.py:155-170`

```
  def reciprocal(self): return Tensor(self.uop.recip())
  def exp2(self): return Tensor(UOp(Ops.EXP2, (self.uop,)))
  def log2(self): return Tensor(UOp(Ops.LOG2, (self.uop,)))
  def exp(self): return (self * LOG2E).exp2()
  def log(self): return self.log2() * LN2
  def sqrt(self): return Tensor(UOp(Ops.SQRT, (self.uop,)))
  def rsqrt(self): return self.sqrt().reciprocal()
  def sin(self): return Tensor(UOp(Ops.SIN, (self.uop,)))
  def cos(self): return (self + math.pi / 2).sin()
  def square(self): return self * self
  def abs(self): return self.maximum(-self)
  def relu(self): return (self > 0).where(self, 0.0)
  def sigmoid(self): return ((-self).exp() + 1).reciprocal()
  def tanh(self): return (self * 2).sigmoid() * 2 - 1
  def gelu(self): return 0.5 * self * (1 + (math.sqrt(2 / math.pi) * (self + 0.044715 * self *
self * self)).tanh())
  def silu(self): return self * self.sigmoid()
```

The only exponential in the IR is `EXP2`, because that is what hardware has (`exp2f` on CPUs, `ex2.approx` on NVIDIA GPUs). `exp(x)` is `exp2(x * log2(e))` and `log(x)` is `log2(x) * ln 2`. You will see the constant `1.4426950408889634f` (that is log2 e) all over the generated C in this chapter. `relu` is a WHERE, `sigmoid` is built from `exp` and a reciprocal, and `tanh` from `sigmoid`. Every one of these gets a gradient for free, because autodiff only needs rules for the primitives.

9.4 Movement ops

The movement methods are thin conveniences over the four primitive movement ops from Chapter 6. `reshape` accepts a `-1` and works out the missing size, `permute` accepts negative axes, `transpose` swaps two axes, `unsqueeze` inserts a 1, `flatten` merges trailing axes. Indexing with ints and step-1 slices is a SHRINK followed by a RESHAPE that drops the integer-indexed axes:

`uopc/tensor.py · Tensor.__getitem__`

```python
def __getitem__(self, idx) -> Tensor:
  """Ints and step-1 slices only."""
  idx = idx if isinstance(idx, tuple) else (idx,)
  arg, squeeze = [], []
  for d, i in enumerate(idx):
    n = self.shape[d]
    if isinstance(i, int):
      i = i % n
      arg.append((i, i + 1)); squeeze.append(d)
    else:
      b, e, step = i.indices(n)
      assert step == 1, "only step-1 slices"
      arg.append((b, max(b, e)))
  arg += [(0, s) for s in self.shape[len(arg):]]
  t = self.shrink(arg)
  return t.reshape(tuple(s for d, s in enumerate(t.shape) if d not in squeeze))
```

`pad` and `flip` call the composite constructions from `composite.py` (built from STACK, RESHAPE and SHRINK; see Chapter 6). Concatenation reuses them: each piece is zero-padded into its place along the axis, and the padded pieces are added.

`uopc/tensor.py · Tensor.cat`

```python
@staticmethod
def cat(ts: list[Tensor], dim: int = 0) -> Tensor:
  """Concatenate via pad + add: each piece is padded into place, then summed."""
  dim = dim % ts[0].ndim
  total = sum(t.shape[dim] for t in ts)
  out, off = None, 0
  for t in ts:
    p = [(0, 0)] * t.ndim
    p[dim] = (off, total - off - t.shape[dim])
    piece = t.pad(p)
    out = piece if out is None else out + piece
    off += t.shape[dim]
  return out
```

Adding zeros sounds wasteful, but after rangeify each padded piece becomes a bounds-checked WHERE, so `cat` costs a few comparisons per element, not extra memory traffic.

9.5 Reductions over arbitrary axes

The IR's REDUCE has a deliberately narrow contract: it reduces the *last n* axes. Users want `x.sum((0, 2))`. The bridge is a permutation: move the kept axes to the front, the reduced axes to the back, then pop:

uopc/tensor.py · Tensor._reduce

```
  def _reduce(self, op: Ops, axis=None, keepdim=False) -> Tensor:
    axes = list(range(self.ndim)) if axis is None else [a % self.ndim for a in (axis if
isinstance(axis, (tuple, list)) else [axis])]
    keep = [i for i in range(self.ndim) if i not in axes]
    t = self.permute(keep + axes)
    r = Tensor(t.uop.reduce(op, len(axes)))
    if keepdim: r = r.reshape(tuple(1 if i in axes else s for i, s in enumerate(self.shape)))
    return r
```

For `x` of shape `(2, 3, 4)` and `axis=(0, 2)`, `keep = [1]`, so we permute by `(1, 0, 2)` to shape `(3, 2, 4)` and REDUCE the last 2 axes, giving shape `(3,)`. With `keepdim=True` a reshape puts the 1s back where the reduced axes were, which is what makes `x - x.max(-1, keepdim=True)` broadcast correctly.

```
>>> x = Tensor(np.arange(24, dtype=np.float32).reshape(2, 3, 4))
>>> x.sum((0, 2)).numpy(), x.sum((0, 2)).uop.src[0].arg
[ 60.  92. 124.] (1, 0, 2)
```

The permutation, like the expand, is free. It only changes which loop variable multiplies which stride:

```
void r_3(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    for (int ridx1 = 0; ridx1 < 2; ridx1++) {
      for (int ridx2 = 0; ridx2 < 4; ridx2++) {
        float val1 = acc0;
        float val2 = data0[(((ridx1*12)+(ridx0*4))+ridx2)];
        acc0 = (val1+val2);
      }
    }
    float val3 = acc0;
    data1[ridx0] = val3;
  }
}
```

The outer loop is the kept axis (stride 4 in memory), the two inner loops are the reduced axes (strides 12 and 1). The accumulator `acc0` lives inside the output loop and outside the reduce loops, exactly the placement rule from Chapter 6.

The derived reductions are short:

uopc/tensor.py · Tensor.mean

```
def mean(self, axis=None, keepdim=False):
  s = (self if self.dtype.is_float else self.float()).sum(axis, keepdim)
  return s * (s.numel() / self.numel())
```

uopc/tensor.py · Tensor.var

```
def var(self, axis=None, keepdim=False):
  return (self - self.mean(axis, keepdim=True)).square().mean(axis, keepdim)
```

`mean` sums in `f32` (casting integer tensors first) and multiplies by `numel(result) / numel(input)`, a Python float that becomes a CONST. `var` is the mean squared deviation. `min` is a negated max of the negation (the IR has only ADD and MAX reductions). `softmax`, `log_softmax` and `logsumexp` get their own section below.

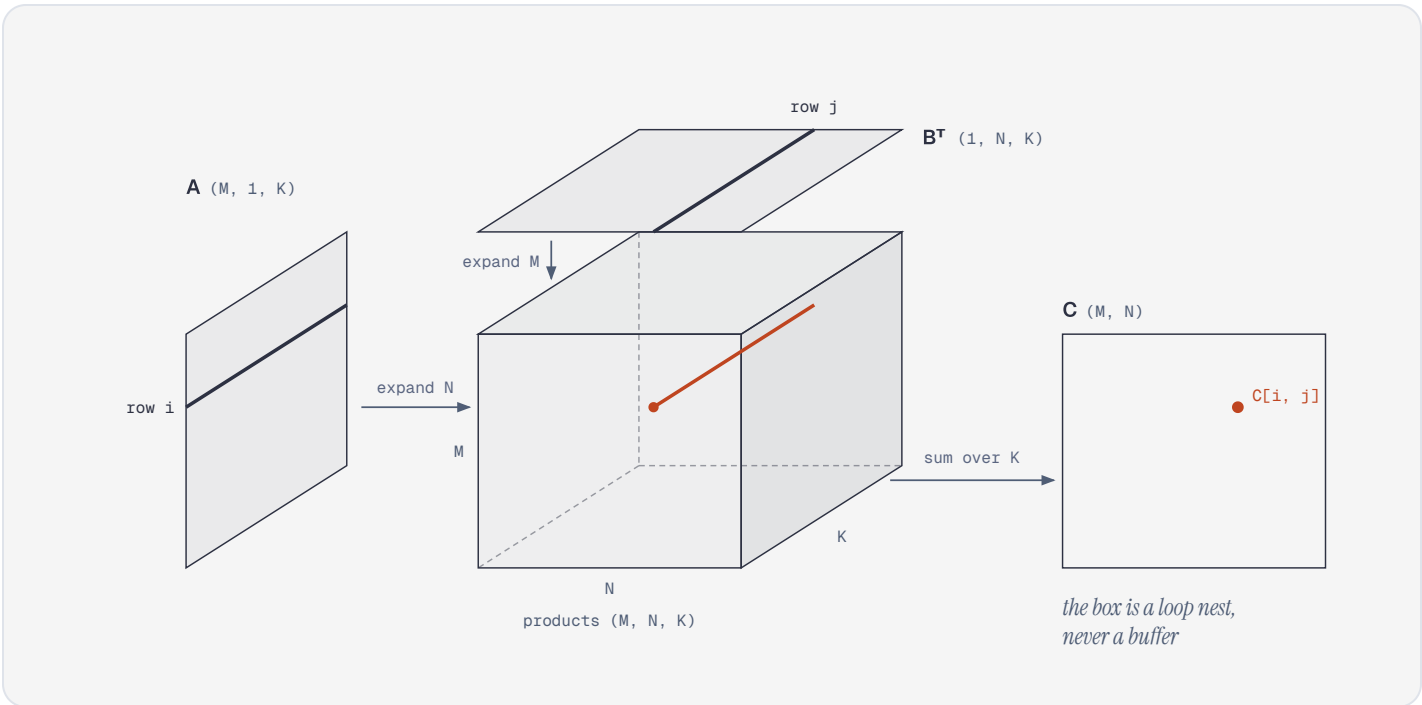
9.6 Matrix multiplication is broadcast, multiply, reduce

Here is the most important three-line function in the library:

uopc/tensor.py · Tensor.matmul

```python
def matmul(self, w: Tensor) -> Tensor:
  """(..., M, K) @ (..., K, N): broadcast to (..., M, N, K), multiply, reduce K."""
  x = self if self.ndim > 1 else self.reshape(1, -1)
  wt = w.reshape(-1, 1) if w.ndim == 1 else w
  wt = wt.transpose()                                    # (..., N, K)
  x = x.reshape(x.shape[:-1] + (1, x.shape[-1]))         # (..., M, 1, K)
  wt = wt.reshape(wt.shape[:-2] + (1,) + wt.shape[-2:])  # (..., 1, N, K)
  out = (x * wt).sum(-1)
  if self.ndim == 1: out = out.reshape(out.shape[:-2] + out.shape[-1:])
  if w.ndim == 1: out = out.reshape(out.shape[:-1])
  return out
```

Read it as geometry. A matrix product `C[i, j] = sum_k A[i, k] * B[k, j]` has three indices, so lay the computation out on an `(M, N, K)` box. `A` has no `j` index, so it is reshaped to `(M, 1, K)` and broadcast along N. `B` has no `i` index; transposed to `(N, K)` and reshaped to `(1, N, K)` it broadcasts along M. Multiplying gives the full box of products `A[i, k] * B[k, j]`, and summing the last axis (K) collapses the box onto the `(M, N)` face, which is `C`.



The box is never stored. It is a *description* of the iteration space, and rangeify turns it into a loop nest in which each multiply happens once, right before it is accumulated. For a `(4, 8) @ (8, 3)` product with the optimizer switched off (`OPT=0`), the graph and the code are:

```
[('REDUCE', (4, 3), (Ops.ADD, 1)), ('MUL', (4, 3, 8), None), ('EXPAND', (4, 3, 8), (4, 3, 8)),
 ('RESHAPE', (4, 1, 8), (4, 1, 8)), ('RESHAPE', (4, 8), (4, 8))]
```

```
void r_4_3(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    for (int ridx1 = 0; ridx1 < 3; ridx1++) {
      float acc0;
      acc0 = 0.0f;
      for (int ridx2 = 0; ridx2 < 8; ridx2++) {
        float val1 = acc0;
        float val2 = data0[((ridx0*8)+ridx2)];
        float val3 = data1[((ridx2*3)+ridx1)];
        acc0 = (val1+(val2*val3));
      }
      float val4 = acc0;
      data2[((ridx0*3)+ridx1)] = val4;
    }
  }
}
```

That is the textbook triple loop, produced by a library function that never mentions a loop. (The first line above follows `src[0]` down from the result; the max error against NumPy was `2.4e-07`.) Look at the two loads in the inner loop. `A` is read at `ridx0*8 + ridx2`: consecutive `k` are adjacent in memory. `B` is read at `ridx2*3 + ridx1`: consecutive `k` are a whole row apart. For a 3-wide matrix that does not matter; for a 1024-wide one it is the difference between 1.9 and 8 GFLOP/s, and it is the first thing Chapter 10 fixes.

Batched matmul falls out of the same code. All leading axes (`...`) ride along untouched, so `(B, M, K) @ (B, K, N)` broadcasts to `(B, M, N, K)` and reduces K. The test `test_ops[matmul_batched]` checks `a @ b.transpose()` on `(2, 3, 4)` tensors against NumPy. `linear` is `x @ w + b`, with the bias broadcast across the batch.

9.7 Convolution as im2col of shrinks

A 2-D convolution with a `KH x KW` kernel computes, for every output pixel, a dot product between the kernel and the `KH x KW` window of input under it, summed over input channels. The classic way to make convolution a matrix multiply is **im2col**: gather every window into a row of a big "patches" matrix, then multiply by the flattened weights. Libraries usually implement im2col as a copy. We get it from movement ops alone:

uopc/tensor.py · Tensor.conv2d

```
  def conv2d(self, w: Tensor, b: Tensor | None = None, padding: int = 0, stride: int = 1) ->
Tensor:
    """x (N, C, H, W), w (O, C, KH, KW). im2col from shrinks, then a matmul."""
    x = self.pad(((0, 0), (0, 0), (padding, padding), (padding, padding))) if padding else self
    N, C, H, W = x.shape
    O, _, KH, KW = w.shape
    OH, OW = (H - KH) // stride + 1, (W - KW) // stride + 1
    cols = []
    for ky in range(KH):
      for kx in range(KW):
        win = x.shrink(((0, N), (0, C), (ky, ky + stride * (OH - 1) + 1), (kx, kx + stride * (OW
- 1) + 1)))
        if stride > 1: win = win._stride2d(stride, OH, OW)
        cols.append(win)                                    # (N, C, OH, OW) each
    patches = cols[0].stack_with(*cols[1:]) if len(cols) > 1 else cols[0].unsqueeze(0)   #
(KH*KW, N, C, OH, OW)
    patches = patches.permute(1, 3, 4, 2, 0).reshape(N, OH, OW, C * KH * KW)
    out = patches @ w.reshape(O, C * KH * KW).T              # (N, OH, OW, O)
    out = out.permute(0, 3, 1, 2)
    return out + b.reshape(1, O, 1, 1) if b is not None else out
```

For each kernel offset `(ky, kx)` there is a SHRINK of the (padded) input that selects the pixels that offset touches for every output position; each has shape `(N, C, OH, OW)`. STACKing the `KH*KW` shrinks gives `(KH*KW, N, C, OH, OW)`. Permuting to `(N, OH, OW, C, KH*KW)` and merging the last two axes gives the patches matrix `(N, OH, OW, C*KH*KW)`, whose column order (channel-major, then `ky`, then `kx`) matches `w.reshape(O, C*KH*KW)`. One matmul later, a permute puts channels back in NCHW order.

Strided convolution needs every `s`-th window. With no strided SHRINK in the IR, `_stride2d` pads the window up to a multiple of `s`, reshapes `(OH*s)` into `(OH, s)`, and takes index 0 of the `s` axis:

uopc/tensor.py · Tensor._stride2d

```
  def _stride2d(self, s: int, OH: int, OW: int) -> Tensor:
    N, C, H, W = self.shape
    t = self.pad(((0, 0), (0, 0), (0, OH * s - H), (0, OW * s - W)))
    return t.reshape(N, C, OH, s, OW, s)[:, :, :, 0, :, 0]
```

Max pooling uses the same reshape trick without the pad: `(H, W)` becomes `(H/k, k, W/k, k)` and the two `k` axes are max-reduced.

Is it right? A `(1, 2, 5, 5)` input, `(3, 2, 3, 3)` weights, `padding=1`, checked against a NumPy loop:

```
(1, 3, 5, 5) max err 1.4305115e-06
```

Is it fast? Here is the reduce kernel, with `OPT=0`. It is worth reading closely, because it is the plain output of a compiler that has not yet learned anything about convolutions:

```
void r_1_5_5_3(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 5; ridx0++) {
    int alu1 = (ridx0*5);
    int alu2 = ((ridx0+5)%6);
    int alu3 = (alu2*5);
    int alu4 = ((ridx0+7)%6);
    int alu5 = (alu4*5);
    for (int ridx1 = 0; ridx1 < 5; ridx1++) {
      int alu6 = (ridx1+5);
      int alu7 = (alu6%6);
      int alu8 = (alu7%5);
      int alu9 = ((ridx1+7)%6);
      int alu10 = (alu9%5);
      int alu11 = (ridx1+17);
      int alu12 = (alu11%6);
      int alu13 = ((ridx1+19)%6);
      int alu14 = (ridx1+29);
      int alu15 = (alu14%6);
      int alu16 = (alu15%5);
      int alu17 = ((ridx1+31)%6);
      int alu18 = (alu17%5);
      for (int ridx2 = 0; ridx2 < 3; ridx2++) {
        float acc0;
        acc0 = 0.0f;
        for (int ridx3 = 0; ridx3 < 18; ridx3++) {
          float val19 = acc0;
          int alu20 = (ridx3%9);
          int alu21 = (ridx3/9);
          int alu22 = (alu21*25);
          int alu23 = (alu22+((alu2%5)*5));
          float val24 = data0[(alu23+alu8)];
          float val25 = data0[(alu23+ridx1)];
          float val26 = data0[(alu23+alu10)];
          int alu27 = (alu22+alu1);
          float val28 = data0[(alu27+(alu12%5))];
          float val29 = data0[(alu27+ridx1)];
          float val30 = data0[(alu27+(alu13%5))];
          int alu31 = (alu22+((alu4%5)*5));
          float val32 = data0[(alu31+alu16)];
          float val33 = data0[(alu31+ridx1)];
          float val34 = data0[(alu31+alu18)];
          float val35 = data1[((((ridx2*18)+(alu21*9))+(((ridx3/3)%3)*3))+(ridx3%3))];
          acc0 = (val19+(((alu20<1)?(((alu6/6)<1)?0.0f:(((alu7/5)<1)?(((((alu1+alu8)+25)/30)<1)?
0.0f:((((alu3+alu8)/25)<1)?val24:0.0f)):0.0f)):((alu20<2)?(((((alu1+ridx1)+25)/30)<1)?0.0f:
((((alu3+ridx1)/25)<1)?val25:0.0f)):((alu20<3)?(((alu9/5)<1)?(((((alu1+alu10)+25)/30)<1)?0.0f:
((((alu3+alu10)/25)<1)?val26:0.0f)):0.0f):((alu20<4)?((((alu11/6)%2)<1)?0.0f:(((alu12/5)<1)?
val28:0.0f)):((alu20<5)?val29:((alu20<6)?(((alu13/5)<1)?val30:0.0f):((alu20<7)?((((alu14/6)%2)
<1)?0.0f:(((alu15/5)<1)?((((alu5+alu16)/25)<1)?val32:0.0f):0.0f)):((alu20<8)?((((alu5+ridx1)/25)
<1)?val33:0.0f):(((alu17/5)<1)?((((alu5+alu18)/25)<1)?val34:0.0f):0.0f))))))))*val35));
        }
        float val36 = acc0;
        data2[(((ridx0*15)+(ridx1*3))+ridx2)] = val36;
```

```
      }
    }
  }
}
```

Three things are visible. The 18-long reduce loop is `C * KH * KW` flattened, and `ridx3 % 9` picks which of the nine kernel taps this iteration belongs to: that is the STACK, lowered to a chain of WHERE (`alu20<1 ? ... : alu20<2 ? ...`). The `%6`, `/30` and `/25` terms are the composite PAD, which Chapter 6 builds from concatenation with zeros; each tap carries its own bounds check. And all nine candidate loads are hoisted *above* the select, so every iteration performs nine loads to use one. The kernel is correct and general, and roughly nine times more memory-hungry than it should be. A convolution-aware compiler would make the tap index a real loop (or unroll it) instead of selecting from a STACK; that is exercise 9.10.

9.8 Softmax, log-softmax and cross-entropy

Softmax turns a vector of scores into a probability distribution: `softmax(x)_i = exp(x_i) / sum_j exp(x_j)`. Computed literally in `f32`, it overflows as soon as a score exceeds about 88 (`exp(89)` is larger than the largest float). The cure is an identity: for any constant `m`, `exp(x_i - m) / sum_j exp(x_j - m)` equals `exp(x_i) / sum_j exp(x_j)`, because the `exp(-m)` factors cancel. Choosing `m = max_j x_j` makes every exponent at most 0, so nothing overflows, and the largest term is exactly 1, so the sum is at least 1 and nothing underflows to a 0/0.

Classification losses want the *log* of the probability of the right class. Taking the log of a softmax is numerically poor (a tiny probability rounds to 0 and its log to `-inf`), so we compute it directly:

`log_softmax(x)_i = (x_i - m) - log(sum_j exp(x_j - m))`.

uopc/tensor.py · Tensor.softmax

```
def softmax(self, axis=-1):
  e = (self - self.max(axis, keepdim=True)).exp()
  return e / e.sum(axis, keepdim=True)
```

uopc/tensor.py · Tensor.log_softmax

```
def log_softmax(self, axis=-1):
  m = self - self.max(axis, keepdim=True)
  return m - m.exp().sum(axis, keepdim=True).log()
```

uopc/tensor.py · Tensor.logsumexp

```
def logsumexp(self, axis=-1):
  m = self.max(axis, keepdim=True)
  return (self - m).exp().sum(axis, keepdim=True).log() + m
```

Cross-entropy for a batch of `B` examples with integer labels `y_b` is `L = -(1/B) * sum_b log_softmax(z_b)[y_b]`. Picking "the entry at index `y_b`" is a gather, and gathers with data-dependent indices are exactly what the IR cannot differentiate (`_rule` for INDEX refuses). So we turn the label into a **one-hot** row and use a multiply-and-sum, which is an ordinary reduce:

uopc/tensor.py · Tensor.one_hot

```
def one_hot(self, n: int) -> Tensor:
  return self.unsqueeze(-1).eq(Tensor.arange(n)).float()
```

uopc/tensor.py · Tensor.cross_entropy

```
def cross_entropy(self, labels: Tensor) -> Tensor:
  """self: (B, C) logits, labels: (B,) i32."""
  return -(self.log_softmax(-1) * labels.one_hot(self.shape[-1])).sum(-1).mean()
```

`one_hot` is itself a broadcast: labels of shape `(B,)` become `(B, 1)`, `arange(n)` has shape `(n,)`, and comparing them broadcasts to `(B, n)` with a 1 exactly where the column equals the label:

```
>>> Tensor(np.array([2, 0, 3], np.int32)).one_hot(4).numpy()
[[0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 1.]]
>>> logits.cross_entropy(labels).item(), numpy_reference
0.31853973865509033 0.3185397696491856
```

How many kernels does `softmax` become? REDUCE is a materialization point (Chapter 7), so: one kernel for the row max, one for the row sum of exponentials (which reads the max), and one elementwise kernel for the output. The test suite pins this down:

tests/test_w05_tensor.py · test_kernelize_fuses_elementwise_and_splits_reductions

```python
def test_kernelize_fuses_elementwise_and_splits_reductions():
  a = Tensor(r(64, 64))
  plan, _ = make_plan([((a * 2 + 1).relu() * 3).uop])
  assert len(plan.kernels) == 1                        # a chain of elementwise ops: one kernel
  plan, _ = make_plan([a.softmax(-1).uop])
  assert len(plan.kernels) == 3                        # max, sum, and the elementwise output
```

Here are the three kernels for a `(4, 8)` softmax, `OPT=0`:

```
void r_4(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float acc0;
    acc0 = -INFINITY;
    for (int ridx1 = 0; ridx1 < 8; ridx1++) {
      float val1 = acc0;
      float val2 = data0[((ridx0*8)+ridx1)];
      acc0 = ((val1>val2)?val1:val2);
    }
    float val3 = acc0;
    data1[ridx0] = val3;
  }
}

void r_4(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    float val1 = data1[ridx0];
    for (int ridx1 = 0; ridx1 < 8; ridx1++) {
      float val2 = acc0;
      float val3 = data0[((ridx0*8)+ridx1)];
      acc0 = (val2+exp2f(((val3+(val1*-1.0f))*1.4426950408889634f)));
    }
    float val4 = acc0;
    data2[ridx0] = val4;
  }
}

void e_4_8(float* restrict data0, float* restrict data1, float* restrict data2, float* restrict
data3) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data1[ridx0];
    float val2 = data2[ridx0];
    for (int ridx1 = 0; ridx1 < 8; ridx1++) {
      int alu3 = ((ridx0*8)+ridx1);
      float val4 = data0[alu3];
      data3[alu3] = (exp2f(((val4+(val1*-1.0f))*1.4426950408889634f))*(1.0f/val2));
    }
  }
}
```

Notice that `exp2f(...)` appears in both the second and the third kernel. The exponentials are not materialized; they are *recomputed* in the consumer, because the kernelizer only cuts at reductions and roots. Recomputing one `exp2f` per element is cheaper than writing and re-reading a temporary, so here fusion wins. When the recomputation is a whole matmul instead of one `exp2f`, the answer flips, and that trade-off is the subject of Chapter 11.

9.9 argmax without an argmax op

To measure accuracy we need the index of the largest logit. The IR has MAX-reduce but no ARGMAX, and adding one would mean a reduction that carries two values (the best value and its index), touching rangeify and the renderer. Instead, a trick with ops we already have:

uopc/tensor.py · Tensor.argmax

```
def argmax(self, axis=-1):
  """Index of the first maximum, as i32."""
  axis = axis % self.ndim
  n = self.shape[axis]
  idx = Tensor.arange(n).reshape((n,) + (1,) * (self.ndim - axis - 1))
  is_max = self.eq(self.max(axis, keepdim=True))
  return is_max.where(n - idx, 0).max(axis) * -1 + n
```

Step by step, for a row of length `n`:

1. `is_max` is a 0/1 mask: where does the row equal its maximum?
2. `is_max.where(n - idx, 0)` puts `n - idx` at every maximal position and 0 elsewhere. Since `idx < n`, the maximal positions all get positive values, and the *smallest* index gets the *largest* value.
3. MAX-reducing picks `n - first_idx`, and `* -1 + n` turns it back into `first_idx`.

So ties go to the first occurrence, matching NumPy:

```
>>> x = Tensor(np.array([[0.1, 3.0, 3.0, -1.0], [2.0, 0.0, 5.0, 5.5]], np.float32))
>>> x.argmax(-1).numpy(), x.argmax(-1).dtype
[1 3] dtypes.i32
```

and it compiles to two reductions and a tiny epilogue (`OPT=0`):

```
void r_2(float* restrict data0, float* restrict data1, int* restrict data2, int* restrict data3)
{
  for (int ridx0 = 0; ridx0 < 2; ridx0++) {
    int acc0;
    acc0 = (-2147483647-1);
    float val1 = data1[ridx0];
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      int val2 = acc0;
      float val3 = data0[((ridx0*4)+ridx1)];
      int val4 = data2[ridx1];
      acc0 = ((val2>(((val3!=val1)!=1)?((val4*-1)+4):0))?val2:(((val3!=val1)!=1)?
((val4*-1)+4):0));
    }
    int val5 = acc0;
    data3[ridx0] = val5;
  }
}
```

The `int` accumulator starting at `INT_MIN` is the `identity()` of an integer MAX-reduce from rangeify, and `(val3!=val1)!=1` is how `eq` renders: "not not-equal". `arange(n)` is a real 4-element BUFFER (`data2`) here; a slightly smarter frontend would make it a RANGE-derived value and save the load.

9.10 Gradients

The Tensor-level autodiff entry point is short because Chapter 8 did the work:

`uopc/tensor.py · Tensor.backward`

```
def backward(self, params: list[Tensor]) -> Tensor:
  """Set p.grad = d(self)/d(p) for each p. Gradients are lazy Tensors like everything else."""
  assert self.shape == (), "backward needs a scalar"
  root = getattr(self, "_graph", self.uop)
  for p, g in zip(params, gradient(root, [p.uop for p in params])):
    p.grad = Tensor(g) if p.grad is None else p.grad + Tensor(g)
  return self
```

`backward` takes the parameter list explicitly, calls `gradient(loss, [p.uop ...])`, and stores each result as a *lazy* Tensor in `p.grad`. Nothing is computed. Gradients are accumulated (`p.grad + g`) if `backward` is called twice, as in PyTorch. Continuing the session from 9.1:

```
>>> l = Linear(3, 4)
realize: 1 kernels, schedule 1.0 ms, run 0.1 ms
>>> [p.shape for p in parameters(l)]
[(3, 4), (4,)]
>>> loss = l(x).square().mean(); loss.backward(parameters(l))
>>> l.weight.grad, l.bias.grad
<Tensor (3, 4) f32> <Tensor (4,) f32>
>>> l.bias.grad.numpy()
realize: 4 kernels, schedule 24.1 ms, run 0.2 ms
[ 0.65893304  1.2755529   0.39464036 -1.0922788 ]
```

(The one kernel during `Linear(3, 4)` is the zero-initialized bias being realized into a buffer.) The gradient tests compare every interesting op against central finite differences in float64, with the same `numgrad` helper you met in Chapter 8:

tests/test_w05_tensor.py · test_gradients

```
@pytest.mark.parametrize("name,fn,npfn", [
  ("tanh_sum", lambda t: t.tanh().sum(), lambda x: np.tanh(x).sum()),
  ("softmax_xent", lambda t: t.log_softmax(-1)[:, 1].sum(), lambda x: (x -
np.log(np.exp(x).sum(-1, keepdims=True)))[:, 1].sum()),
  ("max", lambda t: (t.max(-1) * t.sum(0).max()).sum(), lambda x: (x.max(-1) *
x.sum(0).max()).sum()),
  ("pad_shrink", lambda t: t.pad(((1, 1), (0, 2)))[1:3, 1:].square().sum(),
   lambda x: (np.pad(x, ((1, 1), (0, 2)))[1:3, 1:] ** 2).sum()),
  ("sqrt_recip", lambda t: (t.square() + 1).sqrt().reciprocal().sum(), lambda x: (1 / np.sqrt(x
** 2 + 1)).sum()),
  ("matmul", lambda t: (t @ t.T).sin().sum(), lambda x: np.sin(x @ x.T).sum()),
])
def test_gradients(name, fn, npfn):
  x = r(3, 4)
  t = Tensor(x, requires_grad=True)
  fn(t).backward([t])
  np.testing.assert_allclose(t.grad.numpy(), numgrad(npfn, x), rtol=2e-2, atol=2e-3)
```

9.11 Layers

`nn.py` is written against the Tensor API only; it never touches a UOp. A layer is a Python object that holds parameter Tensors (created with `requires_grad=True`) and has a `__call__`:

uopc/nn.py · Linear

```
class Linear:
  def __init__(self, n_in: int, n_out: int, bias: bool = True):
    bound = 1 / math.sqrt(n_in)
    self.weight = Tensor.uniform(n_in, n_out, low=-bound, high=bound, requires_grad=True)
    self.bias = Tensor.zeros(n_out).realize() if bias else None
    if self.bias is not None: self.bias.requires_grad = True
  def __call__(self, x: Tensor) -> Tensor: return x.linear(self.weight, self.bias)
```

The weight is stored as `(n_in, n_out)` so that `x @ w` needs no transpose in the user's head (the compiler will decide where a transpose is worth making; see Chapter 10). The initialization is PyTorch's default for linear layers: uniform in `+-1/sqrt(n_in)`, which keeps the variance of activations roughly constant from layer to layer. The bias starts at zero and is realized immediately, so it is a BUFFER rather than an EXPAND-of-a-CONST; otherwise "the bias" would be a constant baked into every kernel and its gradient would have nowhere to go.

`Conv2d` is the same pattern around `conv2d`, `LayerNorm` normalizes the last axis (`(x - mean) / sqrt(var + eps) * weight + bias`), and `Embedding` is a one-hot matmul:

uopc/nn.py · Embedding

```
class Embedding:
  """A one-hot matmul: differentiable without a scatter op."""
  def __init__(self, vocab: int, dim: int):
    self.vocab, self.weight = vocab, Tensor.randn(vocab, dim) * 0.02
    self.weight = self.weight.realize(); self.weight.requires_grad = True
  def __call__(self, idx: Tensor) -> Tensor: return idx.one_hot(self.vocab) @ self.weight
```

A real embedding layer is a gather (`weight[idx]`), and its gradient is a scatter-add. Our IR has neither, and autodiff refuses data-dependent INDEX. The one-hot product computes the same thing with ops autodiff understands: `one_hot(idx) @ W` selects rows, and its gradient `one_hot^T @ dY` scatters into them. It costs `vocab` times more FLOPs than a gather, which is fine for a character-level GPT (Chapter 14) and terrible for a 50,000-token vocabulary.

Finally, the optimizer needs a list of parameters. Rather than make every layer register its parameters, `parameters()` walks Python attributes, lists and dicts and collects every Tensor with `requires_grad`:

uopc/nn.py · parameters

```python
def parameters(obj) -> list[Tensor]:
  """Every Tensor with requires_grad reachable through attributes, lists and dicts."""
  out, seen = [], set()
  def walk(o):
    if id(o) in seen: return
    seen.add(id(o))
    if isinstance(o, Tensor):
      if o.requires_grad: out.append(o)
    elif isinstance(o, (list, tuple)): [walk(x) for x in o]
    elif isinstance(o, dict): [walk(x) for x in o.values()]
    elif hasattr(o, "__dict__"): [walk(x) for x in vars(o).values()]
  walk(obj)
  return out
```

The `seen` set handles shared sub-objects (and cycles). The order is the attribute order of each object, which is deterministic, so the optimizer's state lines up with the same parameters every run.

9.12 Optimizers

An optimizer turns gradients into parameter updates. Every update in this library is built as a graph and realized in one batch, together with any extra tensors the caller wants (the loss).

SGD with momentum

Plain stochastic gradient descent is `p <- p - lr * g`. Momentum keeps a velocity `v <- mu * v + g` and steps along it, `p <- p - lr * v`, which smooths noisy gradients and speeds progress along consistent directions:

`uopc/nn.py · SGD`

```python
class SGD(Optimizer):
  def __init__(self, params, lr=0.01, momentum=0.0):
    super().__init__(params)
    self.lr, self.momentum = lr, momentum
    self.v = [Tensor.zeros(*p.shape).realize() for p in params] if momentum else []
  def step(self, *extra: Tensor):
    """Update every parameter. Extra tensors (e.g. the loss) are realized in the same batch."""
    news = []
    for i, p in enumerate(self.params):
      g = p.grad
      if self.momentum:
        self.v[i].assign(self.v[i] * self.momentum + g); g = self.v[i]
      news.append(p.assign(p.detach() - g * self.lr))
    Tensor.realize_all(*news, *self.v, *extra)
```

The test checks the arithmetic by hand for the loss `sum(w^2)` (gradient `2w`):

tests/test_w05_tensor.py · test_sgd_momentum_step

```python
def test_sgd_momentum_step():
  w = Tensor(np.array([1.0, -2.0], np.float32), requires_grad=True)
  opt = SGD([w], lr=0.1, momentum=0.9)
  for _ in range(2):
    loss = w.square().sum()
    opt.zero_grad(); loss.backward(opt.params); opt.step()
  # v1 = 2w0, w1 = w0 - .1*v1 = .8 w0;  v2 = .9 v1 + 2 w1 = 3.4 w0, w2 = .8 w0 - .34 w0 = .46 w0
  np.testing.assert_allclose(w.numpy(), np.array([0.46, -0.92]), rtol=1e-5)
```

AdamW

Adam keeps two running averages per parameter: the first moment `m` (mean of the gradient) and the second moment `v` (mean of its square). With `t` counting steps from 1:

- `m <- b1 * m + (1 - b1) * g`
- `v <- b2 * v + (1 - b2) * g^2`
- `m_hat = m / (1 - b1^t)`, `v_hat = v / (1 - b2^t)` (bias correction: both start at 0, so early averages are too small; dividing by `1 - b^t` undoes that exactly for a constant gradient)
- `p <- p * (1 - lr * wd) - lr * m_hat / (sqrt(v_hat) + eps)`

Dividing by `sqrt(v_hat)` gives every parameter its own step size: parameters with large, noisy gradients take smaller steps. The `(1 - lr * wd)` factor is *decoupled* weight decay, the "W" in AdamW: it shrinks

weights directly instead of adding `wd * p` to the gradient, which in Adam would get divided by `sqrt(v_hat)` and lose its meaning.

uopc/nn.py · AdamW

```
class AdamW(Optimizer):
  def __init__(self, params, lr=1e-3, b1=0.9, b2=0.999, eps=1e-8, wd=0.0):
    super().__init__(params)
    self.lr, self.b1, self.b2, self.eps, self.wd, self.t = lr, b1, b2, eps, wd, 0
    self.m = [Tensor.zeros(*p.shape).realize() for p in params]
    self.v = [Tensor.zeros(*p.shape).realize() for p in params]
  def step(self, *extra: Tensor):
    self.t += 1
    # bias corrections are python floats baked into the graph as constants: each step's
    # graph differs only in two CONSTs, so we pass them as tensors to keep the JIT cache warm
    c1 = Tensor([self.lr / (1 - self.b1 ** self.t)])
    c2 = Tensor([1 / (1 - self.b2 ** self.t)])
    news = []
    for p, m, v in zip(self.params, self.m, self.v):
      g = p.grad
      m.assign(m * self.b1 + g * (1 - self.b1))
      v.assign(v * self.b2 + g.square() * (1 - self.b2))
      upd = m * c1.reshape(*([1] * p.ndim)) / ((v * c2.reshape(*([1] * p.ndim))).sqrt() +
self.eps)
      news.append(p.assign(p.detach() * (1 - self.lr * self.wd) - upd))
    Tensor.realize_all(*news, *self.m, *self.v, *extra)
```

The only surprising lines are `c1` and `c2`. The bias corrections depend on `t`, which changes every step. Written as Python floats they would become CONST nodes in the graph, the graph would be *structurally different* every step, and the structural JIT cache would miss every time. Passed as one-element tensors they are BUFFERs, whose contents do not participate in the cache key (see section 7.8 in Chapter 7). Here is the difference, measured by a scratch subclass `AdamWFloats` whose `step` is identical except that `c1, c2` are floats (time per training step of the MNIST MLP, in ms, first eight steps):

```
AdamW        plans cached  5   step ms: 409 14 16 16 12 11 11 13
AdamWFloats  plans cached 12   step ms: 228 169 187 190 212 179 164 188
```

Same math, same results, more than ten times slower: every step misses the plan cache, so it is re-kernelized, re-rangeified and re-rendered in Python. (In this run gcc did not have to recompile: the `.so` files for these exact constants were already on disk from an earlier run of the same script. On a truly cold cache each step also pays gcc for the update kernels, and an earlier run measured 600-800 ms per step.) This is pitfall 17 in `book/NOTES.md`, and the general lesson is: **anything that changes between iterations must be data, not structure.**

9.13 Training MNIST

MNIST is 70,000 grayscale 28x28 images of handwritten digits (60,000 train, 10,000 test). The loader downloads the four gzip files once into `~/.cache/uopc/data` and returns flat `float32` images scaled to `[0, 1]` and `int32` labels:

examples/datasets.py · mnist

```
def mnist():
  """(X_train (60000, 784) f32 in [0,1], Y_train (60000,) i32, X_test, Y_test)"""
  def images(n): return np.frombuffer(gzip.open(fetch(MNIST_URL + n, n)).read(), np.uint8,
offset=16).reshape(-1, 784).astype(np.float32) / 255
  def labels(n): return np.frombuffer(gzip.open(fetch(MNIST_URL + n, n)).read(), np.uint8,
offset=8).astype(np.int32)
  return (images("train-images-idx3-ubyte.gz"), labels("train-labels-idx1-ubyte.gz"),
          images("t10k-images-idx3-ubyte.gz"), labels("t10k-labels-idx1-ubyte.gz"))
```

The model is a two-layer perceptron, 784 inputs, 128 hidden units with ReLU, 10 outputs, trained with AdamW on random minibatches of 128:

examples/mnist_mlp.py

```python
"""Week 5 milestone: train an MLP on MNIST with our own compiler.

  python examples/mnist_mlp.py            # ~94% test accuracy in 600 steps (~15 ms/step)
  DEBUG=2 python examples/mnist_mlp.py    # print every kernel with its time
"""
import sys, time
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent))
import numpy as np
from datasets import mnist
from uopc.tensor import Tensor
from uopc.nn import Linear, AdamW, parameters


class MLP:
  def __init__(self):
    self.l1, self.l2 = Linear(784, 128), Linear(128, 10)
  def __call__(self, x: Tensor) -> Tensor: return self.l2(self.l1(x).relu())


def main(steps=600, bs=128):
  Xtr, Ytr, Xte, Yte = mnist()
  Tensor.manual_seed(0)
  model = MLP()
  opt = AdamW(parameters(model), lr=1e-3)
  rng = np.random.default_rng(0)
  t0 = time.perf_counter()
  for step in range(steps):
    idx = rng.integers(0, len(Xtr), bs)
    x, y = Tensor(Xtr[idx]), Tensor(Ytr[idx])
    loss = model(x).cross_entropy(y)
    opt.zero_grad()
    loss.backward(opt.params)
    opt.step(loss)
    if step % 100 == 0 or step == steps - 1:
      print(f"step {step:4d}  loss {loss.item():.4f}  {(time.perf_counter() - t0) / (step + 1) * 1e3:.1f} ms/step")
  pred = np.concatenate([model(Tensor(Xte[i:i + 1000])).argmax(-1).numpy() for i in range(0, len(Xte), 1000)])
  acc = (pred == Yte).mean()
  print(f"test accuracy {acc * 100:.2f}%")
  return acc


if __name__ == "__main__": main()
```

A training step is five lines: build the forward graph (`model(x).cross_entropy(y)`), clear the old gradients, build the backward graph, and let the optimizer realize the new parameters, moments and loss in one batch. Evaluation runs the model on the test set in chunks of 1000 and compares `argmax` with the labels. The real

output, from a run with the current code (`book/outputs/mnist.txt` holds an earlier capture with identical losses and slightly slower steps: 460.7 ms for step 0 and 14.4 ms/step at the end):

```
step    0  loss 2.3134  284.8 ms/step
step  100  loss 0.3881  14.2 ms/step
step  200  loss 0.3272  13.1 ms/step
step  300  loss 0.2496  12.4 ms/step
step  400  loss 0.2643  11.9 ms/step
step  500  loss 0.2517  11.6 ms/step
step  599  loss 0.2162  11.6 ms/step
test accuracy 94.06%
```

(The ms/step column is a running average since step 0, so the expensive first step inflates the early rows.) The initial loss 2.31 is about `ln 10 = 2.30`, the loss of a uniform guess over ten classes, which is a good sanity check that initialization is sane. After 600 steps (about 1.3 epochs) the test accuracy is 94.06%. A longer run, a learning-rate schedule or a CNN would do better; the milestone is that a model written in *your* compiler learns.

Step 0 versus the rest. The first step takes about 300 ms because it plans and renders 32 kernels (and, on a cold cache, runs gcc 32 times: with an empty `UOPC_CACHE` the same step took 3.7 s). Every later step has the same graph *structure*: same ops, same shapes, different buffers. `_structure_key` (Chapter 7) maps it to the cached plan, so the steady-state step only builds graphs, hashes them, and calls 32 already-loaded C functions.

9.14 Anatomy of one training step

`DEBUG=2` prints every kernel with its wall time and a rough GFLOP/s. Here is the second training step (the first warm one) of the MLP above, from a run made while this chapter was being written (a rerun with the current code produces the same 32 kernels in the same order; individual times move by tens of percent from run to run):

```
  e_1_128_784                       0.279 ms     0.00 GFLOP/s
  r_128_128                         0.431 ms    59.65 GFLOP/s
  r_128                             0.047 ms     0.19 GFLOP/s
  e_128_128                         0.027 ms     1.84 GFLOP/s
  e_1_10_128                        0.013 ms     0.00 GFLOP/s
  r_128_10                          0.180 ms     1.82 GFLOP/s
  r_128                             0.051 ms     0.05 GFLOP/s
  r_128                             0.030 ms     0.26 GFLOP/s
  r_128                             0.041 ms     0.65 GFLOP/s
  r_128                             0.015 ms     0.43 GFLOP/s
  e_128_10                          0.048 ms     0.75 GFLOP/s
  r_128_128                         0.132 ms     2.48 GFLOP/s
  e_128_128                         0.052 ms     0.95 GFLOP/s
  r_128_784                         0.626 ms    41.01 GFLOP/s
  e_784_128                         0.375 ms     0.80 GFLOP/s
  e_784_128                         0.484 ms     0.83 GFLOP/s
  e_784_128                         0.597 ms     1.51 GFLOP/s
  r_128                             0.052 ms     0.32 GFLOP/s
  e_128                             0.035 ms     0.01 GFLOP/s
  e_128                             0.015 ms     0.03 GFLOP/s
  e_128                             0.025 ms     0.05 GFLOP/s
  r_10_128                          0.187 ms     1.75 GFLOP/s
  e_128_10                          0.038 ms     0.10 GFLOP/s
  e_128_10                          0.017 ms     0.30 GFLOP/s
  e_128_10                          0.029 ms     0.40 GFLOP/s
  r_10                              0.010 ms     0.13 GFLOP/s
  e_10                              0.014 ms     0.00 GFLOP/s
  e_10                              0.013 ms     0.00 GFLOP/s
  e_10                              0.023 ms     0.00 GFLOP/s
  r_128                             0.038 ms     0.41 GFLOP/s
  r_1                               0.010 ms     0.01 GFLOP/s
  e_1                               0.015 ms     0.00 GFLOP/s
realize: 32 kernels, schedule 1.6 ms, run 4.4 ms
```

Kernel names encode the output shape: `r_` kernels contain a reduction, `e_` kernels are purely elementwise. By reading the C (`DEBUG=4`) you can assign every kernel to a role:

Summed, the kernels take 3.95 ms; the `run 4.4 ms` in the `realize` line also includes the Python and ctypes cost of 32 calls. The six kernels that touch `W1`, the 784x128 matrix that holds 99% of the parameters, account for 2.79 ms of it, and the twelve optimizer kernels together cost more than the whole forward pass.

Forward (7 kernels). `e_1_128_784` copies `W1` into a transposed temporary and `r_128_128` is `x @ W1`; the copy is the layout pass from Chapter 10, which makes the reduce axis contiguous for both operands. `e_128_128` is `relu(h + b1)`, materialized by `stage_broadcast_compute` (Chapter 7) because it is broadcast into the next matmul and would otherwise be recomputed for every output column. `e_1_10_128` and `r_128_10` are the copy-and-matmul pair for `W2`. Two `r_128` kernels compute the row max and the sum of exponentials of `log_softmax`.

Backward (10 kernels). Four small kernels for the gradient of cross-entropy with respect to the logits (the last of them, `e_128_10`, writes `dL/dlogits`), `r_128_128` and `e_128_128` for `dL/dh` through `W2` and the ReLU, `r_128_784` for `dW1`, `r_10_128` for `dW2`, and two small reductions over the batch for the bias gradients. The scheduler is free to interleave phases: the third kernel of the step (`r_128` reading the labels) is already a backward kernel, because it depends only on the labels. The ReLU backward kernel is a one-liner:

```
void e_128_128(float* restrict data0, float* restrict data1, float* restrict data2, float*
restrict data3) {
  for (int ridx0 = 0; ridx0 < 128; ridx0++) {
    for (int ridx1 = 0; ridx1 < 128; ridx1++) {
      int alu1 = ((ridx0*128)+ridx1);
      float val2 = data0[alu1];
      float val3 = data1[ridx1];
      float val4 = data2[alu1];
      data3[alu1] = ((0.0f<(val2+val3))?val4:0.0f);
    }
  }
}
```

It recomputes `h + b1 > 0` from the pre-activation instead of storing a mask: that is the WHERE rule from `autodiff.py` (`s[0].where(g, 0)`) fused with the forward expression that produced the condition.

Update (12 kernels). Three per parameter: new `m`, new `v`, new `p`. They are separate kernels because `m`, `v` and `p` are all roots of the `realize_all` call, and roots are materialization points. The `W1` update reads the freshly written `m` and `v`:

```
void e_784_128(float* restrict data0, float* restrict data1, float* restrict data2, float*
restrict data3, float* restrict data4, float* restrict data5) {
  float val1 = data2[0];
  float val2 = data4[0];
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 784; ridx0++) {
    for (int ridx1 = 0; ridx1 < 128; ridx1++) {
      int alu3 = ((ridx0*128)+ridx1);
      float val4 = data0[alu3];
      float val5 = data1[alu3];
      float val6 = data3[alu3];
      data5[alu3] = (val4+(((val5*val1)*(1.0f/(sqrtf((val6*val2))+1e-08f)))*-1.0f));
    }
  }
}
```

`data2[0]` and `data4[0]` are `c1` and `c2`: one-element BUFFERs, loaded once, outside the loop. With `wd = 0` the decay factor `(1 - lr * wd)` folded to 1 and vanished.

Loss (3 kernels). Per-example cross-entropy, the sum over the batch, and the final scale by `-1/128`.

Two observations are worth making before moving on. First, two of the backward kernels (a sum over classes and a tie count) and part of a third compute the gradient that flows through the `max` in `log_softmax`. Mathematically that gradient is exactly zero: shifting all logits by a constant does not change

`log_softmax`, so the derivative with respect to the shift vanishes. Autodiff does not know that, and faithfully computes a zero the expensive way. Second, the `m` and `v` updates for `W1` read the gradient at `data1[((ridx1*784)+ridx0)]`: the gradient is produced as `(128, 784)` (that is `dW1` transposed) and read with a stride of 784. Both are opportunities, not bugs.

9.15 What is slow, and why

The steady-state step costs about 12 ms (the last line of the run in 9.13), and the 32 kernels account for roughly 4 to 6 ms of it. Where does the rest go? A scratch script that times the three phases of 50 warm steps (graph building for the forward pass, `backward`, and `opt.step`) printed:

```
forward graph                  1.87 ms/step
backward graph                 1.71 ms/step
step (structure key+run)       9.81 ms/step
```

and `cProfile` over 50 warm steps (the profiler inflates everything, but the ranking is what matters):

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1600    0.168    0.000    0.198    0.000 uopc/runtime.py:53(__call__)
      150    0.080    0.001    0.157    0.001 uopc/uop.py:131(toposort)
62950/32100    0.076    0.000    0.126    0.000 uopc/uop.py:100(_argkey)
    11750    0.075    0.000    0.125    0.000 uopc/uop.py:230(_infer)
    18650    0.050    0.000    0.326    0.000 uopc/uop.py:111(__new__)
       50    0.037    0.001    0.190    0.004 uopc/kernelize.py:180(_structure_key)
```

`Program.__call__` is the 32 C calls per step (the kernels themselves plus ctypes argument marshalling). Everything else is Python: about 370 UOp constructions per step (18,650 over 50 steps) (each one hash-consed and type-checked by `_infer`), topologically sorting the step's graph, and hashing its structure for the plan cache. This is the tax of a Python compiler that rebuilds the graph every step. NOTES 22 tells how it was once much worse: `UOp.__eq__`/`__hash__` written in Python, and `Enum.__hash__` being a Python function, made the GPT compile take 163 s for 12 steps until identity hashing (in C) cut it to 47 s. Production systems avoid the rebuild entirely by capturing the step once and replaying it (tinygrad's `TinyJit`, CUDA graphs).

What about the kernels themselves? The two big matmuls run at 41-60 GFLOP/s, the elementwise kernels at around 1 GFLOP/s or less. For elementwise work that is expected: they are bound by memory and by

fixed per-call overhead, as Chapter 10 will quantify with the roofline model. Running the whole training with the optimizer switched off (`OPT=0`, `book/outputs/mnist_opt0.txt`) gives the same losses digit for digit and 17.2 ms/step instead of 14.4 (both from the earlier captures): on a model this small the Python overhead hides most of the kernel speedup. On the 1024x1024 GEMM of the next chapter it is the other way around, and the optimizer is worth a factor of more than 50.

9.16 Building it yourself: the week-5 milestone

If you are implementing this for the course, here is an order that keeps every step testable:

1. `Tensor.__init__`, `numpy()`, `realize()` on top of your `realize` from Chapter 7. Test: `Tensor(a).numpy() == a`.
2. `_broadcast_shape`, `broadcast_to`, `_binop`, and the arithmetic dunders. Test against NumPy on random shapes, including `(2, 3, 4) + (4,)`, `(2, 1, 4) * (3, 1)` and scalar operands on both sides.
3. Movement methods and `__getitem__`. Test chains of them against NumPy (`test_ops[getitem_cat]`).
4. `_reduce` with arbitrary axes and `keepdim`; `mean`, `var`, `max`, `min`. Test `sum((0, 2))`.
5. `matmul`, then `softmax` / `log_softmax` (check `softmax(x + 1000)` does not produce NaN), `one_hot`, `cross_entropy`, `argmax` (check ties).
6. `backward`, then gradient-check every op you wrote with finite differences.
7. `Linear`, `parameters`, `SGD`; overfit 64 random points (`test_training_reduces_loss`). Only then `AdamW` and the real dataset.
8. MNIST. Watch the first loss (should be near 2.30) and the ms/step after step 0 (should drop by more than an order of magnitude; if it does not, your JIT cache is missing).

Pitfalls

A changing Python constant defeats the JIT cache (NOTES 17). Section 9.12 measured it: AdamW's bias corrections as floats made every step more than ten times slower. Learning-rate schedules, dropout masks and step counters must all be tensors (see Chapter 7, Pitfalls).

Forgetting to realize optimizer state. `realize_all(*news, *self.m, *self.v, *extra)` is not decoration. A scratch `LeakyAdamW` that realizes only the parameters still trains correctly, because the new `m` and `v` get computed inside the parameter kernels. But `m` and `v` themselves stay lazy, so each step's graph contains the previous step's graph:

```
step 0:     385 ms   graph behind m[0]:  134 nodes
step 1:     263 ms   graph behind m[0]:  235 nodes
step 2:     517 ms   graph behind m[0]:  336 nodes
step 3:     616 ms   graph behind m[0]:  437 nodes
step 4:     771 ms   graph behind m[0]:  538 nodes
step 5:    1102 ms   graph behind m[0]:  639 nodes
```

The graph grows by 101 nodes per step, the structure never repeats, and every step recompiles a bigger program. Any state that outlives a step must be realized every step.

Python scalars took the tensor's dtype. The first version of `_binop` built every Python scalar with `self.dtype`, so `Tensor([1, 2, 3]) * 0.5` multiplied by `int(0.5) == 0`, and `Tensor([1, 2, 3]).mean()` was `0`: NumPy promotes to float, we silently truncated. The fix is the promotion line at the top of `_binop` (an int tensor meeting a Python float is cast to `f32`) and a `mean` that always sums in `f32`. Now:

```
>>> t = Tensor([1, 2, 3])
>>> (t * 0.5).numpy(), t.mean().item()
[0.5 1.  1.5] 2.0
```

tests/test_w05_tensor.py · test_python_float_promotes_int_tensors

```python
def test_python_float_promotes_int_tensors():
  t = Tensor(np.array([1, 2, 3], np.int32))
  np.testing.assert_allclose((t * 0.5).numpy(), [0.5, 1.0, 1.5])
  assert t.mean().item() == 2.0
```

A kernel whose input simplified away overwrote that input. Found while writing this chapter. In `make_plan`, a kernel's argument list (`refs`) was every BUFFER of its *shaped* body, collected before lowering. The renderer, however, emits a C parameter only for buffers the *simplified* program still reads. When simplification deletes a read entirely, the two lists disagree and every pointer shifts by one. `t * 0` on an integer tensor folds to the constant 0, the kernel's signature becomes `void e_3(int* restrict data1)`, and ctypes passed `t`'s own buffer as `data1`: the first version printed `[0 0 0] [0 0 0]` for `(t *`

`0).numpy(), t.numpy()`, writing the "output" into the input. (Floats escape, because `x * 0.0` is not folded: it is not `0.0` for NaN or infinity.) The fix keeps only the slots that survive in the lowered program:

uopc/kernelize.py:159-163

```
    refs = [ph_ref[b] if b.arg.slot < 0 else ("in", in_idx[b]) for b in bufs] + [ref[u]]
    # simplification can delete a read (x*0 -> 0): the C signature then has fewer parameters,
    # so bind exactly the slots that survived, or every later argument shifts by one
    alive = sorted({b.arg.slot for b in ast.toposort() if b.op is Ops.BUFFER})
    refs = [refs[s] for s in alive]
```

The generated kernel still has one parameter, but it is now bound to the right buffer:

```
void e_3(int* restrict data1) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    data1[ridx0] = 0;
  }
}
```

```
>>> (t * 0).numpy(), t.numpy()
[0 0 0] [1 2 3]
```

tests/test_w05_tensor.py · test_deleted_reads_do_not_shift_kernel_arguments

```
def test_deleted_reads_do_not_shift_kernel_arguments():
  t = Tensor(np.array([1, 2, 3], np.int32))
  assert (t * 0).numpy().tolist() == [0, 0, 0]          # simplifies to a constant: t is no
longer read...
  assert t.numpy().tolist() == [1, 2, 3]                # ...and must not be overwritten
```

The general lesson: whenever two passes each compute "the list of parameters" independently, they will eventually disagree. Derive one from the other.

Unstable softmax. If you write `x.exp() / x.exp().sum(-1, keepdim=True)`, it works on small random inputs and returns NaN on real logits above about 88. Always subtract the max; test with `x + 1000`.

Fusion can recompute a lot (NOTES 15). In softmax the recomputed value is one cheap `exp2f`, but in a transformer a LayerNorm output feeding a matmul is recomputed for every output column (see Chapter 7, Pitfalls). If one kernel in `DEBUG=2` output is mysteriously slow, read its C and look for an expensive expression inside the reduce loop.

Reading the loss before `backward` used to zero the gradients. `.item()` realizes the loss, and realizing *rebinds* `loss.uop` to the BUFFER holding the result (9.1). In the first version that was the end of the story: the graph that connected the loss to the parameters was gone, `gradient()` found no path and returned zeros, with no error and no warning:

```
>>> w = Tensor(np.array([1.0, -2.0], np.float32), requires_grad=True)
>>> loss = w.square().sum()
>>> loss.item()
5.0
>>> loss.backward([w]); w.grad.numpy()
[0. 0.]
```

The fix is the `t._graph = t.uop` line in `realize_all` (9.2): a realized tensor remembers the graph it came from, and `backward` differentiates that graph if it exists:

uopc/tensor.py · Tensor.backward

```
def backward(self, params: list[Tensor]) -> Tensor:
  """Set p.grad = d(self)/d(p) for each p. Gradients are lazy Tensors like everything else."""
  assert self.shape == (), "backward needs a scalar"
  root = getattr(self, "_graph", self.uop)
  for p, g in zip(params, gradient(root, [p.uop for p in params])):
    p.grad = Tensor(g) if p.grad is None else p.grad + Tensor(g)
  return self
```

The same session now prints `5.0` and then `[ 2. -4.]`. It is still good practice to read the loss after `opt.step(loss)`, as `mnist_mlp.py` does, so the loss is computed in the same batch of kernels.

`detach()` used to be a no-op for gradients. The first `detach` returned a new Tensor around the *same* UOp, and `gradient()` follows UOps, not Tensor objects, so the gradient flowed straight through. Now `detach` wraps the value in a marker node:

uopc/tensor.py · Tensor.detach

```
def detach(self) -> Tensor:
  """Same value, but gradients stop here. (Kernelize strips the marker, so it costs nothing to
run.)"""
  return Tensor(UOp(Ops.STAGE, (self.uop,), "detach"))
```

Autodiff's rule for STAGE passes the gradient through unchanged, except when the STAGE's `arg` is `"detach"`, where it returns no gradient. The kernelizer removes the marker before anything else, so it costs

nothing at run time:

uopc/kernelize.py · strip_detach

```
strip_detach = PatternMatcher([(UPat(Ops.STAGE, arg="detach", src=(UPat.var("x"),)), lambda x:
x)])
```

Both fixes are pinned by one test: realizing before `backward` keeps the gradient `2w`, and `(w.detach() * w).sum()` has gradient `w`, not `2w`:

tests/test_w05_tensor.py · test_item_before_backward_and_detach

```
def test_item_before_backward_and_detach():
  w = Tensor(np.array([1.0, 2.0], np.float32), requires_grad=True)
  loss = (w * w).sum()
  loss.item()                                          # realizing the loss must keep its graph
  loss.backward([w])
  np.testing.assert_allclose(w.grad.numpy(), [2.0, 4.0])
  w.grad = None
  (w.detach() * w).sum().backward([w])                 # d/dw (c * w) with c = w held constant
  np.testing.assert_allclose(w.grad.numpy(), [1.0, 2.0])
```

Exercises

1. ★ **Count kernels.** Predict how many kernels `(x - x.mean(-1, keepdim=True)).square().sum()` produces for `x` of shape `(64, 64)`, then check with `make_plan`. *Hint:* count REDUCEs.
2. ★ **Predict argmax.** What does `Tensor(np.array([[3., 1., 3.]], np.float32)).argmax(-1).numpy()` print, and why? What would happen if the trick used `idx` instead of `n - idx`?
3. ★ **Std and cross-entropy.** Implement `Tensor.std(axis, keepdim)` and `Tensor.logsumexp`-based `cross_entropy` (use `logsumexp` instead of `log_softmax`). Gradient-check both.
4. ★★ **A differentiable gather.** Implement `Tensor.gather(axis, idx)` for integer index tensors using `one_hot` and a reduce, so that it is differentiable with respect to the data. What does it cost in FLOPs compared to a real gather?
5. ★★ **A learning-rate schedule.** Add a learning-rate schedule (linear warmup, then cosine decay) to `AdamW` without causing JIT cache misses. Verify with `DEBUG=1` that `schedule` stays around a millisecond after step 0.

6. ★★ **Unused buffers.** Revert the `alive` filter in `make_plan` (in a scratch copy), watch `test_deleted_reads_do_not_shift_kernel_arguments` fail, then fix it the *other* way: render every buffer of the body as a C parameter even when it is unused. Which fix is better, and why? *Hint:* think about the structural cache and about identical kernels rendering identical C.
7. ★★ **Dropout.** Implement dropout: `x.dropout(p)` in training mode multiplies by a random 0/(1/(1-p)) mask. Where must the randomness come from so that the graph structure is the same every step?
8. ★★ **A small CNN.** Train a small CNN on MNIST (`Conv2d(1, 8, 3)`, ReLU, `max_pool2d`, `Linear`). Report accuracy and ms/step, and identify the slowest kernel with `DEBUG=2`. Explain its cost from its C source.
9. ★★ **Detach the max.** Use `detach()` to stop the gradient through the max in `log_softmax` (it is mathematically zero, see section 9.14). Count the kernels per training step before and after, and check that the losses are unchanged to the printed precision.
10. ★★★ **Fast convolution.** Make `conv2d` fast: instead of a STACK of shrinks selected with a WHERE chain, express the kernel taps as extra reduce axes (shape `(N, O, OH, OW, C, KH, KW)` reduced over the last three). Compare the generated C and the speed against section 9.7's kernel on a `(64, 16, 28, 28)` input.

Solutions to ★ exercises

1. Two: `make_plan` reports `['r_64', 'r_1']`. The `mean` REDUCE is one materialization point (`r_64`; its `* (1/64)` is elementwise and fuses into its consumer). The subtraction and the square are elementwise and fuse into the final `sum()`, which is both a REDUCE and the root, so it is a single point (`r_1`). Count materialization points, and remember that a root which is itself a REDUCE counts once.

2. `[0]`. The row max is 3, `is_max = [1, 0, 1]`, `where(is_max, 3 - idx, 0) = [3, 0, 1]`, max is 3, and `3 * -1 + 3 = 0`. With `idx` instead of `n - idx` the max would pick the *last* maximal index (2), and the zero for non-maximal positions would collide with index 0.

3. `std = var.sqrt()`. For cross-entropy: `(self.logsumexp(-1).reshape(-1) - (self * labels.one_hot(n)).sum(-1)).mean()`. Both match finite differences to about `5e-8`. (With the first version of `realize_all`, reading the loss with `.item()` before `backward` made this check fail with zero gradients; see the Pitfalls.)

Quiz

1. Why is broadcasting free in this compiler, while in an eager library `np.broadcast_to(b, shape).copy()` costs memory traffic?
2. `REDUCE` only reduces the last `n` axes. How does `x.sum(0)` on a `(4, 5)` tensor become a valid graph?
3. Write `matmul` for `(M, K) @ (K, N)` as a sequence of movement ops, one MUL and one REDUCE, with the shape after each step.
4. Why does `log_softmax` subtract the row max, and why does doing so not change the result?
5. Why are AdamW's bias corrections passed as one-element tensors rather than Python floats?
6. How many kernels does a softmax over the last axis produce, and which values are recomputed rather than stored?
7. Why does `Embedding` use a one-hot matmul rather than indexing?
8. After `y.numpy()`, what is `y.uop`, and why does that matter for graphs built on `y` later?

Answers

1. EXPAND is lowered to an index expression that ignores the broadcast loop (index 0 on the stretched axis); no data is copied, the same element is simply re-read.
2. `_reduce` permutes the reduced axis to the end (`permute(1, 0)`, shape `(5, 4)`) and REDUCEs 1 axis.
3. `A.reshape(M, 1, K).expand(M, N, K)`, `B.permute(1, 0).reshape(1, N, K).expand(M, N, K)`, MUL gives `(M, N, K)`, `REDUCE(ADD, 1)` gives `(M, N)`.
4. To keep every exponent at most 0 (no overflow) with the largest term exactly 1 (no 0/0). The constant cancels between numerator and denominator (or, in log space, between the two terms).
5. They change every step; as CONSTs they would change the graph structure and miss the JIT cache, so every step would recompile.
6. Three: max, sum of exponentials, and the elementwise output. The exponentials are recomputed in the sum kernel and again in the output kernel.
7. Autodiff cannot differentiate a data-dependent INDEX (its gradient is a scatter). One-hot times weights is an ordinary broadcast-multiply-reduce whose gradient is another matmul.
8. A BUFFER (or a reshape of one) holding the result. Later graphs start from that data instead of re-including, and recomputing, the whole chain that produced it.

Further reading

- NumPy documentation, "Broadcasting" (`numpy.org/doc/stable/user/basics.broadcasting.html`): the rules this chapter implements.
- D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015; I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization", ICLR 2019 (AdamW).
- K. Chellapilla, S. Puri, P. Simard, "High Performance Convolutional Neural Networks for Document Processing", 2006: the im2col formulation of convolution.
- Y. LeCun, C. Cortes, C. Burges, "The MNIST database of handwritten digits".
- tinygrad, `tinygrad/tensor.py`: a production-sized version of this file, with the same lazy design.
- A. Karpathy, *micrograd*: the smallest possible autodiff engine, a good contrast to graph-level autodiff.

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

```
uopc/tensor.py · 333 lines

uopc/nn.py · 100 lines

examples/mnist_mlp.py · 43 lines

examples/datasets.py · 29 lines
```

10. Fast CPU kernels

At the end of Chapter 9 the compiler was correct and general. It was not fast. Here is the opening position for this chapter, a 1024x1024x1024 matrix multiply in `f32`, compiled straight from the shape graph, against NumPy (which calls OpenBLAS, decades of hand-tuned assembly):

```
B strided  naive                    5.2 GFLOP/s   max err 2.1e-04
numpy                             449.4 GFLOP/s
```

Eighty-six times slower, for the same 2.15 billion floating-point operations. By the end of the chapter the same compiler, with a few hundred lines of new code, produces a kernel that ran at 238 GFLOP/s in that measurement session and has measured between 114 and 517 GFLOP/s in others, within 5% of NumPy on a quiet machine. Nothing about the math changes. What changes is *where the data is* when the arithmetic needs it: which cache level, which register, which core. This chapter teaches you to reason about that quantitatively (FLOP counts, bytes, arithmetic intensity, the roofline model), then implements the transformations that close the gap: a layout pass, multithreading, register tiling and cache blocking.

What you'll build

- `Opts` in `uopc/rangeify.py`: `upcast` (register tiles), `unroll` (reduce unrolling), `thread` (OpenMP) and `tile` (cache blocks of the output axes), applied *during* rangeify, plus `merge_ends` to let unrolled copies share one loop.
- `uopc/opt.py`: the layout pass (`push_views`, `_last_axis_stride`, `layout_pass`), the hand-written heuristic `hand_opt_choice`, and a tiny autotuner, `search_opt`.
- `examples/bench_gemm.py`: a benchmark that walks the GEMM from 5 to over 200 GFLOP/s step by step.

Where this fits. Chapter 9 built a frontend whose kernels this chapter makes fast. Chapter 11 applies the same reasoning to attention, where the question is not how to tile one kernel but how many kernels to have; Chapter 12 maps the same `Opts` idea onto GPUs.

10.1 The benchmark

`examples/bench_gemm.py` builds the GEMM as a shape graph by hand, lowers it with different `Opts`, renders and runs each version, and checks it against NumPy:

```
examples/bench_gemm.py
```

```python
"""Week 6: GEMM GFLOP/s, step by step, vs numpy (which calls an optimized BLAS).

  python examples/bench_gemm.py 1024

  B strided   C[i,j] = sum_k A[i,k] B[k,j]: the reduce walks down a column of B (stride N)
  B^T         same math, B stored transposed: both operands are contiguous along k
"""
import sys, time
import numpy as np
from uopc.uop import UOp, Ops, ParamArg
from uopc.dtype import dtypes
from uopc.rangeify import lower, Ctx, Opts
from uopc.render_c import render_c
from uopc.runtime import Program
from uopc.tensor import Tensor


def gemm_graph(N, b_transposed=False):
  A = UOp(Ops.BUFFER, (), ParamArg(0, dtypes.f32, (N * N,)))
  B = UOp(Ops.BUFFER, (), ParamArg(1, dtypes.f32, (N * N,)))
  a = A.reshape((N, 1, N)).expand((N, N, N))
  bt = B.reshape((N, N)) if b_transposed else B.reshape((N, N)).permute((1, 0))
  return (a * bt.reshape((1, N, N)).expand((N, N, N))).reduce(Ops.ADD, 1)


def bench(fn, reps=5):
  fn(); best = float("inf")
  for _ in range(reps):
    t0 = time.perf_counter(); fn(); best = min(best, time.perf_counter() - t0)
  return best


if __name__ == "__main__":
  N = int(sys.argv[1]) if len(sys.argv) > 1 else 1024
  a, b = np.random.randn(N, N).astype(np.float32), np.random.randn(N, N).astype(np.float32)
  bt = np.ascontiguousarray(b.T)
  c = np.zeros(N * N, np.float32)
  out = UOp(Ops.BUFFER, (), ParamArg(2, dtypes.f32, (N * N,)))
  flop = 2 * N ** 3
  print(f"N={N}  ({flop / 1e9:.2f} GFLOP per GEMM)")
  rows = [("B strided", False, "naive", Opts()),
          ("B strided", False, "threads", Opts(thread=0)),
          ("B strided", False, "4x4 tile + threads", Opts(upcast={0: 4, 1: 4}, thread=0)),
          ("B^T", True, "naive", Opts()),
          ("B^T", True, "threads", Opts(thread=0)),
          ("B^T", True, "4x4 tile", Opts(upcast={0: 4, 1: 4})),
          ("B^T", True, "4x4 tile + threads", Opts(upcast={0: 4, 1: 4}, thread=0)),
          ("B^T", True, "+ 64x64 cache blocks", Opts(upcast={0: 4, 1: 4}, thread=0, tile={0: 64,
1: 64}))]
  for layout, transposed, name, o in rows:
    prg = Program(render_c(lower(gemm_graph(N, transposed), out, Ctx(opts=o)), "gemm"), "gemm")
    rhs = bt if transposed else b
```

```
    t = bench(lambda: prg(a, rhs, c))
    err = np.abs(c.reshape(N, N) - a @ b).max()
    print(f"  {layout:10s} {name:20s} {flop / t / 1e9:8.1f} GFLOP/s   max err {err:.1e}")
  A, B = Tensor(a), Tensor(b)
  t = bench(lambda: (A @ B).realize())
  print(f"  {'Tensor A@B':10s} {'(layout pass+opt)':20s} {flop / t / 1e9:8.1f} GFLOP/s   (end to
end, includes the transpose copy)")
  t = bench(lambda: a @ b)
  print(f"  {'numpy':31s} {flop / t / 1e9:8.1f} GFLOP/s")
```

Two layouts are compared. "B strided" is the textbook `C = A @ B` with `B` stored row-major as `(K, N)`: the reduction over `k` walks *down a column* of `B`. "B^T" stores `B` transposed as `(N, K)`, which is the same math with both operands contiguous along `k`. The last `B^T` row adds cache blocks to the tiled kernel; 10.11 explains it. The captured run (`book/outputs/bench_gemm_1024.txt`) is the table this chapter explains, row by row:

```
N=1024  (2.15 GFLOP per GEMM)
  B strided  naive                     5.2 GFLOP/s   max err 2.1e-04
  B strided  threads                   7.1 GFLOP/s   max err 2.1e-04
  B strided  4x4 tile + threads       65.1 GFLOP/s   max err 2.1e-04
  B^T        naive                    28.0 GFLOP/s   max err 8.4e-05
  B^T        threads                 149.1 GFLOP/s   max err 8.4e-05
  B^T        4x4 tile                 85.3 GFLOP/s   max err 8.4e-05
  B^T        4x4 tile + threads      237.8 GFLOP/s   max err 8.4e-05
  B^T        + 64x64 cache blocks    288.5 GFLOP/s   max err 8.4e-05
  Tensor A@B (layout pass+opt)       321.4 GFLOP/s   (end to end, includes the transpose copy)
  numpy                              449.4 GFLOP/s
```

Three things should surprise you. Sixteen hardware threads made the strided version only 1.4 times faster. Transposing `B`, which does not change a single FLOP, made the naive kernel five times faster. And register tiling without threads (85.3) is slower than threads without tiling (149.1). By the end of the chapter each of these will be obvious.

A word on measurement. These numbers come from a laptop (Intel i7-13620H, gcc 15, see `book/outputs/machine.txt`), and laptop numbers vary a lot between runs. This table was captured on AC power while other processes kept the CPU busy (load average 6 to 9). Across the sessions in which this chapter was written, the `B^T 4x4 tile + threads` row at N=1024 measured anywhere from 114 GFLOP/s (on battery) to 517 GFLOP/s (on AC, with a quiet machine), and NumPy from 132 to 542. In one run NumPy came out *slower* than our kernel (229 against 475 GFLOP/s), because the two rows ran a few seconds apart and the machine's state changed in between. The reasons:

- **Power source.** On battery the firmware holds the cores at low clocks. The peak-FLOP measurement in 10.3 found one core doing 59 GFLOP/s on battery and 117 on AC.
- **Thermal limits.** A laptop can sustain its maximum turbo clock for seconds, not minutes. A long benchmark slows down as the chip heats up, and the first rows of a table get a cooler chip than the last.
- **Hybrid cores.** The i7-13620H has 6 performance cores (P-cores) and 4 slower efficiency cores (E-cores). Which cores the OS gives a thread, and whether a P-core's second hardware thread is busy with another process, changes from run to run.
- **Background load.** Other processes take cores away from OpenMP's sixteen threads. One thread that shares its core with a browser holds up the whole parallel loop.

So the absolute numbers on a laptop are not stable. Large ratios between rows (5x from the layout, several x from threads and from tiling) survive every run; differences of 20 or 30% do not, and the `+ 64x64 cache blocks` and `Tensor A@B` rows are in that range (10.11 shows the cache-block row changing sign between runs). `bench` in the script takes the best of five runs after a warm-up, which removes some of the noise but not all of it. Compare variants within one session, repeat the session before believing a small difference, and report the machine state with the numbers.

10.2 Counting work: FLOPs, bytes, intensity

Performance engineering starts with two counts.

FLOPs. A GEMM of `(M, K) @ (K, N)` performs `M*N*K` multiplies and `M*N*K` adds: `2*M*N*K` floating point operations. For `N = M = K = 1024` that is 2,147,483,648, the "2.15 GFLOP per GEMM" in the header. A fused multiply-add (FMA, `a*b + c` in one instruction) counts as two. The kernelizer estimates FLOPs for `DEBUG=2` output the same way, one per element of every ALU node plus one per reduced element:

uopc/kernelize.py · _flops

```
def _flops(u: UOp) -> int:
  """Rough FLOP count of a shaped kernel body: one per ALU element computed."""
  from uopc.uop import ALU
  return sum(prod(x.shape or (1,)) for x in u.toposort() if x.op in ALU) + \
         sum(prod(x.src[0].shape) for x in u.toposort() if x.op is Ops.REDUCE)
```

Bytes. The *compulsory* memory traffic is what must cross between DRAM and the chip at least once: for the GEMM, read `A` and `B` and write `C`, `3 * 1024^2 * 4` bytes, about 12.6 MB. An elementwise `out = a * 2` over `n` floats reads `4n` bytes and writes `4n` bytes.

Arithmetic intensity (AI) is the ratio, FLOPs per byte of DRAM traffic:

kernel	FLOPs	compulsory bytes	AI (FLOP/byte)
`out = a * 2`	`n`	`8n`	1/8
row sum `sum(a, 1)`	`n`	`4n`	1/4
GEMM, `N^3`	`2N^3`	`12N^2`	`N/6` = 171 at N=1024
AdamW update for one weight	about `10n`	`16n` (read p, m, v, write p)	about 0.6

Elementwise kernels have constant, tiny intensity: no matter how cleverly you write them, each byte fetched feeds at most a handful of operations. A GEMM's intensity *grows with N*: every element of `A` is used `N` times. That reuse is the opportunity, and the whole chapter is about cashing it in.

10.3 This machine: peak compute and peak bandwidth

To know whether 5.2 GFLOP/s is bad, we need the machine's limits. The i7-13620H has 6 performance cores (two hardware threads each) and 4 efficiency cores, 16 hardware threads in all. A performance core issues AVX2 instructions on 256-bit registers holding 8 `f32` lanes, and can start two FMA instructions per cycle. One FMA on 8 lanes is 16 FLOPs, so one P-core peaks at 32 FLOP/cycle: 157 GFLOP/s at its 4.9 GHz maximum turbo. That number is a ceiling written on a spec sheet; let us measure what we can actually get.

Measuring peak FLOP/s

A microbenchmark must keep both FMA units busy every cycle. An FMA has a latency of about four cycles (its result is available four cycles after it starts), and two can start per cycle, so at least eight *independent* chains of FMAs must be in flight. This scratch program (not part of the repository) runs twelve, on as many threads as asked:

```c
// peak.c: how many f32 FLOP/s can this machine do? 12 independent chains of 8-wide FMAs.
#include <immintrin.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
  int threads = atoi(argv[1]);
  long iters = 100000000;
  double t0 = omp_get_wtime(); float sink = 0;
  #pragma omp parallel num_threads(threads) reduction(+:sink)
  {
    __m256 x = _mm256_set1_ps(0.999f), y = _mm256_set1_ps(0.001f);
    __m256 a0 = _mm256_set1_ps(0.f), a1 = _mm256_set1_ps(1.f), a2 = _mm256_set1_ps(2.f), a3 =
_mm256_set1_ps(3.f),
           a4 = _mm256_set1_ps(4.f), a5 = _mm256_set1_ps(5.f), a6 = _mm256_set1_ps(6.f), a7 =
_mm256_set1_ps(7.f),
           a8 = _mm256_set1_ps(8.f), a9 = _mm256_set1_ps(9.f), a10 = _mm256_set1_ps(10.f), a11 =
_mm256_set1_ps(11.f);
    for (long i = 0; i < iters; i++) {   // 12 independent dependency chains hide the 4-cycle FMA
latency
      a0 = _mm256_fmadd_ps(a0, x, y); a1 = _mm256_fmadd_ps(a1, x, y); a2 = _mm256_fmadd_ps(a2, x,
y);
      a3 = _mm256_fmadd_ps(a3, x, y); a4 = _mm256_fmadd_ps(a4, x, y); a5 = _mm256_fmadd_ps(a5, x,
y);
      a6 = _mm256_fmadd_ps(a6, x, y); a7 = _mm256_fmadd_ps(a7, x, y); a8 = _mm256_fmadd_ps(a8, x,
y);
      a9 = _mm256_fmadd_ps(a9, x, y); a10 = _mm256_fmadd_ps(a10, x, y); a11 =
_mm256_fmadd_ps(a11, x, y);
    }
    __m256 s = a0 + a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9 + a10 + a11;
    sink += s[0];
  }
  double dt = omp_get_wtime() - t0;
  printf("%2d threads: %7.1f GFLOP/s\n", threads, 2.0 * 8 * 12 * iters * threads / dt / 1e9);
  return sink == 12345.f;
}
```

Compiled with `gcc -O2 -march=native -fopenmp` and run on AC power on the same day as the captured benchmark:

```
 1 threads:   117.0 GFLOP/s
 6 threads:   602.6 GFLOP/s
10 threads:   727.7 GFLOP/s
16 threads:   776.8 GFLOP/s
```

Two lessons hide in this tiny program. The first draft initialized every chain to the same two values, and gcc noticed that `a0`, `a2`, `a4`… computed identical things and merged them: the disassembly had two FMAs

per iteration, not twelve, and the "measurement" was off by a factor of six. Always read the assembly of a microbenchmark. Second, 117 GFLOP/s on one core is below the 157 of the spec sheet: 117.0 / 32 FLOP per cycle is about 3.7 GHz, not 4.9. The clock also moves between runs. A second run a minute later printed 74.5 GFLOP/s for one thread and 527.0 for sixteen, and an earlier session on battery measured 59.4 and 293.6 (about 1.9 GHz). A roof is an upper bound, so we use the best values measured on AC power: 777 GFLOP/s for the whole chip, 117 for one core.

Measuring bandwidth

For bandwidth we use our own compiler: a `uopc` kernel that scales a 64 Mi-element array (256 MB, far larger than any cache), and one that sums it, each with and without threads:

```
# bw.py: memory bandwidth as seen by uopc kernels, on 64 Mi floats (256 MB per array)
import time, numpy as np
from uopc.uop import UOp, Ops, ParamArg
from uopc.dtype import dtypes
from uopc.rangeify import lower, Ctx, Opts
from uopc.render_c import render_c
from uopc.runtime import Program
N, R = 1 << 26, 256
A, OUT = (UOp(Ops.BUFFER, (), ParamArg(s, dtypes.f32, (N,))) for s in (0, 1))
ROWS = UOp(Ops.BUFFER, (), ParamArg(1, dtypes.f32, (R,)))
a, out, rows = np.random.randn(N).astype(np.float32), np.zeros(N, np.float32), np.zeros(R,
np.float32)
tests = [("scale  out=a*2", A.reshape((R, N // R)) * 2.0, OUT, out, 8),      # read 4 B + write 4
B per element
         ("rowsum sum(a)", A.reshape((R, N // R)).reduce(Ops.ADD, 1), ROWS, rows, 4)]   # read 4
B per element
for name, g, o_buf, o_arr, bytes_per in tests:
  for tname, o in [("1 thread", Opts()), ("threads", Opts(thread=0))]:
    prg = Program(render_c(lower(g, o_buf, Ctx(opts=o)), "k"), "k")
    prg(a, o_arr); ts = []
    for _ in range(10):
      t0 = time.perf_counter(); prg(a, o_arr); ts.append(time.perf_counter() - t0)
    print(f"{name:15s} {tname:9s} {bytes_per * N / min(ts) / 1e9:6.1f} GB/s")
```

```
scale  out=a*2  1 thread    21.3 GB/s
scale  out=a*2  threads     40.8 GB/s
rowsum sum(a)   1 thread    21.2 GB/s
rowsum sum(a)   threads     55.9 GB/s
```

About 56 GB/s when every core pulls (a run under heavier background load gave 39 GB/s; on battery it was 17.7). One core alone gets about 21 GB/s: a single core cannot keep enough cache misses in flight to saturate

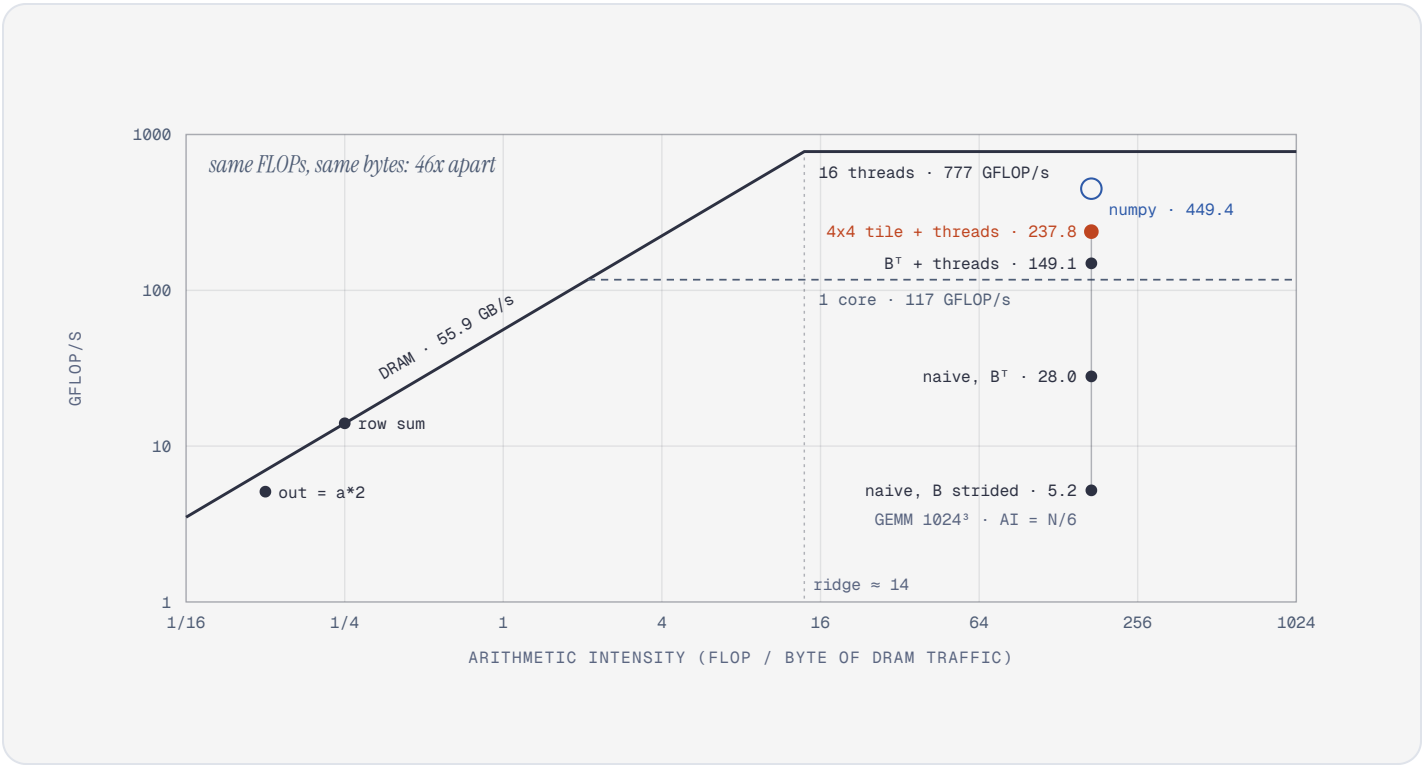
the memory system. The scale kernel's 40.8 GB/s under-counts: writing a cache line the CPU does not have first *reads* it (write-allocate), so the true traffic is 12 bytes per element, not 8, and 40.8 x 12 / 8 = 61 GB/s, close to the sum kernel.

10.4 The roofline model

The roofline model (Williams, Waterman and Patterson, 2009) combines the two limits into one picture. A kernel with arithmetic intensity `AI` cannot run faster than either limit:

```
attainable GFLOP/s = min(peak GFLOP/s, AI * peak GB/s)
```

Plotted on log-log axes this is a slanted line (memory-bound region) meeting a flat line (compute-bound region) at the **ridge point**, `AI = 777 / 55.9 = 14` FLOP per byte on this machine. A kernel to the left of the ridge can only go faster by moving fewer bytes; a kernel to the right can only go faster by doing its arithmetic faster. Here is the machine with the measured kernels:



(The roofs are the best values from 10.3, measured on AC power; the GEMM points are the captured table from 10.1, measured on the same day under background load. On a quieter run the tiled point moves up toward 500.)

The two elementwise kernels sit on the slanted roof: they are as fast as memory allows, and no transformation in this chapter will help them. (Fusing them into their neighbours, so the data is not written and re-read, is the only lever, and that is Chapter 7's business.) The GEMM is far to the right of the ridge: with 171 FLOPs per byte it could, in principle, run at the full 777 GFLOP/s. The naive version runs 150 times below its roof.

> **Key idea:** the DRAM roofline says the naive GEMM is not limited by memory bandwidth. Its problem is lower in the hierarchy: how often it touches the caches, how often registers are reloaded, and how much of the core's parallelism (SIMD lanes, FMA pipelines, cores) it uses. The same roofline idea, applied per level (L1, L2, registers), explains every row of the benchmark.

The roofline also explains the MNIST kernels from Chapter 9. The AdamW update of `W1` touches about 1.6 MB with an intensity near 0.6, so its roof is about 34 GFLOP/s; in the Chapter 9 run it reached 1.5 GFLOP/s. The gap is not the kernel's fault. Its output is a new parameter buffer, freshly allocated every step, so the kernel pays a page fault the first time it touches each 4 KB page. `DEBUG=3` runs each kernel twice to expose this: `e_784_128 0.603 ms ... (again: 0.359 ms)` for the `W1` update in a warm step. (Temporaries used to pay the same price on every step; the plan now keeps its temporary arrays between calls, see Chapter 11. Outputs cannot be reused that way, because they become the new parameters.) Small kernels are dominated by such fixed costs (page faults, waking OpenMP threads, the ctypes call), which is why `hand_opt` only uses threads when there is enough work.

10.5 Caches, lines, pages and the TLB

The memory hierarchy of this machine, as the C library reports it:

```
LEVEL1_DCACHE_SIZE                  49152
LEVEL1_DCACHE_LINESIZE              64
LEVEL2_CACHE_SIZE                   1310720
LEVEL2_CACHE_LINESIZE               64
LEVEL3_CACHE_SIZE                   25165824
LEVEL3_CACHE_LINESIZE               64
```

That is 48 KB of L1 data cache and 1.25 MB of L2 per performance core, and 24 MB of L3 shared by all cores. Three facts govern everything below:

1. **Memory moves in 64-byte lines.** Reading one `float` brings in its 15 neighbours. If you use them next, they are free; if you do not, you paid for 16 floats and used one.
2. **Caches are small.** A 1024-float row is 4 KB; 48 KB of L1 holds twelve of them. A whole 1024x1024 matrix (4 MB) fits in L3 but not in L2.
3. **Addresses are translated in 4 KB pages.** Every load needs its virtual page translated to a physical one, and the translation lookaside buffer (TLB) caches only on the order of a hundred recent translations at its fastest level. Touching a new page on every access misses there constantly.

Now look at the naive strided GEMM's inner loop again. In C, `data1[((ridx2*1024)+ridx1)]`: as `k` (`ridx2`) increases, the address jumps by 4096 bytes. gcc is clever about it and vectorizes *across* `j` instead (eight neighbouring `j` share every cache line), producing this inner loop:

```
.L3:
    vbroadcastss    (%rdx), %ymm0
    vmulps  (%rax), %ymm0, %ymm0
    addq    $4096, %rax
    addq    $4, %rdx
    cmpq    %rax, %rcx
    vaddps  %ymm0, %ymm1, %ymm1
    jne .L3
```

One `A` value is broadcast to all lanes, multiplied by eight consecutive `B` values, accumulated. But `addq $4096, %rax`: every iteration reads from the next row of `B`, which is a new cache line *and* a new 4 KB page. The 1024 lines of one column strip (64 KB) do not fit in L1, so by the time the next strip of `j` comes around they are gone. The loop spends its time waiting for L2 and the TLB, not computing.

Why do sixteen threads make this only 1.4 times faster (5.2 to 7.1 GFLOP/s)? On battery they even made it *slower*, 1.9 to 1.1 GFLOP/s. Each thread runs the same column walk on different rows of `A`, so sixteen hardware threads now each stream a 4 KB-stride pattern, and pairs of them share a core's L1, L2 and TLB. More threads means more simultaneous page walks and more cache lines competing for the same small structures, while the arithmetic was never the bottleneck. This is NOTES 19 in `book/NOTES.md`, and its lesson generalizes: **parallelism multiplies whatever your kernel does, including its bad memory behaviour.** Fix locality first, then add threads.

10.6 Fix 1: layout

If the reduction walks down a column, transpose the matrix so that it walks along a row. With `B^T` stored `(N, K)` the inner loop becomes a dot product of two contiguous rows, `data0[(alu1+ridx2)] * data1[((ridx1*1024)+ridx2)]`, and gcc vectorizes it along `k`:

```
.L3:
    vmovups (%rcx,%rax), %ymm0
    vmulps  (%rdx,%rax), %ymm0, %ymm0
    addq    $32, %rax
    cmpq    $4096, %rax
    vaddps  %ymm0, %ymm1, %ymm1
    jne .L3
```

Eight `k` per instruction, 32 bytes per step, and every byte of every fetched cache line is used. Naive single thread goes from 5.2 to 28.0 GFLOP/s. Why not more? Look at `%ymm1`: there is *one* accumulator, and every `vaddps` must wait for the previous one to finish. The loop is bound by the latency of a single dependency chain, not by the machine's throughput; we will fix that with register tiling.

Users should not have to transpose their matrices. The compiler should do it, and only when it pays. That is the **layout pass** in `opt.py`. It runs on the shape graph before kernelization and asks, for every REDUCE: which operands are read with a non-unit stride along the reduced axis? Those are copied into a contiguous temporary (a STAGE, which Chapter 7 turns into a kernel boundary) *in transposed order*, so the reduce reads them contiguously.

Three helpers do the work. First, movement ops are pushed through elementwise ops down to the leaves, so that every chain of views ends at real data:

`uopc/opt.py · push_views`

```
push_views = PatternMatcher([
  (UPat(MOVEMENT, src=(UPat(ALU, name="a"),), name="m"),
   lambda m, a: a.replace(src=tuple(m.replace(src=(s,)) for s in a.src))),
])
```

`permute(a * b)` is `permute(a) * permute(b)`, and the right-hand form exposes each operand's own view chain. Second, `_views` collects, for the input of a REDUCE, every movement chain that ends at a materialized node (a BUFFER, another REDUCE, or a STAGE):

`uopc/opt.py · _views`

```python
def _views(x: UOp):
  """The movement-op chains feeding x through ALU ops: (top of chain, materialized base)."""
  seen, stack = set(), [x]
  while stack:
    u = stack.pop()
    if u in seen: continue
    seen.add(u)
    if u.op in ALU: stack.extend(u.src)
    elif u.op in MOVEMENT:
      base = u
      while base.op in MOVEMENT: base = base.src[0]
      if base.op in (Ops.BUFFER, Ops.REDUCE, Ops.STAGE): yield u, base
```

Third, `_last_axis_stride` measures how far apart in memory `view[..., k]` and `view[..., k+1]` are. It does not analyze the movement ops symbolically; it *rangeifies* the view against a fake buffer and simplifies the resulting index expression at `k = 0` and `k = 1`:

uopc/opt.py · _last_axis_stride

```python
def _last_axis_stride(view: UOp, base: UOp) -> int | None:
  """Distance in memory between view[..., k] and view[..., k+1], or None if not affine-
looking."""
  if view.shape[-1] == 1: return 0
  fake = UOp(Ops.BUFFER, (), ParamArg(-999, base.dtype, (prod(base.shape),)))
  v = view.substitute({base: fake.reshape(base.shape) if base.shape != fake.shape else fake})
  ctx = Ctx()
  rngs = tuple(ctx.rng(s) for s in v.shape)
  val = rangeify(v, rngs, ctx)
  if val.op is not Ops.LOAD or val.src[0].op is not Ops.INDEX: return None
  e = val.src[0].src[1]
  at = lambda k: simplify(e.substitute({r: UOp.const(k if i == len(rngs) - 1 else 0) for i, r in
enumerate(rngs)}))
  e0, e1 = at(0), at(1)
  return e1.arg - e0.arg if e0.op is Ops.CONST and e1.op is Ops.CONST else None
```

Reusing rangeify as an oracle is the simplest correct stride analysis: whatever chain of reshapes, permutes, expands and shrinks the view has, rangeify already knows how to turn it into an index, and `simplify` turns the index into a constant when the view is affine. With those, the pass is short:

uopc/opt.py · layout_pass

```python
def layout_pass(roots: list[UOp]) -> list[UOp]:
  """Before broadcasting, copy operands whose reduce axis is strided into a contiguous temp (a
STAGE).

  The copy costs O(size of the operand); the reduce it feeds is usually much bigger.
  This is what turns `A @ B` (B read down its columns) into a dot product of rows.
  """
  roots = [graph_rewrite(r, push_views) for r in roots]
  todo: set[UOp] = set()
  for r in UOp(Ops.SINK, tuple(roots)).toposort():
    if r.op is not Ops.REDUCE: continue
    for view, base in _views(r.src[0]):
      s = _last_axis_stride(view, base)
      if s in (0, 1, None): continue
      c = view
      while c.op is Ops.EXPAND: c = c.src[0]                    # stage before the broadcast
      # only stage operands that get broadcast into the reduce: copying them is cheap
      if prod(c.shape) > prod(base.shape) or prod(c.shape) >= prod(r.src[0].shape) or c in todo:
continue
      staged = UOp(Ops.STAGE, (c,))
      if _last_axis_stride(view.substitute({c: staged}), staged) == 1: todo.add(c)
  return UOp.wrap(roots, todo, lambda n: UOp(Ops.STAGE, (n,))) if todo else roots
```

For each strided operand it strips any EXPANDs (we copy the operand *before* it is broadcast, so the copy is `N*K` floats, not `M*N*K`), and keeps the candidate only if staging it really yields stride 1. The guard `prod(c.shape) > prod(base.shape) or prod(c.shape) >= prod(r.src[0].shape)` exists because of a bug. The first version of the pass staged whatever view it found, and in the MNIST backward pass it once materialized a whole `(128, 784, 128)` broadcast product, a 21 ms kernel that was larger than the reduce it was meant to speed up (NOTES 21). Two fixes followed: `push_views`, so that views are found at the leaves rather than around big computed products, and the size guard, so that only operands strictly smaller than the reduce input are ever copied.

The tests check both directions: a strided GEMM gets a STAGE, an already-transposed one does not.

`tests/test_w06_opt.py · test_layout_pass_stages_transposed_operand`

```python
def test_layout_pass_stages_transposed_operand():
  g = gemm_graph(64, 64, 64)
  (g2,) = layout_pass([g])
  assert any(u.op is Ops.STAGE for u in g2.toposort())
  (g3,) = layout_pass([gemm_graph(64, 64, 64, bt=True)])     # already contiguous: nothing to do
  assert not any(u.op is Ops.STAGE for u in g3.toposort())
```

Here is what `Tensor(a) @ Tensor(b)` turns into for 64x64 inputs with the default optimizer (`DEBUG=4`, headers omitted): a copy kernel that writes `B^T`, and the GEMM reading it contiguously.

```
void e_1_64_64(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 64; ridx0++) {
    for (int ridx1 = 0; ridx1 < 64; ridx1++) {
      float val1 = data0[((ridx1*64)+ridx0)];
      data1[((ridx0*64)+ridx1)] = val1;
    }
  }
}
```

At 1024x1024 the copy costs about 0.7 ms against about 5 ms for the GEMM (`DEBUG=2`, steady state, on AC power):

```
  e_1_1024_1024                    0.635 ms     0.00 GFLOP/s
  r_1024_1024                      4.699 ms   456.98 GFLOP/s
realize: 2 kernels, schedule 0.0 ms, run 5.4 ms
```

An `O(N^2)` copy buys an `O(N^3)` kernel a factor of several; the larger the matrix, the better the deal. This is the "Tensor A@B" row of the benchmark: 321.4 GFLOP/s end to end, copy included. In the captured run it came out faster than the hand-built `4x4 tile + threads` row (237.8), although it runs the same kernel plus a copy. That ordering is noise: the rows ran several seconds apart under changing background load.

10.7 Fix 2: threads

Sixteen hardware threads are sitting idle. The output of a GEMM is embarrassingly parallel: each `C[i, j]` is independent. The cheapest way to use the cores is OpenMP: put `#pragma omp parallel for` on the outermost loop and let the runtime split its iterations among threads.

In the IR, a loop's behaviour is recorded in its RANGE's `AxisType` (from the week-6 syllabus: LOOP, UPCAST or THREAD). The optimizer's choices travel to rangeify in an `Opts` record:

`uopc/rangeify.py · Opts`

```python
@dataclass
class Opts:
  """Week 6: how to map a kernel onto the machine. Chosen by uopc.opt, applied during
rangeify."""
  upcast: dict[int, int] = field(default_factory=dict)   # output axis -> unroll factor (register
tile)
  thread: int | None = None                              # output axis run in parallel (OpenMP)
  unroll: int = 1                                        # unroll factor of the innermost reduce
axis
  tile: dict[int, int] = field(default_factory=dict)     # output axis -> cache block size (week
6, loop tiling)
  gpu: bool = False                                      # weeks 7-8: output axes become GLOBAL
(blockIdx) ranges
  local: dict[int, int] = field(default_factory=dict)    # output axis -> threads per block on
that axis (LOCAL)
```

`lower()` gives the THREAD axis the lowest range index, so the linearizer places it outermost, and the renderer emits the pragma for it:

uopc/render_c.py · CRenderer.render_range

```python
  def render_range(self, u: UOp, name: str, end: str) -> list[str]:
    pragma = ["#pragma omp parallel for"] if u.arg.kind is AxisType.THREAD else []
    return pragma + [f"for (int {name} = 0; {name} < {end}; {name}++) {{"]
```

```c
void gemm(float* restrict data0, float* restrict data1, float* restrict data2) {
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 1024; ridx0++) {
    int alu1 = (ridx0*1024);
    for (int ridx1 = 0; ridx1 < 1024; ridx1++) {
      float acc0;
      acc0 = 0.0f;
      for (int ridx2 = 0; ridx2 < 1024; ridx2++) {
        float val2 = acc0;
        float val3 = data0[(alu1+ridx2)];
        float val4 = data1[((ridx1*1024)+ridx2)];
        acc0 = (val2+(val3*val4));
      }
      float val5 = acc0;
      data2[(alu1+ridx1)] = val5;
    }
  }
}
```

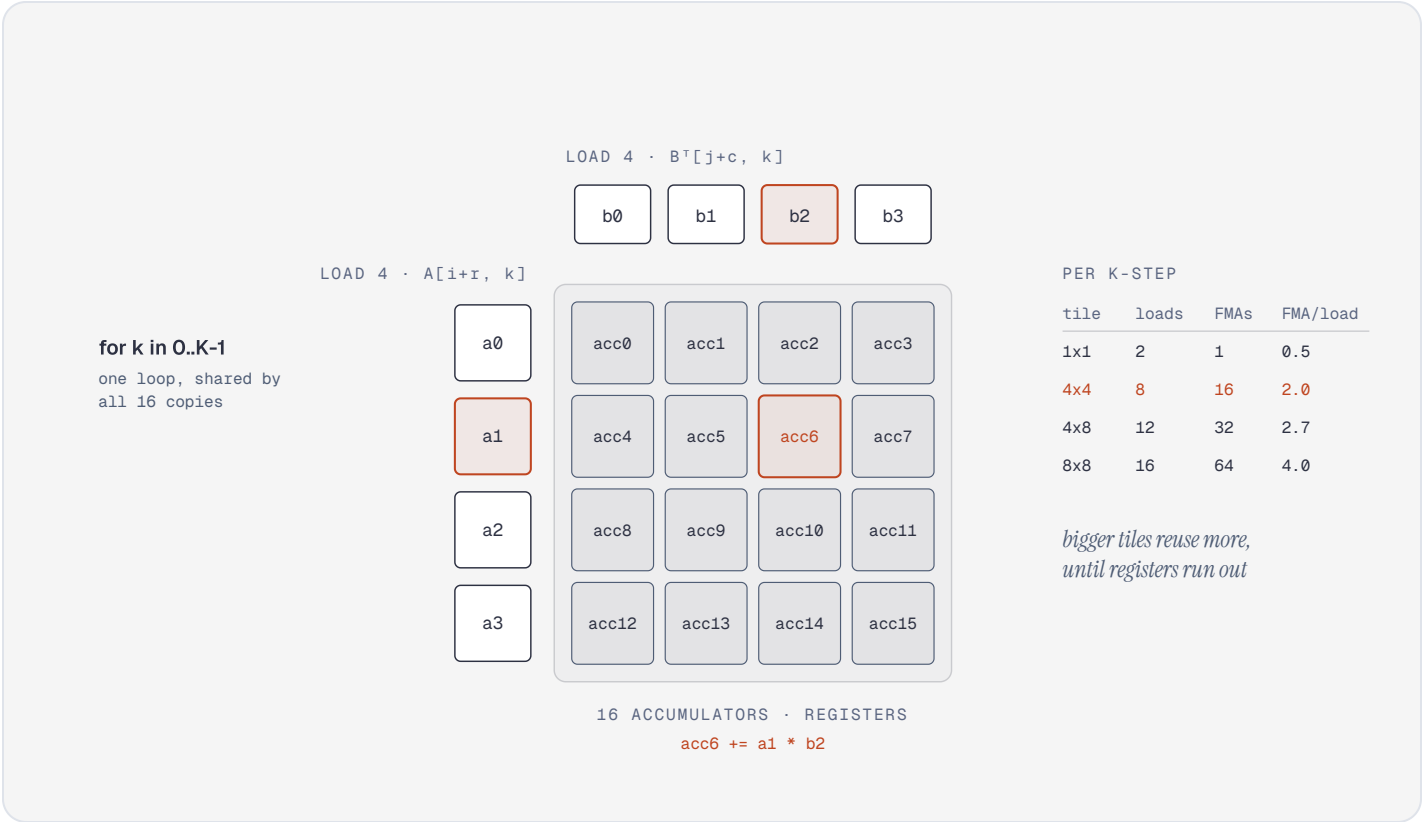
Is this correct? Every iteration of the parallel loop writes a different row of `C`, so there are no races on the output. The accumulator `acc0` is the subtle part: it must be *private* to each thread. It is, because the linearizer declares an ALLOC in the deepest scope common to its users (NOTES 9), which is inside the `ridx1` loop, inside the parallel region, so each thread gets its own. Had it been declared at the top of the function, all threads would share one `acc0` and the result would be garbage. On the contiguous layout, threads take the naive kernel from 28.0 to 149.1 GFLOP/s: about 5.3x on a machine with 6 fast and 4 slow cores. gcc's OpenMP runtime splits the iterations statically into equal chunks, so on a hybrid CPU the performance cores probably finish early and wait for the efficiency cores (exercise 7 measures this).

Threads have a fixed cost: waking the thread pool and joining it at the end costs microseconds, which is a lot for a 128-element kernel. `hand_opt_choice` (below) only threads kernels with at least `2^16` units of work (output elements times reduce size).

10.8 Fix 3: register tiling

The naive kernel computes one output element at a time: for each `k`, load `A[i, k]`, load `B[j, k]`, one FMA. Two loads per FMA. Every `A[i, k]` gets loaded again for the next `j`, and every `B[j, k]` again for the next `i`. The roofline at the register level says: the core can do two FMAs per cycle but only two or three loads per cycle, so a kernel with two loads per FMA can never exceed about one FMA per cycle, half of peak even with perfect caches.

Register tiling computes a small block of outputs at once. With a 4x4 tile, each iteration of `k` loads four values of `A` (rows `i..i+3`) and four of `B` (rows `j..j+3`) and updates all sixteen products, keeping the sixteen running sums in registers for the whole `k` loop:



Eight loads now feed sixteen FMAs: 2 FMAs per load instead of 0.5, a fourfold drop in load traffic per FLOP. Larger tiles reuse more (4x8 gives 2.7, 8x8 gives 4), until the accumulators no longer fit in the 16 vector registers of AVX2. And there is a second, quieter benefit: the sixteen accumulators are sixteen *independent* dependency chains, so the FMA pipelines can overlap them, which removes the single-chain latency bound we saw in 10.6.

How rangeify builds a tile

In the syllabus vocabulary this is UPCAST: an axis that is "fully unrolled, becomes straight-line code". `uopc` implements it without a new kind of loop. An output axis of size `s` with upcast factor `f` becomes a RANGE of size `s/f` plus a constant offset `c in 0..f-1`, and rangeify is simply called once per combination of constants:

uopc/rangeify.py · _axis_parts

```
def _axis_parts(ctx: Ctx, size: int, factor: int, kind: AxisType) -> tuple[UOp, int]:
  """An axis of `size` as (outer RANGE or CONST 0, unroll factor): index = outer * factor + c."""
  if factor > 1 and size % factor == 0:
    return (ctx.rng(size // factor, kind) if size > factor else UOp.const(0)), factor
  return (ctx.rng(size, kind) if size > 1 else UOp.const(0)), 1
```

uopc/rangeify.py · _combos

```
def _combos(parts: list[tuple[UOp, int]]) -> list[tuple[UOp, ...]]:
  """Every index tuple obtained by fixing the unrolled part of each axis to a constant.
  parts[k] = (base, f): the index of axis k is base * f + c for c in 0..f-1."""
  out = [()]
  for b, f in parts:
    known0 = b.op is Ops.CONST and b.arg == 0
    out = [c + (UOp.const(k) if known0 else (b * f + k if f > 1 else b),) for c in out for k in
range(f)]
  return out
```

`lower` computes the index tuples for the whole tile and rangeifies the output once per tuple, producing `f0 * f1` separate stores:

uopc/rangeify.py · lower

```
def lower(root: UOp, out: UOp, ctx: Ctx | None = None) -> UOp:
  """Lower one shaped value into a SINK that stores it, element by element, into BUFFER `out`."""
  ctx = ctx or Ctx()
  o = ctx.opts
  if o.gpu:
    # each output axis gets a GLOBAL range and (optionally) a LOCAL one; neither becomes a loop
on a GPU
    gl = [_gpu_axis(ctx, s, o.local.get(k, 1), o.upcast.get(k, 1)) for k, s in
enumerate(root.shape)]
    plist = gl
  else:
    # the THREAD axis gets the lowest range index, so it is the outermost loop
    order = ([o.thread] if o.thread is not None else []) + [k for k in range(len(root.shape)) if
k != o.thread]
    tiled = {k: t for k, t in o.tile.items() if 1 < t < root.shape[k] and root.shape[k] % t == 0
and t % o.upcast.get(k, 1) == 0}
    parts: dict[int, tuple[UOp, int]] = {}
    # tiled axes: first every block loop (outer), then the loops inside a block. A block of
outputs reuses
    # the same rows of each operand, so they stay in cache while the block is computed
    outer = {k: ctx.rng(root.shape[k] // t, AxisType.THREAD if k == o.thread else AxisType.LOOP)
for k, t in tiled.items()}
    for k in order:
      f = o.upcast.get(k, 1)
      if k in tiled:
        inner = tiled[k] // f
        parts[k] = (outer[k] * inner + ctx.rng(inner) if inner > 1 else outer[k], f)
      else:
        parts[k] = _axis_parts(ctx, root.shape[k], f, AxisType.THREAD if k == o.thread else
AxisType.LOOP)
    plist = [parts[k] for k in range(len(root.shape))]
  stores = []
  for idx in _combos(plist):
    val = rangeify(root, idx, ctx)
    stores.append(out.index(ravel(idx, root.shape) if root.shape else UOp.const(0)).store(val))
  loops = ranges_in(*(b for b, _ in plist))
  body = stores[0] if len(stores) == 1 else UOp(Ops.SINK, tuple(stores))
  return simplify(merge_ends(sink(body.end(*loops))))
```

That alone would produce sixteen *separate* reduce loops, one per tile element, each with its own accumulator: correct, and no faster than before, because nothing is shared. The sharing comes from `_reduce`. It memoizes the reduce RANGEs by the *set of loop ranges* in the output index, ignoring the constants:

`uopc/rangeify.py · _reduce`

```
def _reduce(u: UOp, idx: tuple[UOp, ...], ctx: Ctx) -> UOp:
  rop, n = u.arg
  x = u.src[0]
  rshape = x.shape[len(x.shape) - n:]
  # unrolled copies of the same reduce (they differ only by constants in idx) share one loop
  rkey = (u, frozenset(ranges_in(*idx)))
  if (parts := ctx.reduce_ranges.get(rkey)) is None:
    parts = [_axis_parts(ctx, s, ctx.opts.unroll if k == n - 1 else 1, AxisType.REDUCE) for k, s
in enumerate(rshape)]
    ctx.reduce_ranges[rkey] = parts
  real = [r for r, _ in parts if r.op is Ops.RANGE]
  vals = [rangeify(x, idx + c, ctx) for c in _combos(parts)]
  while len(vals) > 1:                                    # unrolled reduce axis: a balanced tree
    vals = [UOp(rop, (vals[i], vals[i + 1])) if i + 1 < len(vals) else vals[i] for i in range(0,
len(vals), 2)]
  if not real: return vals[0]
  acc = ctx.alloc(u.dtype)
  # the accumulator lives inside the loops the output index uses...
  acc0 = acc.after(*ranges_in(*idx))
  init = acc0.index(0).store(UOp.const(identity(rop, u.dtype), u.dtype))
  # ...is re-read every iteration of the reduce loops...
  accl = acc0.after(init, *real)
  upd = accl.index(0).store(UOp(rop, (accl.index(0).load(), vals[0])))
  # ...and is read once more after the loops end
  return acc0.after(upd.end(*real)).index(0).load()
```

All sixteen tile elements have indices like `(4*ridx0 + c0, 4*ridx1 + c1)`, which mention the same two RANGEs, so `rkey` is the same for all of them and they reuse one set of reduce RANGEs. Each still gets its own accumulator ALLOC and its own update store, but all of them hang off the same loop. (The `unroll` factor for the innermost reduce axis is applied here too; see 10.10.)

There is one more step. Each of the sixteen update stores is closed by its own END over the shared reduce range; the linearizer would see sixteen ENDs of one loop. `merge_ends` groups ENDs that close the same ranges and replaces them with one END over a SINK of their bodies:

`uopc/rangeify.py · merge_ends`

```python
def merge_ends(root: UOp) -> UOp:
  """Unrolled copies each END the same shared loop. Merge them into one END over a SINK."""
  while True:
    groups: dict[tuple, list[UOp]] = {}
    for u in root.toposort():
      if u.op is Ops.END: groups.setdefault(u.src[1:], []).append(u)
    dup = next((g for g in groups.values() if len(g) > 1), None)
    if dup is None: return root
    merged = UOp(Ops.END, (UOp(Ops.SINK, tuple(e.src[0] for e in dup)), *dup[0].src[1:]))
    root = root.substitute({e: merged for e in dup})
```

The result, for a small 8x8 GEMM with a 2x2 tile, is one shared `k` loop updating four accumulators:

```
void gemm(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    int alu1 = (ridx0*16);
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      int alu2 = (ridx1*2);
      int alu3 = (alu1+alu2);
      float acc0;
      acc0 = 0.0f;
      float acc1;
      acc1 = 0.0f;
      float acc2;
      acc2 = 0.0f;
      float acc3;
      acc3 = 0.0f;
      int alu4 = (alu2+alu1);
      for (int ridx2 = 0; ridx2 < 8; ridx2++) {
        float val5 = acc0;
        int alu6 = (alu1+ridx2);
        float val7 = data0[alu6];
        int alu8 = ((ridx1*16)+ridx2);
        float val9 = data1[alu8];
        acc0 = (val5+(val7*val9));
        float val10 = acc1;
        float val11 = data1[(alu8+8)];
        acc1 = (val10+(val7*val11));
        float val12 = acc2;
        float val13 = data0[(alu6+8)];
        acc2 = (val12+(val13*val9));
        float val14 = acc3;
        acc3 = (val14+(val13*val11));
      }
      float val15 = acc0;
      data2[alu3] = val15;
      float val16 = acc1;
      data2[(alu4+1)] = val16;
      float val17 = acc2;
      data2[(alu3+8)] = val17;
      float val18 = acc3;
      data2[(alu4+9)] = val18;
    }
  }
}
```

Two `A` loads (`val7`, `val13`) and two `B` loads (`val9`, `val11`) feed four FMAs. Each value is loaded once and used twice, exactly the outer product of the diagram. After `merge_ends`, the program has exactly two ENDs: one over a SINK of the four update stores closing the reduce range, one over a SINK of the four output stores closing the two output ranges:

```
>>> [(len(e.src), e.src[0].op.name, [r.arg.idx for r in e.src[1:]]) for e in ends]
[(2, 'SINK', [2]), (3, 'SINK', [0, 1])]
```

The test pins the structure down: two output loops plus one shared reduce loop, at least four accumulators:

tests/test_w06_opt.py · test_upcast_shares_reduce_loop

```
def test_upcast_shares_reduce_loop():
  src = render_c(lower(gemm_graph(8, 8, 8), buf(2, 64), Ctx(opts=Opts(upcast={0: 2, 1: 2}))),
"k")
  assert src.count("for (") == 3                        # 2 output loops + ONE shared reduce loop
  assert src.count("acc") >= 4
```

Notice that `alu3 = (alu1+alu2)` and `alu4 = (alu2+alu1)` are the same value computed twice: the simplifier does not put commutative operands in a canonical order, so CSE by hash-consing misses it. gcc removes the duplicate, so it costs nothing at runtime, but it is a small blemish (exercise 9).

What the hardware sees

For the 4x4 tile at N=1024, gcc vectorizes the shared `k` loop: each of the sixteen scalar accumulators becomes an 8-lane vector of partial sums over `k`, and the inner loop is sixteen `vfmadd` instructions fed by eight vector loads:

```
.L6:
    movq    24(%rsp), %rsi
    vmovups (%rcx,%rax), %ymm1
    vmovups (%r14,%rax), %ymm6
    vmovups (%r15,%rax), %ymm4
    vmovups (%rsi,%rax), %ymm2
    vmovups (%rdx,%rax), %ymm0
    vfmadd231ps %ymm6, %ymm1, %ymm10
    vfmadd231ps %ymm4, %ymm1, %ymm11
    vfmadd231ps %ymm2, %ymm1, %ymm15
    vfmadd213ps 160(%rsp), %ymm0, %ymm1
    vmovaps %ymm1, 160(%rsp)
    vmovups (%r9,%rax), %ymm1
    vfmadd231ps %ymm6, %ymm1, %ymm12
    vfmadd231ps %ymm4, %ymm1, %ymm3
    vfmadd231ps %ymm2, %ymm1, %ymm5
    vfmadd213ps 128(%rsp), %ymm0, %ymm1
    vmovaps %ymm1, 128(%rsp)
    vmovups (%r10,%rax), %ymm1
    vfmadd231ps %ymm6, %ymm1, %ymm7
    vfmadd231ps %ymm4, %ymm1, %ymm8
    vfmadd231ps %ymm2, %ymm1, %ymm9
    vfmadd231ps %ymm0, %ymm1, %ymm14
    vmovups (%r11,%rax), %ymm1
    addq    $32, %rax
    vfmadd213ps 96(%rsp), %ymm1, %ymm6
    vfmadd213ps 64(%rsp), %ymm1, %ymm4
    cmpq    $4096, %rax
    vfmadd213ps 32(%rsp), %ymm1, %ymm2
    vfmadd231ps %ymm0, %ymm1, %ymm13
    vmovaps %ymm6, 96(%rsp)
    vmovaps %ymm4, 64(%rsp)
    vmovaps %ymm2, 32(%rsp)
    jne .L6
```

Count the registers: sixteen accumulators plus eight loaded vectors is 24, and AVX2 has 16 `ymm` registers. gcc keeps eleven accumulators in registers and *spills* five to the stack: the `vfmadd213ps 160(%rsp)` / `vmovaps %ymm1, 160(%rsp)` pairs read an accumulator from memory and write it back every iteration. (The first version of `hand_opt_choice` carried a comment claiming that sixteen vector accumulators fit in AVX2's sixteen registers; this assembly showed they do not once the operands need registers too, and the comment now says so.) Spills to L1 are cheap, and with threads this kernel has measured between 114 and 517 GFLOP/s depending on the run, but it is a reminder to check the assembly rather than trust the arithmetic. In one session the autotuner of 10.13 found that, at N=512, an 8x1 tile (eight accumulators, nine operands) beat the 4x4; in another the 4x4 won.

Tiling also explains the curious ordering in the benchmark. Tiling alone (85.3 GFLOP/s) is slower than threads alone (149.1), because one core cannot beat sixteen hardware threads however well it uses its registers. But threads alone are limited by each thread's single dependency chain and two loads per FMA, and tiling alone is limited to one core. Together, each of sixteen threads runs a latency-hiding, load-efficient inner loop: 237.8 GFLOP/s in the captured run. On the strided layout, tiling rescues even the column walk (7.1 to 65.1) because the four `j` of a tile share each fetched cache line.

10.9 The flag that matters most: `-fassociative-math`

Look back at the B^T naive inner loop: `vmulps` then `vaddps` into a vector of partial sums. That loop adds `A[i, 0..7] * B[j, 0..7]` lane by lane into eight separate running sums and combines them at the end. In exact arithmetic that is the same as adding the products left to right. In floating point it is not: `(a + b) + c` and `a + (b + c)` can differ in the last bits. The C standard requires gcc to evaluate `acc0 = acc0 + x` in program order, so without permission it may not vectorize a float reduction at all.

The permission is `-fassociative-math`, which the runtime passes by default:

uopc/runtime.py · CFLAGS

```
CFLAGS = os.environ.get("CFLAGS", "-O3 -march=native -fopenmp -ffp-contract=fast -fassociative-math "
                                  "-fno-signed-zeros -fno-trapping-math").split()
```

Here is the scratch experiment that measured it (same kernels, only `CFLAGS` changed, run on AC power in the same session as the peak measurement in 10.3):

```
CFLAGS: -O3 -march=native -fopenmp -ffp-contract=fast -fassociative-math -fno-signed-zeros -fno-trapping-math
  B^T naive                   24.6 GFLOP/s
  B^T 4x4 tile + threads     452.2 GFLOP/s
CFLAGS: -O3 -march=native -fopenmp -ffp-contract=fast
  B^T naive                    4.1 GFLOP/s
  B^T 4x4 tile + threads      53.3 GFLOP/s
```

A factor of six to eight from one flag, on a kernel that is otherwise identical; NOTES 20 records 17 versus 170 in an earlier run, on battery. Every ML compiler makes this trade: reductions are reordered for speed, so a sum in `f32` is reproducible run to run (our kernels are deterministic) but not bit-identical to a sequential

sum. We stop short of `-ffast-math`, which would also let gcc assume there are no NaNs or infinities; that would break `softmax`'s `-INFINITY` initial value and every test that checks NaN propagation. `-fno-signed-zeros` and `-fno-trapping-math` are the small extra permissions `-fassociative-math` needs to be effective; `-ffp-contract=fast` allows fusing `a*b + c` into one FMA.

10.10 Reduce unrolling

The third knob unrolls the innermost reduce axis by a factor `u`: the reduce RANGE shrinks by `u`, and the `u` values are combined in a balanced tree before being added to the accumulator. It is the same `_axis_parts` machinery, applied to reduce axes in `_reduce`. With `Opts(unroll=4)`:

```
for (int ridx2 = 0; ridx2 < 256; ridx2++) {
  float val3 = acc0;
  int alu4 = (ridx2*4);
  float val5 = data0[(alu1+alu4)];
  float val6 = data1[(alu2+alu4)];
  int alu7 = (alu4+alu1);
  float val8 = data0[(alu7+1)];
  int alu9 = (alu4+alu2);
  float val10 = data1[(alu9+1)];
  float val11 = data0[(alu7+2)];
  float val12 = data1[(alu9+2)];
  float val13 = data0[(alu7+3)];
  float val14 = data1[(alu9+3)];
  acc0 = (val3+(((val5*val6)+(val8*val10))+((val11*val12)+(val13*val14))));
}
```

The tree `((p0 + p1) + (p2 + p3))` shortens the dependency chain through `acc0`, the classic reason to unroll a reduction by hand. With an auto-vectorizer behind us, it mostly gets in the way. Measured at N=1024, single-threaded, in the same session:

```
upcast={} thread=None unroll=1:   24.1 GFLOP/s
upcast={} thread=None unroll=4:   24.7 GFLOP/s
upcast={0: 4, 1: 4} thread=None unroll=1:   82.4 GFLOP/s
upcast={0: 4, 1: 4} thread=None unroll=4:   12.6 GFLOP/s
upcast={0: 2, 1: 4} thread=None unroll=1:   64.9 GFLOP/s
```

On its own, unrolling changes nothing (gcc vectorized the plain loop already). Combined with the 4x4 tile it is a disaster: the assembly of the unrolled tile contains only scalar instructions (`vfmadd231ss`, `vmulss`,

`vaddss`) and not a single packed one. Sixty-four unrolled products feeding sixteen trees were more than gcc's vectorizer would untangle, so it gave up. `hand_opt` uses `unroll` only for short reductions with no tile, where it removes the loop entirely (the `test_reduce_unroll_has_no_inner_loop` test unrolls a 16-element sum into straight-line code). The general lesson: **when you generate C, gcc is part of your backend, and its heuristics are part of your cost model.**

10.11 Cache blocking the output axes

A register tile reuses each loaded value from a register. Cache blocking (also called loop tiling) does the same one level up: it reorders the output loops so that the operand rows a group of outputs needs are still in L1 or L2 when the next output wants them.

The working set

The caches of this machine, as `lscpu -C` reports them:

