共同知识 → Triton（kernel compiler） ⇄ Luminal（graph compiler） → 互补实验 → 第一笔贡献

AI Compiler Drills：Triton + Luminal 双项目 contributor 训练

目标不是“两个项目都看过”，而是 3–4 个月内形成可持续的贡献节奏：在 Triton 弄懂一个 kernel 从 tensor IR 到 PTX 的 lowering 与 layout，在 Luminal 弄懂 graph → e-graph → 搜索 → 执行计划 → CUDA backend，并各完成一个可复现的贡献。每道题都把一个真实的编译器机制（Triton 的 coalesce pass、FileCheck、autotuner；Luminal 的 e-graph、genome 搜索、plan_slab）缩成一个能在笔记本上测试的 Python 练习。

共 **16** 题 | 约 **16 小时 30 分** | 全部练习在 **CPU** 上验收（纯 Python + numpy）

源码链接固定在 commit：Triton · Luminal · egglog | 每题一页的提示与讲解 | 下载 PDF · GitHub

· 进度只保存在这个浏览器里 ·

Part 0 · 基础 0

路线与用法

先看这一节：为什么 Triton 和 Luminal 适合一起学、怎么分配精力、每道题怎么做，以及做完之后你应该能拿出什么。

1. 目标：一个能被识别的技术定位
2. 两个项目是什么关系
3. 路线
4. 每道题怎么做

目标：一个能被识别的技术定位

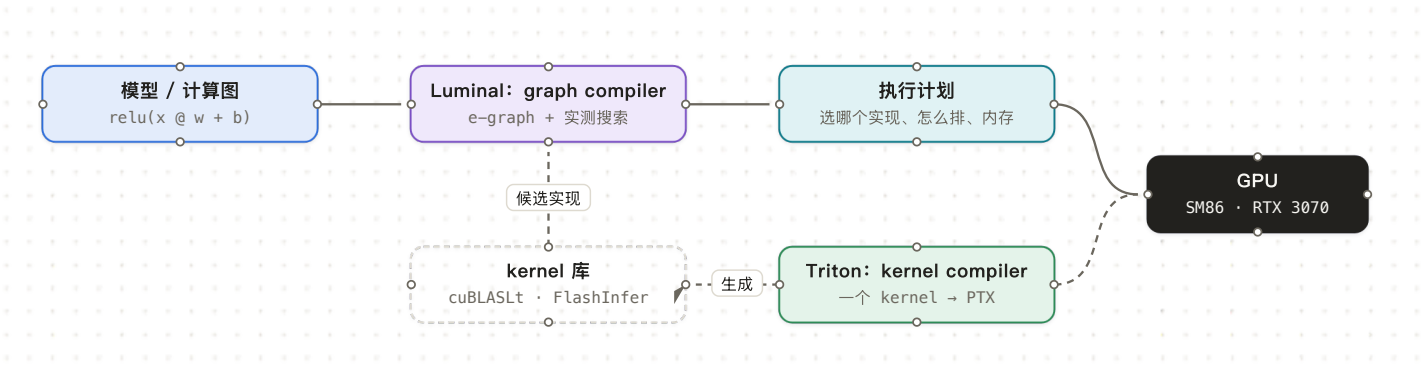
最终想形成的公开定位是：**GPU Compiler / Inference Performance Engineer — kernel lowering, execution planning, and performance correctness.** 具体来说，做完这套题和后面的贡献周期，你应该能说出两句有分量的话：

1. **Triton**：我能解释并优化一个 GPU kernel 从高层 tensor IR 到硬件指令的生成过程（TTIR → TTGIR → LLVM IR → PTX，layout、coalescing、`tl.dot` → MMA）。
2. **Luminal**：我能解释并改进一个 inference compiler 如何搜索、选择和组织 kernel 来执行整个模型（primitive graph → e-graph → 实测搜索 → 执行计划 → CUDA backend）。

两个能力叠加，比只会写 kernel 或只会读 graph compiler 更有区分度。**成功不以 PR 数量衡量**：理想结果是 Triton 有一笔 compiler 层面的实质贡献，Luminal 有一笔 backend 或性能 / 正确性贡献，并且你能用 benchmark 和 IR / profile 解释改动的价值。

两个项目是什么关系

一个模型从 Python 到 GPU：两个项目各管一段



Triton 回答“一个 kernel 怎样生成、怎样跑快”；Luminal 回答“整张图该选哪些实现、怎样组织执行”。Luminal 不是另一个 Triton：它的候选里有现成的 kernel 库（README：cuBLASLt、FlashInfer），你在 Luminal 的贡献不一定是写 kernel。

> **类比 · 菜谱作者和餐厅主厨**
>
> Triton 像一个**把菜谱翻译成精确操作步骤的厨师**：拿到“炒一盘青菜”，决定火候、下锅顺序、每个帮厨（线程）切哪一段（layout）。Luminal 像**餐厅主厨排一整桌菜**：这道菜自己做还是用现成的半成品（cuBLASLt），哪几道合并在一个锅里（fusion），先上哪道、灶台怎么轮用（执行计划和内存）。主厨会试做几种排法、掐表选最快的（实测搜索），而不是只凭经验。

共同问题	Triton 视角	Luminal 视角	本套题
GEMM 怎么实现	tile、layout、MMA lowering、pipeline	选哪个 GEMM 实现 / 库 / 计划	K1、T2、T3、L3
fusion	一个 kernel 内融合，寄存器代价	图级 rewrite、候选和计划搜索	K4、L2、L3
layout	thread / warp / shared memory layout	tensor layout、buffer 排布	T2、L4
怎么测性能	单 kernel microbenchmark、PTX、Nsight Compute	graph latency、search 开销、端到端	K2、T3、L3、L4
怎么保证正确	compiler transformation、kernel 数值	graph rewrite、backend、reference-vs-device	K3、T4、T5、L5

路线

```
Part 0  基础：环境 · GPU 执行模型 · Triton 编译器全景 · Luminal 全景 · 两边对照
   │
共同基础  K1–K4  GEMM 资源账 · benchmark 方法 · 数值正确性 · fusion 取舍
   │
双主线 ├─ T  Triton   T1 IR 四层 · T2 coalesce/layout · T3 autotune · T4 最小复现 · T5 FileCh
       └─ L  Luminal  L1 primitive graph · L2 e-graph · L3 搜索 · L4 计划与 arena · L5 对拍
   │
实战   X1 同一个 fused MLP 两边各量一遍 · P1 第一笔贡献
   │
14 周计划与进度 · 附录（PR 模板、资源）
```

精力分配：前 6 周 Triton 约 60%、Luminal 约 40%；之后按实际 issue / review 机会调整。不要同时在两个项目里做两个大 feature。

每道题怎么做

1. 读“背景”和“要用到的知识”，看图和类比。
2. 在 `mine/<题号>_*.py` 里实现（已给出数据结构、辅助函数和每个函数的文档字符串）。

3. 跑 `./check.sh <题号>`。测试里的期望值都是**按定义手算或对照真实源码得出的**，卡住时先读测试。
4. 实在卡住，依次展开三级提示；最后才看参考解。
5. 打开 hints/ 里这道题的单独讲解页：参考解逐段讲、更多图和例子、钉死版本的源码链接。

```
python3 -m venv .venv && .venv/bin/pip install pytest numpy markdown-it-py
./check.sh k1                # 测你的 mine/k1_gemm_model.py
./check.sh k1 --solutions    # 测参考解
./check.sh all --solutions   # 全部（约 100 个测试，不到 1 秒）
```

为什么全部练习都是纯 Python

Triton 没有 macOS wheel，Luminal 的 CUDA 路径要按仓库当前依赖验证。所以每道题都把一个**真实的编译器机制**缩成能在任何机器上测试的 Python：T2 复刻 Triton 的 coalesce pass（输出和 `coalesce.mlir` 的 CHECK 逐字一致），T5 自己写 FileCheck，L2 从零写 e-graph，L4 移植 Luminal 的 `plan_slab`。每题末尾都有“在 RTX 3070 上对照真实结果”的一节，把纯 Python 结论和真编译器的输出对上。

▼ 自测：为什么说“Luminal 并不是另一个 Triton”?

Triton 编译的是**单个 GPU kernel**（输入是一个 `@triton.jit` 函数，输出是 PTX/cubin）。Luminal 编译的是**整张计算图**：它把模型拆成 primitive ops，用 e-graph 表示等价改写，再通过实测搜索为每一部分挑选实现——这些实现可以是它生成的 kernel，也可以是 cuBLASLt、FlashInfer 等现成库。所以两边的“编译”发生在不同的层次，贡献面也不同。

Part 0 · 基础 1

统一实验环境

先把两个仓库都跑起来，并把环境记清楚：GPU、driver、CUDA Toolkit、Python / PyTorch / Triton、Rust。后面每一个 benchmark、每一个 issue 复现都要引用这份快照。

1. 起点：RTX 3070（Ampere，SM86）
2. 两个仓库的最小跑通
3. 本仓库自己的环境

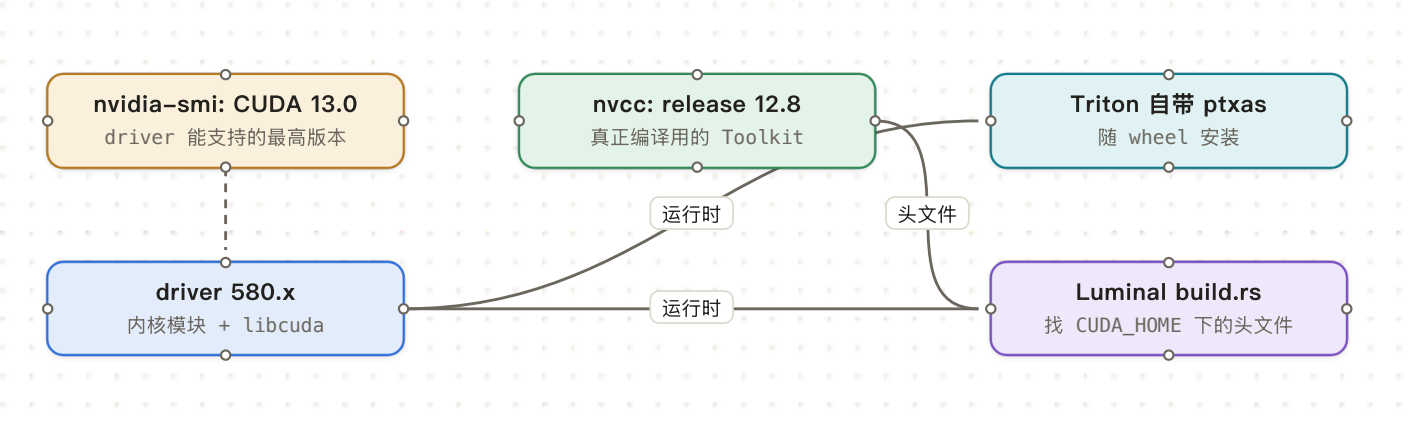
起点：RTX 3070（Ampere，SM86）

`triton › Triton README` 列出的 NVIDIA 支持从 Compute Capability 8.0 起，RTX 3070（SM86）在范围内。Luminal 的 CUDA 路径（`luminal_cuda_lite`）要按仓库当前依赖、CUDA Toolkit 和驱动实际验证——它的一个公开 issue 正是 **CUDA 13.x 下编译失败**（luminal #370，P1 会用它做第一笔贡献的候选）。