```
NAME ONE-SIZE ALL-SIZE WAYS TYPE        LEVEL  SETS PHY-LINE COHERENCY-SIZE
L1d       48K     416K   12 Data            1    64        1             64
L1i       32K     448K    8 Instruction     1    64        1             64
L2       1.3M     9.5M   10 Unified         2  2048        1             64
L3        24M      24M   12 Unified         3 32768        1             64
```

Each P-core has 48 KB of L1d and 1.25 MB of L2 (`/sys/devices/system/cpu/cpu0/cache/index2/size` says `1280K`), shared by its two hardware threads. The four E-cores share one 2 MB L2 (9.5 MB in all is 6 x 1.25 MB + 2 MB). The 24 MB L3 is shared by every core.

Now follow the data through the `4x4 tile + threads` kernel at N = K = 1024. The THREAD loop hands each thread strips of four rows of `A`. For one strip, the loop over `j` visits all 256 groups of four `B^T` rows, and the `k` loop of each tile reads:

- four rows of `A`, 4 x 4 KB = 16 KB. They are the same for every `j` of the strip, so they stay in L1 and are reused 256 times;
- four rows of `B^T`, another 16 KB. Each group is used once per strip. By the time the next strip needs it again, the thread has read all of `B^T`, 4 MB, which does not fit in L2. It comes back from L3.

Per strip that is 4 MB of `B^T` from L3 for 4 x 1024 x 1024 x 2 = 8.4 million FLOPs: 2 FLOPs per byte of L3 traffic.

Blocking the output into 64x64 squares changes the order. All the tiles of one block are computed before moving on, so the block needs 64 rows of `A` (256 KB) and 64 rows of `B^T` (256 KB). That 512 KB fits in a P-core's L2, even with the second hardware thread working on another block. Each `B^T` row is now read from L3 once per block and then reused from L2 by the block's sixteen row strips. L3 traffic per block is 512 KB for 64 x 64 x 1024 x 2 = 8.4 million FLOPs, 16 FLOPs per byte: eight times less traffic to L3 than without blocks.

The implementation

`Opts.tile` maps an output axis to a block size. `lower` splits each tiled axis into a block loop and a loop inside the block, and creates all the block loops first, so they get the lowest range indices and the linearizer places them outermost:

uopc/rangeify.py:180-194

```
    # the THREAD axis gets the lowest range index, so it is the outermost loop
    order = ([o.thread] if o.thread is not None else []) + [k for k in range(len(root.shape)) if
k != o.thread]
    tiled = {k: t for k, t in o.tile.items() if 1 < t < root.shape[k] and root.shape[k] % t == 0
and t % o.upcast.get(k, 1) == 0}
    parts: dict[int, tuple[UOp, int]] = {}
    # tiled axes: first every block loop (outer), then the loops inside a block. A block of
outputs reuses
    # the same rows of each operand, so they stay in cache while the block is computed
    outer = {k: ctx.rng(root.shape[k] // t, AxisType.THREAD if k == o.thread else AxisType.LOOP)
for k, t in tiled.items()}
    for k in order:
      f = o.upcast.get(k, 1)
      if k in tiled:
        inner = tiled[k] // f
        parts[k] = (outer[k] * inner + ctx.rng(inner) if inner > 1 else outer[k], f)
      else:
        parts[k] = _axis_parts(ctx, root.shape[k], f, AxisType.THREAD if k == o.thread else
AxisType.LOOP)
    plist = [parts[k] for k in range(len(root.shape))]
```

A tiled axis of size `s` with block `t` and upcast `f` becomes `outer * (t/f) + inner`, with `outer` of size `s/t` and `inner` of size `t/f`. The upcast constant is added later by `_combos`, as for an untiled axis, so the register tile still works inside a block. If the THREAD axis is tiled, its block loop gets the THREAD type:

OpenMP then divides whole blocks among the threads. A tile size that does not divide the axis, is not a multiple of the upcast, or covers the whole axis is ignored.

For a 128x128 GEMM with `Ctx(opts=Opts(upcast={0: 4, 1: 4}, thread=0, tile={0: 64, 1: 64}))`, `render_c` produces the following (abridged: the `...` lines stand for the same pattern repeated for the other accumulators and stores):

```
void gemm(float* restrict data0, float* restrict data1, float* restrict data2) {
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 2; ridx0++) {
    for (int ridx1 = 0; ridx1 < 2; ridx1++) {
      for (int ridx2 = 0; ridx2 < 16; ridx2++) {
        int alu1 = ((ridx0*8192)+(ridx2*512));
        for (int ridx3 = 0; ridx3 < 16; ridx3++) {
          int alu2 = ((ridx1*64)+(ridx3*4));
          int alu3 = (alu1+alu2);
          float acc0;
          acc0 = 0.0f;
          ...
          float acc15;
          acc15 = 0.0f;
          int alu4 = (alu2+alu1);
          for (int ridx4 = 0; ridx4 < 128; ridx4++) {
            float val5 = acc0;
            int alu6 = (alu1+ridx4);
            float val7 = data0[alu6];
            int alu8 = (((ridx1*8192)+(ridx3*512))+ridx4);
            float val9 = data1[alu8];
            acc0 = (val5+(val7*val9));
            ...
          }
          float val31 = acc0;
          data2[alu3] = val31;
          ...
        }
      }
    }
  }
}
```

`ridx0` and `ridx1` are the block loops (two blocks of 64 per axis), `ridx2` and `ridx3` walk the sixteen 4x4 tiles of a block along each axis, and `ridx4` is the shared reduce loop of 10.8. The row of `A` is `ridx0*64 + ridx2*4` (hence `ridx0*8192 + ridx2*512` in elements), and the row of `B^T` is `ridx1*64 + ridx3*4`. The test checks the results against NumPy for three tilings and counts one extra loop per tiled axis:

`tests/test_w06_opt.py · test_cache_blocking_correct`

```python
@pytest.mark.parametrize("tile", [{0: 8, 1: 16}, {1: 32}, {0: 16}])
def test_cache_blocking_correct(tile):
  M, N, K = 32, 64, 48
  a, b = np.random.randn(M, K).astype(np.float32), np.random.randn(K, N).astype(np.float32)
  c = np.zeros(M * N, np.float32)
  src = render_c(lower(gemm_graph(M, N, K), buf(2, M * N), Ctx(opts=Opts(upcast={0: 4, 1: 4},
thread=0, tile=tile))), "k")
  Program(src, "k")(a, b, c)
  np.testing.assert_allclose(c.reshape(M, N), a @ b, rtol=1e-4, atol=1e-4)
  assert src.count("for (") == 3 + len(tile)                # one extra block loop per tiled axis
```

The measurement

The captured runs show the blocked kernel ahead at N=1024 and level at N=2048 (`book/outputs/bench_gemm_2048.txt`):

```
N=2048  (17.18 GFLOP per GEMM)
  ...
  B^T        4x4 tile + threads      334.7 GFLOP/s   max err 1.2e-04
  B^T        + 64x64 cache blocks    337.9 GFLOP/s   max err 1.2e-04
```

The 1024 row (288.5 against 237.8) looks like a 20% gain, but an earlier run on a quieter machine had it the other way round: 430 with 64x64 blocks against 517 without. To separate the effect from the noise, this scratch script compiles four variants and times them in turn, three rounds, so that each round sees roughly the same machine state:

```python
# tiles.py: 4x4 tile + threads, with and without cache blocks, interleaved over three rounds
import sys; sys.path.insert(0, "examples")
import numpy as np
from bench_gemm import gemm_graph, bench
from uopc.uop import UOp, Ops, ParamArg
from uopc.dtype import dtypes
from uopc.rangeify import lower, Ctx, Opts
from uopc.render_c import render_c
from uopc.runtime import Program
for N in (1024, 2048):
  a, bt, c = np.random.randn(N, N).astype(np.float32), np.random.randn(N, N).astype(np.float32),
np.zeros(N * N, np.float32)
  out = UOp(Ops.BUFFER, (), ParamArg(2, dtypes.f32, (N * N,)))
  variants = [("no blocks", {})] + [(f"{t}x{t} blocks", {0: t, 1: t}) for t in (32, 64, 128)]
  prgs = [(n, Program(render_c(lower(gemm_graph(N, True), out, Ctx(opts=Opts(upcast={0: 4, 1: 4},
thread=0, tile=t))), "gemm"), "gemm")) for n, t in variants]
  res = {n: [] for n, _ in prgs}
  for _ in range(3):
    for n, p in prgs: res[n].append(2 * N**3 / bench(lambda: p(a, bt, c)) / 1e9)
  print(f"N={N}")
  for n, v in res.items(): print(f"  {n:14s}" + "".join(f"{x:8.1f}" for x in v) + " GFLOP/s")
```

```
N=1024
  no blocks        237.9   433.4   429.9 GFLOP/s
  32x32 blocks     327.2   426.4   399.9 GFLOP/s
  64x64 blocks     496.8   403.6   382.0 GFLOP/s
  128x128 blocks   312.1   280.4   296.0 GFLOP/s
N=2048
  no blocks        385.0   379.3   313.3 GFLOP/s
  32x32 blocks     379.4   337.2   317.9 GFLOP/s
  64x64 blocks     330.8   366.9   321.2 GFLOP/s
  128x128 blocks   333.3   377.8   363.6 GFLOP/s
```

Read the columns. In the first round at N=1024 the unblocked kernel, timed first, measured 237.9 and 64x64 blocks 496.8; in the second and third rounds the blocks were slower. At N=2048 no variant wins consistently. The one stable result is that 128x128 blocks are the slowest at N=1024. On this machine, once the layout pass has made the reduce axis contiguous, cache blocking of the output axes did not help reliably. Four reasons:

1. **The traffic it removes was not the bottleneck.** Without blocks, the kernel reads `B^T` from L3 at 2 FLOPs per byte. At 240 to 500 GFLOP/s that is 120 to 250 GB/s across the chip, and the measurements say L3 delivers it: cutting that traffic by 8x did not make the kernel faster. Each `k` loop reads eight rows sequentially, 4 KB each, which is the easiest possible pattern for the hardware prefetchers. They bring the lines of `B^T` into L2 ahead of use, whether or not a block keeps them there.

2. **The register tile already provides the reuse that matters most.** The rows of `A` for a strip already stay in L1 (16 KB of 48), and every loaded vector feeds four FMAs. What limits the inner loop now is in the core: the five spilled accumulators of 10.8, and the horizontal sum at the end of every 1024-long dot product.
3. **Blocks are coarser units of parallel work.** With a tiled THREAD axis, OpenMP divides whole blocks among the threads. At N=1024 there are 1024/64 = 16 block rows for 16 hardware threads, one each, so a thread that lands on an E-core or shares its core with another process holds up the rest. With 128x128 blocks there are only 8 block rows, and half of the hardware threads have nothing to do; that is why the 128x128 row is consistently the slowest at N=1024.
4. **The block's working set grows with `K`.** A 64x64 block needs 2 x 64 x K x 4 bytes of operand rows: 512 KB at K=1024, 1 MB at K=2048 (two hardware threads on one core then need 2 MB, more than the 1.25 MB L2), and 2 MB at K=4096.

Blocking does help when the unblocked kernel really does run out of cache. At N=4096, `B^T` is 64 MB, larger than the 24 MB L3, so without blocks every strip streams `B^T` from DRAM at 2 FLOPs per byte. With 55.9 GB/s that alone would cap the kernel near 112 GFLOP/s, except that threads working on neighbouring strips at the same time share some of what is in L3. With blocks, the operand rows of a block (2 MB for 64x64 at K=4096, 1 MB for 32x32) are reused from L3 or L2 instead. The same script at N=4096:

```
N=4096
  no blocks        253.0   186.7   180.6 GFLOP/s
  32x32 blocks     299.4   250.7   238.5 GFLOP/s
  64x64 blocks     296.4   234.0   231.1 GFLOP/s
  128x128 blocks   294.7   226.0   238.9 GFLOP/s
```

Every blocked variant beats the unblocked one in every round, by 15 to 35%. (An earlier run of the same script had the blocked variants ahead in eight of nine comparisons.)

So blocking the output axes helps when the operands do not fit in L3. When `K` is large, a block's rows no longer fit in L2 however small the block is along `i` and `j`, and the fix is to block the reduce axis as well: split `k` into chunks of, say, 256, and compute every tile of the block for one chunk before moving to the next, so that the block needs 2 x 64 x 256 x 4 = 128 KB of operand rows for any `K`. That changes the accumulators. Today each tile's sixteen sums live in registers from the start of the `k` loop to the store, and the ALLOC sits inside the output loops. With `k` split and the chunk loop outside the tile loops, each tile is visited once per chunk, so its partial sums must survive between visits: the block needs a persistent accumulator buffer of 64 x 64 floats (16 KB) per thread, zeroed before the first chunk and written to `C` after the last, or `C` itself must be zeroed and accumulated into. That is exercise 6. It is also the loop structure of

OpenBLAS and BLIS, where the `k` block is chosen so that a packed panel of `B` fits in L2 (Goto and van de Geijn, 2008).

10.12 SIMD: what gcc already does

Every fast row of the benchmark depends on 8-wide AVX2 instructions, yet `uopc` never emits a vector type. gcc's loop vectorizer does it. You can check this for any kernel by compiling the generated C to assembly with the runtime's own flags and `-S -o -`:

```
# simd.py: compile generated GEMM kernels to assembly with the runtime's flags and count FMAs
import subprocess, sys
sys.path.insert(0, "examples")
from bench_gemm import gemm_graph
from uopc.uop import UOp, Ops, ParamArg
from uopc.dtype import dtypes
from uopc.rangeify import lower, Ctx, Opts
from uopc.render_c import render_c
from uopc.runtime import CFLAGS
N = 1024
out = UOp(Ops.BUFFER, (), ParamArg(2, dtypes.f32, (N * N,)))
for o in [Opts(upcast={0: 4, 1: 4}, thread=0), Opts(upcast={0: 4, 1: 4}, thread=0, unroll=4)]:
  src = render_c(lower(gemm_graph(N, True), out, Ctx(opts=o)), "gemm")
  asm = subprocess.run(["gcc", *CFLAGS, "-x", "c", "-", "-S", "-o", "-"], input=src,
capture_output=True, text=True).stdout
  fma = [l.strip() for l in asm.splitlines() if l.strip().startswith("vfmadd")]
  print(f"unroll={o.unroll}: {sum(f.split()[0].endswith('ps') for f in fma)} packed (ps), "
        f"{sum(f.split()[0].endswith('ss') for f in fma)} scalar (ss) FMA instructions")
  for f in fma[:3]: print("   ", f)
```

```
unroll=1: 16 packed (ps), 0 scalar (ss) FMA instructions
    vfmadd231ps %ymm6, %ymm1, %ymm10
    vfmadd231ps %ymm4, %ymm1, %ymm11
    vfmadd231ps %ymm2, %ymm1, %ymm15
unroll=4: 0 packed (ps), 32 scalar (ss) FMA instructions
    vfmadd231ss -16(%r14), %xmm13, %xmm1
    vfmadd231ss -8(%r14), %xmm7, %xmm0
    vfmadd231ss 4080(%r14), %xmm13, %xmm1
```

`vfmadd231ps` on `ymm` registers is a packed single-precision FMA on eight lanes; `vfmadd231ss` on `xmm` registers works on one lane. The tiled kernel has sixteen packed FMAs, one per accumulator, exactly the loop listed in 10.8. gcc vectorized it along `k`: each scalar accumulator became a vector of eight partial sums, each

`vmovups` loads eight consecutive `k` of one row, and after the loop a short sequence (`vextractf128`, `vaddps`, `vmovhlps`) adds the eight lanes of each accumulator together. That reordering of the sum is what `-fassociative-math` (10.9) permits. With `unroll=4`, the same tile gets no packed FMAs at all, which is the 6.5x slowdown measured in 10.10.

So `uopc` gets SIMD for free, but only as long as gcc's heuristics agree, and only along the axis gcc chooses. Explicit vectorization in the IR would take that decision into the compiler. It needs three things that `uopc` does not have:

- **A vector dtype**, for example `float8` (eight `f32` lanes), rendered in C as `float __attribute__((vector_size(32)))` or as an `__m256`.
- **A VECTORIZE op** that groups the copies an upcast axis produces into one vector value, and its inverse, an op that extracts one lane. When an upcast axis has factor 8 and its index is contiguous in memory, the eight scalar loads become one vector load, the eight FMAs one vector FMA, and the eight stores one vector store. In a GEMM the natural choice is the `j` axis of the output tile: broadcast one `A[i, k]`, multiply it by eight consecutive values along `j`, and the result is eight finished outputs, with no horizontal sum at the end.
- **Aligned loads.** `vmovaps` requires an address that is a multiple of 32 bytes. Using it needs a guarantee that the buffer is aligned and a proof from the symbolic layer that the index is a multiple of 8. Without that proof, the renderer must emit unaligned loads (`vmovups`), which current cores execute at the same speed as aligned ones when the data happens to be aligned.

tinygrad has all three (its `dtype.vec`, `Ops.VECTORIZE` and `Ops.GEP`). Adding them to `uopc` is exercise 10.

10.13 Choosing the opts

A hand-written heuristic

`hand_opt_choice` encodes the lessons of this chapter in a dozen lines:

`uopc/opt.py · hand_opt_choice`

```python
def hand_opt_choice(body: UOp) -> Opts:
  shape, rsize = body.shape, _reduce_size(body)
  o = Opts()
  if rsize > 1:
    # register tile: 4x4 on the last two output axes. With AVX2's 16 vector registers, 16 accumulators plus
    # the loaded operands don't quite fit (gcc spills a few), but measured it beats 2x8, 4x8 and 8x8 here
    budget = 16
    for axis, want in [(len(shape) - 1, 4), (len(shape) - 2, 4)]:
      if axis < 0: continue
      for f in (want, 4, 2):
        if f <= budget and shape[axis] % f == 0 and shape[axis] >= f and f > 1:
          o.upcast[axis] = f; budget //= f; break
    if not o.upcast and rsize % 4 == 0 and rsize < 64: o.unroll = 4
  # threads: only when there is enough work to amortize waking them up
  work = prod(shape or (1,)) * rsize
  if work >= 1 << 16:
    for axis, s in enumerate(shape):
      if s // o.upcast.get(axis, 1) >= 2:
        o.thread = axis; break
  return o
```

If the kernel reduces, tile the last two output axes 4x4 (falling back to 2 if the size is not divisible); if it is a short reduction with no tile, unroll it; and if there is at least `2^16` units of work, thread the first axis that still has at least two iterations after tiling. Applied to a few shapes:

```
GEMM 1024^3            Opts(upcast={1: 4, 0: 4}, thread=0, unroll=1, tile={}, gpu=False, local={})
row max (128,10)       Opts(upcast={0: 4}, thread=None, unroll=1, tile={}, gpu=False, local={})
elementwise (784,128)  Opts(upcast={}, thread=0, unroll=1, tile={}, gpu=False, local={})
elementwise (128,)     Opts(upcast={}, thread=None, unroll=1, tile={}, gpu=False, local={})
sum (16,)              Opts(upcast={}, thread=None, unroll=4, tile={}, gpu=False, local={})
```

The `row max (128, 10)` case is the `log_softmax` kernel from the MNIST step: four rows per iteration, no threads (only 1280 elements). The heuristic never sets `tile`, for the reasons measured in 10.11. `HandOpt` packages the layout pass and this heuristic for the kernelizer; it is the default (`OPT=1`), and `OPT=0` turns both off:

`uopc/opt.py · HandOpt`

```python
class HandOpt:
  """The default optimizer kernelize uses: layout pass + hand-coded opts."""
  layout = staticmethod(layout_pass)
  lower = staticmethod(hand_opt)
```

Search

A heuristic encodes what you know. Search finds what you do not. `search_opt` compiles a list of candidate `Opts` for the kernel, times each on random data, and keeps the fastest:

uopc/opt.py · candidate_opts

```python
def candidate_opts(body: UOp) -> list[Opts]:
  shape, rsize = body.shape, _reduce_size(body)
  outs = [a for a, s in enumerate(shape) if s > 1]
  cands = [Opts(), hand_opt_choice(body)]
  for a, b in itertools.product(outs[-2:], repeat=2):
    for fa, fb in [(4, 4), (8, 4), (4, 8), (8, 8), (16, 2), (2, 16)]:
      up = {a: fa} if a == b else {a: fa, b: fb}
      if all(shape[k] % f == 0 for k, f in up.items()):
        for u in (1, 4) if rsize % 4 == 0 else (1,):
          cands.append(Opts(dict(up), hand_opt_choice(body).thread, u))
  # cache blocking of the last two output axes (helps when operand rows don't fit in cache)
  base = hand_opt_choice(body)
  for t in (32, 64, 128):
    tile = {a: t for a in outs[-2:] if shape[a] % t == 0}
    if tile: cands.append(Opts(dict(base.upcast), base.thread, base.unroll, tile=tile))
  return cands
```

uopc/opt.py · search_opt

```python
def search_opt(body: UOp, out: UOp, inputs: list[np.ndarray] | None = None, reps: int = 3) ->
UOp:
  """Time every candidate on random data, keep the fastest."""
  from uopc.render_c import render_c
  from uopc.runtime import Program, np_dtype
  bufs = sorted({u for u in body.toposort() if u.op is Ops.BUFFER}, key=lambda u: u.arg.slot)
  data = inputs or [np.random.randn(b.arg.shape[0]).astype(np_dtype(b.dtype)) for b in bufs]
  res = np.zeros(out.arg.shape[0], np_dtype(out.dtype))
  best, best_t = None, float("inf")
  seen = set()
  for o in candidate_opts(body):
    key = (tuple(sorted(o.upcast.items())), o.thread, o.unroll, tuple(sorted(o.tile.items())))
    if key in seen: continue
    seen.add(key)
    ast = lower(body, out, Ctx(opts=o))
    prg = Program(render_c(ast, "k"), "k")
    ts = []
    for _ in range(reps):
      t0 = time.perf_counter(); prg(*data, res); ts.append(time.perf_counter() - t0)
    if min(ts) < best_t: best, best_t = ast, min(ts)
  return best
```

It is a one-step version of tinygrad's `BEAM` search: no beam, no iteration, just "try the plausible register tiles on the last two axes, with and without unrolling, plus 32, 64 and 128 cache blocks on those axes, and measure". Enable it with `OPT=2` (`SearchOpt`). Here is what it sees for a 512x512 GEMM (a scratch copy of `search_opt` that prints every candidate's time instead of only keeping the best; AC power, same session as 10.3):

```
29 distinct candidates; hand_opt picks {1: 4, 0: 4} thread=0
   454.2 GFLOP/s  upcast={0: 4, 1: 4} thread=0 unroll=1
   391.5 GFLOP/s  upcast={0: 8} thread=0 unroll=1
   347.5 GFLOP/s  upcast={0: 4, 1: 8} thread=0 unroll=1
   345.7 GFLOP/s  upcast={0: 4} thread=0 unroll=1
   292.2 GFLOP/s  upcast={0: 8, 1: 4} thread=0 unroll=1
   253.1 GFLOP/s  upcast={0: 8, 1: 8} thread=0 unroll=1
  ...
    39.9 GFLOP/s  upcast={1: 16} thread=0 unroll=4
    38.6 GFLOP/s  upcast={0: 4, 1: 8} thread=0 unroll=4
    28.8 GFLOP/s  upcast={} thread=None unroll=1
```

This time the heuristic's 4x4 tile won. In an earlier session on battery the same search put `{0: 8}` first (117.8 GFLOP/s) and the 4x4 second (102.8), so the top two swap places with the machine's state. Every candidate with `unroll=4` is near the bottom in both sessions, as 10.10 predicted. The count is 29 although `candidate_opts` returns 53 entries: most are duplicates. This run was made with the first version of `search_opt`, whose deduplication key left out `tile`, so the three cache-blocked candidates were silently

skipped as well (exercise 11). With `tile` in the key there are 32 candidates; in a later run on the same machine none of the blocked ones won at N=512. Search pays for its answers: compiling and timing the 29 variants took 3.6 seconds of wall time with an empty compile cache, and 0.56 seconds once the runtime had cached the compiled kernels on disk. Timing still runs every time, which is why search is opt-in and why production autotuners also cache their *choices* on disk, keyed by the kernel (exercise 8).

10.14 Results

Putting it together, the captured table from 10.1, with the reason for each step:

version	GFLOP/s	what changed
B strided, naive	5.2	column walk: new line and new page every `k`
B strided, threads	7.1	sixteen threads fighting over caches and TLBs
B strided, 4x4 tile + threads	65.1	four `j` share each line; still strided
B^T, naive	28.0	contiguous and vectorized, but one dependency chain
B^T, threads	149.1	16 threads x the naive inner loop
B^T, 4x4 tile	85.3	2 FMAs per load, 16 independent chains, one core
B^T, 4x4 tile + threads	237.8	both
B^T, + 64x64 cache blocks	288.5	within run-to-run noise at this size (10.11)
Tensor `A @ B` (layout pass + `hand_opt`)	321.4	the compiler does all of the above, transpose included
numpy (OpenBLAS)	449.4	cache blocking with packing, hand-written microkernels

The last four rows are close enough that their order changes between runs; on a quiet machine the `4x4 tile + threads` kernel reached 517 GFLOP/s against NumPy's 542 at N=1024, and at N=2048 392 against 243. What does OpenBLAS do that we do not? Three things, each a good final project. **Cache blocking of all three loops**: it blocks `k` as well as `i` and `j`, so that a block of `B` stays in L2 for any matrix size (10.11 blocks only the output axes). **Packing**: it copies blocks of both operands into contiguous, tile-ordered buffers,

so the microkernel's loads are perfectly sequential. **A hand-scheduled microkernel**: a register tile shaped for the register file (for example 6x16 on AVX2: twelve vector accumulators, a few operand registers, no spills), with explicit broadcast loads and prefetches. Each one is a few dozen lines on top of what you have. They matter most where our kernel is weakest: large `K` and matrices that do not fit in L3, where 10.11 shows the output-only blocking start to help.

Pitfalls

Threads made it slower (NOTES 19). Adding `#pragma omp parallel for` to the strided GEMM took it from 1.9 to 1.1 GFLOP/s on battery, and on AC power sixteen threads gained only 1.4x (5.2 to 7.1). Parallelism multiplies memory pressure along with compute. Check locality (strides along the innermost loop) before threading, and expect threads to help only once each thread's inner loop is efficient.

The missing flag (NOTES 20). Without `-fassociative-math` gcc will not vectorize a float reduction: the same tiled kernel ran at 53.3 instead of 452.2 GFLOP/s. If a reduce kernel's assembly shows only `ss` (scalar single) instructions, check the flags first.

The layout pass that copied too much (NOTES 21). A layout pass that stages any strided view will eventually stage a broadcast product larger than the computation it serves (a `(128, 784, 128)` tensor in MNIST's backward pass, a 21 ms kernel). Copy operands *before* their EXPAND, and only when the copy is strictly smaller than the reduce input.

Spilled accumulators. A 4x4 tile has sixteen vector accumulators, and with eight operand vectors that is more than AVX2's sixteen registers; gcc spills five of them to the stack every iteration. The kernel is still fast, but "it fits in registers" is a claim to verify with `gcc -S`, not to assume.

Unrolling can defeat the vectorizer. `Opts(upcast={0: 4, 1: 4}, unroll=4)` compiled to purely scalar code and ran 6.5x slower than the same tile without unrolling. A transformation that helps a naive compiler can hurt a smart one.

The microbenchmark the compiler optimized away. The first version of `peak.c` gave all FMA chains the same initial values; gcc merged them and the measured "peak" was six times too high per instruction. Read the assembly of every microbenchmark.

Fixed costs dominate small kernels. A fresh output buffer is page-faulted on first touch inside the kernel (`DEBUG=3`'s "again" column runs each kernel a second time to show the difference: `0.603 ms` versus `0.359`

`ms` for an AdamW update), and OpenMP's fork/join costs microseconds. Do not tune a 50-microsecond kernel by staring at its inner loop.

Noisy benchmarks. Laptop clocks depend on power source and temperature, the hybrid P/E cores make thread placement matter, and other processes steal cores. The tiled GEMM at N=1024 measured anywhere from 114 to 517 GFLOP/s in this chapter, and single rows of one table can even swap order. Take the best of several runs, interleave the variants you compare (as `tiles.py` in 10.11 does), and report the machine state with the numbers.

A gain that was noise. The captured benchmark showed 64x64 cache blocks 21% faster than the plain tiled kernel at N=1024; the interleaved measurement in 10.11 showed the sign of that difference changing from round to round. Before you keep an optimization, check that its effect is larger than the spread between repeated runs.

Exercises

1. ★ **LayerNorm on the roofline.** Compute the arithmetic intensity of a LayerNorm over the last axis of a `(B, D)` tensor (read `x`, `weight`, `bias`, write the output). Where does it sit on this machine's roofline, and what is its best possible GFLOP/s?

2. ★ **Predict a register tile.** Predict the generated C for `Opts(upcast={1: 2})` on an 8x8 GEMM: how many accumulators, how many loads of `A` and of `B` per `k` iteration? Check with `render_c(lower(...))`.

3. ★ **Latency-bound adds.** Explain from the assembly in 10.6 why the B^T naive kernel cannot exceed one vector add per add latency. Estimate its GFLOP/s at 3.7 GHz (the clock measured in 10.3) for an add latency of 4 cycles, then compare with the measured 28.0 GFLOP/s. What latency does the measurement imply?

4. ★★ **The largest tile that fits.** Add a `(4, 3)` and a `(2, 8)` tile to `candidate_opts`. Using `gcc -S`, find the largest tile that does not spill on AVX2 and compare its speed with 4x4 at N=1024.

5. ★★ **Split-K.** Implement split-K: for a GEMM with a small output (say `(16, 16) @ (16, 65536)^T`), threads on the output axes have nothing to do. Split the reduce axis into `T` chunks computed in parallel into a temporary `(T, M, N)`, then sum. Where in the pipeline does this belong: `opt.py`, rangeify, or the kernelizer?

6. ★★★ **Block the reduce axis.** Block the reduce axis. Add an `Opts` field `ktile` that splits the innermost reduce axis into an outer loop over chunks of, say, 256 and an inner loop, and place the chunk loop outside the tile loops of a cache block (10.11), so the loop order is block `i`, block `j`, chunk `k`, tile `i`, tile `j`, inner `k`. The sixteen accumulators of a tile can no longer live in registers across the whole reduction: allocate a persistent accumulator buffer of `tile_i x tile_j` floats per block (`Ctx.alloc` takes a size), zero it before the first chunk, add each chunk's partial sums into it, and store it to the output after the last chunk. Check correctness against NumPy for a `K` that is not a power of two, then measure at N=2048 and N=4096 against output blocking alone. *Hint:* where the ALLOC is placed decides whether it is per-thread; reread the scope rule in 10.7.

7. ★★ **OpenMP scheduling on a hybrid CPU.** Make OpenMP use `schedule(dynamic)` or `schedule(guided)` for THREAD loops and measure the B^T threads row on this hybrid (P-core plus E-core) CPU. Explain the result.

8. ★★ **Cache search results.** Give `search_opt` a persistent cache: key it by the kernel's rendered C *before* opts (or by `_structure_key`), store the winning `Opts` in `~/.cache/uopc`, and verify a second run of a benchmark does no timing.

9. ★★ **Duplicate index terms.** Find the cause of the duplicated `alu3 = (alu1+alu2)` / `alu4 = (alu2+alu1)` in the tiled kernels and fix it in the symbolic layer, without breaking `tests/test_w01_uops.py`.

10. ★★★ **Explicit float8 code generation.** Explicit `float8` code generation (10.12). Add a vector dtype (`dtypes.f32.vec(8)`), a VECTORIZE op that packs eight scalar UOps into one vector, and an op that extracts one lane. After rangeify, when an upcast axis has factor 8 and its index is contiguous in memory, rewrite the eight copies of each load, ALU op and store into one vector op. Render vectors as `float __attribute__((vector_size(32)))`, and use an unaligned load (`memcpy` into the vector, or `_mm256_loadu_ps`) unless you can prove the index is a multiple of 8. Vectorize the GEMM tile along `j` (broadcast `A[i, k]`, eight outputs per vector, no horizontal sum), check with `gcc -S` that the result uses packed FMAs even with `unroll=4`, and compare its speed with the best tile from exercise 4.

11. ★ **A key that forgot a field.** The first version of `search_opt` deduplicated candidates by `(upcast, thread, unroll)`. Explain why that silently dropped all three cache-blocked candidates, and why no test failed. Write a test that would have caught it.

12. ★ **Work per thread.** For the tiled GEMM at N=1024 with `thread=0`, how many iterations does the parallel loop have with no cache blocks, with 64x64 blocks and with 128x128 blocks? Use the answer to explain the slowest row of the N=1024 table in 10.11.

Solutions to ★ exercises

1. Per element: subtract mean, square, multiply by the reciprocal standard deviation, multiply by the weight, add the bias, plus the two reductions: roughly 8 FLOPs. Bytes: read `x` (4) and write the output (4); `weight` and `bias` are `D` floats reused across all rows, and if the mean and variance passes re-read `x` from cache rather than DRAM, the compulsory traffic is about 8 bytes per element. AI is about 1 FLOP/byte, far left of the ridge (14): memory bound, at most about 56 GFLOP/s on this machine with the 55.9 GB/s roof of 10.3.

2. Two accumulators (the 2 tile columns), one shared `k` loop; per iteration one load of `A` (`A[i, k]`, shared by both) and two of `B` (`B^T[2j, k]`, `B^T[2j+1, k]`): three loads for two FMAs.

3. Every `vaddps` writes `%ymm1` and the next one reads it, so at most one vector add completes per add latency. Each add finishes 8 lanes x 2 FLOPs (the multiply is independent and overlaps), so with a 4-cycle latency the bound is 4 FLOP/cycle, about 15 GFLOP/s at 3.7 GHz. The measured 28.0 is nearly twice that, which implies about 2 cycles per iteration: 28.0 / 3.7 GHz is 7.6 FLOP/cycle, close to 16 FLOPs every 2 cycles. The P-cores of this CPU (Raptor Cove, derived from Golden Cove) have dedicated floating-point adders with a 2-cycle latency, half that of the FMA unit, so a chain of plain adds runs twice as fast as a chain of FMAs.

11. The cache-blocked candidates copy `upcast`, `thread` and `unroll` from `hand_opt_choice`, so with the old key they looked identical to the heuristic's own candidate, which was seen first, and were skipped. Every test still passed because search only ever picks *some* valid candidate: dropping candidates changes speed, never results. A test that checks the set of timed candidates (for example, that every `tile` from `candidate_opts` reaches the timing loop) catches it. The fix adds `tuple(sorted(o.tile.items()))` to the key.

12. The THREAD loop is the first axis. Without blocks it runs over tiles of 4 rows: 1024 / 4 = 256 iterations, 16 per hardware thread. With blocks it runs over block rows: 1024 / 64 = 16 (one per hardware thread) and 1024 / 128 = 8. With 8 iterations, half of the 16 hardware threads have no work, so the 128x128 variant is the slowest in every round.

Quiz

1. What is the arithmetic intensity of a 1024x1024x1024 `f32` GEMM, counting only compulsory DRAM traffic? Is it memory- or compute-bound on a machine with 777 GFLOP/s and 55.9 GB/s?
2. Why does reading a column of a row-major 1024x1024 `f32` matrix touch a new page on every element?
3. A 4x4 register tile: how many loads and FMAs per `k` step, and why does the tile also help with FMA latency?
4. Why must the accumulator of a parallel GEMM be declared inside the parallel loop?
5. What does `-fassociative-math` allow gcc to do, and why is it acceptable for ML kernels?

6. Why does the layout pass stage the operand *before* its EXPAND rather than the expanded view?
7. Why might adding threads make a kernel slower?
8. In one session `search_opt` measured `{0: 8}` faster than the heuristic's `{0: 4, 1: 4}`. Name one reason a heuristic based on "reuse per load" can pick the wrong tile.
9. 64x64 cache blocks cut the L3 traffic of the 1024 GEMM by 8x, yet did not make it reliably faster. Give two reasons, and name a case where blocking does help.

Answers

1. `2N^3 / (12N^2) = N/6 = 171` FLOP/byte. The ridge is `777 / 55.9 = 14`, so it is compute-bound (by a factor of twelve).
2. Consecutive elements of a column are one row apart: 1024 x 4 bytes = 4096 bytes, exactly one 4 KB page, and also a different 64-byte cache line.
3. 8 loads (4 of `A`, 4 of `B`) and 16 FMAs. The 16 accumulators are independent dependency chains, so the pipelined FMA units can overlap them instead of waiting on one chain's latency.
4. Each thread computes different outputs and needs its own running sum; a shared accumulator would be a data race. The linearizer places ALLOCs in the deepest common scope, inside the THREAD loop.
5. Reorder floating-point additions (for example into 8 lane-wise partial sums), which is what vectorizing a reduction requires. ML results already tolerate last-bit differences in sums; the kernels stay deterministic, and NaN/inf semantics are kept.
6. The unexpanded operand is `N*K` floats; the expanded view is `M*N*K`. Copying after the expand would materialize the broadcast and cost more than the GEMM.
7. Fork/join overhead on small kernels, and memory contention: threads multiply cache, TLB and bandwidth pressure, which can dominate when each thread's access pattern is poor.
8. Register pressure: a tile whose accumulators and operands exceed the register file spills; also gcc's vectorization choices, cache-line alignment of the tile, and the machine's load ports are not in the model. (And run-to-run noise: the two tiles swapped places between sessions.)
9. The removed traffic was not the bottleneck (L3 and the prefetchers already kept up with sequential row reads, and the register tile already keeps the `A` rows in L1); blocks are coarse units of parallel work (16 or 8 per parallel loop at N=1024); and the block's working set grows with `K`. Blocking helps when the operands do not fit in L3: at N=4096 every blocked variant beat the unblocked one.

Further reading

- S. Williams, A. Waterman, D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures", *Communications of the ACM*, 2009.
- K. Goto and R. van de Geijn, "Anatomy of High-Performance Matrix Multiplication", *ACM TOMS*, 2008: the blocking and packing scheme behind OpenBLAS and BLIS.
- R. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, chapters 5 (optimizing program performance) and 6 (the memory hierarchy).
- U. Drepper, "What Every Programmer Should Know About Memory", 2007.
- Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*: latencies and throughputs of FMA and load units.
- The BLIS project's documentation of its microkernel ("BLIS: A Framework for Rapidly Instantiating BLAS Functionality", Van Zee and van de Geijn, *ACM TOMS*, 2015).
- tinygrad's `Opt` / `OptOps` kernel optimizations and its `BEAM` search (`beam_search`, in `tinygrad/engine/search.py` in recent versions): the production version of `Opts` and `search_opt`.

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

```
uopc/opt.py · 163 lines

examples/bench_gemm.py · 59 lines

tests/test_w06_opt.py · 72 lines
```

11. Fusion, recompute, and flash attention

Every transformer layer computes `softmax(Q K^T / sqrt(D)) V`. Written the obvious way, it builds an `N x N` matrix of scores, where `N` is the sequence length: 4 MB at `N = 1024`, 64 MB at 4096, a gigabyte at 16,384. That matrix is pure intermediate: nobody wants it, it is consumed as soon as it is produced, and moving it to memory and back is most of the work. Chapter 10 asked how to make one kernel fast. This chapter asks the question one level up: *where should the kernel boundaries be?* Cutting less (fusing more) saves memory traffic, but a fused consumer may have to recompute what it no longer reads. We will implement attention three ways in `uopc`, measure all three, take apart a bug that made fused attention cubic, and derive the trick that makes attention fusable in a single pass: the **online softmax** at the heart of FlashAttention.

What you'll build

- `examples/flash_attention.py`: unfused attention (the Tensor code from Chapter 9), fused-with-recompute attention (the same code with `fuse_reduce=True`), and a hand-written FlashAttention-style kernel in the week-3 loop IR.
- An understanding of `fuse_reduce`, `stage_broadcast_compute` and accumulator placement good enough to predict, from the shape graph, how many times a kernel will recompute `Q K^T`.

Where this fits. This chapter uses the kernelizer from Chapter 7, the reduce lowering from Chapter 6 and the opts from Chapter 10. Chapter 13 supplies the GPU machinery (shared memory, tensor cores) that real FlashAttention kernels are built from, and Chapter 14 puts attention inside a GPT.

11.1 Fusion is a trade

Chapter 7 cut the graph into kernels at *materialization points*: the roots, every STAGE, and every REDUCE. Everything between two points fuses into one loop nest:

`uopc/kernelize.py · _materialize_points`

```
def _materialize_points(roots: list[UOp], fuse_reduce: bool) -> list[UOp]:
  topo = UOp(Ops.SINK, tuple(roots)).toposort()
  m = []
  for u in topo:
    if u.op is Ops.SINK: continue
    if u.op is Ops.STAGE or (u.op is Ops.REDUCE and not fuse_reduce) or (u in roots and not
is_contiguous_buffer(u)):
      m.append(u)
  return m
```

Each cut has a price and a benefit. The price of a cut is memory traffic: the producer writes its result to a buffer and the consumer reads it back, `2 * size` bytes plus a kernel launch. The benefit is that the value is computed once. Fusing across the cut removes the traffic, but if the consumer reads each producer value more than once (because of a broadcast), the fused consumer recomputes it every time.

So the decision for a candidate fusion is roughly

`fuse if (bytes saved) / (memory bandwidth) > (FLOPs recomputed) / (compute rate)`

and both sides can be large. In Chapter 9's softmax, fusing recomputes one `exp2f` per element in two kernels instead of storing the exponentials, a clear win. NOTES 15 records the other extreme: in a GPT, a LayerNorm output EXPANDed into a matmul was recomputed for every output column, and one training step spent 374 ms in kernels until `stage_broadcast_compute` started staging such tensors (125 ms; see Chapter 7, Section 7.10):

`uopc/kernelize.py · stage_broadcast_compute`

```
def stage_broadcast_compute(roots: list[UOp], factor: int = 4) -> list[UOp]:
  """Fusing is only free if nothing gets recomputed. A computed tensor that is EXPANDed by
  `factor` or more (e.g. a LayerNorm output feeding a matmul) would be recomputed for every
  copy, so we materialize it first. This is the fusion/recompute trade-off, as one rule."""
  from uopc.uop import MOVEMENT, ALU
  todo: set[UOp] = set()
  for u in UOp(Ops.SINK, tuple(roots)).toposort():
    if u.op is not Ops.EXPAND or prod(u.shape) < factor * prod(u.src[0].shape): continue
    base = u.src[0]
    while base.op in MOVEMENT: base = base.src[0]
    if base.op in ALU and base.op is not Ops.CAST and base.shape and _fused_work(base) >= 4096:
todo.add(base)
  return UOp.wrap(roots, todo, lambda n: UOp(Ops.STAGE, (n,))) if todo else roots
```

`uopc/kernelize.py · _fused_work`

```python
def _fused_work(u: UOp) -> int:
  """ALU ops * elements that recomputing u would cost (stopping at materialized nodes)."""
  from uopc.uop import ALU
  n, seen, stack = 0, set(), [u]
  while stack:
    x = stack.pop()
    if x in seen or x.op in (Ops.BUFFER, Ops.REDUCE, Ops.STAGE): continue
    seen.add(x)
    if x.op in ALU: n += 1
    stack.extend(x.src)
  return n * prod(u.shape)
```

The rule is a crude cost model: stage a computed tensor if it is broadcast by a factor of 4 or more *and* recomputing it would cost at least 4096 ALU-op-elements. Remember this function: the first version of fused attention ran straight into it (11.4).

REDUCE is the interesting boundary. A REDUCE consumed through an EXPAND (`x - x.max(-1, keepdim=True)`) must be materialized, or the reduction would be recomputed for every element it is broadcast to. But with `fuse_reduce=True`, `_materialize_points` no longer cuts at REDUCEs: rangeify then lowers a reduce *inside* the consumer's loops, and each output position recomputes the reduction it needs. That is exactly the "fused attention" we want, provided the recomputation happens once per *row*, not once per *element*. Whether it does is decided by accumulator placement, as section 11.5 will show the hard way.

11.2 Attention, and why it is memory-bound

Single-head attention over a sequence of `N` positions with head dimension `D`:

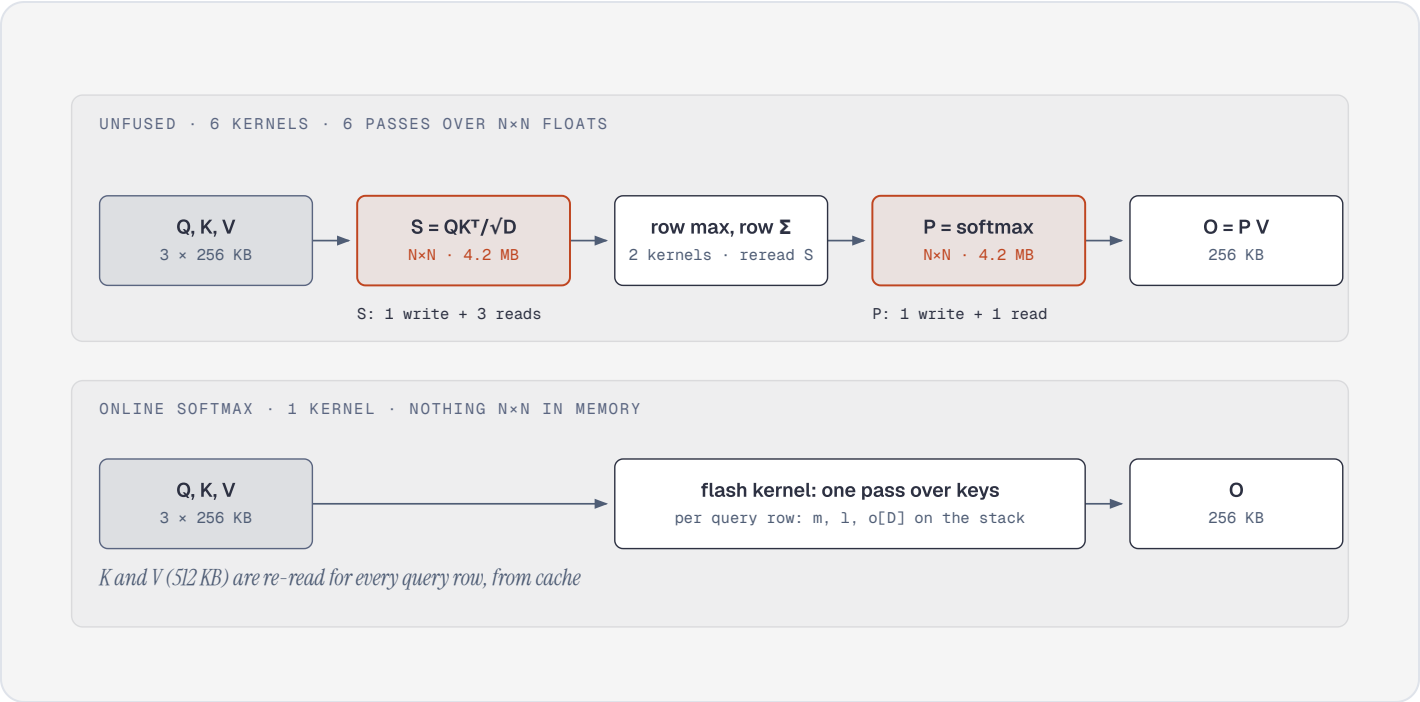
- `S = Q K^T / sqrt(D)`: scores, `(N, N)`. `S[i, j]` is how much query `i` attends to key `j`.
- `P = softmax(S)` along each row: `(N, N)`, each row a probability distribution.
- `O = P V`: `(N, D)`, each output row a weighted average of the value rows.

The FLOPs are two matmuls, `2 N^2 D` each, `4 N^2 D` in total (268 million at `N = 1024, D = 64`), plus `O(N^2)` for the softmax. The memory is dominated by `S` and `P`: `N^2` floats each, 4.2 MB at `N = 1024` and 16.8 MB at 2048, against 256 KB for each of `Q`, `K`, `V`, `O`. On the roofline of Chapter 10, the matmuls have intensity `O(D)` and the softmax has intensity `O(1)`; the more `N^2` intermediates we write and re-read, the further left the whole computation slides.

The Tensor code is one line:

examples/flash_attention.py · attention

```
def attention(q: Tensor, k: Tensor, v: Tensor) -> Tensor:
  return (q @ k.T * (1 / math.sqrt(q.shape[-1]))).softmax(-1) @ v
```



11.3 Implementation 1: unfused

Kernelized normally, the one-liner becomes six kernels. Here are the first and the fifth `realize` of the same attention in one process (`DEBUG=2`, `N = 1024`, `D = 64`):

```
  r_1024_1024                       5.380 ms    24.95 GFLOP/s
  r_1024                            0.786 ms     2.67 GFLOP/s
  r_1024                            1.807 ms     3.48 GFLOP/s
  e_1024_1024                       3.174 ms     2.31 GFLOP/s
  e_1_64_1024                       0.370 ms     0.00 GFLOP/s
  r_1024_64                         1.694 ms    79.22 GFLOP/s
realize: 6 kernels, schedule 139.8 ms, run 13.6 ms
...
  r_1024_1024                       1.980 ms    67.77 GFLOP/s
  r_1024                            0.903 ms     2.32 GFLOP/s
  r_1024                            2.288 ms     2.75 GFLOP/s
  e_1024_1024                       1.869 ms     3.93 GFLOP/s
  e_1_64_1024                       0.283 ms     0.00 GFLOP/s
  r_1024_64                         1.503 ms    89.30 GFLOP/s
realize: 6 kernels, schedule 0.4 ms, run 9.1 ms
```

In order: `S = Q K^T` (a tiled GEMM, the same 4x4 tile as in Chapter 10; the layout pass found `K` already contiguous along `D`, so no copy), the row max of `S` scaled by `0.125`, the row sum of exponentials, `P` itself, a transposed copy of `V` (the layout pass at work, because `P V` reduces over `j` and `V` is `(N, D)`), and `O = P V`. The max and sum kernels each re-read all of `S`:

```
void r_1024(float* restrict data0, float* restrict data1) {
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 256; ridx0++) {
    int alu1 = (ridx0*4);
    float acc0;
    acc0 = -INFINITY;
    float acc1;
    acc1 = -INFINITY;
    float acc2;
    acc2 = -INFINITY;
    float acc3;
    acc3 = -INFINITY;
    for (int ridx1 = 0; ridx1 < 1024; ridx1++) {
      float val2 = acc0;
      int alu3 = ((ridx0*4096)+ridx1);
      float val4 = data0[alu3];
      acc0 = ((val2>(val4*0.125f))?val2:(val4*0.125f));
      float val5 = acc1;
      float val6 = data0[(alu3+1024)];
      acc1 = ((val5>(val6*0.125f))?val5:(val6*0.125f));
      float val7 = acc2;
      float val8 = data0[(alu3+2048)];
      acc2 = ((val7>(val8*0.125f))?val7:(val8*0.125f));
      float val9 = acc3;
      float val10 = data0[(alu3+3072)];
      acc3 = ((val9>(val10*0.125f))?val9:(val10*0.125f));
    }
    float val11 = acc0;
    data1[alu1] = val11;
    float val12 = acc1;
    data1[(alu1+1)] = val12;
    float val13 = acc2;
    data1[(alu1+2)] = val13;
    float val14 = acc3;
    data1[(alu1+3)] = val14;
  }
}
```

and `P` re-reads `S` once more, recomputing the scale and the exponential, and writes the second `N x N` buffer:

```
void e_1024_1024(float* restrict data0, float* restrict data1, float* restrict data2, float*
restrict data3) {
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 1024; ridx0++) {
    float val1 = data1[ridx0];
    float val2 = data2[ridx0];
    for (int ridx1 = 0; ridx1 < 1024; ridx1++) {
      int alu3 = ((ridx0*1024)+ridx1);
      float val4 = data0[alu3];
      data3[alu3] = (exp2f((((val4*0.125f)+(val1*-1.0f))*1.4426950408889634f))*(1.0f/val2));
    }
  }
}
```

Count the passes over `N x N` data: `S` is written once and read three times, `P` is written once and read once. Six passes, 25 MB of traffic at `N = 1024`, for a computation whose inputs and output total 1 MB. And compare the first lines of the two runs: the GEMM writing `S` took 5.4 ms the first time and 2.0 ms the fifth. The difference is not computation, it is the operating system: the first write to each 4 KB page of a fresh 4 MB buffer takes a page fault. Large intermediates cost twice: once in bandwidth, once in page faults.

The first version of the runtime paid the page faults on *every* call, because `realize` allocated the plan's temporaries with `np.empty` each time; a `DEBUG=3` run then showed the `S` kernel at 9.7 ms on its first execution and 1.1 ms when re-run immediately on the same (now mapped) memory. The fix is one field on the cached `Plan`: the temporary arrays are allocated on the first run and reused by every later one.

uopc/kernelize.py · Plan

```
@dataclass
class Plan:
  kernels: list[Kernel]
  temps: list[tuple[DType, int]]
  outs: list[tuple[str, int] | tuple[DType, int]]   # per root: ("in", i) alias, or (dtype, size)
fresh
  temp_arrays: list | None = None   # allocated on first run, then reused: no page faults on
every step
```

and in `realize`:

uopc/kernelize.py:203-204

```
  if plan.temp_arrays is None: plan.temp_arrays = [np.empty(n, np_dtype(d)) for d, n in
plan.temps]
  temps = plan.temp_arrays
```

With that change the captured unfused time at `N = 1024` dropped from 28.4 ms to 9.8 ms (11.8).

11.4 Implementation 2: fused, with recompute

The second implementation changes no math, only the kernelizer's options: `fuse_reduce=True` stops cutting at REDUCEs, and a custom optimizer upcasts the last output axis and threads the first:

`examples/flash_attention.py · FlashOpt`

```
class FlashOpt:
  """fuse_reduce kernels: unroll the head dim (up to 16) so each QK^T value is shared by 16
accumulators."""
  from uopc.opt import layout_pass as layout
  @staticmethod
  def lower(body, out):
    return lower(body, out, Ctx(opts=Opts(upcast={len(body.shape) - 1: min(body.shape[-1], 16)},
thread=0)))
```

With `fuse_reduce`, the reductions of softmax are lowered *inside* the loops of their consumer. The plan shrinks from six kernels and two `N x N` temporaries to two kernels and none:

```
fuse_reduce=False: kernels ['r_1024_1024', 'r_1024', 'r_1024', 'e_1024_1024', 'e_1_64_1024',
'r_1024_64'], temps [1048576, 1024, 1024, 1048576, 65536]
fuse_reduce=True: kernels ['e_1_64_1024', 'r_1024_64'], temps [65536]
```

The only temporary left is the transposed copy of `V` (65,536 floats). Everything else happens in one kernel, `r_1024_64`, which produces `O` directly. Here is its first part:

```
void r_1024_64(float* restrict data0, float* restrict data1, float* restrict data2, float*
restrict data3) {
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 1024; ridx0++) {
    int alu1 = (ridx0*64);
    float acc2;
    acc2 = -INFINITY;
    float acc4;
    acc4 = 0.0f;
    for (int ridx4 = 0; ridx4 < 1024; ridx4++) {
      float val2 = acc2;
      float acc1;
      acc1 = 0.0f;
      for (int ridx5 = 0; ridx5 < 64; ridx5++) {
        float val3 = acc1;
        float val4 = data0[(alu1+ridx5)];
        float val5 = data1[((ridx4*64)+ridx5)];
        acc1 = (val3+(val4*val5));
      }
      float val6 = acc1;
      acc2 = ((val2>(val6*0.125f))?val2:(val6*0.125f));
    }
    float val7 = acc2;
    float alu8 = (val7*-1.0f);
    for (int ridx6 = 0; ridx6 < 1024; ridx6++) {
      float val9 = acc4;
      float acc3;
      acc3 = 0.0f;
      for (int ridx7 = 0; ridx7 < 64; ridx7++) {
        float val10 = acc3;
        float val11 = data0[(alu1+ridx7)];
        float val12 = data1[((ridx6*64)+ridx7)];
        acc3 = (val10+(val11*val12));
      }
      float val13 = acc3;
      acc4 = (val9+exp2f((((val13*0.125f)+alu8)*1.4426950408889634f)));
    }
    float val14 = acc4;
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      int alu15 = (ridx1*16);
      float acc5;
      acc5 = 0.0f;
      ...
      float acc20;
      acc20 = 0.0f;
      int alu16 = (alu15+alu1);
      for (int ridx2 = 0; ridx2 < 1024; ridx2++) {
        float val17 = acc5;
        float acc0;
        acc0 = 0.0f;
        int alu18 = ((ridx1*16384)+ridx2);
        float val19 = data2[alu18];
```

```
...
for (int ridx3 = 0; ridx3 < 64; ridx3++) {
  float val50 = acc0;
  float val51 = data0[(alu1+ridx3)];
  float val52 = data1[((ridx2*64)+ridx3)];
  acc0 = (val50+(val51*val52));
}
float val53 = acc0;
float alu54 = (exp2f((((val53*0.125f)+alu8)*1.4426950408889634f))*(1.0f/val14));
acc5 = (val17+(alu54*val19));
acc6 = (val20+(alu54*val21));
...
```

(the `...` elide the other fourteen accumulators; the whole kernel is 172 lines). This is a *nested* reduction: an inner reduce (the dot product over `e < 64`) inside an outer reduce (max, sum, or the `P V` sum over keys), inside the row loop. Rangeify produced it with no new code: the reduce lowering simply recursed. Per query row `ridx0` there is a max pass over keys (`ridx4`), a sum pass (`ridx6`), and then the output loop `ridx1 < 4`, each iteration of which makes *another* full pass over the keys (`ridx2`), recomputing every dot product and every exponential and feeding sixteen output accumulators. `FlashOpt` upcasts the head dimension by `min(D, 16) = 16`, so the 64 output columns come out in four groups of sixteen, and each group pays for its own pass. That is six full computations of `q_i . k_j` per row instead of one.

The first version of this experiment did not look like this. `stage_broadcast_compute` (11.1) saw `P` EXPANDed by a factor of 64 into the `P V` matmul, measured its recompute cost, and *staged* it, exactly as designed for LayerNorm. The "fused" plan therefore still wrote one `N x N` buffer (`P`), while the script printed "(no N*N buffer, QK^T recomputed)". Two cost-model rules disagreed: `fuse_reduce` meant "never write the big intermediate", and the broadcast rule meant "never recompute an expensive broadcast". The fix makes the user's explicit request win: with `fuse_reduce=True`, `make_plan` skips `stage_broadcast_compute`:

uopc/kernelize.py:121-125

```
roots = [graph_rewrite(r, strip_detach) for r in roots]
# fuse_reduce asks for "never write the big intermediate", so don't stage broadcasts either
if not fuse_reduce: roots = stage_broadcast_compute(roots)
if opt is not None: roots = opt.layout(roots)
points = _materialize_points(roots, fuse_reduce)
```

and the example's docstring now describes the recomputation accurately. The lesson is about debugging, not attention: when a benchmark's description and its plan disagree, believe the plan (`make_plan(...).temps`).

So does fusing win? The captured runs (11.8) say no, by a wide margin: 28.5 ms fused against 9.8 ms unfused at `N = 1024`, and 103.8 against 35.3 ms at 2048. Fusion removed 25 MB of `N x N` traffic and added five extra computations of `Q K^T`, each done without the register tiling that makes the unfused `Q K^T` GEMM fast. On this machine the recomputation costs far more than the traffic did.

> **Key idea:** fusing a reduction into its consumer is only cheap if the reduction is computed once per *consumer row*, and if the consumer itself makes one pass. Every extra pass recomputes everything the fused reduction depends on. The fix is not a better cost model; it is a better algorithm (11.6).

11.5 The O(N^3) bug: when an index lies about its loops

Before the version above existed, fused attention had a much worse problem. At `N = 512` it took 1787 ms; after the fix described below, 7 ms (NOTES 12). This section tells the full story, because it is the cleanest example in this book of a semantic property (*which loops does this value depend on?*) being decided by a syntactic one (*which loop variables does this expression mention?*).

Recall how rangeify lowers a REDUCE (Chapter 6):

`uopc/rangeify.py · _reduce`

```
def _reduce(u: UOp, idx: tuple[UOp, ...], ctx: Ctx) -> UOp:
  rop, n = u.arg
  x = u.src[0]
  rshape = x.shape[len(x.shape) - n:]
  # unrolled copies of the same reduce (they differ only by constants in idx) share one loop
  rkey = (u, frozenset(ranges_in(*idx)))
  if (parts := ctx.reduce_ranges.get(rkey)) is None:
    parts = [_axis_parts(ctx, s, ctx.opts.unroll if k == n - 1 else 1, AxisType.REDUCE) for k, s
in enumerate(rshape)]
    ctx.reduce_ranges[rkey] = parts
  real = [r for r, _ in parts if r.op is Ops.RANGE]
  vals = [rangeify(x, idx + c, ctx) for c in _combos(parts)]
  while len(vals) > 1:                                    # unrolled reduce axis: a balanced tree
    vals = [UOp(rop, (vals[i], vals[i + 1])) if i + 1 < len(vals) else vals[i] for i in range(0,
len(vals), 2)]
  if not real: return vals[0]
  acc = ctx.alloc(u.dtype)
  # the accumulator lives inside the loops the output index uses...
  acc0 = acc.after(*ranges_in(*idx))
  init = acc0.index(0).store(UOp.const(identity(rop, u.dtype), u.dtype))
  # ...is re-read every iteration of the reduce loops...
  accl = acc0.after(init, *real)
  upd = accl.index(0).store(UOp(rop, (accl.index(0).load(), vals[0])))
  # ...and is read once more after the loops end
  return acc0.after(upd.end(*real)).index(0).load()
```

The accumulator is placed with `acc.after(*ranges_in(*idx))`: it lives inside exactly the loops that the output index `idx` *mentions*. For the softmax max inside attention, the output index is the row `i`, so the accumulator sits inside the `i` loop and outside the key loops. One max per row: correct.

Now remember how rangeify handles a RESHAPE: the index is raveled into a flat offset and unraveled into the source shape. When `x` of shape `(N,)` is viewed as `(N, 1)` and expanded to `(N, N)`, the reshape asks for `x[unravel(ravel((i, j)))]`, and the first component is `((i*N + j) // N) % N`. Mathematically that is `i`: `0 <= j < N`, so `j // N = 0`. Syntactically it mentions `j`. A scratch script that prints rangeify's raw index expressions for an `(N, N)` index and what `simplify` makes of them:

```
raw ((((((0*512)+i)*512)+j)//512)%512) mentions ['i', 'j']   simplified i
raw ((((((0*512)+i)*512)+j)//1)%512) mentions ['i', 'j']   simplified j
```

If the index reaching a REDUCE is the raw form, `ranges_in` reports both `i` and `j`, and the accumulator for "max over the row" is placed inside the `j` loop of its consumer. The reduction is then recomputed for every `j`. Each recomputation is itself a pass over `N` keys with a `D`-long dot product inside, so the kernel does `O(N^2 * N * D)` work instead of `O(N * N * D)`: cubic in the sequence length.

The fix is in the RESHAPE rule of `rangeify`: simplify the new indices immediately, so that every later decision sees the truth:

uopc/rangeify.py · rangeify

```
def rangeify(u: UOp, idx: tuple[UOp, ...], ctx: Ctx) -> UOp:
  """The scalar value of u at index idx."""
  key = (u, idx)
  if (r := ctx.memo.get(key)) is not None: return r
  assert len(idx) == len(u.shape), f"{u.op}: {len(idx)} indices for shape {u.shape}"
  op = u.op
  if op is Ops.CONST: r = u
  elif op is Ops.BUFFER: r = u.index(idx[0]).load()
  elif op is Ops.STAGE: r = rangeify(u.src[0], idx, ctx)      # no kernelizer: compute it inline
  elif op in ALU: r = u.replace(src=tuple(rangeify(s, idx, ctx) for s in u.src))
  elif op is Ops.INDEX:
    iv = tuple(rangeify(i, (), ctx) for i in u.src[1:])
    r = rangeify(u.src[0], iv + idx, ctx)
  elif op is Ops.STACK:
    i, rest = idx[0], idx[1:]
    r = rangeify(u.src[-1], rest, ctx)
    for k in range(len(u.src) - 2, -1, -1):
      r = (i < k + 1).where(rangeify(u.src[k], rest, ctx), r)
  elif op is Ops.RESHAPE:
    # simplify now: ((i*N + j)//N) % N mentions j but *is* i. Dependence analysis
    # (which loops a reduce must sit in) and the memo both need the truth.
    r = rangeify(u.src[0], tuple(simplify(i) for i in unravel(ravel(idx, u.shape),
u.src[0].shape)), ctx)
  elif op is Ops.EXPAND:
    r = rangeify(u.src[0], tuple(UOp.const(0) if s == 1 and n != 1 else i for i, s, n in zip(idx,
u.src[0].shape, u.shape)), ctx)
  elif op is Ops.PERMUTE:
    inv = [0] * len(idx)
    for k, p in enumerate(u.arg): inv[p] = idx[k]
    r = rangeify(u.src[0], tuple(inv), ctx)
  elif op is Ops.SHRINK: r = rangeify(u.src[0], tuple(i + b if b else i for i, (b, _) in zip(idx,
u.arg)), ctx)
  elif op is Ops.REDUCE: r = _reduce(u, idx, ctx)
  else: raise NotImplementedError(f"rangeify: {op}")
  ctx.memo[key] = r
  return r
```

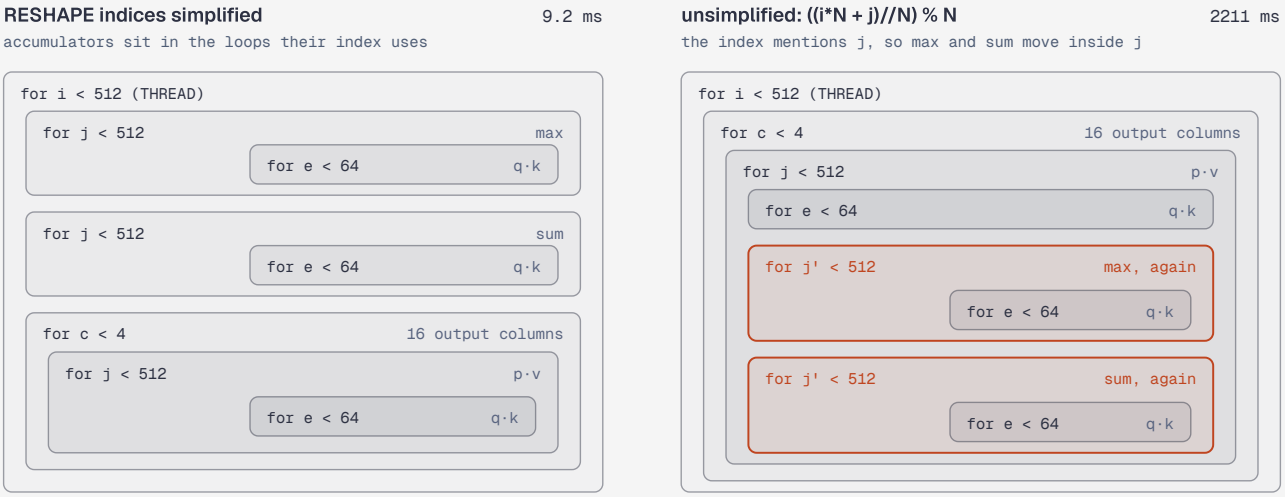
The comment on the RESHAPE branch says why: dependence analysis and the memo both need the simplified form. (The memo matters too: `rangeify` caches on `(node, idx)`, and two spellings of the same index are two different cache keys, so unsimplified indices also defeat memoization.)

We can watch the bug come back. A scratch script patches `uopc.rangeify.simplify` so that it does nothing when called from the RESHAPE rule (and works normally everywhere else, including the final cleanup in

`lower`), runs the fused attention of 11.4, and prints the loops and accumulator initializations of the attention kernel at `N = 512`:

```
  for (int ridx0 = 0; ridx0 < 512; ridx0++) {
    acc2 = -INFINITY;
    acc4 = 0.0f;
    for (int ridx4 = 0; ridx4 < 512; ridx4++) {
      acc1 = 0.0f;
      for (int ridx5 = 0; ridx5 < 64; ridx5++) {
    for (int ridx6 = 0; ridx6 < 512; ridx6++) {
      acc3 = 0.0f;
      for (int ridx7 = 0; ridx7 < 64; ridx7++) {
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      acc5 = 0.0f;
      ...
      acc20 = 0.0f;
      for (int ridx2 = 0; ridx2 < 512; ridx2++) {
        acc0 = 0.0f;
        for (int ridx3 = 0; ridx3 < 64; ridx3++) {
N=512 reshape-simplify on: 9.2 ms  err 3.4e-07
  for (int ridx0 = 0; ridx0 < 512; ridx0++) {
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      acc7 = 0.0f;
      ...
      acc22 = 0.0f;
      for (int ridx2 = 0; ridx2 < 512; ridx2++) {
        acc0 = 0.0f;
        acc2 = -INFINITY;
        acc5 = -INFINITY;
        acc6 = 0.0f;
        for (int ridx3 = 0; ridx3 < 64; ridx3++) {
        for (int ridx4 = 0; ridx4 < 512; ridx4++) {
          acc1 = 0.0f;
          acc4 = 0.0f;
          for (int ridx5 = 0; ridx5 < 64; ridx5++) {
        for (int ridx6 = 0; ridx6 < 512; ridx6++) {
          acc3 = 0.0f;
          for (int ridx7 = 0; ridx7 < 64; ridx7++) {
N=512 reshape-simplify off: 2210.7 ms  err 3.4e-07
```

(lines listing `acc6` to `acc19` and `acc8` to `acc21` elided). With simplification, the row max (`acc2 = -INFINITY`) and row sum (`acc4 = 0.0f`) are initialized once per row `ridx0`, before their passes over keys. Without it, they are initialized inside `ridx2`, the key loop of the `P V` product, so both passes over all keys run again for every key. Same graph, same answer to seven digits, 240 times slower (NOTES 12 recorded 250x):



same graph, same answer (max error 3.4e-07 both ways): one unsimplified index moved two loops inward

Two lessons. First, a compiler's analyses are only as good as the canonical form they see: every "which variables does this depend on" question must be asked of a simplified expression, or it is answered about the spelling instead of the value. Second, this bug produced *correct output*. No test comparing values against NumPy could catch it. Only a performance test (or reading the generated C) could. That is why the course grades performance, and why it pays to assert loop structure in tests, as `test_upcast_shares_reduce_loop` does (see Appendix B on testing).

11.6 Online softmax

The fused kernel needed at least three passes over the keys per row (six with `FlashOpt`) because softmax seems to need its row max before it can compute any exponential, and its row sum before it can normalize. Online softmax (Milakov and Gimelshein, 2018) removes both dependencies, and FlashAttention (Dao et al., 2022) applies it to attention.

Fix one query row. Write `s_j` for its scores (`j = 1..N`), `v_j` for the value rows. The goal is

`o = sum_j softmax(s)_j v_j = (sum_j exp(s_j - m) v_j) / (sum_j exp(s_j - m))`, with `m = max_j s_j`.

(Subtracting `m` changes nothing mathematically and keeps every exponent at most 0, as in Chapter 9.) Now define the same quantities over only the first `j` keys:

- `m_j = max(s_1, ..., s_j)`
- `l_j = sum_{k <= j} exp(s_k - m_j)`
- `o_j = sum_{k <= j} exp(s_k - m_j) v_k`

so that `o_N / l_N` is the answer. Can we get from step `j - 1` to step `j` without revisiting old keys? The new maximum is easy: `m_j = max(m_{j-1}, s_j)`. For the sum, every old term was computed relative to the old maximum; relative to the new one it is smaller by the factor `exp(m_{j-1} - m_j)`:

`exp(s_k - m_j) = exp(s_k - m_{j-1}) * exp(m_{j-1} - m_j)`.

The factor does not depend on `k`, so it comes out of the sum:

`l_j = l_{j-1} * exp(m_{j-1} - m_j) + exp(s_j - m_j)`

and identically, since each `v_k` is just carried along,

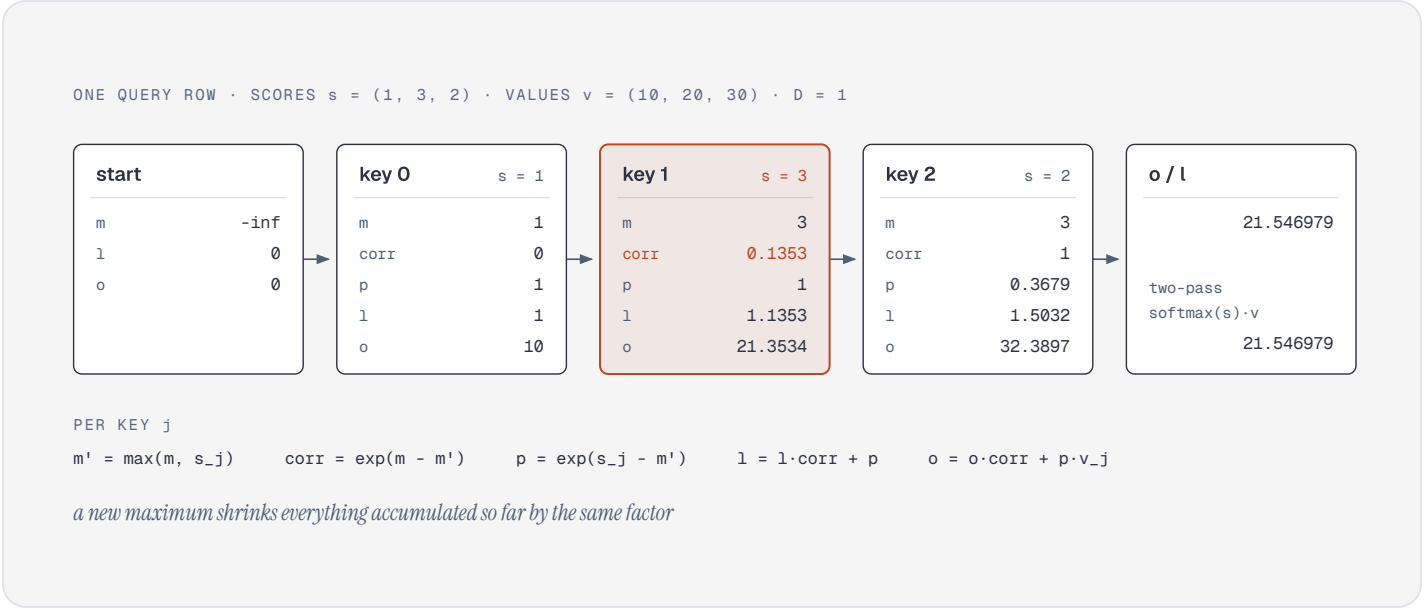
`o_j = o_{j-1} * exp(m_{j-1} - m_j) + exp(s_j - m_j) v_j`.

Start from `m_0 = -inf`, `l_0 = 0`, `o_0 = 0` (the first correction factor is `exp(-inf) = 0`, which multiplies zeros anyway). By induction, after `N` keys `l_N` and `o_N` are exactly the two sums above, and one division finishes the job. Every key is visited once; the state is two scalars and one `D`-vector per query row. When the maximum does not change, the correction factor is exactly 1; when it grows, everything accumulated so far shrinks by the same factor, which is what makes the running sums consistent.

A worked example with one query, three keys and `D = 1`, as a scratch NumPy script that prints the state after each key and compares with the two-pass answer:

```
# online softmax, one query, three keys, head dim 1: watch the running state
import numpy as np
s = np.array([1.0, 3.0, 2.0])          # scores q.k_j (already scaled)
v = np.array([10.0, 20.0, 30.0])       # values v_j
m, l, o = -np.inf, 0.0, 0.0
for j in range(3):
  m_new = max(m, s[j])
  corr = np.exp(m - m_new)             # rescale everything accumulated so far
  p = np.exp(s[j] - m_new)
  l = l * corr + p
  o = o * corr + p * v[j]
  m = m_new
  print(f"key {j}: s={s[j]:.0f}  m={m:.0f}  corr={corr:.4f}  p={p:.4f}  l={l:.4f}  o={o:.4f}")
w = np.exp(s - s.max()); w /= w.sum()
print(f"online o/l = {o / l:.6f}   two-pass softmax(s) @ v = {w @ v:.6f}")
```

```
key 0: s=1  m=1  corr=0.0000  p=1.0000  l=1.0000  o=10.0000
key 1: s=3  m=3  corr=0.1353  p=1.0000  l=1.1353  o=21.3534
key 2: s=2  m=3  corr=1.0000  p=0.3679  l=1.5032  o=32.3897
online o/l = 21.546979   two-pass softmax(s) @ v = 21.546979
```



The same algebra merges two *partial* states, as well as a state and one key. If `(m1, l1, o1)` summarizes one set of keys and `(m2, l2, o2)` another, then with `m = max(m1, m2)` the union is `(m, l1 e^{m1 - m} + l2 e^{m2 - m}, o1 e^{m1 - m} + o2 e^{m2 - m})`. That associativity is what lets GPU implementations process keys in blocks, and split long sequences across thread blocks (Exercise 11.6).

11.7 Implementation 3: online softmax in the loop IR

The Tensor API cannot express a loop-carried running max (there is no "scan" op), so the third implementation is written directly in the loop IR of Chapter 5, the way the week-3 assignment wrote GEMMs by hand:

```
examples/flash_attention.py · online_attention_ir
```

```
def online_attention_ir(N: int, D: int) -> UOp:
  """FlashAttention-style online softmax as a UOp loop program. One query row per thread."""
  f32 = dtypes.f32
  Q, K, V, OUT = (UOp(Ops.BUFFER, (), ParamArg(s, f32, (N * D,))) for s in range(4))
  alloc = lambda slot, n: UOp(Ops.ALLOC, (), ParamArg(slot, f32, (n,)))
  M, L, O, S = alloc(0, 1), alloc(1, 1), alloc(2, D), alloc(3, 1)
  scale = (1 / math.sqrt(D)) * LOG2E                    # work in base 2: exp(x) = exp2(x * log2
e)

  i = range_(N, 0, AxisType.THREAD)
  Mi, Li, Oi = M.after(i), L.after(i), O.after(i)
  init_m = Mi.index(0).store(UOp.const(-math.inf))
  init_l = Li.index(0).store(UOp.const(0.0))
  d0 = range_(D, 1)
  init_o = Oi.index(d0).store(UOp.const(0.0)).end(d0)

  j = range_(N, 2)
  # s = (q_i . k_j) * scale
  Sj = S.after(i, j)
  init_s = Sj.index(0).store(UOp.const(0.0))
  e = range_(D, 3)
  Se = Sj.after(init_s, e)
  dot = Se.index(0).store(Se.index(0).load() + Q.index(i * D + e).load() * K.index(j * D +
e).load()).end(e)
  s = Sj.after(dot).index(0).load() * scale
  # online softmax update
  Mj, Lj = Mi.after(init_m, j), Li.after(init_l, j)
  m_old = Mj.index(0).load()
  m_new = m_old.maximum(s)
  corr = UOp(Ops.EXP2, (m_old - m_new,))                # rescale what we accumulated so far
  p = UOp(Ops.EXP2, (s - m_new,))
  st_l = Lj.index(0).store(Lj.index(0).load() * corr + p)
  d = range_(D, 4)
  Od = Oi.after(init_o, j)
  st_o = Od.index(d).store(Od.index(d).load() * corr + p * V.index(j * D + d).load()).end(d)
  st_m = Mj.index(0).store(m_new)
  loop_j = UOp(Ops.SINK, (st_l, st_o, st_m)).end(j)
  # normalize
  d2 = range_(D, 5)
  out = OUT.index(i * D + d2).store(Oi.after(loop_j).index(d2).load() *
Li.after(loop_j).index(0).load().recip())
  return sink(out.end(d2, i))
```

Walk through it with the online-softmax recurrence in mind.

- **State.** Four ALLOCs: `M` and `L` (one float each: the running max and sum), `O` (`D` floats: the running output row) and `S` (one float: the dot-product accumulator). They are call-scope storage, rendered as local variables.

- **The row loop.** `i` is a THREAD range: each OpenMP thread owns whole query rows. `Mi = M.after(i)` says "the M that lives inside loop `i`", which places the declaration inside the parallel loop: each thread gets private state (NOTES 9).
- **Initialization.** `init_m` stores `-inf`, `init_l` stores 0, `init_o` zeroes the `D` floats in a small loop `d0`.
- **The key loop `j`.** First the dot product `q_i . k_j`, a reduction over `e` written exactly like the week-3 GEMM: initialize `S`, accumulate in loop `e`, `END` it, read it after the loop. It is scaled by `scale = (1/sqrt(D)) * log2(e)`, because the IR's exponential is `EXP2`: `exp(x) = exp2(x * log2 e)`, so folding `log2 e` into the scale makes every later exponential a bare `exp2`.
- **The update.** `m_new = max(m_old, s)`, `corr = exp2(m_old - m_new)`, `p = exp2(s - m_new)`, then `l = l * corr + p` and, in loop `d`, `o[d] = o[d] * corr + p * v[j, d]`. Finally `m = m_new`. `Mj = Mi.after(init_m, j)` is the AFTER-edge idiom from Chapter 4: "read M inside loop `j`, after its initialization".
- **Closing the key loop.** `UOp(Ops.SINK, (st_l, st_o, st_m)).end(j)` closes `j` over all three stores at once; it is the same SINK-under-END shape that `merge_ends` builds for register tiles.
- **Normalization.** After the key loop, `out[i, d] = o[d] / l` in a last loop `d2`.

All the ordering subtleties ("store the new max only after the old one has been used") are carried by data dependencies: `st_m` stores `m_new`, which is computed from `m_old`, which was loaded once; `corr` and `p` are values, not memory reads. The linearizer (Chapter 5) turns this into:

```
void flash(float* restrict data0, float* restrict data1, float* restrict data2, float* restrict
data3) {
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 1024; ridx0++) {
    int alu1 = (ridx0*64);
    float acc2[64];
    float acc1;
    acc1 = 0.0f;
    float acc0;
    acc0 = -INFINITY;
    for (int ridx1 = 0; ridx1 < 64; ridx1++) {
      acc2[ridx1] = 0.0f;
    }
    for (int ridx2 = 0; ridx2 < 1024; ridx2++) {
      float val2 = acc1;
      float val3 = acc0;
      float acc3;
      acc3 = 0.0f;
      int alu4 = (ridx2*64);
      for (int ridx3 = 0; ridx3 < 64; ridx3++) {
        float val5 = acc3;
        float val6 = data0[(alu1+ridx3)];
        float val7 = data1[(alu4+ridx3)];
        acc3 = (val5+(val6*val7));
      }
      float val8 = acc3;
      float alu9 = (val8*0.18033688011112042f);
      float alu10 = ((val3>alu9)?val3:alu9);
      float alu11 = (alu10*-1.0f);
      float alu12 = exp2f((val3+alu11));
      float alu13 = exp2f((alu9+alu11));
      acc1 = ((val2*alu12)+alu13);
      acc0 = alu10;
      for (int ridx4 = 0; ridx4 < 64; ridx4++) {
        float val14 = acc2[ridx4];
        float val15 = data2[(alu4+ridx4)];
        acc2[ridx4] = ((val14*alu12)+(alu13*val15));
      }
    }
    float val16 = acc1;
    for (int ridx5 = 0; ridx5 < 64; ridx5++) {
      float val17 = acc2[ridx5];
      data3[(alu1+ridx5)] = (val17*(1.0f/val16));
    }
  }
}
```

`0.18033688011112042f` is `log2(e) / 8`. `acc2[64]` is the running output row, `acc1` the running sum, `acc0` the running max, and `acc3` the dot product; every one of them is declared inside the parallel loop.

Each query row reads all of `K` and `V` (256 KB each), which stay in the 1.25 MB L2 cache of a core across rows, and nothing `N x N` exists anywhere.

11.8 Results

The captured runs (`book/outputs/flash_1024.txt` and `flash_2048.txt`, `D = 64`; each time is the best of three after a warm-up, and includes building the graph and copying `Q`, `K`, `V` into tensors):

```
unfused      9.82 ms  err 3.3e-07   (materializes 4.2 MB of scores)
fused       28.48 ms  err 3.6e-07   (no N*N buffer, QK^T recomputed)
online      10.00 ms  err 4.0e-07   (one pass, running max/sum)
```

```
unfused     35.28 ms  err 3.0e-07   (materializes 16.8 MB of scores)
fused      103.82 ms  err 3.9e-07   (no N*N buffer, QK^T recomputed)
online      33.46 ms  err 3.6e-07   (one pass, running max/sum)
```

Three stories are in these six numbers. The fully fused kernel is the slowest by a factor of about three at both sizes: six unblocked computations of `Q K^T` per row cost more than 25 MB (or 100 MB at 2048) of traffic. The unfused version is now competitive with the online kernel, because its `Q K^T` and `P V` are register-tiled, threaded GEMMs and, since temporaries are cached, it no longer pays page faults on every call. (With the first version of the runtime, the captured unfused times were 28.4 and 35.8 ms.) And the online kernel, which does the least work and moves the least memory, only ties: at `N = 1024` the two are within noise, at 2048 online is 5% ahead. Its `N x N`-free memory behaviour matters more as `N` grows and as memory gets scarcer, which is why it wins decisively on GPUs, but on this CPU at these sizes its arithmetic is simply less efficient.

Do not overlook how far the online kernel is from the roofline. It performs 268 MFLOP at `N = 1024` in about 10 ms: 27 GFLOP/s, while the unfused version's register-tiled `Q K^T` GEMM ran at 68 GFLOP/s in the noisy run of 11.3 and over 120 GFLOP/s in quieter ones. The reason is everything Chapter 10 taught: each thread handles one query row, so every `k_j` and `v_j` loaded from cache is used for exactly one query (no register tile across queries), and the dot product is a single dependency chain. The natural next step is to process a *block* of queries per thread, so that each loaded key and value row serves the whole block: the register tiling of Chapter 10 applied to attention (Exercise 11.5). That step is precisely what FlashAttention does on a GPU.

11.9 How FlashAttention tiles on a GPU

A GPU (Chapter 12) has far more arithmetic per byte of memory bandwidth than this CPU, so an `N x N` intermediate in global memory (HBM) is even more ruinous there. It also has a small, fast, programmer-managed memory per thread block, *shared memory* (SRAM), covered in Chapter 13. FlashAttention (Dao et al., 2022) and FlashAttention-2 (Dao, 2023) organize the online softmax around it:

1. **Grid.** One thread block per (batch, head, block of `B_r` query rows). FlashAttention-2 made the query-block dimension the parallel one, so long sequences fill the GPU even at batch size 1.
2. **Load a query block.** The block's `B_r x D` slice of `Q` is loaded once and kept on chip.
3. **Loop over key blocks.** For each block of `B_c` keys: load `K_j` and `V_j` (`B_c x D` each) into shared memory; compute the score tile `S = Q_i K_j^T` (`B_r x B_c`) with tensor-core matrix multiplies; update each row's running max `m` and sum `l` from the tile; rescale the output accumulator by `exp(m_old - m_new)` and add `exp(S - m_new) V_j`.
4. **Finish.** Divide the accumulator by `l` and write the `B_r x D` output block. FlashAttention-2 keeps the accumulator unnormalized until this final step (as our kernel does), which saves a division per key block. It also stores each row's log-sum-exp `m + log(l)`.
5. **Backward.** The backward pass needs `P`, which was never stored. It *recomputes* `S` and `P` block by block from `Q`, `K` and the saved log-sum-exp. Recomputing is cheaper than reading an `N x N` matrix from HBM: the fusion trade of 11.1, decided in favour of FLOPs.

The structure is the online softmax of 11.6 with the "block of queries" fix of 11.8 and the shared-memory tiling of a GEMM. Everything `N x N` lives only as a `B_r x B_c` tile in registers. Memory traffic is `O(N D)` per head instead of `O(N^2)`, and the remaining cost is two matmuls per tile, which tensor cores execute near peak. In `uopc` terms, the score tile is a WMMA-lowered GEMM (Chapter 13) inside the key loop of `online_attention_ir`, with `m`, `l` and the `O` tile held in LOCAL ALLOCs. Writing it is a fine final project.

Pitfalls

Unsimplified indices change the loop structure (NOTES 12). An index that mentions a loop variable it does not depend on (`((i*N + j)//N) % N`) makes the reduce lowering place an accumulator inside that loop. Fused attention became cubic: 9.2 ms became 2211 ms at `N = 512` in the reproduction above, with

identical output. Simplify indices before any analysis that asks which loops an expression uses, and test performance as well as values.

A fused kernel's passes multiply its recompute. With `D = 64` and `FlashOpt`'s upcast of 16, the fully fused kernel makes four output passes plus two softmax passes, each recomputing `Q K^T` for the row, and loses to the unfused version by a factor of three. Before fusing a reduction into a consumer, count how many times the consumer's loops will run the fused reduction.

* *The label said "no NN buffer".* In the first version it was wrong: `stage_broadcast_compute` staged `P` because it is broadcast into `P V`, so "fused" attention still wrote one `N x N` buffer. Cost-model rules interact. The fix lets `fuse_reduce=True` skip the broadcast-staging rule (11.4); the general habit is to believe the plan (`make_plan(...).temps`) when it disagrees with a benchmark's description.

Fresh temporaries page-fault. The first version allocated the plan's temporaries with `np.empty` on every `realize`, so every call paid a page fault for each 4 KB page it wrote: the `S` GEMM took 9.7 ms on its first execution and 1.1 ms when re-run on the same memory. Caching the arrays on the `Plan` (11.3) cut the captured unfused time from 28.4 to 9.8 ms. When a kernel that writes a large buffer looks slow, run `DEBUG=3` and compare the "again" column before optimizing its loops.

Online softmax and fully masked rows. With masking (causal attention sets future scores to `-inf`), a row whose scores are all `-inf` gives `m_old = m_new = -inf` and `corr = exp2(-inf - (-inf)) = NaN`. Real implementations special-case `m_new = -inf` (or use a large finite negative instead of `-inf`). Causal attention never has an empty row, but padding masks can.

Forgetting to rescale `o`. Rescaling only `l` when the maximum grows produces outputs that are subtly wrong (too large weights on early keys) and pass a sloppy test where scores are small and the maximum rarely changes. Test with scores whose maximum appears late in the row, as in the worked example.

Per-thread state must be declared inside the parallel loop (NOTES 9). The running max, sum and output row of `online_attention_ir` are ALLOCs placed by `.after(i)` inside the THREAD loop. Declared outside, all threads would share one `acc2[64]` and the output would be a race.

Exercises

1. ★ **Size the matrices.** How large are `S` and `P` for `N = 8192` in `f32`, per head? A GPT-style model with 12 heads, batch 8: how much memory would unfused attention need for one layer's `S`?

2. ★ **Count the passes.** Predict the number of full passes over the keys (each with a `D = 64` dot product) that the fully fused kernel of 11.4 makes per query row with `FlashOpt`'s upcast set to 8, 16, 32 and 64. Which one needs the most registers?
3. ★ **Merge two states.** Prove the merge rule for two partial states `(m1, l1, o1)` and `(m2, l2, o2)` from 11.6, and check it numerically by splitting the worked example's keys into `{0}` and `{1, 2}`.
4. ★★ **Causal mask.** Add a causal mask to `online_attention_ir`: query `i` may only attend to keys `j <= i`. Try two ways: mask the scores with a WHERE (what value must a masked score get, given the NaN pitfall?), and make the key loop's end depend on `i` (does the IR, the linearizer and the renderer accept a RANGE whose end is an expression of another RANGE?). Verify against NumPy and compare the speed.
5. ★★ **Query blocking.** Process a block of 4 query rows per iteration of the THREAD loop in `online_attention_ir`, so that each loaded `k_j` and `v_j` serves four queries. Measure at `N = 1024` and `2048`, and compute the new GFLOP/s.
6. ★★ **Flash-decoding.** For `N_q = 1` query and `N = 65536` keys, the row loop has one iteration and one thread does everything. Split the keys into `T` chunks processed by different threads, each producing a partial `(m, l, o)`, and merge them with Exercise 3's rule in a second kernel.
7. ★★ **Upcast the head.** Give `FlashOpt` an upcast of the full head dimension (`D = 64`), so the fused kernel makes three passes instead of six. Is it faster? Then try the other choice the first version made by accident: run with `fuse_reduce=True` but call `stage_broadcast_compute` anyway, so `P` is stored. Rank the three variants and explain using register pressure and recomputation.
8. ★★ **Find the bug.** A student's online kernel computes `l = l * corr + p` and `o[d] = o[d] + p * v[j, d]`. On which inputs does it pass a test against NumPy, and on which does it fail? Write the failing test.
9. ★★★ **Backward with recompute.** Implement the backward pass of attention with recomputation: store only `O` and the per-row log-sum-exp in the forward pass, and recompute `P` block by block in the backward pass to produce `dQ`, `dK`, `dV`. Check against `gradient()` applied to the unfused Tensor graph.

Solutions to ★ exercises

1. `8192^2 * 4 = 268 MB` per head for `S` (and as much again for `P`). For 12 heads and batch 8: `96 * 268 MB = 25.8 GB` for one layer's `S`, which is why nobody materializes it at long context.

2. Two softmax passes plus `64 / upcast` output passes: 10, 6, 4 and 3 passes for upcasts of 8, 16, 32 and 64. The upcast of 64 needs the most registers: 64 output accumulators live across the whole key loop (far more than AVX2's 16 vector registers, so they spill), which is the price of making only three passes.

3. Each `l` is a sum of terms `exp(s_k - m_x)`; multiplying by `exp(m_x - m)` re-expresses every term relative to `m`, and the union's sum is the sum of the two re-expressed sums (likewise for `o`). Numerically: `{0}` gives `(1, 1, 10)`; `{1, 2}` gives `m = 3`, `l = 1 + e^{-1} = 1.3679`, `o = 20 + 30 e^{-1} = 31.0364`. Merging

with `m = 3`: `l = 1 * e^{-2} + 1.3679 = 1.5032`, `o = 10 e^{-2} + 31.0364 = 32.3897`, matching the sequential result, and `o / l = 21.547`.

Quiz

1. What are the two costs on either side of a fusion decision?
2. Why does unfused attention make six passes over `N x N` data? Name them.
3. With `fuse_reduce=True` and `FlashOpt` at `D = 64`, how many times is each `q_i . k_j` computed, and why?
4. Write the recurrences for `m_j`, `l_j` and `o_j` in online softmax.
5. Why is the correction factor `exp(m_{j-1} - m_j)` allowed to multiply the whole running sum?
6. In the O(N^3) bug, what did the unsimplified index mention, and why did that move the accumulator?
7. Why can a value-comparison test not catch the O(N^3) bug?
8. In FlashAttention's backward pass, why is recomputing `P` cheaper than storing it?

Answers

1. Memory traffic (writing and re-reading the intermediate, plus a launch) against recomputation (FLOPs the fused consumer repeats because it reads each producer value more than once).
2. `S` written by the `Q K^T` kernel and read by the max, sum and `P` kernels (four passes); `P` written by its kernel and read by `P V` (two passes).
3. Six: once in the max pass, once in the sum pass, and once in each of the four output passes, because the head dimension is upcast 16 at a time and each group of 16 output columns makes its own pass.
4. `m_j = max(m_{j-1}, s_j)`; `l_j = l_{j-1} e^{m_{j-1} - m_j} + e^{s_j - m_j}`; `o_j = o_{j-1} e^{m_{j-1} - m_j} + e^{s_j - m_j} v_j`; answer `o_N / l_N`.
5. Every term in the sum was computed relative to `m_{j-1}`; re-expressing each relative to `m_j` multiplies it by the same factor, independent of `k`, so the factor comes out of the sum.
6. `((i*N + j)//N) % N` mentions `j`; the reduce lowering places accumulators inside every loop the output index mentions, so the row max and sum were recomputed for every `j`.
7. The output is identical (the recomputed reductions give the same values); only the amount of work changes. Only timing or structural assertions on the generated code can detect it.
8. Storing `P` costs `O(N^2)` writes and reads of slow global memory; recomputing it from `Q`, `K` and the saved log-sum-exp costs `O(N^2 D)` FLOPs on fast tensor cores, block by block, entirely on chip.

Further reading

- A. Vaswani et al., "Attention Is All You Need", NeurIPS 2017.
- M. Milakov and N. Gimelshein, "Online normalizer calculation for softmax", arXiv:1805.02867, 2018.
- M. Rabe and C. Staats, "Self-attention Does Not Need O(n^2) Memory", arXiv:2112.05682, 2021.
- T. Dao, D. Fu, S. Ermon, A. Rudra, C. Re, "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness", NeurIPS 2022.
- T. Dao, "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning", 2023.
- J. Shah et al., "FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision", 2024.
- The `flash-attention` repository (Dao-AILab on GitHub) for the reference CUDA implementation.

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

```
examples/flash_attention.py · 107 lines
```

12. GPU architecture and mapping kernels

> **Later hardware validation:** the original chapter's emulation and unavailable-driver notes describe its original runs. Chapter 35 now adds a separate real-device GEMM implementation and recorded RTX 3050 results, with its own correctness and timing scope.

Every kernel you have generated so far is a loop nest that one CPU core, or a dozen of them under OpenMP, walks through in order. A GPU offers a different deal. You don't write the outer loops. You describe what *one iteration* does, say how many iterations you want, and the hardware runs tens of thousands of them at once on a chip built for throughput rather than latency. This chapter teaches enough GPU architecture to see why that deal is good and where it bites: SIMT execution, warps, the memory hierarchy, occupancy, and coalescing. Then it shows how small the compiler change is. In `uopc` a GPU kernel is the same linearized loop nest as a CPU kernel, except that some ranges are tagged `GLOBAL` or `LOCAL` and render as `blockIdx` and `threadIdx` where a CPU kernel would have a `for`.

What you'll build

- Two new axis types, `GLOBAL` (one per thread block) and `LOCAL` (one per thread in a block), and the `Opts.gpu` / `Opts.local` knobs that rangeify uses to split an output axis into `(block, thread, upcast)` parts.
- `CUDARenderer`, a 30-line subclass of the C renderer that turns those ranges into `blockIdx.x` and `threadIdx.x`, plus `launch_dims`, which reads the grid and block sizes back off the IR.
- `gpu_opt_choice`, a heuristic that picks a launch configuration for any kernel.
- `emulate`, which tests every GPU kernel on the CPU by rendering it as plain C where the GPU ranges are loops.
- A CUDA runtime in about 90 lines of `ctypes`: NVRTC compiles the source, the driver API launches it.

Where this fits. Chapter 10 made CPU kernels fast with register tiles (UPCAST) and OpenMP threads (THREAD). This chapter reuses that machinery and moves it to a GPU. Chapter 13 adds shared memory, barriers and tensor cores, which are the features that make GPU GEMMs fast.

> **A note on what was measured.** The reference machine for this book (an Intel i7-13620H laptop, see `book/outputs/machine.txt`) has no working NVIDIA driver or CUDA toolkit. Every GPU kernel in this chapter is therefore checked in two ways. The *rendered CUDA source* is shown exactly as `render_cuda` prints it. The *numerical results* come from the CPU emulator, which runs the same index arithmetic.

> Hardware numbers (SM counts, bandwidths, latencies) are general knowledge taken from vendor documentation and are labeled that way. Nothing in this chapter is a GPU timing measured by the author. Section 12.11 shows how a reader who has a GPU can run the real thing.

12.1 Why GPUs: latency machines and throughput machines

A CPU core is a *latency machine*. Out-of-order execution, branch prediction, big caches and prefetchers all exist so that one thread's next instruction can run as soon as possible. In Chapter 10 we pushed one such chip to 114 GFLOP/s on a 1024^3 GEMM, using every core and AVX2 FMAs, against OpenBLAS's 132 (`book/outputs/bench_gemm_1024.txt`).

A GPU is a *throughput machine*. It spends its transistors on arithmetic units and register files instead of on making one thread fast. Here are the numbers for an NVIDIA A100 (SXM, 40 GB), from NVIDIA's A100 whitepaper. These are general knowledge, not measured here:

	i7-13620H (this book's laptop)	A100
cores / SMs	6 P + 4 E cores	108 streaming multiprocessors
peak FP32	under 1 TFLOP/s (AVX2 FMA; see Chapter 10's roofline)	19.5 TFLOP/s
peak FP16 tensor core	none	312 TFLOP/s (dense)
DRAM bandwidth	tens of GB/s (DDR5)	1555 GB/s (HBM2)
threads resident at once	16 hardware threads	108 × 2048 = 221,184

The last row is the most important one. A GPU thread is slow: it issues in order, it has no branch predictor, and a load from DRAM takes on the order of 500 cycles to return. The GPU gets its speed by keeping *hundreds of thousands* of threads resident. When one warp stalls on a load, the scheduler issues an instruction from another warp in the next cycle, at no cost. Latency is hidden by having plenty of other work, not by being clever about any single thread.

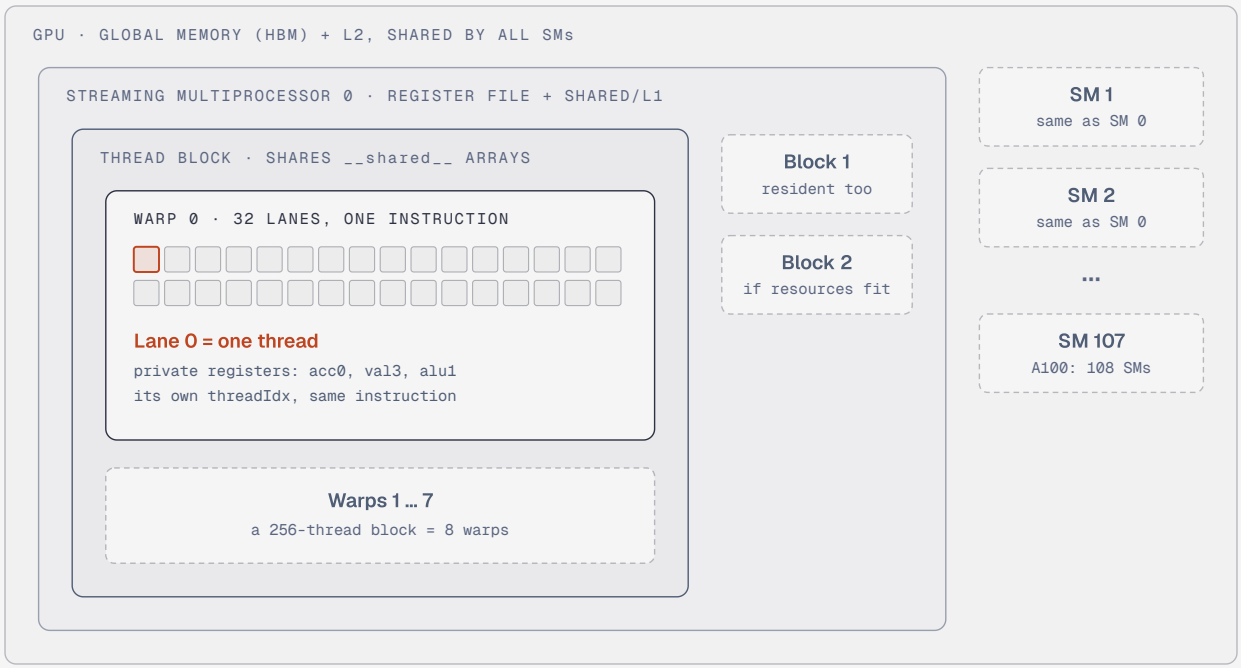
How much other work is needed? **Little's law**: the bytes in flight equal the bandwidth times the latency. At 1555 GB/s and roughly 600 ns of memory latency, $1555 \cdot 10^9 \times 600 \cdot 10^{-9} \approx 930$ KB must be in flight at all times to saturate DRAM. That is about 8.6 KB per SM, or about 2,200 outstanding 4-byte loads per SM, and an SM can hold at most 2,048 threads. So even a full SM barely keeps memory busy if every thread has one scalar load outstanding. Fast kernels also need several independent loads per thread (instruction-level parallelism, which is what UPCAST gives us) or wide vector loads. Keep this calculation in mind. It explains occupancy, upcasting, and much of the folklore about GPU performance.

The roofline from Chapter 10 still applies, with a different ridge point: 19.5 TFLOP/s ÷ 1.555 TB/s ≈ **12.5 FLOP per byte** for FP32. A kernel that does less arithmetic than that per byte of DRAM traffic is memory-bound on an A100. An elementwise add does 1 FLOP per 12 bytes. A naive GEMM that reloads both operands for every multiply-add does 2 FLOPs per 8 bytes. Both sit far below the ridge. Chapter 13 is about pushing GEMM above it.

12.2 The execution model: grids, blocks, warps, threads

CUDA's model has four levels. Learn them now. The rest of the chapter maps the compiler onto them.

- A **kernel** is a function marked `__global__`. The host *launches* it with a grid size and a block size.
- The **grid** is a 1-, 2- or 3-D array of **thread blocks**. `blockIdx.{x,y,z}` says which block you are, and `gridDim` says how many blocks there are.
- A **block** is a 1-, 2- or 3-D array of **threads**, at most 1024 in total. `threadIdx.{x,y,z}` says which thread you are inside the block. Threads in one block can share fast on-chip memory (`__shared__`) and wait for each other (`__syncthreads()`). Threads in *different* blocks can do neither.
- The hardware runs the threads of a block in groups of 32 called **warps**. A warp has one instruction stream. Each cycle the SM's scheduler picks a ready warp and issues one instruction for all of its 32 lanes. The model is SIMT, "single instruction, multiple threads": you write scalar code for one thread, and the hardware runs 32 of them in lockstep, much as one AVX instruction works on 8 floats.



A block lives on one SM for its whole life. Blocks never talk to each other except through global memory.

Blocks are scheduled onto **streaming multiprocessors** (SMs). A block is assigned to one SM when it starts and stays there until it finishes. An SM holds as many blocks at once as its resources allow (Section 12.4), and the blocks in a grid run in an order that is not specified. That is why blocks cannot synchronize with each other. Block 900 may not have started when block 0 finishes, so a barrier between them would deadlock. The only communication between blocks is through global memory, normally across a kernel boundary.

The thread index has a fixed linearization: `tid = threadIdx.x + blockDim.x * (threadIdx.y + blockDim.y * threadIdx.z)`, and warp w is threads `32w … 32w+31`, so **`.x` varies fastest**. Consecutive `threadIdx.x` values sit in the same warp and execute the same load instruction in the same cycle. That single fact decides which loop axis we map to `.x` (Section 12.5).

Divergence. All 32 lanes share one instruction stream. When lanes take different sides of a branch, the warp runs both sides one after the other, with the inactive lanes masked off. A `WHERE` in our IR renders as a `?:` select, which compiles to a predicated select instruction rather than a branch, so `uopc` kernels rarely diverge. Since Volta each thread has its own program counter ("independent thread scheduling"), but divergent paths are still serialized, so the cost model hasn't changed.

12.3 The memory hierarchy

Performance on a GPU is mostly a question of *where the data is*. The levels, with A100 numbers as a reference point (general knowledge; latencies are rough orders of magnitude):

level	scope	size (A100)	latency, roughly	in `uopc`
registers	one thread	64K × 32-bit per SM, ≤ 255 per thread	~1 cycle	`acc0`, `val3`, `alu1`
shared memory	one block	up to 164 KB per SM (carved from 192 KB L1)	~20–30 cycles	`ALLOC` with `AddrSpace.SHARED` (Ch. 13)
L1 cache	one SM	the rest of the 192 KB	~30 cycles	automatic
L2 cache	whole GPU	40 MB	~200 cycles	automatic
global (HBM)	whole GPU and host-visible	40–80 GB	~500 cycles	`BUFFER` (`data0`, ...)

Three consequences matter for the compiler.

1. **Registers are the fastest memory and they are allocated statically.** Every `float acc0;` in a kernel is a register. A 4×4 register tile costs 16 registers per thread *for every thread*. Use too many and fewer threads fit on the SM (next section). Use more than the hardware has and the compiler *spills* to "local memory", which in spite of the name is DRAM-backed and slow.
2. **Shared memory is a scratchpad managed by software.** Unlike a cache it is explicit: you load into it, you synchronize, you read from it. In exchange you get guaranteed on-chip reuse. Chapter 13 is about using it.
3. **Global memory is fast only in bulk.** DRAM moves data in 32-byte *sectors*, and a warp's 32 loads are combined into as few sectors as possible (Section 12.5).

12.4 Occupancy

Occupancy is the ratio of warps resident on an SM to the maximum it could hold (64 warps = 2048 threads on an A100). It matters because of the Little's-law calculation in 12.1: resident warps are what the scheduler can switch to while other warps wait on memory.

Three resources limit how many blocks fit on an SM. Here they are worked through for a block of 256 threads (8 warps), using A100 limits: 65,536 registers, 164 KB of shared memory, 2,048 threads and 32 blocks per SM.

per-thread registers	shared memory per block	blocks limited by regs	by smem	by threads
32	0	65536 / (256·32) = 8	–	2048 / 256 = 8
64	0	65536 / (256·64) = 4	–	8
32	48 KB	8	⌊164 / 48⌋ = 3	8
128	16 KB	65536 / (256·128) = 2	10	8

(Real allocators round registers up to granules of 256 per warp, so treat these numbers as idealized. The CUDA occupancy calculator, or `cudaOccupancyMaxActiveBlocksPerMultiprocessor`, gives exact figures.)

Each `uopc` kernel is declared `__launch_bounds__(N)`, where N is the block size. That is a promise to the compiler that the kernel never launches with more than N threads per block, so it can budget registers for exactly N. Without it, `nvcc`/NVRTC must assume 1024 threads and may restrict registers too much.

Occupancy is a means to an end. Vasily Volkov's well-known GTC 2010 talk, "Better Performance at Lower Occupancy", shows kernels that run faster at 25% occupancy because each thread keeps many independent loads and FMAs in flight. Little's law counts *bytes in flight*. Threads are one way to get them and ILP is another. This is the GPU version of the register-tile argument from Chapter 10: an upcast kernel with 4 accumulators per thread has 4 independent FMA chains, so it needs a quarter as many warps to hide the same latency.

12.5 Coalescing: how a warp talks to DRAM

When a warp executes a load, the hardware collects the 32 addresses and fetches the set of 32-byte sectors that cover them. The cost is the number of distinct sectors, not the number of threads.

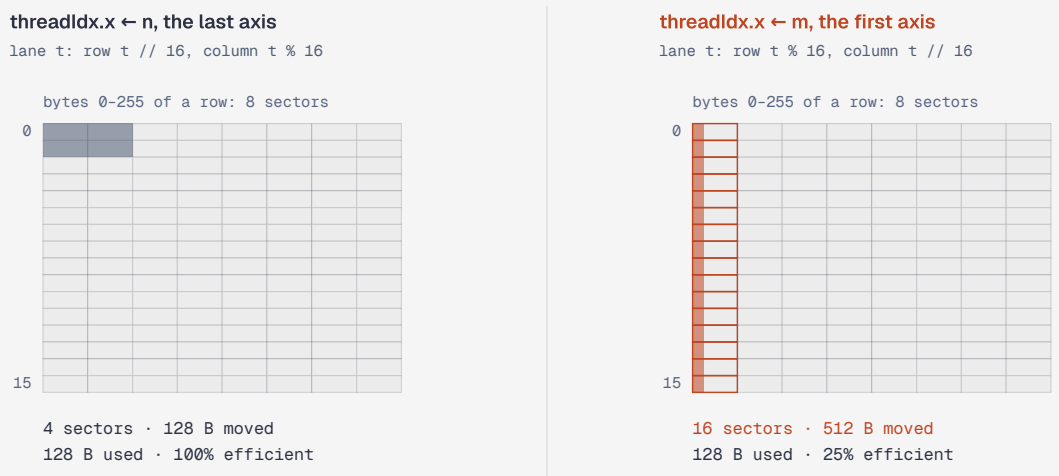
- 32 threads reading 32 consecutive floats (128 bytes, aligned) → **4 sectors**. Every byte fetched is used. This is a *coalesced* access.
- 32 threads reading floats 256 bytes apart → **32 sectors**, of which 4 bytes in 32 are used. The access costs 8× more bandwidth for the same work.
- 32 threads reading *the same* address → **1 sector**, broadcast to all lanes. That is free.

The rule for a code generator follows: **the loop axis that indexes memory contiguously must be driven by `threadIdx.x`.** We can check this on real IR. The script below lowers our usual 64×64×64 GEMM with 16×16 blocks, takes warp 0 of block (0,0), and evaluates every global `INDEX` for its 32 lanes by substituting concrete values for the ranges and calling `simplify`. That is constant folding (Chapter 3) used as an address calculator. It does so twice: once with `.x` driving the last output axis n, as `CUDARenderer` does, and once with `.x` driving the first axis m.

```python
# scratch/coalesce.py (scratch script, not part of the repo)
import sys; sys.path.insert(0, "tests")
from helpers import buf
from uopc.uop import UOp, Ops, AxisType
from uopc.rangeify import Opts
from uopc.symbolic import simplify
from uopc.gpu import gpu_lower

def warp_addresses(ast, x_axis_first_dim: bool):
  """For warp 0 of block (0,0): the byte addresses each global LOAD/STORE touches at reduce step
0."""
  rngs = sorted({u for u in ast.toposort() if u.op is Ops.RANGE}, key=lambda r: r.arg)
  loc = [r for r in rngs if r.arg.kind is AxisType.LOCAL]          # loc[0] from axis 0 (M),
loc[1] from axis 1 (N)
  xs, ys = (loc[1], loc[0]) if not x_axis_first_dim else (loc[0], loc[1])
  out = {}
  for u in ast.toposort():
    if u.op is Ops.INDEX and u.src[0].op is Ops.BUFFER:
      addrs = []
      for t in range(32):                                          # threads 0..31 = one warp
        tx, ty = t % xs.src[0].arg, t // xs.src[0].arg
        sub = {r: UOp.const(0) for r in rngs}; sub[xs] = UOp.const(tx); sub[ys] = UOp.const(ty)
        addrs.append(simplify(u.src[1].substitute(sub)).arg * 4)
      out[f"data{u.src[0].arg.slot}"] = addrs
  return out

M = N = K = 64
A, B = buf(0, M * K), buf(1, K * N)
g = (A.reshape((M, 1, K)).expand((M, N, K)) * B.reshape((K, N)).permute((1, 0)).reshape((1, N,
K)).expand((M, N, K))).reduce(Ops.ADD, 1)
ast = gpu_lower(g, buf(2, M * N), Opts(gpu=True, local={0: 16, 1: 16}))
for label, flag in [("threadIdx.x <- last axis (n)", False), ("threadIdx.x <- first axis (m)",
True)]:
  print(label)
  for name, a in warp_addresses(ast, flag).items():
    sectors = len({x // 32 for x in a})
    print(f"  {name}: first addresses {a[:4]}...  distinct={len(set(a)):2d}  32B sectors=
{sectors:2d}")
```

```
threadIdx.x <- last axis (n)
  data2: first addresses [0, 4, 8, 12]...  distinct=32  32B sectors= 4
  data0: first addresses [0, 0, 0, 0]...  distinct= 2  32B sectors= 2
  data1: first addresses [0, 4, 8, 12]...  distinct=16  32B sectors= 2
threadIdx.x <- first axis (m)
  data2: first addresses [0, 256, 512, 768]...  distinct=32  32B sectors=16
  data0: first addresses [0, 256, 512, 768]...  distinct=16  32B sectors=16
  data1: first addresses [0, 0, 0, 0]...  distinct= 2  32B sectors= 1
```

Read the store to `data2` (the output C) first. With the right mapping, warp 0 writes two rows of 16 consecutive floats into 4 sectors, the minimum possible. With the wrong mapping the same 32 stores land in 16 different sectors, a 4× increase in traffic for the store. The loads show the full picture. With the right mapping, `A[m, k]` (`data0`) is the same address for all 16 lanes of a row, a broadcast, and `B[k, n]` (`data1`) is 16 consecutive floats. With the wrong mapping these swap, and A is now read down a column, 16 sectors per warp.



Same kernel, same data. Only the loop index that the fastest-varying thread id drives has changed.

This is NOTES 23 from the development log: "CUDA's `.x` dimension is the fastest-varying across a warp: map the *last* (contiguous) axis to `.x` or loads/stores are uncoalesced." Our tensors are row-major, so the last output axis is the one whose neighbours are adjacent in memory, at least in the output. `CUDARenderer` sorts the ranges so that the highest-numbered `GLOBAL` and `LOCAL` ranges, which come from the last axis, get `.x`.

12.6 From loops to a grid: GLOBAL and LOCAL ranges

Key idea: A GPU kernel is the body of a loop nest whose outer loops the hardware runs for you. In the IR nothing changes except the *kind* of some ranges. A `GLOBAL` range of size G is "for each of G blocks", and

> a `LOCAL` range of size L is "for each of L threads in the block". The renderer prints a hardware index where a `for` would have been.

Recall `AxisType` from Chapter 10. Weeks 7–8 add the last two members:

uopc/uop.py · AxisType

```
class AxisType(Enum):
  LOOP = auto()      # a plain serial for-loop
  REDUCE = auto()    # a for-loop that a reduction accumulates over
  UPCAST = auto()    # fully unrolled (week 6). uopc unrolls *during* rangeify (Opts.upcast), so no RANGE of
                     # this kind is ever built; the name stays for the course's vocabulary
  THREAD = auto()    # parallel across CPU threads (week 6)
  GLOBAL = auto()    # GPU blockIdx (week 7)
  LOCAL = auto()     # GPU threadIdx (week 7)
  def __repr__(self): return f"AxisType.{self.name}"
  __hash__ = object.__hash__
```

The choice of how to map a kernel is an `Opts`, the same dataclass as in Chapter 10, with two new fields. `gpu=True` turns output axes into `GLOBAL` ranges, and `local` gives, per output axis, the number of threads per block along that axis:

uopc/rangeify.py · Opts

```
@dataclass
class Opts:
  """Week 6: how to map a kernel onto the machine. Chosen by uopc.opt, applied during rangeify."""
  upcast: dict[int, int] = field(default_factory=dict)   # output axis -> unroll factor (register tile)
  thread: int | None = None                              # output axis run in parallel (OpenMP)
  unroll: int = 1                                        # unroll factor of the innermost reduce axis
  tile: dict[int, int] = field(default_factory=dict)     # output axis -> cache block size (week 6, loop tiling)
  gpu: bool = False                                      # weeks 7-8: output axes become GLOBAL (blockIdx) ranges
  local: dict[int, int] = field(default_factory=dict)    # output axis -> threads per block on that axis (LOCAL)
```

Splitting one axis three ways

An output axis of size `s` becomes three nested parts: `s = blocks × local × upcast`. The index is

```
index = (gidx * local + lidx) * upcast + c        c = 0 .. upcast-1, unrolled
```

where `gidx` is a `GLOBAL` range of size `s / (local·upcast)`, `lidx` is a `LOCAL` range of size `local`, and `c` is a constant (each of the `upcast` copies of the body gets its own). That is `_gpu_axis`:

uopc/rangeify.py · _gpu_axis

```
def _gpu_axis(ctx: Ctx, size: int, local: int, factor: int) -> tuple[UOp, int]:
  """size = blocks * local * factor: index = (gidx * local + lidx) * factor + c."""
  assert size % (local * factor) == 0, f"axis {size} not divisible by local {local} * upcast
{factor}"
  g = ctx.rng(size // (local * factor), AxisType.GLOBAL) if size // (local * factor) > 1 else
UOp.const(0)
  if local == 1: return g, factor
  return g * local + ctx.rng(local, AxisType.LOCAL), factor
```

It reuses the machinery from Chapter 10. `_gpu_axis` returns the same `(base, factor)` pair that `_axis_parts` returns for CPU axes, and `_combos` turns those pairs into the unrolled index tuples. Two details matter:

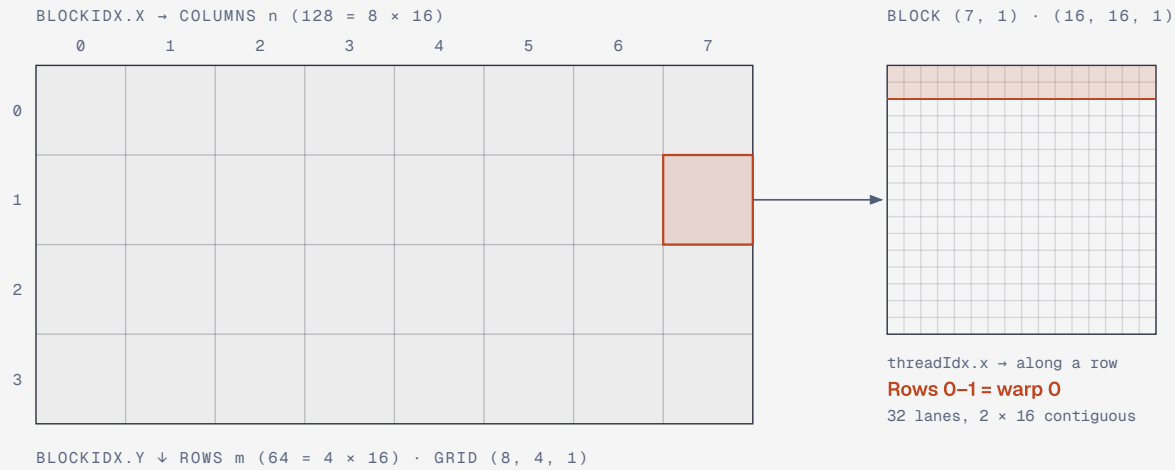
- An axis that needs only one block gets `CONST 0` instead of a range, so a 16×32 kernel run as one block has no `blockIdx` at all (you will see this in the softmax kernel in 12.9).
- The assert demands exact divisibility. Production compilers pad the grid up and guard the stores with `if (idx < n)`. We kept the simpler contract and made `gpu_opt_choice` choose divisors. Exercise 12.5 asks you to add guards.

`lower` uses `_gpu_axis` when `opts.gpu` is set. Here is the whole function again, now that its GPU branch makes sense:

uopc/rangeify.py · lower

```
def lower(root: UOp, out: UOp, ctx: Ctx | None = None) -> UOp:
  """Lower one shaped value into a SINK that stores it, element by element, into BUFFER `out`."""
  ctx = ctx or Ctx()
  o = ctx.opts
  if o.gpu:
    # each output axis gets a GLOBAL range and (optionally) a LOCAL one; neither becomes a loop
on a GPU
    gl = [_gpu_axis(ctx, s, o.local.get(k, 1), o.upcast.get(k, 1)) for k, s in
enumerate(root.shape)]
    plist = gl
  else:
    # the THREAD axis gets the lowest range index, so it is the outermost loop
    order = ([o.thread] if o.thread is not None else []) + [k for k in range(len(root.shape)) if
k != o.thread]
    tiled = {k: t for k, t in o.tile.items() if 1 < t < root.shape[k] and root.shape[k] % t == 0
and t % o.upcast.get(k, 1) == 0}
    parts: dict[int, tuple[UOp, int]] = {}
    # tiled axes: first every block loop (outer), then the loops inside a block. A block of
outputs reuses
    # the same rows of each operand, so they stay in cache while the block is computed
    outer = {k: ctx.rng(root.shape[k] // t, AxisType.THREAD if k == o.thread else AxisType.LOOP)
for k, t in tiled.items()}
    for k in order:
      f = o.upcast.get(k, 1)
      if k in tiled:
        inner = tiled[k] // f
        parts[k] = (outer[k] * inner + ctx.rng(inner) if inner > 1 else outer[k], f)
      else:
        parts[k] = _axis_parts(ctx, root.shape[k], f, AxisType.THREAD if k == o.thread else
AxisType.LOOP)
    plist = [parts[k] for k in range(len(root.shape))]
  stores = []
  for idx in _combos(plist):
    val = rangeify(root, idx, ctx)
    stores.append(out.index(ravel(idx, root.shape) if root.shape else UOp.const(0)).store(val))
  loops = ranges_in(*(b for b, _ in plist))
  body = stores[0] if len(stores) == 1 else UOp(Ops.SINK, tuple(stores))
  return simplify(merge_ends(sink(body.end(*loops))))
```

`_gpu_axis` is called once per output axis, so the range numbers *interleave*: axis 0's `GLOBAL` is range 0, axis 0's `LOCAL` is range 1, axis 1's `GLOBAL` is range 2, and so on. On a GPU the order is irrelevant, because neither kind of range is a loop. It does decide the loop order in the emulator, which you will see in 12.10. The emulated results are correct either way, since in a kernel without shared memory every order of the threads gives the same answer.




```
int gidx1 = blockIdx.y; int lidx1 = threadIdx.y; // row = gidx1*16 + lidx1
int gidx0 = blockIdx.x; int lidx0 = threadIdx.x; // column = gidx0*16 + lidx0
int alu1 = gidx1*2048 + lidx1*128 + gidx0*16 + lidx0; // row*128 + column
```

12.7 The CUDA renderer

The C renderer from Chapter 4 was written with this chapter in mind. It has a few class attributes and hook methods that a subclass can override:

uopc/render_c.py · CRenderer.render_range

```python
def render_range(self, u: UOp, name: str, end: str) -> list[str]:
  pragma = ["#pragma omp parallel for"] if u.arg.kind is AxisType.THREAD else []
  return pragma + [f"for (int {name} = 0; {name} < {end}; {name}++) {{"]
```

uopc/render_c.py · CRenderer.is_loop

```python
def is_loop(self, u: UOp) -> bool:
  """On a CPU every range is a for loop -- GPU ranges too, which makes this an emulator for GPU kernels."""
  return True
```

`is_loop` returning `True` for *every* range, GPU ranges included, is what will make the emulator free. The CUDA renderer overrides it, along with the prefixes, the range naming and the header:

uopc/gpu.py · CUDARenderer

```
class CUDARenderer(CRenderer):
  buffer_prefix = ""
  shared_prefix = "__shared__ "
  barrier = "__syncthreads()"
  restrict = "__restrict__"

  def render(self, sink: UOp, name: str = "kernel") -> str:
    rngs = [u for u in sink.toposort() if u.op is Ops.RANGE]
    # the last (fastest varying) axis gets .x, so neighbouring threads touch neighbouring memory
    self.glob = [u for u in sorted(rngs, key=lambda r: r.arg, reverse=True) if u.arg.kind is
AxisType.GLOBAL]
    self.loc = [u for u in sorted(rngs, key=lambda r: r.arg, reverse=True) if u.arg.kind is
AxisType.LOCAL]
    assert len(self.glob) <= 3 and len(self.loc) <= 3, "fold axes: CUDA has 3 grid and 3 block
dimensions"
    self.kernel_prefix = f'extern "C" __global__ void __launch_bounds__({max(1,
prod(launch_dims(sink)[1]))})'
    # dtypes.f16.ctype is gcc's _Float16; CUDA spells it `half` (from cuda_fp16.h)
    return super().render(sink, name).replace("_Float16", "half")

  def header(self, sink, lin) -> str:
    return ("#include <cuda_fp16.h>\n" if any(u.dtype == dtypes.f16 for u in lin) else "") +
CUDA_PRELUDE

  def is_loop(self, u: UOp) -> bool: return u.arg.kind not in (AxisType.GLOBAL, AxisType.LOCAL)

  def range_name(self, u: UOp) -> str:
    if u.arg.kind is AxisType.GLOBAL: return f"gidx{self.glob.index(u)}"
    if u.arg.kind is AxisType.LOCAL: return f"lidx{self.loc.index(u)}"
    return super().range_name(u)

  def render_range(self, u: UOp, name: str, end: str) -> list[str]:
    if u.arg.kind is AxisType.GLOBAL: return [f"int {name} = blockIdx.
{'xyz'[self.glob.index(u)]}; /* {end} */"]
    if u.arg.kind is AxisType.LOCAL: return [f"int {name} = threadIdx.{'xyz'[self.loc.index(u)]};
/* {end} */"]
    return [f"for (int {name} = 0; {name} < {end}; {name}++) {{"]
```

Line by line:

- `buffer_prefix`, `shared_prefix`, `barrier`, `restrict` are the spelling differences. `__restrict__` is CUDA's `restrict`, and it tells the compiler that `data0` and `data2` never alias, so loads can be reordered and cached in the read-only path. `shared_prefix` and `barrier` are used in Chapter 13.

- `render` sorts the GPU ranges by `RangeArg` in *reverse* order, so the range created last, which comes from the last output axis, becomes `gidx0` / `lidx0` and is bound to `.x`. That is the coalescing rule from 12.5, implemented in one `sorted(..., reverse=True)`.
- The kernel prefix is `extern "C" __global__ void __launch_bounds__(threads)`. `extern "C"` turns off C++ name mangling, so `cuModuleGetFunction` can find the kernel by its plain name.
- `is_loop` is `False` for `GLOBAL` / `LOCAL` ranges. The base renderer then neither indents nor emits a closing `}` for them (look at the `END` case in `CRenderer.render`).
- `render_range` declares `int gidx0 = blockIdx.x;` and keeps the range size as a comment, because the hardware, not the code, enforces the bound.

The prelude replaces `<math.h>`. NVRTC compiles without the host's system headers, so `INFINITY` and `NAN` have to be defined from bit patterns, and the floor-division helpers from Chapter 4 become `__device__` functions:

uopc/gpu.py · CUDA_PRELUDE

```
CUDA_PRELUDE = """#define INFINITY (__int_as_float(0x7f800000))
#define NAN (__int_as_float(0x7fffffff))
__device__ inline int floordiv_i32(int a, int b) { int q = a / b; return (q * b != a && ((a < 0)
!= (b < 0))) ? q - 1 : q; }
__device__ inline int floormod_i32(int a, int b) { int r = a % b; return (r != 0 && ((r < 0) !=
(b < 0))) ? r + b : r; }
"""
```

Finally, `launch_dims` reads the grid and block sizes back off the IR, in the same reversed order the renderer used:

uopc/gpu.py · launch_dims

```
def launch_dims(sink: UOp) -> tuple[tuple[int, ...], tuple[int, ...]]:
  """(grid, block) sizes from the GLOBAL and LOCAL ranges, in x, y, z order."""
  rngs = sorted((u for u in sink.toposort() if u.op is Ops.RANGE), key=lambda r: r.arg,
reverse=True)
  g = tuple(r.src[0].arg for r in rngs if r.arg.kind is AxisType.GLOBAL)
  l = tuple(r.src[0].arg for r in rngs if r.arg.kind is AxisType.LOCAL)
  return (g + (1,) * (3 - len(g)), l + (1,) * (3 - len(l)))
```

First kernel: an elementwise op

Let's put a `(64, 128)` elementwise kernel, `c = a*2 + b`, through the whole path:

```python
# scratch/e1.py
import numpy as np
from uopc.uop import UOp, Ops, ParamArg
from uopc.dtype import dtypes
from uopc.rangeify import Opts
from uopc.gpu import gpu_lower, gpu_opt_choice, emulate, render_cuda, launch_dims
from uopc.runtime import Program
def buf(slot, n, d=dtypes.f32): return UOp(Ops.BUFFER, (), ParamArg(slot, d, (n,)))
# elementwise: c = a*2 + b over (64, 128)
A = buf(0, 64*128).reshape((64,128)); B = buf(1, 64*128).reshape((64,128))
body = A * 2.0 + B
o = gpu_opt_choice(body); print("opts:", o)
ast = gpu_lower(body, buf(2, 64*128), o)
print("launch:", launch_dims(ast))
print(render_cuda(ast, "add2"))
print(emulate(ast, "add2"))
a = np.random.randn(64*128).astype(np.float32); b = np.random.randn(64*128).astype(np.float32); c
= np.zeros_like(a)
Program(emulate(ast, "add2"), "add2")(a, b, c)
print("max error:", np.abs(c - (a*2+b)).max())
```

```
opts: Opts(upcast={}, thread=None, unroll=1, tile={}, gpu=True, local={1: 16, 0: 16})
launch: ((8, 4, 1), (16, 16, 1))
```

```cpp
#define INFINITY (__int_as_float(0x7f800000))
#define NAN (__int_as_float(0x7fffffff))
__device__ inline int floordiv_i32(int a, int b) { int q = a / b; return (q * b != a && ((a < 0)
!= (b < 0))) ? q - 1 : q; }
__device__ inline int floormod_i32(int a, int b) { int r = a % b; return (r != 0 && ((r < 0) !=
(b < 0))) ? r + b : r; }
extern "C" __global__ void __launch_bounds__(256) add2(float* __restrict__ data0, float*
__restrict__ data1, float* __restrict__ data2) {
  int gidx1 = blockIdx.y; /* 4 */
  int lidx1 = threadIdx.y; /* 16 */
  int gidx0 = blockIdx.x; /* 8 */
  int lidx0 = threadIdx.x; /* 16 */
  int alu1 = (((gidx1*2048)+(lidx1*128))+((gidx0*16)+lidx0));
  float val2 = data0[alu1];
  float val3 = data1[alu1];
  data2[alu1] = ((val2*2.0f)+val3);
}
```

There are 8×4 = 32 blocks of 16×16 = 256 threads, one thread per output element. The row is `gidx1*16 + lidx1` and the column is `gidx0*16 + lidx0`. The flat address `row*128 + col` has been simplified into the single sum `alu1`. Nothing in this kernel loops. Its "loop nest" is the launch.

Compare the emulator's version of the same kernel, printed by the same script:

```
void add2(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    for (int ridx1 = 0; ridx1 < 16; ridx1++) {
      for (int ridx2 = 0; ridx2 < 8; ridx2++) {
        for (int ridx3 = 0; ridx3 < 16; ridx3++) {
          int alu1 = (((ridx0*2048)+(ridx1*128))+((ridx2*16)+ridx3));
          float val2 = data0[alu1];
          float val3 = data1[alu1];
          data2[alu1] = ((val2*2.0f)+val3);
        }
      }
    }
  }
}
```

```
max error: 0.0
```

It is the same linearized program with four `for` loops added. Range 0 is axis 0's `GLOBAL`, range 1 is axis 0's `LOCAL`, then axis 1's two ranges. That is the interleaving mentioned in 12.6. The body and its index arithmetic are identical, character for character, apart from the names.

12.8 GEMM on the grid

The elementwise case is easy because every thread is independent and touches one element. A GEMM adds a reduce loop and makes the choices matter. Here are three configurations of the 64×64×64 GEMM from `tests/test_w07_gpu.py`, each rendered, emulated, and compared against numpy:

```python
# scratch/e2.py
import sys; sys.path.insert(0, "tests")
import numpy as np
from helpers import buf
from uopc.uop import Ops
from uopc.rangeify import Opts
from uopc.gpu import gpu_lower, gpu_opt_choice, emulate, render_cuda, launch_dims
from uopc.runtime import Program
def gemm_graph(M, N, K):
  A, B = buf(0, M * K), buf(1, K * N)
  return (A.reshape((M, 1, K)).expand((M, N, K)) * B.reshape((K, N)).permute((1, 0)).reshape((1,
N, K)).expand((M, N, K))).reduce(Ops.ADD, 1)
M, N, K = 64, 64, 64
g = gemm_graph(M, N, K)
for name, o in [("plain", Opts(gpu=True)), ("local", Opts(gpu=True, local={0: 8, 1: 16})),
("auto", gpu_opt_choice(g))]:
  ast = gpu_lower(g, buf(2, M*N), o)
  print(f"=== {name}: {o}\nlaunch (grid, block): {launch_dims(ast)}")
  print(render_cuda(ast, "gemm").split("floormod_i32(int a, int b)")[1].split("\n",1)[1])
  a, b = np.random.randn(M, K).astype(np.float32), np.random.randn(K, N).astype(np.float32); c =
np.zeros(M*N, np.float32)
  Program(emulate(ast, "gemm"), "gemm")(a, b, c)
  print("emulated max error vs numpy:", np.abs(c.reshape(M,N) - a@b).max())
```

Configuration 1: one thread per block.

```
=== plain: Opts(upcast={}, thread=None, unroll=1, tile={}, gpu=True, local={})
launch (grid, block): ((64, 64, 1), (1, 1, 1))
```

```cpp
extern "C" __global__ void __launch_bounds__(1) gemm(float* __restrict__ data0, float*
__restrict__ data1, float* __restrict__ data2) {
  int gidx1 = blockIdx.y; /* 64 */
  int alu1 = (gidx1*64);
  int gidx0 = blockIdx.x; /* 64 */
  float acc0;
  acc0 = 0.0f;
  for (int ridx2 = 0; ridx2 < 64; ridx2++) {
    float val2 = acc0;
    float val3 = data0[(alu1+ridx2)];
    float val4 = data1[((ridx2*64)+gidx0)];
    acc0 = (val2+(val3*val4));
  }
  float val5 = acc0;
  data2[(alu1+gidx0)] = val5;
}
```

```
emulated max error vs numpy: 5.722046e-06
```

This is correct and wasteful. Every block is a single thread, and a warp is still 32 lanes wide, so 31 of every 32 lanes do nothing. You pay for 32 lanes and use one. Notice the reduce loop: it is the ordinary `RANGE` of kind `REDUCE` from Chapter 6, and the accumulator `acc0` is a register. The linearizer placed it inside the `gidx` scope exactly as it would place a CPU accumulator inside an output loop. None of that code changed.

Configuration 2: 16×8 threads per block.

```
=== local: Opts(upcast={}, thread=None, unroll=1, tile={}, gpu=True, local={0: 8, 1: 16})
launch (grid, block): ((4, 8, 1), (16, 8, 1))
```

```
extern "C" __global__ void __launch_bounds__(128) gemm(float* __restrict__ data0, float*
__restrict__ data1, float* __restrict__ data2) {
  int gidx1 = blockIdx.y; /* 8 */
  int lidx1 = threadIdx.y; /* 8 */
  int alu1 = ((gidx1*512)+(lidx1*64));
  int gidx0 = blockIdx.x; /* 4 */
  int lidx0 = threadIdx.x; /* 16 */
  int alu2 = ((gidx0*16)+lidx0);
  float acc0;
  acc0 = 0.0f;
  for (int ridx4 = 0; ridx4 < 64; ridx4++) {
    float val3 = acc0;
    float val4 = data0[(alu1+ridx4)];
    float val5 = data1[((ridx4*64)+alu2)];
    acc0 = (val3+(val4*val5));
  }
  float val6 = acc0;
  data2[(alu1+alu2)] = val6;
}
```

```
emulated max error vs numpy: 5.722046e-06
```

Now 128 threads cooperate in each block. Look at the loads inside the loop with a warp in mind. A warp is two values of `lidx1` × sixteen of `lidx0`. `data0[alu1 + ridx4]` does not depend on `lidx0`, so the 16 lanes in a row read the same word, a broadcast. `data1[ridx4*64 + alu2]` reads 16 consecutive floats per row. Both are coalesced, which is the analysis from 12.5 seen in source.

Configuration 3: what `gpu_opt_choice` picks.

```
=== auto: Opts(upcast={1: 2, 0: 2}, thread=None, unroll=1, tile={}, gpu=True, local={1: 16, 0:
16})
launch (grid, block): ((2, 2, 1), (16, 16, 1))
```

```
extern "C" __global__ void __launch_bounds__(256) gemm(float* __restrict__ data0, float*
__restrict__ data1, float* __restrict__ data2) {
  int gidx1 = blockIdx.y; /* 2 */
  int lidx1 = threadIdx.y; /* 16 */
  int alu1 = ((gidx1*2048)+(lidx1*128));
  int gidx0 = blockIdx.x; /* 2 */
  int lidx0 = threadIdx.x; /* 16 */
  int alu2 = ((gidx0*32)+(lidx0*2));
  int alu3 = (alu1+alu2);
  float acc0;
  acc0 = 0.0f;
  float acc1;
  acc1 = 0.0f;
  float acc2;
  acc2 = 0.0f;
  float acc3;
  acc3 = 0.0f;
  int alu4 = (alu2+alu1);
  for (int ridx4 = 0; ridx4 < 64; ridx4++) {
    float val5 = acc0;
    int alu6 = (alu1+ridx4);
    float val7 = data0[alu6];
    int alu8 = (ridx4*64);
    float val9 = data1[(alu8+alu2)];
    acc0 = (val5+(val7*val9));
    float val10 = acc1;
    float val11 = data1[((alu2+alu8)+1)];
    acc1 = (val10+(val7*val11));
    float val12 = acc2;
    float val13 = data0[(alu6+64)];
    acc2 = (val12+(val13*val9));
    float val14 = acc3;
    acc3 = (val14+(val13*val11));
  }
  float val15 = acc0;
  data2[alu3] = val15;
  float val16 = acc1;
  data2[(alu4+1)] = val16;
  float val17 = acc2;
  data2[(alu3+64)] = val17;
  float val18 = acc3;
  data2[(alu4+65)] = val18;
}
```

```
emulated max error vs numpy: 7.6293945e-06
```

This is Chapter 10's register tile running on a GPU. Each thread owns a 2×2 tile of C: four accumulators, two loads of A and two of B per iteration, four FMAs. That doubles the FLOPs per load and gives four independent FMA chains, which is the ILP that Volkov's argument asks for. Only 4 blocks are launched, though, which would leave 104 of an A100's 108 SMs idle. At 64×64 the problem is simply too small for a GPU. A real launch configuration has to balance work per thread against enough blocks to fill the machine. Exercise 12.8 asks you to add that constraint to the heuristic.

The 2×2 upcast has a cost that is easy to miss. The column index is `alu2 = gidx0*32 + lidx0*2`, so neighbouring lanes are now *two* floats apart. Each B load (`data1[alu8+alu2]` and the `+1` one) touches every other word across 16 lanes. The pair of loads together covers a contiguous range, and the second one will usually hit in L1, but each instruction alone is only half-coalesced. The fix is to make the unrolled copies the *slow* part of the index, `(gidx*upcast + c)*local + lidx` rather than `(gidx*local + lidx)*upcast + c`. That is Exercise 12.4. CPU compilers want the unrolled elements adjacent (one SIMD vector per thread). GPU compilers want the *threads* adjacent (one vector per warp). The same UPCAST needs a different index layout on each target.

One more detail appears in the source. `alu3 = (alu1+alu2)` and `alu4 = (alu2+alu1)` are the same value written twice. The symbolic engine does not put the operands of commutative ops in a canonical order, so hash consing sees two different nodes. NVRTC will merge them, so the cost is cosmetic, but it shows how much generated code quality depends on canonicalization (Chapter 3).

12.9 Choosing a launch configuration

`gpu_opt_choice` is the GPU counterpart of `hand_opt_choice` from Chapter 10. It is a dozen lines of heuristics:

`uopc/gpu.py · gpu_opt_choice`

```python
def gpu_opt_choice(body: UOp, max_threads: int = 256) -> Opts:
  """Up to 3 output axes on the grid. Threads on the last axes (coalesced stores), 2x2 upcast for reduces."""
  shape = body.shape
  o = Opts(gpu=True)
  reduces = any(u.op is Ops.REDUCE for u in body.toposort())
  axes = [k for k, s in enumerate(shape) if s > 1][-3:]
  if len(axes) < len(shape) - shape.count(1):
    raise NotImplementedError("more than 3 non-trivial output axes: reshape first")
  budget = max_threads
  for k in reversed(axes):
    if reduces and shape[k] % 2 == 0 and shape[k] >= 32: o.upcast[k] = 2
    rest = shape[k] // o.upcast.get(k, 1)
    t = 1
    while t * 2 <= min(budget, 16 if len(axes) > 1 else budget) and rest % (t * 2) == 0: t *= 2
    if t > 1: o.local[k] = t; budget //= t
  return o
```

The rules, and the reasons for them:

1. **At most three non-trivial output axes go on the grid**, the last three, because CUDA has three grid dimensions and three block dimensions. A kernel with four (GPT's attention scores are `(B, H, T, T)`) raises an error. The fix is to fold two axes into one with a reshape before lowering (Exercise 12.7), the same trick `tinygrad` uses.
2. **Threads go on the last axes first**, so `.x` gets the contiguous axis (12.5).
3. **Power-of-two thread counts that divide the axis**, at most 16 per axis when there are several axes (16×16 = 256) and up to 256 for a 1-D kernel. 256 threads is a common default: 8 warps, enough to hide latency, and small enough to leave registers for a tile.
4. **A 2×2 upcast if the kernel reduces**, because a reduce loop reloads its inputs on every iteration and a register tile cuts those loads in half (Chapter 10).

Its choices for some shapes, from a scratch script:

```python
# scratch/e3.py (excerpt)
for shape in [(1024,), (4096,), (64, 64), (3, 100), (8, 64, 64)]:
  b = buf(0, int(np.prod(shape))).reshape(shape)
  print(shape, "->", gpu_opt_choice(b + 1.0))
s = (8, 4, 64, 64)
try: gpu_opt_choice(buf(0, int(np.prod(s))).reshape(s) + 1.0)
except NotImplementedError as e: print(s, "-> NotImplementedError:", e)
```

```
(1024,) -> Opts(upcast={}, thread=None, unroll=1, tile={}, gpu=True, local={0: 256})
(4096,) -> Opts(upcast={}, thread=None, unroll=1, tile={}, gpu=True, local={0: 256})
(64, 64) -> Opts(upcast={}, thread=None, unroll=1, tile={}, gpu=True, local={1: 16, 0: 16})
(3, 100) -> Opts(upcast={}, thread=None, unroll=1, tile={}, gpu=True, local={1: 4})
(8, 64, 64) -> Opts(upcast={}, thread=None, unroll=1, tile={}, gpu=True, local={2: 16, 1: 16})
(8, 4, 64, 64) -> NotImplementedError: more than 3 non-trivial output axes: reshape first
```

`(3, 100)` shows the weakness of the divisibility rule. 100 = 4 × 25, so the best power of two is 4 threads per block, and three is odd, so it gets none. The result is 75 blocks of 4 threads, one eighth of a warp each. Padding to 104 or 128 with guarded stores would give full warps. This is the same lesson as Configuration 1: a block size smaller than 32 wastes lanes.

A kernel with a reduction inside

The second test in `tests/test_w07_gpu.py` is the numerator of a softmax, `exp2(x - max(x))` over rows of a 16×32 matrix. The `max` is a `REDUCE` whose result is broadcast back over the row:

```
Opts(upcast={1: 2}, thread=None, unroll=1, tile={}, gpu=True, local={1: 16, 0: 16})
((1, 1, 1), (16, 16, 1))
```

```
extern "C" __global__ void __launch_bounds__(256) softmax_num(float* __restrict__ data0, float*
__restrict__ data1) {
  int lidx1 = threadIdx.y; /* 16 */
  int alu1 = (lidx1*32);
  float acc0;
  acc0 = -INFINITY;
  for (int ridx2 = 0; ridx2 < 32; ridx2++) {
    float val2 = acc0;
    float val3 = data0[(alu1+ridx2)];
    acc0 = ((val2>val3)?val2:val3);
  }
  float val4 = acc0;
  float alu5 = (val4*-1.0f);
  int lidx0 = threadIdx.x; /* 16 */
  int alu6 = (lidx0*2);
  int alu7 = (alu1+alu6);
  float val8 = data0[alu7];
  data1[alu7] = exp2f((val8+alu5));
  int alu9 = ((alu6+alu1)+1);
  float val10 = data0[alu9];
  data1[alu9] = exp2f((val10+alu5));
}
```

```
max rel error: 0.0
```

The whole problem fits in one block, so both axes became `CONST 0` + `LOCAL` and no `blockIdx` appears. The `exp2` axis has `upcast={1: 2}`, so each of the 16 `lidx0` threads writes two adjacent elements. Notice where the linearizer put the reduce loop: *before* `int lidx0 = threadIdx.x;`. The max depends only on the row (`lidx1`), so its scope does not include `lidx0`, and the linearizer emits it as early as it can. On a CPU that ordering would be loop-invariant code motion, computing the max once per row instead of once per element. On a GPU every thread still runs the whole kernel, so all 16 threads of a row compute the same max, each reading the full row. The work is 16× redundant, and it's cheap here only because the row is in cache. A real GPU compiler computes the reduction *cooperatively*: each thread reduces a slice, and the partial results are combined through shared memory or warp shuffles (`__shfl_xor_sync`). Exercise 12.6 sketches it. This is the most important difference between CPU and GPU code generation in this chapter. **Loop-invariant on a CPU means "hoist it". Thread-invariant on a GPU means "everybody does it".**

12.10 Testing without a GPU: the emulator

`emulate` is the shortest function in the chapter:

uopc/gpu.py · emulate

```python
def emulate(sink: UOp, name: str = "kernel") -> str:
  """The C source of a GPU kernel where blockIdx/threadIdx are loops. Run it with
runtime.Program."""
  if any(u.op is Ops.BARRIER for u in sink.toposort()):
    raise NotImplementedError("emulate: kernels with BARRIER need all threads to run in lockstep;
"
                              "running threads one after another (what loops do) would be wrong")
  return CRenderer().render(sink, name)
```

It is the *C* renderer, applied to the GPU-lowered IR. `CRenderer.is_loop` returns `True` for every range, `GLOBAL` and `LOCAL` included, so each one becomes a `for` loop, and the resulting C runs through the ordinary gcc runtime from Chapter 4. Why is that a valid test?

A CUDA kernel without shared memory or barriers specifies one independent program per thread. The program for thread `(b, t)` reads its inputs, computes, and writes *its own* output elements. If no two threads write the same element and no thread reads what another writes (true of everything rangeify

produces: each output element is written by exactly one combination of indices), then every interleaving of the threads gives the same result, and that includes running them one after another in nested loops. The emulator runs the same `alu` expressions, the same accumulator placement and the same unrolled copies as the CUDA kernel. If the emulated kernel computes `a @ b`, then the index math, the reduce lowering and the upcast are all right. Only the spelling of `blockIdx` is left untested, and the rendering tests cover that.

This is how all of `tests/test_w07_gpu.py` runs on the reference machine:

tests/test_w07_gpu.py · test_gpu_gemm_emulated

```
@pytest.mark.parametrize("opts", [Opts(gpu=True), Opts(gpu=True, local={0: 4, 1: 8}),
                                  Opts(gpu=True, local={1: 16}, upcast={0: 2, 1: 2})])
def test_gpu_gemm_emulated(opts):
  M, N, K = 32, 64, 48
  a, b = np.random.randn(M, K).astype(np.float32), np.random.randn(K, N).astype(np.float32)
  c = np.zeros(M * N, np.float32)
  ast = gpu_lower(gemm_graph(M, N, K), buf(2, M * N), opts)
  Program(emulate(ast), "kernel")(a, b, c)
  np.testing.assert_allclose(c.reshape(M, N), a @ b, atol=1e-4)
  src = render_cuda(ast)
  assert "__global__" in src and ("threadIdx" in src) == bool(opts.local)
```

Each parameterization lowers the GEMM with different GPU opts, runs the emulated C, compares with numpy, and then checks that the CUDA rendering mentions `threadIdx` exactly when there are `LOCAL` axes. The run on this machine:

```
tests/test_w07_gpu.py::test_gpu_gemm_emulated[opts0] PASSED              [  9%]
tests/test_w07_gpu.py::test_gpu_gemm_emulated[opts1] PASSED              [ 18%]
tests/test_w07_gpu.py::test_gpu_gemm_emulated[opts2] PASSED              [ 27%]
tests/test_w07_gpu.py::test_gpu_softmax_like_emulated PASSED             [ 36%]
tests/test_w07_gpu.py::test_launch_dims_and_auto_choice PASSED           [ 45%]
tests/test_w07_gpu.py::test_smem_gemm_renders PASSED                     [ 54%]
tests/test_w07_gpu.py::test_wmma PASSED                                  [ 63%]
tests/test_w07_gpu.py::test_cuda_smem_gemm_on_gpu SKIPPED (no CUDA d...) [ 72%]
```

What the emulator *cannot* tell you:

- **Anything involving barriers.** If threads communicate through shared memory, running them one after another is wrong: thread 0 would read the tile before thread 255 has written its part. Chapter 13 shows the failure and works around it. (NOTES 25: "Barriers can't be emulated by running threads one after another".)
- **Races** that only a parallel execution exposes, such as two threads accumulating into the same output.

- **CUDA-specific compile errors.** The emulator compiles C with gcc, not CUDA C++ with NVRTC.
- **Performance.** An emulated kernel's speed says nothing about the GPU's.

The emulator gives up all of that in exchange for a property that matters a lot in a course: every GPU code path in the compiler is tested on every student's laptop, in milliseconds, with an oracle (numpy).

12.11 Running on a real GPU: `cuda.py`

To run on an NVIDIA GPU we need three things: a compiler from CUDA C++ to something the GPU can load, a way to move memory to and from the device, and a way to launch. `uopc` does all three with `ctypes` and two shared libraries, so it needs no `nvcc` and no PyTorch:

- **NVRTC** (`libnvrtc.so`, part of the CUDA toolkit) is a runtime compiler. It takes CUDA source as a string and produces **PTX**, NVIDIA's virtual assembly.
- The **driver API** (`libcuda.so`, installed with the NVIDIA driver) loads PTX, which the driver JIT-compiles to the GPU's machine code (SASS), and manages contexts, memory and launches.

The whole runtime:

`uopc/cuda.py · CUDA`

```python
class CUDA:
  def __init__(self, device: int = 0):
    self.cu = _lib("cuda")
    self.rtc = _lib("nvrtc")
    self._check(self.cu.cuInit(0))
    self.dev = ctypes.c_int()
    self._check(self.cu.cuDeviceGet(ctypes.byref(self.dev), device))
    self.ctx = ctypes.c_void_p()
    self._check(self.cu.cuCtxCreate_v2(ctypes.byref(self.ctx), 0, self.dev))
    major, minor = ctypes.c_int(), ctypes.c_int()
    self._check(self.cu.cuDeviceComputeCapability(ctypes.byref(major), ctypes.byref(minor),
self.dev))
    self.arch = f"sm_{major.value}{minor.value}"

  def _check(self, status):
    if status != 0: raise CUDAError(f"CUDA driver error {status}")

  def compile(self, src: str) -> bytes:
    prog = ctypes.c_void_p()
    if self.rtc.nvrtcCreateProgram(ctypes.byref(prog), src.encode(), b"kernel.cu", 0, None,
None):
      raise CUDAError("nvrtcCreateProgram failed")
    opts = [f"--gpu-architecture=compute_{self.arch[3:]}".encode(), b"--use_fast_math"]
    status = self.rtc.nvrtcCompileProgram(prog, len(opts), (ctypes.c_char_p * len(opts))(*opts))
    if status != 0:
      n = ctypes.c_size_t()
      self.rtc.nvrtcGetProgramLogSize(prog, ctypes.byref(n))
      log = ctypes.create_string_buffer(n.value)
      self.rtc.nvrtcGetProgramLog(prog, log)
      raise CUDAError(f"nvrtc: {log.value.decode()}\n{src}")
    n = ctypes.c_size_t()
    self.rtc.nvrtcGetPTXSize(prog, ctypes.byref(n))
    ptx = ctypes.create_string_buffer(n.value)
    self.rtc.nvrtcGetPTX(prog, ptx)
    return ptx.raw

  def alloc(self, nbytes: int) -> ctypes.c_uint64:
    p = ctypes.c_uint64()
    self._check(self.cu.cuMemAlloc_v2(ctypes.byref(p), ctypes.c_size_t(nbytes)))
    return p

  def to_device(self, a: np.ndarray) -> ctypes.c_uint64:
    p = self.alloc(a.nbytes)
    self._check(self.cu.cuMemcpyHtoD_v2(p, a.ctypes.data_as(ctypes.c_void_p),
ctypes.c_size_t(a.nbytes)))
    return p

  def to_host(self, p: ctypes.c_uint64, a: np.ndarray):
    self._check(self.cu.cuMemcpyDtoH_v2(a.ctypes.data_as(ctypes.c_void_p), p,
ctypes.c_size_t(a.nbytes)))

  def free(self, p): self.cu.cuMemFree_v2(p)
```

```
  def function(self, src: str, name: str):
    mod = ctypes.c_void_p()
    self._check(self.cu.cuModuleLoadData(ctypes.byref(mod), self.compile(src)))
    fn = ctypes.c_void_p()
    self._check(self.cu.cuModuleGetFunction(ctypes.byref(fn), mod, name.encode()))
    return fn

  def launch(self, fn, grid, block, *ptrs, shared: int = 0):
    args = (ctypes.c_void_p * len(ptrs))(*(ctypes.cast(ctypes.byref(p), ctypes.c_void_p) for p in
ptrs))
    self._check(self.cu.cuLaunchKernel(fn, *grid, *block, shared, None, args, None))
    self._check(self.cu.cuCtxSynchronize())
```

The calls, in order:

1. `cuInit(0)`, `cuDeviceGet`, `cuCtxCreate_v2`. A *context* is the GPU equivalent of a process address space. All allocations and modules belong to one. `cuDeviceComputeCapability` returns e.g. `(8, 0)` for an A100, giving `sm_80`.
2. `compile`: `nvrtcCreateProgram` → `nvrtcCompileProgram` with `--gpu-architecture=compute_80` → fetch the PTX. On failure it fetches the compiler log and raises it with the source attached. You will want that the first time a renderer change produces invalid CUDA.
3. `function`: `cuModuleLoadData` (PTX → SASS, inside the driver) and `cuModuleGetFunction` by name, which is why the kernel is `extern "C"`.
4. `alloc` / `to_device` / `to_host`: `cuMemAlloc_v2` returns a 64-bit device pointer, a `CUdeviceptr`, which is just an integer. `cuMemcpyHtoD_v2` / `DtoH_v2` copy to and from numpy arrays.
5. `launch`: `cuLaunchKernel(f, gx, gy, gz, bx, by, bz, sharedBytes, stream, kernelParams, extra)`. `kernelParams` is an array of *pointers to* the arguments. Each argument is a `CUdeviceptr`, so we pass the address of each `c_uint64`. That double indirection is the classic first-time bug, and the reason for the `ctypes.cast(ctypes.byref(p), c_void_p)`. Then `cuCtxSynchronize` waits for the kernel, because launches are asynchronous.

`run_cuda` packages the round trip:

`uopc/cuda.py · run_cuda`

```python
def run_cuda(src: str, name: str, grid, block, *arrays: np.ndarray):
  """Copy arrays to the GPU, run, copy every array back (they may be outputs)."""
  cu = CUDA()
  ptrs = [cu.to_device(a) for a in arrays]
  cu.launch(cu.function(src, name), grid, block, *ptrs)
  for p, a in zip(ptrs, arrays): cu.to_host(p, a); cu.free(p)
```

On this machine `find_library("cuda")` succeeds, since driver libraries are installed but no working driver is loaded, while NVRTC is missing altogether. So the constructor fails before any CUDA call:

```python
from uopc.cuda import CUDA
try: CUDA()
except Exception as e: print(type(e).__name__+':', e)
```

```
OSError: none of ('nvrtc',) found
```

So `cuda.py` has not been executed on the reference machine (NOTES 25). It follows the documented driver and NVRTC APIs, and the test below is the way to check it. Treat it as your week-7 starting point, not as proven code.

How to run the GPU test if you have a GPU

`tests/test_w07_gpu.py · test_cuda_smem_gemm_on_gpu`

```python
@pytest.mark.skipif(not _has_cuda(), reason="no CUDA device / NVRTC")
def test_cuda_smem_gemm_on_gpu():
  from uopc.cuda import run_cuda
  N = 64
  a, b, c = np.random.randn(N, N).astype(np.float32), np.random.randn(N, N).astype(np.float32),
np.zeros(N * N, np.float32)
  ast = smem_gemm_ir(N, 16)
  grid, block = launch_dims(ast)
  run_cuda(render_cuda(ast, "smem_gemm"), "smem_gemm", grid, block, a, b, c)
  np.testing.assert_allclose(c.reshape(N, N), a @ b, atol=1e-3)
```

The test is skipped unless `CUDA()` constructs successfully, so it runs automatically on a machine with a working setup:

1. Install an NVIDIA driver (`nvidia-smi` should list your GPU) and the CUDA toolkit, or just NVRTC, from the CUDA redistributables.

2. Make sure `ctypes.util.find_library("cuda")` and `find_library("nvrtc")` find the libraries. On Linux they look in the `ldconfig` cache, so either run `ldconfig` after installing, or add the toolkit's `lib64` directory to `LD_LIBRARY_PATH`. Check with `python -c "import ctypes.util as u; print(u.find_library('cuda'), u.find_library('nvrtc'))"`.
3. Run `.venv/bin/python -m pytest tests/test_w07_gpu.py::test_cuda_smem_gemm_on_gpu -v`. It should report `PASSED` rather than `SKIPPED`.

If it fails, the most likely culprits are the ones `ctypes` can't check for you. Functions return `int` by default, which is right for `CUresult` and `nvrtcResult`, but wrong if you add any API that returns a pointer. Check the size types (`size_t` versus `int`), and the `_v2` suffixes, which are the real exported symbol names. To time kernels, add `cuEventCreate` / `cuEventRecord` / `cuEventElapsedTime` around the launch. Wall-clock timing in Python also includes the launch overhead, typically several microseconds.

12.12 Beyond CUDA: HIP, Metal, OpenCL

All GPU programming models share the structure of 12.2 and differ in spelling. Since `CUDARenderer` is a set of spellings on top of `CRenderer`, porting to another API means writing another subclass. The mapping:

concept	CUDA	HIP (AMD)	Metal (Apple)	OpenCL
kernel qualifier	`__global__`	`__global__`	`kernel`	`__kernel`
block / thread index	`blockIdx.x`, `threadIdx.x`	same as CUDA	`[[threadgroup_position_in_grid]]`, `[[thread_position_in_threadgroup]]` arguments	`get_group_id(0)`, `get_local_id(0)`
block is called	thread block	workgroup / block	threadgroup	work-group
SIMD width	warp = 32	wavefront = 64 (CDNA), 32 or 64 (RDNA)	SIMD-group = 32	sub-group, device-dependent
shared memory	`__shared__`	`__shared__` (LDS)	`threadgroup`	`__local`
barrier	`__syncthreads()`	`__syncthreads()`	`threadgroup_barrier(mem_flags::mem_threadgroup)`	`barrier(CLK_LOCAL_MEM_FENCE)`
runtime compiler	NVRTC	hipRTC	`newLibraryWithSource`	`clBuildProgram`
launch	`cuLaunchKernel`	`hipModuleLaunchKernel`	`dispatchThreadgroups`	`clEnqueueNDRangeKernel`

HIP is close to source-compatible with CUDA. A `HIPRenderer` is mostly `CUDARenderer` with a different prelude, and `hipRTC` / `hipModuleLaunchKernel` mirror NVRTC and the driver API call for call. One thing does change on AMD: wavefronts on data-center GPUs are 64 wide, so blocks of 64 or 256 threads, and coalescing across 64 lanes, are the units to plan around. Metal and OpenCL pass the indices as kernel arguments or built-in calls rather than as global variables, which means overriding `render_range` and the parameter list. That is still a subclass.

Pitfalls

1. The wrong axis on `.x` (NOTES 23). The obvious way to bind ranges is in creation order, `x, y, z`. That gives the *first* output axis `.x`. Every kernel still produces correct results, in the emulator and on the GPU,

because coalescing is a performance property, not a correctness property. Section 12.5 shows the cost: four times the store traffic for a GEMM output, and A read down a column. The fix is one `reverse=True` in `CUDARenderer.render` and `launch_dims`. The lesson: **a test that only checks numbers will never catch this.** Check the access pattern instead, by evaluating the index expressions for one warp, as the script in 12.5 does.

2. Blocks smaller than a warp. `Opts(gpu=True)` with no `local` launches one thread per block (12.8), and `gpu_opt_choice` on a `(3, 100)` tensor launches blocks of 4. Both are correct and both waste 7/8 or more of the machine. Hardware allocates warps, not threads.

3. Indivisible sizes. `_gpu_axis` asserts `size % (local * factor) == 0`. Without guarded stores, any other choice would write out of bounds. On a GPU an out-of-bounds write may silently corrupt a neighbouring buffer, and the emulator would not catch it either (it is C, without sanitizers). If you add padding (Exercise 12.5), also add an emulator test with a size like 100 that doesn't divide.

4. Four or more axes. `gpu_opt_choice` raises `NotImplementedError` for `(8, 4, 64, 64)`. The attention kernels of Chapter 14 have exactly this shape, so moving GPT to the GPU requires folding axes first.

5. Thinking the emulator covers barriers. Run `emulate` on the shared-memory GEMM of Chapter 13 and it refuses (the first version of `emulate` crashed with an opaque `TypeError` from the renderer's `barrier = None`; it now checks up front):

```
NotImplementedError: emulate: kernels with BARRIER need all threads to run in lockstep; running
threads one after another (what loops do) would be wrong
```

Supplying a no-op barrier spelling to force it through makes it render, and then it computes garbage. Chapter 13 shows the exact numbers. The emulator is sound only for kernels whose threads don't communicate.

6. Hoisting looks free and isn't. The softmax kernel in 12.9 computes each row's max once per *thread*, 16 times per row. Placement decisions that are optimal for a CPU, such as loop-invariant hoisting, become redundant work when the "loop" is a set of parallel threads.

7. `INFINITY` doesn't exist in NVRTC. NVRTC does not see the host's `<math.h>`, so the C prelude's `INFINITY` (used to initialize max-reductions) would fail to compile. `CUDA_PRELUDE` defines it and `NAN` from bit patterns with `__int_as_float`. Every spelling the renderer emits has to exist in the target's dialect, and the emulator can't check that for you, since it compiles with gcc.

Exercises

1. ★ **Predict the launch.** For the 32×64×48 GEMM of `test_gpu_gemm_emulated`, predict `launch_dims` for `Opts(gpu=True, local={0: 4, 1: 8})` and for `Opts(gpu=True, local={1: 16}, upcast={0: 2, 1: 2})`. Then check.
2. ★ **Count the lanes.** For each configuration in 12.8, compute the number of warps launched and the fraction of lanes that do useful work.
3. ★★ **Find the bug.** A student's renderer binds ranges in creation order (`sorted(rngs, key=lambda r: r.arg)`, with no `reverse`). All of their tests pass. Write a test that fails for their renderer and passes for `CUDARenderer`. *Hint:* don't run anything. Inspect `render_cuda(ast)` or evaluate the index of a store for lanes 0 and 1, as in 12.5.
4. ★★ **Coalesced upcasts.** Change the index layout for GPU axes from `(g*local + l)*f + c` to `(g*f + c)*local + l`, so that neighbouring lanes stay adjacent in every unrolled copy. Use the warp-address script to show that every B load in Configuration 3 now touches 2 sectors per instruction, not 4, and that the emulator tests still pass. *Hint:* only `_gpu_axis` and `_combos` need to know the layout.
5. ★★ **Guarded kernels.** Support sizes that local × upcast does not divide. Round the grid up, and wrap each store in a condition `idx < size`. You can express the condition in the IR as a `WHERE` around an `INDEX` to a dummy slot, or add a `gate` source to `STORE`, rendered as `if (...)`. Test `(3, 100)` with 32 threads per block in the emulator.
6. ★★ **Cooperative row max.** Write, by hand in the loop IR (like `smem_gemm_ir` in Chapter 13), a kernel with one block per row and 32 threads per block, where each thread takes the max over a strided slice of the row and lane 0 combines the 32 partials from a `SHARED` array after a `BARRIER`. Render it with `render_cuda`. Explain why the emulator cannot check it.
7. ★★ **Fold axes.** Make `gpu_lower` accept kernels with more than three non-trivial output axes by reshaping the leading axes into one before lowering, and un-reshaping the store index. Test on `(8, 4, 64, 64)` in the emulator.
8. ★★★ **Fill the machine.** Extend `gpu_opt_choice` with a `min_blocks` parameter (say 2 × the SM count). It should drop the upcast, and then shrink `local`, until the grid has at least that many blocks. Write down the trade-off you are making in each step.
9. ★★★ **HIP backend.** Write `HIPRenderer` and a `hip.py` runtime mirroring `cuda.py` (`hiprtc`, `hipModuleLoadData`, `hipModuleLaunchKernel`). Without AMD hardware, test only rendering. With it, port `test_cuda_smem_gemm_on_gpu`.
10. ★★★ **Occupancy calculator.** Write `occupancy(threads_per_block, regs_per_thread, smem_per_block, limits)` returning resident blocks and occupancy, reproduce the table in 12.4, and add

the 256-register allocation granularity per warp from the CUDA occupancy calculator.

Solutions to ★ exercises

1. M=32 with 4 threads per block gives 8 blocks along y. N=64 with 8 threads gives 8 blocks along x. So the grid is `(8, 8, 1)` and the block is `(8, 4, 1)`, with `.x` first and coming from axis 1. With `local={1: 16}` and a 2×2 upcast: axis 0 has 32/2 = 16 blocks and no threads, and axis 1 has 64/(16·2) = 2 blocks of 16 threads. That gives grid `(2, 16, 1)` and block `(16, 1, 1)`. The run:

```
{0: 4, 1: 8} {} ((8, 8, 1), (8, 4, 1))
{1: 16} {0: 2, 1: 2} ((2, 16, 1), (16, 1, 1))
```

2. Plain: 4096 blocks of 1 thread is 4096 warps, with 1/32 = 3.1% of lanes busy. Local: 32 blocks of 128 threads is 128 warps, 100% busy. Auto: 4 blocks of 256 threads is 32 warps, 100% busy, but only 4 SMs used.

Quiz

1. Why is `threadIdx.x`, and not `.y`, bound to the last axis of a row-major output?
2. A block of 256 threads uses 80 registers per thread on an SM with 65,536 registers and a 2,048-thread limit. How many blocks fit, and what is the occupancy?
3. What does `__launch_bounds__(256)` tell the compiler, and why does `uopc` always emit it?
4. Why can't two thread blocks synchronize with each other inside a kernel?
5. Why is it valid to test a GPU kernel by running it as nested CPU loops, and when does it stop being valid?
6. In the softmax kernel of 12.9, how many times is each row's max computed, and why?
7. What does NVRTC produce, and who turns that into machine code?
8. What is a warp called on AMD's data-center GPUs, and how wide is it?

Answers

1. Consecutive `.x` values are consecutive lanes of a warp, so they issue the same load together. Binding them to the contiguous axis puts the warp's 32 addresses into the fewest 32-byte sectors (coalescing).
2. 65536 / (256·80) = 3.2, so 3 blocks (768 threads = 24 warps), and occupancy 24/64 = 37.5%. The thread limit, 8 blocks, doesn't bind.

3. That the kernel is never launched with more than 256 threads per block. The register allocator can then give each thread up to 65536/256 = 256 registers (capped at the 255-register limit) and still fit one block. Without the bound it must assume 1024 threads, or 64 registers each. `uopc` knows the exact block size, so the bound is always tight.
4. Blocks are scheduled in an unspecified order and may not all be resident at once. A barrier could wait forever for a block that cannot start until the waiting ones finish.
5. Without shared memory or communication, each thread computes and writes its own output elements, so every execution order, including a serial one, gives the same result. It stops being valid with barriers or shared memory, and it can't detect races.
6. 16 times, once per thread in `x`. The reduce depends only on `lidx1`, so on a CPU it would be hoisted out of the inner loop. On a GPU every thread runs the hoisted code.
7. PTX, a virtual ISA. The driver (`cuModuleLoadData`) JIT-compiles PTX into SASS for the specific GPU.
8. A wavefront, 64 lanes on CDNA (RDNA supports 32 and 64).

Further reading

- NVIDIA, *CUDA C++ Programming Guide*: the programming model, memory hierarchy, occupancy, and the `__launch_bounds__` and WMMA sections. Also the *CUDA C++ Best Practices Guide* on coalescing.
- NVIDIA, *NVRTC* and *CUDA Driver API* reference manuals: every function `cuda.py` calls.
- NVIDIA, *NVIDIA A100 Tensor Core GPU Architecture* whitepaper (2020): the numbers in 12.1 and 12.3.
- V. Volkov, "Better Performance at Lower Occupancy", GTC 2010.
- D. Kirk and W. Hwu, *Programming Massively Parallel Processors*: the textbook treatment of everything here.
- J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, chapter 4 (data-level parallelism, GPUs).
- tinygrad: `tinygrad/renderer/cstyle.py` (the CUDA, Metal, OpenCL and HIP renderers are subclasses, the same design as `CUDARenderer`) and its GPU dimension handling, which folds and limits axes to the hardware's grid.
- AMD, *HIP Programming Guide*; Apple, *Metal Shading Language Specification*; Khronos, *OpenCL C Specification*.

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

`uopc/gpu.py · 181 lines`

`uopc/cuda.py · 90 lines`

`tests/test_w07_gpu.py · 66 lines`

13. Shared memory and tensor cores

The GPU GEMMs of Chapter 12 are correct, coalesced, and still slow. Each thread computes its output element by streaming a full row of A and a full column of B out of global memory. Across the whole grid every element of A is fetched N times and every element of B is fetched N times. That is 2 FLOPs per 8 bytes, a fiftieth of what an A100 needs to be compute-bound. This chapter fixes it the way every fast GPU GEMM does. Threads in a block *cooperate*: they stage tiles of A and B in on-chip **shared memory**, wait for each other at **barriers**, and reuse each tile T times from there. Then we go one level further, to **tensor cores**, the matrix units that do a 16×16×16 multiply-accumulate as a single warp-wide operation, and to the pattern-matching lowering that `uopc` uses to reach them.

What you'll build

- `AddrSpace.SHARED` allocations, pinned by the linearizer to the top of the kernel (one copy per block).
- The `BARRIER` op, rendered as `__syncthreads()`.
- `smem_gemm_ir`, a tiled GEMM written by hand in the loop IR with two shared tiles and two barriers, and its rendered CUDA.
- A way to check a barrier kernel without a GPU (a lockstep simulation), and an understanding of why the Chapter 12 emulator can't.
- `match_gemm` and `wmma_gemm`: recognize a GEMM in the shaped graph and emit a WMMA tensor-core kernel.

Where this fits. This is the second half of weeks 7–8. Chapter 12 mapped loops to the grid. This chapter adds the memory level between registers and DRAM. Chapter 14 puts the whole compiler to work on a real model. As in Chapter 12, the reference machine has no NVIDIA GPU. CUDA source here is real `render_cuda` output, numerical checks are CPU simulations of the same index arithmetic, and GPU performance figures are general knowledge from vendor documentation, not measurements.

13.1 The traffic problem, in numbers

Take a square GEMM, `C = A @ B` with N = 4096. It performs $2N^3 \approx 137$ GFLOP. How many bytes must cross the DRAM bus?

- **The compulsory minimum** is to read A and B once and write C once: $3 \cdot N^2 \cdot 4$ bytes ≈ 200 MB, of which 134 MB are reads. Against 137 GFLOP that would be about 1000 FLOP per byte, so in principle GEMM is as compute-bound as a kernel can be.
- **The naive kernel** (one thread per output, Chapter 12) loads one A element and one B element per FMA: $2N^3$ loads.
- **A block tile** of BM × BN outputs, computed by one thread block that loads a BM × K panel of A and a K × BN panel of B *once*, reads (BM + BN)·N floats per block. There are (N/BM)(N/BN) blocks.

A scratch script makes the table:

```python
# scratch/traffic.py
N = 4096
flops = 2 * N**3
print(f"N = {N}: {flops/1e9:.0f} GFLOP per GEMM")
print(f"{'scheme':34s} {'DRAM reads':>12s} {'FLOP/byte':>10s}")
def row(name, floats_loaded):
  b = 4 * floats_loaded
  print(f"{name:34s} {b/1e9:9.1f} GB {flops/b:10.1f}")
row("naive: 2 loads per FMA", 2 * N**3)
for BM, BN in [(16, 16), (32, 32), (64, 64), (128, 128)]:
  # each block loads a BM x K panel of A and a K x BN panel of B, (N/BM)*(N/BN) blocks
  row(f"block tile {BM}x{BN} in shared memory", (N // BM) * (N // BN) * (BM + BN) * N)
print(f"compulsory (each input read once): {2*N*N*4/1e6:.0f} MB")
```

```
N = 4096: 137 GFLOP per GEMM
scheme                              DRAM reads  FLOP/byte
naive: 2 loads per FMA                549.8 GB        0.2
block tile 16x16 in shared memory      34.4 GB        4.0
block tile 32x32 in shared memory      17.2 GB        8.0
block tile 64x64 in shared memory       8.6 GB       16.0
block tile 128x128 in shared memory     4.3 GB       32.0
compulsory (each input read once): 134 MB
```

The closed form is intensity = BM·BN / (2(BM + BN)) FLOP per byte, which is **T/4** for a square T × T tile. The A100's FP32 ridge point is about 12.5 FLOP/byte (Chapter 12). So a 16×16 tile is still memory-bound, and 64×64 or 128×128 tiles clear the ridge comfortably. Two caveats apply. First, the naive row overstates real traffic, because L1 and L2 catch many of those repeated loads, so the measured gap is smaller than 140×. Second, caches are not *guaranteed* to catch them, and a 4096^2 matrix (64 MB) does not fit in a 40 MB L2. Shared memory turns "hopefully cached" into "certainly on chip".

> **Key idea:** Tiling is a change of loop order that shortens reuse distance. The block computes the same sums in a different order, so that each value fetched from DRAM is used T times while it is still on chip, and *the programmer*, not a cache policy, decides what stays.

13.2 Shared memory and barriers in the IR

Two additions to the IR make tiling expressible.

Shared allocations. `ALLOC` already took an address space in its `ParamArg` (Chapter 4 used `LOCAL` for accumulators). Weeks 7–8 add one more:

uopc/dtype.py · AddrSpace

```
class AddrSpace(Enum):
  """Where a UOp lives. ALU = a plain value in a register. The rest are memory."""
  ALU = auto()      # a value (week 1)
  GLOBAL = auto()   # device memory, a function argument (week 2)
  LOCAL = auto()    # call-scope storage: `float acc[4];` (week 2)
  SHARED = auto()   # GPU workgroup shared memory (week 7)

  def __repr__(self): return f"AddrSpace.{self.name}"
```

A `SHARED` allocation is one array per *block*, visible to every thread in it. Its placement matters. Recall from Chapter 5 that the linearizer declares an `ALLOC` in the deepest scope shared by all its users, so that each OpenMP thread gets a private accumulator (NOTES 9). For shared memory that rule is wrong. The users of a tile are inside the `LOCAL` (thread) ranges, so the "deepest common scope" would be per thread, which is the opposite of shared. The fix is one line (NOTES 24):

uopc/linearize.py:71-77

```
  for u in topo:
    if u.op is Ops.ALLOC and u.addrspace is AddrSpace.SHARED: target[u] = ()   # shared memory:
whole block, top of kernel
    elif u.op is Ops.ALLOC and users[u]:
      ts = [target[x] for x in users[u]]
      n = 0
      while all(len(t) > n and t[n] is ts[0][n] for t in ts): n += 1
      target[u] = ts[0][:n]
```

A shared allocation always goes in the empty scope, at the top of the kernel. The renderer gives it the `__shared__` prefix (`CRenderer.shared_prefix` is `""` and `CUDARenderer.shared_prefix` is `"__shared__ "`). That is also what CUDA requires in practice: a `__shared__` array has block lifetime wherever it is declared, so declaring it at the top states its real scope.

Barriers. `BARRIER` is an effect, like `STORE`. It has no value (`dtypes.void`) and its sources are the effects it must come after:

```
BARRIER = auto()    # (*deps): __syncthreads()
```

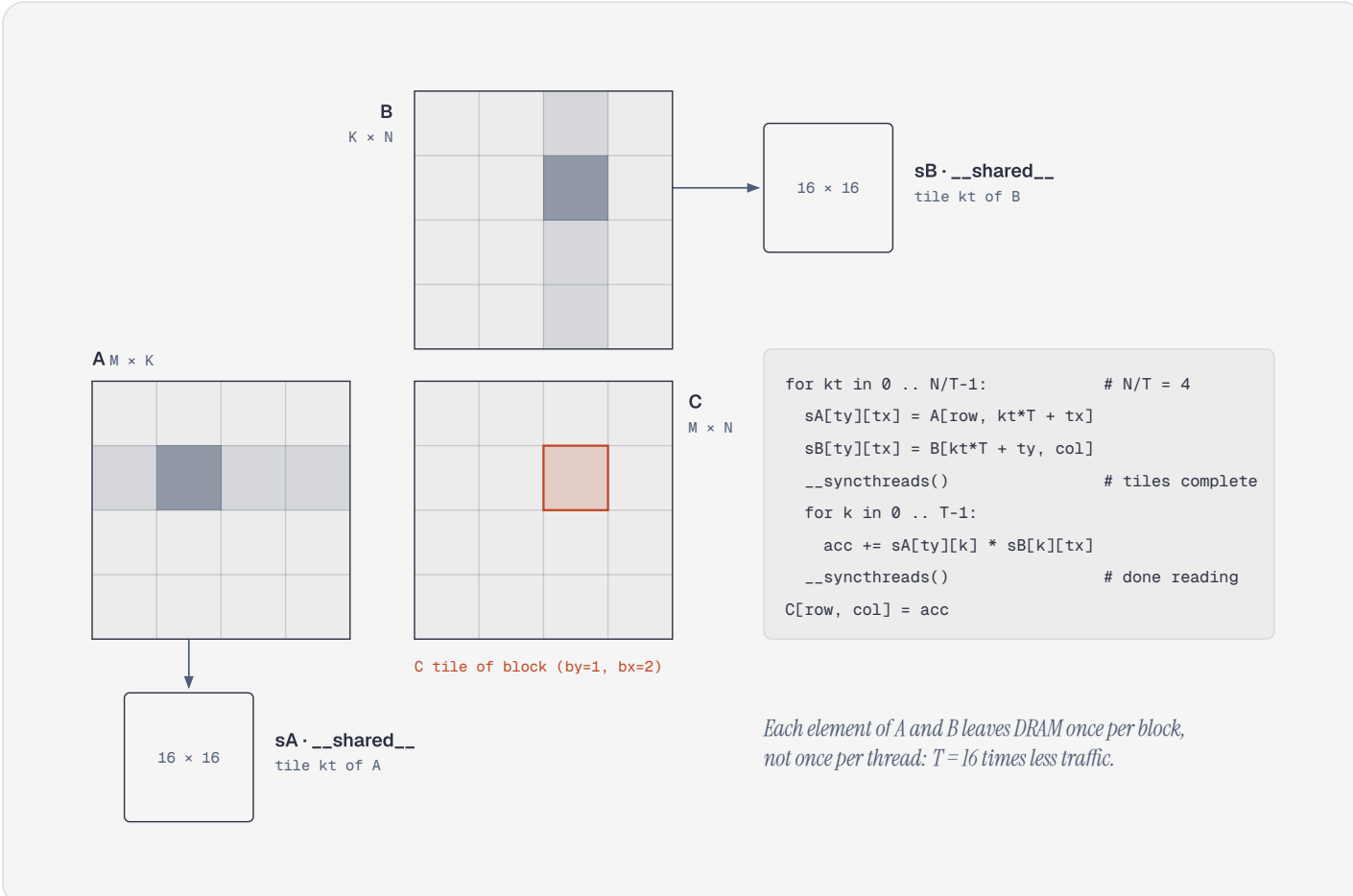
Its semantics: *no thread in the block proceeds past the barrier until every thread in the block has reached it, and all shared-memory writes made before it are visible to all threads after it.* In SSA terms (Chapter 4), a barrier is an `AFTER` that waits on *other threads'* stores as well as this thread's. The renderers spell it with a class attribute: `CUDARenderer.barrier = "__syncthreads()"`, and the base `CRenderer.barrier = None`. The `None` is deliberate: the C renderer has no meaning for a barrier, and `emulate` refuses barrier kernels outright. That is how NOTES 25 shows up in Section 13.5.

13.3 `smem_gemm_ir`, line by line

There is no automatic pass that introduces shared memory in `uopc`. Like the flash-attention kernel of Chapter 11, the tiled GEMM is written by hand in the loop IR, from the same ops the rest of the compiler uses. Writing it by hand is the assignment for week 8, and it is the best way to learn what an automatic pass would have to produce.

`uopc/gpu.py · smem_gemm_ir`

```python
def smem_gemm_ir(N: int, T: int = 16) -> UOp:
  """C = A @ B, N x N, with T x T thread blocks and T x T tiles staged in shared memory.

  Each block computes one T x T tile of C. For each of the N/T steps along K, the
  block cooperatively copies a T x T tile of A and of B into shared memory (one
  element per thread), waits at a barrier, multiplies the tiles out of shared
  memory, and waits again before the next copy overwrites them. Global memory
  traffic drops by a factor of T.
  """
  f32 = dtypes.f32
  A, B, C = (UOp(Ops.BUFFER, (), ParamArg(s, f32, (N * N,))) for s in range(3))
  sA = UOp(Ops.ALLOC, (), ParamArg(0, f32, (T * T,), AddrSpace.SHARED))
  sB = UOp(Ops.ALLOC, (), ParamArg(1, f32, (T * T,), AddrSpace.SHARED))
  acc = UOp(Ops.ALLOC, (), ParamArg(2, f32, (1,)))
  by, bx = range_(N // T, 0, AxisType.GLOBAL), range_(N // T, 1, AxisType.GLOBAL)
  ty, tx = range_(T, 2, AxisType.LOCAL), range_(T, 3, AxisType.LOCAL)
  row, col = by * T + ty, bx * T + tx
  acc0 = acc.after(by, bx, ty, tx)
  init = acc0.index(0).store(UOp.const(0.0))
  kt = range_(N // T, 4)                                    # step along K, one tile at a time
  ld_a = sA.after(kt).index(ty * T + tx).store(A.index(row * N + kt * T + tx).load())
  ld_b = sB.after(kt).index(ty * T + tx).store(B.index((kt * T + ty) * N + col).load())
  sync1 = UOp(Ops.BARRIER, (ld_a, ld_b))
  k = range_(T, 5)
  accl = acc0.after(init, kt, k)
  prod_ = sA.after(sync1).index(ty * T + k).load() * sB.after(sync1).index(k * T + tx).load()
  inner = accl.index(0).store(accl.index(0).load() + prod_).end(k)
  sync2 = UOp(Ops.BARRIER, (inner,))                        # don't overwrite tiles others still
read
  loop = sync2.end(kt)
  out = C.index(row * N + col).store(acc0.after(loop).index(0).load())
  return sink(out.end(by, bx, ty, tx))
```



Line by line, using N = 64 and T = 16:

- **Buffers and allocations.** `A`, `B`, `C` are global (`BUFFER`). `sA` and `sB` are `T*T = 256` floats each in `SHARED`. `acc` is a 1-element `LOCAL` allocation, which the renderer turns into a scalar (a register).
- **The grid.** `by, bx` are `GLOBAL` ranges of size N/T = 4, and `ty, tx` are `LOCAL` ranges of size T = 16. The range indices 0–3 put `bx` and `tx` last, so the renderer binds them to `.x` (Chapter 12's coalescing rule). Each thread owns exactly one C element: `row = by*T + ty`, `col = bx*T + tx`.
- **`acc0 = acc.after(by, bx, ty, tx)`** names the accumulator *for this thread*. The `AFTER` on the ranges puts the `init` store inside the thread's scope (Chapter 6's reduce lowering uses the same idiom).
- **`kt = range_(N // T, 4)`** is the outer reduction loop, one iteration per tile along K.
- **`ld_a`, `ld_b`** are the cooperative copy. Thread `(ty, tx)` copies *one* element of each tile: `A[row, kt*T + tx]` into `sA[ty*T + tx]`, and `B[kt*T + ty, col]` into `sB[ty*T + tx]`. Together the 256 threads copy two whole 16×16 tiles. Look at which index `tx` drives: the *column* of A and the column of B, so both global loads are coalesced. `sA.after(kt)` says "the version of `sA` in iteration `kt`", which puts the store inside the `kt` loop.
- **`sync1 = BARRIER(ld_a, ld_b)`**. Every thread waits until all 512 copies are done.

- **`k = range_(T, 5)` and `inner`** are the ordinary reduce body, reading both operands out of shared memory: `acc += sA[ty*T + k] * sB[k*T + tx]`. The loads go through `sA.after(sync1)`, so they are ordered after the barrier. That data dependency is what places them below the `__syncthreads()` in the linear order.
- **`sync2 = BARRIER(inner)`**. Every thread waits until all threads have finished reading the tiles, and `loop = sync2.end(kt)` closes the K loop.
- **`out`** writes the accumulator to `C[row*N + col]` after the loop (`acc0.after(loop)`), and the `SINK` ends the four grid ranges.

Here is the rendering. It is short enough to read in full:

```
from uopc.gpu import smem_gemm_ir, render_cuda, launch_dims
ast = smem_gemm_ir(64, 16)
print(launch_dims(ast))
print(render_cuda(ast, 'smem_gemm'))
```

```
((4, 4, 1), (16, 16, 1))
```

```
extern "C" __global__ void __launch_bounds__(256) smem_gemm(float* __restrict__ data0, float*
__restrict__ data1, float* __restrict__ data2) {
  __shared__ float smem0[256];
  __shared__ float smem1[256];
  int gidx1 = blockIdx.y; /* 4 */
  int gidx0 = blockIdx.x; /* 4 */
  int lidx1 = threadIdx.y; /* 16 */
  int alu1 = (((gidx1*16)+lidx1)*64);
  int alu2 = (lidx1*16);
  int lidx0 = threadIdx.x; /* 16 */
  int alu3 = ((gidx0*16)+lidx0);
  float acc2;
  acc2 = 0.0f;
  int alu4 = (alu2+lidx0);
  for (int ridx4 = 0; ridx4 < 4; ridx4++) {
    int alu5 = (ridx4*16);
    float val6 = data0[((alu1+alu5)+lidx0)];
    smem0[alu4] = val6;
    float val7 = data1[(((alu5+lidx1)*64)+alu3)];
    smem1[alu4] = val7;
    __syncthreads();
    for (int ridx5 = 0; ridx5 < 16; ridx5++) {
      float val8 = acc2;
      float val9 = smem0[(alu2+ridx5)];
      float val10 = smem1[((ridx5*16)+lidx0)];
      acc2 = (val8+(val9*val10));
    }
    __syncthreads();
  }
  float val11 = acc2;
  data2[(alu1+alu3)] = val11;
}
```

(The `#define` prelude is the same as in Chapter 12 and is omitted.) Every piece of the IR is visible: the two `__shared__` arrays at the top (NOTES 24), the thread's row offset `alu1 = row*64`, its column `alu3`, the `kt` loop (`ridx4`), the copies, the barriers in the right places, and the inner product out of shared memory. The test pins the important properties:

tests/test_w07_gpu.py · test_smem_gemm_renders

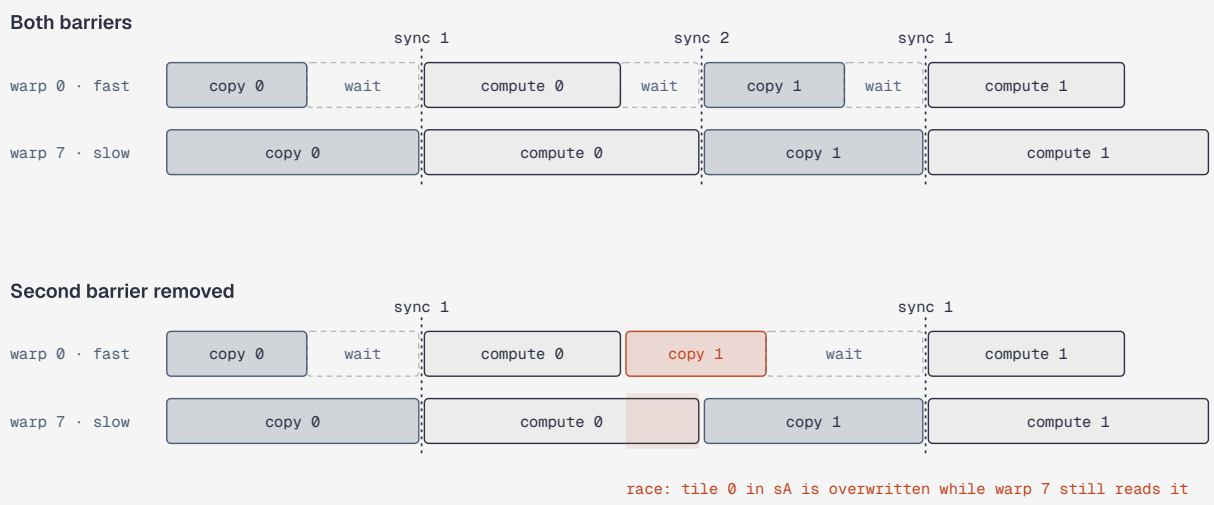
```
def test_smem_gemm_renders():
  src = render_cuda(smem_gemm_ir(64, 16), "smem_gemm")
  assert src.count("__syncthreads();") == 2 and "__shared__ float smem0[256];" in src
  assert launch_dims(smem_gemm_ir(64, 16)) == ((4, 4, 1), (16, 16, 1))
```

Count the traffic for this kernel. Per block and per `kt`, 2 × 256 floats come from global memory and the inner loop performs 256 × 16 FMAs. That is 4096 FMAs per 512 loads, 8 FMAs per load, against 1 per 2 loads in the naive kernel: a factor of T = 16, as the table predicted.

13.4 Why two barriers

Barriers are the part of GPU programming that is easiest to get wrong, so let's be precise about what each one prevents. The warps of a block do not run in lockstep with *each other*. The scheduler interleaves them, and one warp can get well ahead of another.

- **Without** `sync1` a fast warp finishes its copy and starts the inner loop while a slow warp hasn't yet written its part of the tile. The fast warp reads stale shared memory: whatever the previous `kt` left there, or uninitialized garbage on the first iteration. This is a **read-after-write** hazard.
- **Without** `sync2` a fast warp finishes its inner loop, loops around to `kt + 1`, and *overwrites* `sA` and `sB` with the next tiles while a slow warp is still reading the current ones. This is a **write-after-read** hazard.



Sync 1 orders writes before reads (RAW). Sync 2 orders reads before the next overwrite (WAR).

The second barrier is the one people forget, because the code "obviously" reads the tile before the loop comes around again, and in a single thread it does. The comment in `smem_gemm_ir`, *"don't overwrite tiles others still read"*, is there because the hazard can't be seen by reading one thread's code.

Two more rules complete the picture:

1. **Every thread of the block must reach the same barrier.** Putting `__syncthreads()` inside a branch that some threads skip is undefined behaviour, and in practice the kernel often deadlocks. Our IR can't express that mistake, since `BARRIER` has no condition. Exercise 13.7 asks what a guarded store (Chapter 12, Exercise 5) would do near a barrier.
2. **A barrier is per block.** There is no `__syncthreads()` across blocks. Cross-block communication goes through a kernel boundary (or through cooperative groups, which this book does not cover).

The cost of a barrier is the time the fastest warp waits for the slowest. Double buffering (Section 13.9) uses two pairs of tiles so that the copy of tile `kt + 1` overlaps the computation on tile `kt`, and that removes one of the two waits.

13.5 Testing a barrier kernel without a GPU

The Chapter 12 emulator runs threads one at a time, in nested loops. For this kernel the loop order is `by, bx, ty, tx, kt, k`, so thread (0, 0) runs its *whole* `kt` loop before thread (0, 1) starts. When thread (0,0) reads row 0 of `sA` it finds only its own element written. That is exactly the read-after-write hazard above, made deterministic.

First, `emulate` refuses the kernel: it checks for `BARRIER` before rendering anything.

```
from uopc.gpu import smem_gemm_ir, emulate
try: print(emulate(smem_gemm_ir(64,16)))
except Exception as e: print(type(e).__name__+':', e)
```

```
NotImplementedError: emulate: kernels with BARRIER need all threads to run in lockstep; running
threads one after another (what loops do) would be wrong
```

(The first version crashed here with `TypeError: unsupported operand type(s) for +: 'NoneType' and 'str'`, from `self.barrier + ";"` in the renderer. An error should say *what* is unsupported and *why*; this one now does.) Suppose we ignore the warning and force it through by giving the C renderer a no-op barrier:

```python
# scratch/nobarrier.py
import numpy as np
from uopc.render_c import CRenderer
from uopc.runtime import Program
from uopc.gpu import smem_gemm_ir

class SerialThreads(CRenderer):
  barrier = "/* __syncthreads() */"      # threads run one after another: a barrier is a no-op

N = 64
src = SerialThreads().render(smem_gemm_ir(N, 16), "smem_gemm")
a, b = np.random.randn(N, N).astype(np.float32), np.random.randn(N, N).astype(np.float32)
c = np.zeros(N * N, np.float32)
Program(src, "smem_gemm")(a, b, c)
err = np.abs(c.reshape(N, N) - a @ b)
print(f"max error {err.max():.2f}, wrong elements {np.mean(err > 1e-3) * 100:.1f}%")
```

```
max error nan, wrong elements 99.3%
```

Almost every element is wrong, and some are NaN, because they were computed from shared-memory slots that no thread had written yet (uninitialized stack memory in the C version). This is NOTES 25: *barriers cannot be emulated by running threads one after another.*

To check the *algorithm* we need an execution model that respects barriers. The simplest one runs the threads of a block in **lockstep**: every phase between barriers completes for all threads before the next begins. With numpy that comes almost for free, because one array operation over a `(T, T)` grid of thread indices is one phase for all threads:

```python
# scratch/lockstep.py
import numpy as np

def smem_gemm_lockstep(A, B, T=16):
  """Run the smem_gemm schedule with every thread of a block in lockstep: each phase
  between barriers finishes for all T*T threads before the next one starts."""
  N = A.shape[0]
  C = np.zeros((N, N), np.float32)
  ty, tx = np.meshgrid(np.arange(T), np.arange(T), indexing="ij")   # all threads of a block at
once
  for by in range(N // T):
    for bx in range(N // T):
      row, col = by * T + ty, bx * T + tx
      acc = np.zeros((T, T), np.float32)                              # one register per thread
      sA, sB = np.empty(T * T, np.float32), np.empty(T * T, np.float32)
      for kt in range(N // T):
        sA[ty * T + tx] = A[row, kt * T + tx]                         # phase 1: cooperative copy
        sB[ty * T + tx] = B[kt * T + ty, col]
        # __syncthreads()
        for k in range(T):                                            # phase 2: multiply from
smem
          acc += sA[ty * T + k] * sB[k * T + tx]
        # __syncthreads()
      C[row, col] = acc
  return C

N = 64
A, B = np.random.randn(N, N).astype(np.float32), np.random.randn(N, N).astype(np.float32)
print("max error:", np.abs(smem_gemm_lockstep(A, B) - A @ B).max())
```

```
max error: 7.6293945e-06
```

The index expressions are copied from the IR (`row*N + kt*T + tx` and so on). This checks the tiling algorithm and its index math, not the generated code. The generated code is checked by rendering tests, and by `test_cuda_smem_gemm_on_gpu` on a machine with a GPU (Chapter 12, Section 12.11).

A lockstep simulation is a sound model for kernels whose barriers are all reached by all threads, which our IR guarantees. It can be made fully automatic. Split the linearized kernel at its barriers into phases, then emulate each phase as a loop over all threads, with per-thread registers kept in arrays indexed by thread. That is Exercise 13.6. It is also the core idea of MCUDA (Stratton, Stone and Hwu, 2008), which compiled CUDA for CPUs by wrapping each barrier-free region of a kernel in a "thread loop".

13.6 Bank conflicts

Shared memory is not a single array. It is **32 banks**, each 4 bytes wide, with word w in bank $w \bmod 32$. Each bank can serve one word per cycle. When the 32 lanes of a warp access shared memory:

- distinct words in distinct banks are served in one pass;
- lanes reading *the same word* get a broadcast, also one pass;
- *distinct* words in the *same* bank are serialized, one pass per word. This is a **bank conflict**, and an n-way conflict makes the access n times slower.

These rules are general knowledge (CUDA Programming Guide, "Shared Memory"). Let's apply them to our kernel's three shared-memory accesses, and to the classic bad case:

```python
# scratch/banks.py
def conflict(addr_of_thread, nthreads=32):
  """Shared memory: 32 banks of 4 bytes. Threads reading the *same* word are a broadcast (free);
  distinct words in the same bank are serialized. Returns the worst-case number of passes."""
  words = {addr_of_thread(t) for t in range(nthreads)}
  per_bank = {}
  for w in words: per_bank.setdefault(w % 32, set()).add(w)
  return max(len(s) for s in per_bank.values())

T, k = 16, 3
tid = lambda t: (t // T, t % T)                    # (ty, tx) of thread t in a 16x16 block
print("smem_gemm, T=16 (warp = 2 rows of 16 threads)")
print("  store sA[ty*T+tx]  :", conflict(lambda t: tid(t)[0] * T + tid(t)[1]), "pass(es)")
print("  load  sA[ty*T+k]   :", conflict(lambda t: tid(t)[0] * T + k), "pass(es)")
print("  load  sB[k*T+tx]   :", conflict(lambda t: k * T + tid(t)[1]), "pass(es)")
T = 32
tid = lambda t: (t // T, t % T)                    # T=32: a warp is one row of threads
print("column walk, T=32")
print("  load  s[tx*32+k]   :", conflict(lambda t: tid(t)[1] * 32 + k), "pass(es)")
print("  load  s[tx*33+k]   :", conflict(lambda t: tid(t)[1] * 33 + k), "pass(es)  (padded row)")
```

```
smem_gemm, T=16 (warp = 2 rows of 16 threads)
  store sA[ty*T+tx]  : 1 pass(es)
  load  sA[ty*T+k]   : 1 pass(es)
  load  sB[k*T+tx]   : 1 pass(es)
column walk, T=32
  load  s[tx*32+k]   : 32 pass(es)
  load  s[tx*33+k]   : 1 pass(es)  (padded row)
```

Our kernel is conflict-free. The copy writes 32 consecutive words. The `sA` read is two distinct words, one per row of the warp, so two broadcasts in different banks. The `sB` read is 16 consecutive words, the same 16 for both halves of the warp.

The bad case appears as soon as a thread walks *down a column* of a tile whose row length is a multiple of 32. That happens, for example, if you store a tile transposed so that the inner loop reads `s[tx*32 + k]`. Every lane hits bank `k mod 32`: a 32-way conflict, and shared memory becomes slower than L2. There are two standard fixes:

- **Padding.** Allocate rows of 33 floats instead of 32. Element `(r, c)` lives at `r*33 + c`, so a column walk moves one bank per row. One wasted word per row buys conflict-free columns.
- **Swizzling.** Store element `(r, c)` at column `c XOR (r mod 32)` (or a similar permutation). No memory is wasted, and the layout composes with vectorized 16-byte accesses. CUTLASS and Triton swizzle. It's a natural fit for our IR, since the swizzle is just an index expression on the `INDEX` into a `SHARED` allocation (Exercise 13.5).

13.7 Register blocking on top of shared memory

Shared memory removed most of the DRAM traffic. Now the shared-memory traffic is the bottleneck. In `smem_gemm_ir` each FMA needs two shared-memory loads. An SM can serve 32 banks × 4 bytes = 128 bytes of shared memory per cycle, and an A100 SM can issue 64 FP32 FMAs per cycle (general knowledge, whitepaper numbers). At 2 loads × 4 bytes per FMA, keeping 64 FMAs per cycle busy needs 512 bytes per cycle, four times what shared memory delivers.

The fix is the *same* trick as Chapter 10's register tile, one level down the hierarchy. Let each thread compute a TM × TN micro-tile of C. Per step of `k` it loads TM values of `sA` and TN values of `sB` into registers and does TM·TN FMAs:

micro-tile per thread	shared loads per FMA	bytes/cycle to feed 64 FMA/cycle
1 × 1 (`smem_gemm_ir`)	2	512
2 × 2	1	256
4 × 4	0.5	128
8 × 8	0.25	64

So a 4×4 micro-tile is the break-even point and 8×8 leaves headroom. The full hierarchy of a classic FP32 GEMM kernel (the structure of CUTLASS's SIMT kernels and of many "how to write a fast matmul" tutorials) looks like this:

```
block tile    128 x 128 of C       global -> shared, per k-step of 8..32
  warp tile    64 x 32 of C        (8 warps per block)
    thread tile  8 x 8 of C        shared -> registers, 64 accumulators per thread
```

256 threads × 64 outputs = 16,384 = 128 × 128. Every level multiplies reuse: DRAM bytes are reused 128× through shared memory, and shared bytes are reused 8× through registers.

In `uopc` the register level already exists. Chapter 12's `gpu_opt_choice` emits a 2×2 `UPCAST` on GPU kernels, and `Opts(gpu=True, local=..., upcast=...)` can request any size. What the compiler cannot yet do is combine an upcast with a shared-memory stage automatically. Writing `smem_gemm_ir` with a 2×2 or 4×4 thread tile by hand (Exercise 13.3) is the most instructive thing you can do this week. The pieces are all there: four accumulator allocations, a cooperative copy where each thread copies several elements, and the unrolled inner product.

13.8 Tensor cores

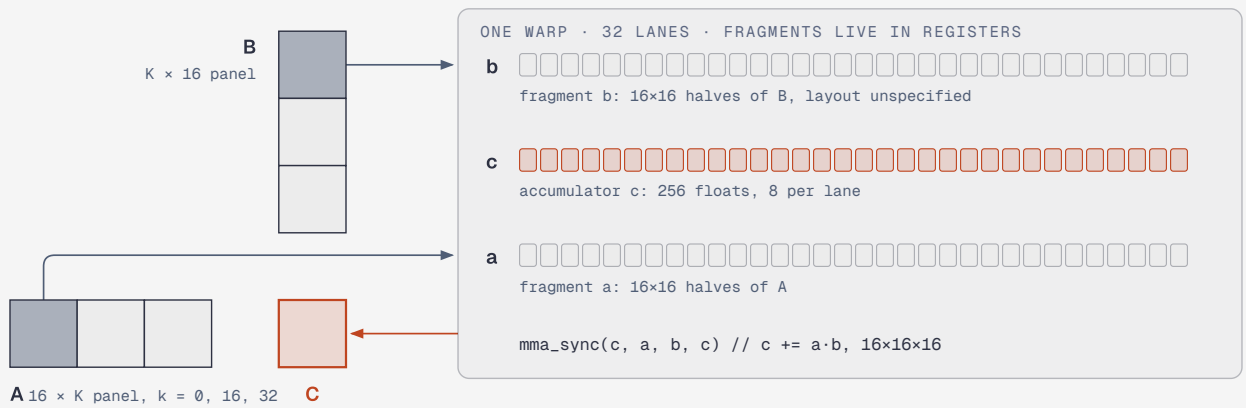
Since Volta (V100, 2017), NVIDIA GPUs have **tensor cores**: units that perform a small matrix multiply-accumulate, `D = A·B + C`, as a single operation, with low-precision inputs and higher-precision accumulation. The throughput difference is large. On an A100, dense FP16 or BF16 tensor-core math peaks at 312 TFLOP/s, against 19.5 TFLOP/s for ordinary FP32 FMAs (whitepaper numbers). Any serious GEMM or attention kernel on such hardware runs on tensor cores, and so does essentially all LLM training.

Three facts define the programming model:

1. **The unit of work is a warp, not a thread.** All 32 lanes jointly issue a matrix operation on tiles that are spread across their registers.
2. **The shapes are fixed.** The C++ WMMA API (`nvcuda::wmma`, Volta and later) offers m16n16k16 (plus m32n8k16 and m8n32k16) for FP16 inputs. The lower-level PTX `mma.sync` instructions expose shapes such as m16n8k16 on Ampere. Hopper's `wgmma` operates on m64nNk16 tiles across a warp *group* of four warps.

3. **Precision is mixed by design.** Inputs are FP16 (5 exponent bits, 10 mantissa bits, max 65504) or BF16 (8 exponent bits like FP32, 7 mantissa bits). Products are formed exactly and accumulated in FP32. On A100, TF32 (FP32 range, 10-bit mantissa) is a third option for FP32 inputs.

Fragments. The WMMA API hides the register layout behind a type, `wmma::fragment<use, M, N, K, T, layout>`. A `matrix_a` fragment holds a 16×16 tile of A, and an `accumulator` fragment holds a 16×16 tile of C as 256 floats across 32 lanes, 8 per lane. *Which* lane holds *which* element is deliberately unspecified and differs between architectures. Your code may only move whole fragments: `load_matrix_sync` (from memory, with a leading dimension), `mma_sync` (multiply-accumulate), `store_matrix_sync`, `fill_fragment`. The `_sync` suffix is a reminder that every lane of the warp must call them together.



The warp, not the thread, is the unit: all 32 lanes call every wmma function together.

Why FP32 accumulation matters

The precision claim is easy to check without a tensor core. Here is a numpy model of the kernel's tile loop, with FP16 inputs, products formed exactly, and FP32 accumulation, compared against an FP16 accumulator:

```python
# scratch/wmma_sim.py
import numpy as np
def wmma_sim(A, B):
  """The tile loop of wmma_gemm in numpy: fp16 inputs, one 16x16x16 MMA with fp32 accumulate per step."""
  M, K = A.shape; N = B.shape[1]
  C = np.zeros((M, N), np.float32)
  for tile_m in range(M // 16):                    # blockIdx.y
    for tile_n in range(N // 16):                  # blockIdx.x
      c = np.zeros((16, 16), np.float32)           # fill_fragment(c, 0)
      for k in range(0, K, 16):
        a = A[tile_m*16:tile_m*16+16, k:k+16]      # load_matrix_sync(a, A + tile_m*16*K + k, K)
        b = B[k:k+16, tile_n*16:tile_n*16+16]      # load_matrix_sync(b, B + k*N + tile_n*16, N)
        c += a.astype(np.float32) @ b.astype(np.float32)   # mma_sync: products exact in f32, f32 sum
      C[tile_m*16:tile_m*16+16, tile_n*16:tile_n*16+16] = c
  return C
rng = np.random.default_rng(0)
M, N, K = 64, 32, 48
A32, B32 = rng.standard_normal((M, K)).astype(np.float32), rng.standard_normal((K, N)).astype(np.float32)
A, B = A32.astype(np.float16), B32.astype(np.float16)
ref = A32.astype(np.float64) @ B32.astype(np.float64)
print("wmma (f16 in, f32 acc) vs f64 of the f32 inputs: max abs err %.4f" % np.abs(wmma_sim(A, B) - ref).max())
print("same f16 inputs, exact f64 math               : max abs err %.6f" % np.abs(wmma_sim(A, B) - A.astype(np.float64) @ B.astype(np.float64)).max())
c16 = np.zeros((M, N), np.float16)
for k in range(K): c16 = (c16 + (A[:, k:k+1] * B[k:k+1, :])).astype(np.float16)
print("f16 inputs AND f16 accumulate                 : max abs err %.4f" % np.abs(c16 - ref).max())
```

```
wmma (f16 in, f32 acc) vs f64 of the f32 inputs: max abs err 0.0076
same f16 inputs, exact f64 math               : max abs err 0.000003
f16 inputs AND f16 accumulate                 : max abs err 0.0330
```

Read the three lines together. With FP32 accumulation, essentially all the error comes from *rounding the inputs* to FP16 (0.0076 against the FP32 data), and almost none from the arithmetic (0.000003 against exact math on the same FP16 inputs). Accumulating in FP16 adds four times more error even at K = 48, and the error grows with K. That is why tensor cores accumulate in FP32, and why mixed-precision training keeps master weights and reductions in FP32 (Chapter 15).

13.9 Pattern-based lowering: `match_gemm` and `wmma_gemm`

How does a compiler get from `A @ B` in a Tensor program to `mma_sync`? There are two broad designs.

1. **Make tensor cores an IR operation.** Add a `WMMA` UOp whose sources are fragments, and teach the optimizer to *reshape* the loop nest so that a 16×16×16 sub-block of the GEMM becomes one `WMMA`, with the remaining axes as ordinary `GLOBAL` / `LOCAL` / `LOOP` ranges. This is what tinygrad does: a tensor-core "opt" splits the M, N and K axes by the MMA shape and the renderer emits the intrinsic. It composes with everything else (fusion, upcasting, other ops in the same kernel), and it is a lot of machinery.
2. **Recognize the pattern and call a template.** Look at the shaped graph *before* rangeify. If a kernel is exactly a GEMM, emit a hand-written kernel for it. This is how many production compilers dispatch to cuBLAS or CUTLASS, and how `uopc` does it.

`uopc` reserves the `WMMA` op (`Ops.WMMA`, with a shape rule in `_infer` and a `render_wmma` hook in the C renderer that raises) but takes route 2 for the course:

uopc/gpu.py · match_gemm

```
def match_gemm(body: UOp) -> tuple[int, int, int] | None:
  """If body is (M,N) = sum_K A(M,K) * B(K,N) over contiguous f16/f32 buffers, return (M, N,
K)."""
  if body.op is not Ops.REDUCE or body.arg != (Ops.ADD, 1) or body.src[0].op is not Ops.MUL:
return None
  if len(body.src[0].shape) != 3 or not body.dtype.is_float: return None
  M, N, K = body.src[0].shape
  # the two operands must be broadcast along *different* axes: A over N (axis 1), B over M (axis
0).
  # Without this check, x*x reduced over its last axis (a sum of squares) "is" a GEMM.
  a, b = body.src[0].src
  def bcast_axes(x: UOp) -> set[int]:
    return {i for i, (s, t) in enumerate(zip(x.src[0].shape, x.shape)) if s == 1 and t > 1} if
x.op is Ops.EXPAND else set()
  if {frozenset(bcast_axes(a)), frozenset(bcast_axes(b))} != {frozenset({1}), frozenset({0})}:
return None
  return (M, N, K) if all(x % 16 == 0 for x in (M, N, K)) else None
```

A GEMM in our shaped IR has a canonical form, the one `Tensor.matmul` builds (Chapter 9): a `REDUCE(ADD, 1)` over a `MUL` of two operands broadcast to `(M, N, K)`. `match_gemm` checks for that form and for sizes that are multiples of 16:

```python
# scratch/wmma.py
import sys; sys.path.insert(0, "tests")
from helpers import buf
from uopc.uop import Ops
from uopc.gpu import match_gemm, wmma_gemm
def gemm_graph(M, N, K):
  A, B = buf(0, M * K), buf(1, K * N)
  return (A.reshape((M, 1, K)).expand((M, N, K)) * B.reshape((K, N)).permute((1, 0)).reshape((1, N, K)).expand((M, N, K))).reduce(Ops.ADD, 1)
print("gemm 64x32x48 :", match_gemm(gemm_graph(64, 32, 48)))
print("gemm 64x32x40 :", match_gemm(gemm_graph(64, 32, 40)))
X = buf(0, 64 * 64).reshape((64, 64))
print("row sum       :", match_gemm(X.reduce(Ops.ADD, 1)))
print("row max       :", match_gemm(X.reduce(Ops.MAX, 1)))
print(wmma_gemm(64, 32, 48))
```

```
gemm 64x32x48 : (64, 32, 48)
gemm 64x32x40 : None
row sum       : None
row max       : None
```

When it matches, `wmma_gemm` produces the kernel. One warp computes one 16×16 tile of C by walking K in steps of 16:

uopc/gpu.py · wmma_gemm

```python
def wmma_gemm(M: int, N: int, K: int) -> str:
  """C(f32, MxN) = A(f16, MxK) @ B(f16, KxN), one 16x16 output tile per warp, via nvcuda::wmma.

  A warp (32 threads) cooperatively owns three "fragments": 16x16 tiles of A, B and C
  spread across its registers in a hardware-defined layout. mma_sync does a
  16x16x16 matrix multiply-accumulate in one instruction on the tensor cores.
  """
  return f"""#include <mma.h>
using namespace nvcuda;
extern "C" __global__ void wmma_gemm(const half* __restrict__ A, const half* __restrict__ B,
float* __restrict__ C) {{
  int tile_m = blockIdx.y, tile_n = blockIdx.x;          // one warp per 16x16 tile of C
  wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a;
  wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b;
  wmma::fragment<wmma::accumulator, 16, 16, 16, float> c;
  wmma::fill_fragment(c, 0.0f);
  for (int k = 0; k < {K}; k += 16) {{
    wmma::load_matrix_sync(a, A + tile_m * 16 * {K} + k, {K});
    wmma::load_matrix_sync(b, B + k * {N} + tile_n * 16, {N});
    wmma::mma_sync(c, a, b, c);
  }}
  wmma::store_matrix_sync(C + tile_m * 16 * {N} + tile_n * 16, c, {N}, wmma::mem_row_major);
}}
/* launch: grid = ({N // 16}, {M // 16}), block = (32, 1, 1) */
"""
```

For M, N, K = 64, 32, 48 the rendered source is:

```cpp
#include <mma.h>
using namespace nvcuda;
extern "C" __global__ void wmma_gemm(const half* __restrict__ A, const half* __restrict__ B,
float* __restrict__ C) {
  int tile_m = blockIdx.y, tile_n = blockIdx.x;          // one warp per 16x16 tile of C
  wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a;
  wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b;
  wmma::fragment<wmma::accumulator, 16, 16, 16, float> c;
  wmma::fill_fragment(c, 0.0f);
  for (int k = 0; k < 48; k += 16) {
    wmma::load_matrix_sync(a, A + tile_m * 16 * 48 + k, 48);
    wmma::load_matrix_sync(b, B + k * 32 + tile_n * 16, 32);
    wmma::mma_sync(c, a, b, c);
  }
  wmma::store_matrix_sync(C + tile_m * 16 * 32 + tile_n * 16, c, 32, wmma::mem_row_major);
}
/* launch: grid = (2, 4), block = (32, 1, 1) */
```

Read it against the diagram. Block `(tile_n, tile_m)` is one warp, `block = (32, 1, 1)`. The A fragment starts at row `tile_m*16`, column `k`, with leading dimension K. The B fragment starts at row `k`, column

`tile_n*16`, with leading dimension N. Three `mma_sync`s accumulate K = 48. The `wmma_sim` function in Section 13.8 is a line-by-line numpy transcription of this loop (its comments name the CUDA calls), and it matches the FP64 reference to within FP16 input rounding. That checks the index arithmetic. The CUDA itself is render-tested only (`test_wmma`), since the reference machine has no GPU.

This kernel is a teaching kernel, not a fast one. It uses one warp per block, so a block has 32 threads (the "too-small block" problem of Chapter 12). It has no shared-memory staging, so each warp reloads its A and B fragments straight from global memory and there is no reuse across warps. And it has one accumulator per warp, so there is no ILP. A competitive tensor-core kernel combines everything in this chapter: a block of 4–8 warps, a 128×128 block tile staged through (swizzled) shared memory, each warp owning a 64×64 warp tile of 4×4 or more accumulator fragments, and double-buffered asynchronous copies.

The pattern had a hole. The first `match_gemm` checked only the *shape* of the graph, `REDUCE(ADD)` of a 3-D `MUL`, not what was being multiplied, so any 3-D product reduced over its last axis matched, including `(X*X).reduce(Ops.ADD, 1)`, a sum of squares. Dispatching that to `wmma_gemm` would compute the wrong thing. The current version also requires the two operands to be `EXPAND`s broadcast along *different* axes (A along N, B along M), which is what makes a product of two tensors a matrix product:

```
X = buf(0, 32*32*32).reshape((32,32,32))
print('sum of squares over last axis:', match_gemm((X*X).reduce(Ops.ADD, 1)))
```

```
sum of squares over last axis: None
```

It is still not fully sound: a production matcher must record the operands' layouts (`wmma_gemm` assumes A row-major M×K and B row-major K×N) and dtypes (it assumes FP16 inputs). That is Exercise 13.2. This is the general weakness of pattern-based lowering: the pattern is a *claim* about semantics, and every claim needs a test that tries to break it.

13.10 What production kernels add

The gap between `smem_gemm_ir` and cuBLAS is mostly made of the following techniques, all general knowledge and all expressible as more of what this chapter did:

- **Double buffering / multi-stage pipelines.** Allocate two (or 3–4) sets of shared tiles. While computing on set *i*, copy the next tile into set *i+1*. On Ampere and later, `cp.async` copies global memory to shared

memory *asynchronously*, without passing through registers, so the copy and the math truly overlap. Hopper's TMA (Tensor Memory Accelerator) moves whole tiles with one instruction.

- **Vectorized global loads.** `float4` loads move 16 bytes per lane per instruction, so fewer instructions are needed to keep enough bytes in flight (the Little's-law argument of Chapter 12).
- **Swizzled shared layouts**, as in Section 13.6, chosen to match the fragment load pattern of `ldmatrix`.
- **Warp tiling** between the block and thread levels, so that each warp's fragments come from a compact region of shared memory.
- **Split-K and stream-K** for shapes with small M·N and large K, where there are too few output tiles to fill 108 SMs. Several blocks share one output tile and reduce their partial sums at the end.
- **Epilogue fusion.** Apply bias, activation, residual add or a type cast to the accumulator before the store, saving a whole kernel and its memory round trip. This is Chapter 7's fusion at the kernel template level.
- **Autotuning** over tile sizes, stages and warp counts. Chapter 10's `search_opt` is the same idea.

FlashAttention-2 (Chapter 11) is this chapter's GEMM structure applied twice per tile, `S = Q·Kᵀ` and `O += P·V` on tensor cores, with the online softmax in registers between them.

Pitfalls

1. The forgotten second barrier. Without `sync2`, the kernel produces correct results *most of the time*. Whether a fast warp overwrites a tile before a slow warp has read it depends on scheduling, clock speeds and what else the GPU is doing. Races like this pass a test 99 times and fail on the 100th, or only on a different GPU. Reason about every shared buffer's write-after-read and read-after-write orderings explicitly (Section 13.4), because the emulator can't help you (Pitfall 3).

2. Shared allocations in the wrong scope (NOTES 24). The generic rule "declare an `ALLOC` in the deepest scope common to its users" is right for accumulators and wrong for shared memory. Applied to `sA`, it would place the declaration inside the thread ranges. In CUDA that would still be one array per block, but the IR would claim per-thread storage, and every pass that reasons about scope (for example, the emulator, which would re-declare it on every iteration) would get it wrong. The linearizer pins `SHARED` allocations to the empty scope.

3. The emulator lies about barriers (NOTES 25). `emulate` now refuses barrier kernels, and if you force one through with a no-op barrier you get the 99.3%-wrong result of Section 13.5. Test barrier kernels with a lockstep model or on real hardware.

4. A matcher that matched things that aren't GEMMs. The first `match_gemm` matched the shape of the graph, not the operands, and accepted a sum of squares (13.9). A pattern-based lowering is only as sound as its matcher, and every matcher needs a test that tries to fool it.

5. Bank conflicts from "harmless" layout changes. Transposing a shared tile so that the inner loop walks columns of a 32-wide tile makes every access a 32-way conflict (Section 13.6). Results are unchanged and the kernel gets several times slower. Pad or swizzle.

6. FP16 overflow and accumulation. FP16's largest finite value is 65504, so activations or gradients can overflow to `inf` where FP32 would not. And accumulating in FP16 loses accuracy quickly with K (Section 13.8). Use FP32 accumulators (the WMMA `accumulator` fragment type above is `float`), BF16 when range matters, and loss scaling when training in FP16.

7. Fragment layouts are opaque. It is tempting to index into `c.x[i]` and assume element `i` is `C[lane/4][...]`. The mapping is unspecified and has changed between GPU generations. Element-wise operations over all fragment elements (scaling, activation) are safe. Anything that depends on position is not.

Exercises

1. ★ **Traffic by hand.** For `smem_gemm_ir(N=1024, T=16)`, compute (a) the number of global loads per block, (b) the total global loads, (c) the ratio to the naive kernel's $2N^3$. Then do it for T = 32. Is T = 32 even possible with one thread per output? *Hint:* count threads per block.
2. ★★ **Make `match_gemm` sound.** Strengthen it to check that the two operands of the `MUL` are an `(M, 1, K)` and a `(1, N, K)` broadcast, each from a single buffer, and return the operands' layouts (whether K is contiguous). Add a test with the sum-of-squares counterexample of 13.9 and with a transposed B. *Hint:* reuse `_last_axis_stride` from `uopc/opt.py` (Chapter 10) to compute strides.
3. ★★ **Register-blocked shared-memory GEMM.** Write `smem_gemm_ir2(N, T, R)` in which a block of (T/R)×(T/R) threads computes a T×T tile, each thread an R×R micro-tile with R^2 accumulators. Each thread now copies R^2 elements of each tile. Render it with `render_cuda`, check it with a lockstep simulation, and fill in the "shared loads per FMA" column of 13.7 for your kernel.
4. ★★ **Predict the conflicts.** For T = 32 and the original `smem_gemm_ir` indexing (`sA[ty*T+k]`, `sB[k*T+tx]`), how many passes does each shared load take? What changes if the block is (32, 8) threads with each thread computing 4 rows? Check your predictions with `banks.py`.
5. ★★ **Swizzle in the IR.** Change `smem_gemm_ir` to store and load `sB` at a swizzled index (`k*T + (tx ^ k)`, written with the IR's integer ops, or `(tx + k) % T` if you don't want to add XOR) and verify with the

lockstep model that results are unchanged. Explain why a swizzle must be applied to both the store and the load.

6. ★★★ **A barrier-aware emulator.** Write `emulate_lockstep(sink)`: linearize the kernel, split the instruction list at `BARRIER`s, and generate C in which each phase is a loop over all threads of a block, with every per-thread value that is live across a barrier (accumulators, indices) stored in an array indexed by thread id. Shared allocations stay scalar arrays per block. Make `smem_gemm_ir(64, 16)` pass against numpy. *Hint:* barriers inside a loop (`kt`) mean the phase split must happen *inside* the loop body. Start by supporting only barriers at loop-body depth 1.
7. ★★ **Barriers and guards.** Suppose you implemented guarded stores (Chapter 12, Exercise 5) by wrapping whole statements in `if (idx < n)`. Construct a shared-memory kernel where that transformation would place a `__syncthreads()` inside the `if`. What goes wrong, and how should the guard be expressed instead?
8. ★★★ **A WMMA UOp.** Implement route 1 of Section 13.9 for one case. Lower a `match_gemm`-able kernel with `GLOBAL` ranges over 16×16 output tiles and a `LOOP` over K/16, and an `Ops.WMMA` node whose sources are the A and B tile addresses and the accumulator. Implement `CUDARenderer.render_wmma` to emit the fragment calls. Compare your output with `wmma_gemm`'s.
9. ★★★ **Double buffering.** Write `smem_gemm_ir_db` with two copies of each tile (allocation slots 0–3), in which iteration `kt` computes on buffer `kt % 2` while copying tile `kt+1` into the other. How many barriers per iteration do you need now? Check it with your Exercise 6 emulator if you built it, or else with a lockstep numpy model.

Solutions to ★ exercises

1. (a) Per `kt` a block loads $2 \cdot T^2 = 512$ floats, and there are $N/T = 64$ values of `kt`, so 32,768 loads per block. (b) There are $(N/T)^2 = 4096$ blocks, so $4096 \times 32{,}768 = 134{,}217{,}728 \approx 1.34 \cdot 10^8$ loads. (c) The naive kernel does $2N^3 = 2.15 \cdot 10^9$, so the ratio is $1/16 = 1/T$. For $T = 32$ the ratio is 1/32. But a 32×32 tile with one thread per output needs 1024 threads per block, the hardware maximum, which leaves almost no registers per thread (64K / 1024 = 64) and a single block per SM. This is why real kernels use smaller thread blocks with register micro-tiles (Section 13.7) instead of one thread per output.

Quiz

1. What is the arithmetic intensity, in FLOP per byte, of a GEMM computed with T×T shared-memory tiles and no register blocking?

2. Why does the linearizer put `SHARED` allocations in the empty scope rather than in the deepest scope of their users?
3. Name the hazard each of the two barriers in `smem_gemm_ir` prevents.
4. Why can't `emulate` check `smem_gemm_ir`, and what kind of simulation can?
5. Thirty-two lanes read `s[lane * 32]` from shared memory. How many passes does that take, and what is the cheapest fix?
6. In `wmma::fragment<wmma::accumulator, 16, 16, 16, float>`, how many floats does each lane hold?
7. Why do tensor cores accumulate in FP32 even when their inputs are FP16?
8. What does `match_gemm` fail to check?

Answers

1. T/4. Each block reads 2·T·N floats (8TN bytes) for 2·T²·N FLOPs.
2. Shared memory is one array per block, visible to all threads. Its users sit inside the thread (`LOCAL`) ranges, so the "deepest common scope" rule would make it look per-thread. Pinning it to the top states its real lifetime and is where CUDA code conventionally declares it (NOTES 24).
3. `sync1`: read-after-write (computing before the tile is fully copied). `sync2`: write-after-read (the next iteration's copy overwriting a tile others still read).
4. The emulator runs threads one after another, so no thread ever waits for the others at a barrier, and reads see unwritten shared memory. A lockstep (phase-by-phase) simulation, which runs every thread up to the barrier before any thread continues, models barriers correctly.
5. Every address is in bank 0 with a distinct word: a 32-way conflict, 32 passes. Pad the row to 33 words (or swizzle).
6. 256 / 32 = 8.
7. FP16 has a 10-bit mantissa. Summing many products in FP16 accumulates rounding error that grows with K, whereas the products themselves are exact in FP32. In Section 13.8, FP16 accumulation had about 4× the error at K = 48.
8. That the operands are really an `(M,1,K)` and a `(1,N,K)` broadcast of two matrices (any 3-D product matches), their memory layouts, and their dtypes.

Further reading

- NVIDIA, *CUDA C++ Programming Guide*: "Shared Memory" (banks, `__syncthreads`) and "Warp Matrix Functions" (WMMA). *PTX ISA*: `mma.sync`, `ldmatrix`, `cp.async`.

- NVIDIA, *Volta* (V100) and *Ampere* (A100) architecture whitepapers: tensor core shapes and throughput.
- NVIDIA, *CUTLASS* documentation, especially "Efficient GEMM in CUDA" (the block/warp/thread tile hierarchy).
- S. Boehm, "How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog" (2022), which builds a GEMM kernel step by step through the ideas in 13.1–13.7.
- V. Volkov and J. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra", SC 2008.
- T. Dao, "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning" (2023).
- P. Micikevicius et al., "Mixed Precision Training", ICLR 2018: FP16 storage, FP32 master weights, loss scaling.
- tinygrad: the tensor-core support in its optimizer (`tensor_cores` in the renderer, and the TC opt that reshapes axes to the MMA shape), route 1 of Section 13.9.
- P. Tillet et al., "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations", MAPL 2019: a compiler whose unit of programming is exactly the block tile of this chapter.

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

```
uopc/gpu.py · 181 lines
```

14. Training a GPT

This is the chapter the course has been building toward. Every piece so far (UOps, rewriting, rangeify, kernelization, autodiff, the layout pass, register tiles) was tested on small, isolated programs. A transformer language model is the first workload that uses *all* of them at once, at a size where a bad decision anywhere costs real time. We will write a GPT from scratch in about a hundred lines of Tensor code (`examples/gpt.py`), follow its shapes through every layer, train it on Tiny Shakespeare until it writes something that looks like a play, and then dissect what the compiler did with it: 243 kernels per training step, a structural cache that compiles them once, and a performance history in which a first step went from 64 seconds to about 9, and a steady-state step from 1.2 seconds to about 0.24.

What you'll build

- A character-level GPT: token embeddings as a one-hot matmul, learned positions, causal multi-head self-attention with a mask, pre-LayerNorm residual blocks, a GELU MLP, a linear head, cross-entropy loss.
- AdamW whose per-step constants don't break the compile cache.
- A training loop with validation and a sampler with a fixed shape.
- A profiling method (DEBUG levels, per-kernel timings, `cProfile`, ablations) and the knowledge of which compiler decisions mattered for this model, with numbers.

Where this fits. Weeks 9–10. The model uses the Tensor API from Chapter 9, autodiff from Chapter 8, kernelization from Chapter 7 and the CPU optimizations from Chapter 10. Chapter 15 takes it further: symbolic shapes, KV caches and mixed precision. All numbers in this chapter come from the reference laptop (i7-13620H, see `book/outputs/machine.txt`) running on the CPU. Chapter 12 explains why the GPU backend isn't used here, and Exercise 14.10 is about closing that gap.

14.1 The task: next-character prediction

The data is Tiny Shakespeare, about 1.1 million characters of plays, the classic char-rnn dataset. It is downloaded once and cached:

`examples/datasets.py · shakespeare`

```python
def shakespeare() -> str:
  return fetch(SHAKESPEARE_URL, "tinyshakespeare.txt").read_text()
```

A **character-level language model** reads a window of T characters and predicts, at *every* position, the next character. Each character is an integer in `[0, V)` with V = 65 distinct characters. A training example is a pair of windows offset by one:

```
x = "First Citizen:\nBefore we proce"     (T characters)
y = "irst Citizen:\nBefore we procee"     (the same, shifted left by one)
```

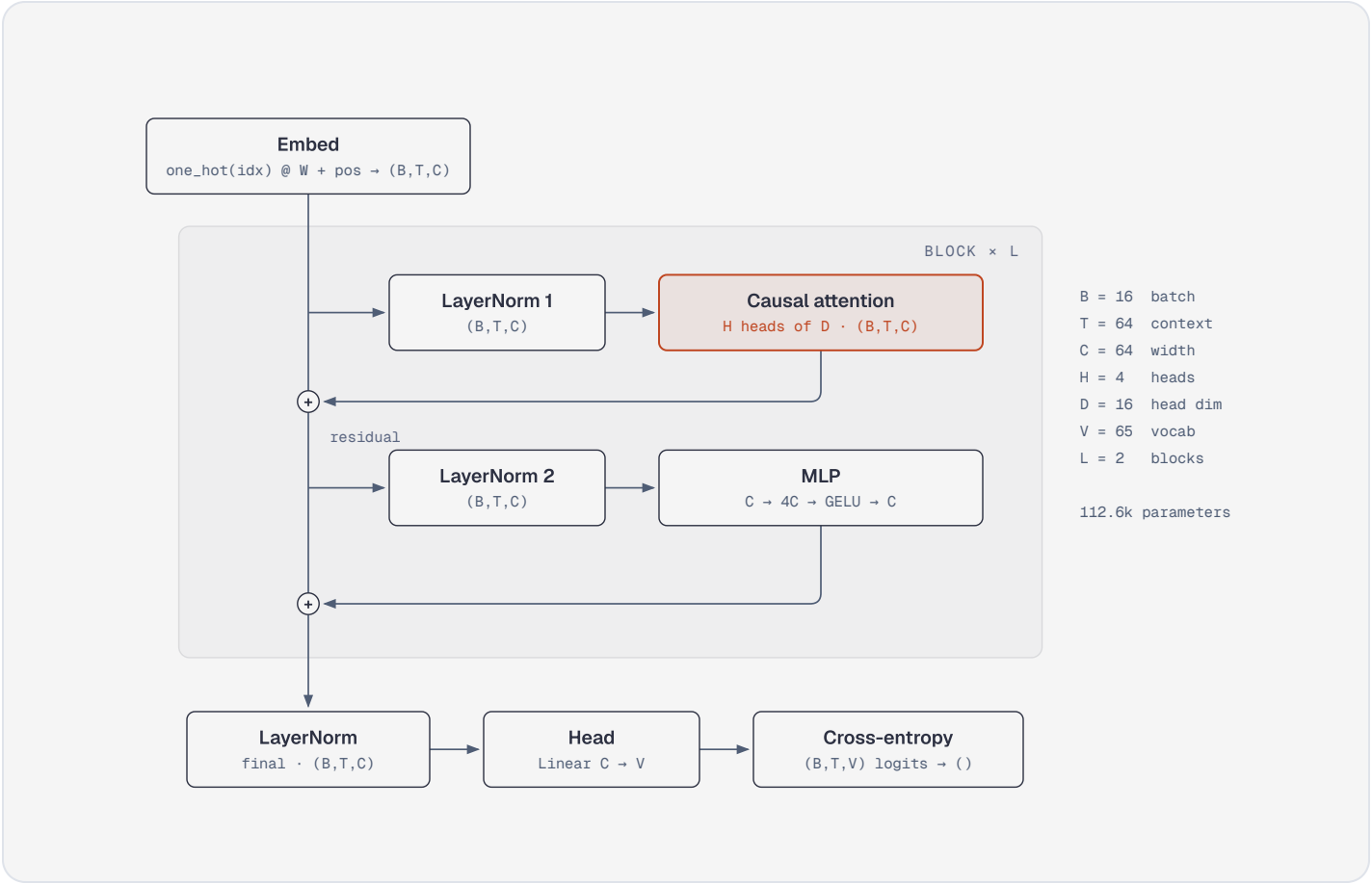
so that position *t* of the output is trained to predict `y[t] = x[t+1]` from `x[0..t]`. One window yields T training signals, and the model must never look at `x[t+1..]` when predicting position *t*. That is what the causal mask enforces.

The loss is the mean **cross-entropy**, the negative log-probability that the model assigns to the right next character. A model that knows nothing assigns 1/65 to each character, and $-\ln(1/65) = \ln 65 \approx$ **4.174**. That's the number to compare the first logged loss against.

14.2 The model at a glance, with every shape

Hyper-parameters of the default run (`examples/gpt.py` with no flags):

symbol	meaning	value
B	batch size (windows per step)	16
T	context length	64
C	model width (embedding dim)	64
H	attention heads	4
D	head dim, C / H	16
V	vocabulary (characters)	65
L	transformer blocks	2



The whole forward pass, with the shape after each step:

step	operation	shape
tokens	integer ids	(B, T) = (16, 64) i32
embed	`one_hot(idx) @ W_tok + pos[:T]`	(16, 64, 64)
block × 2	`x = x + attn(ln1(x))` ; `x = x + mlp(ln2(x))`	(16, 64, 64)
final norm	`ln(x)`	(16, 64, 64)
head	`x @ W_head + b`	(16, 64, 65) logits
loss	`cross_entropy(logits.reshape(1024, 65), y.reshape(1024))`	()

The parameters, counted by hand, which is a good habit before trusting any model code:

tensor	shape	count
token embedding	65 × 64	4,160
position table	64 × 64	4,096
per block: 2 LayerNorms	2 × (64 + 64)	256
per block: QKV projection	64 × 192 + 192	12,480
per block: output projection	64 × 64 + 64	4,160
per block: MLP up	64 × 256 + 256	16,640
per block: MLP down	256 × 64 + 64	16,448
2 blocks	2 × 49,984	99,968
final LayerNorm	64 + 64	128
head	64 × 65 + 65	4,225
total		**112,577**

The script prints `112.6k parameters, vocab 65`, which matches. It is a tiny model, one thousandth of GPT-2 small, chosen so that a full training run takes a few minutes on a laptop CPU. Nothing in the code depends on the size. `--layers 4 --dim 128` gives a bigger model with the same code paths.

14.3 Embeddings as a one-hot matmul

An embedding is a table lookup: row `idx` of a `(V, C)` matrix. Our IR has no gather and no scatter. Its memory ops are the `INDEX` / `LOAD` that rangeify generates, and autodiff only works on pure shape graphs (Chapter 8). So we write the lookup as a matrix product with a one-hot matrix:

`uopc/nn.py · Embedding`

```python
class Embedding:
  """A one-hot matmul: differentiable without a scatter op."""
  def __init__(self, vocab: int, dim: int):
    self.vocab, self.weight = vocab, Tensor.randn(vocab, dim) * 0.02
    self.weight = self.weight.realize(); self.weight.requires_grad = True
  def __call__(self, idx: Tensor) -> Tensor: return idx.one_hot(self.vocab) @ self.weight
```

uopc/tensor.py · Tensor.one_hot

```python
  def one_hot(self, n: int) -> Tensor:
    return self.unsqueeze(-1).eq(Tensor.arange(n)).float()
```

`idx.unsqueeze(-1)` has shape (B, T, 1), and `Tensor.arange(V)` has shape (V,). Comparing them broadcasts to (B, T, V): entry `[b, t, v]` is 1.0 if `idx[b,t] == v`. Multiplying by `W` (V, C) then selects one row per token. The gradient comes for free from the matmul rule: `dW = one_hot⊤ @ dY`, which adds each position's gradient into the row of the token that was there. That is a scatter-add, derived by autodiff with no special case.

It sounds wasteful, since it builds a (16, 64, 65) tensor of mostly zeros. It isn't, because nothing builds it. Here is the embedding kernel the compiler generates (`r_16_64_64`, from the forward plan). It is excerpted to the first of its sixteen 4×4 register-tile accumulators, and the `//` comments are mine:

```c
void r_16_64_64(int* restrict data0, int* restrict data1, float* restrict data2, float* restrict
data3) {
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 16; ridx0++) {
    ...
      int val6 = data0[(alu2+alu5)];                  // the token id idx[b, t]
      ...
        for (int ridx3 = 0; ridx3 < 65; ridx3++) {
          float val16 = acc0;
          int val17 = data1[ridx3];                   // arange(65)[v]
          float alu18 = ((float)(((val6!=val17)!=1)));   // one_hot: (idx == v) as 0.0/1.0
          int alu19 = ((ridx2*260)+ridx3);
          float val20 = data2[alu19];                 // W_tok, transposed by the layout pass
          acc0 = (val16+(alu18*val20));
          ...
```

The one-hot is computed on the fly *inside the reduction*, as a comparison and a cast, and multiplied straight into the accumulator. Rangeify pushed the index of the broadcast into the comparison, and kernelization fused it into the reduce. The `(B, T, V)` tensor exists only as an expression. The cost is B·T·V·C = 4.3 M multiply-adds per forward pass, against a lookup's 65 K copies, which is negligible at V = 65. For a 50,000-

token vocabulary it would not be, and a real compiler would need a gather op (Chapter 15 discusses this). The small `e_1_1_64_65` kernel before it is the layout pass from Chapter 10 copying `W_tok` to a K-contiguous layout.

The positional embedding is a plain learned `(T, C)` tensor `pos`. `self.pos[:T].reshape(1, T, -1)` is a `SHRINK` and a `RESHAPE`, and the add broadcasts it over the batch. As you will see in 14.5, that add never gets its own kernel either.

14.4 Causal self-attention

Attention lets each position read from every earlier position, with weights it computes from the data. For a single head with queries Q, keys K and values V, all (T, D):

```
scores = Q Kᵀ / √D                  (T, T)   how much position i wants position j
scores[i, j] = -inf  for j > i               causality: no peeking at the future
P = softmax(scores, axis=-1)        (T, T)   each row sums to 1
out = P V                           (T, D)   a weighted average of values
```

Multi-head attention runs H such heads in parallel on D = C/H slices of the model width and concatenates the results. Here it is in `uopc`:

examples/gpt.py · Attention

```python
class Attention:
  def __init__(self, dim: int, heads: int, ctx: int):
    self.qkv, self.proj, self.heads = Linear(dim, 3 * dim), Linear(dim, dim), heads
    mask = np.triu(np.full((ctx, ctx), -1e9, np.float32), 1)       # -inf above the diagonal: no peeking
    self.mask = Tensor(mask)

  def __call__(self, x: Tensor) -> Tensor:
    B, T, C = x.shape
    H, D = self.heads, C // self.heads
    qkv = self.qkv(x).reshape(B, T, 3, H, D)
    q, k, v = (qkv[:, :, i].permute(0, 2, 1, 3) for i in range(3))   # (B, H, T, D)
    att = (q @ k.transpose()) * (1 / math.sqrt(D)) + self.mask[:T, :T].reshape(1, 1, T, T)
    y = att.softmax(-1) @ v                                          # (B, H, T, D)
    return self.proj(y.permute(0, 2, 1, 3).reshape(B, T, C))
```

Shapes, line by line, with B=16, T=64, C=64, H=4, D=16:

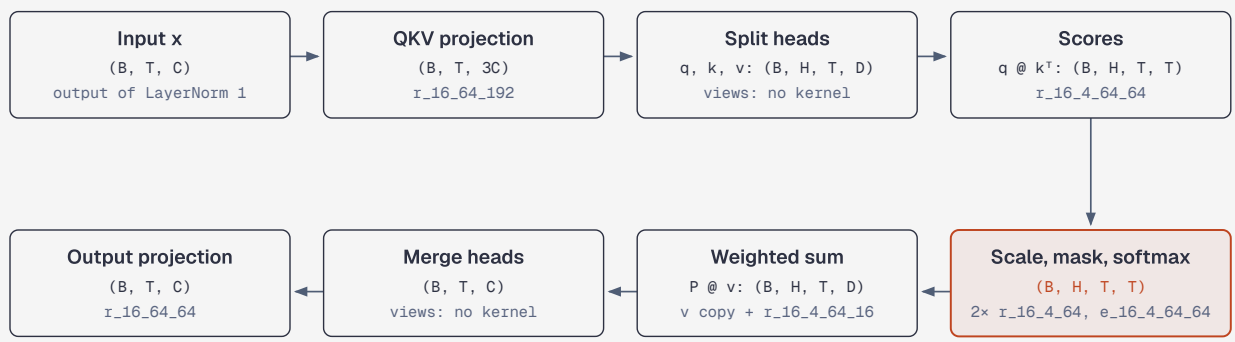
expression	shape	ops in the graph
`x`	(16, 64, 64)	
`self.qkv(x)`	(16, 64, 192)	matmul (REDUCE over C) + bias
`.reshape(B, T, 3, H, D)`	(16, 64, 3, 4, 16)	RESHAPE
`qkv[:, :, i]`	(16, 64, 4, 16)	SHRINK + RESHAPE
`.permute(0, 2, 1, 3)` → q, k, v	(16, 4, 64, 16)	PERMUTE
`k.transpose()`	(16, 4, 16, 64)	PERMUTE
`q @ k.transpose()`	(16, 4, 64, 64)	broadcast to (16, 4, 64, 64, 16), MUL, REDUCE
`* (1 / math.sqrt(D))`	(16, 4, 64, 64)	MUL by 0.25
`+ self.mask[:T, :T].reshape(1, 1, T, T)`	(16, 4, 64, 64)	SHRINK, RESHAPE, broadcast ADD
`.softmax(-1)`	(16, 4, 64, 64)	MAX, ADD reductions, EXP2, RECIP
`@ v`	(16, 4, 64, 16)	broadcast to (16, 4, 64, 16, 64), MUL, REDUCE
`.permute(0, 2, 1, 3).reshape(B, T, C)`	(16, 64, 64)	PERMUTE + RESHAPE
`self.proj(...)`	(16, 64, 64)	matmul + bias

Splitting into heads and merging them back are *views*: reshapes and permutes, which rangeify turns into index arithmetic (Chapter 6). No data is moved for them. The two products are batched matmuls, which `Tensor.matmul` builds by broadcasting to a 5-D `(…, M, N, K)` tensor and reducing the last axis:

`uopc/tensor.py · Tensor.matmul`

```python
def matmul(self, w: Tensor) -> Tensor:
  """(..., M, K) @ (..., K, N): broadcast to (..., M, N, K), multiply, reduce K."""
  x = self if self.ndim > 1 else self.reshape(1, -1)
  wt = w.reshape(-1, 1) if w.ndim == 1 else w
  wt = wt.transpose()                                    # (..., N, K)
  x = x.reshape(x.shape[:-1] + (1, x.shape[-1]))         # (..., M, 1, K)
  wt = wt.reshape(wt.shape[:-2] + (1,) + wt.shape[-2:])  # (..., 1, N, K)
  out = (x * wt).sum(-1)
  if self.ndim == 1: out = out.reshape(out.shape[:-2] + out.shape[-1:])
  if w.ndim == 1: out = out.reshape(out.shape[:-1])
  return out
```

The broadcast product for `q @ kᵀ` would have 16·4·64·64·16 = 4.2 M elements, and like the one-hot, it is never materialized. It lives inside the reduce loop of one kernel.



Kernel names from the default model: B = 16, H = 4, T = 64, D = 16.
The 1/√D = 0.25 scale and the mask fuse into the softmax kernels.

The mask is an ordinary constant tensor, built once in numpy: `-1e9` above the diagonal and 0 elsewhere. Adding it before the softmax drives `exp` of the future positions to exactly 0. We use -10^9 rather than $-\infty$ because infinities invite NaNs: a row that is entirely masked (possible with padding masks, though not with this causal one) would compute `-inf - (-inf)` in the max-subtraction, and `0 · inf` can appear in a backward pass. With a large finite number the arithmetic stays finite, and `exp(-1e9)` is still exactly 0.

The softmax is the numerically stable version from Chapter 9, `exp(x - max(x)) / sum(...)`:

`uopc/tensor.py · Tensor.softmax`

```
  def softmax(self, axis=-1):
    e = (self - self.max(axis, keepdim=True)).exp()
    return e / e.sum(axis, keepdim=True)
```

Here are the kernels the scale, mask and softmax become, taken from the forward plan. First the row max (`r_16_4_64`, excerpted to one of its 16 unrolled accumulators, with comments added):

```
      for (int ridx2 = 0; ridx2 < 64; ridx2++) {
        float val7 = acc0;
        int alu8 = ((alu2+alu5)+ridx2);
        float val9 = data0[alu8];                    // raw score q·k
        int alu10 = (alu5+ridx2);
        float val11 = data1[alu10];                  // mask[i, j]
        acc0 = ((val7>((val9*0.25f)+val11))?val7:((val9*0.25f)+val11));
```

The `0.25f` is 1/√16, folded to a constant, and the mask add is fused into the max. Then a second reduction `r_16_4_64` computes the row sums of `exp2((s*0.25 + mask - max) * log2(e))`, and one elementwise kernel writes the probabilities:

```
void e_16_4_64_64(float* restrict data0, float* restrict data1, float* restrict data2, float*
restrict data3, float* restrict data4) {
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 16; ridx0++) {
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      for (int ridx2 = 0; ridx2 < 64; ridx2++) {
        int alu1 = (ridx2*64);
        int alu2 = (((ridx0*256)+(ridx1*64))+ridx2);
        float val3 = data2[alu2];
        float val4 = data3[alu2];
        for (int ridx3 = 0; ridx3 < 64; ridx3++) {
          int alu5 = ((((ridx0*16384)+(ridx1*4096))+alu1)+ridx3);
          float val6 = data0[alu5];
          float val7 = data1[(alu1+ridx3)];
          data4[alu5] = (exp2f(((((val6*0.25f)+val7)+val3)*1.4426950408889634f))*val4);
        }
      }
    }
  }
}
```

`val3` is −max and `val4` is 1/sum, both computed once per row, outside the inner loop, by the linearizer's scope rules. Two tiny kernels `e_16_4_64_1` compute them (`data1 = data0 * -1.0f` and `data1 = 1.0f/data0`), for a reason that turns out to be an accident of the cost model (Pitfall 9). This is the *unfused*

attention of Chapter 11: it materializes the (B, H, T, T) probabilities, 1 MB here. At T = 64 that is fine. At T = 2048 it is what flash attention exists to avoid.

The test that matters most for attention is not numerical. It checks that the future does not leak:

tests/test_w09_models.py · test_attention_is_causal

```
def test_attention_is_causal():
  Tensor.manual_seed(0)
  att = Attention(8, 2, 6)
  x = np.random.randn(1, 6, 8).astype(np.float32)
  y1 = att(Tensor(x)).numpy()
  x2 = x.copy(); x2[0, 4:] += 10                # change the future
  y2 = att(Tensor(x2)).numpy()
  np.testing.assert_allclose(y1[0, :4], y2[0, :4], atol=1e-5)   # the past doesn't see it
```

Changing positions 4 and 5 of the input must leave outputs 0–3 untouched. A missing mask or a transposed one would fail this test even though every individual kernel is "correct".

14.5 LayerNorm

LayerNorm normalizes each position's C-vector to zero mean and unit variance, then applies a learned per-channel scale and shift:

```
mu  = mean(x, axis=-1)                        (B, T, 1)
var = mean((x - mu)^2, axis=-1)               (B, T, 1)
y   = (x - mu) / sqrt(var + eps) * w + b      (B, T, C)
```

uopc/nn.py · LayerNorm

```
class LayerNorm:
  def __init__(self, n: int, eps: float = 1e-5):
    self.weight, self.bias, self.eps = Tensor.ones(n).realize(), Tensor.zeros(n).realize(), eps
    self.weight.requires_grad = self.bias.requires_grad = True
  def __call__(self, x: Tensor) -> Tensor:
    mu = x.mean(-1, keepdim=True)
    xc = x - mu
    return xc * (xc.square().mean(-1, keepdim=True) + self.eps).rsqrt() * self.weight + self.bias
```

It is checked against numpy:

tests/test_w09_models.py · test_layernorm_matches_numpy

```
def test_layernorm_matches_numpy():
  x = np.random.randn(4, 8).astype(np.float32)
  y = LayerNorm(8)(Tensor(x)).numpy()
  ref = (x - x.mean(-1, keepdims=True)) / np.sqrt(x.var(-1, keepdims=True) + 1e-5)
  np.testing.assert_allclose(y, ref, atol=1e-5)
```

The compiler turns the first LayerNorm of block 1 into four kernels: a sum (`r_16_64`), a sum of squared deviations (`r_16_64`, which recomputes `x - mu` inside the reduce rather than storing it), the reciprocal standard deviation (`e_16_64_1`), and the normalization:

```
void e_16_64_1(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 16; ridx0++) {
    for (int ridx1 = 0; ridx1 < 64; ridx1++) {
      int alu1 = ((ridx0*64)+ridx1);
      float val2 = data0[alu1];
      data1[alu1] = (1.0f/sqrtf(((val2*0.015625f)+1e-05f)));
    }
  }
}
void e_16_64_64(float* restrict data0, float* restrict data1, float* restrict data2, float*
restrict data3, float* restrict data4, float* restrict data5, float* restrict data6) {
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 16; ridx0++) {
    for (int ridx1 = 0; ridx1 < 64; ridx1++) {
      int alu1 = (ridx1*64);
      int alu2 = ((ridx0*64)+ridx1);
      float val3 = data2[alu2];
      float val4 = data3[alu2];
      for (int ridx2 = 0; ridx2 < 64; ridx2++) {
        int alu5 = (((ridx0*4096)+alu1)+ridx2);
        float val6 = data0[alu5];
        float val7 = data1[(alu1+ridx2)];
        float val8 = data4[ridx2];
        float val9 = data5[ridx2];
        data6[alu5] = (((((val6+val7)+((val3*0.015625f)*-1.0f))*val4)*val8)+val9);
      }
    }
  }
}
```

Read the last line of `e_16_64_64`. `val6 + val7` is the token embedding plus the position embedding, `val3*0.015625f` is sum/64 = mean, `val4` is 1/σ, and `val8`, `val9` are `w` and `b`. The embedding add, the

centering, the scaling and the affine transform are all one expression. The embedding sum `x` is never written to memory on its own.

Why is this LayerNorm output written to memory at all, as `e_16_64_64` does, when it could be fused into the QKV matmul that consumes it? Because that matmul broadcasts it 192 times (once per output column), and recomputing a LayerNorm 192 times per element costs more than one round trip through memory. That decision is `stage_broadcast_compute`, and it is the subject of Section 14.10.

14.6 GELU and the MLP

The MLP widens to 4C = 256, applies a nonlinearity, and projects back. GPT-2 uses GELU, `x · Φ(x)` with Φ the standard normal CDF, in its tanh approximation:

`uopc/tensor.py · Tensor.gelu`

```
  def gelu(self): return 0.5 * self * (1 + (math.sqrt(2 / math.pi) * (self + 0.044715 * self *
self * self)).tanh())
```

`tanh` itself is built from `sigmoid`, which is built from `exp`, which is `exp2` times a constant (Chapter 9). The compiler flattens the whole tower into one expression, with the MLP-up bias add fused in:

```
void e_16_64_256(float* restrict data0, float* restrict data1, float* restrict data2) {
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 16; ridx0++) {
    for (int ridx1 = 0; ridx1 < 64; ridx1++) {
      for (int ridx2 = 0; ridx2 < 256; ridx2++) {
        int alu1 = (((ridx0*16384)+(ridx1*256))+ridx2);
        float val2 = data0[alu1];
        float val3 = data1[ridx2];
        float alu4 = (val2+val3);
        data2[alu1] = ((alu4*0.5f)*((((1.0f/(exp2f((((((alu4+
(((alu4*0.044715f)*alu4)*alu4))*0.7978845608028654f)*2.0f)*-1.0f)*1.4426950408889634f))+1.0f))*2.
0f)+-1.0f)+1.0f));
      }
    }
  }
}
```

`0.7978845608028654` is √(2/π). The derivative of all of this, which the backward pass needs, is derived by `gradient()` from the same ops. Nobody wrote a GELU backward.

14.7 Residual blocks and the full model

examples/gpt.py · Block

```
class Block:
  def __init__(self, dim, heads, ctx):
    self.ln1, self.attn, self.ln2 = LayerNorm(dim), Attention(dim, heads, ctx), LayerNorm(dim)
    self.fc, self.out = Linear(dim, 4 * dim), Linear(4 * dim, dim)
  def __call__(self, x: Tensor) -> Tensor:
    x = x + self.attn(self.ln1(x))
    return x + self.out(self.fc(self.ln2(x)).gelu())
```

examples/gpt.py · GPT

```
class GPT:
  def __init__(self, vocab, ctx, dim, heads, layers):
    self.tok, self.pos = Embedding(vocab, dim), Tensor.randn(ctx, dim) * 0.02
    self.pos = self.pos.realize(); self.pos.requires_grad = True
    self.blocks = [Block(dim, heads, ctx) for _ in range(layers)]
    self.ln, self.head = LayerNorm(dim), Linear(dim, vocab)
  def __call__(self, idx: Tensor) -> Tensor:
    B, T = idx.shape
    x = self.tok(idx) + self.pos[:T].reshape(1, T, -1)
    for b in self.blocks: x = b(x)
    return self.head(self.ln(x))
```

This is the **pre-LayerNorm** arrangement used by GPT-2 and later models: each branch normalizes its *input*, and the residual stream `x` itself is never normalized until the final `self.ln`. Gradients flow back through the `+` of each residual connection unchanged, which is what makes deep stacks trainable.

The `Linear` layer is `x @ W + b`, with `W` stored as (in, out):

uopc/nn.py · Linear

```python
class Linear:
  def __init__(self, n_in: int, n_out: int, bias: bool = True):
    bound = 1 / math.sqrt(n_in)
    self.weight = Tensor.uniform(n_in, n_out, low=-bound, high=bound, requires_grad=True)
    self.bias = Tensor.zeros(n_out).realize() if bias else None
    if self.bias is not None: self.bias.requires_grad = True
  def __call__(self, x: Tensor) -> Tensor: return x.linear(self.weight, self.bias)
```

14.8 The loss

uopc/tensor.py · Tensor.log_softmax

```python
  def log_softmax(self, axis=-1):
    m = self - self.max(axis, keepdim=True)
    return m - m.exp().sum(axis, keepdim=True).log()
```

uopc/tensor.py · Tensor.cross_entropy

```python
  def cross_entropy(self, labels: Tensor) -> Tensor:
    """self: (B, C) logits, labels: (B,) i32."""
    return -(self.log_softmax(-1) * labels.one_hot(self.shape[-1])).sum(-1).mean()
```

`log_softmax` subtracts the max before the `exp`, so it stays finite even for large logits, and it never computes `log(softmax(x))`, which would underflow to `log(0)` for unlikely classes. The labels are one-hot encoded and the dot product picks out the log-probability of the right class. That is the same one-hot trick as the embedding, and again the one-hot is never materialized.

14.9 AdamW, and constants that change every step

AdamW keeps two running averages per parameter, the mean of the gradients `m` and of their squares `v`, and updates with their ratio:

```
m ← β1·m + (1-β1)·g
v ← β2·v + (1-β2)·g²
p ← p·(1 - lr·wd) - lr · m̂ / (√v̂ + ε),     m̂ = m / (1-β1ᵗ),  v̂ = v / (1-β2ᵗ)
```

The `(1-β^t)` terms correct the bias of averages that start at zero. "W" means the weight decay is *decoupled* (Loshchilov & Hutter): it shrinks `p` directly rather than being added to the gradient.

uopc/nn.py · AdamW

```python
class AdamW(Optimizer):
  def __init__(self, params, lr=1e-3, b1=0.9, b2=0.999, eps=1e-8, wd=0.0):
    super().__init__(params)
    self.lr, self.b1, self.b2, self.eps, self.wd, self.t = lr, b1, b2, eps, wd, 0
    self.m = [Tensor.zeros(*p.shape).realize() for p in params]
    self.v = [Tensor.zeros(*p.shape).realize() for p in params]
  def step(self, *extra: Tensor):
    self.t += 1
    # bias corrections are python floats baked into the graph as constants: each step's
    # graph differs only in two CONSTs, so we pass them as tensors to keep the JIT cache warm
    c1 = Tensor([self.lr / (1 - self.b1 ** self.t)])
    c2 = Tensor([1 / (1 - self.b2 ** self.t)])
    news = []
    for p, m, v in zip(self.params, self.m, self.v):
      g = p.grad
      m.assign(m * self.b1 + g * (1 - self.b1))
      v.assign(v * self.b2 + g.square() * (1 - self.b2))
      upd = m * c1.reshape(*([1] * p.ndim)) / ((v * c2.reshape(*([1] * p.ndim))).sqrt() +
self.eps)
      news.append(p.assign(p.detach() * (1 - self.lr * self.wd) - upd))
    Tensor.realize_all(*news, *self.m, *self.v, *extra)
```

The comment in `step` is a bug story (NOTES 17). Written as Python floats, the bias corrections become `CONST`s whose values change every step, so the structural cache misses every step and recompiles all 243 kernels. Passing them as 1-element tensors keeps the graph's structure fixed (see Chapter 7, Section 7.8). Section 14.12 measures what the naive version costs.

The update is realized together with everything else. `Tensor.realize_all(*news, *self.m, *self.v, *extra)` is called with the loss as `extra`, so the forward pass, the backward pass, the optimizer update and the loss value are all *one* schedule: one call to `realize`, one cache lookup, 243 kernels.

14.10 Training

`examples/gpt.py · main`

```
def main():
  ap = argparse.ArgumentParser()
  ap.add_argument("--steps", type=int, default=600)
  ap.add_argument("--batch", type=int, default=16)
  ap.add_argument("--ctx", type=int, default=64)
  ap.add_argument("--dim", type=int, default=64)
  ap.add_argument("--heads", type=int, default=4)
  ap.add_argument("--layers", type=int, default=2)
  ap.add_argument("--lr", type=float, default=3e-3)
  a = ap.parse_args()

  text = shakespeare()
  chars = sorted(set(text))
  stoi = {c: i for i, c in enumerate(chars)}
  data = np.array([stoi[c] for c in text], np.int32)
  split = int(0.9 * len(data))
  train, val = data[:split], data[split:]
  rng = np.random.default_rng(0)
  def batch(src):
    ix = rng.integers(0, len(src) - a.ctx - 1, a.batch)
    return Tensor(np.stack([src[i:i + a.ctx] for i in ix])), Tensor(np.stack([src[i + 1:i + a.ctx
+ 1] for i in ix]))

  Tensor.manual_seed(0)
  model = GPT(len(chars), a.ctx, a.dim, a.heads, a.layers)
  params = parameters(model)
  print(f"{sum(p.numel() for p in params) / 1e3:.1f}k parameters, vocab {len(chars)}")
  opt = AdamW(params, lr=a.lr, wd=0.01)
  t0 = time.perf_counter()
  for step in range(a.steps):
    x, y = batch(train)
    loss = model(x).reshape(-1, len(chars)).cross_entropy(y.reshape(-1))
    opt.zero_grad()
    loss.backward(opt.params)
    opt.step(loss)
    if step % 50 == 0 or step == a.steps - 1:
      vx, vy = batch(val)
      vloss = model(vx).reshape(-1, len(chars)).cross_entropy(vy.reshape(-1)).item()
      print(f"step {step:5d}  train {loss.item():.3f}  val {vloss:.3f}  {(time.perf_counter() -
t0) / (step + 1) * 1e3:6.0f} ms/step")

  # sample: feed the last ctx characters, draw the next one from the softmax
  idx = [stoi["\n"]]
  for _ in range(300):
    ctx = idx[-a.ctx:]
    ctx = ctx + [0] * (a.ctx - len(ctx))                          # fixed shape -> one compiled
graph
    logits = model(Tensor(np.array([ctx], np.int32))).numpy()[0, min(len(idx), a.ctx) - 1]
    p = np.exp(logits - logits.max()); p /= p.sum()
```

```
    idx.append(int(rng.choice(len(chars), p=p)))
  print("".join(chars[i] for i in idx))
```

`batch` samples B random windows. Every 50 steps the script also evaluates one batch from the held-out last 10% of the text. Here is the full run, as captured in `book/outputs/gpt_train.txt` (`python examples/gpt.py`, 600 steps):

```
112.6k parameters, vocab 65
step     0  train 4.266  val 3.827    8933 ms/step
step    50  train 2.841  val 2.855     368 ms/step
step   100  train 2.616  val 2.559     288 ms/step
step   150  train 2.503  val 2.577     263 ms/step
step   200  train 2.444  val 2.514     249 ms/step
step   250  train 2.357  val 2.491     240 ms/step
step   300  train 2.265  val 2.336     240 ms/step
step   350  train 2.325  val 2.271     241 ms/step
step   400  train 2.156  val 2.286     239 ms/step
step   450  train 2.274  val 2.276     240 ms/step
step   500  train 2.074  val 2.223     235 ms/step
step   550  train 2.065  val 2.163     235 ms/step
step   599  train 2.015  val 2.110     235 ms/step

ICGLOUTery man!