检查项	命令	用途
GPU	`nvidia-smi`	型号、compute capability、driver
CUDA Toolkit	`nvcc --version`、`echo $CUDA_HOME`	**不要**把 `nvidia-smi` 右上角的“CUDA Version”当成 Toolkit 版本
Python 栈	`python -c "import torch, triton; print(torch.__version__, triton.__version__, torch.cuda.is_available())"`	Triton build / test 与 reference
Rust	`rustc -V`、`cargo -V`	Luminal workspace 与 feature flags
Profiler	`nsys --version`、`ncu --version`	区分 host overhead、kernel latency、memory / occupancy

```
# 基础环境快照：贴进每一个 issue / PR / benchmark 报告
nvidia-smi --query-gpu=name,compute_cap,driver_version --format=csv
nvcc --version | tail -2
python --version
python -c "import torch, triton; print(torch.__version__, triton.__version__, torch.cuda.i
rustc -V && cargo -V
git -C triton rev-parse HEAD && git -C luminal rev-parse HEAD
```

例：driver 版本 ≠ Toolkit 版本



driver 向下兼容旧 Toolkit，所以 `nvidia-smi` 显示 13.0 时，机器上的 Toolkit 完全可能是 12.x。issue #370 这类问题的第一步，就是把两者分开记录。

类比 · 打印机驱动和排版软件

driver 像打印机驱动：它说“我最高能理解 13.0 版本的文件格式”。Toolkit 像你的排版软件：它决定你实际生成的是哪个版本的文件。驱动新、软件旧没问题；但你说“我的排版软件是 13.0”只因为驱动这么写，就会把 bug 报告写错。

两个仓库的最小跑通

```
# Triton（Linux + NVIDIA）：先用 wheel，需要改 compiler 时再源码 build
pip install triton           # 或：git clone https://github.com/triton-lang/triton && pip i
python python/tutorials/01-vector-add.py

# Luminal：先 ReferenceRuntime（CPU），再 CUDA
git clone https://github.com/luminal-ai/luminal && cd luminal
cargo test -p luminal_reference          # 纯 CPU
cargo run --release -p luminal_cuda_lite --example llama3 --features device   # README 里的
```

Luminal `luminal › README` 的 Usage 一节就是最小例子：`Graph::new()` → `cx.tensor(...)` → `a.matmul(b).output()` → `cx.compile(ReferenceRuntime::default(), CompileOptions::default())` → `set_data` → `execute` → `get_f32`。注意 **`a.matmul(b)` 这一行什么都没算**，只是记录了一个节点（L1）。

CUDA build 失败时

> 保留完整日志，**先不要改依赖**。记录 Toolkit、driver、`cargo` 版本和 `git rev-parse HEAD`，再对照 issue 列表看是否已知问题。roadmap 的原话：“若失败，保留完整日志，不要先改依赖”。这份日志很可能就是你第一笔贡献的起点（P1）。

建议用独立环境隔离 Triton 的 Python / CUDA 依赖和 Luminal 的 Rust / CUDA build。**不要为了同时跑两个项目而强行统一 CUDA Toolkit 版本**，先分别记录兼容矩阵。

本仓库自己的环境

```
python3 -m venv .venv && .venv/bin/pip install pytest numpy markdown-it-py
./check.sh all --solutions                  # 约 100 个测试，全部纯 Python
python3 build/sources.py fetch              # 第一次构建页面前：按固定 commit 下载被引用的源码（需
.venv/bin/python build/build.py             # 生成 index.html
.venv/bin/python build/build_hints.py       # 生成 hints/*.html
```

▼ 自测：同事说“我机器是 CUDA 13，所以 Luminal 编不过”，你先问什么？

先问 `nvcc --version` 的输出和 `CUDA_HOME` 指向哪里。`nvidia-smi` 的“CUDA Version”是 driver 能支持的最高运行时版本，不是编译用的 Toolkit。只有确认实际编译用的是 13.x Toolkit，并在当前 main 上复现，才能把问题归到 #370 这类“CUDA 13 API 变化”的范畴。

Part 0 · 基础 2

GPU 执行模型与内存层次（速查）

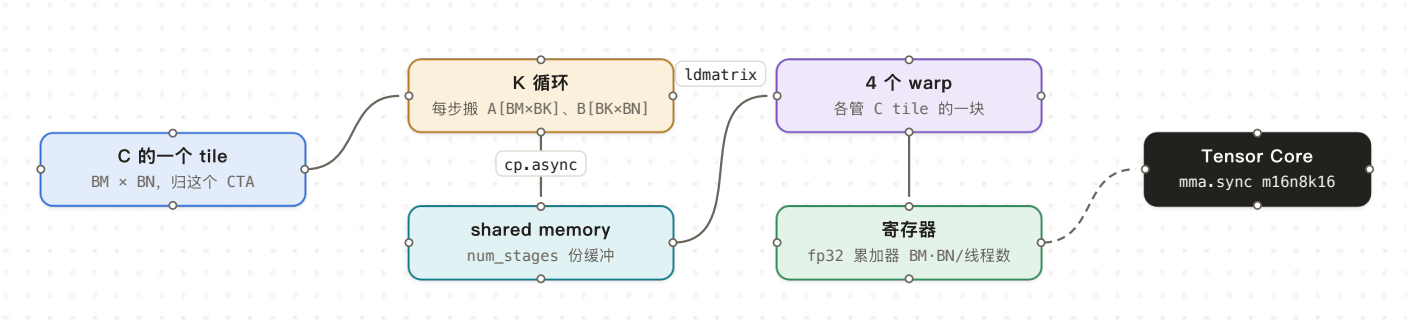
两个编译器最终都要回答同一组硬件问题：活怎么分给 CTA / warp / lane，数据从哪层内存来，一个 SM 上能同时放几个 CTA。这一页是后面所有题的公共词汇表。