For apbeng mantound rear:

Whals pant lachtiow't,
Htaw be pect she sto now thy sost soo with ble oving dity:
Went song rief tin of taut thurfs me dat aournclint,
Goant?


FORE:
Se wat, more-ks in sterdide be tom:
Whinenten wer I thent nor ifor ond he;
Wegon:
If to hard, of his do to
```

How to read it:

- **Step 0's loss is 4.266**, close to the ln 65 = 4.174 of a uniform guess. Random small weights give nearly uniform predictions, which is a sanity check that the loss is wired correctly. A first loss of 40 or 0.1 would mean a bug.
- **The validation loss tracks the training loss** (2.110 against 2.015 at the end), so this small model is not overfitting in 600 steps. Each logged loss is from a single batch, so the steps are noisy: train 2.325 at step

350 after 2.265 at step 300.

- **For scale:** in Andrej Karpathy's "Let's build GPT" lecture a bigram model, which predicts from the previous character only, plateaus around 2.5 on this data, and nanoGPT's character-level Shakespeare config (a 10.7M-parameter model trained on a GPU) reaches a validation loss of about 1.47. Our 112.6k parameters at 2.11 sit in between, where you would expect.
- **The sample** has learned the *format*: speaker names in capitals followed by a colon, line breaks, capitalized verse lines, real short words ("man", "be", "she", "now", "thy", "with", "to", "of", "his"), and plausible pseudo-words. More steps, a wider model (`--dim 128 --layers 4`) or a longer context all improve it.
- **The ms/step column is a running average since step 0**, so it includes the first step's compile. It drops from 8933 ms to 235 ms because steps 1 onward reuse the cached plan. Section 14.11 explains both numbers.

Training itself is checked by a fast test, a tiny GPT on a trivially predictable sequence that must cut its loss by 4× in 30 steps:

`tests/test_w09_models.py · test_tiny_gpt_learns`

```
def test_tiny_gpt_learns():
  Tensor.manual_seed(0)
  V, T = 11, 8
  model = GPT(V, T, 16, 2, 1)
  opt = AdamW(parameters(model), lr=1e-2)
  seq = np.arange(64) % V                          # a trivially predictable sequence
  x = Tensor(np.stack([seq[i:i + T] for i in range(4)]).astype(np.int32))
  y = Tensor(np.stack([seq[i + 1:i + T + 1] for i in range(4)]).astype(np.int32))
  losses = []
  for _ in range(30):
    loss = model(x).reshape(-1, V).cross_entropy(y.reshape(-1))
    opt.zero_grad(); loss.backward(opt.params); opt.step(loss)
    losses.append(loss.item())
  assert losses[-1] < 0.25 * losses[0]
```

Sampling with a fixed shape

Generation feeds the last T characters and draws the next one from the softmax of the logits at the last real position. The prompt grows by one character per step. Feeding the *actual* prompt would make every step's graph a different shape, (1, 1), (1, 2), (1, 3), ..., so the first 64 samples would each compile a new plan. The sampler instead pads the context to T with zeros and reads the logits at position `len(idx) - 1` (NOTES 18). Causality makes the padding harmless: position *t*'s output never sees positions after *t*. With the padding every forward pass has the same graph and compiles once. In the `DEBUG=2` log, the first sampling forward

pass (52 kernels at batch 1) shows `schedule 4423.1 ms`, and every later one shows about 5 ms (that run was on a busy machine, so only the ratio matters). The proper fix is symbolic shapes plus a KV cache, the subject of Chapter 15.

14.11 What the compiler did: anatomy of a training step

Counting kernels

The forward pass alone is 61 kernels. With autodiff and the optimizer, one training step is 243. A scratch script that runs three steps and inspects the structural cache:

```
# scratch/plans.py (after building the default model and optimizer)
n0 = len(K._plans); step(); step(); step()
print("plans cached after 3 steps:", len(K._plans), "(new:", len(K._plans) - n0, ")")
big = max(K._plans.values(), key=lambda p: len(p.kernels))
print("training-step plan:", len(big.kernels), "kernels,", len({k.src for k in big.kernels}),
"distinct C sources,", len(big.temps), "temp buffers")
print("temp memory: %.1f MB" % (sum(n * 4 for _, n in big.temps) / 1e6))
```

```
plans cached after 3 steps: 14 (new: 1 )
training-step plan: 243 kernels, 128 distinct C sources, 152 temp buffers
temp memory: 31.7 MB
```

(A re-run after the compiler's later fixes, such as reusing temp arrays across calls, gave the same counts, 61 forward kernels and identical losses, and `examples/gpt.py --steps 51` reproduced the logged losses exactly. Timings from that re-run were taken on a busy machine and are not comparable, so the numbers below are the quiet-machine ones.)

Three steps produced *one* new plan, so the cache works. Of the 243 kernels only 128 have distinct C source. The rest are byte-identical, the payoff of renumbering buffers locally per kernel (Chapter 7): the two blocks' identical layers render to identical C, which compiles once and is shared through the content-addressed `.so` cache.

DEBUG=2: per-kernel timings

`DEBUG=2` prints one line per kernel launch with its time and an estimated GFLOP/s, and `DEBUG=1` a summary per `realize`. To get a clean listing of a *steady-state* step (not the first, which compiles), the scratch script flips `uopc.kernelize.DEBUG` at run time after three warm-up steps:

```
# scratch/debug_step.py (excerpt)
for _ in range(3): step()
K.DEBUG = 2
buf = io.StringIO()
with contextlib.redirect_stdout(buf): step()
```

```
  r_1024                            0.336 ms     1.39 GFLOP/s
  e_1_1_64_65                       0.047 ms     0.00 GFLOP/s
  r_16_64_64                        0.290 ms    73.52 GFLOP/s
  r_16_64                           0.247 ms     0.53 GFLOP/s
  r_16_64                           0.272 ms     1.45 GFLOP/s
  e_16_64_1                         0.036 ms     0.11 GFLOP/s
  e_16_64_64                        0.141 ms     3.26 GFLOP/s
  e_1_1_192_64                      0.027 ms     0.00 GFLOP/s
  r_16_64_192                       0.254 ms    99.02 GFLOP/s
  e_16_64_192                       0.137 ms     1.44 GFLOP/s
  r_16_4_64_64                      0.276 ms    30.43 GFLOP/s
  r_16_4_64                         0.176 ms     4.46 GFLOP/s
  ...
  r_1024                            0.363 ms     2.20 GFLOP/s
  r_1                               0.049 ms     0.02 GFLOP/s
  e_1                               0.036 ms     0.00 GFLOP/s
realize: 243 kernels, schedule 8.1 ms, run 112.1 ms
243 kernel launches; sum of kernel times 108.4 ms
top kernel names by total time:
  r_16_64_64             x12    24.65 ms
  r_16_4_64              x10    14.84 ms
  e_16_4_64_64           x4     12.74 ms
  r_192_64               x2     12.51 ms
  r_16_64                x20     5.05 ms
  r_16_4_64_64           x4      4.66 ms
  r_64                   x14     4.11 ms
  r_16_64_256            x4      3.16 ms
  r_256_64               x2      2.69 ms
  e_16_64_64             x10     2.40 ms
```

Kernel names are `r_` (contains a reduce) or `e_` (elementwise) followed by the output shape (Chapter 7). Matching them to the model is a useful exercise:

- `r_16_64_192` is the QKV projection, `(B, T, 3C)`. At 99 GFLOP/s it is near the CPU GEMM rate of Chapter 10. `e_1_1_192_64` before it is the layout pass transposing the weight.
- `r_16_4_64_64` is `q @ kᵀ`, and `r_16_4_64` (×2 per layer forward, plus backward) are the softmax reductions.
- `r_192_64`, `r_256_64`, `r_64`, … are **weight gradients**: `dW = xᵀ @ dy` reduces over all B·T = 1024 positions, and a bias gradient sums over them. `r_192_64` takes 6 ms per call for 25 MFLOP, about 4 GFLOP/s, among the slowest kernels per FLOP in the step. Its reduce axis (B·T) is the *outermost* axis of both activations, which is the worst case for the layout heuristics of Chapter 10 (Exercise 14.7).
- `r_1024`, `r_1`, `e_1` are the loss and its scalar plumbing.

The arithmetic: the standard estimate for one training step of a transformer is 6 × parameters × tokens FLOPs, here 6 × 112,577 × 1,024 ≈ 0.69 GFLOP. At about 108 ms of kernel time that's 6.4 GFLOP/s, 5% of what the same laptop reaches on a large GEMM. The model is simply too small to be compute-bound. Most kernels touch a few KB and finish in tens of microseconds, so launch overhead, memory traffic and serial reductions dominate. A bigger model raises the fraction spent in big GEMMs, and with it the efficiency.

Where the rest of the time goes

The kernels take about 110 ms, but a steady-state step takes longer. Splitting the step (from a scratch script, on a quiet machine):

```
build forward+backward graph:   20.4 ms
optimizer graph + realize   :  143.0 ms   (243 kernels)
```

About 20 ms is Python building the forward graph and differentiating it, about 8 ms is computing the structure key and looking up the plan (`schedule 8.1 ms` above), and the remainder is kernel execution plus the Python loop that calls 243 `ctypes` functions. On a quiet machine the steady state measures 140–150 ms per step. The training log's 235 ms includes the running-average effect of step 0 and periodic validation passes, and was measured while the machine was doing other work.

The first step: profiling the compiler

The first step is a different story. It has to build the plan: kernelize, run the layout pass, rangeify, simplify and render 243 kernels, then compile whatever is not already in the `.so` cache with gcc. Two measurements of the same script:

```
$ UOPC_CACHE=scratch/coldcache python ablate.py baseline   # empty .so cache
baseline      first step   25.6 s   steady     148 ms/step   loss 4.286 -> 3.335
$ python ablate.py baseline                                          # warm .so cache
baseline      first step    5.8 s   steady     139 ms/step   loss 4.286 -> 3.335
```

With an empty cache, gcc compiles about 140 shared objects (the 128 distinct kernels of the step plus the initialization kernels) and the first step takes 26 s. With a warm cache it's 6 s, all of it Python. To find out *which* Python, use `cProfile`:

```python
# scratch/profile_step.py (excerpt)
pr = cProfile.Profile()
t0 = time.perf_counter(); pr.enable(); step(); pr.disable(); t1 = time.perf_counter()
print(f"step 0 (builds + compiles): {(t1-t0):.1f} s")
s = io.StringIO(); pstats.Stats(pr, stream=s).sort_stats("cumulative").print_stats(14);
print(s.getvalue()[:3500])
```

```
step 0 (builds + compiles): 20.8 s
         25845652 function calls (23894366 primitive calls) in 20.502 seconds

   Ordered by: cumulative time
   List reduced from 374 to 14 due to restriction <14>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   20.813   20.813 scratch/profile_step.py:21(step)
        1    0.001    0.001   20.756   20.756 uopc/nn.py:87(step)
        1    0.001    0.001   20.726   20.726 uopc/tensor.py:63(realize_all)
        1    0.005    0.005   20.724   20.724 uopc/kernelize.py:178(realize)
        1    0.080    0.080   20.527   20.527 uopc/kernelize.py:111(make_plan)
     2168    1.271    0.001   11.119    0.005 uopc/rewrite.py:90(graph_rewrite)
      243    0.007    0.000    9.888    0.041 uopc/opt.py:109(hand_opt)
      243    0.012    0.000    9.845    0.041 uopc/rangeify.py:170(lower)
29919/1702    0.146    0.000    8.641    0.005 uopc/rangeify.py:74(rangeify)
    11144    0.033    0.000    8.511    0.001 uopc/symbolic.py:282(simplify)
   244507    0.685    0.000    7.839    0.000 uopc/rewrite.py:82(rewrite)
    16254    0.011    0.000    6.871    0.000 uopc/rangeify.py:95(<genexpr>)
     1214    0.030    0.000    6.654    0.005 uopc/rangeify.py:125(_reduce)
17080/3752    0.013    0.000    6.544    0.002 uopc/rangeify.py:83(<genexpr>)
```

`cProfile` makes the step about 3.5× slower (20.8 s against 5.8 s unprofiled, since it instruments 26 million calls), but the proportions are what matter. Almost everything is under `make_plan`. More than half is `graph_rewrite`, called 2,168 times, mostly from `simplify` (11,144 calls) running symbolic rules over index expressions inside `rangeify`, in particular at every `RESHAPE` (line 95) and in `_reduce`. This is the cost of NOTES 12's fix. Simplifying indices eagerly is what keeps reductions out of the wrong loops, and it is

paid for in compile time. `hand_opt` → `lower` is the CPU optimizer rangeifying each kernel with its upcasts. Four rules for reading a profile: `tottime` is time in the function itself and `cumtime` includes callees. Sort by `cumulative` to find the subsystem, then by `tottime` to find the hot function. The `ncalls` column tells you whether a function is slow or just called too often.

14.12 The performance story

When the GPT first ran end to end, the first step took **64 seconds** and each further step about **1.2 seconds**. The shipped compiler takes about **9 s** and **0.24 s** under the conditions of `book/outputs/gpt_train.txt`. The fixes, from the development log (`book/NOTES.md`), in the order of the pipeline:

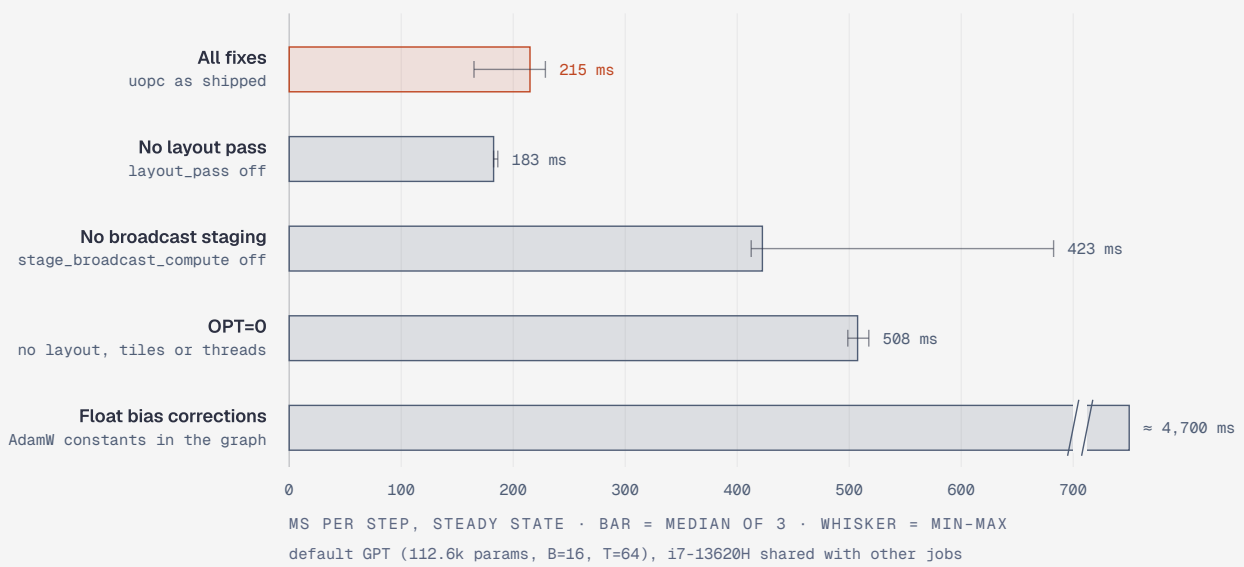
#	symptom	diagnosis	fix	effect
NOTES 12	reductions in attention far too slow	an unsimplified index `((i*N + j)//N) % N` *mentions* j, so the reduce lowering put the accumulator inside the j loop: $O(N^3)$	`simplify` each index at `RESHAPE` in rangeify	fused attention at N=512: 1787 ms → 7 ms
NOTES 14	`((x/4096)%4)` chains in GPT's inner loops	div/mod rules couldn't see individual terms of `(a+b)*c`	distribute integer `*` over `+` in the symbolic rules	div/mod folds, inner loops shrink
NOTES 15	374 ms of kernels per step	LayerNorm output EXPANDed into a matmul was *recomputed* per output column	`stage_broadcast_compute` with a cost model	374 ms → 125 ms of kernel time
NOTES 16	staging missed nested candidates	`substitute({x: STAGE(x)})` wraps the *original* x	`UOp.wrap` rebuilds bottom-up	staging becomes complete
NOTES 17	every step as slow as the first	AdamW's bias corrections were CONSTs, so the structure key changed each step	pass them as tensors	one plan for all steps
NOTES 21	a 21 ms kernel staging a (128,784,128) product	the layout pass staged a whole broadcast product	`push_views` + only stage operands smaller than the reduce input	the MNIST kernel goes away
NOTES 22	163 s of compilation for 12 steps	`UOp.__eq__` / `__hash__` and `Enum.__hash__` in Python	identity hashing (in C) and memoized `simplify`	47 s; first GPT step ~9 s

The intermediate step times between fixes were not all recorded, so instead of reconstructing them, here is an **ablation** of the shipped compiler: remove one fix at a time with a monkeypatch in a scratch script (nothing in `uopc/` changes), and measure steady-state ms/step on the default model:

```
# scratch/ablate.py (the switches)
if mode == "no_stage": K.stage_broadcast_compute = lambda roots, factor=4: roots
if mode == "no_layout":
  class NoLayout(O.HandOpt): layout = staticmethod(lambda roots: roots)
  Tensor.opt = NoLayout()
# mode == "opt0" sets OPT=0 before importing uopc; mode == "float_consts" uses an AdamW subclass
# whose step() bakes c1, c2 into the graph as python floats, as the first version did.
```

Three rounds of the four cheap configurations, interleaved, plus the expensive one:

```
baseline      first step    8.0 s   steady     215 ms/step   loss 4.286 -> 3.335
no_stage      first step   19.8 s   steady     682 ms/step   loss 4.286 -> 3.335
no_layout     first step    6.3 s   steady     186 ms/step   loss 4.286 -> 3.335
opt0          first step    2.9 s   steady     517 ms/step   loss 4.286 -> 3.335
baseline      first step    7.3 s   steady     165 ms/step   loss 4.286 -> 3.335
no_stage      first step   16.9 s   steady     412 ms/step   loss 4.286 -> 3.335
no_layout     first step    5.2 s   steady     182 ms/step   loss 4.286 -> 3.335
opt0          first step    2.8 s   steady     499 ms/step   loss 4.286 -> 3.335
baseline      first step   10.8 s   steady     229 ms/step   loss 4.286 -> 3.335
no_stage      first step   20.4 s   steady     423 ms/step   loss 4.286 -> 3.335
no_layout     first step    7.1 s   steady     183 ms/step   loss 4.286 -> 3.335
opt0          first step    3.0 s   steady     508 ms/step   loss 4.286 -> 3.335
float_consts  first step    6.1 s   steady    4699 ms/step   loss 4.286 -> 3.550
```



(The losses agree to three decimals across configurations, a free correctness check: every variant computes the same numbers. `float_consts` ran 4 steps instead of 9, hence its different final loss.) These runs shared the machine with other jobs (load average 5–10), so the absolute numbers are noisy: the baseline ranged from 165 to 229 ms. The ordering was stable across rounds, though, and it says:

1. **The compile cache is the biggest single factor.** Baking the bias corrections in as floats makes *every* step a first step: about 4.7 s/step (5.5 s in an earlier run), roughly 20–30× slower. One line of `nn.py`.
2. **Broadcast staging is worth about 2×.** Without it, LayerNorm and GELU outputs are recomputed inside every matmul that consumes them. Removing it also makes step 0 about 2.5× slower (20 s against 8 s), because the fused recomputing kernels are much larger graphs to rangeify and simplify.
3. **Register tiles and threads (OPT=0 vs. baseline) are worth about 2.4×.** OPT=0 compiles fastest (no upcasts to unroll) and runs slowest.
4. **The layout pass doesn't pay off on this model.** Removing it was, if anything, slightly *faster* (183 against 215 ms in this set, and 161–218 ms against 140–258 ms in other rounds). At K = 64 a strided operand fits in L1 anyway, so the transposing copies cost more than they save. The same pass was essential for MNIST's 784-wide GEMMs (NOTES 21) and for the 1024^3 GEMM of Chapter 10. An optimization is a bet about the workload. Measure it on each workload you care about.

Pitfalls

1. The $O(N^3)$ attention (NOTES 12). An unsimplified `RESHAPE` index put the softmax reductions inside the j loop, so fused attention did N times too much work (see Chapter 11, Section 11.5).

2. Undistributed products hide div/mod opportunities (NOTES 14). Without distributing `(a + b) * c`, GPT's attention kernels carried `((x/4096)%4)` chains in their inner loops (see Chapter 6, Pitfalls). Some residue remains. The fused `q @ kᵀ` kernel still computes `((alu14+256)%192)` in its inner loop where `alu14 ∈ [0, 63]`, so the value is just `alu14 + 64`. A bounds-based rule (if `vmin(x)//c == vmax(x)//c` then `x % c = x - c*(vmin(x)//c)`) would remove it (Exercise 14.8).

3. The fusion/recompute trap (NOTES 15). A LayerNorm output broadcast into a matmul is recomputed for every output column; staging such tensors took kernel time from 374 ms to 125 ms per step (see Chapter 7, Pitfalls).

4. `substitute` wraps the original (NOTES 16). Staging with `substitute` leaves candidates nested inside a target unstaged; `UOp.wrap` rebuilds bottom-up instead (see Chapter 7, Pitfalls).

5. Python floats break the JIT cache (NOTES 17). Anything that changes every step and enters the graph as a `CONST` (bias corrections, a learning-rate schedule, a dropout seed, a temperature) makes every step a cache miss, about 20–30× slower in the ablation. Pass such values as tensors (see Chapter 7, Pitfalls).

6. Variable shapes break the JIT cache (NOTES 18). A sampler that feeds a growing prompt compiles a new plan per length. Pad to a fixed shape, or use symbolic shapes (Chapter 15).

7. Staging a whole broadcast product (NOTES 21). An early layout pass staged the entire (128, 784, 128) broadcast product, a 21 ms kernel to save a few (see Chapter 10, Section 10.6).

8. Python overhead is compiler overhead (NOTES 22). Python-level `__eq__` / `__hash__` made compiling 12 GPT steps take 163 s; identity hashing and memoized `simplify` brought it to 47 s (see Chapter 1). Profile the compiler as seriously as the kernels.

9. A threshold on the wrong quantity. `stage_broadcast_compute` stages a broadcast tensor when `_fused_work(base) >= 4096`, and `_fused_work` is (ALU ops) × (elements of the *small* tensor):

uopc/kernelize.py · _fused_work

```
def _fused_work(u: UOp) -> int:
  """ALU ops * elements that recomputing u would cost (stopping at materialized nodes)."""
  from uopc.uop import ALU
  n, seen, stack = 0, set(), [u]
  while stack:
    x = stack.pop()
    if x in seen or x.op in (Ops.BUFFER, Ops.REDUCE, Ops.STAGE): continue
    seen.add(x)
    if x.op in ALU: n += 1
    stack.extend(x.src)
  return n * prod(u.shape)
```

The cost of *not* staging is recomputation, which is proportional to the elements of the *broadcast* result. At B = 16 several tiny tensors hit the threshold exactly: softmax's `-max` and `1/sum` (1 op × 16·4·64 = 4096 elements) and LayerNorm's `1/σ` (4 ops × 1024 = 4096). They become nine extra kernels (`e_16_4_64_1`, `e_16_64_1`), and at B = 8 they disappear: the forward pass has 52 kernels instead of 61. In this model the effect on step time was within noise (halving `_fused_work` gave 142–146 ms against 144–149 ms), so it is harmless here. It is still a cost model whose decisions flip with the batch size for reasons unrelated to cost, which is a good thing to find before it matters. (This is NOTES 15's "useless kernels" coming back by another door.)

Exercises

1. ★ **Count parameters.** Derive the 112,577 of Section 14.2 for `--dim 128 --layers 4 --heads 4` (V = 65, T = 64) by hand, then check with `sum(p.numel() for p in parameters(model))`.
2. ★ **Shapes.** For B=2, T=5, C=8, H=2, write the shape after every line of `Attention.__call__`, including the 5-D broadcast inside each matmul.
3. ★ **Read a kernel.** In `e_16_64_64` (Section 14.5), which `dataN` is the position table, which is LayerNorm's weight, and how can you tell from the index expressions alone?
4. ★★ **The cache, broken on purpose.** Add a cosine learning-rate schedule to AdamW. First pass `lr` as a Python float and measure ms/step. Then make it a tensor so the plan is reused. Explain the difference using `_structure_key`.
5. ★★ **Weight tying.** GPT-2 ties the head to the token embedding (`logits = x @ W_tokᵀ`). Implement it. How many parameters does it save here? What happens to the embedding's gradient, and why does autodiff handle the sum of two contributions without any change?
6. ★★ **Find the bug.** A student's causal mask is `np.tril(np.full((ctx, ctx), -1e9), -1)`. Their loss falls fast, to well under 1.0 in a few hundred steps, and their samples are gibberish. Explain both observations, and which test in `tests/test_w09_models.py` catches it.
7. ★★ **Weight-gradient kernels.** `r_192_64` reduces over B·T = 1024 positions at about 4 GFLOP/s. Dump its C with `DEBUG=4`, identify the strided operand, and change the layout heuristic (or add a rule) so that reductions over a leading axis transpose first. Measure the step time before and after.
8. ★★ **The leftover modulo.** Add the bounds-based FLOORMOD/FLOORDIV rule of Pitfall 2 to `uopc/symbolic.py`. Verify that the attention kernel's `% 192` disappears, run the whole test suite, and measure the compile time (it runs on every index, so it must be cheap).
9. ★★★ **A better cost model.** Change `stage_broadcast_compute` to estimate the actual recomputation cost (ops × elements of the broadcast consumer) against the cost of staging (one write and the reads), so that the decision doesn't depend on crossing 4096. Show that the nine small kernels disappear at B=16 and that LayerNorm and GELU outputs are still staged.
10. ★★★ **GPT on the GPU.** Make `make_plan` able to target the GPU path of Chapter 12: lower each kernel with `gpu_lower` (folding axes for 4-D kernels such as `r_16_4_64_64`), render with `render_cuda`, and run each kernel through `emulate` on the CPU so that `test_tiny_gpt_learns` passes on the emulator. With a GPU, run it through `cuda.py` and compare ms/step with the CPU.
11. ★★★ **Gradient clipping.** Add global-norm gradient clipping to the optimizer *without* an extra realize: the norm, the scale factor and the update must all be in the same schedule. Why does it have to be a tensor expression and not `float(norm)`?

Solutions to ★ exercises

1. Token embedding 65·128 = 8,320. Positions 64·128 = 8,192. Per block: two LayerNorms 4·128 = 512, QKV 128·384 + 384 = 49,536, projection 128·128 + 128 = 16,512, MLP up 128·512 + 512 = 66,048, MLP down 512·128 + 128 = 65,664, which totals 198,272, and ×4 = 793,088. Final LayerNorm 256, head 128·65 + 65 = 8,385. Total: 8,320 + 8,192 + 793,088 + 256 + 8,385 = **818,241** (≈ 818.2k).

2. `x` (2,5,8). `qkv(x)` (2,5,24), broadcast to (2,5,24,8) inside the matmul. Reshape (2,5,3,2,4). `qkv[:,:,i]` (2,5,2,4). Permute → q, k, v (2,2,5,4). `k.transpose()` (2,2,4,5). `q @ kᵀ` broadcasts to (2,2,5,5,4) and reduces to (2,2,5,5). The scale and mask keep (2,2,5,5), with the mask reshaped (1,1,5,5). Softmax (2,2,5,5). `@ v` broadcasts to (2,2,5,4,5) and reduces to (2,2,5,4). Permute (2,5,2,4), reshape (2,5,8), proj (2,5,8).

3. `data1[(alu1+ridx2)]` is indexed by (t, c) only, with no batch term: the position table, (T, C). `data4[ridx2]` and `data5[ridx2]` are indexed by c only: LayerNorm's weight and bias, in the order they appear in the expression (`*val8` then `+val9`), so `data4` is the weight. `data0` has the batch term `ridx0*4096`: the token embeddings (B, T, C). `data2` and `data3` are indexed by (b, t): the row sums and 1/σ.

Quiz

1. What loss should an untrained character model with V = 65 report, and why?
2. Why does `Embedding` use a one-hot matmul, and why doesn't it materialize a (B, T, V) tensor?
3. Why is the mask -10^9 and not $-\infty$?
4. In the generated softmax kernels, where did the constant `0.25f` come from?
5. Why must AdamW's bias corrections be tensors rather than Python floats in `uopc`?
6. A training step is 243 kernels but only 128 distinct C sources. Why?
7. Why does the sampler pad its context to T?
8. In the ablation, removing broadcast staging made both the steady state and the *first step* slower. Why the first step?

Answers

1. About ln 65 ≈ 4.17. Near-uniform predictions give each character probability 1/65, and cross-entropy is −ln(1/65).
2. The IR has no gather or scatter, and autodiff works on pure shape graphs. A one-hot matmul expresses the lookup, and its gradient (a scatter-add) comes from the matmul rule. The one-hot is an expression (`idx == v`) pushed into the reduce loop by rangeify and fused by kernelization, so it is never stored.

3. A fully masked row would compute `-inf - (-inf)` = NaN in the max-subtraction. A large finite number gives `exp(-1e9)` = 0 without NaNs.
4. 1/√D with D = 16, `1 / math.sqrt(D)` in `Attention.__call__`, a Python float baked into the graph as a constant and folded into the multiply.
5. They change every step. As CONSTs they change the structure key, so every step misses the plan cache and recompiles (about 20–30× slower in the ablation). As 1-element buffers only their contents change.
6. Kernels are renumbered with local buffer slots, so identical layers (two blocks, repeated LayerNorms, forward and backward patterns) render to byte-identical C, and the content-addressed cache compiles each distinct source once.
7. So that every forward pass has the same graph shape and reuses one compiled plan. Causality guarantees the padding can't affect earlier positions.
8. Without staging, the broadcast computations (LayerNorm, GELU) are inlined into every consumer kernel, making much larger graphs for rangeify and `simplify` to process, which is where compile time goes (the cProfile in 14.11).

Further reading

- A. Vaswani et al., "Attention Is All You Need", NeurIPS 2017.
- A. Radford et al., "Language Models are Unsupervised Multitask Learners" (GPT-2), OpenAI 2019: pre-LayerNorm, GELU, weight tying.
- A. Karpathy, *nanoGPT* and *minGPT* (GitHub), and the lecture "Let's build GPT: from scratch, in code, spelled out" (2023). Our `examples/gpt.py` follows the same architecture. Also *char-rnn*, the source of the Tiny Shakespeare dataset.
- J. Ba, J. Kiros and G. Hinton, "Layer Normalization", 2016.
- D. Hendrycks and K. Gimpel, "Gaussian Error Linear Units (GELUs)", 2016.
- D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015. I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization", ICLR 2019.
- J. Kaplan et al., "Scaling Laws for Neural Language Models", 2020: the 6·N·D FLOP estimate.
- tinygrad: `examples/` (GPT-2 and LLaMA implementations) and its JIT (`TinyJit`), which caches whole schedules much like `_plans`.

Full code for this chapter

Every file this chapter builds, exactly as it is in the repository. Click a file to expand it.

```
examples/gpt.py · 106 lines
```

```
tests/test_w09_models.py · 38 lines
```

15. Symbolic Shapes, Inference, and Where to Go Next

You have a compiler that trains a GPT. Now ask it to *talk*. Generating text one token at a time means running the model on a prompt of length 5, then 6, then 7, and so on, and `uopc` treats every one of those lengths as a brand-new program: a new graph, a new plan, a new trip through gcc. A production system would also keep the keys and values of past tokens in a cache instead of recomputing them, run the matrix multiplies in half precision, and spread the work across several devices. None of that is in the reference compiler. This chapter is about the distance between what you built and what real systems do, with each missing piece measured and turned into a concrete design you can implement as a final project. It ends with a map of the real compilers you are now equipped to read, and with more than twenty project ideas.

What you'll build (all of it as design exercises; nothing in this chapter changes `uopc/`):

- A loop with a run-time trip count, in the week-3 loop IR, and a measurement of what it costs the symbolic engine.
- A bounded `Variable` prototype (a few lines of monkeypatching) that shows exactly which simplifications symbolic shapes need.
- A fixed-capacity KV cache built from today's Tensor API, verified against full causal attention, that compiles once and decodes every later token from cache.
- A mixed-precision GEMM (f16 storage, f32 accumulation) and a simulated data-parallel all-reduce, both checked against numpy.
- A comparison of tinygrad, TVM, Triton, XLA, MLIR/IREE, torch.compile and Halide, and a reading guide to tinygrad's source.

Where this fits: Chapter 14 trained a GPT and padded its sampling loop to a fixed length to dodge recompilation (NOTES 18). This chapter explains why that trick was necessary and what replaces it. Continue into the advanced track for guided compiler, GPU, dynamic execution, and runtime labs. The following reference material remains available in the appendices: the op reference (Appendix A), testing and debugging (Appendix B), the glossary and quiz bank (Appendix C), and the source listing (Appendix D).

> **Labeling convention for this chapter.** Code shown with an `@include` label is in the repository and tested. Code in a plain fenced block marked *scratch* is a small script written for this chapter: it imports `uopc` but is not part of it. Every output shown was produced by running that code on the reference laptop (i7-13620H, gcc 15, Python 3.12). Code marked *sketch* is a design proposal and has not been run.

15.1 An inventory of what is missing

Before designing anything, list what is missing and where in the code the gap shows. Each row is a section of this chapter or a project at the end.

Missing feature	Where you see the gap in `uopc`	Section
Shapes known only at run time	`UOp.shape` is `tuple[int, ...]`; `_structure_key` hashes shapes	15.2
A KV cache for decoding	`examples/gpt.py` re-runs the whole context for every sampled token	15.3
In-place updates	`Tensor.assign` rebinds to a new buffer; there is no "store into an existing buffer" op at the Tensor level	15.3
bf16, loss scaling	`dtypes` has `f16` but no `bf16`; optimizers are f32 only	15.4
More than one device	`BUFFER` has no device; `Program` is one process, one address space	15.5
Real GPU execution	`cuda.py` is untested on the reference machine (NOTES 25)	Projects
Tensor cores from the graph	`Ops.WMMA` is reserved; `wmma_gemm` is a string template	Projects

None of these are embarrassing. They are where the course stops and the field begins.

15.2 Symbolic shapes

Why static shapes cost compiles

The JIT cache in `kernelize.py` keys every plan on the *structure* of the graph. Buffers are anonymized, but their sizes are not, and neither are the shape arguments of movement ops:

`uopc/kernelize.py · _structure_key`

```python
def _structure_key(roots: list[UOp]) -> tuple:
  ids: dict[UOp, int] = {}
  key = []
  for u in UOp(Ops.SINK, tuple(roots)).toposort():
    ids[u] = len(ids)
    arg = ("buf", u.dtype.name, u.arg.shape) if u.op is Ops.BUFFER else _argkey(u.arg)
    key.append((u.op, arg, tuple(ids[s] for s in u.src)))
  return tuple(key)
```

That is the right key for training, where every step has the same shapes. It is the wrong key for generation. Here is a one-layer GPT run on three sequence lengths and then the first length again (scratch script, cold kernel cache):

```python
# scratch: how many plans does a forward pass compile when the sequence length varies?
import sys, time
sys.path.insert(0, "examples")
import numpy as np
from gpt import GPT
from uopc.tensor import Tensor
from uopc import kernelize

Tensor.manual_seed(0)
model = GPT(vocab=65, ctx=16, dim=32, heads=2, layers=1)
for T in (5, 6, 7, 5):
  t0 = time.perf_counter()
  model(Tensor(np.zeros((1, T), np.int32))).numpy()
  print(f"T={T:2d}: {(time.perf_counter() - t0) * 1e3:6.0f} ms, plans so far:
{len(kernelize._plans)}")
```

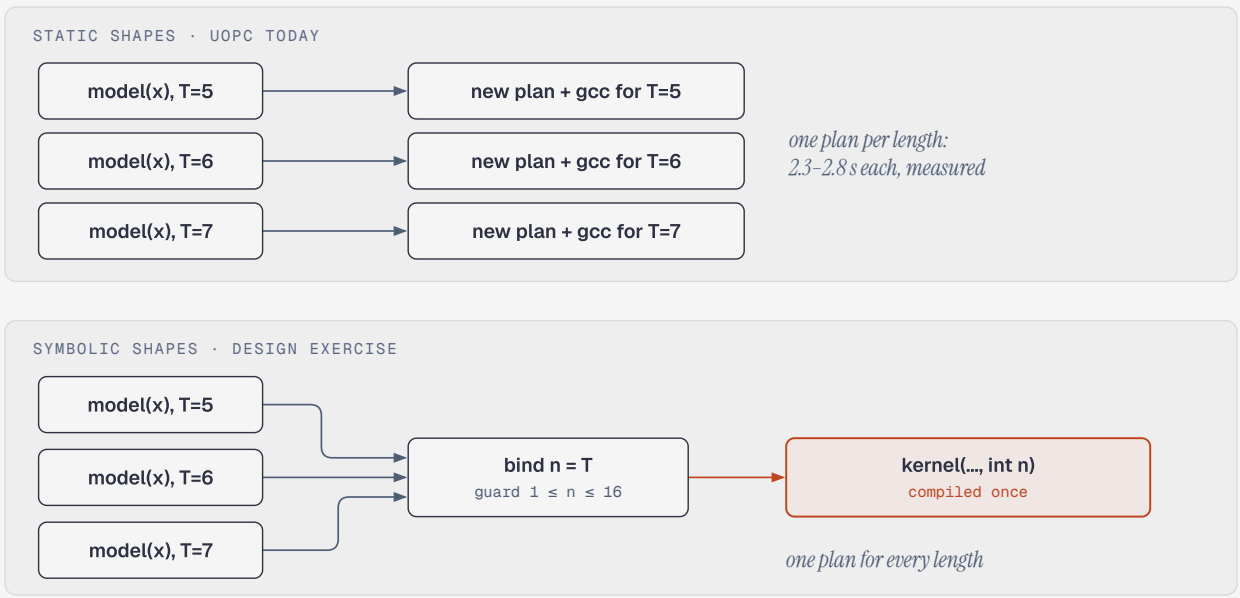
```
T= 5:   2787 ms, plans so far: 8
T= 6:   2432 ms, plans so far: 9
T= 7:   2289 ms, plans so far: 10
T= 5:      7 ms, plans so far: 10
```

(The first eight plans are the model's parameter initializers. Timings in this chapter are single runs on the reference laptop and vary by tens of percent between runs; plan counts and errors do not.) Each new length costs about two and a half seconds of rangeify, rendering and gcc; a repeated length costs 7 ms. A chatbot that answers with 300 tokens would compile 300 times. `examples/gpt.py` avoids this by padding:

`examples/gpt.py:95-102`

```
  # sample: feed the last ctx characters, draw the next one from the softmax
  idx = [stoi["\n"]]
  for _ in range(300):
    ctx = idx[-a.ctx:]
    ctx = ctx + [0] * (a.ctx - len(ctx))                          # fixed shape -> one compiled
graph
    logits = model(Tensor(np.array([ctx], np.int32))).numpy()[0, min(len(idx), a.ctx) - 1]
    p = np.exp(logits - logits.max()); p /= p.sum()
    idx.append(int(rng.choice(len(chars), p=p)))
```

Padding works, but it has a price: every token pays for the full context length, even the first one. Fixed shapes are the right tool when lengths cluster (pad to a few buckets: 64, 128, 256); they are the wrong tool when lengths vary one token at a time.



The loop IR is already symbolic

Look at what `RANGE` takes: its end is `src[0]`, an arbitrary `i32` UOp, not a Python int. So the week-3 loop language can already express loops whose trip count is only known at run time. This scratch program reads `n` from a one-element buffer and sums the first `n` columns of each row:

```python
# scratch: a loop whose trip count is read from memory at run time: out[i] = sum(x[i, :n])
import numpy as np
from uopc.uop import UOp, Ops, ParamArg, sink, range_
from uopc.dtype import dtypes
from uopc.render_c import render_c
from uopc.runtime import run_c
from uopc.symbolic import vmin, vmax

ROWS, MAXN = 4, 8
X   = UOp(Ops.BUFFER, (), ParamArg(0, dtypes.f32, (ROWS * MAXN,)))
LEN = UOp(Ops.BUFFER, (), ParamArg(1, dtypes.i32, (1,)))
OUT = UOp(Ops.BUFFER, (), ParamArg(2, dtypes.f32, (ROWS,)))
n = LEN.index(0).load()                       # the symbolic size: a value, not a python int
i, j = range_(ROWS, 0), range_(n, 1)          # RANGE's end is any i32 UOp
acc = UOp(Ops.ALLOC, (), ParamArg(0, dtypes.f32, (1,))).after(i)
init = acc.index(0).store(UOp.const(0.0))
accj = acc.after(init, j)
upd = accj.index(0).store(accj.index(0).load() + X.index(i * MAXN + j).load())
prog = sink(OUT.index(i).store(acc.after(upd.end(j)).index(0).load()).end(i))

src = render_c(prog)
print(src[src.index("void"):])
x = np.arange(ROWS * MAXN, dtype=np.float32)
for length in (3, 8):
  out = np.zeros(ROWS, np.float32)
  run_c(prog, {0: x, 1: np.array([length], np.int32), 2: out})
  print(f"n={length}: {out}  numpy: {x.reshape(ROWS, MAXN)[:, :length].sum(1)}")
print("bounds of j:", (vmin(j), vmax(j)), " bounds of i*8+j:", (vmin(i * MAXN + j), vmax(i * MAXN
+ j)))
```

```
void kernel(float* restrict data0, int* restrict data1, float* restrict data2) {
  int val1 = data1[0];
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    for (int ridx1 = 0; ridx1 < val1; ridx1++) {
      float val2 = acc0;
      float val3 = data0[((ridx0*8)+ridx1)];
      acc0 = (val2+val3);
    }
    float val4 = acc0;
    data2[ridx0] = val4;
  }
}

n=3: [ 3. 27. 51. 75.]  numpy: [ 3. 27. 51. 75.]
n=8: [ 28.  92. 156. 220.]  numpy: [ 28.  92. 156. 220.]
```

One compiled function, two lengths, correct answers. The linearizer even hoisted the load of `n` above both loops, because the scope of `RANGE` is the scope of its end plus itself (Chapter 5). So the back end needs nothing new. The trouble is upstream, in the algebra:

```
bounds of j: (0, 2147483646)  bounds of i*8+j: (-2147483648, 2147483647)
```

`vmax(j)` is `vmax(n) - 1`, and a loaded `i32` can be anything, so `j` might be two billion. Then `i*8 + j` might overflow `i32`, and `bounds()` does the only sound thing: it gives up and returns the full range. That cascades. Here is the index `(i*8 + j) // 8` simplified with a static `j` (end 8) and with the dynamic one (the same scratch script, continued):

```python
from uopc.symbolic import simplify
from uopc.uir import to_uir
j8 = range_(MAXN, 1)
print("static  j:", to_uir(simplify((i * MAXN + j8) // MAXN)).strip().splitlines()[-1])
print("dynamic j:"); print(to_uir(simplify((i * MAXN + j) // MAXN)), end="")
```

```
static  j: %0 = range 4 : idx=0 kind=loop
dynamic j:
%0 = range 4 : idx=0 kind=loop
%1 = buffer 1 : dtype=i32 slot=1
%2 = index %1, 0
%3 = load %2
%4 = range %3 : idx=1 kind=loop
%5 = floordiv %4, 8
%6 = add %0, %5
```

With `j < 8` known, `j // 8` folds to 0 and the expression is just `i`. With `j` unbounded, `_div_sum` still splits off `i` but `j // 8` survives. This is NOTES 12 again: an unsimplified index mentions a loop variable it does not really depend on, and the reduce lowering puts accumulators inside loops they do not need. Every `RESHAPE` of a symbolic axis produces exactly these `//` and `%` chains.

> **Key idea:** symbolic shapes are not a back-end problem. The loop IR and the C renderer already handle a run-time trip count. What breaks is *bounds analysis*, and everything that depends on it: div/mod simplification, the `/` versus `floordiv_i32` choice in the renderer, and dependence analysis in rangeify.

A bounded Variable

The fix every production system converges on is to give a symbolic size *declared bounds*: "the sequence length is between 1 and 2048". tinygrad calls this a `Variable` with a min and a max; torch.compile attaches value ranges to its symbolic integers. In `uopc` the smallest version is a scalar `i32` leaf whose bounds come from a declaration instead of from `_bounds`. Here is that idea as a short monkeypatch, using a `PARAM` as the leaf (scratch):

```
# scratch: a symbolic-shape variable is a PARAM with declared bounds. Not in uopc --
# we monkeypatch bounds analysis to show what the extra knowledge buys.
import uopc.symbolic as S
from uopc.uop import UOp, Ops, ParamArg, range_
from uopc.dtype import dtypes
from uopc.uir import to_uir

DECLARED: dict[UOp, tuple[int, int]] = {}
def Variable(name_slot: int, lo: int, hi: int) -> UOp:
  v = UOp(Ops.PARAM, (), ParamArg(name_slot, dtypes.i32))
  DECLARED[v] = (lo, hi)
  return v

_old = S._bounds
S._bounds = lambda u: DECLARED.get(u) or _old(u)

n = Variable(0, 1, 8)                    # sequence length: 1 <= n <= 8
i, j = range_(4, 0), range_(n, 1)
print("bounds of j:", (S.vmin(j), S.vmax(j)))
print(to_uir(S.simplify((i * 8 + j) // 8)), end="")
print(to_uir(S.simplify((i * 8 + j) % 8)), end="")
print(to_uir(S.simplify(j < n)), end="")
```

```
bounds of j: (0, 7)
%0 = range 4 : idx=0 kind=loop
%0 = param : slot=0 dtype=i32
%1 = range %0 : idx=1 kind=loop
%0 = param : slot=0 dtype=i32
%1 = range %0 : idx=1 kind=loop
%2 = cmplt %1, %0
```

The existing rule `RANGE: (0, vmax(end) - 1)` now yields `j ∈ [0, 7]`, and both div/mod identities fire: `(i*8 + j) // 8` is `i` and `(i*8 + j) % 8` is `j`. Nothing else in `symbolic.py` changed.

The last line is instructive. `j < n` is always true inside the loop, but interval arithmetic cannot see it: `j ∈ [0, 7]` and `n ∈ [1, 8]` overlap. That fact is *relational* (it relates two variables), and intervals only know about one variable at a time. You need a rule that knows what a `RANGE` means:

```
# sketch: a relational rule for symbolic.py -- not in uopc, not run
(UPat(Ops.CMPLT, src=(UPat(Ops.RANGE, name="r"), UPat.var("e"))),
 lambda r, e: UOp.const(True) if r.src[0] is e else None),
```

Masks of the form `j < n` are everywhere once shapes are symbolic (causal attention, padding), so this rule matters.

The full design

A real implementation touches most modules. Here is the work list, roughly in the order you would do it.

1. **`uop.py`: an op and a shape type.** Add `Ops.VAR` with `arg = VarArg(name: str, lo: int, hi: int)`, dtype `i32`, shape `()`. Let shapes contain UOps: `tuple[int | UOp, ...]`. `_infer` then needs symbolic equality for its checks: `RESHAPE` must verify `prod(arg) == prod(src.shape)` where both sides may be expressions. Use `simplify(a - b)` and accept only if it folds to `CONST 0`; reject otherwise (a sound but incomplete check, like every real system's).
2. **`symbolic.py`: bounds and rules.** `_bounds(VAR) = (lo, hi)`; the relational `RANGE < end` rule above; and a rule that `x // n` with `0 <= x < n` is 0 when `n` is symbolic (the existing rule needs a `CONST` divisor).
3. **`rangeify.py`: almost free.** `ravel` and `unravel` (listed below) multiply and divide by shape entries using `UOp` arithmetic, so they already accept UOp sizes. `Ctx.rng(size)` must accept a UOp end, and `_axis_parts` must refuse to upcast a symbolic axis (an unroll factor must divide the size, and you cannot prove that for `n`). Reductions over a symbolic axis are just loops with a symbolic end.
4. **`kernelize.py`: plans with holes.** `_structure_key` must key a `VAR` by `(name, lo, hi)`, never by its current value; temp buffers are allocated at the upper bound `hi`. `realize` takes the bound values, checks `lo <= v <= hi` (the *guard*), and passes them to every kernel.
5. **`render_c.py` and `runtime.py`: one more parameter.** A `VAR` renders as an `int` parameter of the kernel (`void k(float* restrict data0, ..., int n)`), and `Program.__call__` passes it with `ctypes.c_int`. The CUDA renderer does the same; launch dimensions become expressions evaluated at launch.
6. **`tensor.py`: binding.** A user-facing `Tensor` axis can be a `VAR`. tinygrad's API binds a value at the call site (`Variable("start_pos", 1, MAX).bind(t)`); you can copy that shape of API.

Item 3 is the pleasant surprise. Here is `ravel`; nothing in it requires `s` to be a Python int, because `flat * s` is `UOp.__mul__`, which accepts a UOp operand:

`uopc/rangeify.py · ravel`

```
def ravel(idx: tuple[UOp, ...], shape: tuple[int, ...]) -> UOp:
  flat = UOp.const(0)
  for i, s in zip(idx, shape): flat = flat * s + i
  return flat
```

What does this buy for GPT? One compiled plan for every prompt length up to the context size, and, with a KV cache, decoding work proportional to the current position rather than to the maximum. Which brings us to inference.

15.3 Inference and the KV cache

What decoding recomputes

Causal attention has a useful property: the output at position `t` depends only on tokens `0..t`. So when you append token `t+1`, nothing about the keys and values of tokens `0..t` changes. A sampler that re-runs the model on the whole context recomputes all of them, at every layer, for every new token. For a context of length `T` that is `O(T)` redundant work per token and `O(T²)` per generated sequence, before even counting attention itself.

A **KV cache** stores, for each layer and head, the key and value vectors of every past token. Each decode step then computes `q, k, v` for one new token, appends `k, v` to the cache, and attends with one query against the cached keys. The memory cost is easy to count: two tensors (K and V) per layer, each `heads × T × head_dim` elements:

```
bytes = 2 · layers · T · dim · bytes_per_element
```

For the default model in `examples/gpt.py` (2 layers, `dim` 64, context 64, f32) that is 2·2·64·64·4 = 64 KiB. For a model with 32 layers, `dim` 4096 and a 4096-token context in f16 it is 2·32·4096·4096·2 bytes = 2 GiB *per sequence*, which is why serving systems treat cache memory as their scarcest resource (see PagedAttention in Further reading).

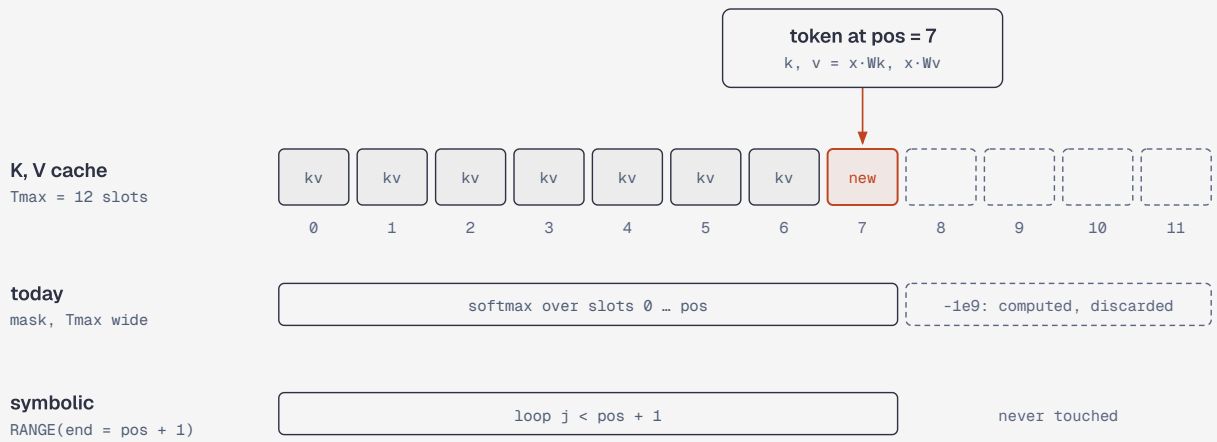
A fixed-capacity cache with today's API

You can build a correct KV cache right now, without symbolic shapes, by making the *position* data instead of shape: a one-element `i32` tensor. Every decode step then has the same graph, so it reuses one plan. AdamW uses the same trick for its bias corrections (NOTES 17). The cache has a fixed capacity `TMAX`; the write is a functional scatter through a one-hot row; unused slots are masked to `-1e9` before the softmax. One attention head, scratch:

```python
# scratch: a fixed-capacity KV cache built from today's Tensor API: no symbolic shapes needed,
# the position is *data* (a 1-element i32 tensor), so every decode step has the same graph.
import math, time
import numpy as np
from uopc.tensor import Tensor
from uopc import kernelize

TMAX, C, D = 64, 32, 16
rng = np.random.default_rng(0)
Wq, Wk, Wv = (Tensor(rng.standard_normal((C, D)).astype(np.float32) / math.sqrt(C)) for _ in
range(3))
xs = rng.standard_normal((TMAX, C)).astype(np.float32)

def reference(T):                                   # causal attention over the first T tokens,
in numpy
  q, k, v = (xs[:T] @ W.numpy() for W in (Wq, Wk, Wv))
  s = q @ k.T / math.sqrt(D) + np.triu(np.full((T, T), -np.inf), 1)
  p = np.exp(s - s.max(-1, keepdims=True)); p /= p.sum(-1, keepdims=True)
  return p @ v

K, V = Tensor.zeros(TMAX, D).realize(), Tensor.zeros(TMAX, D).realize()
slots = Tensor.arange(TMAX)
def decode_step(x: Tensor, pos: Tensor) -> Tensor:
  global K, V
  q, k, v = x @ Wq, x @ Wk, x @ Wv                  # (1, D) each
  here = slots.eq(pos).float().reshape(TMAX, 1)     # one-hot row `pos`: a functional scatter
  K = (K * (1 - here) + here * k).realize()
  V = (V * (1 - here) + here * v).realize()
  seen = slots.reshape(1, TMAX) < pos.reshape(1, 1) + 1   # slots 0..pos are valid
  s = (q @ K.T) * (1 / math.sqrt(D))
  p = seen.where(s, -1e9).softmax(-1)
  return p @ V                                      # (1, D)

outs, plans, times = [], [], []
for t in range(TMAX):
  t0 = time.perf_counter()
  outs.append(decode_step(Tensor(xs[t:t + 1]), Tensor(np.array([t], np.int32))).numpy())
  times.append(time.perf_counter() - t0)
  plans.append(len(kernelize._plans))
print("max |err| vs full causal attention:", float(np.abs(np.concatenate(outs) -
reference(TMAX)).max()))
print("compiled plans after step 1, 2, 64:", plans[0], plans[1], plans[-1])
print(f"step 1: {times[0]*1e3:.0f} ms (compiles)   steps 2-{TMAX}: {np.mean(times[1:])*1e3:.2f}
ms each")
```

```
max |err| vs full causal attention: 8.156349239896166e-07
compiled plans after step 1, 2, 64: 3 3 3
step 1: 114 ms (compiles)   steps 2-64: 4.00 ms each
```

Three plans (the K update, the V update, and the attention output) are built on the first step and reused for all 63 others. The 64 incremental outputs match full causal attention computed by numpy to within `1e-6`. (Step 1 was cheap here because the kernels were already in the on-disk `.so` cache from an earlier run; the plan still had to be built in Python.)



Per token: Θ(Tmax·D) work with a mask; Θ(pos·D) with a symbolic loop end.

The diagram shows what this version still wastes. Every step rewrites the whole `TMAX × D` cache (the one-hot scatter is a full elementwise pass), and attention scores all `TMAX` slots, then throws away the ones past `pos`. Both costs are `Θ(TMAX · D)` per token where `Θ(pos · D)` would do. Two features remove them:

- **An in-place store.** Writing one row should be one `STORE` at index `pos * D + d`, into the *existing* buffer. `uopc` can already express that in the loop IR (it is just `BUFFER.index(...).store(...)`); what is missing is a Tensor-level op that means "mutate this buffer", and a kernelizer that respects it: every kernel that reads the cache must be ordered after the write (an `AFTER` edge between kernels), and the structural cache must treat the cache buffer as an input *and* an output of the same plan.
- **A symbolic loop end.** With `pos` as a bounded `VAR`, the attention reduce runs `RANGE(end = pos + 1)` and the mask disappears.

> **Key idea:** the fixed-capacity cache is the "pad to a bucket" strategy applied to the time axis. It is correct and compiles once, but its cost is set by the capacity, not by the work. Symbolic shapes and in-place updates are what make the cost follow the work.

Batching and the rest of an inference stack

Real serving adds two more ideas worth knowing by name. **Continuous batching** groups requests of different lengths into one batch and lets them join and leave between decode steps; it needs either symbolic batch sizes or padding to buckets. **Paged KV caches** (vLLM's PagedAttention) allocate the cache in fixed-size blocks, like virtual memory pages, and keep a per-sequence block table; attention then gathers keys through an indirection. In `uopc` terms, that indirection is an `INDEX` whose index is itself a `LOAD`: a data-dependent address, which the loop IR already supports and the Tensor API does not (there is no gather; `autodiff` refuses a data-dependent `INDEX`).

15.4 Mixed precision

Half-precision floats (f16) halve memory traffic and, on GPUs with tensor cores, multiply peak FLOP/s several times over. The catch is range and precision: f16 has a 10-bit mantissa and a maximum of 65504. The standard recipe is *store in low precision, accumulate in high precision.* `uopc` has an `f16` dtype (rendered as `_Float16`, which gcc supports on x86-64, and as `half` by the CUDA renderer), so you can measure the recipe directly. Scratch:

```
# scratch: mixed precision on a CPU: f16 storage, f32 arithmetic.
import numpy as np
from uopc.tensor import Tensor
from uopc.dtype import dtypes
from uopc.kernelize import make_plan
rng = np.random.default_rng(0)
a, b = rng.standard_normal((64, 64)).astype(np.float32), rng.standard_normal((64,
64)).astype(np.float32)
A16, B16 = Tensor(a, dtype=dtypes.f16), Tensor(b, dtype=dtypes.f16)
c = A16.float() @ B16.float()                      # load halves, multiply-accumulate in f32
ref = a.astype(np.float16).astype(np.float64) @ b.astype(np.float16).astype(np.float64)
print("max |err| vs f64 on the same f16 inputs:", float(np.abs(c.numpy() - ref).max()))
print("max |err| of f16 rounding of the inputs:", float(np.abs(a @ b - ref).max()))
cc = (A16 @ B16)                                   # everything in f16, including the accumulator
print("all-f16 GEMM max |err|:", float(np.abs(cc.numpy().astype(np.float64) - ref).max()))
plan, _ = make_plan([(A16.float() @ B16.float()).uop])   # unoptimized, to read it
src = plan.kernels[-1].src
print(src[src.index("void"):])
```

```
max |err| vs f64 on the same f16 inputs: 8.231960237026215e-06
max |err| of f16 rounding of the inputs: 0.009531687945127487
all-f16 GEMM max |err|: 0.06902523338794708
void r_64_64(_Float16* restrict data0, _Float16* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 64; ridx0++) {
    int alu1 = (ridx0*64);
    for (int ridx1 = 0; ridx1 < 64; ridx1++) {
      float acc0;
      acc0 = 0.0f;
      for (int ridx2 = 0; ridx2 < 64; ridx2++) {
        float val2 = acc0;
        _Float16 val3 = data0[(alu1+ridx2)];
        _Float16 val4 = data1[((ridx2*64)+ridx1)];
        acc0 = (val2+(((float)(val3))*((float)(val4))));
      }
      float val5 = acc0;
      data2[(alu1+ridx1)] = val5;
    }
  }
}
```

Read the three errors in order. Rounding the *inputs* to f16 changes the product by about `1e-2`; that is the price of half-precision storage. Accumulating those f16 inputs in f32 adds only `8e-6` on top. Accumulating in f16 adds `7e-2`, almost ten thousand times more, from just 64 additions per output. The kernel shows the recipe at the source level: the loads are `_Float16`, the `CAST` nodes became `(float)` conversions fused into the multiply, and the accumulator is a `float`. Tensor cores do exactly this in hardware: f16 or bf16 inputs, f32 accumulators (Chapter 13).

Training adds a second problem: gradients are often tiny, and f16's smallest normal number is about `6e-5`. Mixed-precision training (Micikevicius et al.) keeps an f32 *master copy* of the weights, computes the forward and backward passes in f16, and multiplies the loss by a *loss scale* (say 1024) before `backward` so that gradients land in f16's representable range; the optimizer divides it back out, in f32. **bfloat16** sidesteps most of this: it keeps f32's 8-bit exponent and cuts the mantissa to 7 bits, so it has f32's range and needs no loss scaling. It is not in `uopc`. Adding it is a good small project: C has no portable `bf16` type, so store it as a 16-bit integer, convert to f32 by shifting left 16 bits, and convert back with round-to-nearest-even, all of which you can express with `CAST`, `MUL` and bit tricks in the renderer.

15.5 Multiple devices

Every `uopc` buffer lives in one address space and every kernel runs in one process. Scaling past one device means choosing what to split.

- **Data parallelism** splits the batch. Each device holds a full copy of the weights, computes gradients on its shard, and the devices *all-reduce* (sum, then share) the gradients before the optimizer step.
- **Tensor parallelism** splits individual matrices. A linear layer `Y = X W` with `W` split by columns gives each device a slice of `Y` (concatenate to finish); split by rows, each device produces a partial sum of `Y` (all-reduce to finish). Megatron-LM pairs a column split with a row split so that an MLP needs only one all-reduce.
- **Pipeline parallelism** splits the layers: device 0 runs layers 1–8, device 1 runs 9–16, and micro-batches flow through like an assembly line.

Data parallelism is pure math, so you can check it today. The gradient of a mean loss over a batch is the mean of the gradients over two equal halves (scratch):

```
# scratch: data parallelism, simulated: two "devices" each see half the batch; averaging their
# gradients (an all-reduce) gives exactly the full-batch gradient.
import numpy as np
from uopc.tensor import Tensor
from uopc.nn import Linear

Tensor.manual_seed(0)
layer = Linear(8, 4)
rng = np.random.default_rng(0)
X, Y = rng.standard_normal((32, 8)).astype(np.float32), rng.integers(0, 4, 32).astype(np.int32)
params = [layer.weight, layer.bias]

def grads(x, y):
  for p in params: p.grad = None
  layer(Tensor(x)).cross_entropy(Tensor(y)).backward(params)
  return [p.grad.numpy() for p in params]

full = grads(X, Y)
shards = [grads(X[:16], Y[:16]), grads(X[16:], Y[16:])]        # device 0, device 1
allreduced = [(g0 + g1) / 2 for g0, g1 in zip(*shards)]         # the all-reduce
print("max |full - allreduced|:", max(float(np.abs(a - b).max()) for a, b in zip(full,
allreduced)))
```

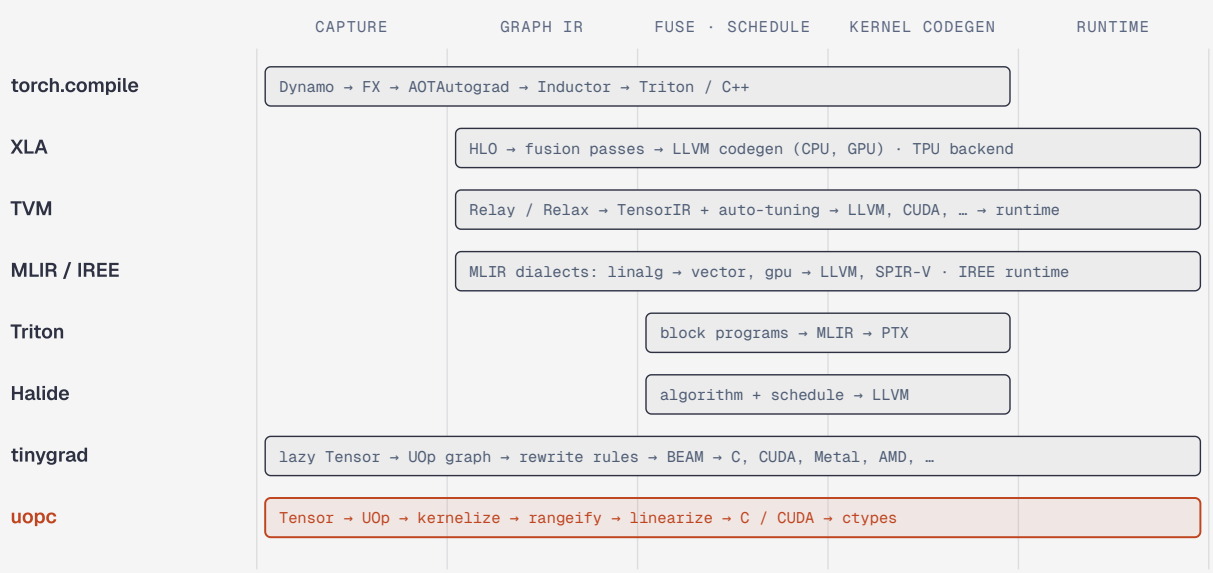
```
max |full - allreduced|: 1.4901161193847656e-08
```

Equal up to f32 rounding of the summation order. The compiler work for a real implementation:

1. **A device on every buffer.** Add `device: str` to `ParamArg` for `BUFFER`. `_infer` rejects an ALU op whose operands live on different devices, exactly like it rejects mismatched dtypes today.
2. **A `COPY` op** (`src = (buffer,)`, `arg = target device`), the only op allowed to cross devices.
3. **Kernelize cuts at copies.** A `COPY` is a materialization point, like `STAGE`, and is executed by the runtime (a `memcpy`, `cuMemcpyPeer`, or a socket send), not by a generated kernel.
4. **All-reduce as a composite** of copies and adds. The bandwidth-optimal *ring* all-reduce sends each gradient around a ring of `p` devices in `2(p - 1)` steps of `N/p` elements, so each device transfers about `2N` elements regardless of `p`.

15.6 Real compilers compared

You now know enough to read the design documents of production ML compilers and see your own decisions in them. They all do the same five jobs: **capture** a program, represent it in a **graph IR**, decide **fusion and scheduling**, **generate kernels**, and **run** them. They differ in where they draw the boundaries, and in which jobs they leave to you.



Every system does the same five jobs; they disagree about where the boundaries go.

The table summarizes the public designs as of this writing. These projects move quickly; treat specifics as pointers to check in their documentation, not as gospel.

System	You write	Main IR(s)	Who fuses	Schedule and tuning	Dynamic shapes
uopc	lazy Python Tensor code	one UOp graph, rewritten in stages	`kerneliz e` rules + a recompute cost rule	`hand_opt` heuristics; `search_opt` timing (`OPT=2`)	none (static)
tinygrad	lazy Python Tensor code	one UOp graph, rewritten in stages	automatic, rule-based	heuristics; BEAM search over optimizations	bounded symbolic `Variable`s
TVM	imported models (ONNX, PyTorch, ...) or tensor expressions	Relay / Relax (graph), TensorIR (loops)	graph-level fusion passes	schedule primitives; AutoTVM, Ansor, MetaSchedule	symbolic shapes in Relax
Triton	one kernel per function, over *blocks*	Triton's MLIR dialects → LLVM	you do (a kernel is a fusion)	`@triton.autotune` over configs you list	sizes are runtime arguments
XLA	JAX, TensorFlow, PyTorch/XLA programs	HLO (StableHLO for exchange)	automatic fusion passes	heuristics and cost models	mostly static; recompiles per shape
MLIR / IREE	models via front ends (StableHLO, TOSA, torch-mlir)	MLIR dialects (linalg, vector, gpu, ...)	dialect-level fusion in IREE	pass pipelines	dynamic dimensions in tensor types
torch.compile	ordinary PyTorch, captured from Python bytecode	FX graphs → Inductor's loop IR	Inductor (pointwise and reduction fusion)	heuristics; `max-autotune` tries kernel templates	symbolic ints with guards; recompile on guard failure
Halide	an algorithm plus a separate	Halide IR	the schedule (`compute_`	hand schedules; autoschedulers	extents are run-time; bounds inference

System	You write	Main IR(s)	Who fuses	Schedule and tuning	Dynamic shapes
	schedule (C++ DSL)		`at`, `compute_root`, inline)		

What to steal from each

tinygrad is `uopc`'s direct ancestor: the hash-consed `UOp`, `UPat` / `PatternMatcher`, `graph_rewrite`, rangeify, a linearizer and C-style renderers all have tinygrad counterparts, and the course syllabus is built around its design. Read it for symbolic `Variable`s with bounds (its LLaMA example uses one for the KV-cache position), for its many renderers sharing one C-style base class, and for BEAM search.

TVM made the *compute/schedule split* (from Halide) the center of an ML compiler: you describe *what* to compute, then apply schedule primitives (`split`, `reorder`, `tile`, `vectorize`, `bind` to GPU threads) that change *how* without changing the result. `uopc`'s `Opts` is a tiny schedule language: `upcast`, `thread`, `unroll`, `tile`, and the GPU mapping (`gpu`, `local`). TVM's Ansor and MetaSchedule search a much larger space with a learned cost model. It is a much larger version of `search_opt`.

Triton moves the boundary the other way. You write one kernel at the level of *blocks* (`tl.load` of a whole tile, `tl.dot` of two tiles), and the compiler handles what Chapters 12 and 13 did by hand: coalescing, shared-memory staging, and tensor-core instructions. Fusion is your job, because a Triton kernel is itself a fusion. torch.compile's Inductor generates Triton code, which is a nice division of labor: Inductor decides what to fuse, Triton decides how to run a block.

XLA pioneered aggressive automatic fusion of whole-program graphs (HLO) for TPUs and GPUs. It prefers static shapes, as `uopc` does, and pays with recompilation when shapes change. You measured the same trade-off in 15.2.

MLIR is not a compiler but a framework for building them: many *dialects* (IRs) at different levels, with conversions between them. `uopc` uses one IR at every level, which is simpler but means a single op set must serve the Tensor API and the C renderer (see the op-lifetime diagram in Appendix A). IREE is an end-to-end compiler and runtime built on MLIR.

torch.compile solves the capture problem that every lazy-tensor design (including `uopc`) dodges: real PyTorch code is eager, full of Python control flow. TorchDynamo captures graphs from Python bytecode and inserts *guards* (checks on shapes, types and values) that trigger recompilation when they fail;

AOTAutograd traces the backward pass ahead of time. Its symbolic shapes are the closest production analogue of the design in 15.2.

Halide is the origin of the algorithm/schedule split, from image processing. Its *bounds inference*, which computes how much of each input a tile of output needs, is the same reasoning as `uopc`'s `vmin` / `vmax` over index expressions, applied to whole regions.

15.7 Reading tinygrad after this book

tinygrad is the best next codebase: it is Python, it is small for what it does, and you already know its vocabulary. Its file layout changes between releases, so navigate by names, not paths:

In `uopc`	Look for in tinygrad	Notes
`uop.py`: `UOp`, `Ops`, `_infer`	`class UOp`, `class Ops`	far more ops (and more dtypes); shape tracking has changed over versions
`rewrite.py`: `UPat`, `PatternMatcher`, `graph_rewrite`	the same names	adds compiled matchers and rewrite contexts
`symbolic.py`: `simplify`, `vmin` / `vmax`	symbolic rule sets, `vmin` / `vmax`	plus `Variable` with bounds
`rangeify.py`: `rangeify`, `lower`	rangeify	the same idea, with many more cases
`kernelize.py`: materialization, `realize`	the scheduler	kernel boundaries, buffer assignment
`opt.py`: `Opts`, `search_opt`	kernel optimization actions, BEAM	`BEAM=<n>` turns search on
`linearize.py`	the linearizer	
`render_c.py`, `gpu.py`	renderers (a C-style base class with per-backend subclasses)	compare with `CRenderer` / `CUDARenderer`
`DEBUG=1..4`	`DEBUG=` levels	the same convention: higher prints more

Three habits make reading it productive. First, run a three-line program with high `DEBUG` and read the kernels before reading any source. Second, read the *tests* for a component before the component: they tell you what it is for. Third, follow one tensor from `Tensor.__add__` down to the generated code with a debugger; you will recognize every stage.

15.8 Final projects

Weeks 11 onward are yours. The syllabus says "implement a paper, port to strange hardware, mostly anything." Good projects have a **measurable** outcome (a speedup, a model that runs, a test suite that passes) and touch **one or two** modules deeply rather than everything shallowly. Difficulty: ★ about one week, ★★ two to three weeks, ★★★ the whole project period.

#	Project	★	What you change	How you know it works
1	Fixed-capacity KV-cache sampler for `examples/gpt.py` (15.3, generalized to all layers and heads)	★	`examples/` only	samples match the padded sampler token for token; per-token time vs padding
2	Graph visualizer: UOp graph → Graphviz `dot`, with kernel boundaries colored	★	new tool	renders the GPT step; used in your report
3	Cross-compile to a Raspberry Pi or RISC-V board (`CC`, `CFLAGS`) and benchmark GEMM	★	`runtime.py` flags	test suite passes on the board; roofline for it
4	Relational symbolic rules (`RANGE < end`, `x // n` for symbolic `n`) with a property-test suite	★	`symbolic.py`	property tests (Appendix B) with 1,000+ seeds
5	Bounded symbolic `Variable` end to end (15.2)	★★★	`uop`, `symbolic`, `rangeify`, `kernelize`, `render_c`	GPT forward on lengths 1..64 compiles one plan
6	In-place `ASSIGN` op and a true KV cache	★★★	`tensor`, `kernelize`	per-token cost grows with `pos`, not `TMAX`
7	bf16 dtype with correct round-to-nearest-even	★★	`dtype`, `render_c`, `symbolic`	bit-exact against a numpy reference conversion
8	Mixed-precision training with dynamic loss scaling	★★	`nn.py`	GPT loss curve within noise of f32
9	int8 weight quantization with per-channel scales	★★	new dtype, `nn.py`	perplexity delta vs f32; GEMM speed
10	Run and debug the CUDA path on a real GPU; compare with cuBLAS/PyTorch	★★	`cuda.py`, `gpu.py`	`test_cuda_smem_gemm_on_gpu` passes; GFLOP/s table
11	A HIP (AMD) or OpenCL renderer	★★	new renderer subclass	the week-7 tests pass through it

#	Project	★	What you change	How you know it works
12	A Metal renderer (Apple GPUs)	★★	new renderer subclass	same
13	WebGPU/WGSL renderer running in a browser	★★★	new renderer + JS runtime	MNIST inference in a web page
14	Lower `WMMA` from the graph (pattern-match GEMM tiles, emit fragments)	★★★	`gpu.py`, `render_c.py`	tensor-core GEMM matches f32 within f16 error
15	Double-buffered shared-memory GEMM with asynchronous copies	★★★	`gpu.py`	GFLOP/s vs `smem_gemm_ir`
16	Explicit SIMD: render vector types or intrinsics for UPCAST groups	★★	`render_c.py`	GEMM GFLOP/s beyond the 114 of Chapter 10
17	Persistent tuning cache: store `search_opt` winners on disk keyed by kernel source	★	`opt.py`	second run of `OPT=2` compiles no candidates
18	Learned cost model for `search_opt` (features of the AST → predicted time)	★★★	`opt.py`	ranks candidates better than `hand_opt` on held-out kernels
19	Memory planner: reuse temp buffers by liveness	★★	`kernelize.py`	peak temp memory of a GPT step drops; same outputs
20	Gradient checkpointing (recompute activations in backward)	★★	`autodiff.py`, `tensor.py`	peak memory vs extra FLOPs, measured
21	Gather/scatter ops: real embedding lookup and its gradient	★★	`uop`, `rangeify`, `autodiff`	`Embedding` without the one-hot matmul; speedup
22	Direct convolution kernels (no im2col) or Winograd F(2,3)	★★★	`tensor.py`, `opt.py`	conv2d GFLOP/s vs im2col
23	Data-parallel training across processes with shared-memory all-reduce	★★	new runtime layer	2-process loss curve equals 1-process curve

#	Project	★	What you change	How you know it works
24	Tensor-parallel linear layers across two "devices" (`COPY` op)	★★★	`uop`, `kernelize`, `runtime`	outputs equal single-device; copy volume measured
25	Implement FlashAttention-2's backward pass in the loop IR	★★★	`examples/`	gradient check against unfused attention
26	Whole-compiler fuzzer: random Tensor programs vs numpy	★★	new test file	finds (and you fix or report) at least one bug
27	ONNX importer: load a small model into uopc Tensors	★★	new front end	ResNet-18 or similar matches onnxruntime outputs
28	Cut compile time: memoize rangeify per kernel structure, profile `make_plan`	★★	`kernelize`, `rangeify`	GPT first-step compile time (Chapter 14's ~9 s) halved

Whatever you choose, write the tests first, keep a lab notebook of measurements the way `book/NOTES.md` does, and keep the code as small as you can. The course rule still applies: if you let slop in now, it compounds.

Pitfalls

- **Python values in the graph are shapes in disguise.** Anything that varies per call and is baked into the graph as a `CONST` or a shape makes every call structurally different and recompiles everything. It happened with AdamW's bias corrections (NOTES 17) and with sampling lengths (NOTES 18); the fixes were the same: make the varying thing *data* (a one-element tensor) or make the shape fixed (padding).
- **Unbounded symbolic values poison bounds analysis.** A trip count loaded from memory gives `j` an upper bound of `2^31 - 2`, `i*8 + j` might overflow, and `bounds()` correctly returns "anything" (15.2). Every rule gated on bounds then silently stops firing; the code stays correct but gets slower. Always attach declared bounds to symbolic sizes.

- **Interval arithmetic cannot prove relational facts.** `j < n` inside `for (j = 0; j < n; j++)` is obviously true and unprovable with intervals. You need a rule that knows what `RANGE` means.
- **Accumulating in low precision.** An all-f16 64-term dot product had an error of `7e-2` versus `8e-6` with an f32 accumulator, on the same f16 inputs (15.4). Cast up before the reduce, not after it.
- **Masking is not skipping.** A masked fixed-capacity cache computes scores for empty slots and throws them away; correct output can hide `O(TMAX)` waste. Measure per-token time as `pos` grows to see it.

Exercises

1. ★ **Predict the bounds.** In the scratch program of 15.2, what does `vmin(i * MAXN + j)` become if `LEN` is replaced by a `CONST 8`? Why?
2. ★ **Multi-head KV cache.** Generalize the fixed-capacity KV cache of 15.3 to `H` heads by giving `K` and `V` shape `(H, TMAX, D)`. Check against the numpy reference per head.
3. ★ **Masking waste.** Measure the masking waste: time `decode_step` for `TMAX = 64, 256, 1024` at `pos = 3`. Plot time vs `TMAX`.
4. ★★ **A relational rule.** Implement the relational rule `RANGE r < r.src[0] → True` in a scratch copy of `symbolic.py`, and write a test that `simplify(j < n)` folds for both a `CONST` and a `PARAM` end.
5. ★★ **Find the bug.** A student adds `Ops.VAR` and makes `_structure_key` include `u.arg` for it, where `arg = VarArg(name, lo, hi, value)`. Every decode step recompiles. Why, and what is the fix?
6. ★★ **bf16 in C.** Implement bf16 conversion functions in C (`float bf16_to_f32(unsigned short)` and the reverse with round-to-nearest-even) and test them exhaustively against numpy on all 65,536 bit patterns.
7. ★★ **Loss scaling.** Add dynamic loss scaling to a scratch copy of `AdamW`: halve the scale on any non-finite gradient and skip the step; double it after 1,000 clean steps.
8. ★★ **Tensor parallelism.** Write tensor parallelism for one linear layer by hand: split `W` by columns into two `Linear`s, run both, concatenate with `Tensor.cat`, and compare against the unsplit layer. Then split by rows and use a sum instead.
9. ★★★ **Design `VAR`.** Design the `VAR` op on paper: its `_infer` rule, its `_bounds`, its rendering, and the changes to `_structure_key` and `realize`. Identify every place in `uopc/` that assumes a shape entry is an `int` (hint: grep for `prod(`).

Solutions to ★ exercises

1. With `end = CONST 8`, `_bounds(RANGE) = (0, 7)`, so `i*8 + j` has bounds `(0*8 + 0, 3*8 + 7) = (0, 31)`; `vmin` is `0`. Nothing can overflow, so bounds are exact. (This is the static case tested in `test_symbolic_ranges`.)

2. Make `here` shape `(1, TMAX, 1)`, `k` shape `(H, 1, D)` (reshape the `(1, H·D)` projection), and the update `K * (1 - here) + here * k`. Scores are `q (H, 1, D) @ K.transpose() (H, D, TMAX)`, masked with `seen` reshaped to `(1, 1, TMAX)`. The numpy reference is the single-head one applied to each head's `D`-wide slice of the projections.

3. Wrap the loop in a function of `TMAX`; skip the first (compiling) call; time 20 calls at a fixed `pos`. Expect time to grow roughly linearly in `TMAX` once the Python overhead (about 4 ms per step at `TMAX = 64`, from 15.3) is no longer dominant: that slope is the masking waste.

Quiz

1. Why does `uopc` compile a new plan when only the sequence length changes?
2. The week-3 loop IR already supports a loop whose end is loaded from memory. What part of the compiler stops working well in that case, and why?
3. What two quantities does a KV cache store, per layer, and why is it correct to reuse them?
4. Compute the KV-cache size for 24 layers, `dim` 2048, 2048 tokens, f16.
5. Why accumulate in f32 when inputs are f16? Give the order of magnitude from 15.4.
6. What is the difference between bf16 and f16, and which one needs loss scaling?
7. In data parallelism, what is communicated each step, and with which collective?
8. Name one thing Triton leaves to the programmer that `uopc`'s kernelizer decides automatically.

Answers

1. `_structure_key` includes every shape and movement-op argument, and a different length changes them, so the graph is structurally new and misses the plan cache.
2. Bounds analysis: a loaded `i32` has full-range bounds, so index expressions might overflow, `bounds()` gives up, and div/mod simplification, the `/` vs `floordiv_i32` choice, and rangeify's dependence analysis all degrade.
3. The key and value vectors of every past token. Causal attention at earlier positions never sees later tokens, so past keys and values never change when a token is appended.
4. 2 · 24 · 2048 · 2048 · 2 bytes = 402,653,184 bytes = 384 MiB per sequence.

5. f16 has a 10-bit mantissa; summing many products loses precision quickly. In 15.4, a 64-term dot product had error ~ `7e-2` with an f16 accumulator vs ~ `8e-6` with f32 on the same inputs.
6. bf16 has an 8-bit exponent (f32's range) and a 7-bit mantissa; f16 has a 5-bit exponent and a 10-bit mantissa. f16 needs loss scaling because small gradients underflow its range; bf16 usually does not.
7. Gradients, combined with an all-reduce (sum or mean), so every replica applies the same update.
8. Fusion: in Triton, the programmer decides what goes in one kernel; `uopc` decides kernel boundaries in `kernelize` (materialization points and `stage_broadcast_compute`).

Further reading

- tinygrad, the source code and its examples (github.com/tinygrad/tinygrad). The LLaMA example shows a KV cache with a bounded symbolic position.
- J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, S. Amarasinghe. *Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines.* PLDI 2013.
- T. Chen et al. *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning.* OSDI 2018.
- L. Zheng et al. *Ansor: Generating High-Performance Tensor Programs for Deep Learning.* OSDI 2020.
- P. Tillet, H. T. Kung, D. Cox. *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations.* MAPL 2019.
- C. Lattner et al. *MLIR: Scaling Compiler Infrastructure for Domain Specific Computation.* CGO 2021.
- J. Ansel et al. *PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation.* ASPLOS 2024.
- The XLA and StableHLO documentation at openxla.org.
- P. Micikevicius et al. *Mixed Precision Training.* ICLR 2018.
- M. Shoeybi et al. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.* arXiv 2019.
- W. Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention.* SOSP 2023.
- R. Pope et al. *Efficiently Scaling Transformer Inference.* MLSys 2023.

Appendix A. UOp and `.uir` Reference

This appendix is the reference card you keep open while writing compiler passes. It documents every op in `uopc.uop.Ops` exactly as the reference implementation treats it: what its sources and `arg` are, how its dtype, shape and address space are inferred, what it means, which week introduces it, and what C it turns into. It also gives the full grammar of the `.uir` text format, the scheduling knobs (`AxisType`, `Opts`), and the environment variables. When this appendix and the code disagree, the code wins: `_infer` in `uopc/uop.py` is the source of truth for types and shapes, `uopc/uir.py` for the grammar, and `uopc/render_c.py` for the generated C.

Where this fits: it collects in one place what Chapters 1–13 introduce one op at a time. For how ops are *tested* see Appendix B; for definitions of the vocabulary see Appendix C.

A.1 The op set at a glance

`uopc/uop.py · Ops`

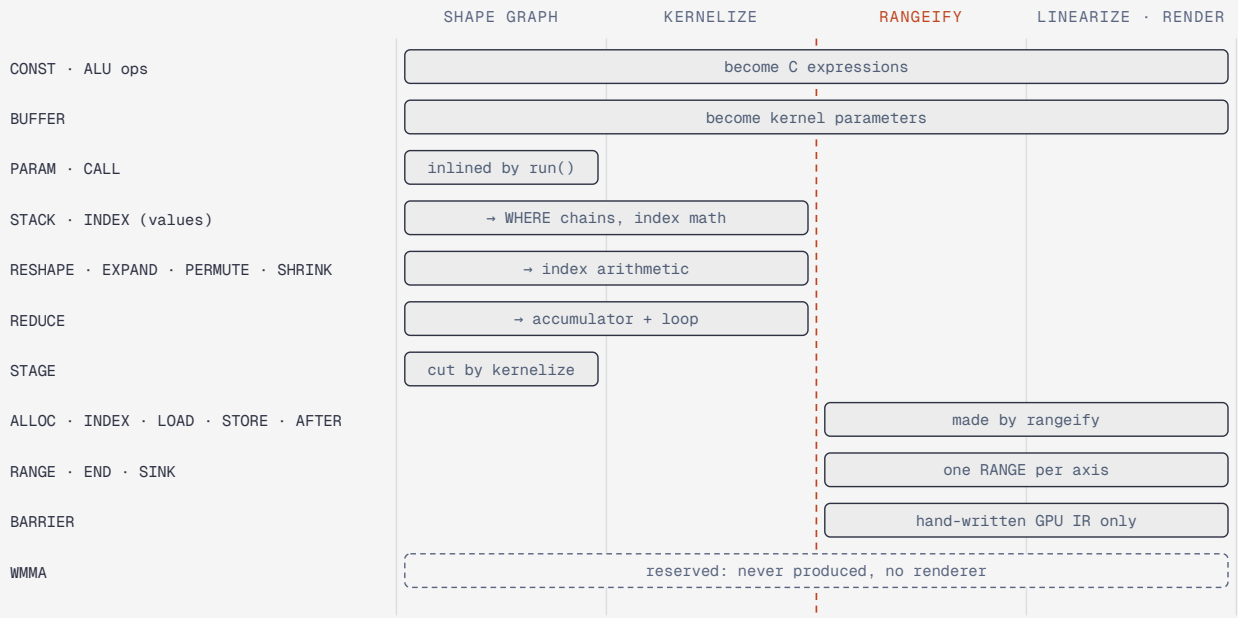
```
class Ops(Enum):
  # ---- week 1: values ----------------------------------------------------
  PARAM = auto()      # arg: ParamArg. An input to the enclosing CALL.
  CALL = auto()       # src[0] = body, src[1:] = arguments. arg: CallArg
  CONST = auto()      # arg: python value (bool/int/float). shape ()
  STACK = auto()      # new leading axis; all srcs share one shape
  INDEX = auto()      # src[0] indexed by src[1:]; pops axes off the left
  CAST = auto()       # arg: DType
  RECIP = auto()      # 1/x
  ADD = auto(); MUL = auto(); CMPLT = auto()
  FLOORDIV = auto(); FLOORMOD = auto()
  WHERE = auto()      # (cond, a, b)
  # extra ALU, introduced when we need them (masks, max-reduce, softmax)
  CMPNE = auto(); MAX = auto()
  EXP2 = auto(); LOG2 = auto(); SQRT = auto(); SIN = auto()
  # ---- week 2: memory ----------------------------------------------------
  BUFFER = auto()     # arg: ParamArg. global persistent memory (never in a CALL body)
  ALLOC = auto()      # arg: ParamArg. call-scope storage `float a[10];`
  LOAD = auto()       # (ptr,)
  STORE = auto()      # (ptr, value)
  AFTER = auto()      # (mem, *deps): `mem`, but only once deps have happened
  SINK = auto()       # (*effects): groups side effects, the root of a program
  # ---- week 3: loops -----------------------------------------------------
  RANGE = auto()      # src[0] = end; arg: RangeArg. value is the loop counter
  END = auto()        # (passthrough, *ranges): closes the ranges
  # ---- week 4: movement + reduction --------------------------------------
  RESHAPE = auto(); EXPAND = auto(); PERMUTE = auto(); SHRINK = auto()
  REDUCE = auto()     # arg: (Ops.ADD | Ops.MAX, n) -- reduces the last n axes
  # ---- week 5: kernels ---------------------------------------------------
  STAGE = auto()      # materialize src[0] into a temp buffer you can index into
  # ---- weeks 7-8: GPUs ---------------------------------------------------
  BARRIER = auto()    # (*deps): __syncthreads()
  WMMA = auto()       # reserved for tensor cores (week 8 lowers GEMMs to WMMA by pattern
instead)

  def __repr__(self): return f"Ops.{self.name}"
  __hash__ = object.__hash__   # members are singletons: identity hashing is correct and much
faster
```

The op groups that rewrite rules and passes test against:

uopc/uop.py:59-63

```
UNARY = {Ops.RECIP, Ops.CAST, Ops.EXP2, Ops.LOG2, Ops.SQRT, Ops.SIN}
BINARY = {Ops.ADD, Ops.MUL, Ops.CMPLT, Ops.CMPNE, Ops.FLOORDIV, Ops.FLOORMOD, Ops.MAX}
ALU = UNARY | BINARY | {Ops.WHERE}
COMMUTATIVE = {Ops.ADD, Ops.MUL, Ops.MAX, Ops.CMPNE}
MOVEMENT = {Ops.RESHAPE, Ops.EXPAND, Ops.PERMUTE, Ops.SHRINK}
```

Ops do not all live equally long. The front end builds *shape graphs*; `kernelize` cuts them into kernels; `rangeify` replaces shapes with loops and memory; the linearizer and renderer see only scalar code. Each op family exists in a contiguous band of that pipeline:



Rangeify is the great divide: shapes to its left, loops and memory to its right.

Op	Week	Sources	`arg`	dtype	shape	addrspace	Rendered as
`CONST`	1	none	`bool`, `int` or `float`	from the Python type	`()`	ALU	literal (`2.0f`, `1`, `INFINITY`)
`PARAM`	1	none	`ParamArg`	`arg.dtype`	`arg.shape`	ALU	not rendered (inlined by `run`)
`CALL`	1	body, args...	`CallArg`	body's	body's	body's	not rendered (inlined by `run`)
`STACK`	1, 4	≥ 1, same shape and dtype	none	src's	`(n,) + src.shape`	ALU	`WHERE` chain after rangeify
`INDEX`	1, 2	x, i32 scalars...	none	x's	x.shape minus leading axes	x's	`x[i]` (memory) or index math
`CAST`	1	1	target `DType`	`arg`	src's	ALU	`((T)(x))`
`RECIP`	1	1 float	none	src's	src's	ALU	`(1.0f/x)`
`EXP2` `LOG2` `SQRT` `SIN`	1 (as needed)	1 float	none	src's	src's	ALU	`exp2f(x)` etc.
`ADD` `MUL`	1	2, same shape and dtype	none	src's	src's	ALU	`(a+b)`, `(a*b)`

Op	Week	Sources	`arg`	dtype	shape	addrspace	Rendered as
`MAX`	1 (as needed)	2, same shape and dtype	none	src's	src's	ALU	`((a>b)?a:b)`
`CMPLT` `CMPNE`	1	2, same shape and dtype	none	`bool`	src's	ALU	`(a<b)`, `(a!=b)`
`FLOORDIV` `FLOORMOD`	1	2 `i32`	none	`i32`	src's	ALU	`/`, `%` or `floordiv_i32` / `floormod_i32`
`WHERE`	1	bool cond, a, b	none	a's	common	ALU	`(c?a:b)`
`BUFFER`	2	none	`ParamArg` (1-D)	`arg.dtype`	`arg.shape`	GLOBAL	kernel parameter `dataN`
`ALLOC`	2	none	`ParamArg` (1-D)	`arg.dtype`	`arg.shape`	LOCAL or `arg.addrspace`	`float accN[n];` / `float accN;` / `__shared__`
`LOAD`	2	memory	none	src's	src's	ALU	`T valN = ptr;`
`STORE`	2	memory, value	none	`void`	`()`	ALU	`ptr = value;`
`AFTER`	2	memory, deps...	none	src[0]'s	src[0]'s	src[0]'s	nothing (an ordering edge)

Op	Week	Sources	`arg`	dtype	shape	addrspace	Rendered as
`SINK`	2	effects...	none	`void`	`()`	ALU	nothing
`RANGE`	3	end (`i32`)	`RangeArg`	`i32`	`()`	ALU	`for (...) {`, `#pragma omp`, `blockIdx` / `threadIdx`
`END`	3	passthrough, ranges...	none	`void`	`()`	ALU	`}`
`RESHAPE`	4	1	new shape	src's	`arg`	src's	index arithmetic
`EXPAND`	4	1	new shape	src's	`arg`	src's	index arithmetic
`PERMUTE`	4	1	permutation	src's	permuted	src's	index arithmetic
`SHRINK`	4	1	`((begin, end), …)`	src's	`end - begin` per axis	src's	index arithmetic
`REDUCE`	4	1	`(Ops.ADD or Ops.MAX, n)`	src's	src.shape minus last `n`	ALU	accumulator + loop
`STAGE`	5	1	none, or `"detach"`	src's	src's	ALU	a kernel boundary (temp buffer); `detach` is stripped
`BARRIER`	7–8	deps...	none	`void`	`()`	ALU	`__syncthreads();` (CUDA only)

Op	Week	Sources	`arg`	dtype	shape	addrspace	Rendered as
`WMMA`	8	a, b, c	(reserved)	src[2]'s	src[2]'s	ALU	not rendered

The rest of this appendix expands every row.

A.2 Types: `DType`, `AddrSpace`, and the args

`uopc/dtype.py · dtypes`

```
class dtypes:
  void = DType("void", 0, "void", None)
  bool = DType("bool", 1, "_Bool", "?")
  i32 = DType("i32", 4, "int", "i")
  f32 = DType("f32", 4, "float", "f")
  f16 = DType("f16", 2, "_Float16", "e")
  all = (void, bool, i32, f32, f16)

  @staticmethod
  def from_name(name: str) -> DType:
    for d in dtypes.all:
      if d.name == name: return d
    raise ValueError(f"unknown dtype {name!r}")

  @staticmethod
  def from_py(x) -> DType:
    if isinstance(x, bool): return dtypes.bool
    if isinstance(x, int): return dtypes.i32
    if isinstance(x, float): return dtypes.f32
    raise TypeError(f"no dtype for python value {x!r}")

  @staticmethod
  def min(d: DType):
    return {"bool": False, "i32": -2**31}.get(d.name, -math.inf)

  @staticmethod
  def max(d: DType):
    return {"bool": True, "i32": 2**31 - 1}.get(d.name, math.inf)
```

dtype	itemsize	C type	numpy	Notes
`void`	0	`void`	none	type of `STORE`, `SINK`, `END`, `BARRIER`
`bool`	1	`_Bool`	`bool_`	result of `CMPLT`/`CMPNE`; `WHERE` condition
`i32`	4	`int`	`int32`	indices, `RANGE` values; folding wraps like two's complement
`f32`	4	`float`	`float32`	folding rounds through `struct.pack('f')` (NOTES 5)
`f16`	2	`_Float16` (`half` in CUDA)	`float16`	storage type for mixed precision and tensor cores

`dtypes.min`/`dtypes.max` give `(-2**31, 2**31 - 1)` for `i32`, `(False, True)` for `bool`, and `(-inf, inf)` for floats; bounds analysis uses them as "anything".

uopc/dtype.py · AddrSpace

```
class AddrSpace(Enum):
  """Where a UOp lives. ALU = a plain value in a register. The rest are memory."""
  ALU = auto()      # a value (week 1)
  GLOBAL = auto()   # device memory, a function argument (week 2)
  LOCAL = auto()    # call-scope storage: `float acc[4];` (week 2)
  SHARED = auto()   # GPU workgroup shared memory (week 7)

  def __repr__(self): return f"AddrSpace.{self.name}"
```

Address space is the third inferred property. It is what makes `LOAD` and `STORE` type-checkable: you can only load from, or store to, something that is not `ALU`. `INDEX`, `AFTER` and the movement ops *preserve* their source's address space, so `BUFFER.index(i)` is a GLOBAL pointer and `BUFFER.reshape(s)` is a GLOBAL view. `REDUCE`, `STAGE` and every ALU op produce plain values (ALU).

The structured args:

uopc/uop.py · ParamArg

```python
@dataclass(frozen=True)
class ParamArg:
  slot: int
  dtype: DType
  shape: tuple[int, ...] = ()
  addrspace: AddrSpace | None = None   # only used by ALLOC (LOCAL or SHARED)
```

uopc/uop.py · CallArg

```python
@dataclass(frozen=True)
class CallArg:
  name: str = "f"
```

uopc/uop.py · RangeArg

```python
@dataclass(frozen=True, order=True)
class RangeArg:
  idx: int                        # loops nest in increasing idx order
  kind: AxisType = AxisType.LOOP
```

`ParamArg.addrspace` is consulted only by `ALLOC` (LOCAL by default, SHARED for GPU shared memory). `RangeArg` is `order=True`: ranges are compared by `idx` first, and *loops nest in increasing* `idx` *order*, an invariant the linearizer relies on (Chapter 5).

A.3 The type checker

Every UOp is type-checked once, at construction, by `_infer`. A malformed node never exists:

uopc/uop.py · _infer

```
def _infer(op: Ops, src: tuple[UOp, ...], arg) -> tuple[DType, tuple[int, ...], AddrSpace]:
  A = AddrSpace.ALU
  if op is Ops.CONST:
    _check(not src, "CONST has no sources")
    return dtypes.from_py(arg), (), A
  if op is Ops.PARAM: return arg.dtype, arg.shape, A
  if op is Ops.BUFFER:
    _check(len(arg.shape) == 1, "BUFFER is flat")
    return arg.dtype, arg.shape, AddrSpace.GLOBAL
  if op is Ops.ALLOC:
    _check(len(arg.shape) == 1, "ALLOC is flat")
    return arg.dtype, arg.shape, arg.addrspace or AddrSpace.LOCAL
  if op is Ops.CALL:
    params = {u.arg.slot: u for u in call_params(src[0])}
    _check(sorted(params) == list(range(len(src) - 1)), f"CALL needs params 0..{len(src)-2}, body
has {sorted(params)}")
    for i, a in enumerate(src[1:]):
      p = params[i]
      _check(a.shape == p.shape and a.dtype == p.dtype, f"CALL arg {i}: {a.dtype}{a.shape} !=
param {p.dtype}{p.shape}")
    return src[0].dtype, src[0].shape, src[0].addrspace
  if op is Ops.STACK:
    _check(len(src) > 0, "STACK of nothing")
    _check(all(s.shape == src[0].shape and s.dtype == src[0].dtype for s in src), "STACK srcs
must agree in shape and dtype")
    return src[0].dtype, (len(src),) + src[0].shape, A
  if op is Ops.INDEX:
    x, idx = src[0], src[1:]
    _check(len(idx) <= len(x.shape), f"INDEX: {len(idx)} indices into shape {x.shape}")
    _check(all(i.dtype == dtypes.i32 and i.shape == () for i in idx), "INDEX indices are i32
scalars")
    return x.dtype, x.shape[len(idx):], x.addrspace
  if op in ALU:
    if op is Ops.WHERE:
      c, a, b = src
      _check(c.dtype == dtypes.bool, "WHERE condition must be bool")
      _check(a.dtype == b.dtype, f"WHERE branches disagree: {a.dtype} vs {b.dtype}")
      _check(c.shape == a.shape == b.shape, f"WHERE shape mismatch {c.shape} {a.shape}
{b.shape}")
      return a.dtype, a.shape, A
    if op in UNARY:
      _check(len(src) == 1, f"{op} is unary")
      if op is Ops.CAST: return arg, src[0].shape, A
      _check(src[0].dtype.is_float, f"{op} needs a float, got {src[0].dtype}")
      return src[0].dtype, src[0].shape, A
    a, b = src
    _check(a.shape == b.shape, f"{op}: shape mismatch {a.shape} vs {b.shape} (broadcast with
EXPAND)")
    _check(a.dtype == b.dtype, f"{op}: dtype mismatch {a.dtype} vs {b.dtype} (use CAST)")
    if op in (Ops.FLOORDIV, Ops.FLOORMOD): _check(a.dtype == dtypes.i32, f"{op} is integer only")
    return (dtypes.bool if op in (Ops.CMPLT, Ops.CMPNE) else a.dtype), a.shape, A
  if op is Ops.LOAD:
```

```
    _check(src[0].addrspace is not A, "LOAD needs memory")
    return src[0].dtype, src[0].shape, A
  if op is Ops.STORE:
    _check(src[0].addrspace is not A, "STORE needs memory")
    _check(src[0].dtype == src[1].dtype, f"STORE of {src[1].dtype} into {src[0].dtype}")
    return dtypes.void, (), A
  if op is Ops.AFTER: return src[0].dtype, src[0].shape, src[0].addrspace
  if op in (Ops.SINK, Ops.END, Ops.BARRIER): return dtypes.void, (), A
  if op is Ops.RANGE:
    _check(src[0].dtype == dtypes.i32, "RANGE end is i32")
    return dtypes.i32, (), A
  if op is Ops.RESHAPE:
    _check(prod(arg) == prod(src[0].shape), f"RESHAPE {src[0].shape} -> {arg}")
    return src[0].dtype, arg, src[0].addrspace
  if op is Ops.EXPAND:
    s = src[0].shape
    _check(len(arg) == len(s) and all(a == b or a == 1 for a, b in zip(s, arg)), f"EXPAND {s} ->
{arg}")
    return src[0].dtype, arg, src[0].addrspace
  if op is Ops.PERMUTE:
    _check(sorted(arg) == list(range(len(src[0].shape))), f"PERMUTE {arg} of {src[0].shape}")
    return src[0].dtype, tuple(src[0].shape[i] for i in arg), src[0].addrspace
  if op is Ops.SHRINK:
    s = src[0].shape
    _check(len(arg) == len(s) and all(0 <= b <= e <= n for (b, e), n in zip(arg, s)), f"SHRINK
{arg} of {s}")
    return src[0].dtype, tuple(e - b for b, e in arg), src[0].addrspace
  if op is Ops.REDUCE:
    rop, n = arg
    _check(rop in (Ops.ADD, Ops.MAX) and 0 <= n <= len(src[0].shape), f"REDUCE {arg} of
{src[0].shape}")
    return src[0].dtype, src[0].shape[:len(src[0].shape) - n], A
  if op is Ops.STAGE: return src[0].dtype, src[0].shape, A
  if op is Ops.WMMA: return src[2].dtype, src[2].shape, A
  raise UOpError(f"no inference rule for {op}")
```

`call_params` finds the PARAMs that belong to a CALL body without descending into a nested CALL's body (NOTES 2):

`uopc/uop.py · call_params`

```python
def call_params(body: UOp) -> list[UOp]:
  """PARAMs that belong to this body. A nested CALL's body is its own scope, so we don't look inside it."""
  out, seen, stack = [], set(), [body]
  while stack:
    u = stack.pop()
    if u in seen: continue
    seen.add(u)
    if u.op is Ops.PARAM: out.append(u)
    stack.extend(u.src[1:] if u.op is Ops.CALL else u.src)
  return out
```

Here is every `UOpError` message, triggered on purpose (scratch script, real output):

```python
# scratch: one malformed construction per check in _infer
from uopc.uop import UOp, Ops, ParamArg, CallArg, UOpError, range_
from uopc.dtype import dtypes
B = lambda n, d=dtypes.f32: UOp(Ops.BUFFER, (), ParamArg(0, d, (n,)))
P = lambda s, d=dtypes.i32, sh=(): UOp(Ops.PARAM, (), ParamArg(s, d, sh))
cases = [
 ("add i32 + f32", lambda: UOp.const(1) + UOp.const(1.0)),
 ("shape mismatch", lambda: B(4).reshape((2,2)) + B(4)),
 ("floordiv float", lambda: UOp(Ops.FLOORDIV, (UOp.const(1.0), UOp.const(2.0)))),
 ("exp2 of int", lambda: UOp(Ops.EXP2, (UOp.const(3),))),
 ("where non-bool", lambda: UOp(Ops.WHERE, (UOp.const(1), UOp.const(1), UOp.const(2)))),
 ("stack mixed", lambda: UOp(Ops.STACK, (UOp.const(1), UOp.const(2.0)))),
 ("index float", lambda: B(4).index(UOp.const(1.0))),
 ("too many idx", lambda: B(4).index(0, 0)),
 ("load of value", lambda: UOp.const(1.0).load()),
 ("store dtype", lambda: B(4).index(0).store(UOp.const(1))),
 ("reshape count", lambda: B(6).reshape((4,))),
 ("expand non-1", lambda: B(6).reshape((2,3)).expand((4,3))),
 ("permute bad", lambda: B(6).reshape((2,3)).permute((0,0))),
 ("shrink oob", lambda: B(6).reshape((2,3)).shrink(((0,3),(0,3)))),
 ("reduce mul", lambda: B(6).reduce(Ops.MUL, 1)),
 ("buffer 2d", lambda: UOp(Ops.BUFFER, (), ParamArg(0, dtypes.f32, (2,3)))),
 ("range f32 end", lambda: UOp(Ops.RANGE, (UOp.const(4.0),), None)),
 ("call slots", lambda: UOp(Ops.CALL, (P(0) + P(1), UOp.const(1)))),
 ("call arg type", lambda: UOp(Ops.CALL, (P(0) * P(0), UOp.const(1.0)))),
]
for name, f in cases:
  try: f(); print(f"{name:16s} OK?!")
  except UOpError as e: print(f"{name:16s} UOpError: {e}")
```

```
add i32 + f32    UOpError: Ops.ADD: dtype mismatch dtypes.i32 vs dtypes.f32 (use CAST)
shape mismatch   UOpError: Ops.ADD: shape mismatch (2, 2) vs (4,) (broadcast with EXPAND)
floordiv float   UOpError: Ops.FLOORDIV is integer only
exp2 of int      UOpError: Ops.EXP2 needs a float, got dtypes.i32
where non-bool   UOpError: WHERE condition must be bool
stack mixed      UOpError: STACK srcs must agree in shape and dtype
index float      UOpError: INDEX indices are i32 scalars
too many idx     UOpError: INDEX: 2 indices into shape (4,)
load of value    UOpError: LOAD needs memory
store dtype      UOpError: STORE of dtypes.i32 into dtypes.f32
reshape count    UOpError: RESHAPE (6,) -> (4,)
expand non-1     UOpError: EXPAND (2, 3) -> (4, 3)
permute bad      UOpError: PERMUTE (0, 0) of (2, 3)
shrink oob       UOpError: SHRINK ((0, 3), (0, 3)) of (2, 3)
reduce mul       UOpError: REDUCE (Ops.MUL, 1) of (6,)
buffer 2d        UOpError: BUFFER is flat
range f32 end    UOpError: RANGE end is i32
call slots       UOpError: CALL needs params 0..0, body has [0, 1]
call arg type    UOpError: CALL arg 0: dtypes.f32() != param dtypes.i32()
```

What `_infer` deliberately does **not** check, so that passes stay simple: the source *count* of `LOAD`, `STORE`, `AFTER`, `END` and `BARRIER`; that a `STORE`'s pointer is scalar; that `BUFFER` never appears in a CALL body (a course rule, not enforced); and anything about `WMMA`. Those properties are guaranteed by the code that builds such nodes (`rangeify`, hand-written loop IR) and by tests.

A.4 Every op

How the examples were produced. Each example below is a `.uir` program parsed with `uir.parse` and printed back with `uir.to_uir` (so you see the canonical form, including desugaring), followed by its inferred type, and then either its value (via `symbolic.run`) or its C. Shaped graphs are lowered with `rangeify.lower` into a fresh output buffer; loop programs are rendered directly with `render_c`. The harness (scratch):

```python
# scratch: the harness behind every example in A.4
from uopc.uop import UOp, Ops, ParamArg
from uopc.uir import parse, to_uir
from uopc.rangeify import lower
from uopc.render_c import render_c
from uopc.symbolic import run, to_python

def show(name, src, mode):
  g = parse(src)
  print(to_uir(g), end="")
  print(f"# -> dtype={g.dtype.name} shape={g.shape} addrspace={g.addrspace.name}")
  assert parse(to_uir(g)) is g, "roundtrip failed"
  if mode == "run": print("# run ->", to_python(run(g)))
  elif mode == "lower":
    n = 1
    for s in g.shape: n *= s
    nb = len({u for u in g.toposort() if u.op is Ops.BUFFER})
    out = UOp(Ops.BUFFER, (), ParamArg(nb, g.dtype, (max(n, 1),)))
    c = render_c(lower(g, out)); print(c[c.index("void kernel"):], end="")
  elif mode == "c":
    c = render_c(g); print(c[c.index("void kernel"):], end="")
```

The `BARRIER` example is rendered with `gpu.render_cuda` instead, since only CUDA has barriers. Every example also passed the assertion `parse(to_uir(g)) is g`: printing and re-parsing returns the very same hash-consed node. The C examples omit the shared prelude (`#include`s and the floor-division helpers, shown in A.6).

Values (week 1)

`CONST`

- **Sources:** none. **Arg:** a Python `bool`, `int` or `float`.
- **Type:** dtype from the Python type (`bool`, `i32`, `f32`), shape `()`, ALU. An `f16` constant is built as `CAST(CONST f32, f16)` because Python has no f16 values. Shaped constants are `CONST` → `RESHAPE` → `EXPAND` (`UOp.const_like`, NOTES 13).
- **Hash consing:** the cache key uses `_argkey`, which adds the type and uses `repr` for floats, so `1`, `1.0`, `True`, `0.0` and `-0.0` are five different nodes (NOTES 1).
- **C:** `cconst` (A.6): `1` / `0` for bools, decimal ints (with `(-2147483647-1)` for `INT_MIN`), `repr` plus `f` for f32, `NAN`, `INFINITY`.

```
%0 = const 2.5
# -> dtype=f32 shape=() addrspace=ALU
# run -> 2.5
```

`PARAM` and `CALL`

- `PARAM`: sources: none; arg: `ParamArg(slot, dtype, shape)`; type straight from the arg; ALU. An input to the *enclosing* CALL body.
- `CALL`: sources: `src[0]` is the body, `src[1:]` the arguments; arg: `CallArg(name)`. The body's own PARAMs (found by `call_params`, which skips nested CALL bodies) must be exactly slots `0..len(args)-1`, and argument `i` must match `PARAM i` in shape and dtype. The CALL has the body's type.
- **Semantics:** beta reduction. `symbolic.inline_call` substitutes arguments for PARAMs, scoped so it never rewrites inside a nested CALL's body (NOTES 2). Rewriting with `symbolic + evaluate` inlines bottom-up.
- **C:** none. `rangeify` raises `NotImplementedError` on PARAM; CALL/PARAM exist for the week-1 "run by rewriting" interpreter.

```
%0 = param : slot=0 dtype=i32
%1 = mul %0, %0
%2 = call %1, 7 : name=sq
# -> dtype=i32 shape=() addrspace=ALU
# run -> 49
```

A shaped PARAM takes a shaped argument:

```
%0 = param : slot=0 dtype=f32 shape=(3,)
%1 = add %0, %0
%2 = stack 1.0, 2.0, 3.0
%3 = call %1, %2 : name=f
# -> dtype=f32 shape=(3,) addrspace=ALU
# run -> [2.0, 4.0, 6.0]
```

`STACK`

- **Sources:** one or more, all the same shape and dtype. **Arg:** none.
- **Type:** shape `(len(src),) + src[0].shape`, dtype of the sources, ALU.

- **Semantics:** a new leading axis. In week 1 its sources are constants or other STACKs (a literal array); from week 4 on they may be any shaped value.
- **Lowering:** `rangeify` turns `stack(a, b, c)[i, rest]` into `where(i < 1, a[rest], where(i < 2, b[rest], c[rest]))`. `symbolic.evaluate` pushes ALU ops through STACKs and resolves `INDEX` of a STACK by a constant.

```
%0 = stack 1, 2, 3
%1 = stack %0, %0
# -> dtype=i32 shape=(2, 3) addrspace=ALU
# run -> [[1, 2, 3], [1, 2, 3]]
```

Shaped, lowered to C (note the `WHERE` chain that STACK became):

```
%0 = buffer 4 : dtype=f32 slot=0
%1 = mul %0, %0
%2 = stack %0, %1
# -> dtype=f32 shape=(2, 4) addrspace=ALU
```

```c
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 2; ridx0++) {
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      float val1 = data0[ridx1];
      data1[((ridx0*4)+ridx1)] = ((ridx0<1)?val1:(val1*val1));
    }
  }
}
```

`INDEX`

- **Sources:** `src[0]` is the indexed value `x`; `src[1:]` are indices, each an `i32` scalar. At most `len(x.shape)` of them.
- **Type:** dtype of `x`; shape `x.shape[len(indices):]` (indices pop axes off the *left*); address space of `x`. So `BUFFER.index(i)` is a GLOBAL scalar: a pointer you can `LOAD` and `STORE`.
- **Two roles.** On a value (week 1), `INDEX` selects a sub-array; `rangeify` prepends the index values to the index tuple it pushes down. On memory (week 2 on), `INDEX(buffer, flat_index)` is an address; the renderer prints `name[index]` (a scalar `ALLOC` of size 1 renders as the bare variable name).