1. 执行层次
2. 内存层次和三个常见瓶颈
3. 三种 bound 和 roofline

执行层次

层	是什么	Triton 里对应	典型数字（RTX 3070, SM86）
grid	一次 launch 的全部 CTA	`kernel[grid](...)`	任意
CTA（block）	跑在一个 SM 上的一组线程，共享 shared memory	一个 **program instance**（`tl.program_id`）	`num_warps × 32` 个线程
warp	32 个 lane 同步执行同一条指令	由 layout 的 `warpsPerCTA` 安排	32 lanes
lane / thread	拥有自己的寄存器	由 `threadsPerWarp`、`sizePerThread` 安排	最多 255 个 32-bit 寄存器
SM	执行 CTA 的硬件单元	—	46 个 SM；每 SM 64K 寄存器、最多 100 KB shared memory、1536 线程

一个 GEMM CTA 的数据分工（BM × BN 输出 tile）



K1 会把每一项都算成数字：这个 CTA 读多少字节、用多少 shared memory、每线程多少寄存器、一个 SM 能放几个这样的 CTA、整个 grid 要几轮（wave）。

内存层次和三个常见瓶颈

层	容量	访问方式	要注意
寄存器	每 SM 256 KB	线程私有	用超了就 **spill** 到 local memory（其实在 DRAM）
shared memory	每 SM 最多 100 KB	CTA 内共享，32 个 bank	**bank conflict**：同一 warp 的多个 lane 访问同一 bank 的不同地址
L2	4 MB	全 GPU 共享	GEMM 的 A/B 重复读取大多在这里命中
DRAM	8 GB，448 GB/s	以 32 B sector 为单位	**coalescing**：一个 warp 的访问落在越少的 sector 越好

> **类比 · 仓库、货架、工作台**
>
> DRAM 是城外的大仓库（大、远、按整箱 32 B 发货），L2 是城里的中转站，shared memory 是车间的公共货架（一个班组共用，取货口有 32 个窗口，同一个窗口排队就是 bank conflict），寄存器是每个工人手边的工作台（最快，但台面很小，放不下就只能堆回仓库——spill）。**coalescing** 就是让一个班组 32 个人要的货尽量装在同几箱里一起运。

三种 bound 和 roofline

一个 kernel 的时间下界是 `max(FLOPs / 峰值算力, 字节数 / 峰值带宽)`，再加上固定的 launch 开销（几微秒）。

- **compute-bound**：算术强度（FLOP/Byte）高于机器平衡点（RTX 3070 FP16 Tensor Core 约 40 TFLOP/s ÷ 448 GB/s ≈ 90 FLOP/B）。大 GEMM 属于这类。
- **memory-bound**：elementwise、reduction、decode attention。优化方向是少搬数据（fusion）、搬得整齐（coalescing、向量化）。
- **launch-bound**：问题太小，算和搬都只要几微秒，launch 本身占大头。优化方向是 fusion、CUDA graph（Luminal 的 `cuda_graph.rs`）。

<details>
<summary>▼ 自测：一个 [4096] fp32 向量加法是哪种 bound?</summary>

读 2 × 16 KB、写 16 KB，共 48 KB；448 GB/s 下约 0.1 µs，远小于约 5 µs 的 launch 开销。所以是 **launch-bound**，测出来的时间几乎全是 launch。要看到 memory-bound 行为，向量至少要几百万元素（K2 的 benchmark 题会让你测出这个拐点）。
</details>

▼ 自测：为什么 occupancy 不是越高越好?

occupancy 只说明一个 SM 上同时驻留多少 warp，用来隐藏访存延迟。GEMM 常常故意用大 tile（每线程更多寄存器、更多 shared memory）换取更高的数据复用，occupancy 降到 1 个 CTA/SM 反而更快。K1 和 T3 会让你看到：最快的配置往往 occupancy 不是最高的。

Part 0 · 基础 3

Triton 编译器全景

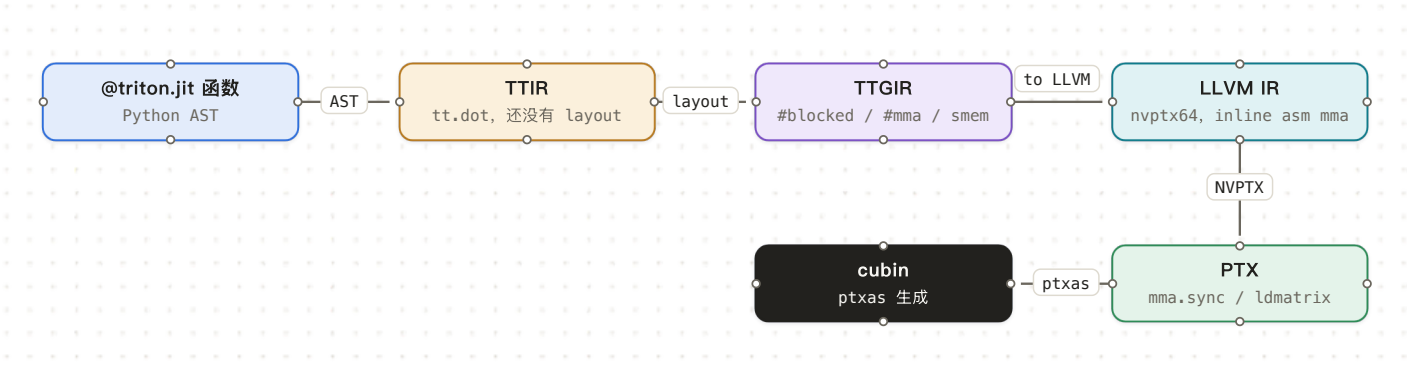
Triton 把一个 Python 函数编译成一个 GPU kernel。这一页画出它的全部阶段、每个阶段新增的概念，以及源码里对应的目录，T1—T5 都在这张地图上。

1. 一条流水线，五个阶段
2. TTGIR 上的 pass 链
3. 读源码的顺序

一条流水线，五个阶段

NVIDIA 后端的阶段定义在 `triton › third_party/nvidia/backend/compiler.py`：`make_ttir` → `make_ttgir` → `make_llir` → `make_ptx` → `make_cubin`。每一步的输出都能在 `kernel[grid](...)` 返回的 `CompiledKernel.asm` 里看到。

@triton.jit → cubin



最重要的一步是 TTIR → TTGIR：从这里开始，每个张量类型都带上 layout encoding（`tensor<64x64xf32, #blocked>`），编译器开始决定“哪个线程拿哪个元素”。T1 让你在四层 IR 里把这些概念逐个找出来。

阶段	关键概念	源码入口
Frontend	`@triton.jit`、AST → IR、`constexpr` 特化	`triton › runtime/jit.py` · `triton › compiler/code_generator.py` · `triton › compiler/compiler.py`
TTIR	program instance、block tensor、mask、`tt.dot`、canonicalization	`triton › TritonOps.td`
TTGIR	Blocked / Slice / MMA / DotOperand / Shared layout, `ttg.convert_layout`	`triton › TritonGPUAttrDefs.td` · `lib/Dialect/TritonGPU/Transforms/`
Lowering	TritonGPU → LLVM/NVVM → PTX	`lib/Conversion/`、`third_party/nvidia/lib/`
Testing	lit + FileCheck、Python 单测	`test/`（`triton › coalesce.mlir` 等） · `triton › test_core.py`

TTGIR 上的 pass 链

`make_ttgir` 里的 pass 顺序本身就是一份“编译器做了哪些优化”的目录（摘自 compiler.py，SM86 会走到的部分）：

```
convert_to_ttgpuir    给每个张量一个默认 layout（num_warps、threads-per-warp=32）
coalesce              为 load/store 选合并访存的 BlockedEncoding        ← T2 复刻这一个
remove_layout_conversions   删掉多余的 ttg.convert_layout（反复跑几次）
optimize_thread_locality
accelerate_matmul     tt.dot 换成 MMA layout（Ampere：nvidia_mma v2）     ← T1 在 IR 里找它
optimize_dot_operands
pipeline              按 num_stages 做软件流水（cp.async + async_wait）
prefetch / reduce_data_duplication / reorder_instructions ...
```

> **类比 · 装修工程的图纸层次**
>
> TTIR 是**户型图**：哪里有墙、哪里是厨房（load、dot、store），不管谁来施工。TTGIR 是**施工分工图**：每面墙分给哪个班组、每人负责哪几块砖（layout），哪些材料要先运到工地仓库（shared memory）。LLVM IR / PTX 是**每个工人的操作单**：第几步拿哪块砖、用哪台机器（`mma.sync`）。`convert_layout` 就是两个班组之间交接材料——同一班组内递一下（shuffle）很便宜，跨班组要先放到仓库再取（shared memory）就贵了。

读源码的顺序

1. **Kernel semantics**：官方 tutorials 和 Triton Puzzles：program instance、block tensor、mask、reduction、dot。
2. **Frontend**：`@triton.jit` → `JITFunction.run` → `compile()`，看 `constexpr` 如何变成特化。
3. **TTIR**：Triton dialect 的 op 定义、canonicalization。
4. **TTGIR**：重点。layout 定义、`triton › Coalesce.cpp`、`triton › RemoveLayoutConversions.cpp`、`triton › AccelerateMatmul.cpp`。
5. **Lowering**：选 `tl.dot` 和 `convert_layout` 两条路径追到 PTX。
6. **Testing**：lit/FileCheck（T5）、Python 测试、最小复现（T4）。

Gluon：自己写 layout

Gluon 是 Triton 里更低层的编程模型：layout 不再由编译器推断，而是你在代码里写出来。T2 算的 BlockedLayout，在 Gluon 里就是 `ttgl.BlockedLayout(size_per_thread=..., threads_per_warp=..., warps_per_cta=..., order=...)`。入门看 `triton › gluon/01-intro.py` 和 `triton › gluon/02-layouts.py`。

▼ 自测：同一个 vector-add 的 TTIR 和 TTGIR，op 集合会变吗?

不会（T1 的测试就断言了这一点）。变的是**类型**：`tensor<1024xf32>` 变成 `tensor<1024xf32, #blocked>`，文件开头多了 `#blocked = #ttg.blocked<{sizePerThread = [4], ...}>` 和 `"ttg.num-warps" = 4` 之类的 module 属性。对 matmul 则不同：TTGIR 会多出 `ttg.local_alloc`、`ttg.async_copy_global_to_local`、`ttg.local_load` 和 `ttg.convert_layout`，这些是 pipeline 和 MMA layout 带来的新 op。