```
%0 = stack 10, 20, 30
%1 = index %0, 2
# -> dtype=i32 shape=() addrspace=ALU
# run -> 30
```

On a shaped buffer view, lowered:

```
%0 = buffer 6 : dtype=f32 slot=0
%1 = reshape %0, (2, 3)
%2 = index %1, 1
# -> dtype=f32 shape=(3,) addrspace=GLOBAL
```

```
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    float val1 = data0[(ridx0+3)];
    data1[ridx0] = val1;
  }
}
```

ALU ops (week 1, extended as needed)

All ALU ops are elementwise on shaped values and scalar after rangeify. Their reference semantics, which the renderer must match, are `exec_alu`:

`uopc/symbolic.py · exec_alu`

```python
def exec_alu(op: Ops, dtype, vals: list):
  # inputs first: an f32 CONST may hold a double (16777217.0) that C rounds when it parses the literal
  if op is not Ops.CAST and dtype.is_float: vals = [truncate(dtype, v) if isinstance(v, float) else v for v in vals]
  if op is Ops.WHERE and dtype.is_float: vals = [vals[0]] + [truncate(dtype, v) for v in vals[1:]]
  match op:
    case Ops.ADD: r = vals[0] + vals[1]
    case Ops.MUL: r = vals[0] * vals[1] if not (dtype.is_float and any(math.isinf(v) for v in vals) and 0 in vals) else math.nan
    case Ops.MAX: r = max(vals[0], vals[1])
    case Ops.CMPLT: return vals[0] < vals[1]
    case Ops.CMPNE: return vals[0] != vals[1]
    case Ops.FLOORDIV:
      if vals[1] == 0 or (dtype == dtypes.i32 and vals[0] == -2**31 and vals[1] == -1): return None   # C traps here
      r = vals[0] // vals[1]
    case Ops.FLOORMOD: r = vals[0] % vals[1] if vals[1] != 0 else None
    case Ops.WHERE: r = vals[1] if vals[0] else vals[2]
    case Ops.RECIP: r = 1.0 / vals[0] if vals[0] != 0 else math.copysign(math.inf, vals[0])
    case Ops.EXP2: r = _safe(math.pow, 2.0, vals[0]) if vals[0] < 1024 else math.inf
    case Ops.LOG2: r = math.log2(vals[0]) if vals[0] > 0 else (-math.inf if vals[0] == 0 else math.nan)
    case Ops.SQRT: r = math.sqrt(vals[0]) if vals[0] >= 0 else math.nan
    case Ops.SIN: r = math.sin(vals[0]) if math.isfinite(vals[0]) else math.nan
    case Ops.CAST:
      v = vals[0]
      if dtype == dtypes.i32:
        if not math.isfinite(v): return None          # undefined behavior in C: refuse to fold
        r = int(v)                                    # C truncates toward zero
      elif dtype == dtypes.bool: return bool(v)
      else: r = float(v)
    case _: return None
  return None if r is None else truncate(dtype, r)
```

and their C forms are `_unop` and `_binop`:

uopc/render_c.py · _unop

```python
def _unop(op: Ops, a: str, d: DType, arg) -> str:
  if op is Ops.CAST: return f"(({arg.ctype})({a}))"
  if op is Ops.RECIP: return f"(1.0f/{a})" if d == dtypes.f32 else f"(({d.ctype})1.0f/{a})"
  return f"{UNARY_FN[op]}{'f' if d == dtypes.f32 else ''}({a})"
```

uopc/render_c.py · _binop

```python
def _binop(op: Ops, a: str, b: str, ua: UOp, ub: UOp, d: DType) -> str:
  match op:
    case Ops.ADD: return f"({a}+{b})"
    case Ops.MUL: return f"({a}*{b})"
    case Ops.CMPLT: return f"({a}<{b})"
    case Ops.CMPNE: return f"({a}!={b})"
    case Ops.MAX: return f"(({a}>{b})?{a}:{b})"
    case Ops.FLOORDIV:
      # C division truncates; it equals floor division when both sides are non-negative
      if vmin(ua) >= 0 and ub.op is Ops.CONST and ub.arg > 0: return f"({a}/{b})"
      return f"floordiv_i32({a},{b})"
    case Ops.FLOORMOD:
      if vmin(ua) >= 0 and ub.op is Ops.CONST and ub.arg > 0: return f"({a}%{b})"
      return f"floormod_i32({a},{b})"
  raise NotImplementedError(op)
```

The `/` vs `floordiv_i32` decision is made per node from bounds analysis: C's truncating division agrees with floor division only when the numerator is provably non-negative and the divisor is a positive constant (NOTES 4).

`CAST`

- **Sources:** 1, any dtype. **Arg:** the target `DType`. **Type:** `arg`, source shape, ALU.
- **Semantics:** C conversion. Float to `i32` truncates toward zero; casting `inf` / `nan` to `i32` is undefined behavior in C, so the folder refuses to fold it (NOTES 6). `UOp.cast` returns the node itself when the dtype already matches.

```
%0 = buffer 4 : dtype=f32 slot=0
%1 = cast %0 : dtype=i32
# -> dtype=i32 shape=(4,) addrspace=ALU
```

```c
void kernel(float* restrict data0, int* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    data1[ridx0] = ((int)(val1));
  }
}
```

`RECIP`

- **Sources:** 1, float. **Type:** source's. **Semantics:** `1/x`; folds `1/0` to `±inf` by the sign of zero. Division `a / b` in the Tensor API is `a * recip(b)`.

```
%0 = buffer 4 : dtype=f32 slot=0
%1 = recip %0
# -> dtype=f32 shape=(4,) addrspace=ALU
```

```
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    data1[ridx0] = (1.0f/val1);
  }
}
```

`EXP2`, `LOG2`, `SQRT`, `SIN`

- **Sources:** 1, float. **Type:** source's. **Semantics:** `2^x`, `log2 x` (`-inf` at 0, `nan` below), `sqrt x` (`nan` below 0), `sin x`. The Tensor API builds `exp`, `log`, `cos`, `tanh`, `sigmoid` and `gelu` from these (`exp(x) = exp2(x · log2 e)`).
- **C:** `exp2f`, `log2f`, `sqrtf`, `sinf` for f32 (the un-suffixed double versions for other float types).

```
%0 = buffer 4 : dtype=f32 slot=0
%1 = exp2 %0
# -> dtype=f32 shape=(4,) addrspace=ALU
```

```
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    data1[ridx0] = exp2f(val1);
  }
}
```

The other three differ only in the function name:

```
    data1[ridx0] = log2f(val1);
    data1[ridx0] = sqrtf(val1);
    data1[ridx0] = sinf(val1);
```

`ADD` and `MUL`

- **Sources:** 2, identical shape and dtype (no implicit broadcasting or promotion in the IR; the error message tells you to use `EXPAND` or `CAST`). Commutative. **Type:** the sources'.

- **Semantics:** `i32` wraps (two's complement), `f32` rounds to f32; `inf * 0` folds to `nan`. Float *inputs* are rounded to f32 before folding too, as C rounds a literal: `16777217.0 + 2.0` folds to `16777218.0`. Constants created by reassociation rules wrap like `i32` (`_i32`), so `(x * 65536) * 65536` simplifies to `0` (`test_folding_matches_c_semantics`).
- **Sugar:** in `.uir`, binary ops broadcast and bare literals take the other operand's dtype. Below, `mul %a, 2` became a reshape/expand of the constant `2.0`:

```
%0 = buffer 4 : dtype=f32 slot=0
%1 = buffer 4 : dtype=f32 slot=1
%2 = add %0, %1
# -> dtype=f32 shape=(4,) addrspace=ALU
```

```
void kernel(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    float val2 = data1[ridx0];
    data2[ridx0] = (val1+val2);
  }
}
```

```
%0 = buffer 4 : dtype=f32 slot=0
%1 = reshape 2.0, (1,)
%2 = expand %1, (4,)
%3 = mul %0, %2
# -> dtype=f32 shape=(4,) addrspace=ALU
```

```
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    data1[ridx0] = (val1*2.0f);
  }
}
```

`MAX`

- **Sources:** 2, same shape and dtype. Commutative. **Semantics:** the larger value. Used for `relu`-style ops, `abs`, `minimum` (as `-max(-a, -b)`) and `REDUCE(MAX)`.

```
%0 = buffer 4 : dtype=f32 slot=0
%1 = reshape 0.0, (1,)
%2 = expand %1, (4,)
%3 = max %0, %2
# -> dtype=f32 shape=(4,) addrspace=ALU
```

```
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    data1[ridx0] = ((val1>0.0f)?val1:0.0f);
  }
}
```

`CMPLT` and `CMPNE`

- **Sources:** 2, same shape and dtype. **Type:** `bool`. `CMPNE` is commutative. Equality is `x.ne(y).ne(True)`; `>` swaps operands of `CMPLT`; `>=` is `not <`.

```
%0 = buffer 4 : dtype=f32 slot=0
%1 = buffer 4 : dtype=f32 slot=1
%2 = cmplt %0, %1
# -> dtype=bool shape=(4,) addrspace=ALU
```

```
void kernel(float* restrict data0, float* restrict data1, _Bool* restrict data2) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    float val2 = data1[ridx0];
    data2[ridx0] = (val1<val2);
  }
}
```

```
%0 = buffer 4 : dtype=i32 slot=0
%1 = reshape 0, (1,)
%2 = expand %1, (4,)
%3 = cmpne %0, %2
# -> dtype=bool shape=(4,) addrspace=ALU
```

```
void kernel(int* restrict data0, _Bool* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    int val1 = data0[ridx0];
    data1[ridx0] = (val1!=0);
  }
}
```

`FLOORDIV` and `FLOORMOD`

- **Sources:** 2 `i32`. **Semantics:** *floor* division and modulo (Python's `//` and `%`): `-7 // 2 = -4`, `-7 % 2 = 1`. Division by zero is not folded, and neither is `INT_MIN // -1`, which traps in C. Floor semantics make `(a*c + b) // c = a + b // c` unconditional, which is what index simplification needs.
- **C:** `/` and `%` when `vmin(numerator) >= 0` and the divisor is a positive constant; otherwise the prelude helpers. Values loaded from a buffer can be negative, so here the helpers appear:

```
%0 = buffer 4 : dtype=i32 slot=0
%1 = reshape 3, (1,)
%2 = expand %1, (4,)
%3 = floordiv %0, %2
# -> dtype=i32 shape=(4,) addrspace=ALU
```

```
void kernel(int* restrict data0, int* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    int val1 = data0[ridx0];
    data1[ridx0] = floordiv_i32(val1,3);
  }
}
```

```
%0 = buffer 4 : dtype=i32 slot=0
%1 = reshape 3, (1,)
%2 = expand %1, (4,)
%3 = floormod %0, %2
# -> dtype=i32 shape=(4,) addrspace=ALU
```

```
void kernel(int* restrict data0, int* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    int val1 = data0[ridx0];
    data1[ridx0] = floormod_i32(val1,3);
  }
}
```

Index arithmetic over `RANGE`s, whose bounds are known, renders as plain `/` and `%` (see `PERMUTE` below).

`WHERE`

- **Sources:** `(cond, a, b)`: `cond` is `bool`, `a` and `b` share a dtype, all three share a shape.
- **Type:** `a`'s dtype and shape. **Semantics:** select. It is how STACK, PAD (bounds-checked loads) and masks are expressed after rangeify.
- **`.uir` sugar:** all three operands broadcast; a bare literal takes the dtype of the *branches*, not of the bool condition.

```
%0 = buffer 4 : dtype=f32 slot=0
%1 = buffer 4 : dtype=f32 slot=1
%2 = cmplt %0, %1
%3 = where %2, %0, %1
# -> dtype=f32 shape=(4,) addrspace=ALU
```

```
void kernel(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    float val2 = data1[ridx0];
    data2[ridx0] = ((val1<val2)?val1:val2);
  }
}
```

Memory (week 2)

`BUFFER`

- **Sources:** none. **Arg:** `ParamArg(slot, dtype, (n,))`; the shape must be 1-D ("BUFFER is flat"); shaped views are `RESHAPE`s of it. **Type:** GLOBAL.
- **Semantics:** persistent device memory, a function argument. In the Tensor runtime a BUFFER UOp owns its numpy array (`kernelize.new_buffer`); kernels see local slots renumbered `0..n-1`, with negative slots used as placeholders during kernelization.
- **C:** a parameter `T* restrict dataN`, parameters sorted by slot.

`ALLOC`

- **Sources:** none. **Arg:** `ParamArg(slot, dtype, (n,), addrspace)`, 1-D. **Type:** address space `arg.addrspace` or LOCAL.

- **Semantics:** call-scope scratch memory. `rangeify` creates one size-1 ALLOC per reduce accumulator. The linearizer declares it in the deepest scope shared by all its users, so each OpenMP thread gets a private copy (NOTES 9); SHARED allocs are pinned to the top of the kernel (NOTES 24).
- **C:** `float accN[n];`, or `float accN;` for a LOCAL `(1,)` alloc, or `__shared__ float smemN[n];` with the CUDA renderer.

`LOAD`, `STORE`, `AFTER`, `SINK`

- `LOAD`: sources: `(ptr,)`, where `ptr` is memory (not ALU). Type: the pointer's dtype and shape, ALU. Always rendered as a named variable `T valN = ptr;`, so a value is read exactly once where it is scheduled.
- `STORE`: sources: `(ptr, value)`, same dtype. Type: `void`. Renders `ptr = value;`.
- `AFTER`: sources: `(mem, *deps)`. Type: exactly `mem`'s (dtype, shape and address space). It means "`mem`, but only once `deps` have happened". SSA graphs have no implicit order between a write and a later read of the same memory; `AFTER` is that order, made explicit as an edge. It renders as nothing (the name of `mem`). `UOp.after()` with no deps returns `mem` itself.
- `SINK`: sources: any effects (`STORE`s, `END`s). Type: `void`. The root of a program; renders nothing.

All six in one loopless program (write `a[0] + a[1]` into a scratch slot, read it back after the write, square it):

```
%0 = buffer 1 : dtype=f32 slot=1
%1 = index %0, 0
%2 = alloc 1 : dtype=f32 slot=0
%3 = index %2, 0
%4 = buffer 2 : dtype=f32 slot=0
%5 = index %4, 0
%6 = load %5
%7 = index %4, 1
%8 = load %7
%9 = add %6, %8
%10 = store %3, %9
%11 = after %2, %10
%12 = index %11, 0
%13 = load %12
%14 = mul %13, %13
%15 = store %1, %14
%16 = sink %15
# -> dtype=void shape=() addrspace=ALU
```

```
void kernel(float* restrict data0, float* restrict data1) {
  float acc0;
  float val1 = data0[0];
  float val2 = data0[1];
  acc0 = (val1+val2);
  float val3 = acc0;
  data1[0] = (val3*val3);
}
```

Loops (week 3)

`RANGE` and `END`

- `RANGE`: sources: `(end,)`, an `i32` UOp (usually a `CONST`, but any `i32` value works; see Chapter 15). Arg: `RangeArg(idx, kind)`. Type: `i32` scalar, ALU. Its *value* is the loop counter, so it can be used in arithmetic. Bounds: `[0, vmax(end) - 1]`.
- `END`: sources: `(passthrough, *ranges)`. Type: `void`. Closes the listed ranges; everything that depends on the END runs after those loops.
- **Scope rule** (the linearizer's contract): a node sits inside every RANGE it transitively depends on, except those closed by an END on the path.
- **C:** `for (int ridxN = 0; ridxN < end; ridxN++) {` … `}`. With `kind=thread` a `#pragma omp parallel for` precedes the loop; GPU kinds are covered in A.7.

```
%0 = buffer 8 : dtype=f32 slot=1
%1 = range 8 : idx=0 kind=loop
%2 = index %0, %1
%3 = buffer 8 : dtype=f32 slot=0
%4 = index %3, %1
%5 = load %4
%6 = mul %5, 2.0
%7 = store %2, %6
%8 = end %7, %1
%9 = sink %8
# -> dtype=void shape=() addrspace=ALU
```

```
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 8; ridx0++) {
    float val1 = data0[ridx0];
    data1[ridx0] = (val1*2.0f);
  }
}
```

The accumulator pattern (`ALLOC` + `AFTER(range)` + `END`) that every reduction lowers to is shown under `REDUCE` below; Chapter 5 builds GEMM and conv1d with it by hand.

Movement and reduction (week 4)

Movement ops never compute; they change how indices map to data. `rangeify` pushes an index tuple through them (the index maps are in the `rangeify` docstring, Chapter 6). They preserve their source's dtype and address space, so a view of a BUFFER is still GLOBAL.

`RESHAPE`

- **Arg:** the new shape. **Check:** same number of elements. **Index map:** ravel the output index in the new shape, unravel it in the old one, then simplify (NOTES 12).

```
%0 = buffer 6 : dtype=f32 slot=0
%1 = reshape %0, (2, 3)
# -> dtype=f32 shape=(2, 3) addrspace=GLOBAL
```

```
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 2; ridx0++) {
    for (int ridx1 = 0; ridx1 < 3; ridx1++) {
      int alu1 = ((ridx0*3)+ridx1);
      float val2 = data0[alu1];
      data1[alu1] = val2;
    }
  }
}
```

`EXPAND`

- **Arg:** the new shape, same rank. **Check:** every axis is unchanged or grows from 1. **Index map:** an expanded axis reads index 0. This is broadcasting; its gradient is a sum (Chapter 8).

```
%0 = buffer 3 : dtype=f32 slot=0
%1 = reshape %0, (1, 3)
%2 = expand %1, (2, 3)
# -> dtype=f32 shape=(2, 3) addrspace=GLOBAL
```

```
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 2; ridx0++) {
    for (int ridx1 = 0; ridx1 < 3; ridx1++) {
      float val1 = data0[ridx1];
      data1[((ridx0*3)+ridx1)] = val1;
    }
  }
}
```

`PERMUTE`

- **Arg:** a permutation of `range(ndim)`; output axis `k` is input axis `arg[k]`. **Index map:** scatter the index back through the inverse permutation.

```
%0 = buffer 6 : dtype=f32 slot=0
%1 = reshape %0, (2, 3)
%2 = permute %1, (1, 0)
# -> dtype=f32 shape=(3, 2) addrspace=GLOBAL
```

```
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    for (int ridx1 = 0; ridx1 < 2; ridx1++) {
      float val1 = data0[((ridx1*3)+ridx0)];
      data1[((ridx0*2)+ridx1)] = val1;
    }
  }
}
```

`SHRINK`

- **Arg:** one `(begin, end)` pair per axis with `0 <= begin <= end <= n`. **Index map:** add `begin`. PAD is *not* a primitive: `composite.pad` builds it from STACK with zeros and SHRINK, and FLIP from an anti-diagonal matrix built with the expand/reshape/shrink trick (Chapter 6).

```
%0 = buffer 12 : dtype=f32 slot=0
%1 = reshape %0, (3, 4)
%2 = shrink %1, ((1, 3), (0, 2))
# -> dtype=f32 shape=(2, 2) addrspace=GLOBAL
```

```
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 2; ridx0++) {
    for (int ridx1 = 0; ridx1 < 2; ridx1++) {
      float val1 = data0[(((ridx0*4)+ridx1)+4)];
      data1[((ridx0*2)+ridx1)] = val1;
    }
  }
}
```

`REDUCE`

- **Arg:** `(Ops.ADD or Ops.MAX, n)` with `0 <= n <= ndim`. **Type:** the source's dtype with the last `n` axes popped; ALU (a reduction produces a value, not a view). To reduce other axes, `PERMUTE` them to the end first (that is what `Tensor.sum(axis)` does).
- **Lowering:** an accumulator `ALLOC` placed inside the loops the output index uses, initialized to the identity (`0` or `-inf`), updated in new `REDUCE`-kind ranges, and read back after their `END`.

```
%0 = buffer 12 : dtype=f32 slot=0
%1 = reshape %0, (3, 4)
%2 = reduce %1 : op=add pop=1
# -> dtype=f32 shape=(3,) addrspace=ALU
```

```
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      float val1 = acc0;
      float val2 = data0[((ridx0*4)+ridx1)];
      acc0 = (val1+val2);
    }
    float val3 = acc0;
    data1[ridx0] = val3;
  }
}
```

Popping both axes with `MAX`:

```
%0 = buffer 12 : dtype=f32 slot=0
%1 = reshape %0, (3, 4)
%2 = reduce %1 : op=max pop=2
# -> dtype=f32 shape=() addrspace=ALU
```

```
void kernel(float* restrict data0, float* restrict data1) {
  float acc0;
  acc0 = -INFINITY;
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      float val1 = acc0;
      float val2 = data0[((ridx0*4)+ridx1)];
      acc0 = ((val1>val2)?val1:val2);
    }
  }
  float val3 = acc0;
  data1[0] = val3;
}
```

Kernels (week 5)

`STAGE`

- **Sources:** 1. **Arg:** `None`, or the string `"detach"`. **Type:** the source's dtype and shape, ALU.
- **Semantics:** "materialize this into a temporary buffer you can index into". It is a request to the kernelizer: every STAGE is a materialization point, so it ends one kernel and the consumers read a temp buffer. `Tensor.contiguous()` creates one; `stage_broadcast_compute` and the layout pass insert them automatically. Without a kernelizer, `rangeify` simply computes the source inline.
- **`arg="detach"`:** `Tensor.detach()` wraps its value in `STAGE(x, "detach")`. Autodiff's rule for it returns no gradient, so it is a stop-gradient; `kernelize` removes it before scheduling with the one-rule matcher `strip_detach`, so it never costs a kernel. Checked in a scratch run: for `x = [1, 2, 3]`, `(x * x.detach()).sum()` has gradient `[1. 2. 3.]` (only the undetached factor contributes), and `x.detach().uop` prints as `Ops.STAGE 'detach'`.

Lowered on its own, a STAGE is transparent:

```
%0 = buffer 4 : dtype=f32 slot=0
%1 = exp2 %0
%2 = stage %1
# -> dtype=f32 shape=(4,) addrspace=ALU
```

```
void kernel(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    data1[ridx0] = exp2f(val1);
  }
}
```

Through `kernelize.make_plan`, it is a cut: two kernels and one temp buffer (scratch):

```
# scratch: STAGE as a kernel boundary
from uopc.uir import parse
from uopc.kernelize import make_plan
g = parse("""%a = buffer 4 : dtype=f32 slot=0
%e = exp2 %a
%s = stage %e
%r = add %s, 1.0""")
plan, inputs = make_plan([g])
for k in plan.kernels:
  print(k.name, k.refs)
  print(k.src[k.src.index("void"):], end="")
print("temps:", plan.temps, " outs:", plan.outs)
```

```
e_4 [('in', 0), ('tmp', 0)]
void e_4(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    data1[ridx0] = exp2f(val1);
  }
}
e_4 [('tmp', 0), ('out', 0)]
void e_4(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    float val1 = data0[ridx0];
    data1[ridx0] = (val1+1.0f);
  }
}
temps: [(dtypes.f32, 4)]  outs: [(dtypes.f32, 4)]
```

GPUs (weeks 7–8)

`BARRIER`

- **Sources:** the effects that must complete first (usually `STORE`s into shared memory). **Type:** `void`.
- **Semantics:** a workgroup-wide barrier. Consumers depend on it through `AFTER(smem, barrier)`.

- **C:** `__syncthreads();` with the CUDA renderer. The plain C renderer has no barrier (`CRenderer.barrier` is `None`), because running GPU threads one after another on a CPU cannot emulate one (NOTES 25); `gpu.emulate` raises `NotImplementedError` for any kernel containing a `BARRIER` rather than produce a wrong program.

A rotate-by-one through shared memory, written by hand (it only renders as CUDA):

```
%0 = buffer 16 : dtype=f32 slot=1
%1 = range 16 : idx=0 kind=local
%2 = index %0, %1
%3 = alloc 16 : dtype=f32 slot=0 space=shared
%4 = index %3, %1
%5 = buffer 16 : dtype=f32 slot=0
%6 = index %5, %1
%7 = load %6
%8 = store %4, %7
%9 = barrier %8
%10 = after %3, %9
%11 = add %1, 1
%12 = floormod %11, 16
%13 = index %10, %12
%14 = load %13
%15 = store %2, %14
%16 = end %15, %1
%17 = sink %16
# -> dtype=void shape=() addrspace=ALU
```

```
extern "C" __global__ void __launch_bounds__(16) kernel(float* __restrict__ data0, float*
__restrict__ data1) {
  __shared__ float smem0[16];
  int lidx0 = threadIdx.x; /* 16 */
  float val1 = data0[lidx0];
  smem0[lidx0] = val1;
  __syncthreads();
  float val2 = smem0[((lidx0+1)%16)];
  data1[lidx0] = val2;
}
```

Note the `%` (not `floormod_i32`): `lidx0 + 1` has bounds `[1, 16]`, provably non-negative. `gpu.smem_gemm_ir` is the full-size example, with two barriers per tile step (Chapter 13).

`WMMA`

- **Sources:** `(a, b, c)`. **Type:** `c`'s dtype and shape; nothing is checked.

- **Status:** reserved. No pass produces it and neither renderer implements it (`render_wmma` raises `NotImplementedError`). Week 8 reaches tensor cores by *pattern*: `gpu.match_gemm` recognizes a GEMM-shaped `REDUCE` and `gpu.wmma_gemm` emits a complete `nvcuda::wmma` kernel as text. Lowering the graph to WMMA nodes is a final project (Chapter 15).

```
%0 = buffer 256 : dtype=f16 slot=0
%1 = buffer 256 : dtype=f16 slot=1
%2 = buffer 256 : dtype=f32 slot=2
%3 = wmma %0, %1, %2
# -> dtype=f32 shape=(256,) addrspace=ALU
```

A.5 The `.uir` text format

`.uir` is the course's interchange format: one SSA statement per line. The parser is `uir.parse`, the printer `uir.to_uir`, and `parse(to_uir(g)) is g` for every graph (the printer is canonical, and hash consing makes equal graphs identical).

Grammar

In EBNF, transcribed from `_parse_line`, `_literal` and `_attr_value`:

```
program   = { line } ;                          (* the value of the LAST statement is the program
*)
line      = [ stmt ] [ "#" { any } ] NEWLINE ;  (* "#" starts a comment anywhere on a line *)
stmt      = "%" name "=" opname [ operands ] [ ":" { attr } ] ;
name      = { any char except "=" } ;           (* "%0", "%acc", "%x1" ... *)
opname    = an Ops member name, any case ;      (* "add", "ADD", "reshape", ... *)
operands  = operand { "," operand } ;           (* commas inside parentheses do not split *)
operand   = "%" name | literal ;
literal   = int | float | "true" | "false" | "inf" | "-inf" | "nan" | tuple ;
                                                (* ints/floats/tuples use Python literal syntax
*)
tuple     = "(" [ literal { "," literal } [ "," ] ] ")" ;
attr      = key "=" value ;                     (* separated by whitespace *)
key       = "dtype" | "slot" | "shape" | "op" | "pop" | "idx" | "kind" | "space" | "name" ;
value     = dtype-name | op-name | axis-kind | addrspace | identifier | literal ;
```

Positional args. For `buffer`, `alloc` and `const` the *first* operand is the arg (the size, or the constant). For `reshape`, `expand`, `permute` and `shrink` the *last* operand is the arg tuple. Every other operand is a source.

Attributes

Key	Used by	Value	Default
`dtype`	`buffer`, `alloc`, `param`, `const`, `cast`	`void`, `bool`, `i32`, `f32`, `f16`	`f32` (`cast` requires it)
`slot`	`buffer`, `alloc`, `param`	int	`0` (`param` requires it)
`shape`	`param`	tuple	`()`
`space`	`alloc`	`alu`, `global`, `local`, `shared`	none (LOCAL)
`op`	`reduce`	`add`, `max`	`add`
`pop`	`reduce`	int	`1`
`idx`	`range`	int	`0`
`kind`	`range`	`loop`, `reduce`, `upcast`, `thread`, `global`, `local`	`loop`
`name`	`call`	identifier	`f`

Sugar

1. **Literals become constants.** A bare literal in a source position becomes a `CONST`. Its dtype is that of the first `%name` operand (for `where`: the first `%name` *branch*), so `mul %f32val, 2` means `2.0`. Exceptions: operands of `index` and `range`, bools, and statements with no `%name` operand use the literal's own Python type.
2. **Binary ops and `where` broadcast.** Missing leading axes are added with `reshape`, size-1 axes are `expand`ed. The printer writes the desugared form, which is why `mul %0, 2` prints as a `reshape` of `2.0`, an `expand`, and the `mul`:

```
%0 = buffer 4 : dtype=f32 slot=0
%1 = reshape 0.0, (1,)
%2 = expand %1, (4,)
%3 = cmplt %0, %2
%4 = reshape 1.0, (1,)
%5 = expand %4, (4,)
%6 = where %3, %0, %5
```

(that is the printout of `cmplt %0, 0.0` followed by `where %1, %0, 1.0`).

Printer rules

`to_uir` walks the graph in topological order and numbers statements `%0, %1, ...`. Constants are printed inline at their uses, never as statements, unless the whole program is a constant. `buffer` / `alloc` print size, `dtype`, `slot` and (for allocs with one) `space`; `range` always prints `idx` and `kind`; `reduce` prints `op` and `pop`. Sources appear in their order in `src`, which is why buffers may print in an order that differs from their slots.

Parse errors

Every error is reported with its line number and text, including type errors raised by `_infer`:

```
ParseError: line 2: '%1 = add %0, %2': undefined %2
ParseError: line 2: '%1 = frob %0': 'FROB'
ParseError: line 3: '%2 = add %0, %1': can't broadcast (2,) with (3,)
ParseError: line 1: '%0 = buffer 4 : dtype=f64 slot=0': unknown dtype 'f64'
ParseError: line 2: '%1 = exp2 %0': Ops.EXP2 needs a float, got dtypes.i32
```

The syllabus example

The syllabus's `gemm_256x256.uir` parses and has shape `(256, 256)`, but it computes `A * rowsum(B)`, not a GEMM (NOTES 11, `test_syllabus_gemm_is_not_a_gemm`). The correct program reshapes `A` to `(N, 1, K)` and the *transpose* of `B` to `(1, N, K)`:

`tests/test_w04_rangeify.py · test_gemm_from_movement_ops`

```python
def test_gemm_from_movement_ops():
  N = 16
  a, b = np.random.randn(N, N).astype(np.float32), np.random.randn(N, N).astype(np.float32)
  g = parse(f"""
  %0 = buffer {N*N} : dtype=f32 slot=0
  %1 = buffer {N*N} : dtype=f32 slot=1
  %2 = reshape %0, ({N}, 1, {N})
  %3 = reshape %1, ({N}, {N})
  %4 = permute %3, (1, 0)
  %5 = reshape %4, (1, {N}, {N})
  %6 = mul %2, %5
  %7 = reduce %6 : op=add pop=1""")
  np.testing.assert_allclose(run_shaped(g, [a.ravel(), b.ravel()]), a @ b, atol=1e-4)
```

A.6 The C prelude and constants

Every C kernel starts with this prelude; the CUDA renderer uses a `__device__` version of the same helpers:

uopc/render_c.py · PRELUDE

```python
PRELUDE = """#include <math.h>
#include <stdbool.h>
static inline int floordiv_i32(int a, int b) { int q = a / b; return (q * b != a && ((a < 0) != (b < 0))) ? q - 1 : q; }
static inline int floormod_i32(int a, int b) { int r = a % b; return (r != 0 && ((r < 0) != (b < 0))) ? r + b : r; }
"""
```

uopc/render_c.py · cconst

```python
def cconst(v, d: DType) -> str:
  if d == dtypes.bool: return "1" if v else "0"
  if d == dtypes.i32: return str(v) if v != -2**31 else "(-2147483647-1)"
  if math.isnan(v): return "NAN"
  if math.isinf(v): return "INFINITY" if v > 0 else "-INFINITY"
  s = repr(float(v))
  return s + "f" if d == dtypes.f32 else f"(({d.ctype}){s}f)"
```

Naming in generated code: `dataN` for BUFFER slot `N`, `accN` / `smemN` for ALLOC slot `N`, `ridxN` for RANGE `idx` `N` (`gidxK` / `lidxK` on GPUs), `valN` for loads, `aluN` for ALU results used more than once.

An ALU result used exactly once is inlined into its user; that is the renderer's only "optimization", and it is enough for gcc to see clean expressions.

A.7 Scheduling: `AxisType` and `Opts`

`uopc/uop.py · AxisType`

```
class AxisType(Enum):
  LOOP = auto()      # a plain serial for-loop
  REDUCE = auto()    # a for-loop that a reduction accumulates over
  UPCAST = auto()    # fully unrolled (week 6). uopc unrolls *during* rangeify (Opts.upcast), so
no RANGE of
                     # this kind is ever built; the name stays for the course's vocabulary
  THREAD = auto()    # parallel across CPU threads (week 6)
  GLOBAL = auto()    # GPU blockIdx (week 7)
  LOCAL = auto()     # GPU threadIdx (week 7)
  def __repr__(self): return f"AxisType.{self.name}"
  __hash__ = object.__hash__
```

How each kind renders:

`AxisType`	Made by	C renderer	CUDA renderer
`LOOP`	`lower` for output axes	`for` loop	`for` loop
`REDUCE`	`_reduce` for reduced axes	`for` loop	`for` loop
`UPCAST`	nothing (see below)	`for` loop	`for` loop
`THREAD`	`lower` for `Opts.thread`	`#pragma omp parallel for` + `for`	`for` loop
`GLOBAL`	`lower` with `Opts.gpu`	`for` loop (the emulator)	`int gidxK = blockIdx.x/y/z;`
`LOCAL`	`lower` with `Opts.local`	`for` loop (the emulator)	`int lidxK = threadIdx.x/y/z;`

`UPCAST` exists as a kind but no pass creates UPCAST ranges: upcasting and unrolling are done *during* rangeify by instantiating the body once per constant offset (`_axis_parts` and `_combos`), so the unrolled

copies never pass through a range at all. The `.x` dimension goes to the *last* (fastest-varying) GPU axis, for coalescing (NOTES 23).

The knobs that choose those kinds are a small dataclass, filled in by `opt.hand_opt_choice`, `opt.search_opt`, or `gpu.gpu_opt_choice`, and applied by `rangeify.lower`:

uopc/rangeify.py · Opts

```
@dataclass
class Opts:
  """Week 6: how to map a kernel onto the machine. Chosen by uopc.opt, applied during rangeify."""
  upcast: dict[int, int] = field(default_factory=dict)   # output axis -> unroll factor (register tile)
  thread: int | None = None                              # output axis run in parallel (OpenMP)
  unroll: int = 1                                        # unroll factor of the innermost reduce axis
  tile: dict[int, int] = field(default_factory=dict)     # output axis -> cache block size (week 6, loop tiling)
  gpu: bool = False                                      # weeks 7-8: output axes become GLOBAL (blockIdx) ranges
  local: dict[int, int] = field(default_factory=dict)    # output axis -> threads per block on that axis (LOCAL)
```

Field	Meaning	Effect on the code
`upcast`	output axis → factor	a register tile: `factor` accumulators sharing one reduce loop (`merge_ends`)
`thread`	one output axis	that axis becomes the outermost loop, kind THREAD (OpenMP)
`unroll`	factor for the innermost reduce axis	the reduce body is replicated and combined as a balanced tree
`tile`	output axis → cache block size	that axis is split into a block loop (outside) and a loop within the block, so a block of outputs reuses operand data while it is in cache; the size must divide the axis and be a multiple of its upcast
`gpu`	bool	output axes become GLOBAL ranges
`local`	output axis → threads per block	splits that axis into GLOBAL × LOCAL ranges

A row sum over an `(8, 8)` matrix with `Opts(thread=0, upcast={0: 2}, unroll=4)`, and the same reduction for a GPU with `Opts(gpu=True, local={0: 4})` (scratch):

```
# scratch: the same REDUCE with CPU opts and with GPU opts
from uopc.uir import parse
from uopc.uop import UOp, Ops, ParamArg
from uopc.dtype import dtypes
from uopc.rangeify import lower, Ctx, Opts
from uopc.render_c import render_c
from uopc.gpu import render_cuda
g = parse("""%a = buffer 64 : dtype=f32 slot=0
%m = reshape %a, (8, 8)
%s = reduce %m : op=add pop=1""")
out = UOp(Ops.BUFFER, (), ParamArg(1, dtypes.f32, (8,)))
for o in [Opts(thread=0, upcast={0: 2}, unroll=4), Opts(gpu=True, local={0: 4})]:
  print("#", o)
  ast = lower(g, out, Ctx(opts=o))
  src = render_cuda(ast) if o.gpu else render_c(ast)
  print(src[src.index("void") if not o.gpu else src.index("extern"):])
```

```
# Opts(upcast={0: 2}, thread=0, unroll=4, tile={}, gpu=False, local={})
void kernel(float* restrict data0, float* restrict data1) {
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 4; ridx0++) {
    int alu1 = (ridx0*2);
    float acc0;
    acc0 = 0.0f;
    int alu2 = (ridx0*16);
    float acc1;
    acc1 = 0.0f;
    for (int ridx1 = 0; ridx1 < 2; ridx1++) {
      float val3 = acc0;
      int alu4 = (ridx1*4);
      int alu5 = (alu2+alu4);
      float val6 = data0[alu5];
      int alu7 = (alu4+alu2);
      float val8 = data0[(alu7+1)];
      float val9 = data0[(alu7+2)];
      float val10 = data0[(alu7+3)];
      acc0 = (val3+((val6+val8)+(val9+val10)));
      float val11 = acc1;
      float val12 = data0[(alu5+8)];
      float val13 = data0[(alu7+9)];
      float val14 = data0[(alu7+10)];
      float val15 = data0[(alu7+11)];
      acc1 = (val11+((val12+val13)+(val14+val15)));
    }
    float val16 = acc0;
    data1[alu1] = val16;
    float val17 = acc1;
    data1[(alu1+1)] = val17;
  }
}

# Opts(upcast={}, thread=None, unroll=1, tile={}, gpu=True, local={0: 4})
extern "C" __global__ void __launch_bounds__(4) kernel(float* __restrict__ data0, float*
__restrict__ data1) {
  int gidx0 = blockIdx.x; /* 2 */
  int lidx0 = threadIdx.x; /* 4 */
  float acc0;
  acc0 = 0.0f;
  for (int ridx2 = 0; ridx2 < 8; ridx2++) {
    float val1 = acc0;
    float val2 = data0[(((gidx0*32)+(lidx0*8))+ridx2)];
    acc0 = (val1+val2);
  }
  float val3 = acc0;
  data1[((gidx0*4)+lidx0)] = val3;
}
```

Read the CPU kernel against the table: the THREAD axis has 4 iterations (8 rows ÷ upcast 2), two scalar accumulators (`acc0`, `acc1`) share one reduce loop of 2 iterations (8 columns ÷ unroll 4), and each iteration adds a balanced tree of four loads. (`alu5` and `alu7` are the same sum in two operand orders; the symbolic canonicalizer only orders constants, so this common subexpression is left for gcc.)

A.8 Environment variables

Variable	Read in	Default	Effect
`DEBUG`	`kernelize.py`	`0`	`1`: one line per `realize` (kernel count, schedule and run time). `2`: plus per-kernel time and GFLOP/s. `3`: plus a second, warm run of each kernel. `4`: plus the C source of each kernel when its plan is first built.
`OPT`	`tensor.py`	`1`	`0`: plain `lower`, no layout pass. `1`: `HandOpt` (layout pass + `hand_opt` heuristics). `2`: `SearchOpt` (times candidate `Opts` per kernel; slow to compile, fast to run).
`CFLAGS`	`runtime.py`	`-O3 -march=native -fopenmp -ffp-contract=fast -fassociative-math -fno-signed-zeros -fno-trapping-math`	whitespace-separated flags for the C compiler (NOTES 20 explains `-fassociative-math`).
`CC`	`runtime.py`	`gcc`	the C compiler.
`UOPC_CACHE`	`runtime.py`	`~/.cache/uopc`	directory of compiled `.so` files, named by a hash of source, compiler and flags.

All of them are read once, at import time. Changing `CFLAGS` or `CC` changes the cache key, so the two configurations never share binaries.

The `DEBUG` levels on a 3×4 softmax (scratch: `x = Tensor(np.arange(12, dtype=np.float32).reshape(3, 4)); print(x.softmax(-1).numpy().round(3))`):

```
--- DEBUG=1
realize: 3 kernels, schedule 140.1 ms, run 0.3 ms
[[0.032 0.087 0.237 0.644]
 [0.032 0.087 0.237 0.644]
 [0.032 0.087 0.237 0.644]]
--- DEBUG=2
  r_3                              0.076 ms     0.00 GFLOP/s
  r_3                              0.029 ms     0.00 GFLOP/s
  e_3_4                            0.033 ms     0.00 GFLOP/s
realize: 3 kernels, schedule 14.2 ms, run 0.3 ms
[[0.032 0.087 0.237 0.644]
 [0.032 0.087 0.237 0.644]
 [0.032 0.087 0.237 0.644]]
--- DEBUG=3
  r_3                              0.078 ms     0.00 GFLOP/s  (again: 0.022 ms)
  r_3                              0.034 ms     0.00 GFLOP/s  (again: 0.017 ms)
  e_3_4                            0.034 ms     0.00 GFLOP/s  (again: 0.018 ms)
realize: 3 kernels, schedule 17.0 ms, run 0.3 ms
[[0.032 0.087 0.237 0.644]
 [0.032 0.087 0.237 0.644]
 [0.032 0.087 0.237 0.644]]
```

Kernel names are `r_` (contains a reduction) or `e_` (elementwise) followed by the output shape. The schedule time varies between runs because the first run also builds and caches the `.so` files. `DEBUG=4` with `OPT=0` prints the three kernels (the max, the sum of exponentials, and the normalized output):

```
--- DEBUG=4 OPT=0
...
void r_3(float* restrict data0, float* restrict data1) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    float acc0;
    acc0 = -INFINITY;
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      float val1 = acc0;
      float val2 = data0[((ridx0*4)+ridx1)];
      acc0 = ((val1>val2)?val1:val2);
    }
    float val3 = acc0;
    data1[ridx0] = val3;
  }
}
...
void r_3(float* restrict data0, float* restrict data1, float* restrict data2) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    float acc0;
    acc0 = 0.0f;
    float val1 = data1[ridx0];
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      float val2 = acc0;
      float val3 = data0[((ridx0*4)+ridx1)];
      acc0 = (val2+exp2f(((val3+(val1*-1.0f))*1.4426950408889634f)));
    }
    float val4 = acc0;
    data2[ridx0] = val4;
  }
}
...
void e_3_4(float* restrict data0, float* restrict data1, float* restrict data2, float* restrict
data3) {
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    float val1 = data1[ridx0];
    float val2 = data2[ridx0];
    for (int ridx1 = 0; ridx1 < 4; ridx1++) {
      int alu3 = ((ridx0*4)+ridx1);
      float val4 = data0[alu3];
      data3[alu3] = (exp2f(((val4+(val1*-1.0f))*1.4426950408889634f))*(1.0f/val2));
    }
  }
}
...
```

(Each `...` elides the four-line prelude, or the timing lines, which are as for `DEBUG=3`.)

A.9 Module map

Week	Module	Key entry points
1	`uop.py`	`UOp`, `Ops`, `_infer`, `UOp.toposort`, `UOp.substitute`, `UOp.wrap`, `sink`, `range_`
1	`rewrite.py`	`UPat`, `PatternMatcher`, `graph_rewrite`
1	`symbolic.py`	`exec_alu`, `vmin`/`vmax`, `simplify`, `run`, `to_python`, `inline_call`
1	`uir.py`	`parse`, `to_uir`
2	`render_c.py`, `runtime.py`	`render_c`, `CRenderer`, `compile_c`, `Program`, `run_c`
2–3	`linearize.py`	`linearize`, `scopes`, `loop_targets`
4	`rangeify.py`, `composite.py`	`rangeify`, `lower`, `Opts`, `Ctx`; `pad`, `flip_axis`
5	`kernelize.py`, `autodiff.py`, `tensor.py`, `nn.py`	`make_plan`, `realize`; `gradient`; `Tensor`; `Linear`, `AdamW`
6	`opt.py`	`layout_pass`, `hand_opt_choice`, `search_opt`, `HandOpt`, `SearchOpt`
7–8	`gpu.py`, `cuda.py`	`gpu_lower`, `render_cuda`, `emulate`, `launch_dims`, `smem_gemm_ir`, `wmma_gemm`; `CUDA`, `run_cuda`

Appendix B. Testing, Debugging, and Slop Management

A compiler bug rarely announces itself where it lives. A wrong rewrite rule in week 1 does not fail a week-1 test; it produces an index that is off by a few elements, which becomes an out-of-bounds load in a week-5 kernel, which corrupts the heap, which aborts the Python process during a GPT training test. That exact chain happened in this appendix, on purpose. The course syllabus calls itself "an exercise in slop management": code that is a little wrong, or a little unclear, compounds every week. This appendix is the defense: how the reference suite is organized, which oracles it trusts, how to write the tests it is missing (fuzzers, property tests, structural tests), how to *test the tests* by re-introducing old bugs, how to debug when something fails anyway, and, for every bug in `book/NOTES.md`, the test that catches it.

Where this fits: it complements every chapter's Pitfalls section. The op reference is Appendix A; the vocabulary is in Appendix C.

> **Labeling convention.** Listings with an `@include` label are in the repository. Code in plain fenced blocks marked *scratch* was written for this appendix and run from outside the repository (the extra tests and the mutation harness live in a scratch directory); every output shown is real.

B.1 The shape of the suite

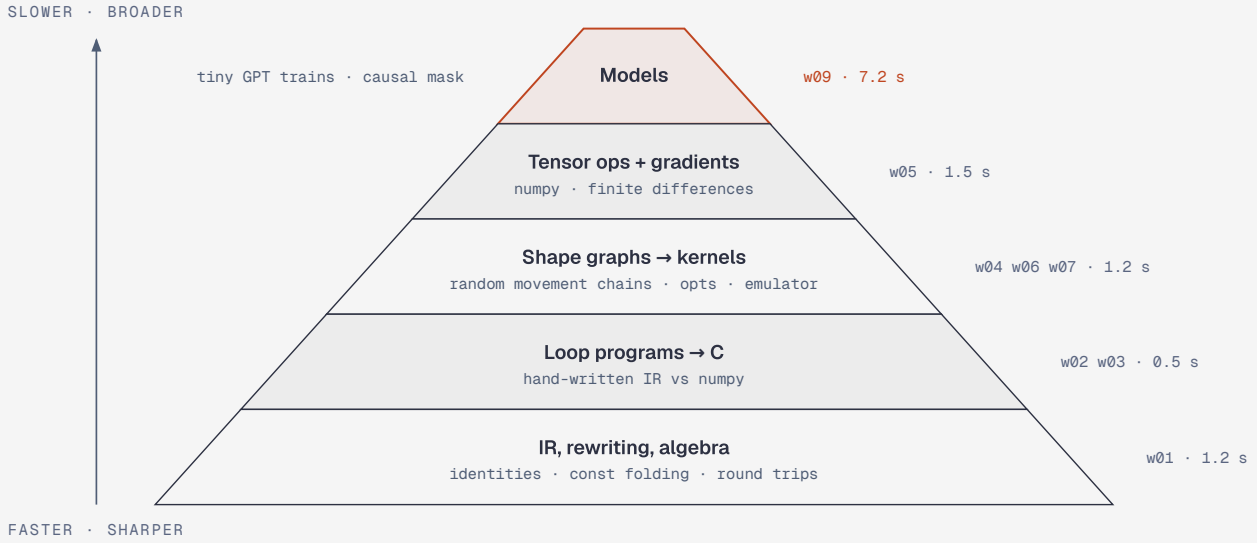
The reference suite has one file per week, named `tests/test_wNN_topic.py`, and a 15-line helper module. With a warm kernel cache it runs in a few seconds:

```
.................................................................... [ 84%]
.........s...                                                        [100%]
============================= slowest 8 durations ==============================
2.63s call     tests/test_w09_models.py::test_tiny_gpt_learns
0.29s call     tests/test_w01_uops.py::test_deep_graph_no_recursion_error
0.26s call     tests/test_w05_tensor.py::test_training_reduces_loss
0.11s call     tests/test_w09_models.py::test_attention_is_causal
0.05s call     tests/test_w06_opt.py::test_gemm_opts_correct[opts5]
0.03s call     tests/test_w05_tensor.py::test_adamw_steps_reuse_one_plan
0.03s call     tests/test_w06_opt.py::test_fused_attention_reduces_are_not_nested
0.03s call     tests/test_w05_tensor.py::test_gradients[matmul-<lambda>-<lambda>]
84 passed, 1 skipped in 3.87s
```

(The skipped test runs the shared-memory GEMM on a real GPU; the reference machine has none, NOTES 25.) From an empty kernel cache the same suite took 33.5 s and left 240 compiled `.so` files behind; the second run took 9.7 s. That ratio is why `runtime.compile_c` caches by content hash, and why the cache directory is configurable (`UOPC_CACHE`, Appendix A).

Per file:

File	Tests	Time	What it checks	Oracle
`test_w01_uops.py`	16	1.1 s	hash consing, inference errors, folding semantics (including C-matching f32 and i32 folding), symbolic identities, CALL scoping (including inlining with a nested CALL still present), no float reassociation, `.uir` round trip	hand-computed values; C semantics
`test_w02_render.py`	5	0.2 s	loopless C, ALLOC/AFTER ordering, floor division with negatives, casts, buffer dtype checks in `run_c`	numpy; hand values
`test_w03_loops.py`	4	0.2 s	hand-written GEMM and conv1d; loop-invariant hoisting; a RANGE without END is an error	numpy (`@`, `np.correlate`); schedule position
`test_w04_rangeify.py`	10	0.2 s	GEMM from views, the syllabus non-GEMM, random movement chains, reductions, STACK/INDEX, PAD/FLIP	numpy
`test_w05_tensor.py`	24	1.2 s	Tensor ops, gradients, kernel counts, training, SGD momentum, kernel arguments after deleted reads, int/float promotion, `item()` and `detach()`, one plan across AdamW steps	numpy; finite differences; closed-form momentum
`test_w06_opt.py`	15	0.5 s	every `Opts` combination on GEMM, cache blocking (`Opts.tile`), shared reduce loops, unrolling, layout pass, fused attention loop depth	numpy; structure of the C
`test_w07_gpu.py`	7 + 1 skipped	0.4 s	GPU lowering through the CPU emulator, launch dims, CUDA text	numpy via emulator; text checks
`test_w09_models.py`	3	8.2 s	LayerNorm, causal masking, a tiny GPT learning	numpy; invariance; loss curve



Each layer catches what the one below cannot see; each failure points at a smaller place.

The pyramid is the classical one, adapted. The base is cheap and *sharp*: when an IR or algebra test fails, it names the function that is wrong. The apex is expensive and *broad*: when the GPT test fails, the bug can be anywhere, but nothing else would have noticed that, say, AdamW's update has the wrong sign. You need both, and the middle layers exist to make apex failures rare. A useful rule of thumb from building `uopc`: every time an apex test caught a bug, a smaller test was written that catches it lower down. That is the last arrow in the triage diagram of B.7.

B.2 Oracles

A test needs an *oracle*: an independent way to know the right answer. A compiler is unusual in having several good ones.

numpy. The workhorse. Every kernel-level test lowers a graph, runs it, and compares with the numpy expression for the same math. The whole harness is one helper:

```
tests/helpers.py · run_shaped
```

```python
def run_shaped(root: UOp, inputs: list[np.ndarray]) -> np.ndarray:
  """Lower a shaped graph whose BUFFERs are slots 0..len(inputs)-1, run it, return the result."""
  n = max(1, prod(root.shape))
  out = buf(len(inputs), n, root.dtype)
  res = np.zeros(n, np_dtype(root.dtype))
  run_c(lower(root, out), {**dict(enumerate(inputs)), len(inputs): res})
  return res.reshape(root.shape)
```

Finite differences. For autodiff, the oracle is calculus itself: `(f(x + ε) − f(x − ε)) / 2ε`, computed in float64 with numpy, entry by entry:

tests/test_w05_tensor.py · numgrad

```python
def numgrad(f, x, eps=1e-3):
  g = np.zeros_like(x, dtype=np.float64)
  for i in np.ndindex(x.shape):
    xp, xm = x.astype(np.float64), x.astype(np.float64)
    xp[i] += eps; xm[i] -= eps
    g[i] = (f(xp) - f(xm)) / (2 * eps)
  return g
```

Note the tolerances in `test_gradients` (`rtol=2e-2, atol=2e-3`): the oracle has truncation error `O(ε²)` and the system under test computes in f32. Too tight and the test is flaky; too loose and a missing factor of 2 slips through. A factor-of-2 bug is far outside these tolerances; a sign error is outside any tolerance.

The interpreter. Week 1's "run by rewriting" (`symbolic.run`) is an independent implementation of the same semantics as the compiled C. For closed programs (no buffers) you can compare three independent paths through the compiler. A scratch harness:

```python
# scratch: evaluate one closed .uir program three independent ways
import sys; sys.path.insert(0, "tests")
import numpy as np
from uopc.uir import parse
from uopc.symbolic import run, to_python
from uopc.tensor import Tensor
from helpers import run_shaped

g = parse("""%0 = stack 1.0, -2.0, 3.0
%1 = stack %0, %0
%2 = permute %1, (1, 0)
%3 = reduce %2 : op=max pop=1
%4 = mul %3, %3""")
paths = {
  "rewrite (symbolic.run)":   lambda: np.array(to_python(run(g))),
  "rangeify + C (lower)":     lambda: run_shaped(g, []),
  "kernelize + opt (Tensor)": lambda: Tensor(g).numpy(),
}
for name, f in paths.items(): print(f"{name:26s}", f().tolist())
```

```
rewrite (symbolic.run)     [1.0, 4.0, 9.0]
rangeify + C (lower)       [1.0, 4.0, 9.0]
kernelize + opt (Tensor)   [1.0, 4.0, 9.0]
```

When the three disagree, the disagreement *localizes* the bug: if `run` and `lower` agree but the Tensor path does not, look in `kernelize` or `opt`. This is differential testing, and it is the core of B.7's bisection.

The emulator. A GPU kernel renders twice: once as CUDA, once with the plain C renderer, where `GLOBAL` and `LOCAL` ranges become ordinary loops. The index arithmetic is identical, so the emulated kernel tests the mapping without a GPU:

`tests/test_w07_gpu.py · test_gpu_gemm_emulated`

```python
@pytest.mark.parametrize("opts", [Opts(gpu=True), Opts(gpu=True, local={0: 4, 1: 8}),
                                  Opts(gpu=True, local={1: 16}, upcast={0: 2, 1: 2})])
def test_gpu_gemm_emulated(opts):
  M, N, K = 32, 64, 48
  a, b = np.random.randn(M, K).astype(np.float32), np.random.randn(K, N).astype(np.float32)
  c = np.zeros(M * N, np.float32)
  ast = gpu_lower(gemm_graph(M, N, K), buf(2, M * N), opts)
  Program(emulate(ast), "kernel")(a, b, c)
  np.testing.assert_allclose(c.reshape(M, N), a @ b, atol=1e-4)
  src = render_cuda(ast)
  assert "__global__" in src and ("threadIdx" in src) == bool(opts.local)
```

Its blind spot is anything that depends on threads running *concurrently*: barriers and shared memory (NOTES 25). That is why `smem_gemm_ir` is only render-tested on the reference machine.

Invariants instead of values. Some properties are easier to state than to compute. `test_attention_is_causal` changes future tokens and asserts that past outputs do not move; `test_tiny_gpt_learns` asserts the loss falls by 4x; `test_uir_roundtrip_and_syllabus_example` asserts `parse(to_uir(g)) is g`. Hash consing makes that last one strong: `is` means the two graphs are structurally identical, node for node.

Floating-point tolerance. The default `CFLAGS` include `-fassociative-math`, so gcc may reorder float additions (NOTES 20). Results are deterministic for a given binary but differ from numpy's summation order at the level of f32 rounding. Never compare floats bitwise across implementations; use `assert_allclose` with tolerances scaled to the reduction length.

B.3 Fuzzing movement chains

Movement ops compose in combinatorially many ways, and each composition is an index map that rangeify must get exactly right. The suite does not enumerate cases; it *generates* them:

`tests/test_w04_rangeify.py · test_random_movement_chains`

```
@pytest.mark.parametrize("seed", range(5))
def test_random_movement_chains(seed):
  rng = np.random.default_rng(seed)
  x = rng.standard_normal(24).astype(np.float32)
  u, ref = buf(0, 24).reshape((2, 3, 4)), x.reshape(2, 3, 4)
  for _ in range(6):
    k = rng.integers(4)
    if k == 0:
      p = tuple(rng.permutation(len(ref.shape))); u, ref = u.permute(p), ref.transpose(p)
    elif k == 1:
      arg = tuple((int(b), int(rng.integers(b + 1, s + 1))) for s in ref.shape for b in
[rng.integers(0, s)])
      u, ref = u.shrink(arg), ref[tuple(slice(b, e) for b, e in arg)]
    elif k == 2:
      u, ref = u.reshape((1,) + u.shape), ref.reshape((1,) + ref.shape)
      u, ref = u.expand((2,) + u.shape[1:]), np.broadcast_to(ref, (2,) + ref.shape[1:])
    else:
      u, ref = u.reshape((int(np.prod(ref.shape)),)), ref.reshape(-1)
  np.testing.assert_allclose(run_shaped(u, [x]), ref)
```

Six random steps from {permute, shrink, expand-a-new-axis, flatten}, applied in lockstep to a UOp and a numpy array, then compared. Five seeds run in milliseconds and exercise hundreds of index-map compositions. Three design points carry over to any fuzzer you write:

1. **Mirror the operation in the oracle step by step.** The generator builds the numpy reference alongside the graph, so there is no separate "compute the expected answer" logic to get wrong.
2. **Seed everything** (`np.random.default_rng(seed)`, parametrized over seeds). A failure is then a test ID you can rerun, not a flake.
3. **Bias toward edge cases.** The shrink step picks `begin` anywhere and `end` above it, so length-1 axes appear often; expand only grows size-1 axes, which is exactly the case broadcasting bugs live in.

B.4 Property tests for the symbolic engine

`symbolic.py` has about forty rewrite rules, and each one is a claim: "this rewrite never changes the value". Example tests check the claims you thought of. A *property test* checks the claim itself, on expressions you did not think of. The property for `simplify` is: for every integer expression `e` over loop variables and every assignment of values to those variables, `e` and `simplify(e)` evaluate to the same number, and that number lies within `[vmin(e), vmax(e)]`. Here is one written without any library (scratch):

```python
# scratch: test_symbolic_props.py
"""Property tests for uopc.symbolic: simplify() must not change the value of any
integer expression, and vmin/vmax must bound every value it can take."""
import random, pytest
from uopc.uop import UOp, Ops, range_
from uopc.symbolic import simplify, vmin, vmax, run

def rand_expr(rng: random.Random, rngs: list[UOp], depth: int) -> UOp:
  if depth == 0 or rng.random() < 0.25:
    return rng.choice(rngs) if rng.random() < 0.7 else UOp.const(rng.randint(-8, 16))
  a, b = rand_expr(rng, rngs, depth - 1), rand_expr(rng, rngs, depth - 1)
  op = rng.choice(["add", "mul", "div", "mod", "lt"])
  if op == "add": return a + b
  if op == "mul": return a * rng.randint(-3, 5)          # keep it affine-ish, like real indices
  if op == "div": return a // rng.randint(1, 9)
  if op == "mod": return a % rng.randint(1, 9)
  return (a < b).where(a, b)

def evaluate(e: UOp, env: dict[UOp, int]) -> int:
  return run(e.substitute({r: UOp.const(v) for r, v in env.items()})).arg

@pytest.mark.parametrize("seed", range(200))
def test_simplify_preserves_value_and_bounds(seed):
  rng = random.Random(seed)
  rngs = [range_(rng.randint(1, 12), k) for k in range(3)]
  e = rand_expr(rng, rngs, 4)
  s = simplify(e)
  for _ in range(10):
    env = {r: rng.randrange(r.src[0].arg) for r in rngs}
    want = evaluate(e, env)
    assert evaluate(s, env) == want, f"seed {seed}: {e} -> {s} at {env}"
    assert vmin(e) <= want <= vmax(e), f"seed {seed}: bounds [{vmin(e)}, {vmax(e)}] miss {want}"
```

```
.......................................................................... [ 36%]
.......................................................................... [ 72%]
.......................................................                  [100%]
200 passed in 1.29s
```

Three details make it work. The generator produces the expressions rangeify actually produces: sums of loop variables times small constants, floor-divided and taken modulo small constants, with an occasional `WHERE`. The evaluator is *not* the system under test's C path; it substitutes constants and lets constant folding (`exec_alu`, the reference semantics) compute the value. And it checks bounds too, because a wrong `vmax` silently turns a correct `floordiv_i32` into a wrong `/` in the renderer (NOTES 4).

Does it find anything? Not on the reference code. So break the code and see. This mutant forgets one check in `_divides` (scratch, applied by monkeypatching, never by editing `uopc/`):

```
# scratch: a plausible bug in symbolic._divides
def m_divides():
  import uopc.symbolic as s
  from uopc.uop import Ops
  old = s._divides
  def divides(u, c):                                # forget to check that the coefficient
divides c
    if u.op is Ops.MUL and u.src[1].op is Ops.CONST: return u.src[0] * (u.src[1].arg // c)
    return old(u, c)
  s._divides = divides
```

With the mutant applied:

- all 13 week-1 tests **pass**, as do weeks 2 and 3;
- the property test fails **21 of 200** seeds, each with the offending expression in the message, for example `seed 175: ... -> UOp(Ops.CONST, arg=0)` where the right answer was `6`;
- the full suite fails **36 tests**, all from week 4 on: movement ops, Tensor ops, gradients, the GPU emulator and the GPT, every one of them far from `_divides`. (Before `kernelize` was fixed to pass only the buffers a kernel still reads, the same mutant made a generated kernel write out of bounds and the process died with `Fatal Python error: Aborted` inside `test_tiny_gpt_learns`.)

The property test is the only check that fails *in the right place*. That is what the pyramid is for.

> **Key idea:** a rewrite rule is a theorem. Example tests check instances; property tests check the theorem on instances you did not choose. For an algebra engine, write the property test first.

B.5 Testing the tests: mutation testing

How do you know a test suite would catch a bug? Put the bug back and run it. *Mutation testing* does exactly that: apply a small, plausible defect (a *mutant*), run the suite, and see whether any test fails (the mutant is *killed*) or not (it *survives*). The bugs in `book/NOTES.md` are perfect mutants: they really happened.

The harness re-introduces each one by monkeypatching before pytest imports the tests, so the repository is never edited (scratch, abridged to four mutants):

```python
# scratch: mutants.py -- re-introduce bugs from book/NOTES.md, then run pytest
import sys, pytest

def m_argkey():
  import uopc.uop as m
  m._argkey = lambda arg: arg                      # NOTES 1: forget the type in the cache key

def m_trunc_div():
  import uopc.render_c as rc
  old = rc._binop
  def binop(op, a, b, ua, ub, d):                  # NOTES 4: C '/' and '%' are floor division,
right?
    from uopc.uop import Ops
    if op is Ops.FLOORDIV: return f"({a}/{b})"
    if op is Ops.FLOORMOD: return f"({a}%{b})"
    return old(op, a, b, ua, ub, d)
  rc._binop = binop

def m_f64_fold():
  import uopc.symbolic as s
  old = s.truncate
  s.truncate = lambda d, x: x if d.name == "f32" else old(d, x)   # NOTES 5: fold floats in f64

def m_const_like():
  from uopc.uop import UOp
  UOp.const_like = lambda self, v: UOp.const(v, self.dtype)       # NOTES 13: forget to broadcast

MUTANTS = {k[2:]: v for k, v in globals().items() if k.startswith("m_")}

if __name__ == "__main__":
  name, *args = sys.argv[1:]
  if name != "none": MUTANTS[name]()
  sys.exit(pytest.main(["-q", "-p", "no:cacheprovider", "--tb=no", "-rN", *args]))
```

Results against the reference suite as it stood with 76 tests (all measured; "+ new test" refers to the tests in B.4 and B.6):

Mutant (NOTES)	Suite result	Killed by
none (baseline)	76 passed, 1 skipped	—
`_argkey` ignores type (1)	**66 failed**	`test_hash_consing` and almost everything downstream
naive `inline_call` (2)	76 passed	**survived**; now killed by `test_inline_call_does_not_enter_nested_bodies`
recursive `graph_rewrite` (3)	1 failed	`test_deep_graph_no_recursion_error`
`/` and `%` for floor ops (4)	1 failed	`test_floordiv_rendering_negative`
f32 folding in f64 (5)	2 failed	`test_const_fold_semantics`, `test_folding_matches_c_semantics`
accumulators at kernel top (9)	76 passed at `-O3`; **3 failed at `-O0`**	`test_gemm_opts_correct[opts1]`, `[opts3]`, `[opts5]`
`const_like` without broadcast (13)	8 failed	`test_reduce_max_and_nested`, 4 gradient tests, training, emulated softmax, tiny GPT
AdamW constants as Python floats (17)	76 passed, **77 s instead of 11 s**	**survived**; now killed by `test_adamw_steps_reuse_one_plan`
`_divides` without divisibility check	36 failed	the new property test (21/200 seeds) is the only week-1 failure

Three survivors teach three lessons.

The CALL scoping mutant survives because `run` rewrites bottom-up: by the time the outer CALL is inlined, the inner one is already gone, so the naive substitution never sees a nested body. NOTES 2 says as much ("bottom-up rewriting also inlines inner calls first"). The bug is real for any *other* caller of `inline_call`, so the fix is a unit test that calls it directly (B.6).

The accumulator mutant survives at `-O3` and dies at `-O0`. Declaring `float acc0;` at the top of the function makes it shared between OpenMP threads: a data race, which is undefined behavior. At `-O3` gcc keeps `acc0` in a register inside each thread's loop, so the race never touches memory and every answer is right. At `-O0` every access goes to the shared variable:

```python
# scratch: alloc_check.py -- the accumulator mutant on a 256x256 GEMM with a THREAD axis, 20 runs
import sys; sys.path.insert(0, "tests"); sys.path.insert(0, ".")
import mutants; mutants.m_alloc_top()          # moves every ALLOC to the top of the kernel
import numpy as np
from uopc.rangeify import lower, Ctx, Opts
from uopc.render_c import render_c
from uopc.runtime import Program
from test_w06_opt import gemm_graph
from helpers import buf
M, N, K = 256, 256, 256
src = render_c(lower(gemm_graph(M, N, K), buf(2, M * N), Ctx(opts=Opts(thread=0))), "k")
a, b = np.random.randn(M, K).astype(np.float32), np.random.randn(K, N).astype(np.float32)
bad = 0
for t in range(20):
  c = np.zeros(M * N, np.float32); Program(src, "k")(a, b, c)
  bad += not np.allclose(c.reshape(M, N), a @ b, atol=1e-3)
print("wrong results in", bad, "of 20 runs")
```

With the default flags it prints `wrong results in 0 of 20 runs`; with `CFLAGS="-O0 -fopenmp"`:

```
wrong results in 20 of 20 runs
```

The kernel it runs declares the accumulator outside the parallel loop:

```
void k(float* restrict data0, float* restrict data1, float* restrict data2) {
  float acc0;
  #pragma omp parallel for
  for (int ridx0 = 0; ridx0 < 256; ridx0++) {
    int alu1 = (ridx0*256);
    for (int ridx1 = 0; ridx1 < 256; ridx1++) {
      acc0 = 0.0f;
      for (int ridx2 = 0; ridx2 < 256; ridx2++) {
        float val2 = acc0;
        float val3 = data0[(alu1+ridx2)];
        float val4 = data1[((ridx2*256)+ridx1)];
        acc0 = (val2+(val3*val4));
      }
      float val5 = acc0;
      data2[(alu1+ridx1)] = val5;
    }
  }
}
```

The lesson: **run the suite under more than one set of compiler flags.** Optimizers hide undefined behavior as often as they expose it. Because `CFLAGS` is part of the kernel-cache key, the two configurations never share

binaries.

The AdamW mutant survives because recompiling every step is not *wrong*, only slow. The suite's wall time went from 11 s to 77 s and nothing failed. Performance bugs need tests that assert on *how* the answer was computed. Those are next.

B.6 Structural tests: assert on the code, not the answer

Some properties of a compiler are about its output program, not its output values: how many loops, which loop a value is computed in, how many kernels, how many compiled plans. The suite already has a few:

`tests/test_w03_loops.py · test_loop_invariant_code_is_hoisted`

```
def test_loop_invariant_code_is_hoisted():
  N = 8
  A, C = buf(0, N * N), buf(1, N * N)
  i, j = range_(N, 0), range_(N, 1)
  row = A.index(i * N).load()            # depends only on i
  s = sink(C.index(i * N + j).store(row + A.index(i * N + j).load()).end(i, j))
  lin = linearize(s)
  pos = {u: n for n, u in enumerate(lin)}
  assert pos[row] < pos[j]                # emitted before the j loop opens
```

`tests/test_w06_opt.py · test_upcast_shares_reduce_loop`

```
def test_upcast_shares_reduce_loop():
  src = render_c(lower(gemm_graph(8, 8, 8), buf(2, 64), Ctx(opts=Opts(upcast={0: 2, 1: 2}))),
"k")
  assert src.count("for (") == 3                        # 2 output loops + ONE shared reduce loop
  assert src.count("acc") >= 4
```

Counting `"for ("` in generated C is crude, and that is fine: it is cheap, it is stable as long as the renderer's output format is stable, and it encodes a design decision ("an upcast GEMM shares one reduce loop") that a refactor could silently undo. Here are the tests the suite *was* missing, each tied to a NOTES entry. They were written for this appendix as scratch files, run against mutants (below), and then adopted into the real suite. They are included here straight from `tests/`:

`tests/test_w01_uops.py · test_inline_call_does_not_enter_nested_bodies`

```python
def test_inline_call_does_not_enter_nested_bodies():            # NOTES 2
  from uopc.symbolic import inline_call
  p0 = param(0)
  inner = UOp(Ops.CALL, (p0 * p0, p0 + 1), CallArg("sq"))
  outer = UOp(Ops.CALL, (inner + p0, UOp.const(3)), CallArg("f"))
  body = inline_call(outer)                  # inline ONLY the outer call; inner must survive
intact
  assert body.src[0].op is Ops.CALL and body.src[0].src[0] is p0 * p0
  assert run(body).arg == 19
```

tests/test_w01_uops.py · test_cast_inf_to_int_is_not_folded

```python
def test_cast_inf_to_int_is_not_folded():                       # NOTES 6
  assert exec_alu(Ops.CAST, dtypes.i32, [math.inf]) is None
  assert exec_alu(Ops.CAST, dtypes.i32, [math.nan]) is None
```

tests/test_w01_uops.py · test_float_constants_do_not_reassociate

```python
def test_float_constants_do_not_reassociate():                  # NOTES 7
  x = UOp(Ops.PARAM, (), ParamArg(0, dtypes.f32))
  assert simplify((x + 1.0) + 2.0) is not simplify(x + 3.0)
```

tests/test_w06_opt.py · max_loop_depth

```python
def max_loop_depth(src: str) -> int:
  return max((len(l) - len(l.lstrip())) // 2 for l in src.splitlines() if
l.lstrip().startswith("for ("))
```

tests/test_w06_opt.py · test_fused_attention_reduces_are_not_nested

```python
def test_fused_attention_reduces_are_not_nested():                # NOTES 12
  import sys
  from pathlib import Path
  sys.path.insert(0, str(Path(__file__).parent.parent / "examples"))
  from flash_attention import attention, FlashOpt
  from uopc.tensor import Tensor
  from uopc.kernelize import make_plan
  N, D = 64, 16
  q, k, v = (Tensor(np.random.randn(N, D).astype(np.float32)) for _ in range(3))
  plan, _ = make_plan([attention(q, k, v).uop], fuse_reduce=True, opt=FlashOpt)
  # loops: i (thread), then a reduce over keys j, inside it a dot product over D: depth 3, never
4
  assert max(max_loop_depth(k.src) for k in plan.kernels) <= 3
```

tests/test_w05_tensor.py · test_adamw_steps_reuse_one_plan

```
def test_adamw_steps_reuse_one_plan():                          # NOTES 17
  from uopc import kernelize
  Tensor.manual_seed(0)
  lin = Linear(4, 2)
  opt = AdamW([lin.weight, lin.bias], lr=1e-2)
  x = Tensor(np.random.randn(8, 4).astype(np.float32))
  def step():
    loss = lin(x).square().mean()
    opt.zero_grad(); loss.backward(opt.params); opt.step(loss)
  step(); step()
  n = len(kernelize._plans)
  for _ in range(3): step()
  assert len(kernelize._plans) == n                             # no new plans: nothing
recompiled
```

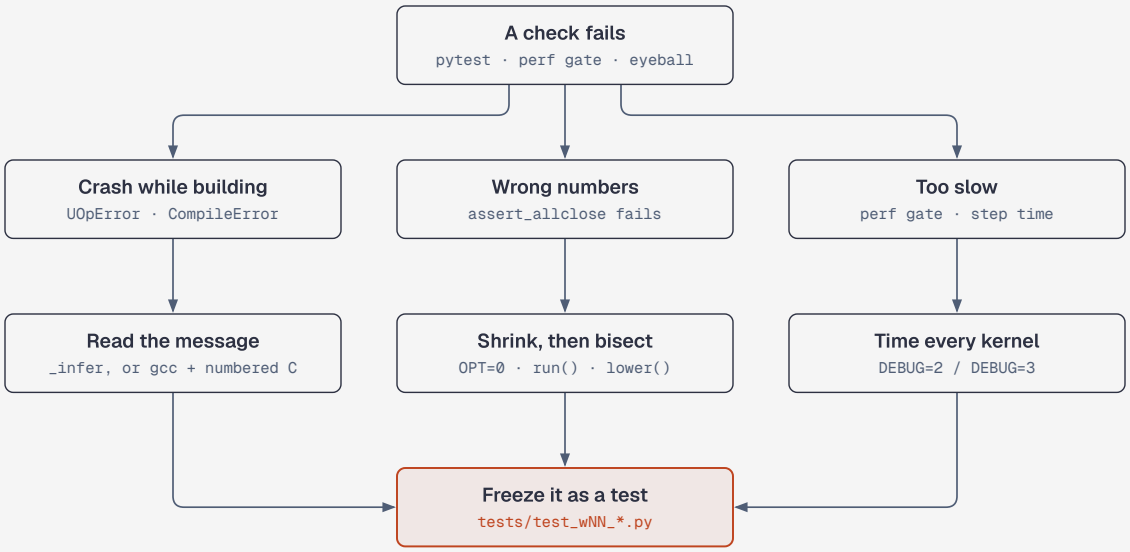
On the reference code all of them pass. Against the mutants, the scratch versions said (paths shortened):

```
== mutant: call_scope   (test_extra.py)
FAILED .../test_extra.py::test_inline_call_does_not_enter_nested_bodies
1 failed, 2 passed in 0.02s
== mutant: adamw_consts (test_structure.py)
.../test_structure.py:33: AssertionError: assert 7 == 4
FAILED .../test_structure.py::test_adamw_steps_reuse_one_plan
1 failed, 1 passed in 1.30s
```

The plan-count test turns a two-minute slowdown into a one-line failure with a number in it: three more steps made three more plans. The loop-depth test pins the structure that NOTES 12's $O(N^3)$ bug destroyed. Its strength is limited: re-creating that bug by simply disabling the `simplify` call at `RESHAPE` did *not* reproduce the old nesting in the current code (other rules now clean up the indices), so this test is a guard on the property, not a demonstrated catch.

> **Key idea:** correctness tests catch wrong answers; structural tests catch right answers computed the wrong way. A compiler course is graded on both, and most performance regressions are invisible to the first kind.

B.7 Debugging: a triage procedure



A bug that is not a test yet is a bug you will meet again.

Crashes while building a graph. `UOpError` is raised in the constructor of the first malformed node, and its message names the op, the dtypes and the shapes (the full list is in Appendix A.3). Read the traceback up to *your* line: that is where the bad node was built, not where it was used. `LinearizeError` means the schedule deadlocked (it prints the open loops and the first stuck nodes with their target scopes) or that a loop was opened and never closed (`RANGE without END: loops [...] are never closed`). A `CompileError` from gcc includes the whole C source with line numbers, so the gcc message and the source line are side by side.

Wrong numbers. Shrink first, then bisect.

1. *Shrink the input.* The smallest shape that still fails (often 2×3) makes the generated C readable. Make sizes distinct (2, 3, 5) so a swapped axis cannot hide behind a square shape.
2. *Turn off optimizations.* `OPT=0` removes the layout pass and all `Opts`. If the answer becomes right, the bug is in `opt.py` or in how `rangeify` applies upcasts and unrolls.
3. *Bisect the pipeline* with the three paths of B.2: `symbolic.run` (no rangeify), `helpers.run_shaped` (rangeify, no kernelize), `Tensor(...).numpy()` (everything). The first path that disagrees with numpy contains the bug.
4. *Read the C* (`DEBUG=4`, or `render_c(...)` directly). Check one output element by hand: pick indices, follow the index expression, and see which input elements it reads.

5. *Change the compiler flags.* If `CFLAGS="-O0 -fopenmp"` changes the answer, suspect a data race or other undefined behavior (B.5).

Too slow. Measure before guessing. `DEBUG=2` prints every kernel's time and GFLOP/s; `DEBUG=3` runs each kernel a second time, which separates first-touch page faults from real work. A kernel with a low GFLOP/s and a deep loop nest is usually a fusion problem (recompute through an `EXPAND`, NOTES 15) or a layout problem (a strided reduce axis, NOTES 19); read its C. If kernels are fast but steps are slow, the time is in Python. Here is a cProfile of the first, compiling, training step of a small GPT (2 layers, `dim` 64, context 32, batch 8), scratch:

```
# scratch: where does the first (compiling) GPT training step spend its time?
import cProfile, pstats, sys
sys.path.insert(0, "examples")
import numpy as np
from gpt import GPT
from uopc.tensor import Tensor
from uopc.nn import AdamW, parameters

Tensor.manual_seed(0)
model = GPT(vocab=65, ctx=32, dim=64, heads=4, layers=2)
opt = AdamW(parameters(model), lr=1e-3)
x = Tensor(np.random.randint(0, 65, (8, 32)).astype(np.int32))
y = Tensor(np.random.randint(0, 65, (8, 32)).astype(np.int32))
def step():
  loss = model(x).reshape(-1, 65).cross_entropy(y.reshape(-1))
  opt.zero_grad(); loss.backward(opt.params); opt.step(loss)
cProfile.run("step()", "step.prof")
pstats.Stats("step.prof").sort_stats("cumulative").print_stats(14)
```

```
         28342210 function calls (26311338 primitive calls) in 38.837 seconds
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.004    0.004   39.079   39.079 uopc/kernelize.py:178(realize)
        1    0.097    0.097   22.546   22.546 uopc/kernelize.py:111(make_plan)
      226    0.003    0.000   16.455    0.073 uopc/runtime.py:47(__init__)
      226    0.008    0.000   16.414    0.073 uopc/runtime.py:27(compile_c)
       97    0.002    0.000   16.307    0.168 subprocess.py:506(run)
```

and the same profile sorted by self time:

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       97   16.166    0.167   16.166    0.167 {method 'poll' of 'select.poll' objects}
2108782/612269    3.785    0.000    4.796    0.000 uopc/rewrite.py:36(match)
   237157    1.490    0.000    2.613    0.000 uopc/uop.py:229(_infer)
     2120    1.419    0.001   12.010    0.006 uopc/rewrite.py:90(graph_rewrite)
    10329    1.339    0.000    2.574    0.000 uopc/uop.py:130(toposort)
   522574    0.999    0.000    6.627    0.000 uopc/uop.py:110(__new__)
```

(Paths shortened; rows for the profiler's own frames omitted.) Read it top down: the step built 226 kernels, 97 of which were new and went through gcc (16.2 s waiting on subprocesses); the other 129 were already in the disk cache. Of the 22.5 s in `make_plan`, most is pattern matching (`UPat.match`, 2.1 million calls) inside `graph_rewrite`. The next optimization is therefore not in gcc but in the matcher: fewer rules tried per node, or memoizing more. That is the same reasoning that took GPT compilation from 163 s to 47 s (NOTES 22), and it is project 28 in Chapter 15.

Freeze it. Whatever the bug was, the last step is a test that fails without the fix and passes with it, placed at the lowest layer of the pyramid that can see it. If the bug was in `symbolic.py` and you found it with GPT, the regression test belongs in week 1.

B.8 Style: "can it be simpler?"

The course grades style at 15%, with one question: *did you write slop?* Slop is code that works today and costs you next week: dead branches, duplicated logic, clever one-liners nobody can check, comments that narrate instead of explain. `uopc` is under 2,800 lines (`wc -l uopc/*.py`) for a compiler that trains a GPT; these are the rules that kept it there.

1. **One idea per function, one screen per function.** `graph_rewrite` is 26 lines. The two longest functions are `_infer` (78 lines, but it is a table: one short case per op) and `linearize` (65 lines, because scheduling is the hardest problem in the compiler).
2. **Rules as data.** Algebra lives in `PatternMatcher` tables, one rule per line, each with a comment for the non-obvious ones ("ints only: float addition isn't associative"). Adding a rule is a one-line diff that can be reviewed in isolation and property-tested (B.4).
3. **Fail at construction, with a message.** Every `_check` in `_infer` names the op and the offending values. A good error message is worth ten debugging sessions.
4. **No recursion over graphs.** Graphs get deep (20,000 nodes in `test_deep_graph_no_recursion_error`); `toposort`, `graph_rewrite`, `substitute` and `call_params` are all iterative (NOTES 3).

5. **Comments say why, not what.** Compare `# C division truncates; it equals floor division when both sides are non-negative` (why) with `# divide a by b` (what). Every comment in `render_c._binop` is the first kind.
6. **No dead code.** An expression like `X if False else Y` is `Y` with extra steps. Two of them crept into the reference tests while it was being written and were removed in review; `grep -n "if False" -r tests uopc examples` now prints nothing. Grep for leftovers like this before every submission.
7. **Measure size.** `wc -l uopc/*.py` per week. If a module doubles in a week, ask why before moving on.
8. **Name things after what they mean in the course.** `Opts.upcast`, `AxisType.THREAD`, `STAGE`: the names in the code are the names in the lectures and in this book, so a reader can go from either to the other.

B.9 A checklist per week

Before submitting each week, answer every question with a test, not with a belief.

Week	Must have a test for
1	hash consing of `1`, `1.0`, `True`, `0.0`, `-0.0`; every `_infer` error you rely on; floor vs truncating division in folding; i32 wraparound; f32 rounding in folding; CALL with nested CALL; a 20,000-node graph; `.uir` round trip; a symbolic property test
2	loopless C for every ALU op against numpy; ALLOC + AFTER ordering; negative operands to floor ops in C; casts, including negative floats to int
3	GEMM and a convolution in the loop IR; sibling loops (`x - max(x)`); loop-invariant hoisting; an accumulator inside a parallel loop
4	each movement op alone; random chains of them (fuzz); REDUCE over 1 and all axes, ADD and MAX; STACK inside rangeify; PAD and FLIP; a real GEMM from views (and the syllabus non-GEMM)
5	each Tensor op vs numpy; a gradient check per autodiff rule; kernel counts for fusion decisions; the plan cache across training steps; a loss that falls
6	every `Opts` combination against the unoptimized result; structure (shared reduce loops, unrolled bodies); the layout pass; the suite under a second set of `CFLAGS`
7–8	every GPU lowering through the emulator; launch dimensions; coalescing (`.x` on the contiguous axis); CUDA text for shared memory and barriers; the real-GPU test where hardware exists
9–10	layer-level checks (LayerNorm, attention causality); a tiny model that learns; plan reuse across steps and across sampling; a step-time budget

B.10 Every pitfall and the test that catches it

The final table maps each entry in `book/NOTES.md` to the test that catches it. "Measured" means the bug was re-introduced as a mutant (B.5) and the named test failed; "existing" means a suite test targets it but it was not mutated here; "proposed" means no test exists yet and the cell says what to write.

NOTES	Bug	Catching test	Status
1	hash-consing merges `1`, `1.0`, `True`	`test_hash_consing` (50 tests fail)	measured
2	CALL inlining rewrites nested bodies	`test_inline_call_does_not_enter_nested_bodies` (B.6); the older call test cannot see it	in suite
3	recursion depth on deep graphs	`test_deep_graph_no_recursion_error`	measured
4	C `/` is not floor division	`test_floordiv_rendering_negative`; `test_folding_matches_c_semantics` also pins `INT_MIN // -1` as not folded	measured
5	f32 constant folding in f64	`test_const_fold_semantics`, `test_folding_matches_c_semantics`	measured
6	folding `(int)inf`	`test_cast_inf_to_int_is_not_folded` (B.6)	in suite
7	float reassociation	`test_float_constants_do_not_reassociate` (B.6)	in suite
8	linearizer deadlocks and misplaced inits	`test_gemm_loops`, `test_conv1d_loops`, `test_reduce_max_and_nested` (sibling loops), `test_loop_invariant_code_is_hoisted`, `test_missing_end_is_an_error`	existing
9	shared accumulators across threads	`test_gemm_opts_correct[opts1]`, `[opts3]`, `[opts5]`, **only under** `CFLAGS="-O0 -fopenmp"`	measured
10	`os.replace` across filesystems	run the suite with `UOPC_CACHE` on a different filesystem from the system temp dir	proposed
11	the syllabus GEMM is not a GEMM	`test_syllabus_gemm_is_not_a_gemm`	existing
12	unsimplified indices nest reductions	`test_fused_attention_reduces_are_not_nested` (B.6)	in suite (guard)
13	`const_like` without broadcasting	`test_reduce_max_and_nested` and 7 more	measured
14	no distribution, `//` chains in inner loops	a structural test that the inner loop of a GPT attention kernel contains no `/` or `%`	proposed

NOTES	Bug	Catching test	Status
15	fusion recomputes through `EXPAND`	a test that `make_plan` of LayerNorm → matmul contains a STAGE and that the plan's total `flops` is within 2x of the unfused count	proposed
16	`substitute` wraps the original node	a test where a staged node contains another staged node, asserting both STAGEs survive `stage_broadcast_compute`	proposed
17	AdamW constants recompile every step	`test_adamw_steps_reuse_one_plan` (B.6)	in suite
18	sampling lengths recompile	the same plan-count pattern across 10 sampling steps	proposed
19	strided GEMM, threads slower	`test_layout_pass_stages_transposed_operand` (layout); GFLOP/s gates as a separate, opt-in benchmark run	existing + proposed
20	no vectorization without `-fassociative-math`	a benchmark gate; or assert the flag is in `runtime.CFLAGS`	proposed
21	layout pass stages the broadcast product	`test_layout_pass_stages_transposed_operand` (stages only the operand)	existing
22	Python overhead in compilation	a compile-time budget for one GPT step, run with a warm `.so` cache	proposed
23	uncoalesced `.x` mapping	assert in `render_cuda` output that `lidx0` (the `.x` index) is multiplied by 1 in the store address	proposed
24	`__shared__` declared inside a loop	`test_smem_gemm_renders` (checks the declaration; extend it to check it is the first line of the body)	existing
25	barriers cannot be emulated	`gpu.emulate` now refuses kernels with a `BARRIER` (`NotImplementedError`); `test_cuda_smem_gemm_on_gpu` runs them, skipped without a GPU	existing

Bugs found while this book was written, after NOTES was closed, got regression tests of their own: `test_run_c_checks_buffer_dtypes` (a numpy array of the wrong dtype or size passed to `run_c`), `test_deleted_reads_do_not_shift_kernel_arguments` (a read removed by simplification shifted the

remaining kernel arguments, so a kernel wrote into its input), `test_python_float_promotes_int_tensors`, and `test_item_before_backward_and_detach`.

Nine NOTES bugs were re-introduced as mutants. The reference suite killed five on its own (1, 3, 4, 5, 13), one only under different compiler flags (9), two survived until a new test was added (2, 17), and one could not be re-created in the current code (12). That is a good suite, and it is not a complete one. Neither is yours. Mutation testing is how you find out which.

Further reading

- The pytest documentation (docs.pytest.org): parametrization, fixtures, `--durations`, `-k` and `-x`.
- K. Claessen, J. Hughes. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs.* ICFP 2000. The origin of property-based testing; Hypothesis is its best-known Python descendant.
- X. Yang, Y. Chen, E. Eide, J. Regehr. *Finding and Understanding Bugs in C Compilers.* PLDI 2011 (Csmith). Differential testing of compilers at scale.
- Y. Jia, M. Harman. *An Analysis and Survey of the Development of Mutation Testing.* IEEE Transactions on Software Engineering, 2011.
- The tinygrad test suite (github.com/tinygrad/tinygrad, `test/`), for how a production ML compiler organizes unit, op-level and model-level tests.

Appendix C. Glossary and Quiz Bank

The first half of this appendix defines the vocabulary of the book, as `uopc` uses it. Where a term has a broader meaning in the literature, the definition gives the general sense first and the `uopc` specifics second, with the module where it lives. The second half collects quiz questions for every week of the course, in the style of the in-class quizzes, with answers in collapsed blocks.

Where this fits: op-level details are in Appendix A; testing practice is in Appendix B; the last appendix, Appendix D, lists the full source.

C.1 Glossary

Accumulator. Scratch memory a reduction adds (or maxes) into. Rangeify creates one size-1 `ALLOC` per reduction, initialized to the identity, updated inside the reduce loop, and read after it. The C renderer turns a size-1 LOCAL accumulator into a plain scalar (`float acc0;`).

`AddrSpace`. The third inferred property of a UOp (after dtype and shape): `ALU` for values, `GLOBAL` for buffers passed to a kernel, `LOCAL` for call-scope scratch memory, `SHARED` for GPU shared memory. `LOAD` and `STORE` require a non-ALU source (`uopc/dtype.py`).

`AFTER`. `AFTER(mem, *deps)` is `mem`, but only once `deps` have executed. It is how an SSA graph, which has no inherent order, expresses "read this memory after that write" or "this accumulator lives inside that loop". It renders as nothing.

All-reduce. A collective operation in which every device contributes an array and every device receives their sum (or another reduction). Data-parallel training all-reduces gradients. A ring all-reduce moves about `2N` elements per device regardless of the device count.

`ALLOC`. Call-scope storage, `float a[n];` in C. Created by rangeify for accumulators or written by hand in loop IR. The linearizer declares it in the deepest scope shared by all its users; SHARED allocs go at the top of the kernel.

ALU op. An op that computes a value from values: `CAST`, `RECIP`, `EXP2`, `LOG2`, `SQRT`, `SIN`, `ADD`, `MUL`, `MAX`, `CMPLT`, `CMPNE`, `FLOORDIV`, `FLOORMOD`, `WHERE` (`uopc.uop.ALU`). Elementwise on shaped values.

Arithmetic intensity. FLOPs performed per byte (or per element) moved from memory. Low intensity means memory-bound. Register tiling raises it by reusing each loaded value many times.

Autodiff (reverse mode). Computing the gradient of a scalar with respect to many inputs by walking the graph from the output back to the inputs, applying a local rule at each node and summing contributions where paths merge. In `uopc`, `autodiff.gradient` builds a new *graph* for the gradient, which the compiler then compiles like any other.

`AxisType`. The kind of a `RANGE`: `LOOP`, `REDUCE`, `UPCAST`, `THREAD`, `GLOBAL`, `LOCAL`. It decides how the renderer emits the loop: a `for`, an OpenMP parallel `for`, or a GPU block/thread index.

Bank conflict. On a GPU, shared memory is split into banks; when threads of one warp access different addresses in the same bank in one instruction, the accesses serialize.

Barrier. A point every thread of a GPU block must reach before any continues (`__syncthreads()`). `Ops.BARRIER` in the loop IR; it cannot be emulated by running threads one after another.

Beta reduction. Replacing a function application by the function's body with arguments substituted for parameters. `inline_call` performs it on `CALL` nodes, scoped so nested CALL bodies are untouched.

bf16 (bfloat16). A 16-bit float with f32's 8-bit exponent and a 7-bit mantissa: f32's range at reduced precision. Not in `uopc`.

Bounds analysis. Computing, for every integer expression, an interval `[vmin, vmax]` guaranteed to contain its value (`symbolic.bounds`). Drives div/mod simplification, the choice of `/` vs `floordiv_i32` in the renderer, and constant-folding of comparisons.

Broadcasting. Making shapes compatible by adding leading size-1 axes (`RESHAPE`) and stretching size-1 axes (`EXPAND`). The IR never broadcasts implicitly; the Tensor API and the `.uir` parser insert the ops.

`BUFFER`. Global, persistent, flat memory: a kernel argument. In the Tensor runtime a BUFFER UOp owns its numpy array.

`CALL` / `PARAM`. A function application and its formal parameters. `CALL(body, *args)`; the body's PARAMs are numbered by slot. Week 1 only: they are eliminated by inlining, never rendered.

Canonical form. A chosen representative among equivalent expressions, so equal things look equal. The symbolic rules move constants to the right of commutative ops; hash consing then makes equal canonical forms the same object.

Coalescing. When the threads of a warp access consecutive addresses, the hardware combines the accesses into few memory transactions. Map the contiguous axis to `threadIdx.x` (NOTES 23).

Common subexpression elimination (CSE). Computing a repeated expression once. Hash consing gives it for free: building the same subexpression twice returns the same node.

Confluence. A rewrite system is confluent if the order in which rules are applied does not change the final normal form. Real rule sets, including `uopc`'s, are designed to be close to confluent but are not proven so.

Constant folding. Evaluating an op whose inputs are all constants at compile time. `exec_alu` is the reference semantics; it refuses to fold undefined behavior (casting `inf` to int).

Data parallelism. Splitting a batch across devices that each hold a full copy of the model, then all-reducing gradients.

`DEBUG`. Environment variable controlling diagnostics in `kernelize.realize`: 1 schedule summary, 2 per-kernel timing, 3 warm re-run timing, 4 generated C.

`DType`. A data type record (`name`, `itemsize`, C type, numpy format): `void`, `bool`, `i32`, `f32`, `f16`.

Emulator. `gpu.emulate`: rendering a GPU kernel with the C renderer, where `GLOBAL` and `LOCAL` ranges are ordinary loops, so its index mapping can be tested on a CPU.

`END`. `END(x, *ranges)` closes loops: nodes that depend on it run after those loops finish.

`EXPAND`. A movement op that stretches size-1 axes; reading an expanded axis reads index 0. Its gradient is a sum over the expanded axes.

f16. IEEE half precision: 5-bit exponent, 10-bit mantissa, maximum 65504. `_Float16` in the generated C, `half` in generated CUDA.

Finite differences. Approximating a derivative by `(f(x+ε) - f(x-ε)) / 2ε`. The oracle for gradient tests.

Fixed point. A state that a transformation no longer changes. `graph_rewrite` rewrites until no rule fires.

Floor division. Division rounding toward negative infinity (Python's `//`), as opposed to C's truncation toward zero. `FLOORDIV`/`FLOORMOD` use floor semantics, which makes `(a*c + b) // c = a + b // c` unconditional.

Fusion. Computing several ops in one kernel without writing intermediates to memory. Its cost is *recompute* when a fused value is consumed through an `EXPAND`.

`fuse_reduce`. A kernelize mode in which `REDUCE`s are not materialization points, so reductions nest inside each other (used for fused attention); it also skips `stage_broadcast_compute`).

Guard. A run-time check that the assumptions a compiled program was specialized on still hold (a shape, a bound on a symbolic size). Failing a guard means recompiling or rejecting the input.

Gradient check. Comparing an autodiff gradient against finite differences.

`graph_rewrite`. The rewrite engine: bottom-up, greedy, to a fixed point, iterative, memoized by node (`uopc/rewrite.py`).

Hash consing. Constructing structurally identical objects only once, so identity is structural equality. `UOp.__new__` looks up `(op, src, _argkey(arg))` in a weak-value cache.

Identity element. The value a reduction starts from: 0 for `ADD`, `-inf` (or `INT_MIN`) for `MAX`.

im2col. Rewriting a convolution as a matrix multiply by gathering each receptive field into a row. `Tensor.conv2d` builds the columns from `SHRINK`s.

`INDEX`. Indexing: on a value it selects a sub-array (popping leading axes); on memory it computes an address that `LOAD` and `STORE` use.

Interval arithmetic. Computing bounds of an expression from bounds of its parts (`[a0,a1] + [b0,b1] = [a0+b0, a1+b1]`). Sound but unable to see relations between variables (it cannot prove `j < n` inside a loop to `n`).

JIT cache (structural cache). `kernelize._plans`, keyed by `_structure_key`: the graph's structure with buffers anonymized. A training step with the same structure as the previous one reuses the compiled plan.

Kernel. One compiled function that reads buffers and writes one output buffer. A *plan* is a list of kernels plus the temp buffers between them.

Kernelize. Cutting a shape graph into kernels at materialization points, and building the plan (`uopc/kernelize.py`).

KV cache. In autoregressive decoding, the stored key and value vectors of all past tokens, per layer, so each new token only computes its own.

Layout pass. `opt.layout_pass`: copying an operand whose reduce axis is strided into a contiguous temp (a `STAGE`) before a reduction reads it, after pushing movement ops down to the data (`push_views`).

Linearizer. The pass that turns a graph into an ordered list of UOps respecting loop scopes (`uopc/linearize.py`), by list scheduling over the loop tree.

`LOAD` / `STORE`. Read a value from memory / write a value to memory, through an `INDEX` address.

Loop-invariant hoisting. Computing an expression outside the loops it does not depend on. The linearizer does it automatically: a node is emitted in the scope of its sources, not of its users.

`lower`. `rangeify.lower(root, out, ctx)`: turn one shaped value into a `SINK` of loops that store it into a buffer.

Materialization point. A node whose value is written to memory, ending a kernel: every root, every `STAGE`, and every `REDUCE` unless `fuse_reduce` is set.

Mixed precision. Storing values in a low-precision type (f16, bf16) and accumulating in a higher one (f32).

Movement op. `RESHAPE`, `EXPAND`, `PERMUTE`, `SHRINK`: ops that change how indices map to data without computing anything.

Mutation testing. Measuring a test suite by injecting small defects (mutants) and checking that some test fails for each.

Normal form. An expression no rule can rewrite further.

OpenMP. A C extension for shared-memory parallelism; `#pragma omp parallel for` splits a loop's iterations across threads. Emitted for `THREAD` ranges.

`OPT`. Environment variable selecting the kernel optimizer in `tensor.py`: 0 none, 1 `HandOpt` (default), 2 `SearchOpt`.

`Opts`. The per-kernel schedule: `upcast` factors, a `thread` axis, `unroll` factor, `tile` (cache block) sizes, and the GPU flag with `local` sizes (`rangeify.Opts`).

`PatternMatcher` / `UPat`. A list of (pattern, function) rewrite rules, indexed by root op; `UPat` matches UOps structurally, binds names, and with a list `src` tries all orders of a commutative op's sources.

`PERMUTE`. A movement op that reorders axes. Its gradient is the inverse permutation.

Placeholder buffer. A `BUFFER` with a negative slot that `kernelize` substitutes for a materialized value while building each kernel, later bound to a temp, input or output.

Property test. A test that checks a general claim on many generated inputs (for example, that `simplify` preserves values) rather than on a few chosen examples.

Rangeify. Lowering a shape graph to scalar loop code by giving every output axis a `RANGE` and pushing the index tuple down through every op (`uopc/rangeify.py`).

`RANGE`. A loop: its value is the loop counter, its source is the trip count, its arg is `RangeArg(idx, kind)`. Loops nest in increasing `idx`.

`ravel` / `unravel`. Converting a multi-dimensional index to a flat offset in row-major order, and back.

Recompute. Computing the same value more than once because it was fused rather than materialized. The trade-off of fusion; `stage_broadcast_compute` bounds it with a cost rule.

`REDUCE`. `REDUCE(x, (op, n))` reduces the last `n` axes with `ADD` or `MAX`.

Register tiling (upcasting). Computing a small tile of outputs per loop iteration, held in registers, so each loaded input is reused across the tile.

`RESHAPE`. A movement op that changes the shape without changing the element count or order.

Roofline model. Attainable performance = min(peak FLOP/s, arithmetic intensity × memory bandwidth).

Scope. The set of loops a node must sit inside: every `RANGE` it depends on, minus those closed by an `END` on the path. The linearizer closes scopes under "encloses" so each is a path in the loop tree.

Shape graph. A graph of shaped UOps (with movement ops and reductions), before rangeify.

Shared memory. Fast, per-block, programmer-managed GPU memory (`__shared__`). `AddrSpace.SHARED`.

`SHRINK`. A movement op that keeps a sub-box `[begin, end)` per axis. Its gradient is a zero-pad.

SIMT. Single instruction, multiple threads: the GPU execution model in which a warp of threads executes one instruction stream on different data.

`SINK`. The root of a program: groups side effects.

SSA. Static single assignment: every value is defined once. A UOp graph is SSA by construction.

`STACK`. Creates a new leading axis from same-shaped sources. Rangeify turns it into a `WHERE` chain.

`STAGE`. "Materialize this into a temp buffer": a user- or pass-inserted materialization point. With `arg="detach"` it is instead a stop-gradient (`Tensor.detach()`), removed by `kernelize` before scheduling.

Symbolic shape. A shape entry known only at run time, represented as an expression (usually a variable with declared bounds) rather than an integer. Not in `uopc`; see Chapter 15.

Tensor core. GPU hardware that performs a small matrix multiply-accumulate (for example 16×16×16 with f16 inputs and f32 accumulation) per instruction, issued cooperatively by a warp.

Term rewriting system. A set of rules `lhs → rhs` applied to expression trees until none applies. The basis of `uopc`'s optimizer and its week-1 interpreter.

Termination. A rewrite system terminates if no infinite sequence of rewrites exists. `graph_rewrite` has a step limit as a safety net, not a proof.

`THREAD`. The `AxisType` of a loop run in parallel across CPU threads with OpenMP.

Toposort. An ordering of graph nodes with every node after its sources. `UOp.toposort` is iterative.

UOp. The IR node: `UOp(op, src, arg)`, immutable and hash-consed, with inferred `dtype`, `shape` and `addrspace`.

`.uir`. The course's text interchange format: one SSA statement per line, parsed by `uir.parse` and printed by `uir.to_uir`.

Unroll. Replicating a loop body so fewer iterations run; in `uopc`, `Opts.unroll` replicates the innermost reduce body and combines the copies as a balanced tree.

`UPCAST`. The `AxisType` for a fully unrolled axis. In `uopc`, upcasting is done during rangeify by instantiating the body per offset, so no UPCAST range is ever created.

`vmin` / `vmax`. The lower and upper bounds computed by bounds analysis.

Warp. A group of 32 GPU threads that execute in lockstep.

`WHERE`. Select: `WHERE(c, a, b)` is `a` where `c` is true, else `b`.

`WMMA`. Warp matrix multiply-accumulate: CUDA's API for tensor cores (`nvcuda::wmma`). `Ops.WMMA` is reserved in `uopc`; tensor-core kernels are produced by pattern (`gpu.wmma_gemm`).

C.2 Quiz bank

Questions are grouped by week. Answers refer to `uopc` and to measured numbers from `book/NOTES.md` where relevant.

Week 1: UOps, rewriting, symbolic

1. Why does `uopc`'s hash-consing cache key use `_argkey(arg)` rather than `arg`?

2. What is `-7 // 2` under `FLOORDIV`, and what does C's `-7 / 2` give?
3. Why is the rule `(x + c1) + c2 → x + (c1 + c2)` restricted to integers?
4. In `graph_rewrite`, why is the result of a rule that fired rewritten again?
5. `x` is an `i32` PARAM. What does `simplify((x*6 + 4) % 2)` return, and why is it valid for every `x`?
6. Why are `toposort` and `graph_rewrite` written with explicit stacks?
7. What are the shape and value of `INDEX(STACK(STACK(1,2,3), STACK(4,5,6)), CONST 1)`?
8. When inlining an outer CALL, why must the substitution not descend into a nested CALL's body?
9. What does folding `16777216.0 + 1.0` in f32 give, and why must the folder round through f32?

Answers, week 1

1. Python treats `1 == 1.0 == True` and `0.0 == -0.0` as equal with equal hashes, so a cache keyed on `arg` would merge an `i32`, an `f32` and a `bool` constant into one node. `_argkey` adds the type and uses `repr` for floats (NOTES 1).
2. `-4` (floor). C gives `-3` (truncation toward zero). That is why the renderer uses `floordiv_i32` unless bounds prove the numerator non-negative.
3. Float addition is not associative: `(x + 1.0) + 2.0` and `x + 3.0` can round differently, so the rewrite would change results (NOTES 7).
4. The result may itself match rules (or create new matches with its parent's context), and the engine promises a normal form: everything is rewritten to a fixed point.
5. `CONST 0`. `6x + 4 = 2·(3x + 2)`, and with floor semantics `(a·c + b) % c = b % c` holds for all integers, so the whole sum is divisible by 2.
6. Graphs get deep: a chain of 20,000 `ADD`s exceeded Python's recursion limit in a recursive rewriter (NOTES 3, `test_deep_graph_no_recursion_error`).
7. Shape `(3,)`, value `[4, 5, 6]`: `INDEX` pops the leading axis.
8. PARAMs are hash-consed, so `PARAM slot=0` in the inner body is the *same object* as in the outer body; a naive substitution would rewrite the inner function's parameter too (NOTES 2).
9. `16777216.0`: `2^24 + 1` is not representable in f32. Folding in Python's f64 would give `16777217.0`, a different answer from the compiled C program (NOTES 5).

Week 2: memory, rendering, runtime

1. What does `AFTER(mem, dep)` express, and why does an SSA graph need it?
2. Exactly when does the C renderer emit `/` for a `FLOORDIV`?
3. Which UOps does the renderer inline as expressions, and which always get a named variable?

4. Name one error that address-space inference catches and dtype/shape inference cannot.
5. Why does `compile_c` compile in a temporary directory *inside* the cache directory?

Answers, week 2

1. "`mem`, but only after `dep` has executed." A graph has no order except data dependence; a read of memory after a write to it has no data edge to the write unless one is added. `AFTER` is that edge.
2. When `vmin(numerator) >= 0` and the divisor is a positive constant; then truncation equals floor division. Otherwise it calls `floordiv_i32`.
3. Inlined: `CONST`, `INDEX`, `AFTER` (which renders as its memory), and ALU ops used exactly once. Named: every `LOAD` (`valN`), ALU ops used more than once (`aluN`); `ALLOC`s and `RANGE`s get declarations.
4. Loading from (or storing into) a plain value, for example `CONST.load()`: "LOAD needs memory".
5. `os.replace` is atomic only within one filesystem; renaming from `/tmp` into `~/.cache` failed across filesystems (NOTES 10). Same-filesystem rename also makes concurrent compiles of one kernel safe.

Week 3: loops and the linearizer

1. Define the *scope* of a node.
2. Why did the first linearizer design ("emit a node when its scope equals the open loops") deadlock on GEMM?
3. Where is an accumulator `ALLOC` declared, and what goes wrong with OpenMP if it is declared at the top of the kernel?
4. In `x - max(x)` computed per row, which two loops are siblings, and which must run first?
5. How does the loop-IR GEMM in `tests/test_w03_loops.py` keep its accumulator inside the `i` and `j` loops but outside `k`?

Answers, week 3

1. The set of `RANGE`s the node must sit inside: the union of its sources' scopes, plus itself for a `RANGE`, minus the ranges an `END` closes.
2. `A[i*K + k]` has scope `{i, k}`, but `k` is nested inside `j`, so the open loops are `(i, j, k)` whenever `k` is open; the scope never equals the open set and the node is never emitted. The fix closes scopes under "encloses" so scopes are paths in the loop tree (NOTES 8).
3. In the deepest scope common to all its users (inside the parallel loop), so each thread has a private copy. Declared at the top it is shared: a data race. At `-O3` gcc happened to keep it in a register and results were right; at `-O0` 20 of 20 runs were wrong (Appendix B).

4. Inside the row loop `i`: the reduce loop over `k` (computing the max) and the output loop over `j`. The `k` loop must run first, because the `j`-loop nodes read its result through its `END`.
5. `acc.after(i, j)` places the accumulator in scope `{i, j}`; its init store is there too; the update reads `acc.after(init, k)` inside `k`; the final read is `acc.after(upd.end(k))`, after the `k` loop closes.

Week 4: movement ops and rangeify

1. What does the syllabus's `gemm_256x256.uir` compute? Give the correct version.
2. State the index map rangeify uses for `EXPAND`, `SHRINK` and `PERMUTE`.
3. Why does rangeify simplify the index at every `RESHAPE`? Quote the measured effect.
4. `REDUCE` pops which axes? How do you sum axis 0 of a `(3, 4)` tensor?
5. What does rangeify turn a `STACK` into?
6. How is `PAD` built without a pad primitive?

Answers, week 4

1. `A * rowsum(B)`: reshaping `A` to `(N, N, 1)` and `B` to `(N, 1, N)` and reducing the last axis sums `B`'s rows. A GEMM reshapes `A` to `(N, 1, K)`, `B` *transposed* to `(1, N, K)`, multiplies and reduces the last axis (NOTES 11).
2. `EXPAND`: an axis that was size 1 reads index 0. `SHRINK`: add each axis's `begin`. `PERMUTE`: place each output index at the input axis it came from (the inverse permutation).
3. Ravel-then-unravel produces expressions like `((i*N + j) // N) % N` that equal `i` but mention `j`; the reduce lowering places accumulators in the loops an index *mentions*, so reductions were nested in `j`, making fused attention $O(N^3)$. Simplifying took N=512 fused attention from 1787 ms to 7 ms (NOTES 12).
4. The last `n`. Permute the axis to the end first: `x.permute((1, 0)).reduce(Ops.ADD, 1)`, shape `(4,)`.
5. A chain of `WHERE`s on the first index: `where(i < 1, a[rest], where(i < 2, b[rest], c[rest]))`.
6. Concatenate the tensor with zeros along the axis (a `STACK` moved next to the axis and reshaped), then `SHRINK` back to the padded size (`composite.pad_axis`). Rangeify plus symbolic turn it into a bounds-checked `WHERE`.

Week 5: kernelize, autodiff, the Tensor library

1. What are the materialization points of `kernelize`?
2. Why can fusing an elementwise result into a reduce through an `EXPAND` be a bad idea? What rule does `uopc` use, and what did it save on GPT?
3. Why does `make_plan` renumber buffers `0..n-1` within each kernel?

4. Why does AdamW pass its bias corrections as one-element tensors rather than Python floats?
5. Give the gradient rules for `EXPAND`, `SHRINK` and `PERMUTE`.
6. How does `uopc` split the gradient of `REDUCE(MAX)` among tied maxima?
7. Why does autodiff refuse `STORE`, `CALL`, `AFTER`, `RANGE` and `END`?
8. Why is `nn.Embedding` a one-hot matmul?

Answers, week 5

1. Every root (unless it is already a contiguous buffer), every `STAGE` (except `STAGE(x, "detach")`, which is stripped), and every `REDUCE` unless `fuse_reduce` is on.
2. The fused value is recomputed for every element it is broadcast to: a LayerNorm output feeding a matmul is recomputed per output column. `stage_broadcast_compute` stages an ALU value expanded by 4x or more when the recompute cost (`_fused_work`) is at least 4096; it took a GPT step's kernel time from 374 ms to 125 ms (NOTES 15).
3. So structurally identical kernels render byte-identical C, which hits the content-addressed `.so` cache.
4. They change every step; as `CONST`s they make each step's graph structurally different, missing the plan cache and recompiling everything (NOTES 17). As data, the graph is identical every step.
5. `EXPAND`: sum the gradient over the expanded axes (`sum_to`). `SHRINK`: pad the gradient with zeros back to the input shape. `PERMUTE`: apply the inverse permutation.
6. It builds a mask `x == max`, counts the ties, and gives each tied element `grad / count`.
7. Those ops carry effects and control structure, not mathematical functions of their inputs; autodiff works on pure shape graphs, before rangeify introduces them.
8. A data-dependent `INDEX` (a gather) has no gradient rule in `uopc`; a one-hot matrix times the embedding table is differentiable with existing rules.

Week 6: fast CPU kernels

1. Why does register tiling raise arithmetic intensity? Use the 4×8 tile from `opt.py`.
2. Why did adding threads make the strided 1024^3 GEMM *slower*?
3. What does `-fassociative-math` allow, and what is the cost?
4. With `Opts(upcast={0: 2, 1: 2})`, how many `for` loops and accumulators does a GEMM kernel have?
5. What does the layout pass do, and when does it stage an operand?
6. How do `hand_opt` and `search_opt` differ?

Answers, week 6

1. Each step of `k` loads 4 values of A and 8 of B (12 loads) and performs 32 multiply-adds into 32 accumulators held in registers: about 2.7 multiply-adds per load, versus 1/2 for the naive loop.
2. Each thread streamed down a column of B (stride N), touching a new cache line and a new 4 KB page every iteration; more threads meant more concurrent cache and TLB misses: 1.9 → 1.1 GFLOP/s (NOTES 19).
3. Reordering float additions, which lets gcc vectorize reductions (17 → 170 GFLOP/s in one experiment, NOTES 20). Results differ from the strict left-to-right sum at the rounding level.
4. Three loops (two output loops and one shared reduce loop) and four accumulators (`test_upcast_shares_reduce_loop`).
5. It pushes movement ops through elementwise ops to the data (`push_views`), finds operands whose innermost reduce axis is strided, and stages them contiguously, but only if the staged view is no larger than the base data and strictly smaller than the reduce input (NOTES 21).
6. `hand_opt` applies fixed rules (a 4×4 tile on the last two axes, threads if the work is at least 2^{16}, unroll 4 for small reduces); `search_opt` compiles candidate `Opts` and keeps the one that runs fastest on random data.

Weeks 7–8: GPUs, shared memory, tensor cores

1. Which kernel axis gets `threadIdx.x`, and why?
2. Why does `smem_gemm_ir` need two barriers per tile step?
3. What does `gpu.emulate` let you test without a GPU, and what can it not test?
4. By what factor does T×T tiling in shared memory cut global-memory traffic of a GEMM?
5. What are the operand and accumulator types of a typical tensor-core MMA, and what tile does `wmma_gemm` use?

Answers, weeks 7–8

1. The last (fastest-varying, contiguous) axis, so neighboring threads of a warp touch neighboring addresses and their accesses coalesce (NOTES 23).
2. The first ensures the whole tile is loaded before any thread reads it; the second ensures every thread has finished reading before the next step's copy overwrites the tile.
3. The index mapping of `GLOBAL` / `LOCAL` ranges (by running them as loops in C). Not anything that needs threads to run concurrently: barriers and shared memory (NOTES 25).
4. A factor of T: each element loaded into shared memory is reused by T threads.
5. f16 (or bf16) inputs with f32 accumulation; `wmma_gemm` uses 16×16×16 fragments, one warp per 16×16 output tile.

Weeks 9–10: real models

1. Why does `examples/gpt.py` pad the sampling context to a fixed length?
2. How does `examples/gpt.py` make attention causal?
3. What state does online softmax keep per query row?
4. What were the measured effects, on GPT, of `stage_broadcast_compute` and of identity-based hashing plus memoized `simplify`?

Answers, weeks 9–10

1. So every forward pass has the same shapes, the same graph structure, and therefore one compiled plan (NOTES 18).
2. It adds a precomputed mask, `-1e9` above the diagonal (`np.triu(..., 1)`), sliced to `[:T, :T]`, to the scores before the softmax.
3. A running maximum `m`, a running sum `l` of exponentials, and the output accumulator `o[D]`, all rescaled by `exp(m_old - m_new)` when the maximum grows.
4. Kernel time per step 374 → 125 ms (NOTES 15); compile time for 12 steps 163 → 47 s, first step about 9 s (NOTES 22).

Beyond (Chapter 15)

1. Give the KV-cache size formula and evaluate it for 2 layers, `dim` 64, 64 tokens, f32.
2. What do *declared bounds* on a symbolic size buy the compiler?
3. In data-parallel training, what is communicated each step?

Answers, Chapter 15

1. `2 · layers · T · dim · bytes`: 2 · 2 · 64 · 64 · 4 = 65,536 bytes (64 KiB).
2. Bounds analysis keeps working: index expressions cannot overflow, div/mod rules fire, the renderer can use `/` instead of `floordiv_i32`, and temp buffers can be allocated at the upper bound. Without bounds a loaded size has full `i32` range and `bounds()` gives up.
3. Gradients, summed (or averaged) across devices with an all-reduce.

Appendix D: Complete source code

This appendix contains the whole repository that the book builds: the compiler (`uopc/`), its tests (`tests/`), and the example programs (`examples/`). Each file is included at build time from the same commit as the text, so it always matches the listings in the chapters. Click a file name to expand it. In the PDF every file is printed in full.

The files are in the order the course introduces them. Line counts include comments and blank lines.