Part 0 · 基础 4

Luminal 全景

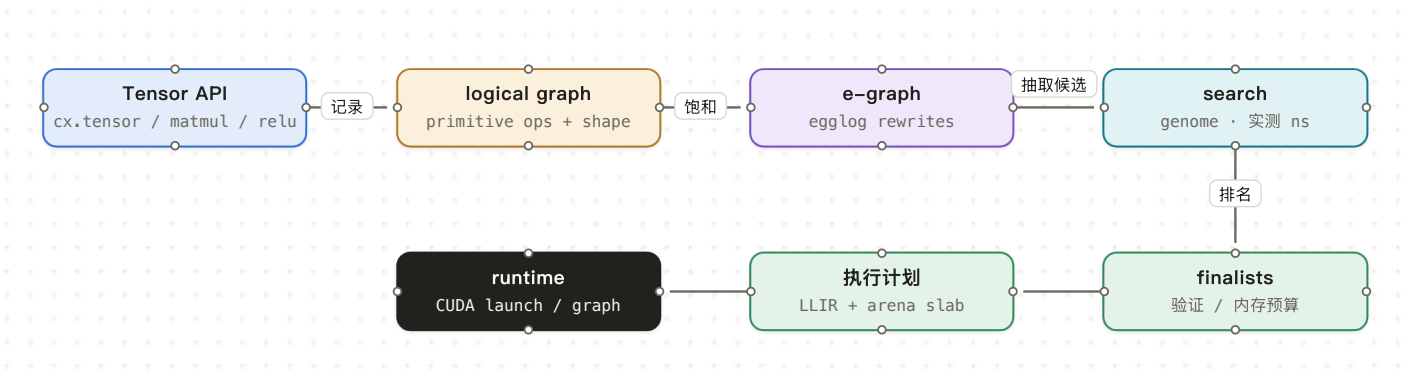
Luminal 是一个“记录整张图、在等价程序里实测搜索、再生成执行计划”的 inference compiler。这一页画出它从 tensor API 到 CUDA launch 的全部环节，以及每个环节在源码里的位置，L1–L5 都在这张地图上。

1. 核心理念（README）
2. 为什么用 e-graph + 搜索，而不是一串 pass
3. 看见它在做什么：环境变量

核心理念（README）

- **RISC 风格**：一切都化成一小组 primitive ops。README 列了 15 个：Unary `Log2, Exp2, Sin, Sqrt, Recip`；Binary `Add, Mul, Mod, LessThan`；Other `SumReduce, MaxReduce, Iota, Gather, Scatter, Cast`。当前 main 的 `luminal › LogicalOp` 枚举比这更长（还有 `Select`、`IndexMapApply`、各种 scan 等），读的时候以源码为准。
- **Compile everything**：`x + y` 不计算，只记录到 DAG；`execute()` 时才跑。
- **Search**：“The best heuristic is no heuristic”。不用一次性、破坏性的改写规则，而是把所有等价程序放进 e-graph，再**实测**挑最快的。
- **First-class dynamism**：形状可以是符号表达式（`(s, 4096)`）。
- **Validated against PyTorch** 与 ReferenceRuntime。

Luminal：从 tensor API 到 GPU



L1 画出 logical graph；L2 从零实现 e-graph 和 equality saturation；L3 实现 genome 搜索、rank_insert 与 finalists；L4 移植 arena 的 plan_slab；L5 写 Reference vs CUDA 的对拍测试。

环节	要搞懂的问题	源码入口
Tensor API / graph	节点、shape、dtype、broadcast 是 view	`luminal › src/graph.rs` · `luminal › frontend/matmul.rs` · `luminal › frontend/unary.rs`
primitive ops	primitive 集合、high-level op 如何拆	`luminal › logical/ir.rs` · `src/logical/op/*`
e-graph	e-class、rewrite、saturation、extraction；rewrite 的合法性和预算	`luminal › egglog_preamble.egg` · `luminal › extraction.rs` · egglog
search / cost	候选如何生成、测量、排名	`luminal › cuda_lite/search.rs` · `luminal › finalists.rs`
计划 / runtime	选中的图如何变成 buffer、slab、launch	`luminal › buffer/arena.rs` · `luminal › cuda_lite/runtime.rs` · `luminal › cuda_graph.rs`
kernel 库	cuBLASLt、FlashInfer 的接入与 dispatch 条件	`luminal › ops/cublaslt/README.md` · `ops/cublaslt/egg/*.egg`
正确性	ReferenceRuntime 的语义、窄浮点模拟	`luminal › reference/runtime.rs` · `luminal › narrow_float.rs`

为什么用 e-graph + 搜索，而不是一串 pass

README 的“What about XLA?”一节给出了动机：传统编译器是一大串**破坏性**（单向）的改写规则，规则只能在“确定有收益”时触发，于是越写越特殊、越写越多。e-graph 的改写是**非破坏性**的：`a+b → b+a` 只是在同一个 e-class 里多加一个 e-node，原来的程序还在。所有选择都推迟到最后，由测量决定。

> **类比 · 一本“同义句词典”而不是一支改稿红笔**
>
> 传统 pass 像编辑拿红笔改稿：改了就回不去，所以只敢做有把握的修改。e-graph 像一本同义句词典：每句话旁边列出所有等价说法（e-class），词典越写越厚但从不删东西。最后由“朗读计时”（实测搜索）决定每处用哪种说法。代价是词典可能爆炸（L2 会量出 3 项加法饱和后就有 15 个 e-node，6 项有 608 个），所以要有预算。

看见它在做什么：环境变量

README 的 Environment flags 一节列了所有开关，L4 的端到端 trace 主要用这几个：

变量	看什么
`LUMINAL_LOG=1`	打开所有日志通道（`SEARCH_LOG`、`EGGLOG_LOG`、`ROLLING_LOG`）
`LUMINAL_LOG_LLIR=1`	每找到一个更快的图，打印规范化的 LLIR（可以直接 `diff` 两次运行）
`LUMINAL_SEARCH_OP_NAMES=1`	在 best-so-far 行后面附上候选由哪些 kernel / host op 组成
`LLIR_DUMP_DIR=dir`	把选中的 finalist 的 LLIR 写成 `.txt` 和 `.dot`
`LUMINAL_CUDA_PROFILE_RECAPTURE=1`	每次 execute 的 prepare / allocate / launch / sync 分解
`LUMINAL_CUBLASLT_AUTOTUNE=1`	prepare 时对 cuBLASLt 算法做 autotune

▼ 自测：Luminal 的 matmul 为什么“不会自动 cast”?