D.1 The compiler: `uopc/`

Week	Modules
1	`uop.py`, `dtype.py`, `rewrite.py`, `symbolic.py`, `uir.py`
2–3	`render_c.py`, `runtime.py`, `linearize.py`
4	`rangeify.py`, `composite.py`
5	`kernelize.py`, `autodiff.py`, `tensor.py`, `nn.py`
6	`opt.py`
7–8	`gpu.py`, `cuda.py`

uopc/ init .py · 13 lines

```python
"""uopc: a machine learning compiler built from scratch, one week at a time.

  week 1   uop, rewrite, symbolic, uir      the IR, rewriting, algebra, the text format
  week 2   render_c, runtime, linearize     memory, C code, compiling and running it
  week 3   linearize                        loops and scopes
  week 4   rangeify, composite              shapes -> loops, PAD/FLIP from primitives
  week 5   kernelize, autodiff, tensor, nn  fusion/fission, gradients, a Tensor library
  week 6   opt                              upcast, unroll, threads, layout
  week 7-8 gpu, cuda                        GPU mapping, CUDA, shared memory, tensor cores
"""
from uopc.uop import UOp, Ops, AxisType, ParamArg, CallArg, RangeArg, sink, range_
from uopc.dtype import dtypes, DType, AddrSpace
from uopc.tensor import Tensor
```

uopc/dtype.py · 72 lines

```python
"""Week 1: data types and address spaces.

A DType is a tiny frozen record. We keep the set deliberately small:
void, bool, i32, f32 (weeks 1-6) and f16 (week 8, tensor cores).
"""
from __future__ import annotations
import math, struct
from dataclasses import dataclass
from enum import Enum, auto


@dataclass(frozen=True, eq=True)
class DType:
  name: str
  itemsize: int
  ctype: str
  fmt: str | None  # struct / numpy format character

  def __repr__(self): return f"dtypes.{self.name}"
  @property
  def is_float(self) -> bool: return self.name in ("f32", "f16")
  @property
  def is_int(self) -> bool: return self.name == "i32"


class dtypes:
  void = DType("void", 0, "void", None)
  bool = DType("bool", 1, "_Bool", "?")
  i32 = DType("i32", 4, "int", "i")
  f32 = DType("f32", 4, "float", "f")
  f16 = DType("f16", 2, "_Float16", "e")
  all = (void, bool, i32, f32, f16)

  @staticmethod
  def from_name(name: str) -> DType:
    for d in dtypes.all:
      if d.name == name: return d
    raise ValueError(f"unknown dtype {name!r}")

  @staticmethod
  def from_py(x) -> DType:
    if isinstance(x, bool): return dtypes.bool
    if isinstance(x, int): return dtypes.i32
    if isinstance(x, float): return dtypes.f32
    raise TypeError(f"no dtype for python value {x!r}")

  @staticmethod
  def min(d: DType):
    return {"bool": False, "i32": -2**31}.get(d.name, -math.inf)

  @staticmethod
  def max(d: DType):
```

```python
    return {"bool": True, "i32": 2**31 - 1}.get(d.name, math.inf)


def truncate(d: DType, x):
  """Bring a python value into the exact value set of dtype d (used by const folding)."""
  if d == dtypes.bool: return bool(x)
  if d == dtypes.i32: return ((int(x) + 2**31) % 2**32) - 2**31   # two's complement
wraparound
  if d == dtypes.f32: return struct.unpack("f", struct.pack("f", float(x)))[0] if
math.isfinite(x) else float(x)
  if d == dtypes.f16: return struct.unpack("e", struct.pack("e", float(x)))[0] if
math.isfinite(x) and abs(x) < 65520 else float(x)
  raise TypeError(f"can't truncate to {d}")


class AddrSpace(Enum):
  """Where a UOp lives. ALU = a plain value in a register. The rest are memory."""
  ALU = auto()      # a value (week 1)
  GLOBAL = auto()   # device memory, a function argument (week 2)
  LOCAL = auto()    # call-scope storage: `float acc[4];` (week 2)
  SHARED = auto()   # GPU workgroup shared memory (week 7)

  def __repr__(self): return f"AddrSpace.{self.name}"
```

uopc/uop.py · 320 lines

```python
"""Week 1: the UOp.

    UOp(op, src: tuple[UOp, ...], arg)

Every program in this course is a DAG of UOps. UOps are *hash consed*:
constructing a UOp that is structurally identical to a live one returns the
very same Python object. That gives us O(1) structural equality (`is`),
free common-subexpression elimination, and cheap memoization keyed on nodes.

dtype, shape and addrspace are *recursive properties*: they are computed
from the op, the arg and the sources, never stored by the user.
"""
from __future__ import annotations
import math, weakref
from dataclasses import dataclass
from enum import Enum, auto
from typing import Any
from uopc.dtype import DType, dtypes, AddrSpace


class Ops(Enum):
  # ---- week 1: values ----------------------------------------------------
  PARAM = auto()      # arg: ParamArg. An input to the enclosing CALL.
  CALL = auto()       # src[0] = body, src[1:] = arguments. arg: CallArg
  CONST = auto()      # arg: python value (bool/int/float). shape ()
  STACK = auto()      # new leading axis; all srcs share one shape
  INDEX = auto()      # src[0] indexed by src[1:]; pops axes off the left
  CAST = auto()       # arg: DType
  RECIP = auto()      # 1/x
  ADD = auto(); MUL = auto(); CMPLT = auto()
  FLOORDIV = auto(); FLOORMOD = auto()
  WHERE = auto()      # (cond, a, b)
  # extra ALU, introduced when we need them (masks, max-reduce, softmax)
  CMPNE = auto(); MAX = auto()
  EXP2 = auto(); LOG2 = auto(); SQRT = auto(); SIN = auto()
  # ---- week 2: memory ----------------------------------------------------
  BUFFER = auto()     # arg: ParamArg. global persistent memory (never in a CALL body)
  ALLOC = auto()      # arg: ParamArg. call-scope storage `float a[10];`
  LOAD = auto()       # (ptr,)
  STORE = auto()      # (ptr, value)
  AFTER = auto()      # (mem, *deps): `mem`, but only once deps have happened
  SINK = auto()       # (*effects): groups side effects, the root of a program
  # ---- week 3: loops -----------------------------------------------------
  RANGE = auto()      # src[0] = end; arg: RangeArg. value is the loop counter
  END = auto()        # (passthrough, *ranges): closes the ranges
  # ---- week 4: movement + reduction --------------------------------------
  RESHAPE = auto(); EXPAND = auto(); PERMUTE = auto(); SHRINK = auto()
  REDUCE = auto()     # arg: (Ops.ADD | Ops.MAX, n) -- reduces the last n axes
  # ---- week 5: kernels ---------------------------------------------------
  STAGE = auto()      # materialize src[0] into a temp buffer you can index into
  # ---- weeks 7-8: GPUs ---------------------------------------------------
  BARRIER = auto()    # (*deps): __syncthreads()
```

```
  WMMA = auto()       # reserved for tensor cores (week 8 lowers GEMMs to WMMA by pattern
instead)

  def __repr__(self): return f"Ops.{self.name}"
  __hash__ = object.__hash__   # members are singletons: identity hashing is correct and
much faster


UNARY = {Ops.RECIP, Ops.CAST, Ops.EXP2, Ops.LOG2, Ops.SQRT, Ops.SIN}
BINARY = {Ops.ADD, Ops.MUL, Ops.CMPLT, Ops.CMPNE, Ops.FLOORDIV, Ops.FLOORMOD, Ops.MAX}
ALU = UNARY | BINARY | {Ops.WHERE}
COMMUTATIVE = {Ops.ADD, Ops.MUL, Ops.MAX, Ops.CMPNE}
MOVEMENT = {Ops.RESHAPE, Ops.EXPAND, Ops.PERMUTE, Ops.SHRINK}


class AxisType(Enum):
  LOOP = auto()      # a plain serial for-loop
  REDUCE = auto()    # a for-loop that a reduction accumulates over
  UPCAST = auto()    # fully unrolled (week 6). uopc unrolls *during* rangeify
(Opts.upcast), so no RANGE of
                     # this kind is ever built; the name stays for the course's vocabulary
  THREAD = auto()    # parallel across CPU threads (week 6)
  GLOBAL = auto()    # GPU blockIdx (week 7)
  LOCAL = auto()     # GPU threadIdx (week 7)
  def __repr__(self): return f"AxisType.{self.name}"
  __hash__ = object.__hash__


@dataclass(frozen=True)
class ParamArg:
  slot: int
  dtype: DType
  shape: tuple[int, ...] = ()
  addrspace: AddrSpace | None = None   # only used by ALLOC (LOCAL or SHARED)


@dataclass(frozen=True)
class CallArg:
  name: str = "f"


@dataclass(frozen=True, order=True)
class RangeArg:
  idx: int                        # loops nest in increasing idx order
  kind: AxisType = AxisType.LOOP


def prod(xs) -> int: return math.prod(xs)


def _argkey(arg):
  # 1 == 1.0 == True and 0.0 == -0.0 in python; the cache must not merge them.
  if isinstance(arg, float): return (float, repr(arg))
```

```
  if isinstance(arg, tuple): return (tuple, tuple(_argkey(a) for a in arg))
  return (type(arg), arg)


class UOp:
  __slots__ = ("op", "src", "arg", "dtype", "shape", "addrspace", "__weakref__", "__dict__")
  _cache: weakref.WeakValueDictionary = weakref.WeakValueDictionary()

  def __new__(cls, op: Ops, src: tuple[UOp, ...] = (), arg: Any = None):
    key = (op, src, _argkey(arg))
    if (ret := UOp._cache.get(key)) is not None: return ret
    ret = super().__new__(cls)
    ret.op, ret.src, ret.arg = op, src, arg
    ret.dtype, ret.shape, ret.addrspace = _infer(op, src, arg)
    UOp._cache[key] = ret
    return ret

  # hash consing means identity *is* structural equality, so we keep object's default
  # __eq__/__hash__ (identity, implemented in C). Note __lt__ etc. build UOps instead.
  def __reduce__(self): return (UOp, (self.op, self.src, self.arg))
  def __repr__(self):
    s = f"UOp({self.op}, arg={self.arg!r}" if self.arg is not None else f"UOp({self.op}"
    return s + (f", src=({', '.join(repr(x) for x in self.src)},))" if self.src else ")")

  def replace(self, **kw) -> UOp:
    return UOp(kw.get("op", self.op), kw.get("src", self.src), kw.get("arg", self.arg))

  # ---- graph helpers ------------------------------------------------------
  def toposort(self) -> list[UOp]:
    """Every node reachable from self, children before parents. Iterative (graphs get
deep)."""
    out, seen, stack = [], set(), [(self, False)]
    while stack:
      u, done = stack.pop()
      if done: out.append(u); continue
      if u in seen: continue
      seen.add(u)
      stack.append((u, True))
      stack.extend((s, False) for s in reversed(u.src) if s not in seen)
    return out

  def substitute(self, mapping: dict[UOp, UOp]) -> UOp:
    """Replace nodes (by identity) everywhere in the graph. Rebuilds only what changed."""
    new: dict[UOp, UOp] = {}
    for u in self.toposort():
      if u in mapping: new[u] = mapping[u]; continue
      src = tuple(new[s] for s in u.src)
      new[u] = u if src == u.src else u.replace(src=src)
    return new[self]

  @staticmethod
  def wrap(roots: list[UOp], targets: set[UOp], fn) -> list[UOp]:
    """Rebuild the graph bottom-up, replacing each node t in targets by fn(t'), where t' is
```

```
t
    with *its* sources already rebuilt. (substitute() would wrap the original, unrewritten
t.)"""
    new: dict[UOp, UOp] = {}
    for u in UOp(Ops.SINK, tuple(roots)).toposort():
      src = tuple(new[s] for s in u.src)
      n = u if src == u.src else u.replace(src=src)
      new[u] = fn(n) if u in targets else n
    return [new[r] for r in roots]

  def ranges(self) -> list[UOp]:
    return sorted((u for u in self.toposort() if u.op is Ops.RANGE), key=lambda u: u.arg)

  # ---- constructors ----------------------------------------------------
  @staticmethod
  def const(v, dtype: DType | None = None) -> UOp:
    if dtype == dtypes.f16: return UOp(Ops.CONST, (), float(v)).cast(dtypes.f16)   # python
has no f16 values
    if dtype is not None and dtype != dtypes.from_py(v):
      v = {"bool": bool, "i32": int}.get(dtype.name, float)(v)
    return UOp(Ops.CONST, (), v)

  def const_like(self, v) -> UOp:
    """A constant with self's dtype *and shape* (broadcast with RESHAPE + EXPAND)."""
    c = UOp.const(v, self.dtype)
    return c.reshape((1,) * len(self.shape)).expand(self.shape) if self.shape else c
  def _c(self, x) -> UOp: return x if isinstance(x, UOp) else self.const_like(x)
  def alu(self, op: Ops, *src) -> UOp: return UOp(op, (self, *(self._c(s) for s in src)))

  def __add__(self, x): return self.alu(Ops.ADD, x)
  def __radd__(self, x): return self._c(x).alu(Ops.ADD, self)
  def __mul__(self, x): return self.alu(Ops.MUL, x)
  def __rmul__(self, x): return self._c(x).alu(Ops.MUL, self)
  def __neg__(self): return self.alu(Ops.MUL, -1 if self.dtype == dtypes.i32 else -1.0)
  def __sub__(self, x): return self + -self._c(x)
  def __rsub__(self, x): return self._c(x) + -self
  def __floordiv__(self, x): return self.alu(Ops.FLOORDIV, x)
  def __mod__(self, x): return self.alu(Ops.FLOORMOD, x)
  def __lt__(self, x): return self.alu(Ops.CMPLT, x)
  def __gt__(self, x): return self._c(x).alu(Ops.CMPLT, self)
  def ne(self, x): return self.alu(Ops.CMPNE, x)
  def eq(self, x): return self.ne(x).ne(True)
  def maximum(self, x): return self.alu(Ops.MAX, x)
  def recip(self): return UOp(Ops.RECIP, (self,))
  def cast(self, d: DType): return self if d == self.dtype else UOp(Ops.CAST, (self,), d)
  def where(self, a, b):
    a = a if isinstance(a, UOp) else (b.const_like(a) if isinstance(b, UOp) else
UOp.const(a))
    b = b if isinstance(b, UOp) else a.const_like(b)
    return UOp(Ops.WHERE, (self, a, b))

  def index(self, *idx) -> UOp:
    return UOp(Ops.INDEX, (self, *(i if isinstance(i, UOp) else UOp.const(int(i)) for i in
```

```
idx)))
  def load(self) -> UOp: return UOp(Ops.LOAD, (self,))
  def store(self, v) -> UOp: return UOp(Ops.STORE, (self, v))
  def after(self, *deps) -> UOp: return UOp(Ops.AFTER, (self, *deps)) if deps else self
  def end(self, *rngs) -> UOp: return UOp(Ops.END, (self, *rngs)) if rngs else self

  def reshape(self, shape): return UOp(Ops.RESHAPE, (self,), tuple(shape))
  def expand(self, shape): return UOp(Ops.EXPAND, (self,), tuple(shape))
  def permute(self, perm): return UOp(Ops.PERMUTE, (self,), tuple(perm))
  def shrink(self, arg): return UOp(Ops.SHRINK, (self,), tuple(tuple(x) for x in arg))
  def reduce(self, op: Ops, n: int = 1): return UOp(Ops.REDUCE, (self,), (op, n))


def sink(*srcs: UOp) -> UOp: return UOp(Ops.SINK, tuple(srcs))
def range_(end: int | UOp, idx: int, kind: AxisType = AxisType.LOOP) -> UOp:
  return UOp(Ops.RANGE, (end if isinstance(end, UOp) else UOp.const(int(end)),),
RangeArg(idx, kind))


# ---------------------------------------------------------------------------
# shape / dtype / addrspace inference. This is the type checker of the IR:
# every malformed UOp dies here, at construction time, with a useful message.

class UOpError(Exception): pass

def _check(cond, msg):
  if not cond: raise UOpError(msg)

def _infer(op: Ops, src: tuple[UOp, ...], arg) -> tuple[DType, tuple[int, ...], AddrSpace]:
  A = AddrSpace.ALU
  if op is Ops.CONST:
    _check(not src, "CONST has no sources")
    return dtypes.from_py(arg), (), A
  if op is Ops.PARAM: return arg.dtype, arg.shape, A
  if op is Ops.BUFFER:
    _check(len(arg.shape) == 1, "BUFFER is flat")
    return arg.dtype, arg.shape, AddrSpace.GLOBAL
  if op is Ops.ALLOC:
    _check(len(arg.shape) == 1, "ALLOC is flat")
    return arg.dtype, arg.shape, arg.addrspace or AddrSpace.LOCAL
  if op is Ops.CALL:
    params = {u.arg.slot: u for u in call_params(src[0])}
    _check(sorted(params) == list(range(len(src) - 1)), f"CALL needs params 0..{len(src)-2},
body has {sorted(params)}")
    for i, a in enumerate(src[1:]):
      p = params[i]
      _check(a.shape == p.shape and a.dtype == p.dtype, f"CALL arg {i}: {a.dtype}{a.shape}
!= param {p.dtype}{p.shape}")
    return src[0].dtype, src[0].shape, src[0].addrspace
  if op is Ops.STACK:
    _check(len(src) > 0, "STACK of nothing")
    _check(all(s.shape == src[0].shape and s.dtype == src[0].dtype for s in src), "STACK
srcs must agree in shape and dtype")
```

```
    return src[0].dtype, (len(src),) + src[0].shape, A
  if op is Ops.INDEX:
    x, idx = src[0], src[1:]
    _check(len(idx) <= len(x.shape), f"INDEX: {len(idx)} indices into shape {x.shape}")
    _check(all(i.dtype == dtypes.i32 and i.shape == () for i in idx), "INDEX indices are i32
scalars")
    return x.dtype, x.shape[len(idx):], x.addrspace
  if op in ALU:
    if op is Ops.WHERE:
      c, a, b = src
      _check(c.dtype == dtypes.bool, "WHERE condition must be bool")
      _check(a.dtype == b.dtype, f"WHERE branches disagree: {a.dtype} vs {b.dtype}")
      _check(c.shape == a.shape == b.shape, f"WHERE shape mismatch {c.shape} {a.shape}
{b.shape}")
      return a.dtype, a.shape, A
    if op in UNARY:
      _check(len(src) == 1, f"{op} is unary")
      if op is Ops.CAST: return arg, src[0].shape, A
      _check(src[0].dtype.is_float, f"{op} needs a float, got {src[0].dtype}")
      return src[0].dtype, src[0].shape, A
    a, b = src
    _check(a.shape == b.shape, f"{op}: shape mismatch {a.shape} vs {b.shape} (broadcast with
EXPAND)")
    _check(a.dtype == b.dtype, f"{op}: dtype mismatch {a.dtype} vs {b.dtype} (use CAST)")
    if op in (Ops.FLOORDIV, Ops.FLOORMOD): _check(a.dtype == dtypes.i32, f"{op} is integer
only")
    return (dtypes.bool if op in (Ops.CMPLT, Ops.CMPNE) else a.dtype), a.shape, A
  if op is Ops.LOAD:
    _check(src[0].addrspace is not A, "LOAD needs memory")
    return src[0].dtype, src[0].shape, A
  if op is Ops.STORE:
    _check(src[0].addrspace is not A, "STORE needs memory")
    _check(src[0].dtype == src[1].dtype, f"STORE of {src[1].dtype} into {src[0].dtype}")
    return dtypes.void, (), A
  if op is Ops.AFTER: return src[0].dtype, src[0].shape, src[0].addrspace
  if op in (Ops.SINK, Ops.END, Ops.BARRIER): return dtypes.void, (), A
  if op is Ops.RANGE:
    _check(src[0].dtype == dtypes.i32, "RANGE end is i32")
    return dtypes.i32, (), A
  if op is Ops.RESHAPE:
    _check(prod(arg) == prod(src[0].shape), f"RESHAPE {src[0].shape} -> {arg}")
    return src[0].dtype, arg, src[0].addrspace
  if op is Ops.EXPAND:
    s = src[0].shape
    _check(len(arg) == len(s) and all(a == b or a == 1 for a, b in zip(s, arg)), f"EXPAND
{s} -> {arg}")
    return src[0].dtype, arg, src[0].addrspace
  if op is Ops.PERMUTE:
    _check(sorted(arg) == list(range(len(src[0].shape))), f"PERMUTE {arg} of
{src[0].shape}")
    return src[0].dtype, tuple(src[0].shape[i] for i in arg), src[0].addrspace
  if op is Ops.SHRINK:
    s = src[0].shape
```

```
    _check(len(arg) == len(s) and all(0 <= b <= e <= n for (b, e), n in zip(arg, s)),
f"SHRINK {arg} of {s}")
    return src[0].dtype, tuple(e - b for b, e in arg), src[0].addrspace
  if op is Ops.REDUCE:
    rop, n = arg
    _check(rop in (Ops.ADD, Ops.MAX) and 0 <= n <= len(src[0].shape), f"REDUCE {arg} of
{src[0].shape}")
    return src[0].dtype, src[0].shape[:len(src[0].shape) - n], A
  if op is Ops.STAGE: return src[0].dtype, src[0].shape, A
  if op is Ops.WMMA: return src[2].dtype, src[2].shape, A
  raise UOpError(f"no inference rule for {op}")


def call_params(body: UOp) -> list[UOp]:
  """PARAMs that belong to this body. A nested CALL's body is its own scope, so we don't
look inside it."""
  out, seen, stack = [], set(), [body]
  while stack:
    u = stack.pop()
    if u in seen: continue
    seen.add(u)
    if u.op is Ops.PARAM: out.append(u)
    stack.extend(u.src[1:] if u.op is Ops.CALL else u.src)
  return out
```

uopc/rewrite.py · 115 lines

```python
"""Week 1: pattern matching and graph rewriting.

A *rewrite rule* is (pattern, function). The pattern is a UPat tree that matches
UOps structurally and binds names; the function receives the bindings as keyword
arguments and returns a replacement UOp, or None for "rule does not apply".

graph_rewrite applies a PatternMatcher bottom-up and greedily: children are
rewritten to a fixed point before their parent is looked at, and whenever a rule
fires the result is itself rewritten again. Because UOps are hash consed, the
memo table is just a dict keyed by node.
"""
from __future__ import annotations
import itertools
from typing import Any, Callable
from uopc.uop import UOp, Ops
from uopc.dtype import DType


class UPat:
  __slots__ = ("op", "src", "name", "arg", "dtype", "allow_any_len")

  def __init__(self, op: Ops | tuple[Ops, ...] | set[Ops] | None = None, src=None, name: str
| None = None,
               arg: Any = None, dtype: DType | tuple[DType, ...] | None = None,
allow_any_len: bool = False):
    self.op = None if op is None else (frozenset([op]) if isinstance(op, Ops) else
frozenset(op))
    self.dtype = None if dtype is None else (frozenset([dtype]) if isinstance(dtype, DType)
else frozenset(dtype))
    # src: tuple -> match in order. list -> match in any order (for commutative ops).
    self.src, self.name, self.arg, self.allow_any_len = src, name, arg, allow_any_len

  @staticmethod
  def var(name: str | None = None, dtype=None) -> UPat: return UPat(name=name, dtype=dtype)
  @staticmethod
  def cvar(name: str | None = None, dtype=None) -> UPat: return UPat(Ops.CONST, name=name,
dtype=dtype)

  def __repr__(self): return f"UPat({self.op and set(self.op)}, name={self.name})"

  def match(self, u: UOp, store: dict[str, UOp]) -> list[dict[str, UOp]]:
    """All ways this pattern matches u, each a dict of bindings."""
    if self.op is not None and u.op not in self.op: return []
    if self.dtype is not None and u.dtype not in self.dtype: return []
    if self.arg is not None and u.arg != self.arg: return []
    if self.name is not None:
      if self.name in store and store[self.name] is not u: return []   # same name must bind
same node
      store = {**store, self.name: u}
    if self.src is None: return [store]
    orders = itertools.permutations(self.src) if isinstance(self.src, list) else [self.src]
    out = []
```

```
    for pats in orders:
      if len(pats) != len(u.src) and not (self.allow_any_len and len(pats) <= len(u.src)):
continue
      stores = [store]
      for p, s in zip(pats, u.src):
        stores = [m for st in stores for m in p.match(s, st)]
        if not stores: break
      out.extend(stores)
    return out

  # sugar so rules read like math: UPat.var("x") + UPat.cvar("c")
  def _bin(self, op, other, swap=False):
    other = other if isinstance(other, UPat) else UPat(Ops.CONST, arg=other)
    src = [other, self] if swap else [self, other]
    return UPat(op, src if op in (Ops.ADD, Ops.MUL, Ops.MAX, Ops.CMPNE) else tuple(src))
  def __add__(self, o): return self._bin(Ops.ADD, o)
  def __mul__(self, o): return self._bin(Ops.MUL, o)
  def __floordiv__(self, o): return self._bin(Ops.FLOORDIV, o)
  def __mod__(self, o): return self._bin(Ops.FLOORMOD, o)
  def __lt__(self, o): return self._bin(Ops.CMPLT, o)


Rule = tuple[UPat, Callable[..., UOp | None]]


class PatternMatcher:
  def __init__(self, rules: list[Rule]):
    self.rules = rules
    # index by root op so we only try rules that could possibly match
    self.by_op: dict[Ops, list[Rule]] = {}
    for pat, fn in rules:
      for op in (pat.op if pat.op is not None else Ops):
        self.by_op.setdefault(op, []).append((pat, fn))

  def __add__(self, other: PatternMatcher) -> PatternMatcher: return
PatternMatcher(self.rules + other.rules)

  def rewrite(self, u: UOp, ctx=None) -> UOp | None:
    for pat, fn in self.by_op.get(u.op, ()):
      for binding in pat.match(u, {}):
        kw = {**binding, "ctx": ctx} if ctx is not None else binding
        if (ret := fn(**kw)) is not None and ret is not u: return ret
    return None


def graph_rewrite(root: UOp, pm: PatternMatcher, ctx=None, max_steps: int = 10_000_000) ->
UOp:
  """Bottom-up greedy rewrite to a fixed point. Iterative, so deep graphs don't blow the
stack.

  Stage 0: schedule children.   Stage 1: children done -> rebuild, try rules.
  Stage 2: a rule fired and we rewrote its result -> record that as our answer.
  """
```

```
  done: dict[UOp, UOp] = {}
  stack: list[tuple[UOp, int, UOp | None]] = [(root, 0, None)]
  steps = 0
  while stack:
    u, stage, aux = stack.pop()
    if u in done and stage != 2: continue
    if stage == 0:
      stack.append((u, 1, None))
      stack.extend((s, 0, None) for s in reversed(u.src) if s not in done)
    elif stage == 1:
      src = tuple(done[s] for s in u.src)
      n = u if src == u.src else u.replace(src=src)
      steps += 1
      if steps > max_steps: raise RuntimeError("graph_rewrite did not converge (a rule
loops?)")
      if (m := pm.rewrite(n, ctx)) is None: done[u] = n
      elif m in done: done[u] = done[m]
      else: stack.extend([(u, 2, m), (m, 0, None)])
    else:
      done[u] = done[aux]
  return done[root]
```

uopc/symbolic.py · 302 lines

```python
"""Week 1: constant folding, symbolic simplification, and "running" programs by rewriting.

Three rule sets:
  * fold      -- evaluate ALU ops whose inputs are all constants
  * symbolic  -- algebra: identities, reassociation, div/mod, bounds (vmin/vmax)
  * evaluate  -- push ALU through STACK, resolve INDEX of STACK, inline CALLs

graph_rewrite(program, fold + symbolic + evaluate) on a closed program (no PARAMs,
no memory) reduces it all the way to CONST/STACK: rewriting *is* the interpreter.
"""
from __future__ import annotations
import math
from uopc.uop import UOp, Ops, ALU, call_params, prod
from uopc.dtype import dtypes, truncate
from uopc.rewrite import UPat, PatternMatcher, graph_rewrite


# ---------------------------------------------------------------------------
# 1. the reference semantics of every ALU op. The C renderer must agree with this.

def _safe(f, *a):
  try: return f(*a)
  except (OverflowError, ValueError, ZeroDivisionError): return None

def exec_alu(op: Ops, dtype, vals: list):
  # inputs first: an f32 CONST may hold a double (16777217.0) that C rounds when it parses
the literal
  if op is not Ops.CAST and dtype.is_float: vals = [truncate(dtype, v) if isinstance(v,
float) else v for v in vals]
  if op is Ops.WHERE and dtype.is_float: vals = [vals[0]] + [truncate(dtype, v) for v in
vals[1:]]
  match op:
    case Ops.ADD: r = vals[0] + vals[1]
    case Ops.MUL: r = vals[0] * vals[1] if not (dtype.is_float and any(math.isinf(v) for v
in vals) and 0 in vals) else math.nan
    case Ops.MAX: r = max(vals[0], vals[1])
    case Ops.CMPLT: return vals[0] < vals[1]
    case Ops.CMPNE: return vals[0] != vals[1]
    case Ops.FLOORDIV:
      if vals[1] == 0 or (dtype == dtypes.i32 and vals[0] == -2**31 and vals[1] == -1):
return None   # C traps here
      r = vals[0] // vals[1]
    case Ops.FLOORMOD: r = vals[0] % vals[1] if vals[1] != 0 else None
    case Ops.WHERE: r = vals[1] if vals[0] else vals[2]
    case Ops.RECIP: r = 1.0 / vals[0] if vals[0] != 0 else math.copysign(math.inf, vals[0])
    case Ops.EXP2: r = _safe(math.pow, 2.0, vals[0]) if vals[0] < 1024 else math.inf
    case Ops.LOG2: r = math.log2(vals[0]) if vals[0] > 0 else (-math.inf if vals[0] == 0
else math.nan)
    case Ops.SQRT: r = math.sqrt(vals[0]) if vals[0] >= 0 else math.nan
    case Ops.SIN: r = math.sin(vals[0]) if math.isfinite(vals[0]) else math.nan
    case Ops.CAST:
      v = vals[0]
```

```
      if dtype == dtypes.i32:
        if not math.isfinite(v): return None          # undefined behavior in C: refuse to
fold
        r = int(v)                                    # C truncates toward zero
      elif dtype == dtypes.bool: return bool(v)
      else: r = float(v)
    case _: return None
  return None if r is None else truncate(dtype, r)


# ---------------------------------------------------------------------------
# 2. bounds analysis. vmin/vmax are conservative: the true value is always inside.

def bounds(u: UOp) -> tuple:
  if (b := u.__dict__.get("_bounds")) is not None: return b
  b = _bounds(u)
  if u.dtype == dtypes.i32 and (b[0] < dtypes.min(u.dtype) or b[1] > dtypes.max(u.dtype)):
    b = (dtypes.min(u.dtype), dtypes.max(u.dtype))   # might wrap around: give up
  u.__dict__["_bounds"] = b
  return b

def vmin(u: UOp): return bounds(u)[0]
def vmax(u: UOp): return bounds(u)[1]

def _bounds(u: UOp) -> tuple:
  d = u.dtype
  full = (dtypes.min(d), dtypes.max(d))
  if u.op is Ops.CONST: return (u.arg, u.arg)
  if u.op is Ops.RANGE: return (0, max(0, vmax(u.src[0]) - 1))
  if d.is_float or u.shape != (): return full
  if u.op is Ops.CAST and u.src[0].dtype == dtypes.bool: return (int(vmin(u.src[0])),
int(vmax(u.src[0])))
  if u.op is Ops.WHERE:
    (c0, c1), (a0, a1), (b0, b1) = bounds(u.src[0]), bounds(u.src[1]), bounds(u.src[2])
    if c0: return (a0, a1)
    if not c1: return (b0, b1)
    return (min(a0, b0), max(a1, b1))
  if u.op not in ALU or len(u.src) != 2: return full
  (a0, a1), (b0, b1) = bounds(u.src[0]), bounds(u.src[1])
  match u.op:
    case Ops.ADD: return (a0 + b0, a1 + b1)
    case Ops.MUL:
      if any(math.isinf(x) for x in (a0, a1, b0, b1)): return full
      ps = [a0 * b0, a0 * b1, a1 * b0, a1 * b1]
      return (min(ps), max(ps))
    case Ops.MAX: return (max(a0, b0), max(a1, b1))
    case Ops.FLOORDIV if b0 == b1 and b0 > 0: return (a0 // b0, a1 // b0)
    case Ops.FLOORMOD if b0 == b1 and b0 > 0:
      return (a0, a1) if 0 <= a0 and a1 < b0 else (0, b0 - 1)   # floor mod by c>0 is always
in [0, c)
    case Ops.CMPLT: return (True, True) if a1 < b0 else (False, False) if a0 >= b1 else
(False, True)
    case Ops.CMPNE: return (True, True) if a1 < b0 or b1 < a0 else (False, False) if a0 ==
```

```
a1 == b0 == b1 else (False, True)
  return full


# ---------------------------------------------------------------------------
# 3. rules

def _fold(x: UOp) -> UOp | None:
  if x.shape != () or not all(s.op is Ops.CONST for s in x.src): return None
  r = exec_alu(x.op, x.dtype, [s.arg for s in x.src])
  return None if r is None else UOp.const(r, x.dtype)

fold = PatternMatcher([(UPat(ALU, name="x"), _fold)])


def _divides(u: UOp, c: int) -> UOp | None:
  """u / c if every term of u is a multiple of c, else None."""
  if u.op is Ops.CONST: return u.const_like(u.arg // c) if u.arg % c == 0 else None
  if u.op is Ops.MUL and u.src[1].op is Ops.CONST:
    return u.src[0] * (u.src[1].arg // c) if u.src[1].arg % c == 0 else None
  if u.op is Ops.ADD:
    if (a := _divides(u.src[0], c)) is not None and (b := _divides(u.src[1], c)) is not
None: return a + b
  return None

def _split_terms(u: UOp) -> list[UOp]:
  return _split_terms(u.src[0]) + _split_terms(u.src[1]) if u.op is Ops.ADD else [u]

def _div_sum(x: UOp, c: UOp, mod: bool) -> UOp | None:
  """(a*c + b) // c == a + b//c  and  (a*c + b) % c == b % c. Always true for *floor*
division."""
  if c.arg <= 0: return None
  terms = _split_terms(x)
  quot, rest = [], []
  for t in terms: (quot if _divides(t, c.arg) is not None else rest).append(t)
  if not quot: return None
  r = sum(rest[1:], rest[0]) if rest else x.const_like(0)
  if mod: return r % c
  return sum((_divides(t, c.arg) for t in quot[1:]), _divides(quot[0], c.arg)) + r // c

def _const_bounds(x: UOp) -> UOp | None:
  # an int/bool expression whose bounds pin it to one value *is* that value
  if x.shape == () and not x.dtype.is_float and x.op in ALU and (b := bounds(x))[0] == b[1]:
    return UOp.const(b[0], x.dtype)
  return None

def _lt_mul(x: UOp, c1: UOp, c2: UOp) -> UOp | None:
  # x*c1 < c2  <=>  x < ceil(c2/c1)  for c1 > 0
  return x < -(-c2.arg // c1.arg) if c1.arg > 0 else None

def _i32(v: int) -> int: return truncate(dtypes.i32, v)   # int32 arithmetic wraps, so
folded constants must too
```

```
x, y, z = UPat.var("x"), UPat.var("y"), UPat.var("z")
c1, c2 = UPat.cvar("c1"), UPat.cvar("c2")
ints = (dtypes.i32, dtypes.bool)
xi = UPat.var("x", dtype=dtypes.i32)

symbolic = fold + PatternMatcher([
  # canonical form: constants on the right of commutative ops
  (UPat((Ops.ADD, Ops.MUL, Ops.MAX, Ops.CMPNE), src=(UPat(Ops.CONST, name="c"),
UPat(name="y")), name="u"),
   lambda u, c, y: u.replace(src=(y, c)) if y.op is not Ops.CONST else None),
  # identities
  (x + 0, lambda x: x), (x + 0.0, lambda x: x),   # x+0.0 is not exact for x=-0.0; we accept
that
  (x * 1, lambda x: x), (x * 1.0, lambda x: x),
  (UPat.var("x", dtype=ints) * 0, lambda x: x.const_like(0)),
  (x // 1, lambda x: x), (x % 1, lambda x: x.const_like(0)),
  (UPat(Ops.MAX, src=(x, x)), lambda x: x),
  (UPat(Ops.WHERE, src=(UPat(), x, x)), lambda x: x),
  (UPat(Ops.WHERE, src=(UPat.cvar("c"), x, y)), lambda c, x, y: x if c.arg else y),
  (UPat(Ops.WHERE, src=(z, UPat(Ops.WHERE, src=(z, x, UPat())), y)), lambda z, x, y:
z.where(x, y)),
  (UPat(Ops.WHERE, src=(z, x, UPat(Ops.WHERE, src=(z, UPat(), y)))), lambda z, x, y:
z.where(x, y)),
  (UPat(Ops.CAST, src=(x,), name="c"), lambda c, x: x if c.dtype == x.dtype else None),
  (UPat(Ops.CMPNE, src=(UPat.var("x", dtype=dtypes.bool), UPat(Ops.CONST, arg=False))),
lambda x: x),
  (UPat((Ops.CMPLT, Ops.CMPNE), src=(xi, xi)), lambda x: UOp.const(False)),
  (xi + xi, lambda x: x * 2),
  (xi + xi * -1, lambda x: x.const_like(0)),
  # reassociate constants outward, so that they can meet and fold
  ((xi + c1) + c2, lambda x, c1, c2: x + _i32(c1.arg + c2.arg)),   # ints only: float
addition isn't associative
  ((xi * c1) * c2, lambda x, c1, c2: x * _i32(c1.arg * c2.arg)),
  ((xi + c1) * c2, lambda x, c1, c2: x * c2 + _i32(c1.arg * c2.arg)),
  # distribute constants over sums so div/mod can see every term: (a + b) * c -> a*c + b*c
  (UPat(Ops.MUL, src=(UPat(Ops.ADD, src=(xi, UPat.var("y")), name="s"), UPat.cvar("c"))),
lambda x, y, s, c: x * c + y * c),
  (UPat(Ops.ADD, src=[xi + c1, UPat.var("y")]), lambda x, c1, y: (x + y) + c1 if y.op is not
Ops.CONST else None),
  (xi * c1 + xi * c2, lambda x, c1, c2: x * _i32(c1.arg + c2.arg)),
  (xi * c1 + xi, lambda x, c1: x * _i32(c1.arg + 1)),
  # division and modulo (floor semantics make these unconditional)
  ((xi // c1) // c2, lambda x, c1, c2: x // (c1.arg * c2.arg) if c1.arg > 0 and c2.arg > 0
and c1.arg * c2.arg < 2**31 else None),
  ((xi % c1) % c2, lambda x, c1, c2: x % c2 if c2.arg > 0 and c1.arg % c2.arg == 0 else
None),
  ((xi % c1) // c2, lambda x, c1, c2: (x // c2) % (c1.arg // c2.arg) if c2.arg > 0 and
c1.arg % c2.arg == 0 else None),
  (UPat(Ops.FLOORDIV, src=(xi, UPat.cvar("c"))), lambda x, c: _div_sum(x, c, mod=False)),
  (UPat(Ops.FLOORMOD, src=(xi, UPat.cvar("c"))), lambda x, c: _div_sum(x, c, mod=True)),
  (UPat(Ops.FLOORDIV, src=(xi, UPat.cvar("c"))), lambda x, c: x.const_like(0) if 0 <=
vmin(x) and vmax(x) < c.arg else None),
  (UPat(Ops.FLOORMOD, src=(xi, UPat.cvar("c"))), lambda x, c: x if 0 <= vmin(x) and vmax(x)
```

```
< c.arg else None),
  # recombine: (x//c)*c + x%c == x
  (UPat(Ops.ADD, src=[(xi // c1) * c1, xi % c1]), lambda x, c1: x),
  (UPat(Ops.ADD, src=[(xi // c1) * c1 + y, xi % c1]), lambda x, c1, y: x + y),
  # comparisons
  ((xi + c1) < c2, lambda x, c1, c2: x < (c2.arg - c1.arg) if -2**31 <= c2.arg - c1.arg <
2**31 else None),
  ((xi * c1) < c2, _lt_mul),
  # bounds say this int expression is a constant
  (UPat(ALU, name="x"), _const_bounds),
])


# ---------------------------------------------------------------------------
# 4. evaluation: STACK, INDEX and CALL

def _elem(s: UOp, i: int) -> UOp: return s.src[i] if s.op is Ops.STACK else s.index(i)

def _push_stack(x: UOp) -> UOp | None:
  # f(STACK(a0, a1), STACK(b0, b1)) -> STACK(f(a0, b0), f(a1, b1))
  if x.shape == () or not any(s.op is Ops.STACK for s in x.src): return None
  return UOp(Ops.STACK, tuple(x.replace(src=tuple(_elem(s, i) for s in x.src)) for i in
range(x.shape[0])))

def _index_stack(s: UOp, i: UOp, rest: tuple[UOp, ...]) -> UOp | None:
  if not 0 <= i.arg < len(s.src): raise IndexError(f"index {i.arg} out of bounds for STACK
of {len(s.src)}")
  return s.src[i.arg].index(*rest) if rest else s.src[i.arg]

def inline_call(call: UOp) -> UOp:
  """Beta reduction: replace each PARAM of the body with its argument.
  Scoped: a nested CALL's body (its src[0]) has its own PARAMs, so we never rewrite inside
it."""
  args = {p: call.src[1 + p.arg.slot] for p in call_params(call.src[0])}
  order, seen, stack = [], set(), [(call.src[0], False)]
  while stack:
    u, done = stack.pop()
    if done: order.append(u); continue
    if u in seen: continue
    seen.add(u)
    stack.append((u, True))
    stack.extend((s, False) for s in (u.src[1:] if u.op is Ops.CALL else u.src))
  new: dict[UOp, UOp] = {}
  for u in order:
    if u in args: new[u] = args[u]; continue
    src = (u.src[0], *(new[s] for s in u.src[1:])) if u.op is Ops.CALL else tuple(new[s] for
s in u.src)
    new[u] = u if src == u.src else u.replace(src=src)
  return new[call.src[0]]

def _literal(u: UOp):
  """A CONST/STACK tree as (flat list of python values, shape), or None if anything is
symbolic."""
```

```
  if u.op is Ops.CONST: return [u.arg], ()
  if u.op is not Ops.STACK: return None
  parts = [_literal(s) for s in u.src]
  if any(p is None for p in parts): return None
  return [v for p in parts for v in p[0]], u.shape

def _from_flat(flat: list, shape: tuple, dtype) -> UOp:
  if not shape: return UOp.const(flat[0], dtype)
  n = len(flat) // shape[0]
  return UOp(Ops.STACK, tuple(_from_flat(flat[i*n:(i+1)*n], shape[1:], dtype) for i in
range(shape[0])))

def _unravel(i: int, shape) -> list[int]:
  out = []
  for s in reversed(shape): out.append(i % s); i //= s
  return out[::-1]

def _ravel(idx, shape) -> int:
  i = 0
  for x, s in zip(idx, shape): i = i * s + x
  return i

def _eval_movement(u: UOp) -> UOp | None:
  """Movement ops and REDUCE on fully known values: just compute them."""
  if (lit := _literal(u.src[0])) is None: return None
  flat, ishape = lit
  oshape, n_out = u.shape, prod(u.shape)
  match u.op:
    case Ops.RESHAPE: out = flat
    case Ops.EXPAND: out = [flat[_ravel([0 if s == 1 else x for x, s in zip(_unravel(i,
oshape), ishape)], ishape)] for i in range(n_out)]
    case Ops.PERMUTE:
      inv = [u.arg.index(k) for k in range(len(ishape))]
      out = [flat[_ravel([_unravel(i, oshape)[inv[k]] for k in range(len(ishape))], ishape)]
for i in range(n_out)]
    case Ops.SHRINK: out = [flat[_ravel([x + b for x, (b, _) in zip(_unravel(i, oshape),
u.arg)], ishape)] for i in range(n_out)]
    case Ops.REDUCE:
      rop, k = u.arg
      inner = prod(ishape[len(ishape) - k:])
      out = []
      for i in range(n_out):
        acc = flat[i * inner]
        for v in flat[i * inner + 1:(i + 1) * inner]: acc = exec_alu(rop, u.dtype, [acc, v])
        out.append(acc)
  return _from_flat(out, oshape, u.dtype)

evaluate = PatternMatcher([
  (UPat({Ops.RESHAPE, Ops.EXPAND, Ops.PERMUTE, Ops.SHRINK, Ops.REDUCE}, name="u"),
_eval_movement),
  (UPat(Ops.INDEX, src=(x,)), lambda x: x),
  (UPat(Ops.INDEX, src=(UPat(Ops.INDEX, name="a"),), allow_any_len=True, name="b"),
   lambda a, b: a.src[0].index(*a.src[1:], *b.src[1:])),
```

```
  (UPat(Ops.INDEX, src=(UPat(Ops.STACK, name="s"), UPat.cvar("i")), allow_any_len=True,
name="u"),
   lambda s, i, u: _index_stack(s, i, u.src[2:])),
  (UPat(ALU, name="x"), _push_stack),
  (UPat(Ops.CALL, name="call"), inline_call),
])


_simplified: dict[UOp, UOp] = {}

def simplify(u: UOp) -> UOp:
  """graph_rewrite(u, symbolic), memoized: rewriting is a pure function of a (hash-consed)
node."""
  if (r := _simplified.get(u)) is not None: return r
  if len(_simplified) > 200_000: _simplified.clear()
  r = _simplified[u] = graph_rewrite(u, symbolic)
  _simplified[r] = r
  return r
def run(u: UOp) -> UOp: return graph_rewrite(u, symbolic + evaluate)

def to_python(u: UOp):
  """A fully evaluated program as nested python lists."""
  if u.op is Ops.CONST: return u.arg
  if u.op is Ops.STACK: return [to_python(s) for s in u.src]
  raise ValueError(f"program did not evaluate to a value, stuck at {u.op}")
```

uopc/uir.py · 178 lines

```python
"""Week 1: the .uir interchange format.

    # gemm_256x256.uir
    %0 = buffer 65536 : dtype=f32 slot=0
    %1 = buffer 65536 : dtype=f32 slot=1
    %2 = reshape %0, (256, 1, 256)
    %3 = reshape %1, (1, 256, 256)
    %4 = mul %2, %3
    %5 = reduce %4 : op=add pop=1

Grammar (one statement per line, `#` starts a comment):

    line     := '%' NAME '=' opname operands? (':' attr*)?
    operands := operand (',' operand)*
    operand  := '%' NAME | literal          literal := int | float | true | false | tuple
    attr     := key '=' value

Integer/float literals in a *source* position become CONSTs (`add %x, 1.0`, `range 10`).
Tuple literals are the arg of movement ops. The last statement is the program.

Two pieces of sugar make hand-written programs readable:
  * binary ops broadcast: size-1 axes are EXPANDed, missing leading axes RESHAPEd in
  * a bare literal adopts the dtype of the other operand (`mul %f32, 2` means 2.0)
"""
from __future__ import annotations
import ast, math, re
from uopc.uop import UOp, Ops, ParamArg, CallArg, RangeArg, AxisType, BINARY, UOpError
from uopc.dtype import dtypes, AddrSpace

POSITIONAL_ARG = {Ops.BUFFER, Ops.ALLOC, Ops.CONST, Ops.RESHAPE, Ops.EXPAND, Ops.PERMUTE,
Ops.SHRINK}
LITERAL_WORDS = {"true": True, "false": False, "inf": math.inf, "-inf": -math.inf, "nan":
math.nan}


class ParseError(Exception): pass


def _split_top(s: str) -> list[str]:
  """Split on commas that are not inside parentheses."""
  out, depth, cur = [], 0, ""
  for ch in s:
    if ch == "," and depth == 0: out.append(cur.strip()); cur = ""; continue
    depth += (ch == "(") - (ch == ")")
    cur += ch
  if cur.strip(): out.append(cur.strip())
  return out


def _literal(tok: str):
  if tok in LITERAL_WORDS: return LITERAL_WORDS[tok]
  try: return ast.literal_eval(tok)
```

```
  except (ValueError, SyntaxError): raise ParseError(f"bad literal {tok!r}") from None


def _attr_value(key: str, v: str):
  if key == "dtype": return dtypes.from_name(v)
  if key == "op": return Ops[v.upper()]
  if key == "kind": return AxisType[v.upper()]
  if key == "space": return AddrSpace[v.upper()]
  if key == "name": return v
  return _literal(v)


def _broadcast(a: UOp, b: UOp) -> tuple[UOp, UOp]:
  if a.shape == b.shape: return a, b
  n = max(len(a.shape), len(b.shape))
  a = a.reshape((1,) * (n - len(a.shape)) + a.shape) if len(a.shape) < n else a
  b = b.reshape((1,) * (n - len(b.shape)) + b.shape) if len(b.shape) < n else b
  out = []
  for x, y in zip(a.shape, b.shape):
    if x != y and 1 not in (x, y): raise ParseError(f"can't broadcast {a.shape} with
{b.shape}")
    out.append(max(x, y))
  return (a if a.shape == tuple(out) else a.expand(out)), (b if b.shape == tuple(out) else
b.expand(out))


def parse(text: str) -> UOp:
  env: dict[str, UOp] = {}
  last = None
  for lineno, raw in enumerate(text.splitlines(), 1):
    line = raw.split("#", 1)[0].strip()
    if not line: continue
    try: last = _parse_line(line, env)
    except (ParseError, UOpError, KeyError, ValueError, TypeError) as e:
      raise ParseError(f"line {lineno}: {raw.strip()!r}: {e}") from None
  if last is None: raise ParseError("empty program")
  return last


def _parse_line(line: str, env: dict[str, UOp]) -> UOp:
  lhs, rhs = (p.strip() for p in line.split("=", 1))
  if not lhs.startswith("%"): raise ParseError("statement must start with %name =")
  body, _, attrs_s = rhs.partition(":")
  opname, _, ops_s = body.strip().partition(" ")
  op = Ops[opname.upper()]
  attrs = {}
  for k, v in re.findall(r"(\w+)=(\([^=]*\)|\S+)", attrs_s):
    attrs[k] = _attr_value(k, v)
  toks = _split_top(ops_s)
  lit = None
  if op in POSITIONAL_ARG:
    if op in (Ops.BUFFER, Ops.ALLOC, Ops.CONST): lit, toks = _literal(toks[0]), toks[1:]
    else: lit, toks = _literal(toks[-1]), toks[:-1]
```

```
  src = []
  for t in toks:
    if t.startswith("%"):
      if t[1:] not in env: raise ParseError(f"undefined {t}")
      src.append(env[t[1:]])
    else: src.append(_literal(t))
  env[lhs[1:]] = u = _build(op, src, lit, attrs)
  return u


def _build(op: Ops, src: list, lit, attrs: dict) -> UOp:
  # bare literals in source position become CONSTs, typed like their neighbours
  # (a WHERE's condition is bool: its branches decide the literal's type)
  like = next((s for s in (src[1:] if op is Ops.WHERE else src) if isinstance(s, UOp)),
None)
  def const(v):
    if op in (Ops.INDEX, Ops.RANGE) or like is None or isinstance(v, bool): return
UOp.const(v)
    return UOp.const(v, like.dtype)
  src = [s if isinstance(s, UOp) else const(s) for s in src]
  match op:
    case Ops.BUFFER | Ops.ALLOC:
      return UOp(op, (), ParamArg(attrs.get("slot", 0), attrs.get("dtype", dtypes.f32),
(lit,), attrs.get("space")))
    case Ops.PARAM: return UOp(op, (), ParamArg(attrs["slot"], attrs.get("dtype",
dtypes.f32), tuple(attrs.get("shape", ()))))
    case Ops.CONST: return UOp.const(lit, attrs.get("dtype"))
    case Ops.CALL: return UOp(op, tuple(src), CallArg(attrs.get("name", "f")))
    case Ops.CAST: return src[0].cast(attrs["dtype"])
    case Ops.RANGE: return UOp(op, tuple(src), RangeArg(attrs.get("idx", 0),
attrs.get("kind", AxisType.LOOP)))
    case Ops.REDUCE: return src[0].reduce(attrs.get("op", Ops.ADD), attrs.get("pop", 1))
    case Ops.RESHAPE | Ops.EXPAND | Ops.PERMUTE: return UOp(op, (src[0],), tuple(lit))
    case Ops.SHRINK: return src[0].shrink(lit)
  if op in BINARY:
    a, b = _broadcast(*src)
    return UOp(op, (a, b))
  if op is Ops.WHERE:
    c, a, b = src
    a, b = _broadcast(a, b); c, a = _broadcast(c, a); c, b = _broadcast(c, b)
    return UOp(op, (c, a, b))
  return UOp(op, tuple(src))


# ---------------------------------------------------------------------------
# printing

def _fmt_lit(v) -> str:
  if isinstance(v, bool): return "true" if v else "false"
  if isinstance(v, float):
    if math.isnan(v): return "nan"
    if math.isinf(v): return "inf" if v > 0 else "-inf"
    return repr(v)
```

```
  if isinstance(v, tuple): return "(" + ", ".join(_fmt_lit(x) for x in v) + ("," if len(v)
== 1 else "") + ")"
  return str(v)


def to_uir(root: UOp) -> str:
  names: dict[UOp, str] = {}
  lines = []
  for u in root.toposort():
    if u.op is Ops.CONST and u is not root: continue   # constants are printed inline
    names[u] = f"%{len(names)}"
    ops = [_fmt_lit(s.arg) if s.op is Ops.CONST else names[s] for s in u.src]
    attrs = []
    a = u.arg
    match u.op:
      case Ops.BUFFER | Ops.ALLOC:
        ops = [str(a.shape[0])]
        attrs = [f"dtype={a.dtype.name}", f"slot={a.slot}"] + ([f"space=
{a.addrspace.name.lower()}"] if a.addrspace else [])
      case Ops.PARAM: attrs = [f"slot={a.slot}", f"dtype={a.dtype.name}"] + ([f"shape=
{_fmt_lit(a.shape)}"] if a.shape else [])
      case Ops.CONST: ops = [_fmt_lit(a)]
      case Ops.CALL: attrs = [f"name={a.name}"]
      case Ops.CAST: attrs = [f"dtype={a.name}"]
      case Ops.RANGE: attrs = [f"idx={a.idx}", f"kind={a.kind.name.lower()}"]
      case Ops.REDUCE: attrs = [f"op={a[0].name.lower()}", f"pop={a[1]}"]
      case Ops.RESHAPE | Ops.EXPAND | Ops.PERMUTE | Ops.SHRINK: ops.append(_fmt_lit(a))
    line = f"{names[u]} = {u.op.name.lower()}"
    if ops: line += " " + ", ".join(ops)
    if attrs: line += " : " + " ".join(attrs)
    lines.append(line)
  return "\n".join(lines) + "\n"
```

uopc/render_c.py · 143 lines

```python
"""Weeks 2-3: rendering a linearized program as a C function.

Each UOp becomes at most one line of C. CONST, INDEX, AFTER and pure ALU ops
used once are rendered inline as expressions; everything else gets a named
variable. BUFFERs become function parameters, sorted by slot.
"""
from __future__ import annotations
import math
from uopc.uop import UOp, Ops, AxisType, ALU
from uopc.dtype import dtypes, DType, AddrSpace
from uopc.linearize import linearize
from uopc.symbolic import vmin

PRELUDE = """#include <math.h>
#include <stdbool.h>
static inline int floordiv_i32(int a, int b) { int q = a / b; return (q * b != a && ((a < 0)
!= (b < 0))) ? q - 1 : q; }
static inline int floormod_i32(int a, int b) { int r = a % b; return (r != 0 && ((r < 0) !=
(b < 0))) ? r + b : r; }
"""

def cconst(v, d: DType) -> str:
  if d == dtypes.bool: return "1" if v else "0"
  if d == dtypes.i32: return str(v) if v != -2**31 else "(-2147483647-1)"
  if math.isnan(v): return "NAN"
  if math.isinf(v): return "INFINITY" if v > 0 else "-INFINITY"
  s = repr(float(v))
  return s + "f" if d == dtypes.f32 else f"(({d.ctype}){s}f)"

def ctype(d: DType) -> str: return d.ctype

def _binop(op: Ops, a: str, b: str, ua: UOp, ub: UOp, d: DType) -> str:
  match op:
    case Ops.ADD: return f"({a}+{b})"
    case Ops.MUL: return f"({a}*{b})"
    case Ops.CMPLT: return f"({a}<{b})"
    case Ops.CMPNE: return f"({a}!={b})"
    case Ops.MAX: return f"(({a}>{b})?{a}:{b})"
    case Ops.FLOORDIV:
      # C division truncates; it equals floor division when both sides are non-negative
      if vmin(ua) >= 0 and ub.op is Ops.CONST and ub.arg > 0: return f"({a}/{b})"
      return f"floordiv_i32({a},{b})"
    case Ops.FLOORMOD:
      if vmin(ua) >= 0 and ub.op is Ops.CONST and ub.arg > 0: return f"({a}%{b})"
      return f"floormod_i32({a},{b})"
  raise NotImplementedError(op)

UNARY_FN = {Ops.EXP2: "exp2", Ops.LOG2: "log2", Ops.SQRT: "sqrt", Ops.SIN: "sin"}

def _unop(op: Ops, a: str, d: DType, arg) -> str:
  if op is Ops.CAST: return f"(({arg.ctype})({a}))"
  if op is Ops.RECIP: return f"(1.0f/{a})" if d == dtypes.f32 else f"(({d.ctype})1.0f/{a})"
```

```
  return f"{UNARY_FN[op]}{'f' if d == dtypes.f32 else ''}({a})"


class CRenderer:
  """Render a SINK of stores into one C function. Subclassed for CUDA in week 7."""
  kernel_prefix = "void"
  buffer_prefix = ""
  shared_prefix = ""
  barrier = None
  restrict = "restrict"

  def render_range(self, u: UOp, name: str, end: str) -> list[str]:
    pragma = ["#pragma omp parallel for"] if u.arg.kind is AxisType.THREAD else []
    return pragma + [f"for (int {name} = 0; {name} < {end}; {name}++) {{"]

  def render(self, sink: UOp, name: str = "kernel") -> str:
    lin = linearize(sink)
    uses: dict[UOp, int] = {}
    for u in lin:
      for s in u.src: uses[s] = uses.get(s, 0) + 1
    r: dict[UOp, str] = {}
    lines: list[str] = []
    depth = 1
    bufs = sorted({u for u in lin if u.op is Ops.BUFFER}, key=lambda u: u.arg.slot)
    counter = 0
    def var(prefix: str) -> str:
      nonlocal counter
      counter += 1
      return f"{prefix}{counter}"
    def line(s: str): lines.append("  " * depth + s)

    for u in lin:
      op = u.op
      if op is Ops.CONST: r[u] = cconst(u.arg, u.dtype)
      elif op is Ops.BUFFER: r[u] = f"data{u.arg.slot}"
      elif op is Ops.ALLOC:
        r[u] = f"{'smem' if u.addrspace is AddrSpace.SHARED else 'acc'}{u.arg.slot}"
        pre = self.shared_prefix if u.addrspace is AddrSpace.SHARED else ""
        if self._scalar(u): line(f"{ctype(u.dtype)} {r[u]};")     # a size-1 accumulator is
just a variable
        else: line(f"{pre}{ctype(u.dtype)} {r[u]}[{u.arg.shape[0]}];")
      elif op is Ops.AFTER: r[u] = r[u.src[0]]
      elif op is Ops.INDEX:
        base = u.src[0]
        while base.op is Ops.AFTER: base = base.src[0]
        r[u] = r[u.src[0]] if base.op is Ops.ALLOC and self._scalar(base) else f"
{r[u.src[0]]}[{r[u.src[1]]}]"
      elif op is Ops.LOAD:
        r[u] = var("val")
        line(f"{ctype(u.dtype)} {r[u]} = {r[u.src[0]]};")
      elif op is Ops.STORE: line(f"{r[u.src[0]]} = {r[u.src[1]]};")
      elif op is Ops.RANGE:
        r[u] = self.range_name(u)
```

```
        for l in self.render_range(u, r[u], r[u.src[0]]): line(l)
        if self.is_loop(u): depth += 1
      elif op is Ops.END:
        for rng in u.src[1:]:
          if self.is_loop(rng):
            depth -= 1
            line("}")
      elif op is Ops.BARRIER: line(self.barrier + ";")
      elif op is Ops.SINK: pass
      elif op is Ops.WHERE:
        expr = f"({r[u.src[0]]}?{r[u.src[1]]}:{r[u.src[2]]})"
        self._assign(u, expr, uses, r, line, var)
      elif op in ALU:
        if len(u.src) == 1: expr = _unop(op, r[u.src[0]], u.dtype, u.arg)
        else: expr = _binop(op, r[u.src[0]], r[u.src[1]], u.src[0], u.src[1], u.dtype)
        self._assign(u, expr, uses, r, line, var)
      elif op is Ops.WMMA: self.render_wmma(u, r, line)
      else: raise NotImplementedError(f"C renderer can't handle {op}")

    params = ", ".join(f"{self.buffer_prefix}{ctype(b.dtype)}* {self.restrict}
data{b.arg.slot}" for b in bufs)
    return self.header(sink, lin) + f"{self.kernel_prefix} {name}({params}) {{\n" +
"\n".join(lines) + "\n}\n"

  def header(self, sink: UOp, lin: list[UOp]) -> str: return PRELUDE

  def _scalar(self, alloc: UOp) -> bool: return alloc.arg.shape == (1,) and alloc.addrspace
is AddrSpace.LOCAL

  def range_name(self, u: UOp) -> str: return f"ridx{u.arg.idx}"

  def is_loop(self, u: UOp) -> bool:
    """On a CPU every range is a for loop -- GPU ranges too, which makes this an emulator
for GPU kernels."""
    return True

  def _assign(self, u, expr, uses, r, line, var):
    if uses.get(u, 0) <= 1: r[u] = expr   # used once: inline it
    else:
      r[u] = var("alu")
      line(f"{ctype(u.dtype)} {r[u]} = {expr};")

  def render_wmma(self, u, r, line): raise NotImplementedError("WMMA needs the CUDA
renderer")


def render_c(sink: UOp, name: str = "kernel") -> str: return CRenderer().render(sink, name)
```

uopc/runtime.py · 70 lines

```python
"""Week 2: the runtime. Compile C with the system compiler, load it, call it.

Compiled shared objects are cached on disk by a hash of (source, flags), so a
program is only compiled once per machine.
"""
from __future__ import annotations
import ctypes, hashlib, os, subprocess, tempfile
from pathlib import Path
import numpy as np
from uopc.dtype import DType, dtypes

CC = os.environ.get("CC", "gcc")
# -fassociative-math lets gcc vectorize reductions (it reorders float adds, like every ML
compiler does).
# NaN/inf semantics are kept. Set CFLAGS to override, e.g. CFLAGS="-O2 -fopenmp" for strict
IEEE order.
CFLAGS = os.environ.get("CFLAGS", "-O3 -march=native -fopenmp -ffp-contract=fast -
fassociative-math "
                                "-fno-signed-zeros -fno-trapping-math").split()
CACHE = Path(os.environ.get("UOPC_CACHE", Path.home() / ".cache" / "uopc"))

NP = {dtypes.f32: np.float32, dtypes.i32: np.int32, dtypes.bool: np.bool_, dtypes.f16:
np.float16}

def np_dtype(d: DType): return NP[d]


class CompileError(Exception): pass


def compile_c(src: str, flags: list[str] | None = None) -> Path:
  flags = CFLAGS if flags is None else flags
  key = hashlib.sha256((src + "\0" + " ".join([CC, *flags])).encode()).hexdigest()[:24]
  CACHE.mkdir(parents=True, exist_ok=True)
  so = CACHE / f"{key}.so"
  if so.exists(): return so
  with tempfile.TemporaryDirectory(dir=CACHE) as td:   # same filesystem, so os.replace is
atomic
    c = Path(td) / "k.c"
    c.write_text(src)
    tmp = Path(td) / "k.so"
    p = subprocess.run([CC, *flags, "-shared", "-fPIC", "-o", str(tmp), str(c), "-lm"],
capture_output=True, text=True)
    if p.returncode != 0:
      numbered = "\n".join(f"{i+1:4d} {l}" for i, l in enumerate(src.splitlines()))
      raise CompileError(f"{CC} failed:\n{p.stderr}\n--- source ---\n{numbered}")
    os.replace(tmp, so)   # atomic: concurrent compiles of the same kernel are fine
  return so


class Program:
  """A compiled C function taking one pointer per buffer slot."""
```

```python
  def __init__(self, src: str, name: str = "kernel", flags: list[str] | None = None):
    self.src, self.name = src, name
    self.lib = ctypes.CDLL(str(compile_c(src, flags)))
    self.fn = getattr(self.lib, name)
    self.fn.restype = None

  def __call__(self, *bufs: np.ndarray):
    for b in bufs:
      if not b.flags.c_contiguous: raise ValueError("buffers must be contiguous")
    self.fn(*(ctypes.c_void_p(b.ctypes.data) for b in bufs))


def run_c(sink, buffers: dict[int, np.ndarray], name: str = "kernel") -> str:
  """Render, compile and run a SINK of stores. buffers maps slot -> numpy array. Returns the
C source."""
  from uopc.render_c import render_c
  from uopc.uop import Ops
  src = render_c(sink, name)
  bufs = sorted({u for u in sink.toposort() if u.op is Ops.BUFFER}, key=lambda u:
u.arg.slot)
  for b in bufs:
    a = buffers[b.arg.slot]
    if a.dtype != np_dtype(b.dtype) or a.size < b.arg.shape[0]:
      raise TypeError(f"buffer slot {b.arg.slot}: kernel wants {b.arg.shape[0]} x
{b.dtype.name}, got {a.size} x {a.dtype}")
  Program(src, name)(*(buffers[b.arg.slot] for b in bufs))
  return src
```

uopc/linearize.py · 128 lines

```python
"""Weeks 2-3: turning a graph into a straight-line list of instructions with loops.

A graph has no order; C does. Linearizing means picking a topological order
that also respects *scope*: the set of loops a node must sit inside.

    scope(RANGE r)       = scope(end) | {r}
    scope(END x, *rs)    = scope(x) - rs
    scope(anything else) = union of scope(srcs)

Loops nest in increasing RangeArg.idx order. If two ranges a < b ever appear in
one scope, b is nested inside a, so *every* node that uses b must also sit inside a.
Closing scopes under "encloses" turns each scope into a path in the loop tree:
a sorted tuple that is exactly the stack of open loops when the node runs.

Then we list-schedule:
  1. emit a ready node whose scope is exactly the open loops
  2. else open a loop that pending nodes need next (sibling loops whose END
     another sibling's nodes read must go first)
  3. else close the innermost loop
ALLOCs are declared in the deepest scope shared by all their users (so each CPU
thread gets its own private accumulator).
"""
from __future__ import annotations
import heapq
from uopc.uop import UOp, Ops
from uopc.dtype import AddrSpace


class LinearizeError(Exception): pass


def scopes(sink: UOp) -> dict[UOp, frozenset[UOp]]:
  sc: dict[UOp, frozenset[UOp]] = {}
  for u in sink.toposort():
    s = frozenset().union(*(sc[x] for x in u.src))
    if u.op is Ops.RANGE: s = s | {u}
    if u.op is Ops.END: s = s - set(u.src[1:])
    sc[u] = s
  return sc


def _key(s) -> tuple[UOp, ...]: return tuple(sorted(s, key=lambda r: r.arg))


def loop_targets(topo: list[UOp], sink: UOp) -> dict[UOp, tuple[UOp, ...]]:
  sc = scopes(sink)
  # enclosing loops of each range: every smaller range it ever shares a scope with
  enc: dict[UOp, set[UOp]] = {}
  for s in set(sc.values()):
    t = _key(s)
    for j, r in enumerate(t): enc.setdefault(r, set()).update(t[:j])
  def close(s):
```

```
    out, todo = set(s), list(s)
    while todo:
      for a in enc.get(todo.pop(), ()):
        if a not in out: out.add(a); todo.append(a)
    return _key(out)
  target = {u: close(sc[u]) for u in topo}
  for u in topo:
    if u.op is Ops.RANGE: target[u] = close(sc[u] - {u} | enc.get(u, set()))
  return target


def linearize(sink: UOp) -> list[UOp]:
  topo = sink.toposort()
  order = {u: i for i, u in enumerate(topo)}
  target = loop_targets(topo, sink)
  users: dict[UOp, list[UOp]] = {u: [] for u in topo}
  for u in topo:
    for s in set(u.src): users[s].append(u)
  for u in topo:
    if u.op is Ops.ALLOC and u.addrspace is AddrSpace.SHARED: target[u] = ()   # shared
memory: whole block, top of kernel
    elif u.op is Ops.ALLOC and users[u]:
      ts = [target[x] for x in users[u]]
      n = 0
      while all(len(t) > n and t[n] is ts[0][n] for t in ts): n += 1
      target[u] = ts[0][:n]

  waiting = {u: len(set(u.src)) for u in topo}
  ready: set[UOp] = set()                          # ready RANGE / END nodes
  heaps: dict[tuple, list[tuple[int, UOp]]] = {}   # ready everything-else, bucketed by
scope
  pending, stack, out = set(topo), [], []

  def make_ready(u: UOp):
    if u.op in (Ops.RANGE, Ops.END): ready.add(u)
    else: heapq.heappush(heaps.setdefault(target[u], []), (order[u], u))

  def emit(u: UOp):
    out.append(u); pending.discard(u); ready.discard(u)
    for x in users[u]:
      waiting[x] -= 1
      if waiting[x] == 0: make_ready(x)

  for u in topo:
    if waiting[u] == 0: make_ready(u)
  while pending:
    cur = tuple(stack)
    # 1. anything that belongs exactly here
    if heaps.get(cur):
      emit(heapq.heappop(heaps[cur])[1]); continue
    # 2. open a loop that pending nodes need next
    need = {u: target[u][len(cur)] for u in pending
            if u.op is not Ops.RANGE and len(target[u]) > len(cur) and target[u][:len(cur)]
```

```
== cur}
    cands = {r for r in need.values() if r in ready and target[r] == cur}
    if cands:
      # if nodes needing loop A read (through an END) the results of sibling loop B, open B
first
      ended: dict[UOp, set[UOp]] = {}
      blocked: set[UOp] = set()
      for u in topo:
        if u not in pending: continue
        e = set(u.src[1:]) & cands if u.op is Ops.END else set()
        for x in u.src:
          if x in pending: e |= ended.get(x, set())
        ended[u] = e
        if u in need and e - {need[u]}: blocked.add(need[u])
      r = min(cands - blocked or cands, key=lambda r: r.arg)
      emit(r); stack.append(r); continue
    # 3. close the innermost loop(s)
    ends = [u for u in ready if u.op is Ops.END and target[u] == cur[:len(cur) - (len(u.src)
- 1)]
            and set(u.src[1:]) == set(cur[len(cur) - (len(u.src) - 1):])]
    if ends:
      e = min(ends, key=order.__getitem__)
      emit(e); del stack[len(stack) - (len(e.src) - 1):]; continue
    stuck = sorted(pending, key=order.__getitem__)[:5]
    raise LinearizeError(f"can't schedule; open loops {[r.arg.idx for r in cur]}, stuck at "
                         f"{[(u.op, [r.arg.idx for r in target[u]]) for u in stuck]}")
  if stack: raise LinearizeError(f"RANGE without END: loops {[r.arg.idx for r in stack]} are
never closed")
  return out
```

uopc/rangeify.py · 201 lines

```python
"""Week 4: rangeify. From shapes to loops.

A shaped graph says *what* each element is. Rangeify says *how to compute one
element*: give every output axis a RANGE, then ask each node "what is your value
at index (i0, i1, ...)?" and push the index down the graph:

    BUFFER    value[i]               -> LOAD(INDEX(buf, i))
    ALU       f(a, b)[idx]           -> f(a[idx], b[idx])
    RESHAPE   x.reshape(s)[idx]      -> x[unravel(ravel(idx, s), x.shape)]
    EXPAND    x.expand(s)[idx]       -> x[0 where x had size 1]
    PERMUTE   x.permute(p)[idx]      -> x[idx permuted back]
    SHRINK    x.shrink(b)[idx]       -> x[idx + begin]
    STACK     stack(a, b)[i, *rest]  -> where(i < 1, a[rest], b[rest])
    REDUCE    sum over the last n    -> new RANGEs, an accumulator, a loop

After rangeify no op has a shape: everything is a scalar in some loop. Symbolic
simplification then cleans up the index arithmetic (ravel . unravel == id).
"""
from __future__ import annotations
from dataclasses import dataclass, field
from uopc.uop import UOp, Ops, ParamArg, AxisType, ALU, MOVEMENT, sink, range_
from uopc.dtype import dtypes, DType
from uopc.symbolic import simplify


def ravel(idx: tuple[UOp, ...], shape: tuple[int, ...]) -> UOp:
  flat = UOp.const(0)
  for i, s in zip(idx, shape): flat = flat * s + i
  return flat

def unravel(flat: UOp, shape: tuple[int, ...]) -> tuple[UOp, ...]:
  out, stride = [], 1
  for s in reversed(shape):
    out.append((flat // stride) % s if s != 1 else UOp.const(0))
    stride *= s
  return tuple(reversed(out))

def identity(op: Ops, d: DType):
  if op is Ops.ADD: return 0.0 if d.is_float else 0
  return -float("inf") if d.is_float else dtypes.min(d)


@dataclass
class Opts:
  """Week 6: how to map a kernel onto the machine. Chosen by uopc.opt, applied during
rangeify."""
  upcast: dict[int, int] = field(default_factory=dict)   # output axis -> unroll factor
(register tile)
  thread: int | None = None                              # output axis run in parallel
(OpenMP)
  unroll: int = 1                                        # unroll factor of the innermost
reduce axis
```

```
  tile: dict[int, int] = field(default_factory=dict)     # output axis -> cache block size
(week 6, loop tiling)
  gpu: bool = False                                      # weeks 7-8: output axes become
GLOBAL (blockIdx) ranges
  local: dict[int, int] = field(default_factory=dict)    # output axis -> threads per block
on that axis (LOCAL)


@dataclass
class Ctx:
  next_range: int = 0
  next_alloc: int = 0
  memo: dict = field(default_factory=dict)
  reduce_ranges: dict = field(default_factory=dict)
  opts: Opts = field(default_factory=Opts)

  def rng(self, size: int, kind: AxisType = AxisType.LOOP) -> UOp:
    self.next_range += 1
    return range_(size, self.next_range - 1, kind)

  def alloc(self, d: DType, n: int = 1) -> UOp:
    self.next_alloc += 1
    return UOp(Ops.ALLOC, (), ParamArg(self.next_alloc - 1, d, (n,)))


def ranges_in(*us: UOp) -> list[UOp]:
  return sorted({r for u in us for r in u.toposort() if r.op is Ops.RANGE}, key=lambda r:
r.arg)


def rangeify(u: UOp, idx: tuple[UOp, ...], ctx: Ctx) -> UOp:
  """The scalar value of u at index idx."""
  key = (u, idx)
  if (r := ctx.memo.get(key)) is not None: return r
  assert len(idx) == len(u.shape), f"{u.op}: {len(idx)} indices for shape {u.shape}"
  op = u.op
  if op is Ops.CONST: r = u
  elif op is Ops.BUFFER: r = u.index(idx[0]).load()
  elif op is Ops.STAGE: r = rangeify(u.src[0], idx, ctx)      # no kernelizer: compute it
inline
  elif op in ALU: r = u.replace(src=tuple(rangeify(s, idx, ctx) for s in u.src))
  elif op is Ops.INDEX:
    iv = tuple(rangeify(i, (), ctx) for i in u.src[1:])
    r = rangeify(u.src[0], iv + idx, ctx)
  elif op is Ops.STACK:
    i, rest = idx[0], idx[1:]
    r = rangeify(u.src[-1], rest, ctx)
    for k in range(len(u.src) - 2, -1, -1):
      r = (i < k + 1).where(rangeify(u.src[k], rest, ctx), r)
  elif op is Ops.RESHAPE:
    # simplify now: ((i*N + j)//N) % N mentions j but *is* i. Dependence analysis
    # (which loops a reduce must sit in) and the memo both need the truth.
    r = rangeify(u.src[0], tuple(simplify(i) for i in unravel(ravel(idx, u.shape),
```

```
u.src[0].shape)), ctx)
  elif op is Ops.EXPAND:
    r = rangeify(u.src[0], tuple(UOp.const(0) if s == 1 and n != 1 else i for i, s, n in
zip(idx, u.src[0].shape, u.shape)), ctx)
  elif op is Ops.PERMUTE:
    inv = [0] * len(idx)
    for k, p in enumerate(u.arg): inv[p] = idx[k]
    r = rangeify(u.src[0], tuple(inv), ctx)
  elif op is Ops.SHRINK: r = rangeify(u.src[0], tuple(i + b if b else i for i, (b, _) in
zip(idx, u.arg)), ctx)
  elif op is Ops.REDUCE: r = _reduce(u, idx, ctx)
  else: raise NotImplementedError(f"rangeify: {op}")
  ctx.memo[key] = r
  return r


def _axis_parts(ctx: Ctx, size: int, factor: int, kind: AxisType) -> tuple[UOp, int]:
  """An axis of `size` as (outer RANGE or CONST 0, unroll factor): index = outer * factor +
c."""
  if factor > 1 and size % factor == 0:
    return (ctx.rng(size // factor, kind) if size > factor else UOp.const(0)), factor
  return (ctx.rng(size, kind) if size > 1 else UOp.const(0)), 1

def _combos(parts: list[tuple[UOp, int]]) -> list[tuple[UOp, ...]]:
  """Every index tuple obtained by fixing the unrolled part of each axis to a constant.
  parts[k] = (base, f): the index of axis k is base * f + c for c in 0..f-1."""
  out = [()]
  for b, f in parts:
    known0 = b.op is Ops.CONST and b.arg == 0
    out = [c + (UOp.const(k) if known0 else (b * f + k if f > 1 else b),) for c in out for k
in range(f)]
  return out


def _reduce(u: UOp, idx: tuple[UOp, ...], ctx: Ctx) -> UOp:
  rop, n = u.arg
  x = u.src[0]
  rshape = x.shape[len(x.shape) - n:]
  # unrolled copies of the same reduce (they differ only by constants in idx) share one loop
  rkey = (u, frozenset(ranges_in(*idx)))
  if (parts := ctx.reduce_ranges.get(rkey)) is None:
    parts = [_axis_parts(ctx, s, ctx.opts.unroll if k == n - 1 else 1, AxisType.REDUCE) for
k, s in enumerate(rshape)]
    ctx.reduce_ranges[rkey] = parts
  real = [r for r, _ in parts if r.op is Ops.RANGE]
  vals = [rangeify(x, idx + c, ctx) for c in _combos(parts)]
  while len(vals) > 1:                                    # unrolled reduce axis: a balanced
tree
    vals = [UOp(rop, (vals[i], vals[i + 1])) if i + 1 < len(vals) else vals[i] for i in
range(0, len(vals), 2)]
  if not real: return vals[0]
  acc = ctx.alloc(u.dtype)
  # the accumulator lives inside the loops the output index uses...
```

```
  acc0 = acc.after(*ranges_in(*idx))
  init = acc0.index(0).store(UOp.const(identity(rop, u.dtype), u.dtype))
  # ...is re-read every iteration of the reduce loops...
  accl = acc0.after(init, *real)
  upd = accl.index(0).store(UOp(rop, (accl.index(0).load(), vals[0])))
  # ...and is read once more after the loops end
  return acc0.after(upd.end(*real)).index(0).load()


def merge_ends(root: UOp) -> UOp:
  """Unrolled copies each END the same shared loop. Merge them into one END over a SINK."""
  while True:
    groups: dict[tuple, list[UOp]] = {}
    for u in root.toposort():
      if u.op is Ops.END: groups.setdefault(u.src[1:], []).append(u)
    dup = next((g for g in groups.values() if len(g) > 1), None)
    if dup is None: return root
    merged = UOp(Ops.END, (UOp(Ops.SINK, tuple(e.src[0] for e in dup)), *dup[0].src[1:]))
    root = root.substitute({e: merged for e in dup})


def _gpu_axis(ctx: Ctx, size: int, local: int, factor: int) -> tuple[UOp, int]:
  """size = blocks * local * factor: index = (gidx * local + lidx) * factor + c."""
  assert size % (local * factor) == 0, f"axis {size} not divisible by local {local} * upcast
{factor}"
  g = ctx.rng(size // (local * factor), AxisType.GLOBAL) if size // (local * factor) > 1
else UOp.const(0)
  if local == 1: return g, factor
  return g * local + ctx.rng(local, AxisType.LOCAL), factor


def lower(root: UOp, out: UOp, ctx: Ctx | None = None) -> UOp:
  """Lower one shaped value into a SINK that stores it, element by element, into BUFFER
`out`."""
  ctx = ctx or Ctx()
  o = ctx.opts
  if o.gpu:
    # each output axis gets a GLOBAL range and (optionally) a LOCAL one; neither becomes a
loop on a GPU
    gl = [_gpu_axis(ctx, s, o.local.get(k, 1), o.upcast.get(k, 1)) for k, s in
enumerate(root.shape)]
    plist = gl
  else:
    # the THREAD axis gets the lowest range index, so it is the outermost loop
    order = ([o.thread] if o.thread is not None else []) + [k for k in
range(len(root.shape)) if k != o.thread]
    tiled = {k: t for k, t in o.tile.items() if 1 < t < root.shape[k] and root.shape[k] % t
== 0 and t % o.upcast.get(k, 1) == 0}
    parts: dict[int, tuple[UOp, int]] = {}
    # tiled axes: first every block loop (outer), then the loops inside a block. A block of
outputs reuses
    # the same rows of each operand, so they stay in cache while the block is computed
    outer = {k: ctx.rng(root.shape[k] // t, AxisType.THREAD if k == o.thread else
```

```
AxisType.LOOP) for k, t in tiled.items()}
    for k in order:
      f = o.upcast.get(k, 1)
      if k in tiled:
        inner = tiled[k] // f
        parts[k] = (outer[k] * inner + ctx.rng(inner) if inner > 1 else outer[k], f)
      else:
        parts[k] = _axis_parts(ctx, root.shape[k], f, AxisType.THREAD if k == o.thread else
AxisType.LOOP)
    plist = [parts[k] for k in range(len(root.shape))]
  stores = []
  for idx in _combos(plist):
    val = rangeify(root, idx, ctx)
    stores.append(out.index(ravel(idx, root.shape) if root.shape else
UOp.const(0)).store(val))
  loops = ranges_in(*(b for b, _ in plist))
  body = stores[0] if len(stores) == 1 else UOp(Ops.SINK, tuple(stores))
  return simplify(merge_ends(sink(body.end(*loops))))
```

uopc/composite.py · 92 lines

```python
"""Week 4: composite movement ops, built only from RESHAPE/EXPAND/PERMUTE/SHRINK/STACK.

PAD and FLIP are not primitives in this course. Building them out of primitives is
a good exercise in thinking about index maps, and rangeify + symbolic turn the
result back into tight code (PAD becomes a bounds-checked WHERE).
"""
from __future__ import annotations
from uopc.uop import UOp, Ops, prod


def zeros_like(x: UOp) -> UOp:
  z = UOp.const(0, x.dtype)
  return z.reshape((1,) * len(x.shape)).expand(x.shape) if x.shape else z


def move_axis(x: UOp, src: int, dst: int) -> UOp:
  order = [i for i in range(len(x.shape)) if i != src]
  order.insert(dst, src)
  return x.permute(order) if order != list(range(len(x.shape))) else x


def concat2(a: UOp, b: UOp, axis: int) -> UOp:
  """Concatenate two same-shaped tensors along `axis`: STACK, move the new axis next to
`axis`, merge."""
  assert a.shape == b.shape
  s = UOp(Ops.STACK, (a, b))                        # (2, *shape)
  s = move_axis(s, 0, axis)                         # (..., 2, n, ...)
  shape = list(a.shape)
  shape[axis] *= 2
  return s.reshape(tuple(shape))


def _shrink_axis(x: UOp, axis: int, b: int, e: int) -> UOp:
  arg = [(0, s) for s in x.shape]
  arg[axis] = (b, e)
  return x.shrink(arg) if (b, e) != (0, x.shape[axis]) else x


def pad_axis(x: UOp, axis: int, before: int, after: int) -> UOp:
  """Zero-pad one axis by doubling with zeros and shrinking back."""
  while after > 0:
    n = x.shape[axis]
    x = _shrink_axis(concat2(x, zeros_like(x), axis), axis, 0, n + min(after, n))
    after -= min(after, n)
  while before > 0:
    n = x.shape[axis]
    x = _shrink_axis(concat2(zeros_like(x), x, axis), axis, n - min(before, n), 2 * n)
    before -= min(before, n)
  return x


def pad(x: UOp, arg: tuple[tuple[int, int], ...]) -> UOp:
```

```python
  for axis, (b, a) in enumerate(arg): x = pad_axis(x, axis, b, a)
  return x


def anti_eye(n: int, dtype) -> UOp:
  """M[i, j] = (i + j == n - 1), from one-hot + the expand/reshape/shrink "diagonal trick".

  u is one-hot at n-1, length 2n-1. Expand u to (n+1, 2n-1), flatten, keep n*2n
  elements, view as (n, 2n): element (i, j) is u[(2n*i + j) mod (2n-1)] = u[i + j].
  """
  if n == 1: return UOp.const(1, dtype).reshape((1, 1))
  one = UOp.const(1, dtype).reshape((1,))
  u = pad_axis(one, 0, n - 1, n - 1)                              # (2n-1,)
  m = u.reshape((1, 2 * n - 1)).expand((n + 1, 2 * n - 1)).reshape(((n + 1) * (2 * n - 1),))
  m = m.shrink(((0, n * 2 * n),)).reshape((n, 2 * n))
  return m.shrink(((0, n), (0, n)))


def flip_axis(x: UOp, axis: int) -> UOp:
  """Reverse one axis: y[i] = sum_j x[j] * [i + j == n-1]. O(n^2), but only primitives."""
  n = x.shape[axis]
  if n == 1: return x
  xl = move_axis(x, axis, len(x.shape) - 1)                      # (..., n)
  lead = xl.shape[:-1]
  xe = xl.reshape(lead + (1, n)).expand(lead + (n, n))
  m = anti_eye(n, x.dtype).reshape((1,) * len(lead) + (n, n)).expand(lead + (n, n))
  y = (xe * m).reduce(Ops.ADD, 1)                                 # (..., n)
  return move_axis(y, len(x.shape) - 1, axis)


def sum_to(g: UOp, shape: tuple[int, ...]) -> UOp:
  """Reverse of EXPAND: sum g over the axes where `shape` has 1 and g doesn't."""
  axes = [i for i, (s, t) in enumerate(zip(g.shape, shape)) if s != t]
  if not axes: return g
  keep = [i for i in range(len(shape)) if i not in axes]
  return g.permute(keep + axes).reduce(Ops.ADD, len(axes)).reshape(shape)


def is_contiguous_buffer(u: UOp) -> bool:
  while u.op is Ops.RESHAPE: u = u.src[0]
  return u.op is Ops.BUFFER and prod(u.shape) == u.arg.shape[0]
```

uopc/kernelize.py · 220 lines

```python
"""Week 5: kernel fission and fusion.

Rangeify can lower *any* shape graph into one giant loop nest, but that is often
terrible: a REDUCE consumed through an EXPAND would be recomputed for every
element it is broadcast to ((A@B)@C would recompute A@B for every output). So we cut the
graph into
kernels at *materialization points*:

  * every root (the tensors the user asked for)
  * every STAGE (the user, or a rule, asked for a temp buffer)
  * every REDUCE, unless fuse_reduce is on (then reductions nest, see flash attention)

Everything between materialization points is fused into one kernel. A kernel
reads BUFFERs and writes exactly one. The schedule is the list of kernels in
dependency order. Whole schedules are cached on the *structure* of the graph,
so a training step only pays for Python-side compilation once.
"""
from __future__ import annotations
import os, time
from dataclasses import dataclass
import numpy as np
from uopc.uop import UOp, Ops, ParamArg, prod, _argkey
from uopc.dtype import DType
from uopc.rangeify import lower
from uopc.composite import is_contiguous_buffer
from uopc.runtime import Program, np_dtype
from uopc.rewrite import UPat, PatternMatcher, graph_rewrite

DEBUG = int(os.environ.get("DEBUG", "0"))


# ---------------------------------------------------------------------------
# buffers: a BUFFER UOp owns a numpy array. When the UOp dies, so does the memory.

_next_slot = 0

def new_buffer(dtype: DType, size: int, data: np.ndarray | None = None) -> UOp:
  global _next_slot
  _next_slot += 1
  u = UOp(Ops.BUFFER, (), ParamArg(_next_slot, dtype, (max(size, 1),)))
  if data is not None: u.__dict__["np"] = np.ascontiguousarray(data.reshape(-1),
dtype=np_dtype(dtype))
  return u

def buffer_data(u: UOp) -> np.ndarray:
  if "np" not in u.__dict__: u.__dict__["np"] = np.zeros(u.arg.shape[0], np_dtype(u.dtype))
  return u.__dict__["np"]

def base_buffer(u: UOp) -> UOp:
  while u.op is Ops.RESHAPE: u = u.src[0]
  return u
```

```python
# ---------------------------------------------------------------------------
# the schedule

@dataclass
class Kernel:
  name: str
  src: str
  refs: list[tuple[str, int]]      # one per local slot: ("in", i) | ("tmp", j) | ("out", r)
  program: Program | None = None
  flops: int = 0

@dataclass
class Plan:
  kernels: list[Kernel]
  temps: list[tuple[DType, int]]
  outs: list[tuple[str, int] | tuple[DType, int]]   # per root: ("in", i) alias, or (dtype,
size) fresh
  temp_arrays: list | None = None   # allocated on first run, then reused: no page faults on
every step


def _materialize_points(roots: list[UOp], fuse_reduce: bool) -> list[UOp]:
  topo = UOp(Ops.SINK, tuple(roots)).toposort()
  m = []
  for u in topo:
    if u.op is Ops.SINK: continue
    if u.op is Ops.STAGE or (u.op is Ops.REDUCE and not fuse_reduce) or (u in roots and not
is_contiguous_buffer(u)):
      m.append(u)
  return m


# Tensor.detach() wraps a node in STAGE(arg="detach"): autodiff stops there, execution
ignores it
strip_detach = PatternMatcher([(UPat(Ops.STAGE, arg="detach", src=(UPat.var("x"),)), lambda
x: x)])


def stage_broadcast_compute(roots: list[UOp], factor: int = 4) -> list[UOp]:
  """Fusing is only free if nothing gets recomputed. A computed tensor that is EXPANDed by
  `factor` or more (e.g. a LayerNorm output feeding a matmul) would be recomputed for every
  copy, so we materialize it first. This is the fusion/recompute trade-off, as one rule."""
  from uopc.uop import MOVEMENT, ALU
  todo: set[UOp] = set()
  for u in UOp(Ops.SINK, tuple(roots)).toposort():
    if u.op is not Ops.EXPAND or prod(u.shape) < factor * prod(u.src[0].shape): continue
    base = u.src[0]
    while base.op in MOVEMENT: base = base.src[0]
    if base.op in ALU and base.op is not Ops.CAST and base.shape and _fused_work(base) >=
4096: todo.add(base)
  return UOp.wrap(roots, todo, lambda n: UOp(Ops.STAGE, (n,))) if todo else roots
```

```python
def _fused_work(u: UOp) -> int:
  """ALU ops * elements that recomputing u would cost (stopping at materialized nodes)."""
  from uopc.uop import ALU
  n, seen, stack = 0, set(), [u]
  while stack:
    x = stack.pop()
    if x in seen or x.op in (Ops.BUFFER, Ops.REDUCE, Ops.STAGE): continue
    seen.add(x)
    if x.op in ALU: n += 1
    stack.extend(x.src)
  return n * prod(u.shape)


def _kernel_name(root: UOp) -> str:
  red = any(u.op is Ops.REDUCE for u in root.toposort())
  return ("r_" if red else "e_") + "_".join(map(str, root.shape or (1,)))


def make_plan(roots: list[UOp], fuse_reduce: bool = False, opt=None) -> tuple[Plan,
list[UOp]]:
  """Kernelize. Returns the plan and the canonical list of input BUFFERs it refers to as
("in", i)."""
  inputs = [u for u in UOp(Ops.SINK, tuple(roots)).toposort() if u.op is Ops.BUFFER]
  in_idx = {u: i for i, u in enumerate(inputs)}
  roots = [graph_rewrite(r, strip_detach) for r in roots]
  # fuse_reduce asks for "never write the big intermediate", so don't stage broadcasts
either
  if not fuse_reduce: roots = stage_broadcast_compute(roots)
  if opt is not None: roots = opt.layout(roots)
  points = _materialize_points(roots, fuse_reduce)
  # every materialization point gets a placeholder buffer; consumers read it through a
reshape
  ref: dict[UOp, tuple[str, int]] = {}
  temps: list[tuple[DType, int]] = []
  outs: list = []
  place: dict[UOp, UOp] = {}
  for k, u in enumerate(points):
    if u in roots: continue
    ref[u] = ("tmp", len(temps))
    temps.append((u.dtype, max(1, prod(u.shape))))
  for r, u in enumerate(roots):
    if is_contiguous_buffer(u): outs.append(("in", in_idx[base_buffer(u)]))
    else:
      ref[u] = ("out", r)
      outs.append((u.dtype, max(1, prod(u.shape))))
  for u in points:
    ph = UOp(Ops.BUFFER, (), ParamArg(-1 - len(place), u.dtype, (max(1, prod(u.shape)),)))
# negative slots: placeholders
    place[u] = ph.reshape(u.shape) if u.shape != ph.shape else ph
  ph_ref = {base_buffer(place[u]): ref[u] for u in points}

  kernels = []
```

```
  for u in points:
    body = u.src[0] if u.op is Ops.STAGE else u
    body = body.substitute({p: v for p, v in place.items() if p is not u})
    # renumber buffers 0..n-1 locally (output last) so identical kernels render identical C
    bufs = [b for b in body.toposort() if b.op is Ops.BUFFER]
    local = {b: UOp(Ops.BUFFER, (), ParamArg(i, b.dtype, b.arg.shape)) for i, b in
enumerate(bufs)}
    out = UOp(Ops.BUFFER, (), ParamArg(len(bufs), u.dtype, (max(1, prod(u.shape)),)))
    body = body.substitute(local)
    kname = _kernel_name(body)
    if opt is not None: ast = opt.lower(body, out)
    else: ast = lower(body, out)
    from uopc.render_c import render_c
    src = render_c(ast, kname)
    refs = [ph_ref[b] if b.arg.slot < 0 else ("in", in_idx[b]) for b in bufs] + [ref[u]]
    # simplification can delete a read (x*0 -> 0): the C signature then has fewer
parameters,
    # so bind exactly the slots that survived, or every later argument shifts by one
    alive = sorted({b.arg.slot for b in ast.toposort() if b.op is Ops.BUFFER})
    refs = [refs[s] for s in alive]
    kernels.append(Kernel(kname, src, refs, flops=_flops(body)))
  return Plan(kernels, temps, outs), inputs


def _flops(u: UOp) -> int:
  """Rough FLOP count of a shaped kernel body: one per ALU element computed."""
  from uopc.uop import ALU
  return sum(prod(x.shape or (1,)) for x in u.toposort() if x.op in ALU) + \
         sum(prod(x.src[0].shape) for x in u.toposort() if x.op is Ops.REDUCE)


# ---------------------------------------------------------------------------
# structural cache ("JIT"): same graph shape -> same plan, whatever the buffers hold

_plans: dict[tuple, Plan] = {}

def _structure_key(roots: list[UOp]) -> tuple:
  ids: dict[UOp, int] = {}
  key = []
  for u in UOp(Ops.SINK, tuple(roots)).toposort():
    ids[u] = len(ids)
    arg = ("buf", u.dtype.name, u.arg.shape) if u.op is Ops.BUFFER else _argkey(u.arg)
    key.append((u.op, arg, tuple(ids[s] for s in u.src)))
  return tuple(key)


def realize(roots: list[UOp], fuse_reduce: bool = False, opt=None) -> list[UOp]:
  """Compute each root. Returns, per root, a BUFFER UOp holding the result."""
  uniq = list(dict.fromkeys(roots))
  t0 = time.perf_counter()
  key = (_structure_key(uniq), fuse_reduce, opt)
  inputs = [u for u in UOp(Ops.SINK, tuple(uniq)).toposort() if u.op is Ops.BUFFER]
  if (plan := _plans.get(key)) is None:
```

```
    plan, _ = make_plan(uniq, fuse_reduce, opt)
    for k in plan.kernels:
      k.program = Program(k.src, k.name)
      if DEBUG >= 4: print(k.src)
    _plans[key] = plan
  t1 = time.perf_counter()
  if plan.temp_arrays is None: plan.temp_arrays = [np.empty(n, np_dtype(d)) for d, n in
plan.temps]
  temps = plan.temp_arrays
  outs = [inputs[o[1]] if o[0] == "in" else new_buffer(o[0], o[1]) for o in plan.outs]
  def get(rf):
    kind, i = rf
    return buffer_data(inputs[i]) if kind == "in" else temps[i] if kind == "tmp" else
buffer_data(outs[i])
  for k in plan.kernels:
    ts = time.perf_counter()
    k.program(*(get(r) for r in k.refs))
    if DEBUG >= 2:
      dt = time.perf_counter() - ts
      extra = ""
      if DEBUG >= 3:   # run again: the second run has warm caches and no first-touch page
faults
        ts = time.perf_counter(); k.program(*(get(r) for r in k.refs)); extra = f"  (again:
{(time.perf_counter()-ts)*1e3:.3f} ms)"
      print(f"  {k.name:30s} {dt*1e3:8.3f} ms  {k.flops/max(dt, 1e-9)/1e9:7.2f}
GFLOP/s{extra}")
  if DEBUG >= 1: print(f"realize: {len(plan.kernels)} kernels, schedule {(t1-t0)*1e3:.1f}
ms, run {(time.perf_counter()-t1)*1e3:.1f} ms")
  by_root = dict(zip(uniq, outs))
  return [by_root[r] for r in roots]
```

uopc/autodiff.py · 82 lines

```python
"""Week 5: reverse-mode automatic differentiation on shape graphs.

Autodiff is just another graph transformation: given the graph for `loss` and
some target nodes, build the graph for d(loss)/d(target). It works on pure
shape graphs only -- no STORE, CALL, AFTER, RANGE or END -- which is why we do
it *before* rangeify.

Each rule maps (node, grad of node) -> grad of each source (None = no gradient).
"""
from __future__ import annotations
import math
from uopc.uop import UOp, Ops
from uopc.composite import sum_to, pad, zeros_like

LN2 = math.log(2)


def _full(like: UOp, v: float) -> UOp:
  c = UOp.const(v, like.dtype)
  return c.reshape((1,) * len(like.shape)).expand(like.shape) if like.shape else c

def _rule(u: UOp, g: UOp) -> tuple[UOp | None, ...]:
  s = u.src
  match u.op:
    case Ops.ADD: return (g, g)
    case Ops.MUL: return (g * s[1], g * s[0])
    case Ops.MAX:
      a, b = s
      half = g * _full(g, 0.5)
      zero = zeros_like(g)
      return ((b < a).where(g, (a < b).where(zero, half)), (a < b).where(g, (b <
a).where(zero, half)))
    case Ops.RECIP: return (-(g * u * u),)
    case Ops.EXP2: return (g * u * _full(u, LN2),)
    case Ops.LOG2: return (g * s[0].recip() * _full(u, 1 / LN2),)
    case Ops.SQRT: return (g * u.recip() * _full(u, 0.5),)
    case Ops.SIN: return (g * UOp(Ops.SIN, (s[0] + _full(u, math.pi / 2),)),)
    case Ops.CAST: return (g.cast(s[0].dtype) if s[0].dtype.is_float else None,)
    case Ops.WHERE: return (None, s[0].where(g, zeros_like(g)), s[0].where(zeros_like(g),
g))
    case Ops.CMPLT | Ops.CMPNE | Ops.FLOORDIV | Ops.FLOORMOD: return (None,) * len(s)
    case Ops.RESHAPE: return (g.reshape(s[0].shape),)
    case Ops.EXPAND: return (sum_to(g, s[0].shape),)
    case Ops.PERMUTE:
      inv = [0] * len(u.arg)
      for i, p in enumerate(u.arg): inv[p] = i
      return (g.permute(inv),)
    case Ops.SHRINK: return (pad(g, tuple((b, n - e) for (b, e), n in zip(u.arg,
s[0].shape))),)
    case Ops.STACK: return tuple(g.index(i) for i in range(len(s)))
    case Ops.STAGE: return (None,) if u.arg == "detach" else (g,)
    case Ops.REDUCE:
```

```
      rop, n = u.arg
      x = s[0]
      ge = g.reshape(u.shape + (1,) * n).expand(x.shape)
      if rop is Ops.ADD: return (ge,)
      mx = u.reshape(u.shape + (1,) * n).expand(x.shape)
      mask = x.eq(mx).cast(x.dtype)                    # ties share the gradient
      cnt = mask.reduce(Ops.ADD, n).reshape(u.shape + (1,) * n).expand(x.shape)
      return (ge * mask * cnt.recip(),)
    case Ops.INDEX:
      x, idx = s[0], s[1:]
      if not all(i.op is Ops.CONST for i in idx): raise NotImplementedError("gradient of a
data-dependent INDEX (use a one-hot matmul)")
      gx = g
      for i, n in zip(reversed(idx), reversed(x.shape[:len(idx)])):
        z = zeros_like(gx)
        gx = UOp(Ops.STACK, tuple(gx if k == i.arg else z for k in range(n)))
      return (gx,) + (None,) * len(idx)
    case Ops.CONST | Ops.BUFFER | Ops.PARAM: return ()
  raise NotImplementedError(f"no gradient rule for {u.op} (autodiff only works on pure shape
graphs)")


def gradient(root: UOp, targets: list[UOp]) -> list[UOp]:
  topo = root.toposort()
  # only walk nodes that lie on a path to some target
  on_path = set(targets)
  for u in topo:
    if any(x in on_path for x in u.src): on_path.add(u)
  grads: dict[UOp, UOp] = {root: _full(root, 1.0)}
  for u in reversed(topo):
    if u not in grads or u not in on_path or u in targets and not u.src: continue
    for x, gx in zip(u.src, _rule(u, grads[u])):
      if gx is None or x not in on_path: continue
      grads[x] = grads[x] + gx if x in grads else gx
  return [grads.get(t, zeros_like(t)) for t in targets]
```

uopc/tensor.py · 333 lines

```python
"""Week 5: a lazy Tensor frontend.

A Tensor is a thin wrapper around a shaped UOp graph. Nothing runs until you call
.realize() / .numpy(): then the graph is kernelized, rangeified, rendered to C,
compiled, and executed. Every op below is a few lines of graph building on top of
the course's primitives -- there is no other math library underneath.
"""
from __future__ import annotations
import math, os
import numpy as np
from uopc.uop import UOp, Ops, prod
from uopc.dtype import DType, dtypes
from uopc.autodiff import gradient
from uopc.composite import pad as _pad, flip_axis
from uopc.kernelize import new_buffer, buffer_data, base_buffer, realize as _realize
from uopc.runtime import np_dtype

LOG2E, LN2 = 1 / math.log(2), math.log(2)


def _broadcast_shape(*shapes):
  n = max(len(s) for s in shapes)
  shapes = [(1,) * (n - len(s)) + tuple(s) for s in shapes]
  out = []
  for dims in zip(*shapes):
    d = max(dims)
    if any(x not in (1, d) for x in dims): raise ValueError(f"can't broadcast {shapes}")
    out.append(d)
  return tuple(out)


class Tensor:
  rng = np.random.default_rng(0)
  training = False
  fuse_reduce = False
  opt = None   # week 6: kernel optimizer (set below)

  def __init__(self, data, dtype: DType | None = None, requires_grad: bool = False):
    if isinstance(data, UOp): self.uop = data
    else:
      arr = np.asarray(data)
      if dtype is None: dtype = dtypes.f32 if arr.dtype.kind == "f" else dtypes.bool if
arr.dtype.kind == "b" else dtypes.i32
      arr = arr.astype(np_dtype(dtype))
      buf = new_buffer(dtype, arr.size, arr)
      self.uop = buf.reshape(arr.shape) if arr.shape != buf.shape else buf
    self.requires_grad = requires_grad
    self.grad: Tensor | None = None

  # ---- basics ------------------------------------------------------------
  @property
  def shape(self) -> tuple[int, ...]: return self.uop.shape
```

```
  @property
  def dtype(self) -> DType: return self.uop.dtype
  @property
  def ndim(self) -> int: return len(self.shape)
  def numel(self) -> int: return prod(self.shape)
  def __repr__(self): return f"<Tensor {self.shape} {self.dtype.name}{' grad' if
self.requires_grad else ''}>"

  def realize(self) -> Tensor:
    Tensor.realize_all(self)
    return self

  @staticmethod
  def realize_all(*ts: Tensor):
    todo = [t for t in ts if t.uop.op is not Ops.BUFFER and not (t.uop.op is Ops.RESHAPE and
base_buffer(t.uop).op is Ops.BUFFER and prod(t.shape) == base_buffer(t.uop).arg.shape[0])]
    if not todo: return
    bufs = _realize([t.uop for t in todo], Tensor.fuse_reduce, Tensor.opt)
    for t, b in zip(todo, bufs):
      t._graph = t.uop   # keep the graph: loss.item() before loss.backward() must still
have gradients
      t.uop = b.reshape(t.shape) if b.shape != t.shape else b

  def numpy(self) -> np.ndarray:
    self.realize()
    return buffer_data(base_buffer(self.uop)).reshape(self.shape).copy()

  def item(self): return self.numpy().item()
  def detach(self) -> Tensor:
    """Same value, but gradients stop here. (Kernelize strips the marker, so it costs
nothing to run.)"""
    return Tensor(UOp(Ops.STAGE, (self.uop,), "detach"))

  def assign(self, x: Tensor) -> Tensor:
    """Make this tensor (e.g. a parameter) hold x's value. Functional: rebinds to a new
buffer."""
    assert x.shape == self.shape and x.dtype == self.dtype, f"assign {x.shape} to
{self.shape}"
    self.uop = x.uop
    return self

  # ---- creation ------------------------------------------------------------
  @staticmethod
  def full(shape, v, dtype: DType = dtypes.f32) -> Tensor:
    c = UOp.const(v, dtype)
    return Tensor(c.reshape((1,) * len(shape)).expand(tuple(shape)) if shape else c)
  @staticmethod
  def zeros(*shape, dtype=dtypes.f32) -> Tensor: return Tensor.full(shape, 0, dtype)
  @staticmethod
  def ones(*shape, dtype=dtypes.f32) -> Tensor: return Tensor.full(shape, 1, dtype)
  @staticmethod
  def randn(*shape, requires_grad=False) -> Tensor:
    return Tensor(Tensor.rng.standard_normal(shape).astype(np.float32),
```

```
requires_grad=requires_grad)
  @staticmethod
  def uniform(*shape, low=-1.0, high=1.0, requires_grad=False) -> Tensor:
    return Tensor(Tensor.rng.uniform(low, high, shape).astype(np.float32),
requires_grad=requires_grad)
  @staticmethod
  def kaiming(*shape, requires_grad=True) -> Tensor:
    bound = 1 / math.sqrt(prod(shape[1:]) if len(shape) > 1 else shape[0])
    return Tensor.uniform(*shape, low=-bound, high=bound, requires_grad=requires_grad)
  @staticmethod
  def arange(n: int) -> Tensor: return Tensor(np.arange(n, dtype=np.int32))
  @staticmethod
  def manual_seed(s: int): Tensor.rng = np.random.default_rng(s)

  # ---- elementwise -----------------------------------------------------------
  def _const(self, x) -> Tensor: return x if isinstance(x, Tensor) else Tensor.full((), x,
self.dtype)

  def broadcast_to(self, shape) -> Tensor:
    shape = tuple(shape)
    if self.shape == shape: return self
    u = self.uop
    if len(u.shape) < len(shape): u = u.reshape((1,) * (len(shape) - len(u.shape)) +
u.shape)
    return Tensor(u.expand(shape) if u.shape != shape else u)

  def _binop(self, op: Ops, x, reverse=False) -> Tensor:
    # numpy-style promotion for python scalars: int tensor (op) 0.5 is a float computation
    if isinstance(x, float) and not self.dtype.is_float: return self.float()._binop(op, x,
reverse)
    x = self._const(x)
    a, b = (x, self) if reverse else (self, x)
    shape = _broadcast_shape(a.shape, b.shape)
    return Tensor(UOp(op, (a.broadcast_to(shape).uop, b.broadcast_to(shape).uop)))

  def __add__(self, x): return self._binop(Ops.ADD, x)
  def __radd__(self, x): return self._binop(Ops.ADD, x, True)
  def __mul__(self, x): return self._binop(Ops.MUL, x)
  def __rmul__(self, x): return self._binop(Ops.MUL, x, True)
  def __neg__(self): return self * -1
  def __sub__(self, x): return self + -self._const(x)
  def __rsub__(self, x): return self._const(x) + -self
  def __truediv__(self, x): return self * self._const(x).reciprocal()
  def __rtruediv__(self, x): return self._const(x) * self.reciprocal()
  def __lt__(self, x): return self._binop(Ops.CMPLT, x)
  def __gt__(self, x): return self._binop(Ops.CMPLT, x, True)
  def __ge__(self, x): return (self < x).logical_not()
  def __le__(self, x): return (self > x).logical_not()
  def eq(self, x): return self._binop(Ops.CMPNE, x).logical_not()
  def ne(self, x): return self._binop(Ops.CMPNE, x)
  def logical_not(self): return self._binop(Ops.CMPNE, True)
  def maximum(self, x): return self._binop(Ops.MAX, x)
  def minimum(self, x): return -((-self).maximum(-self._const(x)))
```

```
  def where(self, a, b) -> Tensor:
    a = a if isinstance(a, Tensor) else Tensor.full((), a)
    b = b if isinstance(b, Tensor) else Tensor.full((), b, a.dtype)
    shape = _broadcast_shape(self.shape, a.shape, b.shape)
    return Tensor(UOp(Ops.WHERE, (self.broadcast_to(shape).uop, a.broadcast_to(shape).uop,
b.broadcast_to(shape).uop)))
  def cast(self, d: DType) -> Tensor: return Tensor(self.uop.cast(d))
  def float(self) -> Tensor: return self.cast(dtypes.f32)

  def reciprocal(self): return Tensor(self.uop.recip())
  def exp2(self): return Tensor(UOp(Ops.EXP2, (self.uop,)))
  def log2(self): return Tensor(UOp(Ops.LOG2, (self.uop,)))
  def exp(self): return (self * LOG2E).exp2()
  def log(self): return self.log2() * LN2
  def sqrt(self): return Tensor(UOp(Ops.SQRT, (self.uop,)))
  def rsqrt(self): return self.sqrt().reciprocal()
  def sin(self): return Tensor(UOp(Ops.SIN, (self.uop,)))
  def cos(self): return (self + math.pi / 2).sin()
  def square(self): return self * self
  def abs(self): return self.maximum(-self)
  def relu(self): return (self > 0).where(self, 0.0)
  def sigmoid(self): return ((-self).exp() + 1).reciprocal()
  def tanh(self): return (self * 2).sigmoid() * 2 - 1
  def gelu(self): return 0.5 * self * (1 + (math.sqrt(2 / math.pi) * (self + 0.044715 * self
* self * self)).tanh())
  def silu(self): return self * self.sigmoid()

  # ---- movement ----------------------------------------------------------------
  def reshape(self, *shape) -> Tensor:
    shape = tuple(shape[0]) if len(shape) == 1 and isinstance(shape[0], (tuple, list)) else
shape
    if -1 in shape:
      known = prod(s for s in shape if s != -1)
      shape = tuple(self.numel() // known if s == -1 else s for s in shape)
    return self if shape == self.shape else Tensor(self.uop.reshape(shape))
  def permute(self, *order) -> Tensor:
    order = tuple(order[0]) if len(order) == 1 and isinstance(order[0], (tuple, list)) else
order
    order = tuple(o % self.ndim for o in order)
    return self if order == tuple(range(self.ndim)) else Tensor(self.uop.permute(order))
  def transpose(self, a=-1, b=-2) -> Tensor:
    order = list(range(self.ndim))
    order[a], order[b] = order[b], order[a]
    return self.permute(order)
  @property
  def T(self) -> Tensor: return self.transpose()
  def expand(self, *shape) -> Tensor:
    shape = tuple(shape[0]) if len(shape) == 1 and isinstance(shape[0], (tuple, list)) else
shape
    return self.broadcast_to(tuple(s if s != -1 else o for s, o in zip(shape, self.shape)))
  def flatten(self, start=0) -> Tensor: return self.reshape(self.shape[:start] + (-1,))
  def unsqueeze(self, d) -> Tensor:
    d = d % (self.ndim + 1)
```

```
    return self.reshape(self.shape[:d] + (1,) + self.shape[d:])
  def shrink(self, arg) -> Tensor:
    arg = tuple((0, s) if a is None else a for a, s in zip(arg, self.shape))
    return self if all(a == (0, s) for a, s in zip(arg, self.shape)) else
Tensor(self.uop.shrink(arg))
  def pad(self, arg) -> Tensor: return Tensor(_pad(self.uop, tuple(arg)))
  def flip(self, axis: int) -> Tensor: return Tensor(flip_axis(self.uop, axis % self.ndim))
  def __getitem__(self, idx) -> Tensor:
    """Ints and step-1 slices only."""
    idx = idx if isinstance(idx, tuple) else (idx,)
    arg, squeeze = [], []
    for d, i in enumerate(idx):
      n = self.shape[d]
      if isinstance(i, int):
        i = i % n
        arg.append((i, i + 1)); squeeze.append(d)
      else:
        b, e, step = i.indices(n)
        assert step == 1, "only step-1 slices"
        arg.append((b, max(b, e)))
    arg += [(0, s) for s in self.shape[len(arg):]]
    t = self.shrink(arg)
    return t.reshape(tuple(s for d, s in enumerate(t.shape) if d not in squeeze))
  @staticmethod
  def cat(ts: list[Tensor], dim: int = 0) -> Tensor:
    """Concatenate via pad + add: each piece is padded into place, then summed."""
    dim = dim % ts[0].ndim
    total = sum(t.shape[dim] for t in ts)
    out, off = None, 0
    for t in ts:
      p = [(0, 0)] * t.ndim
      p[dim] = (off, total - off - t.shape[dim])
      piece = t.pad(p)
      out = piece if out is None else out + piece
      off += t.shape[dim]
    return out
  def stack_with(self, *others) -> Tensor: return Tensor(UOp(Ops.STACK, (self.uop, *(o.uop
for o in others))))

  # ---- reductions -------------------------------------------------------------
  def _reduce(self, op: Ops, axis=None, keepdim=False) -> Tensor:
    axes = list(range(self.ndim)) if axis is None else [a % self.ndim for a in (axis if
isinstance(axis, (tuple, list)) else [axis])]
    keep = [i for i in range(self.ndim) if i not in axes]
    t = self.permute(keep + axes)
    r = Tensor(t.uop.reduce(op, len(axes)))
    if keepdim: r = r.reshape(tuple(1 if i in axes else s for i, s in
enumerate(self.shape)))
    return r
  def sum(self, axis=None, keepdim=False): return self._reduce(Ops.ADD, axis, keepdim)
  def max(self, axis=None, keepdim=False): return self._reduce(Ops.MAX, axis, keepdim)
  def min(self, axis=None, keepdim=False): return -((-self).max(axis, keepdim))
  def mean(self, axis=None, keepdim=False):
```

```
    s = (self if self.dtype.is_float else self.float()).sum(axis, keepdim)
    return s * (s.numel() / self.numel())
  def var(self, axis=None, keepdim=False):
    return (self - self.mean(axis, keepdim=True)).square().mean(axis, keepdim)
  def softmax(self, axis=-1):
    e = (self - self.max(axis, keepdim=True)).exp()
    return e / e.sum(axis, keepdim=True)
  def log_softmax(self, axis=-1):
    m = self - self.max(axis, keepdim=True)
    return m - m.exp().sum(axis, keepdim=True).log()
  def logsumexp(self, axis=-1):
    m = self.max(axis, keepdim=True)
    return (self - m).exp().sum(axis, keepdim=True).log() + m
  def argmax(self, axis=-1):
    """Index of the first maximum, as i32."""
    axis = axis % self.ndim
    n = self.shape[axis]
    idx = Tensor.arange(n).reshape((n,) + (1,) * (self.ndim - axis - 1))
    is_max = self.eq(self.max(axis, keepdim=True))
    return is_max.where(n - idx, 0).max(axis) * -1 + n

  # ---- linear algebra -------------------------------------------------------
  def matmul(self, w: Tensor) -> Tensor:
    """(..., M, K) @ (..., K, N): broadcast to (..., M, N, K), multiply, reduce K."""
    x = self if self.ndim > 1 else self.reshape(1, -1)
    wt = w.reshape(-1, 1) if w.ndim == 1 else w
    wt = wt.transpose()                                  # (..., N, K)
    x = x.reshape(x.shape[:-1] + (1, x.shape[-1]))       # (..., M, 1, K)
    wt = wt.reshape(wt.shape[:-2] + (1,) + wt.shape[-2:])  # (..., 1, N, K)
    out = (x * wt).sum(-1)
    if self.ndim == 1: out = out.reshape(out.shape[:-2] + out.shape[-1:])
    if w.ndim == 1: out = out.reshape(out.shape[:-1])
    return out
  __matmul__ = matmul
  def linear(self, w: Tensor, b: Tensor | None = None) -> Tensor:
    out = self @ w
    return out + b if b is not None else out

  def conv2d(self, w: Tensor, b: Tensor | None = None, padding: int = 0, stride: int = 1) ->
Tensor:
    """x (N, C, H, W), w (O, C, KH, KW). im2col from shrinks, then a matmul."""
    x = self.pad(((0, 0), (0, 0), (padding, padding), (padding, padding))) if padding else
self
    N, C, H, W = x.shape
    O, _, KH, KW = w.shape
    OH, OW = (H - KH) // stride + 1, (W - KW) // stride + 1
    cols = []
    for ky in range(KH):
      for kx in range(KW):
        win = x.shrink(((0, N), (0, C), (ky, ky + stride * (OH - 1) + 1), (kx, kx + stride *
(OW - 1) + 1)))
        if stride > 1: win = win._stride2d(stride, OH, OW)
        cols.append(win)                                  # (N, C, OH, OW) each
```

```
    patches = cols[0].stack_with(*cols[1:]) if len(cols) > 1 else cols[0].unsqueeze(0)   # (KH*KW, N, C, OH, OW)
    patches = patches.permute(1, 3, 4, 2, 0).reshape(N, OH, OW, C * KH * KW)
    out = patches @ w.reshape(O, C * KH * KW).T              # (N, OH, OW, O)
    out = out.permute(0, 3, 1, 2)
    return out + b.reshape(1, O, 1, 1) if b is not None else out

  def _stride2d(self, s: int, OH: int, OW: int) -> Tensor:
    N, C, H, W = self.shape
    t = self.pad(((0, 0), (0, 0), (0, OH * s - H), (0, OW * s - W)))
    return t.reshape(N, C, OH, s, OW, s)[:, :, :, 0, :, 0]

  def max_pool2d(self, k: int = 2) -> Tensor:
    N, C, H, W = self.shape
    return self[:, :, :H // k * k, :W // k * k].reshape(N, C, H // k, k, W // k, k).max((3, 5))

  # ---- losses ----------------------------------------------------------------
  def one_hot(self, n: int) -> Tensor:
    return self.unsqueeze(-1).eq(Tensor.arange(n)).float()
  def cross_entropy(self, labels: Tensor) -> Tensor:
    """self: (B, C) logits, labels: (B,) i32."""
    return -(self.log_softmax(-1) * labels.one_hot(self.shape[-1])).sum(-1).mean()

  # ---- autodiff --------------------------------------------------------------
  def backward(self, params: list[Tensor]) -> Tensor:
    """Set p.grad = d(self)/d(p) for each p. Gradients are lazy Tensors like everything else."""
    assert self.shape == (), "backward needs a scalar"
    root = getattr(self, "_graph", self.uop)
    for p, g in zip(params, gradient(root, [p.uop for p in params])):
      p.grad = Tensor(g) if p.grad is None else p.grad + Tensor(g)
    return self

  # ---- staging ---------------------------------------------------------------
  def contiguous(self) -> Tensor:
    """Ask for a materialization point: the kernelizer will write this into a temp buffer."""
    return Tensor(UOp(Ops.STAGE, (self.uop,)))


if os.environ.get("OPT", "1") != "0":
  from uopc.opt import HandOpt, SearchOpt
  Tensor.opt = SearchOpt() if os.environ.get("OPT") == "2" else HandOpt()
```

uopc/nn.py · 100 lines

```python
"""Week 5: layers and optimizers, written against the Tensor API only."""
from __future__ import annotations
import math
from uopc.tensor import Tensor


class Linear:
  def __init__(self, n_in: int, n_out: int, bias: bool = True):
    bound = 1 / math.sqrt(n_in)
    self.weight = Tensor.uniform(n_in, n_out, low=-bound, high=bound, requires_grad=True)
    self.bias = Tensor.zeros(n_out).realize() if bias else None
    if self.bias is not None: self.bias.requires_grad = True
  def __call__(self, x: Tensor) -> Tensor: return x.linear(self.weight, self.bias)


class Conv2d:
  def __init__(self, c_in: int, c_out: int, k: int, padding: int = 0, stride: int = 1):
    bound = 1 / math.sqrt(c_in * k * k)
    self.weight = Tensor.uniform(c_out, c_in, k, k, low=-bound, high=bound,
requires_grad=True)
    self.bias = Tensor.zeros(c_out).realize()
    self.bias.requires_grad = True
    self.padding, self.stride = padding, stride
  def __call__(self, x: Tensor) -> Tensor: return x.conv2d(self.weight, self.bias,
self.padding, self.stride)


class LayerNorm:
  def __init__(self, n: int, eps: float = 1e-5):
    self.weight, self.bias, self.eps = Tensor.ones(n).realize(), Tensor.zeros(n).realize(),
eps
    self.weight.requires_grad = self.bias.requires_grad = True
  def __call__(self, x: Tensor) -> Tensor:
    mu = x.mean(-1, keepdim=True)
    xc = x - mu
    return xc * (xc.square().mean(-1, keepdim=True) + self.eps).rsqrt() * self.weight +
self.bias


class Embedding:
  """A one-hot matmul: differentiable without a scatter op."""
  def __init__(self, vocab: int, dim: int):
    self.vocab, self.weight = vocab, Tensor.randn(vocab, dim) * 0.02
    self.weight = self.weight.realize(); self.weight.requires_grad = True
  def __call__(self, idx: Tensor) -> Tensor: return idx.one_hot(self.vocab) @ self.weight


def parameters(obj) -> list[Tensor]:
  """Every Tensor with requires_grad reachable through attributes, lists and dicts."""
  out, seen = [], set()
  def walk(o):
    if id(o) in seen: return
```

```
    seen.add(id(o))
    if isinstance(o, Tensor):
      if o.requires_grad: out.append(o)
    elif isinstance(o, (list, tuple)): [walk(x) for x in o]
    elif isinstance(o, dict): [walk(x) for x in o.values()]
    elif hasattr(o, "__dict__"): [walk(x) for x in vars(o).values()]
  walk(obj)
  return out


class Optimizer:
  def __init__(self, params: list[Tensor]): self.params = params
  def zero_grad(self):
    for p in self.params: p.grad = None


class SGD(Optimizer):
  def __init__(self, params, lr=0.01, momentum=0.0):
    super().__init__(params)
    self.lr, self.momentum = lr, momentum
    self.v = [Tensor.zeros(*p.shape).realize() for p in params] if momentum else []
  def step(self, *extra: Tensor):
    """Update every parameter. Extra tensors (e.g. the loss) are realized in the same
batch."""
    news = []
    for i, p in enumerate(self.params):
      g = p.grad
      if self.momentum:
        self.v[i].assign(self.v[i] * self.momentum + g); g = self.v[i]
      news.append(p.assign(p.detach() - g * self.lr))
    Tensor.realize_all(*news, *self.v, *extra)


class AdamW(Optimizer):
  def __init__(self, params, lr=1e-3, b1=0.9, b2=0.999, eps=1e-8, wd=0.0):
    super().__init__(params)
    self.lr, self.b1, self.b2, self.eps, self.wd, self.t = lr, b1, b2, eps, wd, 0
    self.m = [Tensor.zeros(*p.shape).realize() for p in params]
    self.v = [Tensor.zeros(*p.shape).realize() for p in params]
  def step(self, *extra: Tensor):
    self.t += 1
    # bias corrections are python floats baked into the graph as constants: each step's
    # graph differs only in two CONSTs, so we pass them as tensors to keep the JIT cache
warm
    c1 = Tensor([self.lr / (1 - self.b1 ** self.t)])
    c2 = Tensor([1 / (1 - self.b2 ** self.t)])
    news = []
    for p, m, v in zip(self.params, self.m, self.v):
      g = p.grad
      m.assign(m * self.b1 + g * (1 - self.b1))
      v.assign(v * self.b2 + g.square() * (1 - self.b2))
      upd = m * c1.reshape(*([1] * p.ndim)) / ((v * c2.reshape(*([1] * p.ndim))).sqrt() +
self.eps)
```

```
        news.append(p.assign(p.detach() * (1 - self.lr * self.wd) - upd))
    Tensor.realize_all(*news, *self.m, *self.v, *extra)
```

uopc/opt.py · 163 lines

```python
"""Week 6: making kernels fast on a CPU.

Three knobs, all applied while rangeifying (see rangeify.Opts):

  UPCAST  unroll output axes into a register tile. A 4x8 tile of a GEMM keeps 32
          accumulators in registers and reuses each loaded A value 8 times and each
          B value 4 times: arithmetic intensity goes from 1/2 to ~2.7 FLOP/load.
  UNROLL  unroll the innermost reduce axis; the loop body becomes a balanced tree.
  THREAD  run the outermost output loop on every core with OpenMP.
  TILE    split output axes into cache blocks (Opts.tile). Only search_opt tries it: with
the layout
          pass making the reduce axis contiguous, it measured no faster on the reference
machine.

`hand_opt` picks them with simple rules. `search_opt` times candidates and keeps
the fastest (a tiny BEAM search).
"""
from __future__ import annotations
import itertools, os, time
import numpy as np
from uopc.uop import UOp, Ops, ParamArg, ALU, MOVEMENT, prod
from uopc.rangeify import lower, rangeify, Ctx, Opts
from uopc.symbolic import simplify
from uopc.rewrite import UPat, PatternMatcher, graph_rewrite

NTHREADS = os.cpu_count() or 1


# ---------------------------------------------------------------------------
# layout: make the innermost reduce axis contiguous in memory

def _views(x: UOp):
  """The movement-op chains feeding x through ALU ops: (top of chain, materialized base)."""
  seen, stack = set(), [x]
  while stack:
    u = stack.pop()
    if u in seen: continue
    seen.add(u)
    if u.op in ALU: stack.extend(u.src)
    elif u.op in MOVEMENT:
      base = u
      while base.op in MOVEMENT: base = base.src[0]
      if base.op in (Ops.BUFFER, Ops.REDUCE, Ops.STAGE): yield u, base

def _last_axis_stride(view: UOp, base: UOp) -> int | None:
  """Distance in memory between view[..., k] and view[..., k+1], or None if not affine-
looking."""
  if view.shape[-1] == 1: return 0
  fake = UOp(Ops.BUFFER, (), ParamArg(-999, base.dtype, (prod(base.shape),)))
  v = view.substitute({base: fake.reshape(base.shape) if base.shape != fake.shape else
fake})
  ctx = Ctx()
```

```
  rngs = tuple(ctx.rng(s) for s in v.shape)
  val = rangeify(v, rngs, ctx)
  if val.op is not Ops.LOAD or val.src[0].op is not Ops.INDEX: return None
  e = val.src[0].src[1]
  at = lambda k: simplify(e.substitute({r: UOp.const(k if i == len(rngs) - 1 else 0) for i,
r in enumerate(rngs)}))
  e0, e1 = at(0), at(1)
  return e1.arg - e0.arg if e0.op is Ops.CONST and e1.op is Ops.CONST else None

# movement ops commute with elementwise ops: permute(a * b) == permute(a) * permute(b).
# Pushing them down to the leaves makes every view chain end at real data.
push_views = PatternMatcher([
  (UPat(MOVEMENT, src=(UPat(ALU, name="a"),), name="m"),
   lambda m, a: a.replace(src=tuple(m.replace(src=(s,)) for s in a.src))),
])

def layout_pass(roots: list[UOp]) -> list[UOp]:
  """Before broadcasting, copy operands whose reduce axis is strided into a contiguous temp
(a STAGE).

  The copy costs O(size of the operand); the reduce it feeds is usually much bigger.
  This is what turns `A @ B` (B read down its columns) into a dot product of rows.
  """
  roots = [graph_rewrite(r, push_views) for r in roots]
  todo: set[UOp] = set()
  for r in UOp(Ops.SINK, tuple(roots)).toposort():
    if r.op is not Ops.REDUCE: continue
    for view, base in _views(r.src[0]):
      s = _last_axis_stride(view, base)
      if s in (0, 1, None): continue
      c = view
      while c.op is Ops.EXPAND: c = c.src[0]                    # stage before the broadcast
      # only stage operands that get broadcast into the reduce: copying them is cheap
      if prod(c.shape) > prod(base.shape) or prod(c.shape) >= prod(r.src[0].shape) or c in
todo: continue
      staged = UOp(Ops.STAGE, (c,))
      if _last_axis_stride(view.substitute({c: staged}), staged) == 1: todo.add(c)
  return UOp.wrap(roots, todo, lambda n: UOp(Ops.STAGE, (n,))) if todo else roots


def _reduce_size(body: UOp) -> int:
  return max((prod(u.src[0].shape[len(u.src[0].shape) - u.arg[1]:]) for u in body.toposort()
if u.op is Ops.REDUCE), default=1)


def hand_opt_choice(body: UOp) -> Opts:
  shape, rsize = body.shape, _reduce_size(body)
  o = Opts()
  if rsize > 1:
    # register tile: 4x4 on the last two output axes. With AVX2's 16 vector registers, 16
accumulators plus
    # the loaded operands don't quite fit (gcc spills a few), but measured it beats 2x8, 4x8
and 8x8 here
```

```
    budget = 16
    for axis, want in [(len(shape) - 1, 4), (len(shape) - 2, 4)]:
      if axis < 0: continue
      for f in (want, 4, 2):
        if f <= budget and shape[axis] % f == 0 and shape[axis] >= f and f > 1:
          o.upcast[axis] = f; budget //= f; break
    if not o.upcast and rsize % 4 == 0 and rsize < 64: o.unroll = 4
  # threads: only when there is enough work to amortize waking them up
  work = prod(shape or (1,)) * rsize
  if work >= 1 << 16:
    for axis, s in enumerate(shape):
      if s // o.upcast.get(axis, 1) >= 2:
        o.thread = axis; break
  return o


def hand_opt(body: UOp, out: UOp) -> UOp:
  return lower(body, out, Ctx(opts=hand_opt_choice(body)))


def candidate_opts(body: UOp) -> list[Opts]:
  shape, rsize = body.shape, _reduce_size(body)
  outs = [a for a, s in enumerate(shape) if s > 1]
  cands = [Opts(), hand_opt_choice(body)]
  for a, b in itertools.product(outs[-2:], repeat=2):
    for fa, fb in [(4, 4), (8, 4), (4, 8), (8, 8), (16, 2), (2, 16)]:
      up = {a: fa} if a == b else {a: fa, b: fb}
      if all(shape[k] % f == 0 for k, f in up.items()):
        for u in (1, 4) if rsize % 4 == 0 else (1,):
          cands.append(Opts(dict(up), hand_opt_choice(body).thread, u))
  # cache blocking of the last two output axes (helps when operand rows don't fit in cache)
  base = hand_opt_choice(body)
  for t in (32, 64, 128):
    tile = {a: t for a in outs[-2:] if shape[a] % t == 0}
    if tile: cands.append(Opts(dict(base.upcast), base.thread, base.unroll, tile=tile))
  return cands


def search_opt(body: UOp, out: UOp, inputs: list[np.ndarray] | None = None, reps: int = 3) -
> UOp:
  """Time every candidate on random data, keep the fastest."""
  from uopc.render_c import render_c
  from uopc.runtime import Program, np_dtype
  bufs = sorted({u for u in body.toposort() if u.op is Ops.BUFFER}, key=lambda u:
u.arg.slot)
  data = inputs or [np.random.randn(b.arg.shape[0]).astype(np_dtype(b.dtype)) for b in bufs]
  res = np.zeros(out.arg.shape[0], np_dtype(out.dtype))
  best, best_t = None, float("inf")
  seen = set()
  for o in candidate_opts(body):
    key = (tuple(sorted(o.upcast.items())), o.thread, o.unroll,
tuple(sorted(o.tile.items())))
    if key in seen: continue
```

```
    seen.add(key)
    ast = lower(body, out, Ctx(opts=o))
    prg = Program(render_c(ast, "k"), "k")
    ts = []
    for _ in range(reps):
      t0 = time.perf_counter(); prg(*data, res); ts.append(time.perf_counter() - t0)
    if min(ts) < best_t: best, best_t = ast, min(ts)
  return best


class HandOpt:
  """The default optimizer kernelize uses: layout pass + hand-coded opts."""
  layout = staticmethod(layout_pass)
  lower = staticmethod(hand_opt)

class SearchOpt(HandOpt):
  """Like HandOpt, but times candidate opts for every new kernel (slow to compile, fast to
run)."""
  lower = staticmethod(search_opt)
```

uopc/gpu.py · 181 lines

```python
"""Weeks 7-8: GPUs.

A GPU kernel is the body of a loop nest whose outer loops the hardware runs for
you: GLOBAL ranges become blockIdx, LOCAL ranges become threadIdx. Everything
else (reduce loops, upcasts, accumulators) is exactly what we already have.

  * CUDARenderer   -- renders the same linearized UOps as CUDA C
  * gpu_opt        -- picks blocks/threads/upcasts for a kernel
  * emulate        -- renders a GPU kernel with the *C* renderer, where GLOBAL and
                      LOCAL ranges are ordinary loops. Same index math, runs on a CPU,
                      so every GPU kernel is testable without a GPU.
  * smem_gemm_ir   -- a tiled GEMM written by hand in the loop IR, using SHARED memory
                      and BARRIER: the core trick of every fast GPU GEMM
  * wmma_gemm      -- tensor cores: pattern-match a GEMM and emit a WMMA kernel
"""
from __future__ import annotations
from uopc.uop import UOp, Ops, ParamArg, AxisType, prod, sink, range_
from uopc.dtype import dtypes, AddrSpace
from uopc.render_c import CRenderer
from uopc.rangeify import lower, Ctx, Opts

CUDA_PRELUDE = """#define INFINITY (__int_as_float(0x7f800000))
#define NAN (__int_as_float(0x7fffffff))
__device__ inline int floordiv_i32(int a, int b) { int q = a / b; return (q * b != a && ((a
< 0) != (b < 0))) ? q - 1 : q; }
__device__ inline int floormod_i32(int a, int b) { int r = a % b; return (r != 0 && ((r < 0)
!= (b < 0))) ? r + b : r; }
"""


class CUDARenderer(CRenderer):
  buffer_prefix = ""
  shared_prefix = "__shared__ "
  barrier = "__syncthreads()"
  restrict = "__restrict__"

  def render(self, sink: UOp, name: str = "kernel") -> str:
    rngs = [u for u in sink.toposort() if u.op is Ops.RANGE]
    # the last (fastest varying) axis gets .x, so neighbouring threads touch neighbouring
memory
    self.glob = [u for u in sorted(rngs, key=lambda r: r.arg, reverse=True) if u.arg.kind is
AxisType.GLOBAL]
    self.loc = [u for u in sorted(rngs, key=lambda r: r.arg, reverse=True) if u.arg.kind is
AxisType.LOCAL]
    assert len(self.glob) <= 3 and len(self.loc) <= 3, "fold axes: CUDA has 3 grid and 3
block dimensions"
    self.kernel_prefix = f'extern "C" __global__ void __launch_bounds__({max(1,
prod(launch_dims(sink)[1]))})'
    # dtypes.f16.ctype is gcc's _Float16; CUDA spells it `half` (from cuda_fp16.h)
    return super().render(sink, name).replace("_Float16", "half")

  def header(self, sink, lin) -> str:
```

```
    return ("#include <cuda_fp16.h>\n" if any(u.dtype == dtypes.f16 for u in lin) else "") +
CUDA_PRELUDE

  def is_loop(self, u: UOp) -> bool: return u.arg.kind not in (AxisType.GLOBAL,
AxisType.LOCAL)

  def range_name(self, u: UOp) -> str:
    if u.arg.kind is AxisType.GLOBAL: return f"gidx{self.glob.index(u)}"
    if u.arg.kind is AxisType.LOCAL: return f"lidx{self.loc.index(u)}"
    return super().range_name(u)

  def render_range(self, u: UOp, name: str, end: str) -> list[str]:
    if u.arg.kind is AxisType.GLOBAL: return [f"int {name} = blockIdx.
{'xyz'[self.glob.index(u)]}; /* {end} */"]
    if u.arg.kind is AxisType.LOCAL: return [f"int {name} = threadIdx.
{'xyz'[self.loc.index(u)]}; /* {end} */"]
    return [f"for (int {name} = 0; {name} < {end}; {name}++) {{"]


def launch_dims(sink: UOp) -> tuple[tuple[int, ...], tuple[int, ...]]:
  """(grid, block) sizes from the GLOBAL and LOCAL ranges, in x, y, z order."""
  rngs = sorted((u for u in sink.toposort() if u.op is Ops.RANGE), key=lambda r: r.arg,
reverse=True)
  g = tuple(r.src[0].arg for r in rngs if r.arg.kind is AxisType.GLOBAL)
  l = tuple(r.src[0].arg for r in rngs if r.arg.kind is AxisType.LOCAL)
  return (g + (1,) * (3 - len(g)), l + (1,) * (3 - len(l)))


def render_cuda(sink: UOp, name: str = "kernel") -> str: return CUDARenderer().render(sink,
name)


# ---------------------------------------------------------------------------
# picking a launch configuration

def gpu_opt_choice(body: UOp, max_threads: int = 256) -> Opts:
  """Up to 3 output axes on the grid. Threads on the last axes (coalesced stores), 2x2
upcast for reduces."""
  shape = body.shape
  o = Opts(gpu=True)
  reduces = any(u.op is Ops.REDUCE for u in body.toposort())
  axes = [k for k, s in enumerate(shape) if s > 1][-3:]
  if len(axes) < len(shape) - shape.count(1):
    raise NotImplementedError("more than 3 non-trivial output axes: reshape first")
  budget = max_threads
  for k in reversed(axes):
    if reduces and shape[k] % 2 == 0 and shape[k] >= 32: o.upcast[k] = 2
    rest = shape[k] // o.upcast.get(k, 1)
    t = 1
    while t * 2 <= min(budget, 16 if len(axes) > 1 else budget) and rest % (t * 2) == 0: t
*= 2
    if t > 1: o.local[k] = t; budget //= t
  return o
```

```
def gpu_lower(body: UOp, out: UOp, opts: Opts | None = None) -> UOp:
  return lower(body, out, Ctx(opts=opts or gpu_opt_choice(body)))


def emulate(sink: UOp, name: str = "kernel") -> str:
  """The C source of a GPU kernel where blockIdx/threadIdx are loops. Run it with
runtime.Program."""
  if any(u.op is Ops.BARRIER for u in sink.toposort()):
    raise NotImplementedError("emulate: kernels with BARRIER need all threads to run in
lockstep; "
                              "running threads one after another (what loops do) would be
wrong")
  return CRenderer().render(sink, name)


# ---------------------------------------------------------------------------
# shared memory: a tiled GEMM in the loop IR

def smem_gemm_ir(N: int, T: int = 16) -> UOp:
  """C = A @ B, N x N, with T x T thread blocks and T x T tiles staged in shared memory.

  Each block computes one T x T tile of C. For each of the N/T steps along K, the
  block cooperatively copies a T x T tile of A and of B into shared memory (one
  element per thread), waits at a barrier, multiplies the tiles out of shared
  memory, and waits again before the next copy overwrites them. Global memory
  traffic drops by a factor of T.
  """
  f32 = dtypes.f32
  A, B, C = (UOp(Ops.BUFFER, (), ParamArg(s, f32, (N * N,))) for s in range(3))
  sA = UOp(Ops.ALLOC, (), ParamArg(0, f32, (T * T,), AddrSpace.SHARED))
  sB = UOp(Ops.ALLOC, (), ParamArg(1, f32, (T * T,), AddrSpace.SHARED))
  acc = UOp(Ops.ALLOC, (), ParamArg(2, f32, (1,)))
  by, bx = range_(N // T, 0, AxisType.GLOBAL), range_(N // T, 1, AxisType.GLOBAL)
  ty, tx = range_(T, 2, AxisType.LOCAL), range_(T, 3, AxisType.LOCAL)
  row, col = by * T + ty, bx * T + tx
  acc0 = acc.after(by, bx, ty, tx)
  init = acc0.index(0).store(UOp.const(0.0))
  kt = range_(N // T, 4)                                    # step along K, one tile at a
time
  ld_a = sA.after(kt).index(ty * T + tx).store(A.index(row * N + kt * T + tx).load())
  ld_b = sB.after(kt).index(ty * T + tx).store(B.index((kt * T + ty) * N + col).load())
  sync1 = UOp(Ops.BARRIER, (ld_a, ld_b))
  k = range_(T, 5)
  acc1 = acc0.after(init, kt, k)
  prod_ = sA.after(sync1).index(ty * T + k).load() * sB.after(sync1).index(k * T +
tx).load()
  inner = acc1.index(0).store(acc1.index(0).load() + prod_).end(k)
  sync2 = UOp(Ops.BARRIER, (inner,))                        # don't overwrite tiles others
still read
  loop = sync2.end(kt)
  out = C.index(row * N + col).store(acc0.after(loop).index(0).load())
```

```
  return sink(out.end(by, bx, ty, tx))


# ---------------------------------------------------------------------------
# tensor cores: pattern-match a GEMM, emit a WMMA kernel

def match_gemm(body: UOp) -> tuple[int, int, int] | None:
  """If body is (M,N) = sum_K A(M,K) * B(K,N) over contiguous f16/f32 buffers, return (M, N,
K)."""
  if body.op is not Ops.REDUCE or body.arg != (Ops.ADD, 1) or body.src[0].op is not Ops.MUL:
return None
  if len(body.src[0].shape) != 3 or not body.dtype.is_float: return None
  M, N, K = body.src[0].shape
  # the two operands must be broadcast along *different* axes: A over N (axis 1), B over M
(axis 0).
  # Without this check, x*x reduced over its last axis (a sum of squares) "is" a GEMM.
  a, b = body.src[0].src
  def bcast_axes(x: UOp) -> set[int]:
    return {i for i, (s, t) in enumerate(zip(x.src[0].shape, x.shape)) if s == 1 and t > 1}
if x.op is Ops.EXPAND else set()
  if {frozenset(bcast_axes(a)), frozenset(bcast_axes(b))} != {frozenset({1}),
frozenset({0})}: return None
  return (M, N, K) if all(x % 16 == 0 for x in (M, N, K)) else None


def wmma_gemm(M: int, N: int, K: int) -> str:
  """C(f32, MxN) = A(f16, MxK) @ B(f16, KxN), one 16x16 output tile per warp, via
nvcuda::wmma.

  A warp (32 threads) cooperatively owns three "fragments": 16x16 tiles of A, B and C
  spread across its registers in a hardware-defined layout. mma_sync does a
  16x16x16 matrix multiply-accumulate in one instruction on the tensor cores.
  """
  return f"""#include <mma.h>
using namespace nvcuda;
extern "C" __global__ void wmma_gemm(const half* __restrict__ A, const half* __restrict__ B,
float* __restrict__ C) {{
  int tile_m = blockIdx.y, tile_n = blockIdx.x;          // one warp per 16x16 tile of C
  wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a;
  wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b;
  wmma::fragment<wmma::accumulator, 16, 16, 16, float> c;
  wmma::fill_fragment(c, 0.0f);
  for (int k = 0; k < {K}; k += 16) {{
    wmma::load_matrix_sync(a, A + tile_m * 16 * {K} + k, {K});
    wmma::load_matrix_sync(b, B + k * {N} + tile_n * 16, {N});
    wmma::mma_sync(c, a, b, c);
  }}
  wmma::store_matrix_sync(C + tile_m * 16 * {N} + tile_n * 16, c, {N}, wmma::mem_row_major);
}}
/* launch: grid = ({N // 16}, {M // 16}), block = (32, 1, 1) */
"""
```

uopc/cuda.py · 90 lines

```python
"""Weeks 7-8: a minimal CUDA runtime over ctypes: NVRTC to compile, the driver API to run.

NOTE: this file needs an NVIDIA GPU, a working driver (libcuda.so) and NVRTC
(libnvrtc.so, from the CUDA toolkit). The reference machine for this book had
neither working, so this module is exercised only by `tests/test_w07_gpu.py`
when CUDA is available; elsewhere GPU kernels are tested with gpu.emulate().
"""
from __future__ import annotations
import ctypes, ctypes.util
import numpy as np


def _lib(*names):
  for n in names:
    path = ctypes.util.find_library(n)
    if path: return ctypes.CDLL(path)
  raise OSError(f"none of {names} found")


class CUDAError(Exception): pass


class CUDA:
  def __init__(self, device: int = 0):
    self.cu = _lib("cuda")
    self.rtc = _lib("nvrtc")
    self._check(self.cu.cuInit(0))
    self.dev = ctypes.c_int()
    self._check(self.cu.cuDeviceGet(ctypes.byref(self.dev), device))
    self.ctx = ctypes.c_void_p()
    self._check(self.cu.cuCtxCreate_v2(ctypes.byref(self.ctx), 0, self.dev))
    major, minor = ctypes.c_int(), ctypes.c_int()
    self._check(self.cu.cuDeviceComputeCapability(ctypes.byref(major), ctypes.byref(minor),
self.dev))
    self.arch = f"sm_{major.value}{minor.value}"

  def _check(self, status):
    if status != 0: raise CUDAError(f"CUDA driver error {status}")

  def compile(self, src: str) -> bytes:
    prog = ctypes.c_void_p()
    if self.rtc.nvrtcCreateProgram(ctypes.byref(prog), src.encode(), b"kernel.cu", 0, None,
None):
      raise CUDAError("nvrtcCreateProgram failed")
    opts = [f"--gpu-architecture=compute_{self.arch[3:]}".encode(), b"--use_fast_math"]
    status = self.rtc.nvrtcCompileProgram(prog, len(opts), (ctypes.c_char_p * len(opts))
(*opts))
    if status != 0:
      n = ctypes.c_size_t()
      self.rtc.nvrtcGetProgramLogSize(prog, ctypes.byref(n))
      log = ctypes.create_string_buffer(n.value)
      self.rtc.nvrtcGetProgramLog(prog, log)
```

```
      raise CUDAError(f"nvrtc: {log.value.decode()}\n{src}")
    n = ctypes.c_size_t()
    self.rtc.nvrtcGetPTXSize(prog, ctypes.byref(n))
    ptx = ctypes.create_string_buffer(n.value)
    self.rtc.nvrtcGetPTX(prog, ptx)
    return ptx.raw

  def alloc(self, nbytes: int) -> ctypes.c_uint64:
    p = ctypes.c_uint64()
    self._check(self.cu.cuMemAlloc_v2(ctypes.byref(p), ctypes.c_size_t(nbytes)))
    return p

  def to_device(self, a: np.ndarray) -> ctypes.c_uint64:
    p = self.alloc(a.nbytes)
    self._check(self.cu.cuMemcpyHtoD_v2(p, a.ctypes.data_as(ctypes.c_void_p),
ctypes.c_size_t(a.nbytes)))
    return p

  def to_host(self, p: ctypes.c_uint64, a: np.ndarray):
    self._check(self.cu.cuMemcpyDtoH_v2(a.ctypes.data_as(ctypes.c_void_p), p,
ctypes.c_size_t(a.nbytes)))

  def free(self, p): self.cu.cuMemFree_v2(p)

  def function(self, src: str, name: str):
    mod = ctypes.c_void_p()
    self._check(self.cu.cuModuleLoadData(ctypes.byref(mod), self.compile(src)))
    fn = ctypes.c_void_p()
    self._check(self.cu.cuModuleGetFunction(ctypes.byref(fn), mod, name.encode()))
    return fn

  def launch(self, fn, grid, block, *ptrs, shared: int = 0):
    args = (ctypes.c_void_p * len(ptrs))(*(ctypes.cast(ctypes.byref(p), ctypes.c_void_p) for
p in ptrs))
    self._check(self.cu.cuLaunchKernel(fn, *grid, *block, shared, None, args, None))
    self._check(self.cu.cuCtxSynchronize())


def run_cuda(src: str, name: str, grid, block, *arrays: np.ndarray):
  """Copy arrays to the GPU, run, copy every array back (they may be outputs)."""
  cu = CUDA()
  ptrs = [cu.to_device(a) for a in arrays]
  cu.launch(cu.function(src, name), grid, block, *ptrs)
  for p, a in zip(ptrs, arrays): cu.to_host(p, a); cu.free(p)
```

D.2 Tests: `tests/`

One file per week. `helpers.py` lowers and runs a shaped graph against numpy; `conftest.py` puts it on the path.

tests/conftest.py · 3 lines

```
import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent))
```

tests/helpers.py · 15 lines

```
import numpy as np
from uopc.uop import UOp, Ops, ParamArg, prod
from uopc.dtype import dtypes
from uopc.rangeify import lower
from uopc.runtime import run_c, np_dtype

def buf(slot, n, dtype=dtypes.f32): return UOp(Ops.BUFFER, (), ParamArg(slot, dtype, (n,)))

def run_shaped(root: UOp, inputs: list[np.ndarray]) -> np.ndarray:
  """Lower a shaped graph whose BUFFERs are slots 0..len(inputs)-1, run it, return the
result."""
  n = max(1, prod(root.shape))
  out = buf(len(inputs), n, root.dtype)
  res = np.zeros(n, np_dtype(root.dtype))
  run_c(lower(root, out), {**dict(enumerate(inputs)), len(inputs): res})
  return res.reshape(root.shape)
```

tests/test_w01_uops.py · 128 lines

```python
"""Week 1: UOps, hash consing, rewriting, symbolic, evaluation by rewriting."""
import math, pytest
from uopc.uop import UOp, Ops, ParamArg, CallArg, UOpError, range_
from uopc.dtype import dtypes
from uopc.rewrite import UPat, PatternMatcher, graph_rewrite
from uopc.symbolic import run, simplify, to_python, vmin, vmax, exec_alu
from uopc.uir import parse, to_uir

def stack(*xs): return UOp(Ops.STACK, tuple(x if isinstance(x, UOp) else UOp.const(x) for x
in xs))
def param(slot, dtype=dtypes.i32, shape=()): return UOp(Ops.PARAM, (), ParamArg(slot, dtype,
shape))

def test_hash_consing():
  a, b = UOp.const(1) + UOp.const(2), UOp.const(1) + UOp.const(2)
  assert a is b
  assert UOp.const(1) is not UOp.const(1.0) is not UOp.const(True)
  assert UOp.const(0.0) is not UOp.const(-0.0)

def test_shape_dtype_inference():
  s = stack(stack(1, 2, 3), stack(4, 5, 6))
  assert s.shape == (2, 3) and s.dtype == dtypes.i32
  assert s.index(UOp.const(1)).shape == (3,)
  assert (UOp.const(1) < UOp.const(2)).dtype == dtypes.bool
  with pytest.raises(UOpError): UOp.const(1) + UOp.const(1.0)
  with pytest.raises(UOpError): stack(1, 2) + stack(1, 2, 3)
  with pytest.raises(UOpError): stack(1, 2.0)

def test_const_fold_semantics():
  assert exec_alu(Ops.FLOORDIV, dtypes.i32, [-7, 2]) == -4          # floor, not truncation
  assert exec_alu(Ops.FLOORMOD, dtypes.i32, [-7, 2]) == 1
  assert exec_alu(Ops.ADD, dtypes.i32, [2**31 - 1, 1]) == -2**31     # wraps like int32
  assert exec_alu(Ops.CAST, dtypes.i32, [-2.7]) == -2                # C truncation
  assert exec_alu(Ops.ADD, dtypes.f32, [0.1, 0.2]) == pytest.approx(0.3, abs=1e-7)
  assert exec_alu(Ops.MUL, dtypes.f32, [16777216.0, 1.0]) == 16777216.0
  assert exec_alu(Ops.ADD, dtypes.f32, [16777216.0, 1.0]) == 16777216.0   # f32 rounding

def test_run_stack_program():
  a, b = stack(1, 2, 3), stack(10, 20, 30)
  assert to_python(run(a * b + a)) == [11, 42, 93]
  assert to_python(run(stack(a, b).index(UOp.const(1)).index(UOp.const(2)))) == 30
  w = (a < stack(2, 2, 2)).where(a, b)
  assert to_python(run(w)) == [1, 20, 30]

def test_call_inlining_and_scoping():
  p0 = param(0)
  sq = UOp(Ops.CALL, (p0 * p0, UOp.const(7)), CallArg("sq"))
  assert run(sq).arg == 49
  # nested: f(x) = sq(x) + x ; the inner PARAM 0 is a different scope from the outer one
  inner = UOp(Ops.CALL, (p0 * p0, p0 + 1), CallArg("sq"))
  outer = UOp(Ops.CALL, (inner + p0, UOp.const(3)), CallArg("f"))
  assert run(outer).arg == 16 + 3
```

```python
def test_call_type_checks():
  with pytest.raises(UOpError): UOp(Ops.CALL, (param(0) + param(1), UOp.const(1)))

def test_symbolic_identities():
  x = param(0)
  assert simplify(x + 0) is x and simplify(x * 1) is x and simplify(x * 0).arg == 0
  assert simplify((x + 3) + 4) is simplify(x + 7)
  assert simplify((x * 4) // 4) is x
  assert simplify((x * 6 + 4) % 2).arg == 0

def test_symbolic_ranges():
  i, j = range_(4, 0), range_(8, 1)
  flat = i * 8 + j
  assert simplify(flat // 8) is i and simplify(flat % 8) is j
  assert simplify((flat // 8) * 8 + flat % 8) is simplify(flat)
  assert simplify(j // 8).arg == 0 and simplify(j % 8) is j
  assert simplify(i < 4).arg is True and simplify(i < 0).arg is False
  assert (vmin(flat), vmax(flat)) == (0, 31)

def test_custom_pattern_matcher():
  pm = PatternMatcher([(UPat(Ops.MUL, src=[UPat.var("x"), UPat.cvar("c",
dtype=dtypes.i32)]),
                        lambda x, c: x + x if c.arg == 2 else None)])
  x = param(0)
  assert graph_rewrite(x * 2, pm) is x + x

def test_deep_graph_no_recursion_error():
  x = UOp.const(0)
  for i in range(20000): x = x + 1
  assert run(x).arg == 20000

def test_uir_roundtrip_and_syllabus_example():
  src = """
  %0 = buffer 65536 : dtype=f32 slot=0
  %1 = buffer 65536 : dtype=f32 slot=1
  %2 = reshape %0, (256, 256, 1)
  %3 = reshape %1, (256, 1, 256)
  %4 = mul %2, %3
  %5 = reduce %4 : op=add pop=1
  """
  g = parse(src)
  assert g.shape == (256, 256)
  assert parse(to_uir(g)) is g

def test_uir_evaluates():
  g = parse("""
  %0 = stack 1.0, 2.0, 3.0
  %1 = stack %0, %0
  %2 = mul %1, 2
  %3 = permute %2, (1, 0)
  %4 = reduce %3 : op=add pop=1
  """)
```

```python
    assert to_python(run(g)) == [4.0, 8.0, 12.0]

def test_folding_matches_c_semantics():
    # f32 inputs are rounded like the C compiler rounds the literal 16777217.0f
    assert simplify(UOp.const(16777217.0) + UOp.const(2.0)).arg == 16777218.0
    # constants made by reassociation wrap like int32 arithmetic: 65536*65536 == 0 (mod 2^32)
    assert simplify((param(0) * 65536) * 65536).arg == 0
    # INT_MIN // -1 traps in C: leave it to run time
    assert exec_alu(Ops.FLOORDIV, dtypes.i32, [-2**31, -1]) is None

def test_inline_call_does_not_enter_nested_bodies():            # NOTES 2
    from uopc.symbolic import inline_call
    p0 = param(0)
    inner = UOp(Ops.CALL, (p0 * p0, p0 + 1), CallArg("sq"))
    outer = UOp(Ops.CALL, (inner + p0, UOp.const(3)), CallArg("f"))
    body = inline_call(outer)                  # inline ONLY the outer call; inner must
survive intact
    assert body.src[0].op is Ops.CALL and body.src[0].src[0] is p0 * p0
    assert run(body).arg == 19

def test_cast_inf_to_int_is_not_folded():                        # NOTES 6
    assert exec_alu(Ops.CAST, dtypes.i32, [math.inf]) is None
    assert exec_alu(Ops.CAST, dtypes.i32, [math.nan]) is None

def test_float_constants_do_not_reassociate():                   # NOTES 7
    x = UOp(Ops.PARAM, (), ParamArg(0, dtypes.f32))
    assert simplify((x + 1.0) + 2.0) is not simplify(x + 3.0)
```

tests/test_w02_render.py · 45 lines

```python
"""Week 2: rendering loopless C, the runtime, memory (BUFFER/ALLOC/LOAD/STORE/AFTER)."""
import numpy as np
from uopc.uop import UOp, Ops, ParamArg, sink
from uopc.dtype import dtypes
from uopc.render_c import render_c
from uopc.runtime import run_c
from helpers import buf

def test_elementwise_loopless():
  a, b, c = buf(0, 4), buf(1, 4), buf(2, 4)
  s = sink(*[c.index(i).store(a.index(i).load() * b.index(i).load() + 1.0) for i in range(4)])
  A, B, C = np.arange(4, dtype=np.float32), np.full(4, 2, np.float32), np.zeros(4, np.float32)
  run_c(s, {0: A, 1: B, 2: C})
  np.testing.assert_allclose(C, A * B + 1)

def test_alloc_and_after():
  # tmp[0] = a[0] + a[1]; out[0] = tmp[0] * tmp[0]  -- AFTER orders the read after the write
  a, out = buf(0, 2), buf(1, 1)
  tmp = UOp(Ops.ALLOC, (), ParamArg(0, dtypes.f32, (1,)))
  w = tmp.index(0).store(a.index(0).load() + a.index(1).load())
  v = tmp.after(w).index(0).load()
  O = np.zeros(1, np.float32)
  run_c(sink(out.index(0).store(v * v)), {0: np.array([1, 2], np.float32), 1: O})
  assert O[0] == 9

def test_floordiv_rendering_negative():
  a, out = buf(0, 2, dtypes.i32), buf(1, 2, dtypes.i32)
  x = a.index(0).load()
  s = sink(out.index(0).store(x // a.index(1).load()), out.index(1).store(x %
a.index(1).load()))
  O = np.zeros(2, np.int32)
  run_c(s, {0: np.array([-7, 2], np.int32), 1: O})
  assert list(O) == [-4, 1]
  assert "floordiv_i32" in render_c(s)

def test_casts_and_where():
  a, out = buf(0, 3), buf(1, 3, dtypes.i32)
  s = sink(*[out.index(i).store((a.index(i).load() < 0.0).where(UOp.const(-1),
a.index(i).load().cast(dtypes.i32))) for i in range(3)])
  O = np.zeros(3, np.int32)
  run_c(s, {0: np.array([-2.5, 3.7, 0.2], np.float32), 1: O})
  assert list(O) == [-1, 3, 0]

def test_run_c_checks_buffer_dtypes():
  import pytest
  out = buf(0, 4)
  with pytest.raises(TypeError): run_c(sink(out.index(0).store(UOp.const(1.0))), {0:
np.ones(4)})   # float64
```

tests/test_w03_loops.py · 54 lines

```python
"""Week 3: RANGE/END. Hand-written gemm and conv in the loop language."""
import numpy as np
from uopc.uop import UOp, Ops, ParamArg, sink, range_
from uopc.dtype import dtypes
from uopc.runtime import run_c
from uopc.linearize import linearize
from helpers import buf

def gemm(N, M, K):
  A, B, C = buf(0, N * K), buf(1, K * M), buf(2, N * M)
  i, j, k = range_(N, 0), range_(M, 1), range_(K, 2)
  acc = UOp(Ops.ALLOC, (), ParamArg(0, dtypes.f32, (1,))).after(i, j)
  init = acc.index(0).store(UOp.const(0.0))
  acck = acc.after(init, k)
  upd = acck.index(0).store(acck.index(0).load() + A.index(i * K + k).load() * B.index(k * M
+ j).load())
  return sink(C.index(i * M + j).store(acc.after(upd.end(k)).index(0).load()).end(i, j))

def test_gemm_loops():
  N, M, K = 17, 9, 33
  a, b = np.random.randn(N, K).astype(np.float32), np.random.randn(K, M).astype(np.float32)
  c = np.zeros((N, M), np.float32)
  src = run_c(gemm(N, M, K), {0: a, 1: b, 2: c})
  np.testing.assert_allclose(c, a @ b, atol=1e-4)
  assert src.count("for (") == 3

def test_conv1d_loops():
  # out[o] = sum_k x[o + k] * w[k]
  n, kw = 20, 5
  X, W, O = buf(0, n), buf(1, kw), buf(2, n - kw + 1)
  o, k = range_(n - kw + 1, 0), range_(kw, 1)
  acc = UOp(Ops.ALLOC, (), ParamArg(0, dtypes.f32, (1,))).after(o)
  init = acc.index(0).store(UOp.const(0.0))
  acck = acc.after(init, k)
  upd = acck.index(0).store(acck.index(0).load() + X.index(o + k).load() *
W.index(k).load())
  s = sink(O.index(o).store(acc.after(upd.end(k)).index(0).load()).end(o))
  x, w, out = np.random.randn(n).astype(np.float32), np.random.randn(kw).astype(np.float32),
np.zeros(n - kw + 1, np.float32)
  run_c(s, {0: x, 1: w, 2: out})
  np.testing.assert_allclose(out, np.correlate(x, w, "valid"), atol=1e-5)

def test_loop_invariant_code_is_hoisted():
  N = 8
  A, C = buf(0, N * N), buf(1, N * N)
  i, j = range_(N, 0), range_(N, 1)
  row = A.index(i * N).load()            # depends only on i
  s = sink(C.index(i * N + j).store(row + A.index(i * N + j).load()).end(i, j))
  lin = linearize(s)
  pos = {u: n for n, u in enumerate(lin)}
  assert pos[row] < pos[j]                # emitted before the j loop opens
```

```python
def test_missing_end_is_an_error():
  import pytest
  from uopc.linearize import LinearizeError
  out, r = buf(0, 4), range_(4, 0)
  with pytest.raises(LinearizeError, match="never closed"):
linearize(sink(out.index(r).store(UOp.const(1.0))))
```

tests/test_w04_rangeify.py · 77 lines

```python
"""Week 4: movement ops, broadcasting, reduction, rangeify, composite PAD/FLIP."""
import numpy as np, pytest
from uopc.uop import UOp, Ops
from uopc.uir import parse
from uopc.composite import pad, flip_axis, anti_eye
from uopc.symbolic import run, to_python
from helpers import buf, run_shaped

def test_gemm_from_movement_ops():
  N = 16
  a, b = np.random.randn(N, N).astype(np.float32), np.random.randn(N, N).astype(np.float32)
  g = parse(f"""
  %0 = buffer {N*N} : dtype=f32 slot=0
  %1 = buffer {N*N} : dtype=f32 slot=1
  %2 = reshape %0, ({N}, 1, {N})
  %3 = reshape %1, ({N}, {N})
  %4 = permute %3, (1, 0)
  %5 = reshape %4, (1, {N}, {N})
  %6 = mul %2, %5
  %7 = reduce %6 : op=add pop=1""")
  np.testing.assert_allclose(run_shaped(g, [a.ravel(), b.ravel()]), a @ b, atol=1e-4)

def test_syllabus_gemm_is_not_a_gemm():
  # the example in the syllabus computes A * rowsum(B): a good reminder to test everything
  N = 8
  a, b = np.random.randn(N, N).astype(np.float32), np.random.randn(N, N).astype(np.float32)
  g = parse(f"""
  %0 = buffer {N*N} : dtype=f32 slot=0
  %1 = buffer {N*N} : dtype=f32 slot=1
  %2 = reshape %0, ({N}, {N}, 1)
  %3 = reshape %1, ({N}, 1, {N})
  %4 = mul %2, %3
  %5 = reduce %4 : op=add pop=1""")
  np.testing.assert_allclose(run_shaped(g, [a.ravel(), b.ravel()]), a * b.sum(1,
keepdims=True), atol=1e-4)

@pytest.mark.parametrize("seed", range(5))
def test_random_movement_chains(seed):
  rng = np.random.default_rng(seed)
  x = rng.standard_normal(24).astype(np.float32)
  u, ref = buf(0, 24).reshape((2, 3, 4)), x.reshape(2, 3, 4)
  for _ in range(6):
    k = rng.integers(4)
    if k == 0:
      p = tuple(rng.permutation(len(ref.shape))); u, ref = u.permute(p), ref.transpose(p)
    elif k == 1:
      arg = tuple((int(b), int(rng.integers(b + 1, s + 1))) for s in ref.shape for b in
[rng.integers(0, s)])
      u, ref = u.shrink(arg), ref[tuple(slice(b, e) for b, e in arg)]
    elif k == 2:
      u, ref = u.reshape((1,) + u.shape), ref.reshape((1,) + ref.shape)
      u, ref = u.expand((2,) + u.shape[1:]), np.broadcast_to(ref, (2,) + ref.shape[1:])
```

```
    else:
      u, ref = u.reshape((int(np.prod(ref.shape)),)), ref.reshape(-1)
  np.testing.assert_allclose(run_shaped(u, [x]), ref)

def test_reduce_max_and_nested():
  x = np.random.randn(6, 5).astype(np.float32)
  u = buf(0, 30).reshape((6, 5))
  np.testing.assert_allclose(run_shaped(u.reduce(Ops.MAX, 1), [x.ravel()]), x.max(1))
  # x - max(x) broadcast back: the reduce becomes a sibling loop, computed once per row (no
kernelizer needed)
  m = u.reduce(Ops.MAX, 1).reshape((6, 1)).expand((6, 5))
  np.testing.assert_allclose(run_shaped(u + -m, [x.ravel()]), x - x.max(1, keepdims=True))
  np.testing.assert_allclose(run_shaped(u.reduce(Ops.ADD, 2), [x.ravel()]), x.sum(),
rtol=1e-5)

def test_stack_and_index_in_rangeify():
  x = np.random.randn(4).astype(np.float32)
  u = buf(0, 4)
  s = UOp(Ops.STACK, (u, u * u, u + u))
  np.testing.assert_allclose(run_shaped(s, [x]), np.stack([x, x * x, x + x]))
  np.testing.assert_allclose(run_shaped(s.index(UOp.const(1)), [x]), x * x)

def test_pad_flip_composites():
  x = np.random.randn(3, 4).astype(np.float32)
  u = buf(0, 12).reshape((3, 4))
  np.testing.assert_allclose(run_shaped(pad(u, ((1, 5), (2, 0))), [x.ravel()]), np.pad(x,
((1, 5), (2, 0))))
  np.testing.assert_allclose(run_shaped(flip_axis(u, 1), [x.ravel()]), x[:, ::-1])
  np.testing.assert_allclose(run_shaped(flip_axis(u, 0), [x.ravel()]), x[::-1])
  assert to_python(run(anti_eye(3, u.dtype))) == [[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

tests/test_w05_tensor.py · 111 lines

```python
"""Week 5: the Tensor frontend, kernelization (fusion/fission), autodiff."""
import math
import numpy as np, pytest
from uopc.tensor import Tensor
from uopc.kernelize import make_plan
from uopc.nn import Linear, SGD, AdamW, parameters

rng = np.random.default_rng(0)
def r(*s): return rng.standard_normal(s).astype(np.float32)

@pytest.mark.parametrize("name,fn,ref", [
  ("add_broadcast", lambda a, b: a + b[0], lambda a, b: a + b[0]),
  ("mul_sub_div", lambda a, b: (a * b - 3) / (b.abs() + 1), lambda a, b: (a * b - 3) /
(np.abs(b) + 1)),
  ("exp_log", lambda a, b: (a.exp() + 1).log(), lambda a, b: np.log(np.exp(a) + 1)),
  ("sum_axes", lambda a, b: a.sum((0, 2)), lambda a, b: a.sum((0, 2))),
  ("max_keepdim", lambda a, b: a.max(1, keepdim=True), lambda a, b: a.max(1,
keepdims=True)),
  ("mean_var", lambda a, b: a.var(-1), lambda a, b: a.var(-1)),
  ("softmax", lambda a, b: a.softmax(-1), lambda a, b: np.exp(a - a.max(-1, keepdims=True))
/ np.exp(a - a.max(-1, keepdims=True)).sum(-1, keepdims=True)),
  ("matmul_batched", lambda a, b: a @ b.transpose(), lambda a, b: a @ b.transpose(0, 2, 1)),
  ("where", lambda a, b: (a > b).where(a, b * 2), lambda a, b: np.where(a > b, a, b * 2)),
  ("getitem_cat", lambda a, b: Tensor.cat([a[:, 1:], b[:, :1]], 1), lambda a, b:
np.concatenate([a[:, 1:], b[:, :1]], 1)),
  ("pad_flip", lambda a, b: a.pad(((0, 0), (1, 2), (0, 1))).flip(1), lambda a, b: np.pad(a,
((0, 0), (1, 2), (0, 1)))[:, ::-1]),
])
def test_ops(name, fn, ref):
  a, b = r(2, 3, 4), r(2, 3, 4)
  np.testing.assert_allclose(fn(Tensor(a), Tensor(b)).numpy(), ref(a, b), rtol=1e-4,
atol=1e-5)

def numgrad(f, x, eps=1e-3):
  g = np.zeros_like(x, dtype=np.float64)
  for i in np.ndindex(x.shape):
    xp, xm = x.astype(np.float64), x.astype(np.float64)
    xp[i] += eps; xm[i] -= eps
    g[i] = (f(xp) - f(xm)) / (2 * eps)
  return g

@pytest.mark.parametrize("name,fn,npfn", [
  ("tanh_sum", lambda t: t.tanh().sum(), lambda x: np.tanh(x).sum()),
  ("softmax_xent", lambda t: t.log_softmax(-1)[:, 1].sum(), lambda x: (x -
np.log(np.exp(x).sum(-1, keepdims=True)))[:, 1].sum()),
  ("max", lambda t: (t.max(-1) * t.sum(0).max()).sum(), lambda x: (x.max(-1) *
x.sum(0).max()).sum()),
  ("pad_shrink", lambda t: t.pad(((1, 1), (0, 2)))[1:3, 1:].square().sum(),
   lambda x: (np.pad(x, ((1, 1), (0, 2)))[1:3, 1:] ** 2).sum()),
  ("sqrt_recip", lambda t: (t.square() + 1).sqrt().reciprocal().sum(), lambda x: (1 /
np.sqrt(x ** 2 + 1)).sum()),
  ("matmul", lambda t: (t @ t.T).sin().sum(), lambda x: np.sin(x @ x.T).sum()),
```

```
])
def test_gradients(name, fn, npfn):
  x = r(3, 4)
  t = Tensor(x, requires_grad=True)
  fn(t).backward([t])
  np.testing.assert_allclose(t.grad.numpy(), numgrad(npfn, x), rtol=2e-2, atol=2e-3)

def test_kernelize_fuses_elementwise_and_splits_reductions():
  a = Tensor(r(64, 64))
  plan, _ = make_plan([((a * 2 + 1).relu() * 3).uop])
  assert len(plan.kernels) == 1                        # a chain of elementwise ops: one
kernel
  plan, _ = make_plan([a.softmax(-1).uop])
  assert len(plan.kernels) == 3                        # max, sum, and the elementwise
output

def test_training_reduces_loss():
  Tensor.manual_seed(1)
  X = r(64, 8); Y = (X.sum(1) > 0).astype(np.int32)
  l1, l2 = Linear(8, 16), Linear(16, 2)
  opt = AdamW(parameters([l1, l2]), lr=1e-2)
  losses = []
  for _ in range(40):
    loss = l2(l1(Tensor(X)).relu()).cross_entropy(Tensor(Y))
    opt.zero_grad(); loss.backward(opt.params); opt.step(loss)
    losses.append(loss.item())
  assert losses[-1] < 0.3 * losses[0]

def test_sgd_momentum_step():
  w = Tensor(np.array([1.0, -2.0], np.float32), requires_grad=True)
  opt = SGD([w], lr=0.1, momentum=0.9)
  for _ in range(2):
    loss = w.square().sum()
    opt.zero_grad(); loss.backward(opt.params); opt.step()
  # v1 = 2w0, w1 = w0 - .1*v1 = .8 w0;  v2 = .9 v1 + 2 w1 = 3.4 w0, w2 = .8 w0 - .34 w0 =
.46 w0
  np.testing.assert_allclose(w.numpy(), np.array([0.46, -0.92]), rtol=1e-5)

def test_deleted_reads_do_not_shift_kernel_arguments():
  t = Tensor(np.array([1, 2, 3], np.int32))
  assert (t * 0).numpy().tolist() == [0, 0, 0]          # simplifies to a constant: t is no
longer read...
  assert t.numpy().tolist() == [1, 2, 3]                # ...and must not be overwritten

def test_python_float_promotes_int_tensors():
  t = Tensor(np.array([1, 2, 3], np.int32))
  np.testing.assert_allclose((t * 0.5).numpy(), [0.5, 1.0, 1.5])
  assert t.mean().item() == 2.0

def test_item_before_backward_and_detach():
  w = Tensor(np.array([1.0, 2.0], np.float32), requires_grad=True)
  loss = (w * w).sum()
  loss.item()                                          # realizing the loss must keep its
```

```
graph
  loss.backward([w])
  np.testing.assert_allclose(w.grad.numpy(), [2.0, 4.0])
  w.grad = None
  (w.detach() * w).sum().backward([w])                # d/dw (c * w) with c = w held
constant
  np.testing.assert_allclose(w.grad.numpy(), [1.0, 2.0])

def test_adamw_steps_reuse_one_plan():                            # NOTES 17
  from uopc import kernelize
  Tensor.manual_seed(0)
  lin = Linear(4, 2)
  opt = AdamW([lin.weight, lin.bias], lr=1e-2)
  x = Tensor(np.random.randn(8, 4).astype(np.float32))
  def step():
    loss = lin(x).square().mean()
    opt.zero_grad(); loss.backward(opt.params); opt.step(loss)
  step(); step()
  n = len(kernelize._plans)
  for _ in range(3): step()
  assert len(kernelize._plans) == n                               # no new plans: nothing
recompiled
```

tests/test_w06_opt.py · 72 lines

```python
"""Week 6: upcast, unroll, threads, layout. Optimized kernels must compute the same
thing."""
import numpy as np, pytest
from uopc.uop import Ops
from uopc.rangeify import lower, Ctx, Opts
from uopc.render_c import render_c
from uopc.runtime import Program
from uopc.opt import hand_opt_choice, layout_pass
from helpers import buf

def gemm_graph(M, N, K, bt=False):
  A, B = buf(0, M * K), buf(1, K * N)
  b = B.reshape((N, K)) if bt else B.reshape((K, N)).permute((1, 0))
  return (A.reshape((M, 1, K)).expand((M, N, K)) * b.reshape((1, N, K)).expand((M, N,
K))).reduce(Ops.ADD, 1)

@pytest.mark.parametrize("opts", [
  Opts(), Opts(thread=0), Opts(upcast={1: 4}), Opts(upcast={0: 4, 1: 4}, thread=0),
  Opts(unroll=4), Opts(upcast={0: 2, 1: 8}, unroll=4, thread=1), Opts(upcast={0: 32})])
def test_gemm_opts_correct(opts):
  M, N, K = 32, 48, 64
  a, b = np.random.randn(M, K).astype(np.float32), np.random.randn(K, N).astype(np.float32)
  c = np.zeros(M * N, np.float32)
  src = render_c(lower(gemm_graph(M, N, K), buf(2, M * N), Ctx(opts=opts)), "k")
  Program(src, "k")(a, b, c)
  np.testing.assert_allclose(c.reshape(M, N), a @ b, rtol=1e-4, atol=1e-4)
  if opts.thread is not None: assert "#pragma omp parallel for" in src

def test_upcast_shares_reduce_loop():
  src = render_c(lower(gemm_graph(8, 8, 8), buf(2, 64), Ctx(opts=Opts(upcast={0: 2, 1:
2}))), "k")
  assert src.count("for (") == 3                        # 2 output loops + ONE shared reduce
loop
  assert src.count("acc") >= 4

def test_reduce_unroll_has_no_inner_loop():
  x = buf(0, 16)
  src = render_c(lower(x.reduce(Ops.ADD, 1), buf(1, 1), Ctx(opts=Opts(unroll=16))), "k")
  assert "for (" not in src

def test_layout_pass_stages_transposed_operand():
  g = gemm_graph(64, 64, 64)
  (g2,) = layout_pass([g])
  assert any(u.op is Ops.STAGE for u in g2.toposort())
  (g3,) = layout_pass([gemm_graph(64, 64, 64, bt=True)])     # already contiguous: nothing
to do
  assert not any(u.op is Ops.STAGE for u in g3.toposort())

def test_hand_opt_choice():
  o = hand_opt_choice(gemm_graph(64, 64, 64))
  assert o.upcast and o.thread is not None
```

```python
def max_loop_depth(src: str) -> int:
  return max((len(l) - len(l.lstrip())) // 2 for l in src.splitlines() if
l.lstrip().startswith("for "))

def test_fused_attention_reduces_are_not_nested():                # NOTES 12
  import sys
  from pathlib import Path
  sys.path.insert(0, str(Path(__file__).parent.parent / "examples"))
  from flash_attention import attention, FlashOpt
  from uopc.tensor import Tensor
  from uopc.kernelize import make_plan
  N, D = 64, 16
  q, k, v = (Tensor(np.random.randn(N, D).astype(np.float32)) for _ in range(3))
  plan, _ = make_plan([attention(q, k, v).uop], fuse_reduce=True, opt=FlashOpt)
  # loops: i (thread), then a reduce over keys j, inside it a dot product over D: depth 3,
never 4
  assert max(max_loop_depth(k.src) for k in plan.kernels) <= 3

@pytest.mark.parametrize("tile", [{0: 8, 1: 16}, {1: 32}, {0: 16}])
def test_cache_blocking_correct(tile):
  M, N, K = 32, 64, 48
  a, b = np.random.randn(M, K).astype(np.float32), np.random.randn(K, N).astype(np.float32)
  c = np.zeros(M * N, np.float32)
  src = render_c(lower(gemm_graph(M, N, K), buf(2, M * N), Ctx(opts=Opts(upcast={0: 4, 1:
4}, thread=0, tile=tile))), "k")
  Program(src, "k")(a, b, c)
  np.testing.assert_allclose(c.reshape(M, N), a @ b, rtol=1e-4, atol=1e-4)
  assert src.count("for (") == 3 + len(tile)               # one extra block loop per tiled
axis
```

tests/test_w07_gpu.py · 66 lines

```python
"""Weeks 7-8: GPU mapping. Every kernel is checked on the CPU through the emulator;
the CUDA test only runs when a GPU and NVRTC are available."""
import numpy as np, pytest
from uopc.uop import UOp, Ops, AxisType
from uopc.dtype import dtypes
from uopc.runtime import Program
from uopc.rangeify import Opts
from uopc.gpu import gpu_lower, gpu_opt_choice, emulate, render_cuda, launch_dims,
smem_gemm_ir, wmma_gemm, match_gemm
from helpers import buf

def gemm_graph(M, N, K):
  A, B = buf(0, M * K), buf(1, K * N)
  return (A.reshape((M, 1, K)).expand((M, N, K)) * B.reshape((K, N)).permute((1,
0)).reshape((1, N, K)).expand((M, N, K))).reduce(Ops.ADD, 1)

@pytest.mark.parametrize("opts", [Opts(gpu=True), Opts(gpu=True, local={0: 4, 1: 8}),
                                  Opts(gpu=True, local={1: 16}, upcast={0: 2, 1: 2})])
def test_gpu_gemm_emulated(opts):
  M, N, K = 32, 64, 48
  a, b = np.random.randn(M, K).astype(np.float32), np.random.randn(K, N).astype(np.float32)
  c = np.zeros(M * N, np.float32)
  ast = gpu_lower(gemm_graph(M, N, K), buf(2, M * N), opts)
  Program(emulate(ast), "kernel")(a, b, c)
  np.testing.assert_allclose(c.reshape(M, N), a @ b, atol=1e-4)
  src = render_cuda(ast)
  assert "__global__" in src and ("threadIdx" in src) == bool(opts.local)

def test_gpu_softmax_like_emulated():
  x = np.random.randn(16, 32).astype(np.float32)
  X = buf(0, 16 * 32).reshape((16, 32))
  m = X.reduce(Ops.MAX, 1).reshape((16, 1)).expand((16, 32))
  body = UOp(Ops.EXP2, (X + -m,))
  out = np.zeros(16 * 32, np.float32)
  ast = gpu_lower(body, buf(1, 16 * 32))
  Program(emulate(ast), "kernel")(x, out)
  np.testing.assert_allclose(out.reshape(16, 32), np.exp2(x - x.max(1, keepdims=True)),
rtol=1e-5)

def test_launch_dims_and_auto_choice():
  o = gpu_opt_choice(gemm_graph(64, 64, 64))
  ast = gpu_lower(gemm_graph(64, 64, 64), buf(2, 64 * 64), o)
  grid, block = launch_dims(ast)
  assert np.prod(grid) * np.prod(block) * np.prod(list(o.upcast.values()) or [1]) == 64 * 64

def test_smem_gemm_renders():
  src = render_cuda(smem_gemm_ir(64, 16), "smem_gemm")
  assert src.count("__syncthreads();") == 2 and "__shared__ float smem0[256];" in src
  assert launch_dims(smem_gemm_ir(64, 16)) == ((4, 4, 1), (16, 16, 1))

def test_wmma():
  assert match_gemm(gemm_graph(32, 32, 32)) == (32, 32, 32)
```

```python
  assert "mma_sync" in wmma_gemm(32, 32, 32)

def _has_cuda():
  try:
    from uopc.cuda import CUDA
    CUDA(); return True
  except Exception: return False

@pytest.mark.skipif(not _has_cuda(), reason="no CUDA device / NVRTC")
def test_cuda_smem_gemm_on_gpu():
  from uopc.cuda import run_cuda
  N = 64
  a, b, c = np.random.randn(N, N).astype(np.float32), np.random.randn(N,
N).astype(np.float32), np.zeros(N * N, np.float32)
  ast = smem_gemm_ir(N, 16)
  grid, block = launch_dims(ast)
  run_cuda(render_cuda(ast, "smem_gemm"), "smem_gemm", grid, block, a, b, c)
  np.testing.assert_allclose(c.reshape(N, N), a @ b, atol=1e-3)
```

tests/test_w09_models.py · 38 lines

```python
"""Weeks 9-10: a tiny GPT trains (loss goes down) and its pieces match numpy."""
import sys
from pathlib import Path
import numpy as np
sys.path.insert(0, str(Path(__file__).parent.parent / "examples"))
from gpt import GPT, Attention
from uopc.tensor import Tensor
from uopc.nn import LayerNorm, AdamW, parameters

def test_layernorm_matches_numpy():
  x = np.random.randn(4, 8).astype(np.float32)
  y = LayerNorm(8)(Tensor(x)).numpy()
  ref = (x - x.mean(-1, keepdims=True)) / np.sqrt(x.var(-1, keepdims=True) + 1e-5)
  np.testing.assert_allclose(y, ref, atol=1e-5)

def test_attention_is_causal():
  Tensor.manual_seed(0)
  att = Attention(8, 2, 6)
  x = np.random.randn(1, 6, 8).astype(np.float32)
  y1 = att(Tensor(x)).numpy()
  x2 = x.copy(); x2[0, 4:] += 10                  # change the future
  y2 = att(Tensor(x2)).numpy()
  np.testing.assert_allclose(y1[0, :4], y2[0, :4], atol=1e-5)   # the past doesn't see it

def test_tiny_gpt_learns():
  Tensor.manual_seed(0)
  V, T = 11, 8
  model = GPT(V, T, 16, 2, 1)
  opt = AdamW(parameters(model), lr=1e-2)
  seq = np.arange(64) % V                          # a trivially predictable sequence
  x = Tensor(np.stack([seq[i:i + T] for i in range(4)]).astype(np.int32))
  y = Tensor(np.stack([seq[i + 1:i + T + 1] for i in range(4)]).astype(np.int32))
  losses = []
  for _ in range(30):
    loss = model(x).reshape(-1, V).cross_entropy(y.reshape(-1))
    opt.zero_grad(); loss.backward(opt.params); opt.step(loss)
    losses.append(loss.item())
  assert losses[-1] < 0.25 * losses[0]
```

D.3 Examples: `examples/`

examples/datasets.py · 29 lines

```python
"""Tiny dataset loaders. Files are cached in ~/.cache/uopc/data."""
import gzip, urllib.request
from pathlib import Path
import numpy as np

DATA = Path.home() / ".cache" / "uopc" / "data"
MNIST_URL = "https://storage.googleapis.com/cvdf-datasets/mnist/"
SHAKESPEARE_URL = "https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt"


def fetch(url: str, name: str) -> Path:
  DATA.mkdir(parents=True, exist_ok=True)
  p = DATA / name
  if not p.exists():
    print(f"downloading {url}")
    urllib.request.urlretrieve(url, p)
  return p


def mnist():
  """(X_train (60000, 784) f32 in [0,1], Y_train (60000,) i32, X_test, Y_test)"""
  def images(n): return np.frombuffer(gzip.open(fetch(MNIST_URL + n, n)).read(), np.uint8, offset=16).reshape(-1, 784).astype(np.float32) / 255
  def labels(n): return np.frombuffer(gzip.open(fetch(MNIST_URL + n, n)).read(), np.uint8, offset=8).astype(np.int32)
  return (images("train-images-idx3-ubyte.gz"), labels("train-labels-idx1-ubyte.gz"),
          images("t10k-images-idx3-ubyte.gz"), labels("t10k-labels-idx1-ubyte.gz"))


def shakespeare() -> str:
  return fetch(SHAKESPEARE_URL, "tinyshakespeare.txt").read_text()
```

examples/mnist_mlp.py · 43 lines

```python
"""Week 5 milestone: train an MLP on MNIST with our own compiler.

  python examples/mnist_mlp.py            # ~94% test accuracy in 600 steps (~15 ms/step)
  DEBUG=2 python examples/mnist_mlp.py    # print every kernel with its time
"""
import sys, time
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent))
import numpy as np
from datasets import mnist
from uopc.tensor import Tensor
from uopc.nn import Linear, AdamW, parameters


class MLP:
  def __init__(self):
    self.l1, self.l2 = Linear(784, 128), Linear(128, 10)
  def __call__(self, x: Tensor) -> Tensor: return self.l2(self.l1(x).relu())


def main(steps=600, bs=128):
  Xtr, Ytr, Xte, Yte = mnist()
  Tensor.manual_seed(0)
  model = MLP()
  opt = AdamW(parameters(model), lr=1e-3)
  rng = np.random.default_rng(0)
  t0 = time.perf_counter()
  for step in range(steps):
    idx = rng.integers(0, len(Xtr), bs)
    x, y = Tensor(Xtr[idx]), Tensor(Ytr[idx])
    loss = model(x).cross_entropy(y)
    opt.zero_grad()
    loss.backward(opt.params)
    opt.step(loss)
    if step % 100 == 0 or step == steps - 1:
      print(f"step {step:4d}  loss {loss.item():.4f}  {(time.perf_counter() - t0) / (step + 1)
* 1e3:.1f} ms/step")
  pred = np.concatenate([model(Tensor(Xte[i:i + 1000])).argmax(-1).numpy() for i in range(0,
len(Xte), 1000)])
  acc = (pred == Yte).mean()
  print(f"test accuracy {acc * 100:.2f}%")
  return acc


if __name__ == "__main__": main()
```

examples/bench_gemm.py · 59 lines

```python
"""Week 6: GEMM GFLOP/s, step by step, vs numpy (which calls an optimized BLAS).

  python examples/bench_gemm.py 1024

  B strided   C[i,j] = sum_k A[i,k] B[k,j]: the reduce walks down a column of B (stride N)
  B^T         same math, B stored transposed: both operands are contiguous along k
"""
import sys, time
import numpy as np
from uopc.uop import UOp, Ops, ParamArg
from uopc.dtype import dtypes
from uopc.rangeify import lower, Ctx, Opts
from uopc.render_c import render_c
from uopc.runtime import Program
from uopc.tensor import Tensor


def gemm_graph(N, b_transposed=False):
  A = UOp(Ops.BUFFER, (), ParamArg(0, dtypes.f32, (N * N,)))
  B = UOp(Ops.BUFFER, (), ParamArg(1, dtypes.f32, (N * N,)))
  a = A.reshape((N, 1, N)).expand((N, N, N))
  bt = B.reshape((N, N)) if b_transposed else B.reshape((N, N)).permute((1, 0))
  return (a * bt.reshape((1, N, N)).expand((N, N, N))).reduce(Ops.ADD, 1)


def bench(fn, reps=5):
  fn(); best = float("inf")
  for _ in range(reps):
    t0 = time.perf_counter(); fn(); best = min(best, time.perf_counter() - t0)
  return best


if __name__ == "__main__":
  N = int(sys.argv[1]) if len(sys.argv) > 1 else 1024
  a, b = np.random.randn(N, N).astype(np.float32), np.random.randn(N, N).astype(np.float32)
  bt = np.ascontiguousarray(b.T)
  c = np.zeros(N * N, np.float32)
  out = UOp(Ops.BUFFER, (), ParamArg(2, dtypes.f32, (N * N,)))
  flop = 2 * N ** 3
  print(f"N={N}  ({flop / 1e9:.2f} GFLOP per GEMM)")
  rows = [("B strided", False, "naive", Opts()),
          ("B strided", False, "threads", Opts(thread=0)),
          ("B strided", False, "4x4 tile + threads", Opts(upcast={0: 4, 1: 4}, thread=0)),
          ("B^T", True, "naive", Opts()),
          ("B^T", True, "threads", Opts(thread=0)),
          ("B^T", True, "4x4 tile", Opts(upcast={0: 4, 1: 4})),
          ("B^T", True, "4x4 tile + threads", Opts(upcast={0: 4, 1: 4}, thread=0)),
          ("B^T", True, "+ 64x64 cache blocks", Opts(upcast={0: 4, 1: 4}, thread=0, tile={0:
64, 1: 64}))]
  for layout, transposed, name, o in rows:
    prg = Program(render_c(lower(gemm_graph(N, transposed), out, Ctx(opts=o)), "gemm"),
"gemm")
```

```
    rhs = bt if transposed else b
    t = bench(lambda: prg(a, rhs, c))
    err = np.abs(c.reshape(N, N) - a @ b).max()
    print(f"  {layout:10s} {name:20s} {flop / t / 1e9:8.1f} GFLOP/s   max err {err:.1e}")
  A, B = Tensor(a), Tensor(b)
  t = bench(lambda: (A @ B).realize())
  print(f"  {'Tensor A@B':10s} {'(layout pass+opt)':20s} {flop / t / 1e9:8.1f} GFLOP/s
(end to end, includes the transpose copy)")
  t = bench(lambda: a @ b)
  print(f"  {'numpy':31s} {flop / t / 1e9:8.1f} GFLOP/s")
```

examples/flash_attention.py · 107 lines

```python
"""Week 6: three ways to compute attention, out = softmax(Q K^T / sqrt(D)) V.

  1. unfused     -- the Tensor code, kernelized normally: S = QK^T is materialized (N*N
floats)
  2. fused       -- the same Tensor code with fuse_reduce=True: reduces nest inside each
other,
                    nothing N*N ever touches memory; QK^T is recomputed by every pass (max,
sum,
                    output) and, in the output pass, once per 16-wide tile of the head
dimension
  3. online      -- FlashAttention's online softmax, written by hand in the week-3 loop IR:
                    one pass over keys, running max m, running sum l, rescaled accumulator
o[D]

  python examples/flash_attention.py 1024 64
"""
import math, sys, time
import numpy as np
from uopc.uop import UOp, Ops, ParamArg, AxisType, sink, range_
from uopc.dtype import dtypes
from uopc.render_c import render_c
from uopc.runtime import Program
from uopc.rangeify import lower, Ctx, Opts
from uopc.tensor import Tensor

LOG2E = 1 / math.log(2)


def attention(q: Tensor, k: Tensor, v: Tensor) -> Tensor:
  return (q @ k.T * (1 / math.sqrt(q.shape[-1]))).softmax(-1) @ v


def online_attention_ir(N: int, D: int) -> UOp:
  """FlashAttention-style online softmax as a UOp loop program. One query row per thread."""
  f32 = dtypes.f32
  Q, K, V, OUT = (UOp(Ops.BUFFER, (), ParamArg(s, f32, (N * D,))) for s in range(4))
  alloc = lambda slot, n: UOp(Ops.ALLOC, (), ParamArg(slot, f32, (n,)))
  M, L, O, S = alloc(0, 1), alloc(1, 1), alloc(2, D), alloc(3, 1)
  scale = (1 / math.sqrt(D)) * LOG2E                    # work in base 2: exp(x) = exp2(x *
log2 e)

  i = range_(N, 0, AxisType.THREAD)
  Mi, Li, Oi = M.after(i), L.after(i), O.after(i)
  init_m = Mi.index(0).store(UOp.const(-math.inf))
  init_l = Li.index(0).store(UOp.const(0.0))
  d0 = range_(D, 1)
  init_o = Oi.index(d0).store(UOp.const(0.0)).end(d0)

  j = range_(N, 2)
  # s = (q_i . k_j) * scale
  Sj = S.after(i, j)
  init_s = Sj.index(0).store(UOp.const(0.0))
```

```
  e = range_(D, 3)
  Se = Sj.after(init_s, e)
  dot = Se.index(0).store(Se.index(0).load() + Q.index(i * D + e).load() * K.index(j * D +
e).load()).end(e)
  s = Sj.after(dot).index(0).load() * scale
  # online softmax update
  Mj, Lj = Mi.after(init_m, j), Li.after(init_l, j)
  m_old = Mj.index(0).load()
  m_new = m_old.maximum(s)
  corr = UOp(Ops.EXP2, (m_old - m_new,))                # rescale what we accumulated so far
  p = UOp(Ops.EXP2, (s - m_new,))
  st_l = Lj.index(0).store(Lj.index(0).load() * corr + p)
  d = range_(D, 4)
  Od = Oi.after(init_o, j)
  st_o = Od.index(d).store(Od.index(d).load() * corr + p * V.index(j * D + d).load()).end(d)
  st_m = Mj.index(0).store(m_new)
  loop_j = UOp(Ops.SINK, (st_l, st_o, st_m)).end(j)
  # normalize
  d2 = range_(D, 5)
  out = OUT.index(i * D + d2).store(Oi.after(loop_j).index(d2).load() *
Li.after(loop_j).index(0).load().recip())
  return sink(out.end(d2, i))


class FlashOpt:
  """fuse_reduce kernels: unroll the head dim (up to 16) so each QK^T value is shared by 16
accumulators."""
  from uopc.opt import layout_pass as layout
  @staticmethod
  def lower(body, out):
    return lower(body, out, Ctx(opts=Opts(upcast={len(body.shape) - 1: min(body.shape[-1],
16)}, thread=0)))


def bench(fn, reps=3):
  fn(); best = math.inf
  for _ in range(reps):
    t0 = time.perf_counter(); fn(); best = min(best, time.perf_counter() - t0)
  return best


if __name__ == "__main__":
  N = int(sys.argv[1]) if len(sys.argv) > 1 else 1024
  D = int(sys.argv[2]) if len(sys.argv) > 2 else 64
  rng = np.random.default_rng(0)
  q, k, v = (rng.standard_normal((N, D)).astype(np.float32) for _ in range(3))
  s = q @ k.T / math.sqrt(D)
  p = np.exp(s - s.max(-1, keepdims=True))
  ref = (p / p.sum(-1, keepdims=True)) @ v

  def run_tensor():
    return attention(Tensor(q), Tensor(k), Tensor(v)).numpy()
  t = bench(run_tensor)
```

```
  print(f"unfused  {t*1e3:8.2f} ms  err {np.abs(run_tensor() - ref).max():.1e}
(materializes {N*N*4/1e6:.1f} MB of scores)")

  Tensor.fuse_reduce, old = True, Tensor.opt
  Tensor.opt = FlashOpt
  t = bench(run_tensor)
  print(f"fused    {t*1e3:8.2f} ms  err {np.abs(run_tensor() - ref).max():.1e}   (no N*N
buffer, QK^T recomputed)")
  Tensor.fuse_reduce, Tensor.opt = False, old

  prg = Program(render_c(online_attention_ir(N, D), "flash"), "flash")
  out = np.zeros(N * D, np.float32)
  t = bench(lambda: prg(q, k, v, out))
  print(f"online   {t*1e3:8.2f} ms  err {np.abs(out.reshape(N, D) - ref).max():.1e}   (one
pass, running max/sum)")
```

examples/gpt.py · 106 lines

```python
"""Weeks 9-10: train a GPT on Tiny Shakespeare with our compiler, then sample from it.

  python examples/gpt.py                    # small model, a few minutes on a laptop CPU
  python examples/gpt.py --steps 2000 --layers 4 --dim 128

Everything -- embeddings, causal attention, LayerNorm, GELU, AdamW, the backward pass --
is graph building on the Tensor API. Every kernel it runs was generated by uopc.
"""
import argparse, math, sys, time
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent))
import numpy as np
from datasets import shakespeare
from uopc.tensor import Tensor
from uopc.nn import Linear, LayerNorm, Embedding, AdamW, parameters


class Attention:
  def __init__(self, dim: int, heads: int, ctx: int):
    self.qkv, self.proj, self.heads = Linear(dim, 3 * dim), Linear(dim, dim), heads
    mask = np.triu(np.full((ctx, ctx), -1e9, np.float32), 1)       # -inf above the
diagonal: no peeking
    self.mask = Tensor(mask)

  def __call__(self, x: Tensor) -> Tensor:
    B, T, C = x.shape
    H, D = self.heads, C // self.heads
    qkv = self.qkv(x).reshape(B, T, 3, H, D)
    q, k, v = (qkv[:, :, i].permute(0, 2, 1, 3) for i in range(3))   # (B, H, T, D)
    att = (q @ k.transpose()) * (1 / math.sqrt(D)) + self.mask[:T, :T].reshape(1, 1, T, T)
    y = att.softmax(-1) @ v                                          # (B, H, T, D)
    return self.proj(y.permute(0, 2, 1, 3).reshape(B, T, C))


class Block:
  def __init__(self, dim, heads, ctx):
    self.ln1, self.attn, self.ln2 = LayerNorm(dim), Attention(dim, heads, ctx),
LayerNorm(dim)
    self.fc, self.out = Linear(dim, 4 * dim), Linear(4 * dim, dim)
  def __call__(self, x: Tensor) -> Tensor:
    x = x + self.attn(self.ln1(x))
    return x + self.out(self.fc(self.ln2(x)).gelu())


class GPT:
  def __init__(self, vocab, ctx, dim, heads, layers):
    self.tok, self.pos = Embedding(vocab, dim), Tensor.randn(ctx, dim) * 0.02
    self.pos = self.pos.realize(); self.pos.requires_grad = True
    self.blocks = [Block(dim, heads, ctx) for _ in range(layers)]
    self.ln, self.head = LayerNorm(dim), Linear(dim, vocab)
  def __call__(self, idx: Tensor) -> Tensor:
    B, T = idx.shape
```

```
    x = self.tok(idx) + self.pos[:T].reshape(1, T, -1)
    for b in self.blocks: x = b(x)
    return self.head(self.ln(x))


def main():
  ap = argparse.ArgumentParser()
  ap.add_argument("--steps", type=int, default=600)
  ap.add_argument("--batch", type=int, default=16)
  ap.add_argument("--ctx", type=int, default=64)
  ap.add_argument("--dim", type=int, default=64)
  ap.add_argument("--heads", type=int, default=4)
  ap.add_argument("--layers", type=int, default=2)
  ap.add_argument("--lr", type=float, default=3e-3)
  a = ap.parse_args()

  text = shakespeare()
  chars = sorted(set(text))
  stoi = {c: i for i, c in enumerate(chars)}
  data = np.array([stoi[c] for c in text], np.int32)
  split = int(0.9 * len(data))
  train, val = data[:split], data[split:]
  rng = np.random.default_rng(0)
  def batch(src):
    ix = rng.integers(0, len(src) - a.ctx - 1, a.batch)
    return Tensor(np.stack([src[i:i + a.ctx] for i in ix])), Tensor(np.stack([src[i + 1:i +
a.ctx + 1] for i in ix]))

  Tensor.manual_seed(0)
  model = GPT(len(chars), a.ctx, a.dim, a.heads, a.layers)
  params = parameters(model)
  print(f"{sum(p.numel() for p in params) / 1e3:.1f}k parameters, vocab {len(chars)}")
  opt = AdamW(params, lr=a.lr, wd=0.01)
  t0 = time.perf_counter()
  for step in range(a.steps):
    x, y = batch(train)
    loss = model(x).reshape(-1, len(chars)).cross_entropy(y.reshape(-1))
    opt.zero_grad()
    loss.backward(opt.params)
    opt.step(loss)
    if step % 50 == 0 or step == a.steps - 1:
      vx, vy = batch(val)
      vloss = model(vx).reshape(-1, len(chars)).cross_entropy(vy.reshape(-1)).item()
      print(f"step {step:5d}  train {loss.item():.3f}  val {vloss:.3f}
{(time.perf_counter() - t0) / (step + 1) * 1e3:6.0f} ms/step")

  # sample: feed the last ctx characters, draw the next one from the softmax
  idx = [stoi["\n"]]
  for _ in range(300):
    ctx = idx[-a.ctx:]
    ctx = ctx + [0] * (a.ctx - len(ctx))                          # fixed shape -> one
compiled graph
    logits = model(Tensor(np.array([ctx], np.int32))).numpy()[0, min(len(idx), a.ctx) - 1]
```

```
    p = np.exp(logits - logits.max()); p /= p.sum()
    idx.append(int(rng.choice(len(chars), p=p)))
  print("".join(chars[i] for i in idx))


if __name__ == "__main__": main()
```

D.4 Project files

pyproject.toml · 22 lines

```
[project]
name = "uopc"
version = "0.1.0"
description = "A machine learning compiler built from scratch, one week at a time"
requires-python = ">=3.11"
dependencies = ["numpy"]

[project.optional-dependencies]
dev = ["pytest", "markdown", "pygments"]

[build-system]
requires = ["setuptools>=61"]
build-backend = "setuptools.build_meta"

[tool.setuptools]
packages = ["uopc", "minicc", "tensorjit", "labs", "labs.kernels"]

[tool.setuptools.package-data]
"labs.kernels" = ["*.cuh"]

[tool.pytest.ini_options]
testpaths = ["tests"]
```

20. The Syllabus Contract and Completion Gates

The original course is ambitious: build a compiler, optimize it, and train modern language models in one semester. This self-study edition uses that specification as its core and adds prerequisite and advanced labs. Reading a chapter, passing a correctness test, and demonstrating competitive performance are three different achievements. Keep an evidence ledger for each.

The source specification is the supplied Compilers for Machine Learning syllabus. The table below follows its week 4–5 ordering. The existing narrative teaches general movement ops before kernelization; this is a teaching-order difference. When following the course strictly, restrict the week-4 student interface to rank below two and unlock general shapes in week 5. `uopc` itself is a cumulative reference implementation and does not enforce weekly feature gates.

The syllabus is a minimum. Chapters 26–29 extend the track into numerical analysis, compiler-driven scheduling, direct backends, irregular workloads, hardware codesign, production reliability, distributed training, and research. The final chapter defines specialization paths and portfolio gates for frontier-lab engineering roles.

20.1 Requirement-to-evidence map

Syllabus requirement	Teaching	Evidence to produce
Hash-consed UOps; recursive dtype and shape	Chapters 1–3	Identity, type-error, and shape inference tests
PARAM/CALL with scoped parameters	Chapters 1–2	Nested-call evaluation without scope capture
CONST/STACK/INDEX; CAST/RECIPROCAL; ADD/MUL/CMPLT; FLOORDIV/FLOORMOD; WHERE	Chapters 1–3, Appendix A	Constant evaluation, negative division, casts, and selection tests
dtype broadcasting	Section 20.2	Explicit promotion contract and mixed-type cases
GLOBAL/ALU; BUFFER/ALLOC/LOAD/STORE/AFTER	Chapter 4	Typed memory operations and ordered load/store tests
C rendering, compilation, runtime, memory management	Chapter 4	Execute a generated shared library and compare outputs
RANGE/END; GEMM and convolution loops	Chapters 5–6	Rectangular GEMM and convolution versus independent references
Rank < 2 week-4 subset; REDUCE/STAGE; implicit loads	Chapters 6–7; section 20.3	Vector reduction, explicit materialization, recomputation accounting
STACK/general movement ops; rank >= 2; PAD/FLIP composites	Chapters 6 and 9	Index maps and noncontiguous/view composition tests
EXP2/LOG2 and extensible elementwise ops	Chapters 3, 8–9; Appendix A	Evaluation, rendering, and derivative tests
Autodiff only on pure shape graphs	Chapter 8	Shared-use accumulation and unsupported-effect rejection
MNIST training	Chapter 9	Reproducible loss and held-out accuracy

Syllabus requirement	Teaching	Evidence to produce
FLOPs, memory hierarchy, threads, upcasting	Chapter 10	Optimization ablations and a machine roofline
CPU GEMM, reductions, convolution; FlashAttention	Chapters 10–11 and 22	Separate workload tables; attention correctness and memory use
GPU kernels, axis mappings, accelerators, tensor cores	Chapters 12–13 and 22	Real-device correctness, synchronization checks, measured performance
CUDA/HIP competitive with PyTorch	Sections 20.4 and 22.6	Matched backend, precision, shapes, and timing protocol
Real LLM training; symbolic execution	Chapters 14–15 and 23	Training record plus variable-shape execution and cache behavior
Implement a paper or hardware port	Chapters 15 and 25	Reproduction or port with correctness and performance report
Frequent quizzes, correctness, style, performance	Appendix C and Chapter 25	Closed-book explanations, independent tests, code review, benchmark ledger

20.2 Dtype broadcasting is a semantic decision

The supplied syllabus requests dtype broadcasting without specifying its complete promotion table. The reference IR instead requires equal operand dtypes for many arithmetic operations, with explicit CAST nodes. Its tests intentionally reject `i32 + f32`. Tensor shape broadcasting is a separate rule. This is a documented difference from the requested interface, not evidence that mixed arithmetic works.

To implement promotion, first write a table for every supported pair. A possible teaching policy is `bool < i32 < f32`, promoting arithmetic operands to the higher type, while keeping comparisons boolean. This policy is a project choice, not an inferred official oracle rule. Specify integer division, boolean arithmetic, f16 interactions, and float-to-int conversion independently. Insert casts at a single boundary so the low-level IR

retains simple equal-type invariants. Test the table exhaustively, including `i32(16777217)` promoted to f32, where representability changes the value.

Gate: either implement and test the course's agreed promotion policy or mark this requirement incomplete. A shape-broadcasting test cannot satisfy the dtype requirement.

20.3 Materialization before general movement ops

In the week-4 subset, compile `sum(x*x)` and `sum(x*x) + sum(x*x)` for one-dimensional x. First inline all work, then introduce a STAGE boundary and count loads, multiplies, and temporary bytes. A stage can save repeated compute while adding an allocation and memory traffic. It is not always an improvement.

After this works, unlock general movement ops and derive matrix multiplication with broadcasted products and a reduction. Follow Chapter 6's index maps to avoid materializing the full `(M,N,K)` product. Verify that implicit loads lower to explicit memory reads before rendering.

Gate: show generated loops for both the restricted vector case and the general matrix case. Explain where the temporary exists and why its store happens before any dependent load.

20.4 Competitive performance is a measured claim

“SOTA-competitive” and “torch-competitive” are course targets, not properties implied by correct code. Choose comparison thresholds before tuning and publish every workload, including regressions. A credible table includes small and large GEMMs, rectangular shapes, reductions, and convolutions; one favorable square GEMM is insufficient. Match input storage, accumulation, numerical policy, layout, device, and thread count. Compare steady-state device execution separately from end-to-end latency including transfers and compilation.

The reference book's CPU measurements are examples from its recorded machine. GPU render tests and emulation establish narrower properties. The workspace may include real-device tests in `tests/test_w08_cuda.py`; their existence does not establish that they ran or passed on your hardware. HIP support is a backend project; emitting CUDA does not demonstrate AMD execution.

Gate: retain raw timing samples, commands, versions, tolerances, profiler captures, and generated kernels. Mark hardware-dependent results “not run” when hardware is unavailable.

20.5 A small GPT and a modern LLM milestone

Chapter 14's small character GPT teaches the training pipeline. To meet a larger language-model milestone, specify the architecture, parameter count, tokenizer, data, context length, precision, optimizer, checkpoint format, and available compute before starting. Verify a tiny version first. Load compatible pretrained weights for a forward comparison if the project permits it; this checks integration but does not replace training.

Scale gradually and record tokens per second, peak memory, compile time, loss, and validation quality. Save optimizer state and random-number state as well as weights; compare a resumed run against an uninterrupted run. A short training run on a larger architecture demonstrates execution, not state-of-the-art model quality. Set the intended training and quality criterion explicitly.

Gate: reproduce the tiny training run, pass forward and gradient checks, demonstrate checkpoint resume, then report the largest configuration actually trained. Never relabel the character GPT as SOTA LLM training.

20.6 Completion ledger

For each row in section 20.1, record `explained`, `implemented`, `correctness verified`, and `performance verified` separately, with a commit, command, and artifact. Chapters 21–29 provide the advanced learning path; completing their labs requires additional implementation. The book teaches the route and acceptance criteria without claiming the reference compiler already fulfills every target.

21. Control Flow, SSA, Analysis, and Lowering

The UOp graph makes pure tensor expressions easy to share and rewrite. A general compiler also needs to represent branches, loops, mutable memory, and facts that hold only on particular paths. Build a small scalar compiler beside `uopc` to learn these ideas without complicating its tensor IR immediately.

21.1 From syntax to a control-flow graph

Start with integer expressions, variables, assignment, `if`, `while`, and `return`. A lexer turns text into tokens. A parser groups tokens into an abstract syntax tree, respecting precedence; `a+b*c` means `a+(b*c)`. Name resolution binds each variable use to a declaration, and type checking rejects invalid operations before lowering.

A basic block is a sequence of instructions with a single entry and a terminating transfer of control. Lower `if p: x=a+1; else: x=a-1; return x*2` into:

```
entry: branch p, then, else
then:  x_then = a + 1; jump merge
else:  x_else = a - 1; jump merge
merge: x = phi [x_then, then], [x_else, else]; return x * 2
```

The phi selects the value from the predecessor that actually executed. It is not a normal operation that evaluates both branches. Verify that every block ends with a terminator, every target exists, and every phi has an incoming value for each predecessor.

Lab: implement an interpreter and a CFG dumper. Test nested branches, zero-trip loops, and a loop-carried accumulator. Define whether uninitialized variables are rejected and test the rule.

21.2 Dominance and SSA

A block d dominates b if every path from entry to b passes through d. On reachable blocks, initialize `Dom(entry)={entry}` and other dominator sets to all reachable blocks. Iterate:

```
Dom(b) = {b} union intersection(Dom(p) for p in predecessors(b))
```

until nothing changes. Unreachable blocks need separate handling. In static single assignment form, each name has one definition, and ordinary uses must be dominated by their definition. Phi operands are uses on incoming edges, so their dominance check is edge-specific.

For the example above, `then` does not dominate `merge`: execution can arrive through `else`. Merely using `x_then` in merge would be invalid. A loop's phi joins an initial value from the preheader with an updated value from the backedge.

Lab: compute dominators, place phis for a small mutable-variable language, and rename definitions. Convert an accumulator loop and verify its results against the source interpreter. Add an intentionally invalid use and ensure the verifier rejects it. Explain where the tensor DAG avoids these issues.

21.3 Dataflow and effects

Liveness answers whether a computed value may be used later. For each block:

```
OUT[b] = union(IN[s] for s in successors(b))
IN[b]  = USE[b] union (OUT[b] - DEF[b])
```

Solve backward to a fixed point. Count a value as USE only if it is used before a definition in that block. In SSA, account for phi uses on predecessor edges. Liveness later determines when a runtime can reuse a buffer or a backend can reuse a register.

Dead-code elimination removes an unused pure add. It cannot remove a store just because the store has no ordinary result. Memory effects require ordering and alias reasoning. In `a[0]=1; b[0]=2; return a[0]`, the return is 2 when a and b alias and 1 when they do not. Hoisting the load before the second store is valid only with proof that the relevant addresses do not overlap. `restrict` in generated C is a promise the compiler must actually uphold.

Lab: implement liveness and effect-aware dead-code elimination. Test both aliased and disjoint inputs. State which transformations require no-alias assumptions and ensure the runtime enforces them.

21.4 Loops, legality, and machine code

Loop interchange and tiling change execution order. Before choosing a profitable schedule, prove that it preserves dependencies. `a[i]=a[i-1]+1` carries a dependence across iterations and cannot be parallelized as independent work. A reduction may be parallelizable under a declared floating-point policy, but its rounding can change.

Machine lowering introduces another set of constraints: legal instruction types, register classes, calling conventions, and finite registers. Instruction selection maps IR operations to target instructions. Register allocation maps live values to registers; excess live values spill to memory. Unrolling can reduce loop overhead while increasing live values and spills. C rendering delegates these tasks to the host compiler; a direct backend must handle them itself.

Lab: build an interference graph from live scalar values and implement a small register allocator with a fixed register budget. Report spills before and after unrolling. Separately inspect compiler assembly for a generated C kernel and identify loads, arithmetic, branches, and vector instructions.

21.5 LLVM and MLIR bridge

Use the official LLVM frontend tutorial for parsing, LLVM IR generation, control flow, and JIT integration. Port one expression and one loop from your scalar language. Ask LLVM's verifier to check the module before optimization.

Use the MLIR Toy tutorial to study operations, dialects, shape inference, rewrites, and progressive lowering. Translate a small tensor expression into an appropriate structured representation, then lower toward loops and a backend. Record the IR after every pass. Pin the LLVM/MLIR revision because APIs and pass names evolve.

A tensor value describes logical elements; a memory reference adds storage, layout, and aliasing. Bufferization must decide when an output may reuse existing storage. A use of the old value after an update can force a copy. Compare that decision with Chapter 7's materialization boundaries.

Exit gate: provide a verified CFG/SSA compiler, an effect-safe optimization, and a runnable LLVM or MLIR example. Given a wrong output, identify the first pass whose output changes semantics.

21.6 Worked lab: why phi operands live on edges

Run the executable analysis from the repository root:

```
.venv/bin/python -m labs.ssa
```

The diamond has a parameter x, a branch condition p, a left result a, a right result b, and a merge phi z. The entry dominates both branches and the merge. Neither branch dominates the merge, which is why the merge cannot simply use a. The phi is the only place the two branch results meet.

The liveness calculation should produce `OUT(left)={a}` and `OUT(right)={b}`. If both a and b appear live out of both branches, the analysis has treated phi arguments as ordinary merge-block uses. That creates impossible interference and can force unnecessary register pressure. At the merge itself, z is already defined by its phi, so z is not live into the block.

Here is the complete implementation. It accepts already constructed SSA and verifies it; it does not yet parse source or insert phis automatically. Unreachable blocks are excluded from dominance and ordinary-use checks. Phi incoming edges still match the complete predecessor set.

`labs/ssa.py`

```python
"""Reachability, dominators, liveness, and SSA verification on a branching CFG.

Run: python -m labs.ssa
Phi operands are uses on predecessor edges, not ordinary uses in the merge block.
"""
from dataclasses import dataclass


@dataclass(frozen=True)
class Instruction:
  result: str | None
  args: tuple[str, ...] = ()


@dataclass(frozen=True)
class Block:
  instructions: tuple[Instruction, ...]
  successors: tuple[str, ...] = ()
  # (result, ((predecessor, value), ...))
  phis: tuple[tuple[str, tuple[tuple[str, str], ...]], ...] = ()


def graph(blocks: dict[str, Block], entry: str):
  if entry not in blocks: raise ValueError("missing entry")
  predecessors = {b: set() for b in blocks}
  for name, block in blocks.items():
    for successor in block.successors:
      if successor not in blocks: raise ValueError(f"unknown successor: {successor}")
      predecessors[successor].add(name)
  reachable, todo = set(), [entry]
  while todo:
    name = todo.pop()
    if name in reachable: continue
    reachable.add(name)
    todo.extend(blocks[name].successors)
  return predecessors, reachable


def dominators(blocks: dict[str, Block], entry: str):
  predecessors, reachable = graph(blocks, entry)
  dom = {b: ({b} if b == entry else set(reachable)) for b in reachable}
  changed = True
  while changed:
    changed = False
    for b in sorted(reachable - {entry}):
      incoming = [dom[p] for p in predecessors[b] & reachable]
      value = {b} | set.intersection(*incoming)
      if value != dom[b]: dom[b], changed = value, True
  return dom


def verify(blocks: dict[str, Block], entry: str, parameters: tuple[str, ...] = ()):
```

```
  predecessors, reachable = graph(blocks, entry)
  dom = dominators(blocks, entry)
  definitions = {}
  for b in reachable:
    for name, incoming in blocks[b].phis:
      if len(incoming) != len(dict(incoming)) or set(dict(incoming)) != predecessors[b]:
        raise ValueError(f"phi {name}: incoming edges do not match predecessors")
      if name in parameters or name in definitions: raise ValueError(f"duplicate definition:
{name}")
      definitions[name] = (b, -1)
    for i, instruction in enumerate(blocks[b].instructions):
      if (name := instruction.result) is not None:
        if name in parameters or name in definitions: raise ValueError(f"duplicate definition:
{name}")
        definitions[name] = (b, i)

  def use(value, block, position):
    if value in parameters: return
    if value not in definitions: raise ValueError(f"undefined value: {value}")
    definition_block, definition_position = definitions[value]
    if definition_block not in dom[block] or (definition_block == block and definition_position
>= position):
      raise ValueError(f"definition of {value} does not dominate its use in {block}")

  for b in reachable:
    for _, incoming in blocks[b].phis:
      for predecessor, value in incoming:
        if predecessor in reachable: use(value, predecessor,
len(blocks[predecessor].instructions))
    for i, instruction in enumerate(blocks[b].instructions):
      for value in instruction.args: use(value, b, i)


def liveness(blocks: dict[str, Block], entry: str):
  _, reachable = graph(blocks, entry)
  uses, definitions = {}, {}
  for b in reachable:
    defined = {name for name, _ in blocks[b].phis}
    used = set()
    for instruction in blocks[b].instructions:
      used.update(set(instruction.args) - defined)
      if instruction.result is not None: defined.add(instruction.result)
    uses[b], definitions[b] = used, defined
  live_in, live_out = {b: set() for b in reachable}, {b: set() for b in reachable}
  changed = True
  while changed:
    changed = False
    for b in sorted(reachable, reverse=True):
      outgoing = set()
      for successor in blocks[b].successors:
        phi_defs = {name for name, _ in blocks[successor].phis}
        edge_uses = {dict(incoming)[b] for _, incoming in blocks[successor].phis}
        outgoing |= (live_in[successor] - phi_defs) | edge_uses
```

```
      incoming = uses[b] | (outgoing - definitions[b])
      if incoming != live_in[b] or outgoing != live_out[b]:
        live_in[b], live_out[b], changed = incoming, outgoing, True
  return live_in, live_out


def diamond():
  return {
    "entry": Block((Instruction(None, ("p",)),), ("left", "right")),
    "left": Block((Instruction("a", ("x",)),), ("merge",)),
    "right": Block((Instruction("b", ("x",)),), ("merge",)),
    "merge": Block((Instruction("y", ("z",)), Instruction(None, ("y",))),
                   phis=(("z", (("left", "a"), ("right", "b"))),)),
  }


def main():
  blocks = diamond()
  verify(blocks, "entry", ("p", "x"))
  for name, values in sorted(dominators(blocks, "entry").items()):
    print(f"Dom({name}) = {','.join(sorted(values))}")
  incoming, outgoing = liveness(blocks, "entry")
  for name in sorted(blocks): print(f"{name}: IN={sorted(incoming[name])}, OUT=
{sorted(outgoing[name])}")


if __name__ == "__main__": main()
```

Experiment: change the merge's ordinary use from z to a. The verifier must report that a's definition does not dominate its use. Then replace one phi incoming edge with a nonexistent branch. That must fail before liveness analysis. Finally construct a loop with an initial parameter and a backedge value, and show why a header phi may legally use the later-written backedge value.

Next implementation: add AST lowering, dominance frontiers, phi placement, and renaming. Validate against a source interpreter. The executable lab supplies analysis and verification, not that entire frontend. Use `tests/test_advanced_labs.py` as a starting example of independent semantic checks.

21.7 Continue into a complete executable pipeline

The isolated analysis above is now extended by Chapters 30–32. They implement a source frontend, CFG construction, definite assignment, pruned SSA construction, SCCP, effect-aware dead-code removal, phi destruction, register allocation with executed spills, and a bounds-checked native C backend. Run

`.venv/bin/python -m minicc` to execute the pipeline. The earlier “next implementation” assignment can now be studied against an actual reference implementation.

22. GPU Engineering: Correctness, Throughput, and Attention

Chapters 12–13 explain GPU mapping and shared-memory kernels. This chapter turns those ideas into a hardware investigation. Every performance result here is an assignment to measure on your GPU; there are no implied benchmark results. Start with the architecture sections of the official CUDA Programming Guide and record the GPU model, supported compute capability, compiler, and driver used for your experiment.

22.1 Execution, memory, and synchronization

A block contains threads that execute in warps. Blocks may execute in any order; a conventional kernel cannot depend on a different block reaching a software barrier. Use separate launches or a supported cooperative mechanism when global coordination is required.

A block barrier requires participating threads to reach it consistently. This is unsafe when the last block is partial:

```
if global_index < n:
    shared[local_index] = input[global_index]
    block_barrier()
```

Move the barrier outside the divergent condition and initialize invalid lanes to the operation's identity. For a sum use zero; for a maximum use negative infinity. Ensure consumers cannot read an uninitialized slot. Warp-level primitives have their own active-mask and participation requirements; lockstep intuition is not a substitute for the specified synchronization contract.

Lab: implement vector addition, a block reduction, and a multi-block reduction. Test lengths 1, 31, 32, 33, and 1003. Run the available memory, race, and synchronization checking tools on real hardware. Keep a deliberately broken version to demonstrate which check catches the error.

22.2 Coalescing and shared-memory banks

For a row-major matrix, adjacent threads loading adjacent columns typically access nearby addresses. If adjacent threads instead stride across rows, the warp may need many more memory transactions. Logical bytes read and physical bytes transferred can therefore differ substantially.

A tiled transpose loads contiguous rows into shared memory and stores contiguous rows of the transposed result. Shared-memory layout can create bank conflicts. Under the common teaching model of 32 banks and 4-byte words, bank index is `(byte_address/4) mod 32`; a 32×32 float tile accessed by column maps those accesses to the same bank. Padding its row stride to 33 spreads those accesses. Check the actual architecture's rules, access width, and broadcast behavior before generalizing.

Lab: compare naive transpose, tiled transpose, and padded tiled transpose. Handle rectangular sizes and edge tiles. Report useful bandwidth `2*M*N*sizeof(element)/time` and relevant profiler transaction and bank-conflict evidence. Explain any mismatch between predicted and measured behavior.

22.3 The GEMM optimization ladder

For a block output tile `BM × BN` and reduction chunk BK, load `BM*BK + BK*BN` elements into shared memory, then perform approximately `2*BM*BN*BK` operations. With f32 inputs, the idealized input arithmetic intensity per chunk is:

```
2*BM*BN*BK / (4*(BM*BK + BK*BN)) FLOPs per byte
```

This ignores output traffic, cache effects, and repeated block loads, so it is a model to check. Each thread can retain several output accumulators in registers. That increases reuse but also register demand. Shared memory per block, registers per thread, thread count, and architectural limits jointly constrain residency. High occupancy alone does not imply high throughput.

Build and retain each rung: naive global loads; shared-memory tiling; per-thread register tiles; vectorized loads where alignment permits; tensor-core operations; pipelined memory transfer. For each rung, measure correctness, time, register use, shared-memory use, spills, and occupancy. A larger tile that spills can lose to a smaller tile with less nominal reuse.

For tensor cores, match supported tile shapes, storage and accumulator types, layouts, alignment, and warp participation rules. WMMA fragments are implementation-defined distributed register representations; never assume each thread owns a simple contiguous subsection of the matrix. Compare f16-input/f32-accumulate output against a reference using the same quantized inputs.

Lab: benchmark square and rectangular GEMMs with aligned and irregular dimensions. For each optimization predict its benefit, then inspect generated code and profiler evidence. Demonstrate at least one configuration where an apparently stronger optimization loses and explain why.

22.4 Asynchronous pipelines

A sequential tile loop loads tile t, synchronizes, computes it, then starts t+1. A double-buffered pipeline can overlap loading t+1 with computing t. It needs separate storage for tiles in flight, a completion mechanism before consumption, and protection against reusing storage too early.

Write a timeline before code:

```
prologue: begin load tile 0
iteration t: wait tile t; begin load t+1 into other buffer; compute tile t
boundary: ensure all readers finished before reusing tile t's buffer
```

Exact wait and barrier operations depend on the target instruction and architecture. A block barrier does not automatically replace an asynchronous-copy completion protocol. More stages consume more shared memory, potentially reducing residency enough to remove the benefit.

Lab: add a supported asynchronous-copy pipeline to your tiled GEMM. Validate the final partial K tile, buffer reuse, and all synchronization paths. Compare two and three stages across several K sizes. If the hardware lacks the feature, retain the design and mark the device experiment incomplete.

22.5 FlashAttention beyond the formula

For one query row, process scores in tiles and maintain running maximum m, normalizer l, and unnormalized weighted-value accumulator o. Starting with `m=-infinity, l=0, o=0`, a new score tile S

updates:

```
m_new = max(m, max(S))
p = exp(S - m_new)
alpha = exp(m - m_new)
l_new = alpha*l + sum(p)
o_new = alpha*o + p @ V_tile
output = o / l
```

Rescaling old state is what makes the tiled result match a stable full softmax in real arithmetic. Floating-point reduction order still changes rounding. Causal masking must exclude future positions before they contribute to maxima and sums. Fully masked rows require explicit semantics to avoid `-infinity - -infinity` producing NaN.

For backward, with `P=softmax(S)` and output `O=P*V`, compute `dV=P^T*dO`, `dP=dO*V^T`, and `dS=P*(dP-row_sum(dP*P))`. Then apply the score scale when forming dQ and dK. A memory-efficient backward recomputes needed probabilities from scores and saved row statistics instead of storing all of P. Its scheduling and race-free gradient accumulation are substantial implementation work.

Lab: implement forward and backward against an unfused reference. Test causal and noncausal attention, irregular sequence lengths, several head dimensions, and extreme logits. Run finite differences on tiny double-precision cases. Measure peak memory and latency as sequence length grows; report both forward and backward. Add dropout only after defining reproducible masks.

22.6 Framework comparison and backend portability

Use the official Triton tutorials as a second implementation route for vector operations, reductions, matrix multiplication, and attention. Compare the schedule expressed by a block program with the thread and shared-memory schedule your CUDA renderer emits. Read the intermediate representations for one kernel.

For benchmarks, warm up compilation and caches, measure repeated device work with synchronization or device events, and report median plus spread. Host wall time without waiting often measures enqueue latency. Keep transfers and cold compilation as separate end-to-end measurements. Match PyTorch's precision settings and verify that the compared operation performs equivalent work.

For HIP or another backend, list the portable semantics and the architecture-specific assumptions: subgroup width, matrix instructions, memory spaces, barriers, launch limits, and runtime APIs. A textual rename of

CUDA builtins is insufficient. Port a reduction before attempting tensor cores.

Exit gate: a correctness suite on real hardware, a transpose and GEMM optimization notebook, an attention forward/backward implementation, and a workload-wide comparison. Competitive performance is earned separately for each tested workload and device.

22.7 Worked lab: derive and execute tiled attention backward

Let scores be `S=Q*K^T/sqrt(d)` and probabilities `P=softmax(S)`. For a loss gradient G on `O=P*V`, the chain rule gives `dV=P^T*G` and `dP=G*V^T`. A single softmax row has Jacobian `diag(P)-P*P^T`, so its vector-Jacobian product is `dS=P*(dP-sum(dP*P))`.

The row correction can be computed without materializing dP or P:

```
sum_j dP[i,j]*P[i,j]
= sum_j sum_c G[i,c]*V[j,c]*P[i,j]
= sum_c G[i,c]*O[i,c]
```

That identity lets backward save output O and one log-normalizer per query row, then recompute each probability tile as `exp(scores-log_normalizer)`. For a score tile of width T, temporary score and probability storage is proportional to `queries*T`, rather than `queries*keys`. Q, K, V, output, and gradients still occupy memory: the algorithm removes quadratic score storage, not all storage.

For a scalar online-softmax example, process scores 1000 and 1001 in separate tiles. After the first tile, m=1000 and l=1. After the second, m=1001 and l=`exp(-1)+1`. The first contribution to the weighted-value sum is also rescaled by `exp(-1)`. Forgetting that rescaling produces wrong results even though the normalizer looks stable.

Run the implementation and its numerical checks:

```
.venv/bin/python -m labs.attention
.venv/bin/python -m pytest tests/test_advanced_labs.py -q -k attention
```

`labs/attention.py`

```python
"""Tiled online attention and recomputing backward, using NumPy as a CPU lab.

Single head, Q:(queries,dim), K:(keys,dim), V:(keys,value_dim).
Causal queries use absolute positions starting at query_offset. Fully masked rows
produce zero output, -inf log-normalizer, and zero gradients by explicit policy.
Run: python -m labs.attention
"""
import numpy as np


def _inputs(q, k, v, tile, query_offset):
  if q.ndim != 2 or k.ndim != 2 or v.ndim != 2 or q.shape[1] != k.shape[1] or k.shape[0] !=
v.shape[0]:
    raise ValueError("expected compatible rank-two Q, K, V")
  if min(*q.shape, *k.shape, *v.shape) < 1 or tile < 1 or query_offset < 0:
    raise ValueError("dimensions and tile must be positive; query_offset nonnegative")
  if not all(np.issubdtype(x.dtype, np.floating) and np.isfinite(x).all() for x in (q, k, v)):
    raise ValueError("this lab requires finite floating-point inputs")


def _scores(q, k_tile, start, scale, causal, query_offset):
  scores = (q @ k_tile.T) * scale
  if causal:
    mask = np.arange(start, start + len(k_tile))[None, :] > (np.arange(len(q)) + query_offset)[:,
None]
    scores = np.where(mask, -np.inf, scores)
  return scores


def online_attention(q, k, v, tile=32, causal=False, query_offset=0):
  _inputs(q, k, v, tile, query_offset)
  dtype = np.result_type(q, k, v, np.float32)
  q, k, v = (x.astype(dtype, copy=False) for x in (q, k, v))
  maximum = np.full(len(q), -np.inf, dtype=dtype)
  normalizer = np.zeros(len(q), dtype=dtype)
  output = np.zeros((len(q), v.shape[1]), dtype=dtype)
  scale = 1 / np.sqrt(q.shape[1])
  for start in range(0, len(k), tile):
    scores = _scores(q, k[start:start+tile], start, scale, causal, query_offset)
    next_maximum = np.maximum(maximum, scores.max(axis=1))
    # Zero is a harmless exponent reference for a still-fully-masked row.
    safe_maximum = np.where(np.isfinite(next_maximum), next_maximum, 0)
    alpha = np.exp(maximum - safe_maximum)
    probability = np.exp(scores - safe_maximum[:, None])
    output = alpha[:, None] * output + probability @ v[start:start+tile]
    normalizer = alpha * normalizer + probability.sum(axis=1)
    maximum = next_maximum
  output = np.divide(output, normalizer[:, None], out=np.zeros_like(output), where=normalizer[:,
None] != 0)
  log_normalizer = np.full_like(maximum, -np.inf)
  valid = normalizer > 0
  log_normalizer[valid] = maximum[valid] + np.log(normalizer[valid])
```

```
  return output, log_normalizer


def attention_backward(q, k, v, output, log_normalizer, grad_output, tile=32, causal=False,
query_offset=0):
  """Recompute probability tiles; never allocate a full queries-by-keys matrix."""
  _inputs(q, k, v, tile, query_offset)
  dtype = np.result_type(q, k, v, grad_output, np.float32)
  q, k, v = (x.astype(dtype, copy=False) for x in (q, k, v))
  if output.shape != (len(q), v.shape[1]) or grad_output.shape != output.shape or
log_normalizer.shape != (len(q),):
    raise ValueError("invalid saved state or output gradient shape")
  dq, dk, dv = np.zeros_like(q), np.zeros_like(k), np.zeros_like(v)
  correction = (grad_output * output).sum(axis=1)
  scale = 1 / np.sqrt(q.shape[1])
  valid = np.isfinite(log_normalizer)
  safe_log = np.where(valid, log_normalizer, 0)
  for start in range(0, len(k), tile):
    scores = _scores(q, k[start:start+tile], start, scale, causal, query_offset)
    probability = np.exp(scores - safe_log[:, None])
    probability[~valid] = 0
    dp = grad_output @ v[start:start+tile].T
    ds = probability * (dp - correction[:, None])
    dq += (ds @ k[start:start+tile]) * scale
    dk[start:start+tile] += (ds.T @ q) * scale
    dv[start:start+tile] += probability.T @ grad_output
  return dq, dk, dv


def main():
  rng = np.random.default_rng(0)
  q, k, v = (rng.normal(size=shape) for shape in ((7, 5), (11, 5), (11, 3)))
  output, log_normalizer = online_attention(q, k, v, tile=4, causal=True)
  gradients = attention_backward(q, k, v, output, log_normalizer, np.ones_like(output), tile=4,
causal=True)
  print("output:", output.shape, "saved row statistics:", log_normalizer.shape)
  print("gradient shapes:", *(x.shape for x in gradients))
  print("CPU algorithm lab; no GPU performance claim")


if __name__ == "__main__": main()
```

This lab is a CPU implementation of the algorithm. It supports one head, finite floating-point inputs, causal masking, and a nonnegative query-position offset for decoding-style subproblems. Its loops do not establish GPU synchronization correctness or performance. Finite inputs can still overflow a dot product at extreme magnitudes; the tested numerical domain must be reported.

Experiment: vary tile size through 1, 4, and larger than the key length. Forward results should agree within rounding error. Perturb Q, K, and V separately in random directions, comparing the dense loss finite

difference with the analytic gradient's directional inner product. The test suite uses an independent dense computation, so a shared error in tiled forward and backward cannot validate itself.

Next implementation: lower the tile loop into a real GPU kernel, then handle parallel dK/dV accumulation without races. Compare memory and device time with the dense baseline at matched precision. The serial CPU loop's safe accumulation order is not an automatic GPU scheduling solution.

22.8 Continue into measured register tiling

Chapter 35 implements a 64×64 shared-memory output tile with per-thread register accumulators, masks partial dimensions, and measures the actual CUDA kernels on an RTX 3050. It includes the kernel, event-timing harness, independent numerical checks, and raw recorded results.

23. Symbolic Shapes, Guards, Mutation, and Graph Capture

A program with a different sequence length should not necessarily require a new executable. But removing sizes from a cache key without changing the generated program is incorrect. Dynamic execution requires a consistent contract across shapes, proofs, generated loops, and dispatch.

23.1 A symbolic dimension is a constrained value

Represent a dimension N as an expression with constraints such as `1 <= N <= 4096`. Separate its identity and bounds from its current runtime binding. For a contiguous `(B,N)` tensor, address `b*N+j` includes runtime N; its stride is dynamic too. An allocation may use a maximum capacity while logical sizes remain smaller.

For `0 <= j < N`, intervals alone may say `j <= 4095` and `N >= 1`, which cannot prove `j < N`. That fact comes from the loop relation. Keep facts scoped to the region where they hold. Never reuse an inner-loop assumption outside that loop. Check overflow of index arithmetic before using identities that assume mathematical integers.

Lab: implement bounded dimensions, runtime bindings, and dynamic indexing for vector operations and one reduction. Execute lengths 1, 7, 31, and 64 with one plan. Log the actual plan count and inspect that generated loop bounds depend on runtime arguments. Reject bindings outside declared bounds.

23.2 Guards and specialization

A guard is an executable check for a compile-time assumption. Suppose a fast kernel assumes contiguous f32 input, aligned addresses, and `N % 16 == 0`. Its dispatch contract must check all relevant assumptions, then select a generic fallback when they do not hold. A shape guard alone cannot establish strides or alignment.

A useful cache key includes graph semantics, target architecture, dtypes, layout class, numerical policy, compiler version, and specialization constraints. Runtime values belong in the key only when the generated

code specializes on them. Guard failure may trigger another compiled variant; control variant growth to avoid compiling one executable per request.

Lab: build a guarded dispatcher with a generic implementation and one divisible-by-16 fast path. Call it with repeated and alternating shapes and strided inputs. Record compile count, guard failures, and which path executed. Verify every output against the eager reference. Corrupt one guard deliberately and demonstrate a test that detects the resulting wrong dispatch.

23.3 Mutation and views

For `y = view(x); x[...] = new_value; use(y)`, whether y sees the update depends on the language's aliasing semantics. A pure expression graph cannot represent this merely by rebinding the Python variable x. Introduce explicit effects or transform mutation into functional values while preserving aliases and ordering.

One approach versions memory: a store consumes memory state M0 and produces M1, and later loads consume M1. Another functionalizes mutation by generating new values and rebuilding affected views. Both require an explicit treatment of overlapping slices and external aliases.

Lab: define mutation semantics before implementation. Test disjoint slices, overlapping slices, a view surviving an assignment, and a parameter update followed by a new forward pass. Ensure host and device copies cannot silently diverge. Tie in-place reuse to an alias and liveness proof.

23.4 Capturing a host-language program

Lazy tensor APIs capture operations because every tensor method records a node. Capturing an eager host program also encounters branches, loops, scalar conversions, side effects, and unsupported calls. Tracing one branch records that execution; it does not automatically capture the other branch. A branch on tensor data needs representable control flow, a graph break, or a correct fallback.

Start with a restricted function tracer that supports tensor arithmetic and fixed control flow. Document exactly which operations are captured and reject unsupported effects rather than dropping them. Then add

shape-dependent branching with guards. General Python bytecode capture is a later project with a much larger semantic surface.

Lab: trace a two-layer model and compare eager and captured forward/gradient results. Exercise both outcomes of a shape-dependent branch. Add a tensor-data branch and demonstrate the chosen fallback. Record graph breaks and recompilations as part of the result, not only kernel time.

23.5 A true KV cache

Allocate K and V storage per layer with shape `(batch, heads, capacity, head_dim)`. At position p, write the new key and value into slot p, then attend only to `0..p`. The cache needs a length and ownership policy; reset it between independent sequences and reject capacity overflow.

A fixed-capacity masked implementation can be correct while doing work proportional to capacity on every token. A dynamic loop over the populated prefix avoids this wasted work. Its GEMM shape, parallel reduction strategy, and compilation behavior still need evaluation.

Exit gate: compare cached and uncached logits at every decoding step, verify one reusable plan where intended, and plot latency against populated length and capacity separately. Demonstrate cache reset, batch isolation, and an out-of-capacity error.

23.6 Worked lab: one C kernel for many lengths

A dynamic length can be data without changing a function's code. The lab passes N in an int32 buffer and generates a C loop whose bound reads that buffer. Contiguous f32 vectors take the compiled path; other floating-point vectors use a NumPy fallback preserving their dtype and logical layout. Lengths outside the declared bound are rejected rather than silently extending a compiled assumption.

```
.venv/bin/python -m labs.dynamic
```

Expected counters after the example are `variants built=1`, `compiled calls=5`, and `layout fallbacks=1`. “Variants built” counts Program construction in this instance; the runtime's on-disk cache can

make the actual C compiler invocation unnecessary. This distinction matters when measuring cold and warm compilation.

`labs/dynamic.py`

```python
"""Guarded C JIT: runtime length, shape bounds, and a layout fallback.

Run: python -m labs.dynamic. Requires gcc; uses the book's C runtime.
"""
import numpy as np
from uopc.runtime import Program


SOURCE = r"""
void square_plus_one(float *x, float *out, int *length) {
  for (int i = 0; i < length[0]; i++) out[i] = x[i]*x[i] + 1.0f;
}
"""


class GuardedJIT:
  def __init__(self, maximum=4096):
    if maximum < 1: raise ValueError("maximum must be positive")
    self.maximum, self.program = maximum, None
    self.compilations = self.fast_calls = self.fallback_calls = 0

  def __call__(self, x):
    if not isinstance(x, np.ndarray) or x.ndim != 1 or not np.issubdtype(x.dtype, np.floating):
      raise ValueError("expected a floating-point vector")
    if not 1 <= len(x) <= self.maximum: raise ValueError("length outside declared bounds")
    if x.dtype != np.float32 or not x.flags.c_contiguous or not x.flags.aligned:
      self.fallback_calls += 1
      return x*x + 1
    if self.program is None:
      self.program = Program(SOURCE, "square_plus_one")
      self.compilations += 1
    output = np.empty_like(x)
    self.program(x, output, np.array([len(x)], dtype=np.int32))
    self.fast_calls += 1
    return output


def main():
  jit = GuardedJIT(64)
  for n in (1, 7, 31, 64, 7):
    x = np.arange(n, dtype=np.float32)
    np.testing.assert_allclose(jit(x), x*x+1)
  x = np.arange(20, dtype=np.float32)[::2]
  np.testing.assert_allclose(jit(x), x*x+1)
  print(f"variants built={jit.compilations}, compiled calls={jit.fast_calls}, layout fallbacks=
{jit.fallback_calls}")


if __name__ == "__main__": main()
```

A noncontiguous slice `x[::2]` has a larger physical stride. Passing its pointer to this C kernel would read adjacent physical values instead of adjacent logical values. The layout guard prevents that error. A general implementation could instead pass a runtime stride; then the generated address would become `x[i*stride]`, and vectorization assumptions would need their own guards.

Experiment: pass a reversed view, a float64 vector, and lengths 1, 7, 31, and 64. Check outputs and counters. Then remove the contiguity check in a scratch copy and observe a failing strided-input test. Do not turn the failure into an unconditional copy without measuring the copy's cost.

Next implementation: replace the fixed kernel with lowering from symbolic tensor expressions. Keep expression identity, bounds, target, numerical policy, and layout specialization in the plan contract. This small dispatcher does not provide general symbolic shape inference or Python capture.

23.7 Continue into a dynamic tensor graph compiler

Chapter 33 goes beyond the fixed-kernel dispatcher. Its `tensorjit` compiler emits runtime-sized loops for tensor expressions, checks symbolic dimension bindings, fuses pointwise producers into consumers, and runs the result with planned temporary storage. It implements a bounded subset; general Python capture and arbitrary symbolic relations remain separate projects.

24. Memory Planning, Precision, and Distributed Execution

A fast kernel is one component of a fast model. Allocation, transfer, synchronization, saved activations, and communication can dominate the rest. Extend the runtime in small, measurable steps.

24.1 Buffer liveness and reuse

For a linear kernel schedule, assign each temporary an interval from its first write through its last read. Inputs, outputs, aliases, and externally retained values may outlive the schedule. Two temporaries can share storage when their lifetimes do not overlap, their sizes and alignments fit, and the execution order guarantees the earlier work has finished.

Example: t1 is produced by kernel 0 and last read by kernel 2. t2 is produced by kernel 3. Their storage may be reused on a sequential stream. A temporary written by kernel 2 cannot blindly reuse t1 while that kernel is still reading t1. In-place operation requires a separate legality proof.

Lab: implement a best-fit free-list planner with explicit alignment. Compare planned peak bytes against allocating every temporary separately. Test a branched graph with a late consumer, retained outputs, and unequal temporary sizes. For asynchronous multi-stream execution, replace simple list positions with happens-before reasoning; otherwise keep one stream.

24.2 Gradient checkpointing

Reverse mode saves forward values needed by backward. Checkpointing discards selected intermediates and recomputes them later. For a chain of layers, saving every s-th boundary trades fewer saved activations for extra forward work during backward. The best placement depends on tensor sizes and operation costs, not just layer count.

Recomputation must preserve random operations, mutation, and numerical policy. Dropout needs the same random mask or an equivalent reproducible generator state. Replaying a stateful update twice changes the program.

Lab: checkpoint a pure block of the small GPT. Compare outputs and gradients, peak allocated memory, and step time. Add a random operation and demonstrate deterministic recomputation. Report extra work rather than describing reduced memory as a free optimization.

24.3 Mixed precision and quantization

Storage precision, multiplication precision, accumulation precision, and optimizer-state precision are independent choices. Low-precision input does not require low-precision accumulation. Keep an explicit conversion policy, including rounding, saturation, subnormal handling, and exceptional values.

For loss scaling, compute gradients of `scale*loss`, unscale before the optimizer update, and check for nonfinite values. On overflow skip the update and lower the scale. Growth after sustained finite steps is a policy to test. bf16's wider exponent range reduces one motivation for scaling, but does not eliminate numerical error or all overflow risks.

For a symmetric per-output-channel weight quantizer, one simple policy is:

```
scale[j] = max(abs(W[:,j])) / 127
q[:,j] = clamp(round(W[:,j] / scale[j]), -127, 127)
W_approx[:,j] = q[:,j] * scale[j]
```

Handle an all-zero channel without division by zero. Define the rounding rule. Storage savings do not imply a speedup: dequantization overhead and hardware support matter. Activation quantization adds calibration and outlier concerns; weight-only quantization is a smaller first project.

Lab: test conversion boundaries, small values, NaN/infinity policy, zero channels, and outliers. Compare mixed-precision training with f32 on the same seed and batches. For quantization report model quality delta, peak memory, and latency, including conversions.

24.4 Data and tensor parallelism

With P equal-sized data-parallel batches and a mean loss on each worker, averaging worker gradients matches the full-batch gradient apart from rounding. Unequal local batch sizes require weighting by sample

count. An all-reduce sums or averages gradients; every worker then applies the same update.

For row-vector inputs and `Y=XW`, splitting W by output columns produces independent pieces of Y that can be concatenated. Splitting W by input rows also splits X and produces partial outputs that must be summed. Track bias placement: adding the full bias to every partial sum overcounts it.

Communication cost is often approximated as latency plus bytes divided by effective bandwidth. Many tiny collectives pay latency repeatedly; bucketing helps but can delay communication overlap. An asynchronous collective cannot consume gradients before they are ready or release their buffers early. A real runtime also needs a policy for failed workers and communication errors.

Lab: compare a two-process data-parallel update with one full-batch update, including an unequal batch case. Then partition one linear layer both ways and verify outputs and gradients. Record actual communication bytes. A CPU process simulation teaches algebra; complete a separate real-device measurement before claiming distributed GPU scaling.

24.5 Inference scheduling

Prefill processes prompt tokens together; decode repeatedly handles small additions to existing contexts. They have different matrix shapes and resource needs. A serving scheduler must manage request lengths, cache capacity, cancellation, and admission without corrupting per-request state. Paged KV storage introduces a logical-to-physical block mapping so a request need not reserve one large contiguous allocation. Indirection and allocation policy become part of the execution system.

Lab: build a simulator with variable prompt/output lengths and a finite cache budget. Compare fixed batches and continuous admission. Report throughput, time to first token, inter-token latency, and latency distribution. Then implement one policy with the actual model. Synthetic scheduler results alone do not establish end-to-end serving performance.

Exit gate: correct buffer reuse, checkpointing, one measured precision optimization, a verified parallel update, and an inference scheduling experiment. Keep each artifact independently reproducible.

24.6 Worked lab: buffer reuse with observable lifetimes

Take three temporaries: a uses 1024 bytes during kernels 0–2, b uses 512 bytes during kernel 1, and c uses 768 bytes during kernels 3–4. Place a at offset 0 and b at offset 1024. At kernel 3, a is dead, so c reuses a's 1024-byte slot. The arena needs 1536 bytes instead of 2304 bytes of independent payload storage. Padding and different allocation strategies can change the comparison.

Lifetimes are inclusive. If b were produced by kernel 2 instead, it could not share a's storage: kernel 2 may need to read a while writing b. The strict condition is `old.last < new.first`. An in-place kernel is a different contract that this planner does not infer.

```
.venv/bin/python -m labs.memory
```

`labs/memory.py`

```python
"""Aligned interval reuse for a sequential runtime. Run: python -m labs.memory

Intervals are inclusive: a value read by kernel k cannot share a new output
written by kernel k. The caller must exclude aliases and externally retained buffers.
"""
from dataclasses import dataclass


@dataclass(frozen=True)
class Temporary:
  name: str
  nbytes: int
  first: int
  last: int
  alignment: int = 64


@dataclass(frozen=True)
class Allocation:
  offset: int
  capacity: int


def align_up(value, alignment): return ((value + alignment - 1) // alignment) * alignment


def plan_memory(temporaries):
  temporaries = list(temporaries)
  if len({t.name for t in temporaries}) != len(temporaries): raise ValueError("duplicate name")
  if any(t.nbytes <= 0 or t.first < 0 or t.last < t.first or t.alignment <= 0 for t in
temporaries):
    raise ValueError("invalid lifetime, size, or alignment")
  active, free, allocations, peak = [], [], {}, 0
  for t in sorted(temporaries, key=lambda t: (t.first, t.name)):
    still_active = []
    for last, allocation in active:
      if last < t.first: free.append(allocation)
      else: still_active.append((last, allocation))
    active = still_active
    candidates = [(a.capacity, i, a) for i, a in enumerate(free)
                  if a.capacity >= t.nbytes and a.offset % t.alignment == 0]
    if candidates:
      _, i, allocation = min(candidates)
      free.pop(i)
    else:
      offset = align_up(peak, t.alignment)
      allocation = Allocation(offset, t.nbytes)
      peak = offset + t.nbytes
    allocations[t.name] = allocation
    active.append((t.last, allocation))
  return allocations, peak
```

```
def verify_plan(temporaries, allocations):
  temporaries = list(temporaries)
  for i, a in enumerate(temporaries):
    aa = allocations[a.name]
    if aa.offset < 0 or aa.offset % a.alignment or aa.capacity < a.nbytes: raise ValueError("bad
allocation")
    for b in temporaries[i+1:]:
      bb = allocations[b.name]
      lifetimes_overlap = max(a.first, b.first) <= min(a.last, b.last)
      storage_overlaps = max(aa.offset, bb.offset) < min(aa.offset+aa.capacity,
bb.offset+bb.capacity)
      if lifetimes_overlap and storage_overlaps: raise ValueError(f"live storage overlap:
{a.name}, {b.name}")


def main():
  temporaries = [Temporary("a", 1024, 0, 2), Temporary("b", 512, 1, 1), Temporary("c", 768, 3,
4)]
  allocations, peak = plan_memory(temporaries)
  verify_plan(temporaries, allocations)
  for name, allocation in allocations.items(): print(name, allocation)
  print(f"peak={peak} bytes; independent allocations={sum(t.nbytes for t in temporaries)} bytes")


if __name__ == "__main__": main()
```

The planner uses best fit among whole freed slots. It does not split or coalesce slots and is not an optimal packing algorithm. Retaining capacity makes its reuse decision simple and auditable. Alignment is a condition on offsets relative to an appropriately aligned arena base; allocating an arena with an insufficiently aligned base would violate the complete runtime contract.

Experiment: generate random lifetimes, write a distinct byte pattern when each temporary is produced, and check every live pattern at every step. This simulates execution independently of the overlap verifier. The included tests do exactly that; a verifier that repeats the planner's mistake would otherwise provide false confidence.

Next implementation: infer lifetimes from `Plan` references, exclude aliases and retained outputs, allocate an aligned device arena, and bind kernel arguments to slices. Keep the initial integration on one ordered stream. Multi-stream reuse needs event-based happens-before proofs, not these integer schedule positions alone.

24.7 Run the planner inside an actual compiler runtime

Chapter 33 derives lifetimes from rendered kernel dependencies, allocates an aligned arena, and passes planned buffer slices to native kernels. Its tests check actual matrix-chain results, memory savings, and retained-output safety. This implements the single-schedule integration described above in a separate float32 compiler, without modifying `uopc`'s existing device runtime.

25. From Course Completion to Independent Engineering

Expertise means handling unfamiliar programs, locating errors, and defending performance claims. Use this chapter as an assessment and a repeatable research workflow. There is no fixed number of weeks that guarantees mastery; progress depends on prior background, available hardware, and depth.

25.1 A capstone with a bounded question

Choose one project with a specific hypothesis. Examples: an asynchronous pipeline improves long-K GEMM on a named GPU; symbolic shapes eliminate recompilation for a bounded family of sequence lengths; a memory planner reduces GPT peak temporary allocation; an attention backward avoids quadratic saved state. State the input domain and numerical contract before coding.

Deliver the implementation, correctness suite, baseline, benchmark harness, raw results, analysis, and a short reproduction guide. Include at least one negative result or limitation. A hardware port must execute on the target; a paper reproduction must identify which algorithmic features and reported experiments it actually reproduces.

25.2 Correctness beyond examples

Generate random small legal graphs, evaluate them with an independent reference, and compare compiled results. Bias the generator toward edge cases: singleton dimensions, irregular tiles, negative integer division, shared subexpressions, views, repeated reductions, and deep graphs. Restrict operations to well-defined inputs or explicitly check the error contract.

When a failure appears, save its seed and shrink the program by removing operations, reducing shapes, and simplifying constants while preserving the failure. A three-node counterexample is more useful than a thousand-node crash. Add the minimized case as a regression test.

Competitive testing can be useful: contributors submit valid programs intended to expose compiler bugs. Validate each test with an independent reference before counting failures. Keep correctness scoring separate

from performance scoring and cap execution resources for each submitted case.

Lab: build a differential fuzzer and reducer. Introduce a known bad rewrite and show the fuzzer finds it. Then test an unmodified compiler and report findings, including “no failure found” honestly.

25.3 Reading and changing a larger compiler

Pick a codebase aligned with the question: a tensor compiler for graph scheduling, a kernel compiler for layout lowering, or LLVM/MLIR for analysis and intermediate representations. Pin a revision. Start from a tiny running example and trace one operation from public API to generated kernel. Read tests and print pass outputs before trying to understand the entire repository.

Make one bounded change: fix a minimized correctness case, add a missing legal lowering, or improve a measured schedule. Run the project's relevant checks and document both favorable and unfavorable workloads. An upstream-ready patch is a useful artifact even if you choose not to submit it.

25.4 Closed-book oral assessment

Explain these with a whiteboard and a short program, without reading the implementation:

1. Why does hash consing simplify common-subexpression sharing, and why is effect ordering different?
2. Which index transformation implements transpose followed by reshape, and when is a copy needed?
3. Why can integer and floating-point rewrites require different legality rules?
4. Where must a phi appear in a branch or loop, and what does dominance establish?
5. When does fusion save memory traffic but increase compute or register pressure?
6. Why can a correct barrier kernel fail under partial blocks?
7. Why can a lower-occupancy GEMM be faster, and which measurements distinguish explanations?
8. Derive the online softmax recurrence and its backward gradient.
9. What belongs in a compiled-plan key, and which assumptions require guards?
10. Under which ordering and alias conditions can two temporaries share storage?
11. How does gradient averaging change with unequal local batch sizes?

12. Which evidence distinguishes correct small-GPT training from competitive large-model training?

For each answer, produce a counterexample to an incorrect simplification. If you cannot, revisit the corresponding lab before adding more features.

25.5 Final acceptance report

Dimension	Required evidence
Semantics	Written operation, type, shape, effect, and numerical contracts
Correctness	Independent references, gradient checks, minimized regressions
Compiler understanding	Pass-by-pass trace and legality argument for one nontrivial optimization
GPU understanding	Actual device execution, profiler evidence, synchronization validation
Performance	Matched baselines, diverse shapes, raw samples, compile/runtime separation
Runtime	Memory accounting, cache/guard behavior, checkpoint resume
Scale	Clearly bounded model and device configuration actually exercised
Maintainability	Small reviewable modules, useful diagnostics, reproducible commands

A completed report supports concrete claims about the work you performed. Keep expanding its scope through new architectures, workloads, and external contributions rather than treating “hero” as a permanent certification.

26. Numerical Analysis for Compiler and Kernel Engineers

Optimizations change arithmetic order, precision, and sometimes the approximation itself. A frontier systems engineer needs to explain the resulting error, not just increase a tolerance until tests pass. This chapter supplies the numerical reasoning behind the precision experiments in Chapter 24.

26.1 Conditioning versus implementation error

Conditioning describes how sensitive the mathematical problem is to input perturbations. Stability describes the error introduced by an algorithm. A perfectly implemented low-precision operation can still produce a large output error for an ill-conditioned problem.

For a sum, compare `abs(sum(x))` with `sum(abs(x))`. When positive and negative terms nearly cancel, a tiny relative perturbation of inputs can cause a large relative change in the result. Consequently, relative error alone is a poor diagnostic. Report maximum absolute error, relative error away from zero, and a norm-based error suited to the operation.

Lab: sum `[1e8, 1, -1e8]` in f32 in different orders, then compare naive, pairwise, and compensated summation on random and cancellation-heavy arrays. Use a higher-precision reference and state its limitations. Separate input conversion error from accumulation error.

26.2 Dot products and error budgets

Under the usual model `fl(a op b)=(a op b)*(1+delta)` with `abs(delta)<=u` for normal finite results, a sequential sum has an error bound involving `gamma_n=n*u/(1-n*u)` when `n*u<1`. A dot-product bound scales with `sum(abs(a_i*b_i))`; severe cancellation can make relative output error large. Overflow, underflow, subnormals, and exceptional values require separate treatment.

Fused multiply-add rounds a multiply-plus-add once rather than rounding the product separately. It can improve accuracy while changing bitwise results. Tensor-core products and framework reference kernels

may have different input conversion and accumulation policies. “Both f32” is not a complete numerical specification.

Lab: sweep reduction length and input scale for each supported precision path. Predict where error should grow, compare against measurement, and identify saturation or underflow. Write an error budget for GEMM plus bias plus activation instead of choosing an unexplained global tolerance.

26.3 Transcendentals, softmax, and normalization

Approximations for exp, log, reciprocal, and reciprocal square root have domains and error profiles. Range reduction maps the argument to a smaller interval, a polynomial or table approximates the function there, and reconstruction maps back. Exceptional inputs and boundary continuity still matter.

Stable softmax subtracts the row maximum before exponentiation. This prevents many overflows but does not make every masking configuration meaningful. Normalization statistics can lose precision when computed as `mean(x*x)-mean(x)^2` for nearly constant large values; a centered or online method can be more stable. Define population versus sample variance explicitly.

Lab: implement an online mean/variance algorithm and a merge rule for two partial groups. Compare with the naive expression on near-constant inputs. Audit a fused normalization kernel's variance, epsilon placement, accumulator precision, and backward formula.

26.4 Gradient and training validation

Finite differences test a local derivative but have truncation and cancellation errors. Sweep the step size rather than trusting one value. Directional derivatives reduce cost: compare a finite difference along v with the inner product of the analytic gradient and v. A vector-Jacobian product and Jacobian-vector product satisfy an adjoint identity that can test implementations without constructing the full Jacobian.

Training validation also requires optimizer equations, reduction conventions, gradient accumulation, clipping, random-number behavior, and checkpoint state. Two correct nondeterministic kernels can produce diverging trajectories; compare distributions and short matched steps before attributing all differences to a bug or all differences to randomness.

Exit gate: a numerical test report for GEMM, softmax, normalization, and gradients; explicit precision policies; and a short training comparison that explains observed deviations.

26.5 Worked lab: mergeable variance and quantization error

A running variance state stores `(n, mean, M2)`, where M2 is the sum of squared deviations from the current mean. After receiving x, set `delta=x-mean`, update the mean by `delta/(n+1)`, and add `delta*(x-new_mean)` to M2. This avoids subtracting two nearly equal large second moments.

For two groups A and B, let `delta=mean_B-mean_A` and `n=n_A+n_B`. The merged M2 is `M2_A+M2_B+delta^2*n_A*n_B/n`. The extra term accounts for the groups having different centers. An empty group is the identity. This merge rule makes a parallel reduction possible, with the usual floating-point dependence on reduction order.