`luminal › matmul 的文档注释`写得很明确：matmul 就是两个同 dtype 操作数上的 broadcast + reduce，**从不插入 cast**；想要不同的计算精度，必须由用户写 `lhs.cast(D).matmul(rhs.cast(D))`，让 cast 成为图里可见的节点。两个 dtype 不同直接报错。这样做的好处是：精度选择在图里是显式的，搜索和对拍（L5）都不会被隐式类型转换干扰。

Part 0 · 基础 5

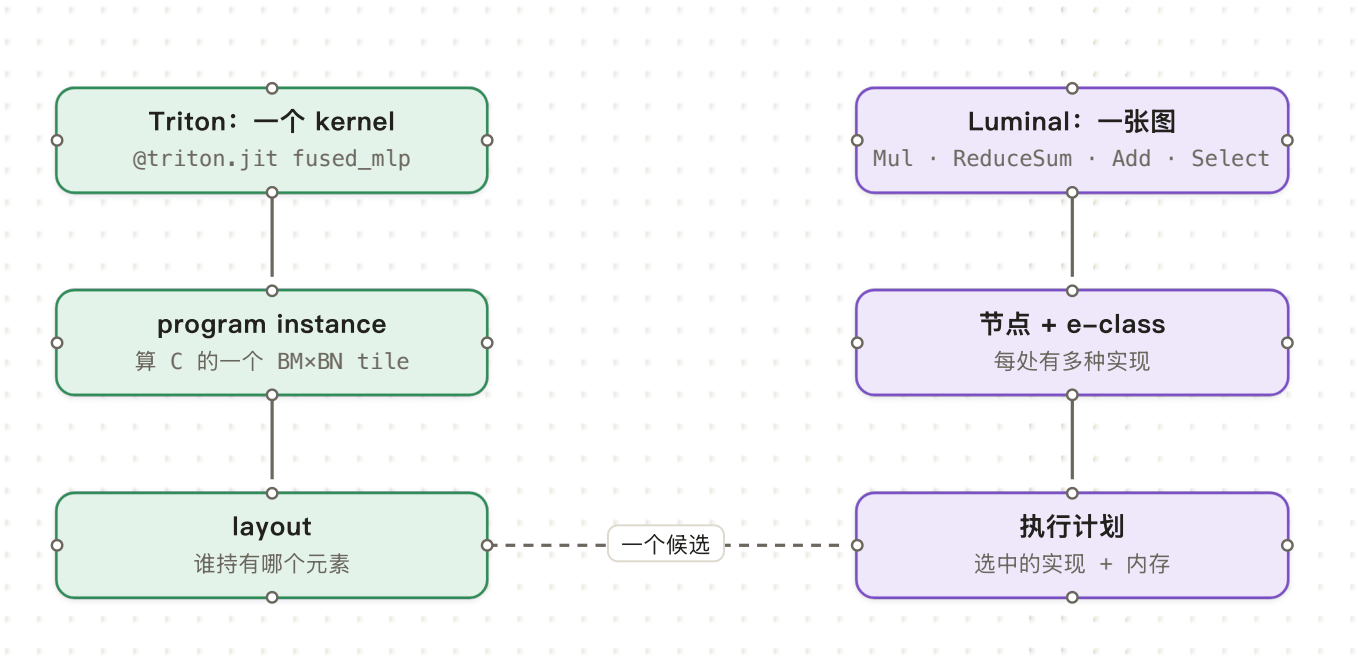
两边对照：互补而不是重复

两个项目不是重复学习，而是同一个问题的两个层次。这一页把它们并排放：同样一个 GEMM / fusion / layout / benchmark / 正确性问题，Triton 和 Luminal 分别怎么回答。

1. 抽象层次
2. 两种“搜索”
3. 最好的共同实验

抽象层次

同一个 relu(x @ w + b)，两种“程序”



Triton 的基本单位是 program instance（一个 CTA 算一块输出）；Luminal 的基本单位是图上的节点和 e-class。一个 Triton kernel 在 Luminal 看来只是某个 e-class 的一个候选实现。

问题	Triton 视角	Luminal 视角
如何实现 GEMM?	tile、layout、MMA lowering、pipeline（T1、T2、T3）	选哪个 GEMM implementation / library / plan（L3）
如何 fusion?	一个 kernel 内的 fusion 与寄存器代价（K4）	图级 rewrite、fusion candidate 和计划搜索（L2、L3）
如何优化 layout?	thread / warp / shared memory layout（T2）	tensor layout、buffer / operation arrangement（L4）
如何测性能?	单 kernel microbenchmark、PTX、Nsight Compute（K2、T3）	graph latency、search overhead、端到端 benchmark（L3、L4）
如何保证正确性?	compiler transformation / kernel 数值（T4、T5）	graph rewrite、backend、reference-vs-device（L2、L5）

两种“搜索”

两个项目都在搜索，但搜索的东西不一样：

- **Triton autotune**（T3）：kernel 写好了，只搜 `BLOCK_M/N/K`、`num_warps`、`num_stages` 这几个**数值参数**；空间小（几十到几百个配置），先用 `early_config_prune` 剪掉跑不了的、再用 `perf_model` + `top_k` 只测最有希望的几个。
- **Luminal search**（L3）：搜的是**程序结构**：每个 e-class 选哪个 e-node（genome）。空间是各 e-class 选项数的乘积，可能极大，所以用遗传算法 + 实测排名 + 预算。

> **类比 · 调一辆车 vs 规划一条路线**
>
> Triton autotune 像给一辆已经造好的车调胎压和档位比（参数）；Luminal search 像在地图上规划路线（结构）：走高速还是走小路、在哪里换乘（用 cuBLASLt 还是自己生成的 kernel、哪几步合并）。前者空间小、可以几乎穷举；后者空间组合爆炸，必须有策略和预算。两者都必须**实测**，不能只信估算。

最好的共同实验

选同一组 GEMM / fused MLP workload：先用 Triton 写出并 profile 一个 kernel，理解它的硬件成本（K1、T3）；再在 Luminal 观察 graph / search 如何为它选择执行实现（L3、L4）。X1 就是这个实验的完整步骤。

▼ 自测：为什么 Luminal 的搜索比 Triton autotune 更依赖“预算”?