```
.venv/bin/python -m labs.numerics
```

`labs/numerics.py`

```python
"""Mergeable variance and per-channel int8 quantization. Run: python -m labs.numerics"""
from dataclasses import dataclass
import numpy as np


@dataclass(frozen=True)
class Moments:
  count: int = 0
  mean: float = 0.0
  m2: float = 0.0

  def add(self, value):
    count = self.count + 1
    delta = float(value) - self.mean
    mean = self.mean + delta / count
    return Moments(count, mean, self.m2 + delta * (float(value) - mean))

  def merge(self, other):
    if self.count == 0: return other
    if other.count == 0: return self
    count = self.count + other.count
    delta = other.mean - self.mean
    return Moments(count, self.mean + delta * other.count / count,
                   self.m2 + other.m2 + delta*delta * self.count * other.count / count)

  def variance(self):
    if self.count == 0: raise ValueError("empty population")
    return self.m2 / self.count


def moments(values):
  state = Moments()
  for value in values: state = state.add(value)
  return state


def quantize_weights(weights):
  """Symmetric per-column, ties-to-even; input must be a finite nonempty matrix."""
  weights = np.asarray(weights, dtype=np.float64)
  if weights.ndim != 2 or min(weights.shape) < 1 or not np.isfinite(weights).all():
    raise ValueError("expected finite nonempty weight matrix")
  maxima = np.max(np.abs(weights), axis=0)
  scale = np.where(maxima == 0, 1.0, maxima / 127)
  quantized = np.clip(np.rint(weights / scale), -127, 127).astype(np.int8)
  return quantized, scale


def main():
  rng = np.random.default_rng(0)
  values = 1e8 + rng.normal(size=1000)
  state = moments(values[:500]).merge(moments(values[500:]))
  print(f"variance: merged={state.variance():.8f}, reference={np.var(values):.8f}")
```

```
    weights = rng.normal(size=(64, 8)); weights[:, 0] = 0
    q, scale = quantize_weights(weights)
    print(f"int8 reconstruction max error={np.abs(q*scale - weights).max():.6f}")


if __name__ == "__main__": main()
```

The lab evaluates near-constant data around 1e8 using double precision. Changing it to f32 before the calculation can erase small variations at input conversion; no variance algorithm can recover information already rounded away. Separate storage error from algorithmic error.

For per-channel int8 quantization, the chosen scale makes the largest magnitude map to 127. With nearest rounding and no additional clipping of finite in-range values, reconstruction error is at most half a scale step, apart from floating-point rounding in the quantizer. Zero channels use scale 1 and quantized value 0, avoiding division by zero. NumPy's rounding specifies ties to even here.

Experiment: check the half-step error bound channel by channel, add a single extreme outlier, and compare per-channel with a global scale. Then evaluate the perturbed weights in a model: a small weight error bound does not alone bound validation loss. This is a numerical reference lab, not an integer hardware GEMM or a full quantized inference runtime.

27. Scheduling, Autotuning, and Backend Construction

A compiler must choose a legal implementation, predict whether it is useful, and lower it to the machine. This chapter extends the scalar analysis in Chapter 21 and GPU experiments in Chapter 22 into a compiler-driven search rather than a collection of hand-selected kernels.

27.1 Iteration domains and dependencies

For GEMM, an iteration domain is `{(i,j,k): 0<=i<M, 0<=j<N, 0<=k<K}`. Access maps send an iteration to `A[i,k]`, `B[k,j]`, and `C[i,j]`. A schedule maps iterations to execution order. Tiling introduces outer tile coordinates and inner coordinates, with guards for partial tiles.

Dependences constrain schedules. A producer must precede its consumer unless a transformation replaces the dependence with an equivalent mechanism. Reduction parallelization introduces partial accumulators and a combine step; it changes floating-point order. For affine bounds and accesses, integer-set reasoning can establish useful properties. Data-dependent indices require other methods or conservative fallback. A polyhedral representation is powerful within its supported domain, not a proof engine for arbitrary programs.

Lab: represent a two-dimensional iteration domain and affine access maps. Implement interchange and tiling, then reject an interchange that violates a dependence in an in-place recurrence. Print both the legality reason and resulting schedule.

27.2 Layout as a compiler decision

Logical tensor shape does not determine lane ownership, register arrangement, shared-memory swizzle, or vector access width. A matrix-instruction operand needs an exact mapping from logical elements to participating lanes and registers. A layout conversion may be register shuffles, shared-memory traffic, or a combination. Its cost can erase an arithmetic speedup.

Track layouts explicitly through the kernel IR. Require a verifier to check that consumers accept the producer layout or that a legal conversion is present. Estimate conversion bytes and instructions alongside compute. Test layout maps for injectivity where each element needs a unique owner and account for intentional replication separately.

Lab: define two lane layouts for a small tile, implement their conversion, and round-trip random values. Count communication and compare with a kernel that keeps one layout throughout. Investigate one existing compiler's printed layout IR at a pinned revision.

27.3 Search and cost models

Define schedule parameters with legality constraints: tile sizes, warps, vector widths, pipeline stages, and shared-memory budgets. Prune impossible configurations before compilation. Record compiler failures and correctness failures separately from slow valid schedules. Keep a generic fallback.

An analytical model can estimate arithmetic intensity and resource limits. A learned model can rank candidates from features and measurements, but its quality depends on the training distribution. Split evaluation by workloads or shape families, not merely repeated timings of the same kernels. Include compilation and search cost when judging whether tuning pays off.

Lab: compare a heuristic, random search, and a ranked search on a fixed tuning budget. Evaluate on held-out shapes. Report best latency, total tuning time, compilation failures, and variance. Cache winners with target, compiler version, dtype, layout, and numerical policy in the key.

27.4 Direct backend essentials

A backend legalizes unsupported operations, selects instructions, schedules them, allocates registers, and emits executable code or an intermediate format accepted by another toolchain. Start with a small virtual target so each responsibility is visible. Define registers, loads/stores, branches, calling convention, and instruction encoding before writing an emitter.

Liveness and interference constrain register reuse. Spills need stack or local-memory slots and additional instructions. Phi elimination introduces edge copies, whose parallel-copy semantics must be preserved even

when values form a swap cycle. Instruction scheduling must honor data and effect dependences; latency hiding cannot violate the target's memory model.

Lab: lower the scalar compiler into a tiny register machine, allocate a deliberately small number of registers, resolve phi copies, and execute the resulting program in an emulator. Add a disassembler and compare pre/post-allocation behavior. Then inspect a real backend's code for the same example.

27.5 Compilation as an observable service

Track time and memory per pass, IR sizes, cache hit rates, and diagnostic context. Incremental compilation should reuse results only when their dependencies remain valid. A parallel compiler must keep shared caches and contexts thread-safe and preserve deterministic artifacts where required. Persistent executable caches need atomic writes, compatibility checks, and a policy for untrusted or corrupted artifacts; do not load arbitrary cached machine code solely because its filename matches.

Exit gate: legal schedule transformations, verified layouts, a held-out tuning evaluation, a small functioning backend, and a compiler profile identifying where compilation time is spent.

28. Irregular Workloads and Heterogeneous Accelerators

Dense GEMM is central but cannot represent every workload a frontier lab encounters. Sparse data, variable-length batches, mixture-of-experts routing, and multiple hardware targets expose assumptions hidden by static contiguous tensors.

28.1 Gather, scatter, and atomic reductions

Gather reads values at data-dependent indices. Scatter writes to data-dependent destinations, so duplicate indices require explicit semantics: overwrite, ordered update, or reduction. Parallel floating-point scatter-add often uses atomics and may be nondeterministic because update order varies. A deterministic alternative can sort or group updates, at additional cost.

Lab: implement embedding lookup and its gradient with repeated token IDs. Test out-of-range indices and state the error policy. Compare atomic and grouped reductions for correctness, reproducibility, and skewed index distributions. Dense random indices alone miss contention hotspots.

28.2 Sparse formats and ragged batches

CSR stores row offsets, column indices, and nonzero values. Row r spans offsets `[rowptr[r],rowptr[r+1])`. Work per row is uneven, so assigning one fixed unit of work per row can leave hardware underused. Block-sparse formats trade some wasted arithmetic for regular tiles and reduced metadata overhead. Sparsity percentage alone cannot predict speed: distribution, block structure, index traffic, and conversion cost matter.

Ragged sequences use lengths or offsets rather than padding every example to the maximum length. This reduces wasted arithmetic but complicates bounds, scheduling, and memory ownership. Dynamic batch composition can change the best schedule from one request to the next.

Lab: implement CSR matrix-vector multiply and a segmented reduction. Test empty rows, long rows, and heavily skewed lengths. Compare against dense and padded baselines including format conversion. Explain

where sparse execution becomes worthwhile for your actual hardware.

28.3 Mixture-of-experts dispatch

A routed layer selects experts per token, groups tokens by destination, computes expert outputs, and combines them with routing weights. Correctness includes token ordering, capacity policy, multiple selected experts, and the reverse mapping in backward. Distributed routing adds communication and load imbalance; average tokens per expert can hide an overloaded expert that sets step time.

Lab: build a CPU reference with explicit token-to-expert lists, then implement grouped dispatch and combine. Compare uniform and skewed routing. Measure padding waste, dispatch cost, expert compute, and communication separately. Test gradients with repeated destinations and a defined overflow policy.

28.4 Portability and hardware/software codesign

Identify which compiler layers express mathematical semantics and which encode hardware assumptions. A portable operation can still need a target-specific schedule, layout, synchronization mechanism, and runtime ABI. Capability queries should select supported implementations and provide clear errors or fallbacks for unsupported features.

For a new accelerator, write a machine model: compute units, supported numeric formats and operations, memory hierarchy, bandwidth, local storage, communication topology, execution ordering, and launch cost. Estimate a roofline before implementing a backend. Compare the estimate with microbenchmarks and revise it; a simulator is useful only within the fidelity it has demonstrated.

Lab: port elementwise work and a reduction to a second target, or build a documented toy-accelerator simulator if hardware is unavailable. Lower one tiled matrix operation and explain each architecture assumption. Label simulated performance separately from measured performance.

Exit gate: correct irregular operations, a skew-sensitive benchmark suite, a routed-layer prototype, and a portability report grounded in an explicit target model.

28.5 Worked lab: duplicate indices change the operation

A CSR matrix may contain several entries for the same column in one row. Matrix-vector multiplication must add all their contributions. For offsets `[0,3,3,4]`, the second row is empty; its sum is zero. Neither empty rows nor duplicates should be treated as malformed when the format contract allows them.

Embedding backward has the same accumulation issue. If tokens are `[1,1,2]`, gradients for the first two positions both contribute to vocabulary row 1. NumPy advanced-index assignment `result[indices] += gradients` uses temporary indexing and does not provide the required accumulation for repeated indices. `np.add.at` makes the intended repeated updates explicit in the CPU reference.

```
.venv/bin/python -m labs.irregular
```

`labs/irregular.py`

```python
"""CSR and scatter-add semantics, including duplicate indices.

Run: python -m labs.irregular. CPU reference algorithms, not optimized kernels.
"""
import numpy as np


def csr_matvec(rowptr, columns, values, vector):
  rowptr, columns, values, vector = map(np.asarray, (rowptr, columns, values, vector))
  if any(x.ndim != 1 for x in (rowptr, columns, values, vector)) or len(rowptr) < 1:
    raise ValueError("expected one-dimensional CSR arrays and vector")
  if not all(np.issubdtype(x.dtype, np.integer) for x in (rowptr, columns)):
    raise ValueError("offsets and indices must be integer")
  if rowptr[0] != 0 or rowptr[-1] != len(values) or len(columns) != len(values) or
np.any(np.diff(rowptr) < 0):
    raise ValueError("invalid CSR offsets")
  if np.any(columns < 0) or np.any(columns >= len(vector)): raise ValueError("column out of
bounds")
  result = np.zeros(len(rowptr)-1, dtype=np.result_type(values, vector))
  for row in range(len(result)):
    start, end = rowptr[row:row+2]
    result[row] = np.sum(values[start:end] * vector[columns[start:end]])
  return result


def embedding_gradient(indices, grad_output, vocabulary_size):
  indices, grad_output = np.asarray(indices), np.asarray(grad_output)
  if indices.ndim != 1 or not np.issubdtype(indices.dtype, np.integer) or grad_output.ndim != 2
or len(indices) != len(grad_output):
    raise ValueError("invalid indices or output gradient")
  if vocabulary_size < 1 or np.any(indices < 0) or np.any(indices >= vocabulary_size):
    raise ValueError("embedding index out of bounds")
  result = np.zeros((vocabulary_size, grad_output.shape[1]), dtype=grad_output.dtype)
  # result[indices] += grad_output would lose repeated-index updates.
  np.add.at(result, indices, grad_output)
  return result


def main():
  print("CSR with an empty row:", csr_matvec([0, 2, 2, 3], [0, 2, 1], [2., 3., 4.], [1., 2.,
3.]))
  print("repeated-token gradient:", embedding_gradient([1, 1, 2], np.ones((3, 2)), 4).tolist())


if __name__ == "__main__": main()
```

The CSR implementation checks offsets, array lengths, index types, and bounds before execution. This validation belongs at an import or construction boundary for trusted repeated workloads, not necessarily in every timed kernel launch. Measure validation separately if it is amortized.

Experiment: compare CSR output against a dense matrix assembled by adding entries, not overwriting duplicates. For embeddings, send every token to one vocabulary row and verify its gradient equals the sum of all token gradients. That distribution is also a useful GPU contention stress case.

Next implementation: implement a real GPU scatter-add, then a sorted segmented reduction. Compare precision, determinism, scratch memory, and skew sensitivity. A deterministic CPU reference provides semantics; it does not require every parallel implementation to have identical rounding order.

29. A Frontier-Lab Preparation Map

This track targets ML compiler, GPU kernel, and training/inference systems engineering. It does not claim to cover every role in a research lab, and completing a book does not guarantee hiring or research success. The objective is demonstrable independence: turn an unfamiliar workload into a correct implementation, diagnose its limits, improve it, and make the result reproducible.

29.1 Common foundation and specialization

Finish the prerequisite labs and core compiler, then work through Chapters 21–28. Choose one primary specialization for deep projects while retaining working knowledge of the others.

Track	Core depth	Evidence of independence
ML compiler	SSA/effects, lowering, symbolic proofs, layouts, scheduling, cost models	Nontrivial verified pass in a larger compiler; legality and performance analysis
GPU kernels	Memory hierarchy, matrix instructions, asynchronous pipelines, numerics	Forward/backward kernel suite with hardware profiles and diverse shape results
Training systems	Autodiff, optimizers, memory, parallelism, checkpointing, reliability	Resumable distributed training experiment with scaling and failure analysis
Inference systems	Dynamic shapes, KV memory, quantization, batching, tail latency	End-to-end serving experiment under a finite cache and latency budget
Hardware/compiler codesign	Target models, backend legality, layouts, sparse workloads	Executable backend or target port, microbenchmarks, and model-versus-device analysis

Read related mathematical material when a lab demands it: linear algebra for tensor transformations, calculus for derivatives, probability for stochastic training, numerical analysis for precision, and statistics for experiments. For systems depth, study concurrency, virtual memory, networking, storage, and distributed

failure models. A gap exposed by a project becomes required study, not an excuse to extrapolate from a toy implementation.

29.2 Distributed training beyond two workers

Chapter 24 establishes parallel algebra. Extend it with gradient accumulation, data/tensor/pipeline partitioning, optimizer-state sharding, and activation recomputation. For pipeline parallelism, draw a microbatch timeline and account for idle stages, activation lifetime, and weight-version consistency. For sharding, state which worker owns each parameter, gradient, and optimizer state at each phase.

Build a per-step memory budget: parameters, gradients, optimizer states, activations, communication buffers, allocator fragmentation, and temporary workspace. Build a communication budget by collective, message size, topology, and frequency. Measure overlap; asynchronous APIs alone do not demonstrate it.

Lab: implement two partitioning strategies on a bounded model. Compare with a single-device reference, report strong-scaling efficiency, and account for global batch and optimizer semantics. Restore after an injected worker failure from a consistent checkpoint. Verify dataset position, optimizer state, random state, and parameter consistency. Simulated ranks establish logic; hardware experiments establish scaling.

29.3 End-to-end diagnosis

A model slowdown may come from data loading, host dispatch, compiler misses, kernels, transfers, collectives, or synchronization. Capture a timeline before optimizing individual kernels. Derive a critical path and estimate the maximum benefit of the proposed change. If a kernel occupies 10% of step time, even eliminating it entirely cannot yield a 2× end-to-end speedup.

Lab: create one deliberate bottleneck in each layer: input pipeline, compilation, launch overhead, memory traffic, compute, and communication. Diagnose each from evidence rather than knowing which bottleneck was injected. Maintain a short investigation record: observation, hypothesis, experiment, result, and next action.

29.4 Production reliability and review

A production runtime needs resource ownership, bounded caches, cancellation, timeouts, useful errors, and cleanup after partial failure. Asynchronous errors can surface long after their cause; attach kernel and graph context to diagnostics. Validate imported model formats and allocation sizes before trusting them. Define how the system behaves when memory is exhausted or a device disappears.

Lab: inject allocation failure, corrupted cache metadata, invalid model dimensions, interrupted checkpoint writes, and failed collectives. Verify clean failure or recovery with no silently wrong outputs. Review an optimization for semantic assumptions, numerical impact, concurrency, cache compatibility, and regressions —not just code style.

29.5 Research literacy and original experiments

Read a paper in three passes: identify the problem and claimed contribution; reconstruct the algorithm and assumptions; examine the evidence and missing comparisons. Separate algorithmic changes from hardware, precision, compiler, and workload differences. Implement the smallest experiment capable of falsifying the central claim before attempting a full reproduction.

A useful new result may be a negative result, a counterexample, a better measurement, or a narrower algorithmic improvement. State the hypothesis in advance, use held-out workloads for tuning decisions, and report uncertainty and failed runs. Keep datasets, seeds, revisions, environment, and commands sufficient for another engineer to reproduce the conclusion.

Lab: reproduce one bounded result, identify one limitation, and test an extension that addresses it. Write a short paper-style report: question, method, baselines, results, ablations, limitations, and reproduction instructions. Obtain technical review from another person when possible and resolve their strongest counterargument with an experiment.

29.6 Portfolio and readiness gates

Produce three substantial artifacts: a compiler change with semantic tests; a real-device kernel study with matched baselines; and an end-to-end training or inference system with memory and failure analysis. At least one should work inside an established external codebase at a pinned revision. The projects may share a model but must establish distinct claims.

You are ready to attempt independent work when you can explain unfamiliar IR, minimize a wrong-code bug, reason about numerical differences, find a measured bottleneck, design a legal optimization, and communicate a reproducible result under review. Continue learning through work with larger systems and experienced reviewers. The curriculum supplies the preparation and the tests of progress; real projects supply the remaining depth.

29.7 What you can execute now

The worked compiler analysis, attention forward/backward, guarded C execution, buffer planner, numerical methods, and irregular operation references are executable baselines with independent checks. Run their modules from the repository root and read the corresponding tests before extending them.

The next milestone for each is integration: SSA construction and machine lowering; an actual GPU attention kernel; symbolic tensor compilation; runtime buffer binding; model-level precision validation; and parallel irregular operations. The later research and scale assignments remain substantial work. Keep the completion ledger honest about that distinction while building your portfolio.

29.8 Deeper implemented compiler paths

The next block of the book replaces several remaining isolated exercises with integrated programs. Chapter 30 constructs and verifies SSA from source; Chapter 31 implements path-sensitive constant propagation and effect-aware DCE; Chapter 32 executes a register allocation and emits native C; and Chapter 33 compiles dynamic float32 tensor expressions with fusion and actual memory reuse. These are implemented

and tested CPU paths. Their measurements and limitations are distinct from the later real-GPU and distributed-scale milestones.

Chapter 34 adds shape-aware reverse-mode autodiff, independently checked gradients, and generated forward/backward training with changing minibatch sizes and one executable.

Chapter 35 supplies a real-device shared/register-tiled GEMM experiment, including irregular-shape checks and device-event measurements. Profiler evidence, tensor-core optimization, and matched competitive baselines remain additional milestones.

30. Building a Source Compiler: Contracts, CFGs, and SSA Construction

The analysis exercise in Chapter 21 began with a hand-written graph. This chapter builds the graph from source. The implementation is `minicc`, a separate compiler whose purpose is to expose control flow, effects, verification, and backend decisions that tensor expressions can hide. It complements `uopc`; it does not replace its autodiff or GPU pipeline.

Run the complete example before reading the implementation:

```
.venv/bin/python -m minicc
.venv/bin/python -m pytest tests/test_minicc.py -q
```

The example compiles a loop with a conditional accumulation and obtains 87 through an SSA interpreter, a register-constrained machine, and generated native C. Agreement across these paths is useful, but not sufficient by itself: they share the IR. The tests also execute trusted source programs with Python as an independent oracle over a small range that avoids overflow. Overflow and memory errors have separate tests against the compiler's explicitly different semantics.

30.1 Define the language before optimizing it

The source syntax is a restricted Python function, parsed by Python's `ast` module. This delegates lexing and parsing, not execution or language semantics. No source function is evaluated by the compiler. Function calls, comprehensions, imports, general objects, exceptions, chained comparisons, short-circuit `and` / `or`, and unsupported statements are rejected on the lowered path.

The supported language includes scalar variables, integer constants, `+`, `-`, `*`, comparisons, `not`, assignment, `if`, `while`, `break`, `continue`, and explicit `return`. Buffer parameters use an annotation naming `buffer`; scalar parameters may be unannotated or annotated `int`. Buffers are rank-one indexed storage. An integer result is required on every structurally reachable exit. Unreachable statements after a terminator need not be lowered; this is not a general Python validator.

Every scalar value is signed i32. Arithmetic wraps modulo `2^32` and is reinterpreted as signed. Comparisons operate on those signed values and produce 0 or 1. A condition is true when nonzero. Input

scalar integers and constants are normalized to that domain. For example, `2147483647+1` is `-2147483648`. A compiler that tests only tiny positive values could accidentally implement mathematical integers or undefined C overflow without noticing.

Buffer indices must be in `[0,length)`. Negative indexing does not have Python's meaning. A failed access reports a bounds error; earlier stores remain visible, and later instructions do not execute. Two buffer arguments may overlap. The compiler therefore cannot assume that storing through a leaves loads through b unchanged. No generated `restrict` promise is made.

These choices define optimization legality. Removing an unused add is valid. Removing an unused load is not generally valid, because the load can fail. Reordering a store before a potentially failing load can change the state visible after an error. Treat trapping and memory effects as part of the program, rather than incidental implementation details.

30.2 An inspectable intermediate representation

A function owns parameters, named blocks, and an entry label. An instruction has an operation, an optional result, and operands that are either integer literals or value names. Stores have no scalar result. A terminator is a jump, conditional branch, or return.

A block's instruction order preserves memory effects. The representation does not yet have memory SSA or general alias analysis: it establishes safety by declining to reorder accesses. That is a useful baseline against which a future alias analysis must prove additional transformations.

`minicc/ir.py`

```python
"""Small control-flow IR with explicit effects and signed, wrapping i32 arithmetic."""
from __future__ import annotations
from dataclasses import dataclass, field
from enum import Enum

Operand = str | int


class Kind(Enum):
  I32 = "i32"
  BUFFER = "buffer"


@dataclass(frozen=True)
class Parameter:
  name: str
  kind: Kind = Kind.I32


@dataclass
class Instruction:
  op: str
  dest: str | None
  args: tuple[Operand, ...]


@dataclass
class Phi:
  base: str
  dest: str
  incoming: dict[str, Operand] = field(default_factory=dict)


@dataclass
class Terminator:
  op: str
  args: tuple[Operand, ...] = ()
  targets: tuple[str, ...] = ()


@dataclass
class Block:
  instructions: list[Instruction] = field(default_factory=list)
  terminator: Terminator | None = None
  phis: list[Phi] = field(default_factory=list)


@dataclass
class Function:
  name: str
  parameters: tuple[Parameter, ...]
  blocks: dict[str, Block]
```

```
  entry: str = "entry"
  ssa: bool = False


PURE = {"copy": 1, "add": 2, "sub": 2, "mul": 2, "lt": 2, "le": 2, "eq": 2, "ne": 2}
EFFECTS = {"load": 2, "store": 3}  # Even an unused load can raise a bounds error.


def i32(value: int) -> int:
  value = int(value) & 0xffffffff
  return value - 0x100000000 if value & 0x80000000 else value


def alu(op: str, args) -> int:
  if op == "copy": return i32(args[0])
  a, b = args
  if op == "add": return i32(a + b)
  if op == "sub": return i32(a - b)
  if op == "mul": return i32(a * b)
  if op == "lt": return int(a < b)
  if op == "le": return int(a <= b)
  if op == "eq": return int(a == b)
  if op == "ne": return int(a != b)
  raise ValueError(f"unknown arithmetic operation: {op}")


def successors(function, name):
  term = function.blocks[name].terminator
  if term is None: raise ValueError(f"unterminated block: {name}")
  return term.targets


def graph(function):
  if function.entry not in function.blocks: raise ValueError("missing entry")
  predecessors = {name: set() for name in function.blocks}
  for name in function.blocks:
    for target in successors(function, name):
      if target not in function.blocks: raise ValueError(f"unknown block: {target}")
      predecessors[target].add(name)
  reachable, todo = set(), [function.entry]
  while todo:
    name = todo.pop()
    if name in reachable: continue
    reachable.add(name)
    todo.extend(successors(function, name))
  return predecessors, reachable


def prune(function):
  _, reachable = graph(function)
  function.blocks = {name: block for name, block in function.blocks.items() if name in reachable}
  predecessors, _ = graph(function)
  for name, block in function.blocks.items():
```

```
    for phi in block.phis:
      phi.incoming = {p: value for p, value in phi.incoming.items() if p in predecessors[name]}
  return function


def operands(args): return {arg for arg in args if isinstance(arg, str)}


def format_ir(function):
  params = ", ".join(f"{p.name}: {p.kind.value}" for p in function.parameters)
  lines = [f"{'ssa ' if function.ssa else ''}func {function.name}({params}):"]
  for name, block in function.blocks.items():
    lines.append(f"{name}:")
    for phi in block.phis:
      incoming = ", ".join(f"{p}: {v}" for p, v in sorted(phi.incoming.items()))
      lines.append(f"  {phi.dest} = phi [{incoming}]")
    for instruction in block.instructions:
      lhs = f"{instruction.dest} = " if instruction.dest else ""
      lines.append(f"  {lhs}{instruction.op} {', '.join(map(str, instruction.args))}")
    term = block.terminator
    if term is not None: lines.append(f"  {term.op} {', '.join(map(str, (*term.args,
*term.targets)))}")
  return "\n".join(lines)
```

Trace: lower `x=a[0]; b[0]=2; return x`. The return uses the loaded scalar x even if a and b alias. Change it to `b[0]=2; return a[0]`; now the aliasing case returns the updated value. An expression-level common-subexpression rule must not identify the two loads across an intervening possibly aliased store.

30.3 Constructing blocks from structured syntax

An assignment emits its expression's operations followed by a copy into the mutable variable name. Temporary names are compiler-generated and cannot collide with source identifiers. Right-hand-side expressions are lowered before the assignment target, preserving the chosen evaluation order.

An `if` creates then, else, and merge blocks. Both arms end by jumping to the merge unless they already returned or transferred control. If both arms return, the merge is unreachable and is removed. A `while` creates a header, body, and exit. The current block jumps to the header; the header computes the condition and branches; a normal body completion jumps back to the header.

A stack of `(header,exit)` pairs implements nested `continue` and `break`: they target the innermost loop. After emitting any terminator, the frontend stops appending statements to that path. This prevents an

apparent assignment after a return from becoming an executed side effect.

`minicc/frontend.py`

```python
"""Lower a deliberately restricted Python AST to mutable-variable control-flow IR.

Python supplies lexing/parsing only. minicc defines its own i32 and buffer semantics.
Supported: assignments, integer expressions/comparisons, if, while, break, continue,
return, buffer indexing/stores. Unsupported Python syntax is rejected, never executed.
"""
import ast
from minicc.ir import Block, Function, Instruction, Kind, Parameter, Terminator, i32, prune


class Frontend:
  def __init__(self, source):
    module = ast.parse(source)
    if len(module.body) != 1 or not isinstance(module.body[0], ast.FunctionDef):
      raise ValueError("source must contain exactly one function")
    self.node = node = module.body[0]
    args = node.args
    if node.decorator_list or args.posonlyargs or args.kwonlyargs or args.defaults or
args.kw_defaults or args.vararg or args.kwarg:
      raise ValueError("decorators, defaults, and special argument forms are unsupported")
    if node.returns is not None and (not isinstance(node.returns, ast.Name) or node.returns.id !=
"int"):
      raise ValueError("return annotation must be int")
    parameters = []
    for arg in args.args:
      if arg.annotation is None: kind = Kind.I32
      elif isinstance(arg.annotation, ast.Name) and arg.annotation.id in ("int", "buffer"):
        kind = Kind.BUFFER if arg.annotation.id == "buffer" else Kind.I32
      else: raise ValueError("parameter annotations must be int or buffer")
      parameters.append(Parameter(arg.arg, kind))
    if len({p.name for p in parameters}) != len(parameters): raise ValueError("duplicate
parameter")
    self.function = Function(node.name, tuple(parameters), {"entry": Block()})
    self.buffers = {p.name for p in parameters if p.kind == Kind.BUFFER}
    self.current, self.counter, self.temp_counter = "entry", 0, 0
    self.loops = []

  @property
  def block(self): return self.function.blocks[self.current]

  def new_block(self, hint):
    self.counter += 1
    name = f"{hint}_{self.counter}"
    self.function.blocks[name] = Block()
    return name

  def emit(self, op, *args):
    self.temp_counter += 1
    name = f"$t{self.temp_counter}"
    self.block.instructions.append(Instruction(op, name, args))
    return name
```

```
  def scalar_name(self, node):
    if not isinstance(node, ast.Name) or node.id in self.buffers: raise ValueError("expected
scalar variable")
    return node.id

  def address(self, node):
    if not isinstance(node, ast.Subscript) or not isinstance(node.value, ast.Name) or
node.value.id not in self.buffers:
      raise ValueError("indexing requires a buffer parameter")
    return node.value.id, self.expression(node.slice)

  def expression(self, node):
    if isinstance(node, ast.Constant) and type(node.value) in (int, bool): return i32(node.value)
    if isinstance(node, ast.Name): return self.scalar_name(node)
    if isinstance(node, ast.Subscript): return self.emit("load", *self.address(node))
    if isinstance(node, ast.UnaryOp):
      if isinstance(node.op, ast.USub): return self.emit("sub", 0, self.expression(node.operand))
      if isinstance(node.op, ast.Not): return self.emit("eq", self.expression(node.operand), 0)
    if isinstance(node, ast.BinOp):
      operations = {ast.Add: "add", ast.Sub: "sub", ast.Mult: "mul"}
      if type(node.op) in operations:
        return self.emit(operations[type(node.op)], self.expression(node.left),
self.expression(node.right))
    if isinstance(node, ast.Compare) and len(node.ops) == 1:
      operations = {ast.Lt: "lt", ast.LtE: "le", ast.Eq: "eq", ast.NotEq: "ne", ast.Gt: "lt",
ast.GtE: "le"}
      op = type(node.ops[0])
      if op in operations:
        left, right = self.expression(node.left), self.expression(node.comparators[0])
        if op in (ast.Gt, ast.GtE): left, right = right, left
        return self.emit(operations[op], left, right)
    raise ValueError(f"unsupported expression: {ast.dump(node)}")

  def statements(self, statements):
    for node in statements:
      if self.block.terminator is not None: break
      self.statement(node)

  def statement(self, node):
    if isinstance(node, ast.Assign) and len(node.targets) == 1:
      value = self.expression(node.value)
      target = node.targets[0]
      if isinstance(target, ast.Name):
        self.block.instructions.append(Instruction("copy", self.scalar_name(target), (value,)))
      elif isinstance(target, ast.Subscript):
        self.block.instructions.append(Instruction("store", None, (*self.address(target),
value)))
      else: raise ValueError("unsupported assignment target")
    elif isinstance(node, ast.AugAssign) and isinstance(node.target, ast.Name):
      name = self.scalar_name(node.target)
      operations = {ast.Add: "add", ast.Sub: "sub", ast.Mult: "mul"}
      if type(node.op) not in operations: raise ValueError("unsupported augmented operation")
      self.block.instructions.append(Instruction(operations[type(node.op)], name, (name,
```

```
self.expression(node.value))))
    elif isinstance(node, ast.Return) and node.value is not None:
      self.block.terminator = Terminator("return", (self.expression(node.value),))
    elif isinstance(node, ast.If):
      condition = self.expression(node.test)
      yes, no, merge = self.new_block("then"), self.new_block("else"), self.new_block("merge")
      self.block.terminator = Terminator("branch", (condition,), (yes, no))
      for name, statements in ((yes, node.body), (no, node.orelse)):
        self.current = name
        self.statements(statements)
        if self.block.terminator is None: self.block.terminator = Terminator("jump", targets=
(merge,))
      self.current = merge
    elif isinstance(node, ast.While) and not node.orelse:
      head, body, end = self.new_block("while"), self.new_block("body"), self.new_block("end")
      self.block.terminator = Terminator("jump", targets=(head,))
      self.current = head
      condition = self.expression(node.test)
      self.block.terminator = Terminator("branch", (condition,), (body, end))
      self.loops.append((head, end))
      self.current = body
      self.statements(node.body)
      if self.block.terminator is None: self.block.terminator = Terminator("jump", targets=
(head,))
      self.loops.pop()
      self.current = end
    elif isinstance(node, (ast.Break, ast.Continue)):
      if not self.loops: raise ValueError("break/continue outside a loop")
      self.block.terminator = Terminator("jump", targets=(self.loops[-1][1 if isinstance(node,
ast.Break) else 0],))
    elif isinstance(node, ast.Pass): pass
    else: raise ValueError(f"unsupported statement: {ast.dump(node)}")

  def lower(self):
    self.statements(self.node.body)
    # Unreachable merge blocks may remain after both arms return. Give all blocks
    # a structural terminator, then prune. Reachable fallthrough is rejected.
    for block in self.function.blocks.values():
      if block.terminator is None: block.terminator = Terminator("missing_return")
    prune(self.function)
    if any(b.terminator.op == "missing_return" for b in self.function.blocks.values()):
      raise ValueError("function may fall through without returning")
    return self.function


def lower(source): return Frontend(source).lower()
```

Exercise: draw the graph for a loop whose body contains an early `continue`, a nested loop, and an early return. Identify every backedge and the destination of each `break`. Compare your graph with `format_ir(lower(source))`, then execute the mutable CFG before introducing SSA.

30.4 Definite assignment is a must analysis

Consider a variable x assigned only in one arm of an unknown branch, then used at the merge. A may-have-been-assigned analysis is insufficient. Every predecessor must establish x. For each block, intersect predecessor OUT sets, then add definitions performed by the block:

```
IN(entry) = parameters
IN(b) = intersection(OUT(p) for p in predecessors(b))
OUT(b) = IN(b) union DEF(b)
```

Initialize non-entry sets to the universe of names and iterate downward to a fixed point. Using the empty set would compute the wrong extremal solution for some cycles. After convergence, scan each block in order, checking every use before adding its instruction's result to the defined set.

A loop that assigns x only in its body cannot establish x after the loop, because the loop may execute zero times. An assignment `x=x+1` cannot establish the old x required by its own right-hand side. The tests exercise both errors. This conservative analysis runs before branch optimization; it can reject source that is safe only because a later constant-propagation pass could prove a branch always executes. That restriction is explicit rather than a hidden inconsistency.

30.5 Dominators, immediate dominators, and frontiers

The fixed-point dominator sets from Chapter 21 let us form a tree. For each non-entry block b, its immediate dominator is the strict dominator nearest to b. Strict dominators form a chain; in this implementation, the one with the deepest dominator set is the nearest.

The dominance frontier of d contains blocks where paths from the region dominated by d meet a path not strictly dominated by d. At a diamond merge, each arm has the merge in its frontier. A loop header can belong to its own frontier: it dominates the backedge source but does not strictly dominate itself. That self-frontier is how a loop-carried assignment discovers its header phi.

For each join b and predecessor p, walk from p up the immediate-dominator tree until reaching `idom(b)`, placing b in each visited node's frontier. Analyze reachable blocks only, and require a separate entry without predecessors to make the initial parameter definitions unambiguous.

minicc/analysis.py

```python
"""Fixed-point dominance/liveness, definite assignment, and IR/SSA verification."""
from minicc.ir import EFFECTS, PURE, Kind, graph, operands, successors


def dominators(function):
  predecessors, reachable = graph(function)
  entry = function.entry
  result = {b: ({b} if b == entry else set(reachable)) for b in reachable}
  while True:
    changed = False
    for b in sorted(reachable - {entry}):
      value = {b} | set.intersection(*(result[p] for p in predecessors[b] & reachable))
      if value != result[b]: result[b], changed = value, True
    if not changed: return result


def dominance_tree(function):
  dom = dominators(function)
  immediate = {b: max(dom[b] - {b}, key=lambda d: len(dom[d])) for b in dom if b !=
function.entry}
  children = {b: [] for b in dom}
  for b, parent in immediate.items(): children[parent].append(b)
  for values in children.values(): values.sort()
  return immediate, children


def dominance_frontiers(function):
  predecessors, reachable = graph(function)
  immediate, _ = dominance_tree(function)
  result = {b: set() for b in reachable}
  for b in reachable:
    if len(predecessors[b] & reachable) < 2: continue
    for parent in predecessors[b] & reachable:
      runner = parent
      while runner != immediate.get(b):
        result[runner].add(b)
        if runner not in immediate: break
        runner = immediate[runner]
  return result


def liveness(function):
  _, reachable = graph(function)
  uses, defs = {}, {}
  for name in reachable:
    block = function.blocks[name]
    defined, used = {p.dest for p in block.phis}, set()
    for instruction in block.instructions:
      used |= operands(instruction.args) - defined
      if instruction.dest is not None: defined.add(instruction.dest)
    used |= operands(block.terminator.args) - defined
    uses[name], defs[name] = used, defined
```

```
  incoming, outgoing = {b: set() for b in reachable}, {b: set() for b in reachable}
  while True:
    changed = False
    for b in sorted(reachable, reverse=True):
      live = set()
      for target in successors(function, b):
        block = function.blocks[target]
        live |= (incoming[target] - {p.dest for p in block.phis}) | operands(p.incoming[b] for p
in block.phis)
      value = uses[b] | (live - defs[b])
      if value != incoming[b] or live != outgoing[b]: incoming[b], outgoing[b], changed = value,
live, True
    if not changed: return incoming, outgoing


def definite_assignment(function):
  predecessors, reachable = graph(function)
  parameters = {p.name for p in function.parameters}
  definitions = {b: {i.dest for i in function.blocks[b].instructions if i.dest is not None} for b
in reachable}
  universe = parameters | set.union(set(), *definitions.values())
  incoming = {b: (set(parameters) if b == function.entry else set(universe)) for b in reachable}
  outgoing = {b: incoming[b] | definitions[b] for b in reachable}
  while True:
    changed = False
    for b in sorted(reachable):
      value = set(parameters) if b == function.entry else set.intersection(*(outgoing[p] for p in
predecessors[b] & reachable))
      after = value | definitions[b]
      if value != incoming[b] or after != outgoing[b]: incoming[b], outgoing[b], changed = value,
after, True
    if not changed: break
  for name in reachable:
    defined = set(incoming[name])
    block = function.blocks[name]
    for instruction in block.instructions:
      missing = operands(instruction.args) - defined
      if missing: raise ValueError(f"possibly uninitialized in {name}: {sorted(missing)}")
      if instruction.dest is not None: defined.add(instruction.dest)
    missing = operands(block.terminator.args) - defined
    if missing: raise ValueError(f"possibly uninitialized in {name}: {sorted(missing)}")


def verify(function):
  predecessors, reachable = graph(function)
  if reachable != set(function.blocks): raise ValueError("prune unreachable blocks before
verification")
  if predecessors[function.entry]: raise ValueError("entry must not have predecessors")
  parameters = {p.name: p.kind for p in function.parameters}
  if len(parameters) != len(function.parameters): raise ValueError("duplicate parameter")
  definitions = {p.name: (function.entry, -2) for p in function.parameters}
  buffers = {name for name, kind in parameters.items() if kind == Kind.BUFFER}
  for name, block in function.blocks.items():
```

```
    if block.phis and not function.ssa: raise ValueError("phis require SSA form")
    for phi in block.phis:
      if set(phi.incoming) != predecessors[name]: raise ValueError(f"phi edge mismatch in
{name}")
    for position, instruction in enumerate(block.instructions):
      if any(type(arg) not in (int, str) for arg in instruction.args): raise ValueError("operands
must be i32 literals or value names")
      if any(isinstance(arg, int) and not -(2**31) <= arg < 2**31 for arg in instruction.args):
raise ValueError("literal outside i32 domain")
      arity = (PURE | EFFECTS).get(instruction.op)
      if arity is None or len(instruction.args) != arity: raise ValueError(f"invalid instruction:
{instruction}")
      if (instruction.dest is None) != (instruction.op == "store"): raise ValueError("invalid
instruction result")
      if instruction.dest in buffers: raise ValueError("cannot redefine buffer parameter")
      if instruction.op in EFFECTS:
        if instruction.args[0] not in buffers: raise ValueError("memory operation requires a
buffer")
        scalar_args = instruction.args[1:]
      else: scalar_args = instruction.args
      if operands(scalar_args) & buffers: raise ValueError("buffer used as scalar")
    term = block.terminator
    if term.op not in ("return", "branch", "jump"): raise ValueError("invalid terminator")
    if len(term.args) != (0 if term.op == "jump" else 1) or len(term.targets) != {"return": 0,
"branch": 2, "jump": 1}[term.op]:
      raise ValueError("invalid terminator arity")
    if any(type(arg) not in (int, str) for arg in term.args): raise ValueError("invalid
terminator operand")
    if any(isinstance(arg, int) and not -(2**31) <= arg < 2**31 for arg in term.args): raise
ValueError("literal outside i32 domain")
    if operands(term.args) & buffers: raise ValueError("buffer used as scalar condition or
return")
    if function.ssa:
      for position, dest in [(-1, p.dest) for p in block.phis] + [(i, instruction.dest) for i,
instruction in enumerate(block.instructions) if instruction.dest is not None]:
        if dest in definitions: raise ValueError(f"duplicate SSA definition: {dest}")
        definitions[dest] = (name, position)
  if not function.ssa:
    definite_assignment(function)
    return
  dom = dominators(function)
  def check(args, block, position):
    for arg in args:
      if isinstance(arg, int):
        if not -(2**31) <= arg < 2**31: raise ValueError("literal outside i32 domain")
        continue
      if arg not in definitions: raise ValueError(f"undefined SSA value: {arg}")
      origin, index = definitions[arg]
      if origin not in dom[block] or (origin == block and index >= position):
        raise ValueError(f"{arg} does not dominate use in {block}")
  for name, block in function.blocks.items():
    for phi in block.phis:
      if operands(phi.incoming.values()) & buffers: raise ValueError("buffer phi unsupported")
```

```
        for parent, value in phi.incoming.items(): check((value,), parent,
len(function.blocks[parent].instructions))
    for position, instruction in enumerate(block.instructions): check(instruction.args, name,
position)
    check(block.terminator.args, name, len(block.instructions))
```

The same module verifies block structure, instruction arity, buffer/scalar use, literal range, unique SSA definitions, phi edge sets, and dominance of ordinary and edge uses. Verification is part of the pipeline. A malformed program should fail close to the pass that created it, rather than survive until a native crash.

30.6 Phi placement needs an iterated frontier

For a variable with definitions in blocks D, put phis at their dominance frontiers. A newly inserted phi is itself a definition and may require another phi in a further frontier. Continue until no new placement is needed. This is the iterated dominance frontier.

Placing every possible phi is correct but noisy. Pruned SSA inserts one only when the original variable is live into the candidate block. If both arms assign a dead variable that is never read, the merge needs no phi. Compute pre-SSA liveness before placement; there are no phi edge uses yet.

For the running example, i and total receive header phis joining initial values with backedge values. The conditional update of total receives a second phi at the branch merge. The result of that merge feeds total's loop-header phi. Placing phis only at the first visible diamond would miss the loop-carried value.

30.7 Renaming with scoped stacks

Maintain one stack of current SSA versions per mutable variable. Parameters seed their stacks. Walking the dominator tree, first assign names to block phis, then rewrite instruction operands using current stack tops and push fresh names for definitions. After the terminator, fill every successor phi's incoming operand for this edge from the current stacks.

Recursively process dominated children, then pop exactly the definitions introduced in this block. Without popping, a name from one branch leaks into its sibling. Filling phi edges at the predecessor also handles loop

backedges: the header phi name already exists when the body is renamed.

`minicc/ssa.py`

```python
"""Pruned iterated-dominance-frontier phi placement and dominator-tree renaming."""
from collections import defaultdict
from copy import deepcopy
from minicc.analysis import dominance_frontiers, dominance_tree, liveness, verify
from minicc.ir import Phi


def to_ssa(original):
  if original.ssa: raise ValueError("already in SSA")
  function = deepcopy(original)
  verify(function)
  frontiers = dominance_frontiers(function)
  _, children = dominance_tree(function)
  live_in, _ = liveness(function)
  defsites = defaultdict(set)
  for parameter in function.parameters: defsites[parameter.name].add(function.entry)
  for name, block in function.blocks.items():
    for instruction in block.instructions:
      if instruction.dest is not None: defsites[instruction.dest].add(name)
  for variable, sites in sorted(defsites.items()):
    pending, placed = sorted(sites), set()
    while pending:
      origin = pending.pop()
      for target in sorted(frontiers[origin]):
        if target in placed or variable not in live_in[target]: continue
        function.blocks[target].phis.append(Phi(variable, variable))
        placed.add(target)
        if target not in sites: pending.append(target)
  stacks = defaultdict(list)
  counters = defaultdict(int)
  for parameter in function.parameters: stacks[parameter.name].append(parameter.name)

  def read(operand):
    if isinstance(operand, int): return operand
    if not stacks[operand]: raise ValueError(f"uninitialized during rename: {operand}")
    return stacks[operand][-1]

  def define(variable, pushed):
    counters[variable] += 1
    name = f"{variable}.{counters[variable]}"
    stacks[variable].append(name)
    pushed.append(variable)
    return name

  def rename(name):
    block, pushed = function.blocks[name], []
    for phi in block.phis: phi.dest = define(phi.base, pushed)
    for instruction in block.instructions:
      instruction.args = tuple(read(arg) for arg in instruction.args)
      if instruction.dest is not None: instruction.dest = define(instruction.dest, pushed)
    block.terminator.args = tuple(read(arg) for arg in block.terminator.args)
    for target in block.terminator.targets:
```

```
      for phi in function.blocks[target].phis: phi.incoming[name] = read(phi.base)
    for child in children[name]: rename(child)
    for variable in reversed(pushed): stacks[variable].pop()

  rename(function.entry)
  function.ssa = True
  verify(function)
  return function
```

Acceptance: inspect the generated loop phis, run zero-trip and multi-trip loops, and mutate a phi edge or an ordinary use to demonstrate verifier rejection. Then generate small trusted programs and compare mutable execution, SSA execution, and a source oracle. The next chapter adds transformations on this verified representation.

31. Optimization with Executable-Path Reasoning

Constant folding evaluates an operation with known arguments. Conditional constant propagation also reasons about which control-flow edges can execute. That difference lets it ignore an unreachable phi input while remaining conservative about unknown runtime values.

31.1 Three lattice states, not two

Each SSA value is initially UNDEFINED, a known constant, or OVERDEFINED. UNDEFINED means analysis has not established information yet; it does not mean an uninitialized source variable is legal. The frontend already rejected those. OVERDEFINED means the value cannot be represented as one constant by this analysis. Parameters begin OVERDEFINED.

The join rules are:

Join	Result
UNDEFINED and x	x
Constant c and the same c	c
Different constants	OVERDEFINED
OVERDEFINED and any established value	OVERDEFINED

A value's state only moves upward. It can become a constant, then become OVERDEFINED when another path contributes a different value; it never becomes a constant again after that. Executable edges also only accumulate. These monotonic properties give a finite fixed-point computation.

31.2 Control flow participates in the proof

Start with the entry executable and no executable edges. Evaluate instructions in executable blocks. An unconditional jump enables its edge. A constant condition enables only its selected edge. An OVERDEFINED condition enables both edges. An UNDEFINED condition enables neither yet; later scans may establish more information and enable the correct edges.

For a phi, join operands only from executable incoming edges. Consider:

```
def f(n):
    flag = 1
    if flag:
        x = 5
    else:
        x = n
    return x * 2
```

Joining every syntactic incoming operand would combine 5 with unknown n and lose the constant. Executable-edge reasoning proves the else edge unavailable, so x is 5 and the result is 10. After rewriting, the dead arm is pruned and the unused phi disappears.

Now replace `flag=1` with `flag=n`. Both branches become executable. The phi is OVERDEFINED; returning 10 unconditionally would be wrong. Tests need both cases to distinguish valid path reasoning from an optimizer that simply follows the first path it encounters.

31.3 Loops require revisiting optimistic facts

For `x=0; while x<n: x+=1; return x`, the header phi first sees the entry value 0. If the body can execute, its backedge eventually contributes 1, then the phi becomes OVERDEFINED. The analysis must revisit instructions and branch decisions affected by that change. Assuming the initially observed zero is permanent would miscompile every multi-iteration case.

This implementation scans to a fixed point for readability. A production sparse implementation would keep value-user and executable-block worklists, evaluating only affected nodes. The lattice, edge policy, and correctness conditions remain the same; the scan is not a claim about compile-time scalability.

31.4 Rewrite only after the analysis converges

Once facts stabilize, substitute constants into operands and pure results, convert constant branches to jumps, remove unreachable blocks, and repair phi edge maps. Keep trapping loads and stores in order. The result is reverified before any later pass consumes it.

An unreachable bounds error can disappear: that instruction cannot execute. A reachable load whose result is unused must remain: its failure is still observable. This distinction is precisely why “dead result” and “dead instruction” are different ideas.

`minicc/optimize.py`

```python
"""Sparse conditional constant propagation and effect-preserving dead code removal.

The worklist is expressed as a fixed-point scan to keep the lattice mechanics visible.
Values move only UNDEFINED -> constant -> OVERDEFINED; executable edges only grow.
"""
from copy import deepcopy
from enum import Enum
from minicc.analysis import verify
from minicc.ir import EFFECTS, PURE, Terminator, alu, operands, prune


class State(Enum):
  UNDEFINED = 0
  OVERDEFINED = 1


def join(a, b):
  if a == State.UNDEFINED: return b
  if b == State.UNDEFINED or a == b: return a
  return State.OVERDEFINED


def sccp(original):
  if not original.ssa: raise ValueError("SCCP requires SSA")
  verify(original)
  function = deepcopy(original)
  values = {p.name: State.OVERDEFINED for p in function.parameters}
  executable, edges = {function.entry}, set()

  def value(arg): return arg if isinstance(arg, int) else values.get(arg, State.UNDEFINED)

  def update(dest, result):
    nonlocal changed
    combined = join(values.get(dest, State.UNDEFINED), result)
    if combined != values.get(dest, State.UNDEFINED):
      values[dest], changed = combined, True

  while True:
    changed = False
    for name, block in function.blocks.items():
      if name not in executable: continue
      for phi in block.phis:
        result = State.UNDEFINED
        for parent, arg in phi.incoming.items():
          if (parent, name) in edges: result = join(result, value(arg))
        update(phi.dest, result)
      for instruction in block.instructions:
        if instruction.dest is None: continue
        args = [value(arg) for arg in instruction.args]
        if instruction.op in EFFECTS or State.OVERDEFINED in args: result = State.OVERDEFINED
        elif State.UNDEFINED in args: result = State.UNDEFINED
        else: result = alu(instruction.op, args)
```

```
        update(instruction.dest, result)
      term = block.terminator
      if term.op == "jump": targets = term.targets
      elif term.op == "return": targets = ()
      else:
        condition = value(term.args[0])
        if condition == State.UNDEFINED: targets = ()
        elif condition == State.OVERDEFINED: targets = term.targets
        else: targets = (term.targets[0 if condition else 1],)
      for target in targets:
        if (name, target) not in edges: edges.add((name, target)); changed = True
        if target not in executable: executable.add(target); changed = True
    if not changed: break

  def replace(arg):
    candidate = value(arg)
    return candidate if isinstance(candidate, int) else arg

  for name, block in function.blocks.items():
    for phi in block.phis: phi.incoming = {p: replace(arg) for p, arg in phi.incoming.items()}
    for instruction in block.instructions:
      instruction.args = tuple(replace(arg) for arg in instruction.args)
      if instruction.op in PURE and isinstance(value(instruction.dest), int):
        instruction.op, instruction.args = "copy", (value(instruction.dest),)
    term = block.terminator
    term.args = tuple(replace(arg) for arg in term.args)
    if term.op == "branch" and isinstance(term.args[0], int):
      block.terminator = Terminator("jump", targets=(term.targets[0 if term.args[0] else 1],))
  prune(function)
  verify(function)
  return function


def eliminate_dead_code(original):
  if not original.ssa: raise ValueError("dead-code removal requires SSA")
  function = deepcopy(original)
  definitions = {}
  roots = set()
  for block in function.blocks.values():
    roots |= operands(block.terminator.args)
    for phi in block.phis: definitions[phi.dest] = operands(phi.incoming.values())
    for instruction in block.instructions:
      if instruction.dest is not None: definitions[instruction.dest] = operands(instruction.args)
      if instruction.op in EFFECTS: roots |= operands(instruction.args)
  live, pending = set(), list(roots)
  while pending:
    value = pending.pop()
    if value in live: continue
    live.add(value)
    pending.extend(definitions.get(value, ()))
  for block in function.blocks.values():
    block.phis = [phi for phi in block.phis if phi.dest in live]
    block.instructions = [i for i in block.instructions if i.op in EFFECTS or i.dest in live]
```

```
    verify(function)
    return function


def optimize(function): return eliminate_dead_code(sccp(function))
```

The transfer function deliberately declines identities that require more reasoning. For instance, an unknown integer multiplied by zero can be simplified in this language, but basic constant propagation leaves it unknown. Missing a valid optimization is different from making an invalid one. Add identities separately with their semantic domain and counterexample tests.

31.5 Dead-code elimination is backward reachability

Seed a live-value set with return and branch operands plus the operands needed by effectful instructions. Follow each live definition backward through its dependencies; a phi contributes its incoming values. Remove pure instructions and phis whose results are not reached. Preserve all reachable loads and stores, even when their scalar results are dead.

A cycle of phis and arithmetic with no connection to an observable root can be removed. Conversely, a value that computes an index for a store remains live even if no scalar return depends on it. Thinking of DCE as graph reachability handles both without relying on source-line intuition.

31.6 Experiments that can falsify the optimizer

The test suite compares pre-optimization and post-optimization execution against trusted source oracles, then checks separately specified cases that differ from Python:

- Wrapping arithmetic at signed boundaries and in large products.
- Unknown branch conditions, not only constant conditions.
- Loops with zero, one, and multiple iterations.
- Stores through aliased buffers followed by loads through the other name.
- An unused invalid load, which must still raise an error.

- A store before a trapping load, which remains visible after failure.
- A later store after that load, which must not execute.

Run these directly:

```
.venv/bin/python -m pytest tests/test_minicc.py -q
```

Extension: add copy propagation, algebraic simplification, global value numbering, and loop-invariant code motion one at a time. For load motion, first implement an effect/alias model capable of proving that intervening writes and traps permit the move. Do not reuse pure-expression legality rules for memory operations. For each new pass, verify the output and retain a minimized test that would fail if its key precondition were omitted.

32. Executing SSA, Destroying Phis, and Allocating Registers

A verified optimized graph still needs an execution strategy. This chapter implements three: an SSA interpreter for debugging, a constrained teaching machine for register allocation, and a native C backend. Each exposes different errors and delegates different responsibilities.

32.1 Phi evaluation is simultaneous

On entry to a block, select each phi operand using the predecessor actually taken. Read all selected values before writing any phi result. For a loop that swaps a and b, sequentially assigning `a=b` and then `b=a` loses the old a. A dictionary of pending results followed by one environment update preserves the parallel semantics.

The interpreter bounds-checks accesses before indexing so Python's negative-index convention cannot leak into the language. It normalizes loaded and stored integers to i32 and executes instructions in source-effect order. A block-step budget catches nontermination during experiments; this is a debugging facility, not a proof of termination.

`minicc/interpreter.py`

```python
"""Execute mutable or SSA IR with ordered memory effects and simultaneous phi updates."""
from minicc.analysis import verify
from minicc.ir import Kind, alu, i32


def bind(function, arguments):
  if len(arguments) != len(function.parameters): raise ValueError("wrong argument count")
  result = {}
  for parameter, value in zip(function.parameters, arguments):
    if parameter.kind == Kind.I32:
      if not isinstance(value, int): raise ValueError("scalar parameters require Python
integers")
      result[parameter.name] = i32(value)
    else:
      if not hasattr(value, "__len__") or not hasattr(value, "__getitem__") or not hasattr(value,
"__setitem__"):
        raise ValueError("buffer parameter requires mutable indexed storage")
      result[parameter.name] = value
  return result


def memory(op, args):
  buffer, index = args[:2]
  if not 0 <= index < len(buffer): raise IndexError(f"buffer index {index} outside [0,
{len(buffer)})")
  if op == "load": return i32(buffer[index])
  buffer[index] = i32(args[2])
  return None


def execute(function, *arguments, max_steps=100000):
  verify(function)
  env = bind(function, arguments)
  current, previous = function.entry, None
  def read(value): return value if isinstance(value, int) else env[value]
  for _ in range(max_steps):
    block = function.blocks[current]
    incoming = {phi.dest: read(phi.incoming[previous]) for phi in block.phis}
    env.update(incoming)
    for instruction in block.instructions:
      args = [read(arg) for arg in instruction.args]
      value = memory(instruction.op, args) if instruction.op in ("load", "store") else
alu(instruction.op, args)
      if instruction.dest is not None: env[instruction.dest] = value
    term = block.terminator
    if term.op == "return": return read(term.args[0])
    target = term.targets[0] if term.op == "jump" else term.targets[0 if read(term.args[0]) else
1]
    previous, current = current, target
  raise RuntimeError("execution step limit exceeded")
```

Experiment: compile a loop initialized with a=1 and b=2 that swaps them each iteration. Return `10*a+b`. Even iteration counts must return 12, odd counts 21. This small program exposes phi-copy bugs that ordinary sums rarely detect.

32.2 Phi destruction belongs on edges

An SSA phi in block merge says which assignment happens for each incoming edge. Lower it to copies on those edges. Putting a predecessor-specific copy unconditionally at the end of a branching predecessor can change the value on the wrong path. Putting all copies at the merge loses the identity of the incoming edge.

The implementation creates an edge block for every incoming edge to a phi-bearing block, redirects that branch or jump, saves all phi sources into fresh temporaries, then writes all destinations. Saving every source is conservative but handles arbitrary copy cycles. A later parallel-copy scheduler could emit acyclic copies directly and use one scratch value only when breaking a cycle.

This transformation leaves the graph in a non-SSA form: the same destination may be assigned on several incoming edges. Re-run definite assignment and structural verification under that form. Do not incorrectly apply the single-definition verifier after intentionally destroying SSA.

32.3 Liveness gives an interference graph

At each program point, two distinct live values need different locations unless a separate value- equality argument permits sharing. Compute block live-in/live-out sets, then scan instructions backward. For a definition d, connect d to every value live after its definition, remove d from that live set, and add the instruction's operands.

Parameters are installed before entering the function. This implementation connects all scalar parameters to one another, including unused ones, so initialization cannot overwrite a live parameter. A better allocator could avoid binding dead parameters, but omitting both protections is incorrect. Buffers remain separate typed resources and never receive scalar-register colors.

Color the interference graph greedily in descending degree order. Pick an available register color; if none is available, assign a distinct spill slot. This policy is simple and deterministic but not optimal. It does not

perform coalescing, live-range splitting, rematerialization, or spill-slot reuse. Those are subsequent compiler projects with measurable effects on this baseline.

32.4 Execute the allocation, not just inspect it

The teaching machine reads and writes the allocated physical locations. If the allocator assigns interfering values the same register, the computation is actually corrupted and the source-oracle test fails. A test that only checks the allocator's own graph would be weaker: the graph and allocator could agree on the same incorrect assumptions.

Spilled values live in a frame and their reads and writes are counted. The teaching instruction set permits spill-slot operands directly. A register-only hardware ISA would require explicit reload and spill instructions, scratch-register management, stack layout, and calling-convention support. The counts are algorithmic traffic, not measured CPU or GPU cycles.

`minicc/machine.py`

```python
"""Phi destruction, graph coloring, and a bounded-register interpreter with spills.

This teaching machine permits spill-slot operands directly. It counts their traffic,
not host CPU cycles. It is not a physical ISA encoder or a hardware cost model.
"""
from copy import deepcopy
from dataclasses import dataclass
from itertools import combinations
from minicc.analysis import liveness, verify
from minicc.interpreter import bind, memory
from minicc.ir import Block, Instruction, Kind, Terminator, alu, operands


def destroy_phis(original):
  if not original.ssa: raise ValueError("phi destruction requires SSA")
  verify(original)
  function = deepcopy(original)
  counter = 0
  for parent, block in list(function.blocks.items()):
    targets = list(block.terminator.targets)
    for index, target in enumerate(targets):
      phis = function.blocks[target].phis
      if not phis: continue
      counter += 1
      edge = f"$edge{counter}"
      reads, writes = [], []
      # Save every source before writing any destination: parallel-copy semantics.
      for i, phi in enumerate(phis):
        temporary = f"$edge{counter}_copy{i}"
        reads.append(Instruction("copy", temporary, (phi.incoming[parent],)))
        writes.append(Instruction("copy", phi.dest, (temporary,)))
      function.blocks[edge] = Block(reads + writes, Terminator("jump", targets=(target,)))
      targets[index] = edge
    block.terminator.targets = tuple(targets)
  for block in function.blocks.values(): block.phis.clear()
  function.ssa = False
  verify(function)
  return function


def interference(function):
  if function.ssa or any(b.phis for b in function.blocks.values()): raise ValueError("destroy
phis first")
  verify(function)
  _, outgoing = liveness(function)
  buffers = {p.name for p in function.parameters if p.kind == Kind.BUFFER}
  neighbors = {}
  def ensure(values):
    for value in values - buffers: neighbors.setdefault(value, set())
  def connect(a, b):
    if a == b or a in buffers or b in buffers: return
    ensure({a, b})
```

```
    neighbors[a].add(b); neighbors[b].add(a)
  parameters = {p.name for p in function.parameters if p.kind == Kind.I32}
  ensure(parameters)
  # Arguments are installed together before entry. Protect even unused parameters
  # from clobbering another parameter during that initial binding.
  for a, b in combinations(sorted(parameters), 2): connect(a, b)
  for name, block in function.blocks.items():
    live = (outgoing[name] | operands(block.terminator.args)) - buffers
    ensure(live)
    for instruction in reversed(block.instructions):
      if instruction.dest is not None:
        ensure({instruction.dest})
        live.discard(instruction.dest)
        for value in live: connect(instruction.dest, value)
      live |= operands(instruction.args) - buffers
      ensure(live)
  return neighbors


@dataclass(frozen=True)
class Location:
  kind: str
  index: int


@dataclass
class Allocation:
  function: object
  locations: dict[str, Location]
  registers: int
  slots: int
  neighbors: dict[str, set[str]]


def allocate(function, registers=4):
  if registers < 1: raise ValueError("at least one register is required")
  neighbors = interference(function)
  locations, spills = {}, 0
  # Deterministic largest-degree-first greedy coloring; not optimal coloring.
  for value in sorted(neighbors, key=lambda v: (-len(neighbors[v]), v)):
    unavailable = {locations[n].index for n in neighbors[value] if n in locations and
locations[n].kind == "register"}
    available = next((r for r in range(registers) if r not in unavailable), None)
    if available is None:
      locations[value] = Location("slot", spills)
      spills += 1
    else: locations[value] = Location("register", available)
  allocation = Allocation(function, locations, registers, spills, neighbors)
  for value, adjacent in neighbors.items():
    for other in adjacent:
      if locations[value] == locations[other]: raise ValueError("interfering values share
storage")
  return allocation
```

```python
def execute_allocated(allocation, *arguments, max_steps=100000):
  function = allocation.function
  env = bind(function, arguments)
  buffers = {p.name: env[p.name] for p in function.parameters if p.kind == Kind.BUFFER}
  storage = {"register": [0]*allocation.registers, "slot": [0]*allocation.slots}
  counts = {"spill_reads": 0, "spill_writes": 0, "instructions": 0}
  def write(name, value):
    location = allocation.locations[name]
    storage[location.kind][location.index] = value
    if location.kind == "slot": counts["spill_writes"] += 1
  def read(value):
    if isinstance(value, int): return value
    if value in buffers: return buffers[value]
    location = allocation.locations[value]
    if location.kind == "slot": counts["spill_reads"] += 1
    return storage[location.kind][location.index]
  for parameter in function.parameters:
    if parameter.kind == Kind.I32: write(parameter.name, env[parameter.name])
  current = function.entry
  for _ in range(max_steps):
    block = function.blocks[current]
    for instruction in block.instructions:
      counts["instructions"] += 1
      args = [read(value) for value in instruction.args]
      result = memory(instruction.op, args) if instruction.op in ("load", "store") else
alu(instruction.op, args)
      if instruction.dest is not None: write(instruction.dest, result)
    term = block.terminator
    if term.op == "return": return read(term.args[0]), counts
    current = term.targets[0] if term.op == "jump" else term.targets[0 if read(term.args[0]) else
1]
  raise RuntimeError("execution step limit exceeded")
```

Experiment: run the same compiled function with register budgets 1, 2, 4, and 16. Check semantic agreement before comparing spill slots and traffic. Then implement a copy-propagation or coalescing pass and determine whether it reduces actual dynamic spill traffic, not only graph size.

32.5 Native C needs a declared numerical and memory contract

A naive C emitter prints signed `a+b`, but overflowing signed arithmetic is undefined in C. The backend converts operands to uint32, computes modulo arithmetic there, and copies the resulting bit pattern into int32 with `memcpy`. Comparisons still use signed operands. The minimum i32 literal is emitted without depending on a positive 2147483648 having int type.

Every buffer parameter has a pointer and explicit length. Every access checks its signed index before dereferencing. An error pointer communicates a trap to the wrapper, which raises an `IndexError` after the native function returns. Earlier writes remain visible. Scalar results are not mistaken for error codes; a legitimate result of zero remains distinguishable from failure.

The wrapper declares every `ctypes` argument type and return type. It accepts only aligned, writable, contiguous rank-one int32 buffers. Rejecting a noncontiguous view avoids treating a logical stride as physical adjacency. Writable buffers simplify the contract even for functions that only read; relaxing that guard requires per-parameter effect information.

Phi assignments in C are again emitted with parallel-copy semantics on branch edges. All SSA scalar names receive safe generated C identifiers, so source identifiers cannot inject generated code. The original source is parsed structurally, never interpolated as C statements.

`minicc/c_backend.py`

```python
"""Render verified SSA as bounds-checked C and execute it through a declared ABI."""
import ctypes
import shutil
import subprocess
from pathlib import Path
from tempfile import TemporaryDirectory
import numpy as np
from minicc.analysis import verify
from minicc.ir import Kind


def render_c(function):
  if not function.ssa: raise ValueError("C backend expects SSA")
  verify(function)
  names = {}
  def symbol(name):
    if name not in names: names[name] = f"v{len(names)}"
    return names[name]
  def operand(value):
    if isinstance(value, int): return f"INT32_C({value})" if value != -(2**31) else "(-
INT32_C(2147483647)-1)"
    return symbol(value)
  arguments = []
  parameters = {p.name for p in function.parameters}
  for parameter in function.parameters:
    name = symbol(parameter.name)
    arguments.append(f"int32_t *{name}" if parameter.kind == Kind.BUFFER else f"int32_t {name}")
    if parameter.kind == Kind.BUFFER: arguments.append(f"int32_t {name}_length")
  arguments.append("int32_t *error")
  labels = {name: f"block_{i}" for i, name in enumerate(function.blocks)}
  for block in function.blocks.values():
    for phi in block.phis: symbol(phi.dest)
    for instruction in block.instructions:
      if instruction.dest is not None: symbol(instruction.dest)
  lines = ["#include <stdint.h>", "#include <string.h>",
           # Bit-preserving conversion avoids signed-overflow UB and avoids
           # relying on an out-of-range unsigned-to-signed C cast.
           "static int32_t signed_bits(uint32_t x) { int32_t y; memcpy(&y, &x, sizeof(y)); return
y; }",
           f"int32_t compiled({', '.join(arguments)}) {{", "  *error = 0;"]
  for name, c_name in names.items():
    if name not in parameters: lines.append(f"  int32_t {c_name};")
  lines.append(f"  goto {labels[function.entry]};")
  def edge(parent, target, indent):
    lines.append(indent + "{")
    phis = function.blocks[target].phis
    for i, phi in enumerate(phis): lines.append(indent + f"  int32_t edge_{i} =
{operand(phi.incoming[parent])};")
    for i, phi in enumerate(phis): lines.append(indent + f"  {symbol(phi.dest)} = edge_{i};")
    lines.append(indent + f"  goto {labels[target]};")
    lines.append(indent + "}")
  comparisons = {"lt": "<", "le": "<=", "eq": "==", "ne": "!="}
```

```
  arithmetic = {"add": "+", "sub": "-", "mul": "*"}
  for name, block in function.blocks.items():
    lines.append(labels[name] + ": ;")
    for instruction in block.instructions:
      args = [operand(arg) for arg in instruction.args]
      if instruction.op in ("load", "store"):
        base, index = args[:2]
        lines.append(f"  if ({index} < 0 || {index} >= {base}_length) {{ *error = 1; return 0;
}}")
        statement = f"{base}[{index}] = {args[2]}" if instruction.op == "store" else f"
{symbol(instruction.dest)} = {base}[{index}]"
      else:
        if instruction.op == "copy": expression = args[0]
        elif instruction.op in comparisons: expression = f"({args[0]}
{comparisons[instruction.op]} {args[1]})"
        else: expression = f"signed_bits((uint32_t){args[0]} {arithmetic[instruction.op]}
(uint32_t){args[1]})"
        statement = f"{symbol(instruction.dest)} = {expression}"
      lines.append(f"  {statement};")
    term = block.terminator
    if term.op == "return": lines.append(f"  return {operand(term.args[0])};")
    elif term.op == "jump": edge(name, term.targets[0], "  ")
    else:
      lines.append(f"  if ({operand(term.args[0])})")
      edge(name, term.targets[0], "  ")
      lines.append("  else")
      edge(name, term.targets[1], "  ")
  lines.append("}")
  return "\n".join(lines)


class NativeFunction:
  def __init__(self, function):
    compiler = shutil.which("gcc")
    if compiler is None: raise RuntimeError("gcc is required for native execution")
    self.function, self.source = function, render_c(function)
    self.directory = TemporaryDirectory(prefix="minicc-")
    source, library = Path(self.directory.name)/"kernel.c", Path(self.directory.name)/"kernel.so"
    source.write_text(self.source)
    subprocess.run([compiler, "-std=c11", "-O2", "-shared", "-fPIC", str(source), "-o",
str(library)],
                   check=True, capture_output=True, text=True)
    self.library = ctypes.CDLL(str(library))
    self.call = self.library.compiled
    types = []
    for parameter in function.parameters:
      types.extend([ctypes.POINTER(ctypes.c_int32), ctypes.c_int32] if parameter.kind ==
Kind.BUFFER else [ctypes.c_int32])
    self.call.argtypes, self.call.restype = [*types, ctypes.POINTER(ctypes.c_int32)],
ctypes.c_int32

  def __call__(self, *arguments):
    if len(arguments) != len(self.function.parameters): raise ValueError("wrong argument count")
```

```
    native = []
    for parameter, value in zip(self.function.parameters, arguments):
      if parameter.kind == Kind.I32:
        if not isinstance(value, int): raise ValueError("scalar parameters require Python
integers")
        native.append(ctypes.c_int32(value))
      else:
        if not isinstance(value, np.ndarray) or value.dtype != np.int32 or value.ndim != 1 or not
value.flags.c_contiguous or not value.flags.aligned or not value.flags.writeable:
          raise ValueError("native buffers require aligned, writable, contiguous rank-one int32
arrays")
        if len(value) >= 2**31: raise ValueError("buffer exceeds i32 length domain")
        native.extend([value.ctypes.data_as(ctypes.POINTER(ctypes.c_int32)),
ctypes.c_int32(len(value))])
    error = ctypes.c_int32()
    result = self.call(*native, ctypes.byref(error))
    if error.value: raise IndexError("compiled buffer access out of bounds")
    return result
```

32.6 What these backends establish

The native backend delegates instruction selection, scheduling, register allocation, assembly, and linking to GCC. The register-machine backend exposes allocation but does not emit a physical ISA. Together they make two useful boundaries visible instead of pretending the C emitter performs every backend task.

Native execution has no interpreter step budget; do not run a possibly nonterminating test in-process without a process timeout. The generated library and its temporary source remain owned by the `NativeFunction` object for its lifetime. This is an experimental JIT, not a persistent executable cache or a sandbox for adversarial programs.

Acceptance: execute a program through mutable IR, SSA, optimized SSA, the allocated machine, and native C. Compare returns, final buffers, and failures against independent expected behavior. Keep the generated C when diagnosing a native-only mismatch, and identify the first stage that changes the specified semantics.

33. A Dynamic Tensor Compiler with Fusion and Planned Memory

The guarded square kernel in Chapter 23 showed a dispatch contract, but its program was fixed. `tensorjit` now compiles a tensor expression graph into float32 C loops with symbolic dimensions, pointwise fusion, materialization boundaries, and actual temporary-buffer reuse. It is a second executable compiler path, designed to connect the earlier isolated labs into one runtime.

```
.venv/bin/python -m tensorjit
.venv/bin/python -m pytest tests/test_tensorjit.py -q
```

A matrix multiplication, bias, ReLU, and row sum run for several `(batch,inner,width)` shapes using one compiled library. The example emits two kernels: a matrix product and a reduction that inlines its bias and ReLU producers. It does not recompile when a runtime dimension changes within bounds.

33.1 Symbols are identities with constraints

A `Dim("N",1,64)` represents one dimension in `[1,64]`. Reusing N in multiple input positions asserts that their actual dimensions agree. Two symbols with different names are not assumed equal merely because they have equal bounds. Symbols with the same name but inconsistent bounds are rejected. This makes shape assumptions inspectable rather than relying on an accidental example shape.

Broadcasting aligns trailing axes. Equal dimensions match; a static dimension of one repeats along the other extent. Otherwise graph construction fails. Matrix multiplication requires rank two and identical reduction dimensions. These are deliberately conservative proofs: runtime-disjunctive broadcasting, symbolic affine equalities, zero-length dimensions, and arbitrary-strided inputs are not yet supported.

Nodes are immutable, and shared subexpressions are shared by object identity. The compiler does not yet hash-cons distinct structurally equal nodes. Reusing a node expresses intended sharing; constructing an equal-looking expression twice may duplicate work. That is a useful experiment for a future common-subexpression pass.

`tensorjit/graph.py`

```python
"""Immutable tensor expressions with bounded symbolic dimensions.

Supported operations are float32 and pure: add/mul/ReLU, transpose, reduction,
and rank-two matrix multiplication. Symbols identify equal logical dimensions.
"""
from __future__ import annotations
from dataclasses import dataclass
import math


@dataclass(frozen=True)
class Dim:
  name: str
  lower: int = 1
  upper: int = 4096

  def __post_init__(self):
    if not isinstance(self.name, str) or not self.name.isidentifier() or type(self.lower) is not
int or type(self.upper) is not int or self.lower < 1 or self.upper < self.lower or self.upper >=
2**31:
      raise ValueError("invalid dimension name or bounds")


@dataclass(frozen=True, eq=False)
class Expr:
  op: str
  shape: tuple[int | Dim, ...]
  sources: tuple[Expr, ...] = ()
  arg: object = None

  def __add__(self, other): return binary("add", self, other)
  def __mul__(self, other): return binary("mul", self, other)
  def __neg__(self): return binary("mul", -1, self)
  def __sub__(self, other): return binary("add", self, -(other if isinstance(other, Expr) else
constant(other)))
  def __rsub__(self, other): return binary("add", other, -self)
  def __radd__(self, other): return binary("add", other, self)
  def __rmul__(self, other): return binary("mul", other, self)
  def relu(self): return Expr("relu", self.shape, (self,))

  def expand_dims(self, axis):
    if not -len(self.shape)-1 <= axis <= len(self.shape): raise ValueError("invalid inserted
axis")
    axis %= len(self.shape)+1
    return Expr("expand_dims", self.shape[:axis]+(1,)+self.shape[axis:], (self,), axis)

  def broadcast_to(self, shape):
    shape = tuple(shape)
    if any(not isinstance(d, Dim) and (type(d) is not int or d < 1 or d >= 2**31) for d in
shape):
      raise ValueError("invalid target shape")
    if broadcast(self.shape, shape) != shape: raise ValueError("cannot broadcast to target
```

```
shape")
    return Expr("broadcast", shape, (self,))

  def transpose(self):
    if len(self.shape) != 2: raise ValueError("transpose requires a matrix")
    return Expr("transpose", self.shape[::-1], (self,))

  def sum(self, axis=-1):
    if not self.shape or not -len(self.shape) <= axis < len(self.shape): raise
ValueError("invalid reduction axis")
    axis %= len(self.shape)
    return Expr("sum", self.shape[:axis] + self.shape[axis+1:], (self,), axis)

  def __matmul__(self, other):
    if not isinstance(other, Expr) or len(self.shape) != 2 or len(other.shape) != 2 or
self.shape[1] != other.shape[0]:
      raise ValueError("matmul requires rank-two operands with equal reduction dimensions")
    return Expr("matmul", (self.shape[0], other.shape[1]), (self, other))


def input(name, shape):
  shape = tuple(shape)
  if not name.isidentifier() or any(not isinstance(d, Dim) and (type(d) is not int or d < 1 or d
>= 2**31) for d in shape):
    raise ValueError("invalid input name or shape")
  return Expr("input", shape, arg=name)


def constant(value):
  if type(value) not in (int, float) or not math.isfinite(value): raise ValueError("constant must
be finite")
  if abs(value) > 3.4028234663852886e38: raise ValueError("constant outside finite float32
range")
  return Expr("constant", (), arg=float(value))


def broadcast(a, b):
  output = []
  for i in range(1, max(len(a), len(b))+1):
    left, right = a[-i] if i <= len(a) else 1, b[-i] if i <= len(b) else 1
    if left == right: output.append(left)
    elif left == 1: output.append(right)
    elif right == 1: output.append(left)
    else: raise ValueError(f"incompatible broadcast dimensions: {left}, {right}")
  return tuple(reversed(output))


def binary(op, left, right):
  left = left if isinstance(left, Expr) else constant(left)
  right = right if isinstance(right, Expr) else constant(right)
  return Expr(op, broadcast(left.shape, right.shape), (left, right))
```

```
def toposort(outputs):
  seen, order = set(), []
  pending = [(node, False) for node in reversed(outputs)]
  while pending:
    node, expanded = pending.pop()
    if node in seen: continue
    if expanded:
      seen.add(node); order.append(node)
    else:
      pending.append((node, True))
      pending.extend((source, False) for source in reversed(node.sources))
  return order
```

33.2 Materialization and fusion must agree

Walk the graph topologically and count uses. Inputs are external buffers. Constants are inline literals. Reductions and matrix products become materialization boundaries. A pointwise node is also materialized when it has several consumers or is requested as an output. Other pointwise nodes are recursively inlined into their consumer's scalar expression.

This is a heuristic, not a cost model. Materializing a multi-use node avoids recomputation but adds memory traffic and a launch. Inlining it might still win when the computation is cheap or cache reuse is good. Keep `fuse=False` as a correctness and measurement baseline.

For `sum(relu(matmul(X,W)+bias), axis=1)`, matmul is one stage. The next stage iterates output rows, reduces over columns, and reads `matmul_result[row,column] + bias[column]`, applying ReLU before accumulating. It never allocates separate bias-add or ReLU matrices.

For `(X+row)*column`, index maps express broadcasting by selecting zero for a repeated singleton axis. Transpose reverses matrix coordinates. These are logical mappings into a producer, so an inlined transpose does not require an intermediate transpose buffer.

33.3 Render dynamic loops and wide addresses

Each kernel receives the input buffers it actually reads, one output buffer, and the dimension bindings. Loop bounds refer to runtime dimension arguments. Flattened offsets use int64 arithmetic; using int32 for a product of dimensions could overflow even when each dimension fits int32. Allocation checks bound the size of materialized tensors before a native call.

The C emitter uses ordinary sequential float32 accumulation, disables contraction, and avoids fast-math. ReLU follows `fmaxf(x,0)` semantics: a NaN input selects the numeric zero operand. This choice is tested explicitly against `np.fmax`, not silently compared with a different NaN policy. BLAS matrix products can still differ in rounding because they use different accumulation orders.

`tensorjit/compiler.py`

```python
"""Fuse a tensor DAG and render runtime-sized C loops.

No fast-math, SIMD schedule search, GPU lowering, or advanced GEMM optimization is
implied. C uses int64 address arithmetic; checked capacities bound every allocation.
"""
from collections import Counter
from dataclasses import dataclass
from tensorjit.graph import Dim, Expr, toposort


@dataclass(frozen=True)
class Kernel:
  node: Expr
  dependencies: tuple[Expr, ...]
  source: str


@dataclass
class Plan:
  inputs: tuple[Expr, ...]
  outputs: tuple[Expr, ...]
  kernels: tuple[Kernel, ...]
  symbols: tuple[Dim, ...]
  names: dict[Expr, str]
  source: str


def compile_plan(outputs, fuse=True):
  outputs = tuple(outputs)
  if not outputs or any(not isinstance(x, Expr) for x in outputs): raise ValueError("nonempty
tensor outputs required")
  if len(set(outputs)) != len(outputs): raise ValueError("duplicate outputs are unsupported")
  order = toposort(outputs)
  inputs = tuple(node for node in order if node.op == "input")
  if len({node.arg for node in inputs}) != len(inputs): raise ValueError("input names must be
unique")
  symbols = {}
  for node in order:
    for dim in node.shape:
      if isinstance(dim, Dim):
        if dim.name in symbols and symbols[dim.name] != dim: raise ValueError("same symbol name
has inconsistent bounds")
        symbols[dim.name] = dim
  symbols = tuple(symbols[name] for name in sorted(symbols))
  dimension_names = {dim: f"dim_{i}" for i, dim in enumerate(symbols)}
  counts = Counter(source for node in order for source in node.sources)
  stages = {node for node in order if node.op not in ("input", "constant") and
            (not fuse or node.op in ("sum", "matmul") or counts[node] > 1 or node in outputs)}
  stages.update(node for node in outputs if node.op in ("input", "constant"))
  names = {node: f"buf_{i}" for i, node in enumerate(order)}
  def dim_code(dim): return dimension_names[dim] if isinstance(dim, Dim) else str(dim)
  def offset(shape, indices):
```

```
    if not shape: return "0"
    expression = f"(int64_t)({indices[0]})"
    for dim, index in zip(shape[1:], indices[1:]): expression = f"(({expression})*
({dim_code(dim)})+({index}))"
    return expression
  def map_indices(node, indices):
    return ["0" if d == 1 else i for d, i in zip(node.shape, indices[len(indices)-
len(node.shape):])]

  kernels = []
  for stage in (node for node in order if node in stages):
    dependencies, lines = set(), []
    def expression(node, indices):
      if node.op == "constant":
        # repr always includes a decimal/exponent for Python float.
        return f"{node.arg!r}f"
      if node.op == "input" or (node in stages and node is not stage):
        dependencies.add(node)
        return f"{names[node]}[{offset(node.shape, indices)}]"
      if node.op in ("add", "mul"):
        args = [expression(source, map_indices(source, indices)) for source in node.sources]
        return f"({args[0]} {'+' if node.op == 'add' else '*'} {args[1]})"
      if node.op == "relu": return f"fmaxf({expression(node.sources[0], indices)}, 0.0f)"
      if node.op == "transpose": return expression(node.sources[0], indices[::-1])
      if node.op == "expand_dims": return expression(node.sources[0],
indices[:node.arg]+indices[node.arg+1:])
      if node.op == "broadcast": return expression(node.sources[0], map_indices(node.sources[0],
indices))
      if node.op == "positive": return f"({expression(node.sources[0], indices)} > 0.0f ? 1.0f :
0.0f)"
      raise ValueError(f"cannot inline {node.op}; expected a materialization boundary")
    indices = [f"i_{i}" for i in range(len(stage.shape))]
    depth = 1
    for index, dim in zip(indices, stage.shape):
      lines.append("  "*depth + f"for (int32_t {index} = 0; {index} < {dim_code(dim)}; {index}++)
{{")
      depth += 1
    indent = "  "*depth
    if stage.op in ("sum", "matmul"):
      if stage.op == "sum":
        source = stage.sources[0]
        source_indices = list(indices)
        source_indices.insert(stage.arg, "reduction")
        extent = source.shape[stage.arg]
        product = expression(source, source_indices)
      else:
        left, right = stage.sources
        extent = left.shape[1]
        product = f"({expression(left, [indices[0], 'reduction'])} * {expression(right,
['reduction', indices[1]])})"
      lines.extend([indent+"float accumulator = 0.0f;", indent+f"for (int32_t reduction = 0;
reduction < {dim_code(extent)}; reduction++) {{",
                    indent+f"  accumulator += {product};", indent+"}"])
```

```
      value = "accumulator"
    else:
      value = expression(stage, indices)
    destination = f"out_{names[stage]}"
    lines.append(indent + f"{destination}[{offset(stage.shape, indices)}] = {value};")
    for i in reversed(range(len(stage.shape))): lines.append("  "*(i+1) + "}")
    ordered = tuple(node for node in order if node in dependencies)
    arguments = [f"const float *{names[node]}" for node in ordered] + [f"float *{destination}"]
    arguments.extend(f"int32_t {dimension_names[dim]}" for dim in symbols)
    kernel_name = f"kernel_{len(kernels)}"
    source = f"void {kernel_name}({', '.join(arguments)}) {{\n" + "\n".join(lines) + "\n}"
    kernels.append(Kernel(stage, ordered, source))
  source = "#include <stdint.h>\n#include <math.h>\n" + "\n\n".join(k.source for k in kernels)
  return Plan(inputs, outputs, tuple(kernels), symbols, names, source)
```

Trace: print `compiled.source` for the example and mark which graph nodes correspond to a buffer, a loop, an address expression, and an inline scalar expression. Count multiplication and addition work separately from intermediate bytes written. Then disable fusion and compare source and kernel count before timing either implementation.

33.4 Guard inputs before allocating outputs

The runtime checks the complete input-name set, dtype, rank, contiguity, alignment, static extents, symbol bounds, and equal-symbol bindings. Guard failure occurs before native work. A mismatched reduction dimension cannot be repaired by accidentally reusing a prior binding.

Input bindings, or checked explicit metadata for shape-only graphs, determine every output and temporary size. Python integer multiplication computes sizes without silent machine overflow, and the configured byte budget rejects excessive payload and workspace requests. The byte accounting covers logical input/output payload plus the planned arena extent; Python object overhead, compiled code, and up to 63 extra arena-alignment bytes are outside that accounting. It is a local allocation guard, not a process memory limit.

The runtime compiles one library at construction. Calls with compatible shapes allocate new output and temporary storage but reuse that executable. Compilation count means library builds for this object; there is no persistent cross-process cache. Calls serialize around the object's diagnostics; this is not a multi-stream or concurrent-serving runtime.

33.5 Derive lifetimes from the fused schedule

After fusion, the graph's original edges no longer fully describe temporary consumption. A reduction may directly read a distant materialized producer through several inlined expressions. Each rendered kernel therefore records its actual buffer dependencies. Lifetime starts at its producer kernel and ends at the last kernel whose rendered expression reads that buffer.

For three matrix products followed by a reduction, a is consumed by b and b by c. The first matrix's slot can be reused for c after b finishes, but a and b cannot share storage: b's kernel reads a while writing b. The planner uses inclusive intervals and the strict reuse condition from Chapter 24.

Requested outputs are retained separately. They cannot enter the reuse pool, and a later call allocates fresh outputs so it cannot overwrite values returned by an earlier call. An input requested as an output is copied into owned output storage. The arena base is aligned to 64 bytes, and each planned offset respects the same alignment contract.

`tensorjit/runtime.py`

```python
"""Guarded dynamic-shape runtime, aligned memory planning, and native execution."""
import ctypes
import math
import shutil
import subprocess
import threading
from pathlib import Path
from tempfile import TemporaryDirectory
import numpy as np
from labs.memory import Temporary, plan_memory, verify_plan
from tensorjit.compiler import compile_plan
from tensorjit.graph import Dim


class CompiledGraph:
  def __init__(self, outputs, *, fuse=True, max_bytes=256*1024*1024):
    if type(max_bytes) is not int or max_bytes < 1: raise ValueError("max_bytes must be a
positive integer")
    self.plan = compile_plan(outputs, fuse=fuse)
    self.max_bytes = max_bytes
    self.directory = TemporaryDirectory(prefix="tensorjit-")
    self.source = self.plan.source
    path = Path(self.directory.name)
    (path/"kernels.c").write_text(self.source)
    compiler = shutil.which("gcc")
    if compiler is None: raise RuntimeError("gcc is required")
    subprocess.run([compiler, "-std=c11", "-shared", "-fPIC", "-O2", "-ffp-contract=off",
str(path/"kernels.c"), "-lm", "-o", str(path/"kernels.so")],
                   check=True, capture_output=True, text=True)
    self.library = ctypes.CDLL(str(path/"kernels.so"))
    self.functions = []
    pointer = ctypes.POINTER(ctypes.c_float)
    for i, kernel in enumerate(self.plan.kernels):
      function = getattr(self.library, f"kernel_{i}")
      function.argtypes = [pointer] * (len(kernel.dependencies)+1) +
[ctypes.c_int32]*len(self.plan.symbols)
      function.restype = None
      self.functions.append(function)
    self.compilations, self.calls = 1, 0
    self.last_memory = {}
    # Native calls release the GIL. Serialize calls so observable counters describe
    # one completed invocation; the actual arenas are already per-invocation.
    self.lock = threading.Lock()

  def bind(self, arguments, explicit=None):
    expected = {node.arg for node in self.plan.inputs}
    if set(arguments) != expected: raise ValueError(f"expected inputs {sorted(expected)},
received {sorted(arguments)}")
    bindings = dict(explicit or {})
    symbol_table = {dim.name: dim for dim in self.plan.symbols}
    if set(bindings) - set(symbol_table): raise ValueError("unknown explicit dimension")
    for name, value in bindings.items():
```

```
      dim = symbol_table[name]
      if type(value) is not int or not dim.lower <= value <= dim.upper:
        raise ValueError(f"{name}: invalid explicit dimension binding")
    total_input_bytes = 0
    for node in self.plan.inputs:
      array = arguments[node.arg]
      if not isinstance(array, np.ndarray) or array.dtype != np.float32 or array.ndim !=
len(node.shape) or not array.flags.c_contiguous or not array.flags.aligned:
        raise ValueError(f"{node.arg}: expected aligned contiguous float32 with rank
{len(node.shape)}")
      total_input_bytes += array.nbytes
      if total_input_bytes > self.max_bytes: raise ValueError("input bytes exceed configured
budget")
      for expected_dim, actual in zip(node.shape, array.shape):
        if isinstance(expected_dim, Dim):
          if not expected_dim.lower <= actual <= expected_dim.upper: raise ValueError(f"
{expected_dim.name}: dimension outside bounds")
          if expected_dim.name in bindings and bindings[expected_dim.name] != actual: raise
ValueError(f"{expected_dim.name}: inconsistent dimension binding")
          bindings[expected_dim.name] = actual
        elif expected_dim != actual: raise ValueError("static dimension mismatch")
    # Output symbols must all be grounded in input metadata, never guessed.
    if set(bindings) != {dim.name for dim in self.plan.symbols}: raise ValueError("unbound output
dimension")
    return bindings

  def __call__(self, **arguments):
    with self.lock: return self._run(arguments)

  def run_with_bindings(self, bindings, **arguments):
    """Supply metadata for symbols absent from all data inputs (e.g. constant gradients)."""
    with self.lock: return self._run(arguments, bindings)

  def _run(self, arguments, explicit=None):
    bindings = self.bind(arguments, explicit)
    def shape(node): return tuple(bindings[d.name] if isinstance(d, Dim) else d for d in
node.shape)
    def nbytes(node):
      size = 4*math.prod(shape(node))
      if size > min(self.max_bytes, 2**63-1): raise ValueError("tensor allocation exceeds
configured budget")
      return size
    # Derive liveness from the actual fused schedule, not original graph edges.
    index = {kernel.node: i for i, kernel in enumerate(self.plan.kernels)}
    retained = set(self.plan.outputs)
    temporaries = []
    for kernel in self.plan.kernels:
      node = kernel.node
      if node in retained: continue
      last = max([index[node]] + [i for i, consumer in enumerate(self.plan.kernels) if node in
consumer.dependencies])
      temporaries.append(Temporary(self.plan.names[node], nbytes(node), index[node], last))
    allocations, peak = plan_memory(temporaries)
```

```
    verify_plan(temporaries, allocations)
    output_bytes = sum(nbytes(node) for node in self.plan.outputs)
    input_bytes = sum(array.nbytes for array in arguments.values())
    if input_bytes+peak+output_bytes > self.max_bytes: raise ValueError("inputs, workspace, and
outputs exceed configured budget")
    arena = np.empty(peak+63, dtype=np.uint8)
    shift = (-arena.ctypes.data) % 64
    arena = arena[shift:shift+peak]
    arrays = {node: arguments[node.arg] for node in self.plan.inputs}
    outputs = {node: np.empty(shape(node), dtype=np.float32) for node in self.plan.outputs}
    for node, array in outputs.items():
      if node.op != "input": arrays[node] = array
    for kernel in self.plan.kernels:
      if kernel.node not in retained:
        allocation = allocations[self.plan.names[kernel.node]]
        arrays[kernel.node] = np.ndarray(shape(kernel.node), dtype=np.float32, buffer=arena,
offset=allocation.offset)
    ptr = ctypes.POINTER(ctypes.c_float)
    dimensions = [bindings[d.name] for d in self.plan.symbols]
    for kernel, function in zip(self.plan.kernels, self.functions):
      # An input requested as output is copied, preserving caller ownership.
      destination = outputs[kernel.node] if kernel.node in retained else arrays[kernel.node]
      function(*(arrays[node].ctypes.data_as(ptr) for node in kernel.dependencies),
destination.ctypes.data_as(ptr), *dimensions)
    self.calls += 1
    self.last_memory = {"temporary_bytes": peak, "unreused_temporary_bytes": sum(t.nbytes for t
in temporaries),
                        "output_bytes": output_bytes, "input_bytes": input_bytes, "allocations":
allocations, "bindings": bindings,
                        "kernel_count": len(self.plan.kernels)}
    return tuple(outputs[node] for node in self.plan.outputs)
```

Experiment: compile a chain of three 8×8 matrix products and a reduction. Three independent matrix temporaries would require 768 payload bytes. The planner uses a 512-byte arena because the third matrix reuses the first matrix's slot. Retain the first matrix as an additional output and check that no later kernel or invocation changes it. The test suite evaluates both the numerical results and the actual runtime memory report.

33.6 A measured baseline, including its losses

Run the benchmark as a module so imports resolve from the repository root:

```
.venv/bin/python -m examples.bench_tensorjit --sizes 16 32 64 --repeat 10 --output
book/outputs/tensorjit.json
```

The checked-in run uses a three-product chain, bias, ReLU, and row sum. It records raw end-to-end call samples, separate compilation time, environment metadata, numerical error, kernel count, and planned temporary bytes. These are CPU measurements that include guards, planning, allocation, Python dispatch, and native execution. They are not isolated kernel timings.

Square size	Unfused median ms	Fused median ms	Kernels before/after	Planned temporary b
16	0.275	0.193	6 / 4	2,048
32	0.307	0.230	6 / 4	8,192
64	0.993	0.828	6 / 4	32,768

These particular samples show a fusion benefit while remaining much slower than the NumPy expression on this machine. The implementation has naive matrix multiplication and substantial per-call Python work. This is a useful baseline with identifiable bottlenecks, not competitive GEMM. Timing spread and machine conditions matter; rerun before making a performance claim.

In this chain, fusion reduces total temporary payload produced but does not reduce peak arena bytes: the matrix products already need two overlapping buffers. Reporting only intermediate allocation count would conceal that distinction. Compilation in the recorded run took roughly 166 ms unfused and 132 ms fused; those costs must be amortized separately from steady-state calls.

The raw report is `book/outputs/tensorjit.json`; the harness is included below so the measurements are reproducible rather than detached prose.

`examples/bench_tensorjit.py`

```python
"""Measure dynamic tensor compilation, fusion, and workspace reuse.

Run from root: python -m examples.bench_tensorjit --sizes 16 32 64 --repeat 10
Use --output path.json to retain raw samples and the environment. CPU only.
The NumPy reference is an end-to-end expression, not a matched low-level GEMM.
"""
import argparse
import json
import os
import platform
import subprocess
import time
from pathlib import Path
import numpy as np
from tensorjit import CompiledGraph, Dim, input


def samples(call, repeat):
  call()  # Warm up; NumPy may initialize its BLAS thread pool.
  result = []
  for _ in range(repeat):
    start = time.perf_counter()
    call()
    result.append((time.perf_counter()-start)*1000)
  return result


def benchmark(sizes, repeat):
  if repeat < 1 or not sizes or min(sizes) < 1 or max(sizes) > 512:
    raise ValueError("positive repeats and sizes in [1,512] required")
  n = Dim("N", 1, 512)
  x, a, b, c, bias = (input(name, shape) for name, shape in
                      (("x", (n, n)), ("a", (n, n)), ("b", (n, n)), ("c", (n, n)), ("bias",
(n,))))
  output = ((((x @ a) @ b) @ c) + bias).relu().sum(1)
  variants, compile_ms = {}, {}
  for fuse in (False, True):
    key = "fused" if fuse else "unfused"
    start = time.perf_counter()
    variants[key] = CompiledGraph((output,), fuse=fuse)
    compile_ms[key] = (time.perf_counter()-start)*1000
  report = {"platform": platform.platform(), "processor": platform.processor(), "numpy":
np.__version__,
            "compiler": subprocess.run(["gcc", "--version"], capture_output=True, text=True,
check=True).stdout.splitlines()[0],
            "thread_environment": {key: os.environ.get(key) for key in ("OMP_NUM_THREADS",
"OPENBLAS_NUM_THREADS", "MKL_NUM_THREADS")},
            "compile_ms": compile_ms, "cases": []}
  rng = np.random.default_rng(17)
  for size in sizes:
    arrays = {name: rng.normal(size=(size, size)).astype(np.float32)*np.float32(1/np.sqrt(size))
for name in ("x", "a", "b", "c")}
```

```
    arrays["bias"] = rng.normal(size=size).astype(np.float32)
    def reference(): return np.fmax(((arrays["x"] @ arrays["a"]) @ arrays["b"]) @ arrays["c"] +
arrays["bias"], 0).sum(1)
    expected = reference()
    case = {"shape": [size, size], "numpy_ms": samples(reference, repeat)}
    for key, compiled in variants.items():
      actual, = compiled(**arrays)
      np.testing.assert_allclose(actual, expected, atol=3e-5, rtol=3e-5)
      timing = samples(lambda: compiled(**arrays), repeat)
      case[key] = {"samples_ms": timing, "median_ms": float(np.median(timing)),
                   "kernels": compiled.last_memory["kernel_count"],
                   "temporary_bytes": compiled.last_memory["temporary_bytes"],
                   "unreused_temporary_bytes": compiled.last_memory["unreused_temporary_bytes"],
                   "max_absolute_error": float(np.max(np.abs(actual-expected)))}
    report["cases"].append(case)
  return report


def main():
  parser = argparse.ArgumentParser(description=__doc__)
  parser.add_argument("--sizes", type=int, nargs="+", default=[16, 32, 64])
  parser.add_argument("--repeat", type=int, default=10)
  parser.add_argument("--output", type=Path)
  args = parser.parse_args()
  report = benchmark(args.sizes, args.repeat)
  content = json.dumps(report, indent=2) + "\n"
  if args.output: args.output.write_text(content)
  print(content, end="")


if __name__ == "__main__": main()
```

33.7 Integration tests and the next bottleneck

Tests compare fused and unfused results with independent NumPy expressions across dynamic and irregular shapes, all supported reduction axes, broadcast rows and columns, shared nodes, scalar inputs, and retained outputs. They also exercise guard rejection and byte-budget failure before execution. The original compiler's tests remain a separate suite; passing this suite does not establish integration with `uopc` autodiff or its CUDA device runtime.

The reverse-mode graph and reusable training executable are implemented next in Chapter 34. Further tasks are explicit: cache shape-specific memory plans under a bounded policy; tile/vectorize matmul; add

more general symbolic relations; and lower the same graph to a real accelerator. Measure those changes against this baseline and keep the guard, lifetime, and numerical contracts intact.

34. Differentiating and Training Through the Dynamic Compiler

A forward tensor compiler is not yet a training compiler. Backward needs graph accumulation, shape-aware derivative rules, and a runtime that observes updated parameters without baking their values or batch sizes into the executable. This chapter adds those pieces to `tensorjit` and runs an actual generated forward/backward program.

```
.venv/bin/python -m examples.train_tensorjit
.venv/bin/python -m pytest tests/test_tensorjit.py -q
```

The model is deliberately small enough to validate against a known answer. Data are generated by `target = X*true_W + true_bias`. The compiler differentiates squared error with respect to W and bias, then returns loss and both gradients. NumPy performs the SGD update outside the compiled graph. This is generated-kernel training with a host optimizer, not a claim that the entire training loop has been captured or that the model exercises large-scale language-model behavior.

34.1 Differentiate before scheduling

Reverse mode operates on the pure tensor graph, where matrix multiplication, reduction, and broadcasting still carry their mathematical meaning. It does not differentiate generated stores, allocation choices, or C loop syntax. Once backward is a graph, the same materialization, fusion, shape-binding, and runtime machinery compiles it.

Seed a scalar loss with gradient 1 and visit graph nodes in reverse topological order. Each local rule contributes gradients to its sources. If a source receives contributions from several consumers, add them. For `loss=sum(x*x)`, the same x appears twice as a multiplication source; both contributions are required. Replacing the earlier contribution instead of accumulating it halves the derivative.

A topological traversal follows node identity and visits each shared node once. The adjoint graph can still refer to the original forward values. The combined output graph of loss and requested gradients lets the scheduler identify common forward/backward producers and materialize shared work. This is graph sharing, not checkpointing: recomputation policies are a later extension.

34.2 Reverse broadcasting with reductions

For `Z=X+b` with `X:(B,N)` and `b:(N)`, an upstream `dZ:(B,N)` contributes directly to X. The gradient for b must sum over B. Broadcasting repeated b's value in each row, so reverse mode accumulates the contributions of all those logical uses.

General unbroadcasting first removes extra leading axes by summing axis zero. Then, for every original singleton dimension, reduce the corresponding gradient axis and reinsert the singleton. The final gradient shape must equal the source shape exactly. Do not merely rely on a later broadcast to hide a wrong derivative shape.

For `X:(M,1)` broadcast to `(M,N)`, dX has shape `(M,1)`, not `(M,)`. Removing the reduced axis without reinserting it would silently reinterpret which axis repeats in subsequent arithmetic. The test with an explicit `(2,1)` broadcast to `(2,4)` checks both values and shape.

34.3 Reduction and movement derivatives

A sum removes one axis. Backward inserts that axis as a singleton, then broadcasts to the source shape. A transpose reverses its matrix axes, and its derivative applies the same transpose to the upstream gradient. Inserting a singleton reverses to a reduction over that axis; explicit broadcast reverses through unbroadcasting.

These movement nodes need no standalone buffers when they have one consumer and can be expressed as index maps. Their semantics remain visible during differentiation, then their index expressions are inlined during code generation. This is the same separation of mathematical and storage concerns used in the original compiler, now exercised with runtime dimension arguments.

34.4 Matrix derivatives preserve symbolic dimensions

For `Y=XW`, with `X:(B,K)` and `W:(K,N)`, the upstream gradient has shape `(B,N)`:

```
dX = dY @ transpose(W)       # (B,N) @ (N,K) -> (B,K)
dW = transpose(X) @ dY       # (K,B) @ (B,N) -> (K,N)
```

The reduction extent B in dW is the original batch symbol, not the batch size of an example call. Compile once and bind B from each invocation's inputs. If differentiation accidentally reads a concrete batch length while constructing the graph, changing batch length can produce stale loops or a new graph per shape. The training test alternates several lengths and checks one library build.

ReLU contributes `dY * (X>0)`, with zero derivative chosen at zero. The implementation uses a float32 mask node; it is not a differentiable general comparison primitive. Higher derivatives through that mask are explicitly unsupported. Numerical derivative tests use finite values away from ReLU's nondifferentiable points.

`tensorjit/autodiff.py`

```python
"""Reverse-mode differentiation of the pure dynamic tensor graph.

Differentiation precedes fusion and code generation. The shape expressions in
backward retain the original symbols; no concrete runtime extent is captured.
"""
from tensorjit.graph import Expr, constant, toposort


def unbroadcast(gradient, shape):
  while len(gradient.shape) > len(shape): gradient = gradient.sum(0)
  for axis, extent in enumerate(shape):
    if extent == 1 and gradient.shape[axis] != 1:
      gradient = gradient.sum(axis).expand_dims(axis)
  if gradient.shape != shape: raise ValueError("gradient shape does not match source")
  return gradient


def gradients(loss, wrt):
  if not isinstance(loss, Expr) or loss.shape != (): raise ValueError("loss must be a scalar
expression")
  wrt = tuple(wrt)
  if any(not isinstance(node, Expr) or node.op != "input" for node in wrt): raise
ValueError("differentiate with respect to input nodes")
  order = toposort((loss,))
  accumulated = {loss: constant(1)}
  for node in reversed(order):
    if node not in accumulated: continue
    gradient, sources = accumulated[node], node.sources
    if node.op in ("input", "constant"): continue
    if node.op == "add": contributions = (unbroadcast(gradient, sources[0].shape),
unbroadcast(gradient, sources[1].shape))
    elif node.op == "mul":
      contributions = (unbroadcast(gradient*sources[1], sources[0].shape),
unbroadcast(gradient*sources[0], sources[1].shape))
    elif node.op == "relu": contributions = (gradient*Expr("positive", sources[0].shape,
sources),)
    elif node.op == "transpose": contributions = (gradient.transpose(),)
    elif node.op == "sum": contributions =
(gradient.expand_dims(node.arg).broadcast_to(sources[0].shape),)
    elif node.op == "expand_dims": contributions = (gradient.sum(node.arg),)
    elif node.op == "broadcast": contributions = (unbroadcast(gradient, sources[0].shape),)
    elif node.op == "matmul": contributions = (gradient @ sources[1].transpose(),
sources[0].transpose() @ gradient)
    elif node.op == "positive":
      raise ValueError("higher derivatives through the ReLU mask are unsupported")
    else: raise ValueError(f"missing derivative rule: {node.op}")
    for source, contribution in zip(sources, contributions):
      if contribution.shape != source.shape: raise ValueError("derivative rule produced wrong
shape")
      accumulated[source] = accumulated[source]+contribution if source in accumulated else
contribution
```

```
    return tuple(accumulated[node] if node in accumulated else constant(0).broadcast_to(node.shape)
  for node in wrt)
```

34.5 Shape information can outlive data dependencies

The derivative of `sum(X)` is all ones. Its generated graph may no longer read X at all, but the shape of the ones still contains X's symbolic dimensions. Removing a data dependency cannot justify inventing a dimension binding.

Ordinary calls infer symbols from actual input array metadata. For shape-only graphs, use `compiled.run_with_bindings({"N": length}, **inputs)`. Explicit bindings are checked for known names, integer type, and declared bounds. If an input also establishes N, the explicit binding must match it. This supports constant gradients without forcing a dummy data read or bypassing the shape contract.

An absent differentiated input gets a zero expression broadcast to its declared shape. If that shape contains an otherwise unbound symbol, supply the same checked metadata. No unrequested buffer value is read just to obtain an arbitrary default dimension.

Experiment: differentiate a sum of `2*(X+row)` with respect to X, row, and a disconnected input. The gradients are all twos, repeated-row counts times two, and zeros, respectively. Compile the gradients alone and vary N through explicit bindings. Then provide a conflicting input N and verify that dispatch fails before native work.

34.6 A reusable forward/backward executable

The training loss is:

```
E = XW + bias - target
L = sum(E*E) * scale
```

Scale is a scalar float32 input equal to `1/(batch_size*output_width)`, keeping the mean-loss convention consistent across batch sizes. Encoding that changing normalization as data avoids rebuilding the graph.

Alternatively, a richer symbolic compiler could expose dimension values as scalar graph expressions; this implementation keeps the boundary explicit.

Initialize W and bias to zero, generate deterministic batches with sizes 8, 16, and 32 in rotation, execute `(loss,dW,dbias)`, and update the actual input arrays with learning rate 0.15. Since the runtime reads parameters on every invocation, the next call observes the updated values. It does not cache buffer contents as constants.

`examples/train_tensorjit.py`

```python
"""Train a linear model through generated dynamic forward/backward kernels.

Run: python -m examples.train_tensorjit. Requires gcc; no PyTorch dependency.
Changing minibatch length reuses one compiled library. NumPy performs SGD updates.
"""
import numpy as np
from tensorjit import CompiledGraph, Dim, gradients, input


def train(steps=100, seed=7):
  if steps < 1: raise ValueError("steps must be positive")
  batch = Dim("B", 1, 64)
  x, y = input("x", (batch, 3)), input("y", (batch, 2))
  w, bias, scale = input("w", (3, 2)), input("bias", (2,)), input("scale", ())
  error = x @ w + bias - y
  loss = (error*error).sum(0).sum(0)*scale
  dw, db = gradients(loss, (w, bias))
  compiled = CompiledGraph((loss, dw, db))
  rng = np.random.default_rng(seed)
  true_w, true_bias = rng.normal(size=(3, 2)).astype(np.float32),
rng.normal(size=2).astype(np.float32)
  weights, biases = np.zeros((3, 2), dtype=np.float32), np.zeros(2, dtype=np.float32)
  history = []
  for step in range(steps):
    length = (8, 16, 32)[step % 3]
    data = rng.normal(size=(length, 3)).astype(np.float32)
    target = data @ true_w + true_bias
    actual_loss, grad_w, grad_b = compiled(x=data, y=target, w=weights, bias=biases,
scale=np.array(1/(length*2), dtype=np.float32))
    history.append(float(actual_loss))
    weights -= np.float32(0.15)*grad_w
    biases -= np.float32(0.15)*grad_b
  return {"initial_loss": history[0], "final_loss": history[-1], "parameter_max_error":
float(np.abs(weights-true_w).max()),
          "bias_max_error": float(np.abs(biases-true_bias).max()), "compiled_variants":
compiled.compilations,
          "calls": compiled.calls, "kernels": len(compiled.plan.kernels)}


def main():
  result = train()
  for key, value in result.items(): print(f"{key}: {value}")
  assert result["parameter_max_error"] < 1e-3 and result["bias_max_error"] < 1e-3


if __name__ == "__main__": main()
```

A recorded 100-step CPU run with seed 7 produced:

Quantity	Observed value
First minibatch loss	3.1259913444519043
Final minibatch loss	1.9817480989559044e-14
Maximum weight error versus generating weights	1.1920928955078125e-7
Maximum bias error	1.1920928955078125e-7
Native library builds	1
Calls	100
Generated kernels for combined outputs	9

These are different minibatches, so the loss sequence need not be monotonic. The stronger result is recovery of the known generating parameters, not simply a favorable first/last loss comparison. The dataset is noiseless and the model is linear; this is an integration test, not a generalization claim or evidence of neural-network training quality.

34.7 Validate backward independently of its own forward

Agreement between generated forward and generated backward can conceal a shared mistake. The tests compute the loss with an independent double-precision NumPy expression, perturb each input along a random direction, and compare its central finite difference with the inner product of the compiled gradient and that direction. Perturbations are evaluated in the double-precision oracle rather than being rounded into the compiled float32 inputs.

Separate tests cover repeated uses, leading-axis broadcasting, singleton-axis broadcasting, transpose, reduction, disconnected inputs, dynamic batch sizes, and explicit shape-only metadata. The training test checks parameter recovery and executable reuse after 100 parameter updates. A gradient test should fail on either wrong values or wrong shapes; an accidentally broadcast-compatible answer is not accepted.

Next implementation: add stable softmax and normalization with independently derived gradients; checkpoint a pure nonlinear block; implement Adam with explicit state; and lower this dynamic forward/backward path to a GPU backend. Keep numerical policy, alias ownership, and state updates

specified before optimizing. This chapter establishes the CPU training path that those extensions must preserve.

35. Register-Tiled GEMM on a Real GPU

The earlier GPU chapters separate index reasoning from hardware validation. This chapter adds an executable naive-versus-register-tiled GEMM experiment, numerical checks on a real device, and an honest timing protocol. It is not an emulator result and does not claim tensor-core or cuBLAS parity.

```
.venv/bin/python -m labs.gpu_gemm --shape 256 256 256 --repeat 10 --batch 20 --output
book/outputs/gpu-register-gemm.json
.venv/bin/python -m pytest tests/test_gpu_gemm_lab.py -q
```

A working NVIDIA driver and NVRTC must be discoverable by `uopc.cuda`. A CUDA toolkit configuration satisfying the runtime's library search is sufficient; environments with different NVRTC packaging may need their library paths configured. Hardware tests explicitly skip when this setup is absent. A skip establishes no device correctness or performance claim.

35.1 Derive output ownership first

The tiled kernel computes a 64×64 output block with a `(32,8)` thread block: 256 threads. Each thread owns eight rows and two columns, or sixteen scalar accumulators. For thread coordinates `(tx,ty)`, its output coordinates within the block are:

```
row = ty + i*8       for i in 0..7
col = tx + j*32      for j in 0..1
```

Every output element has one owner: dividing the row by 8 yields i and its remainder yields ty; dividing the column by 32 yields j and its remainder yields tx. Thus all 4096 elements are covered without duplicate writes. Prove this mapping before optimizing the loads.

Adjacent lanes differ in tx, so at a fixed i and j they write adjacent columns. An alternative mapping `col=tx*2+j` would make each instruction's adjacent-lane addresses stride by two. Both mappings can be correct, but their transaction behavior differs. Layout is a performance decision constrained by ownership and hardware access rules.

35.2 Cooperative loads and input reuse

Process the reduction in chunks of 16. One chunk stages `64*16` A elements and `16*64` B elements, 2048 float32 values or 8192 bytes of shared memory. Flatten the thread index as `lane=ty*32+tx`. Each thread loads linear positions `lane`, `lane+256`, `lane+512`, and `lane+768` for each tile. These positions cover each 1024-element tile exactly once.

The idealized input arithmetic intensity is:

```
2 * 64 * 64 * 16 / (4 * (64*16 + 16*64)) = 16 FLOPs/byte
```

This excludes output writes, edge waste, cache effects, and repeated loads between blocks. It is not a prediction of measured bandwidth. Compare its assumptions with the actual launch and shape. At a 1×1 problem, the kernel still allocates and executes much of a large tile, making the idealized model particularly misleading.

For each inner k, a thread reads eight A values and two B values into local fragments. It updates all sixteen pairwise products. Each loaded A fragment value is reused twice, and each B fragment value eight times within that thread. Accumulators persist in registers across chunks.

35.3 Shared-memory access and register cost

Within a warp, ty is constant and tx runs from 0 to 31. A fragment load at a fixed i and k reads the same shared address across the warp, using broadcast behavior. B fragment loads at a fixed j read adjacent columns. This avoids the obvious bank-conflict patterns discussed in Chapter 13. Actual instruction widths and generated accesses still need inspection before claiming zero conflicts.

Sixteen accumulators plus fragments and address values create register demand. The compiler can reuse fragment registers or spill values; source-level array size is not an exact physical-register count. Inspect compiler/profiler resource reports for the target. This example does not collect register, occupancy, or stall metrics automatically, so its throughput result cannot establish which resource limits performance.

A useful extension varies output tile size and per-thread tile shape while keeping the same operation and correctness checks. Record registers, spills, active blocks, shared bytes, and device time. A larger register tile

may improve reuse and still lose by reducing residency or causing spills.

35.4 Why both barriers are required

The first block barrier separates cooperative writes from all consumers of the staged tiles. Without it, one warp may read values another warp has not written. The second barrier ensures every consumer has finished before the next chunk overwrites the shared arrays. The first barrier of the next chunk is too late: overwrite can already race with a previous-chunk read.

Threads in partial edge blocks remain active through both barriers. Invalid global coordinates load zero into their shared positions. Only final output stores are masked. An early return before the barriers would violate the cooperative execution contract for a partial block.

`labs/kernels/gemm.cuh`

```
// Two float32 GEMMs with identical row-major input/output contracts.
// No tensor cores or reduced-precision input conversion. All partial tiles are masked.
extern "C" __global__ void gemm_naive(const float *a, const float *b, float *c,
                                      const int *dimensions) {
  const int m = dimensions[0], n = dimensions[1], k = dimensions[2];
  const int row = blockIdx.y * blockDim.y + threadIdx.y;
  const int col = blockIdx.x * blockDim.x + threadIdx.x;
  if (row >= m || col >= n) return;
  float accumulator = 0.0f;
  for (int inner = 0; inner < k; inner++)
    accumulator += a[(long long)row*k + inner] * b[(long long)inner*n + col];
  c[(long long)row*n + col] = accumulator;
}

// Output block: 64x64; reduction chunk: 16; thread block: (32,8).
// Each thread owns 8x2 accumulator values. Adjacent lanes own adjacent columns.
extern "C" __global__ void gemm_tiled(const float *a, const float *b, float *c,
                                      const int *dimensions) {
  const int m = dimensions[0], n = dimensions[1], k = dimensions[2];
  const int tx = threadIdx.x, ty = threadIdx.y;
  const int lane = ty*32 + tx;
  const int row_base = blockIdx.y*64, col_base = blockIdx.x*64;
  __shared__ float a_tile[64][16];
  __shared__ float b_tile[16][64];
  float accumulator[8][2] = {};
  for (int reduction_base = 0; reduction_base < k; reduction_base += 16) {
    // 256 threads cooperatively load each 1024-element tile. Invalid elements
    // become zero. No thread exits before either block-wide barrier.
    for (int linear = lane; linear < 64*16; linear += 256) {
      const int row = linear / 16, inner = linear % 16;
      a_tile[row][inner] = row_base+row < m && reduction_base+inner < k
        ? a[(long long)(row_base+row)*k + reduction_base+inner] : 0.0f;
    }
    for (int linear = lane; linear < 16*64; linear += 256) {
      const int inner = linear / 64, col = linear % 64;
      b_tile[inner][col] = reduction_base+inner < k && col_base+col < n
        ? b[(long long)(reduction_base+inner)*n + col_base+col] : 0.0f;
    }
    __syncthreads();  // Every staged element is visible before any consumer.
    #pragma unroll
    for (int inner = 0; inner < 16; inner++) {
      float a_fragment[8], b_fragment[2];
      #pragma unroll
      for (int i = 0; i < 8; i++) a_fragment[i] = a_tile[ty+i*8][inner];
      #pragma unroll
      for (int j = 0; j < 2; j++) b_fragment[j] = b_tile[inner][tx+j*32];
      #pragma unroll
      for (int i = 0; i < 8; i++) {
        #pragma unroll
        for (int j = 0; j < 2; j++) accumulator[i][j] += a_fragment[i]*b_fragment[j];
      }
    }
```

```
    __syncthreads();  // No producer overwrites a tile while another warp reads it.
  }
  #pragma unroll
  for (int i = 0; i < 8; i++) {
    #pragma unroll
    for (int j = 0; j < 2; j++) {
      const int row = row_base + ty+i*8, col = col_base + tx+j*32;
      if (row < m && col < n) c[(long long)row*n+col] = accumulator[i][j];
    }
  }
}
```

Experiment: use dimensions `(31,33,17)` and `(63,65,33)`, where both output blocks and the final reduction chunk are partial. Then deliberately remove one mask or barrier in a scratch version and run numerical checks plus an available memory/race/synchronization checker. Numerical agreement in a few runs cannot prove absence of a synchronization race.

35.5 Make the benchmark capable of detecting missing stores

Before each kernel's correctness launch, fill the device output with NaNs. If an edge element is never written, comparison fails. Reusing the correct naive output as the tiled kernel's initial output would conceal some missing-store bugs, so the harness explicitly resets it between kernels.

The reference multiplies the already-quantized float32 inputs in double precision on the CPU. This separates input conversion error from device accumulation error. Every output element is checked with declared absolute and relative tolerances. Tiny and irregular shapes are tested separately from the benchmark shape; one aligned square matrix is inadequate correctness coverage.

35.6 Device events and host-submission gaps

Compilation and transfers occur before timing. Warm-up launches run before event samples. A sample records a CUDA start event, enqueues a batch of kernels, records a stop event, waits for it, and divides the elapsed interval by batch count. Repeat to obtain a distribution instead of one favorable number.

The harness enqueues directly through the driver API so it does not accidentally synchronize after each launch. The default stream provides launch ordering. Events are destroyed and device buffers are freed after use. Context/module ownership remains with the existing teaching runtime; this is not a complete device-resource lifecycle service.

Device events do not magically remove host submission gaps. For tiny kernels, Python/ctypes may submit work slowly enough that the GPU becomes idle within the measured interval. Those gaps are included. CUDA graph replay or another controlled submission method is a subsequent experiment. Do not compare these intervals to another library's differently scoped measurements as if they were identical hardware kernel times.

`labs/gpu_gemm.py`

```python
"""Real-device GEMM lab: shared tiles, register tiles, tails, and event timing.

Run: python -m labs.gpu_gemm --shape 256 256 256 --repeat 10 --batch 20
Needs a working CUDA driver and NVRTC discoverable by uopc.cuda (toolkit setup).
This module never substitutes CPU emulation for a GPU result.
"""
import argparse
import ctypes
import hashlib
import json
from pathlib import Path
import time
import numpy as np
from uopc.cuda import CUDA

SOURCE_PATH = Path(__file__).parent / "kernels" / "gemm.cuh"


def enqueue(driver, function, grid, block, pointers):
  args = (ctypes.c_void_p * len(pointers))(*(ctypes.cast(ctypes.byref(p), ctypes.c_void_p) for p
in pointers))
  driver._check(driver.cu.cuLaunchKernel(function, *grid, *block, 0, None, args, None))


def event_samples(driver, function, grid, block, pointers, repeat, batch):
  start, stop = ctypes.c_void_p(), ctypes.c_void_p()
  driver._check(driver.cu.cuEventCreate(ctypes.byref(start), 0))
  try:
    driver._check(driver.cu.cuEventCreate(ctypes.byref(stop), 0))
    try:
      for _ in range(5): enqueue(driver, function, grid, block, pointers)
      driver._check(driver.cu.cuCtxSynchronize())
      samples = []
      for _ in range(repeat):
        driver._check(driver.cu.cuEventRecord(start, None))
        for _ in range(batch): enqueue(driver, function, grid, block, pointers)
        driver._check(driver.cu.cuEventRecord(stop, None))
        driver._check(driver.cu.cuEventSynchronize(stop))
        elapsed = ctypes.c_float()
        driver._check(driver.cu.cuEventElapsedTime(ctypes.byref(elapsed), start, stop))
        samples.append(elapsed.value / batch)
      return samples
    finally: driver._check(driver.cu.cuEventDestroy_v2(stop))
  finally: driver._check(driver.cu.cuEventDestroy_v2(start))


def benchmark(shape=(256, 256, 256), repeat=10, batch=20):
  if len(shape) != 3 or min(shape) < 1 or max(shape) > 2048 or repeat < 1 or batch < 1:
    raise ValueError("positive M,N,K <=2048, repeat, and batch required")
  m, n, k = map(int, shape)
  driver = CUDA()
  device_name = ctypes.create_string_buffer(256)
```

```
  driver._check(driver.cu.cuDeviceGetName(device_name, len(device_name), driver.dev))
  driver_version, rtc_major, rtc_minor = ctypes.c_int(), ctypes.c_int(), ctypes.c_int()
  driver._check(driver.cu.cuDriverGetVersion(ctypes.byref(driver_version)))
  if driver.rtc.nvrtcVersion(ctypes.byref(rtc_major), ctypes.byref(rtc_minor)):
    raise RuntimeError("could not query NVRTC version")
  rng = np.random.default_rng(11)
  a, b = rng.normal(size=(m, k)).astype(np.float32), rng.normal(size=(k, n)).astype(np.float32)
  # A double-precision CPU oracle checks the quantized float32 input values.
  expected = a.astype(np.float64) @ b.astype(np.float64)
  output = np.empty((m, n), dtype=np.float32)
  dimensions = np.array([m, n, k], dtype=np.int32)
  source = SOURCE_PATH.read_text()
  pointers = []
  results = {}
  try:
    for array in (a, b, output, dimensions): pointers.append(driver.to_device(array))
    for name, block, grid in (("gemm_naive", (16, 16, 1), ((n+15)//16, (m+15)//16, 1)),
                              ("gemm_tiled", (32, 8, 1), ((n+63)//64, (m+63)//64, 1))):
      output.fill(np.nan)
      driver._check(driver.cu.cuMemcpyHtoD_v2(pointers[2],
output.ctypes.data_as(ctypes.c_void_p), ctypes.c_size_t(output.nbytes)))
      started = time.perf_counter()
      function = driver.function(source, name)
      compile_ms = (time.perf_counter()-started)*1000
      enqueue(driver, function, grid, block, pointers)
      driver._check(driver.cu.cuCtxSynchronize())
      driver.to_host(pointers[2], output)
      np.testing.assert_allclose(output, expected, atol=3e-4, rtol=3e-4)
      timings = event_samples(driver, function, grid, block, pointers, repeat, batch)
      median = float(np.median(timings))
      results[name] = {"samples_ms": timings, "median_ms": median, "compile_ms": compile_ms,
                       "gflops": 2*m*n*k/(median*1e6), "max_absolute_error": float(np.abs(output-
expected).max()),
                       "block": block, "grid": grid}
  finally:
    for pointer in pointers: driver.free(pointer)
  return {"device": device_name.value.decode(), "architecture": driver.arch, "driver_version":
driver_version.value,
          "nvrtc_version": [rtc_major.value, rtc_minor.value], "shape": [m, n, k],
          "source_sha256": hashlib.sha256(source.encode()).hexdigest(), "repeat": repeat,
"batch": batch,
          "precision": "float32 inputs/accumulators; runtime NVRTC fast-math policy; no tensor
cores",
          "timing": "CUDA event interval divided by batch; includes any host submission gaps;
excludes compilation and transfers",
          "results": results}


def main():
  parser = argparse.ArgumentParser(description=__doc__)
  parser.add_argument("--shape", type=int, nargs=3, default=[256, 256, 256], metavar=("M", "N",
"K"))
  parser.add_argument("--repeat", type=int, default=10)
```

```
    parser.add_argument("--batch", type=int, default=20)
    parser.add_argument("--output", type=Path)
    args = parser.parse_args()
    content = json.dumps(benchmark(args.shape, args.repeat, args.batch), indent=2)+"\n"
    if args.output: args.output.write_text(content)
    print(content, end="")


if __name__ == "__main__": main()
```

35.7 Recorded result and its limits

A recorded run on an NVIDIA GeForce RTX 3050 6GB Laptop GPU (`sm_86`), driver version 13020 and NVRTC 12.9, used float32 inputs and accumulators and the teaching runtime's NVRTC fast-math policy. It did not use tensor cores. The report stores the kernel source SHA-256, device, launch dimensions, raw samples, compilation time, and numerical error.

256×256×256 kernel	Median event interval per launch	Useful GFLOP/s	Max absolute error vers
Naive	0.11594 ms	289.4	4.55e-5
Shared and register tiled	0.04426 ms	758.1	4.55e-5

This is about 2.62× for this comparison. A preceding run measured different absolute times while showing a similar ratio, consistent with the need to record power, thermal, and timing variation. Useful FLOPs count the mathematical `2*M*N*K`, not arithmetic wasted on masked tile padding.

The report does not compare cuBLAS or framework matmul, does not demonstrate tensor-core performance, and does not contain profiler or sanitizer captures. Those are remaining evidence requirements, not results to infer from a speedup over naive code. The hardware tests passed four shapes, including partial tiles; they do not prove correctness for every shape, compiler version, or GPU architecture.

35.8 The next optimization must have a hypothesis

The staged loop alternates transfer and compute rather than overlapping them. A double-buffered asynchronous-copy pipeline could reduce exposed memory latency, but it adds shared storage and synchronization constraints. Establish that transfer latency is material with profiling before adding that complexity.

Other projects include vectorized loads under explicit alignment guards, different thread tiles, architecture-supported matrix instructions, and a dispatch policy for small or irregular problems. Keep the naive kernel as a fallback and correctness baseline. Compare the same float32 mathematical work, record the precision policy, and expand the workload table before making a competitive claim.