Triton autotune 的配置空间是几个参数的笛卡尔积，通常几十个，剪枝后全部实测也只要几秒。Luminal 的空间是每个 e-class 选项数的**乘积**，而 e-graph 饱和本身就可能指数增长（L2：n 项加法的交换律 + 结合律饱和后有 $3^n - 2^{n+1} + 1 + n$ 个 e-node）。不设迭代 / 节点 / 搜索预算，编译时间会失控。所以 L3 要把“搜索时间”和“最终 latency”放在一起报告，并算 break-even：多花的搜索时间要多少次执行才能赚回来。

共同基础

Track K · 共同知识主线

GEMM 资源账、benchmark 方法、数值正确性、fusion 取舍：两个项目都要用的基本功。

共同基础 · Track K · 共同知识主线 · 第 1 题

K1 GEMM tile 账本：FLOPs、搬运量、smem / 寄存器、occupancy、wave

在跑任何 benchmark 之前，先用纸笔（这里是 Python）把一个 GEMM 配置的账算清楚：多少 FLOPs、多少字节、多少 shared memory 和寄存器、一个 SM 放几个 CTA、要几轮 wave、受哪个上限约束。T3 的 autotuner 和 L3 的搜索都建立在这本账上。

45 min | 难度 易 | 你写 `mine/k1_gemm_model.py` | 验收 `./check.sh k1`

参考解 `solutions/k1_gemm_model.py` | 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 题目要求
5. 题干
6. 验收
7. 提示
8. 参考解
9. 在 RTX 3070 上对照

背景

roadmap 的共同知识表里，GEMM tiling 的练习输出是“手算 tile 的 FLOPs、数据搬运量和资源占用”。这件事有两个用处：

1. **预测**：扫参数之前就能排除跑不了的配置（shared memory 超限、寄存器 spill），并大致知道哪些配置值得测。这正是 Triton autotuner 的 `early_config_prune` 和 `perf_model` 在做的事（T3）。
2. **解释**：Nsight Compute 报告里的 registers、shared memory、achieved occupancy、DRAM bytes，每一项都能和你算的数对上；对不上的地方，往往就是编译器做了你没想到的事（或者有 bug）。

源码与资料

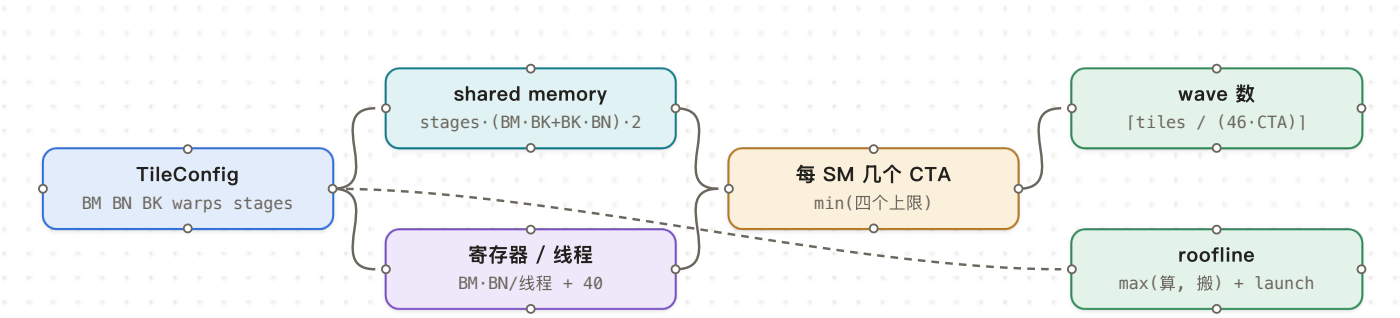
`triton › Triton tutorial 03: matmul` · `triton › autotuner.py: early_config_prune / perf_model` · CUDA Programming Guide：occupancy · Nsight Compute

要用到的知识

一个 tiled GEMM 让每个 CTA 负责 C 的一块 `BM × BN`，沿 K 每次搬 `A[BM×BK]` 和 `B[BK×BN]` 进 shared memory。

量	公式	说明
FLOPs	`2·M·N·K`	每个 (m, n, k) 一次乘一次加
最少字节	`(MK + KN)·in + MN·out`	每个矩阵只读 / 写一次
实际字节（不计 L2）	`MK·[N/BN]·in + KN·[M/BM]·in + MN·out`	A 每列 tile 读一遍，B 每行 tile 读一遍
shared memory	`num_stages·(BM·BK + BK·BN)·in`	流水线有几级，就有几份缓冲
寄存器（一阶估计）	`BM·BN / 线程数 + 40`	fp32 累加器摊到每个线程，再加地址、计数器、操作数片段
每 SM 的 CTA 数	`min(线程限制, 寄存器限制, smem 限制, 16)`	最小的那一项就是 **limiter**
wave	`[tiles / (SM 数 × 每 SM CTA 数)]`	最后一轮可能大半空着（tail effect）

一个配置的账：从 tile 到 limiter



例：128×128×64、4 warp、3 stage 的 fp16 配置，shared memory = 3·(128·64+64·128)·2 = 98,304 B，RTX 3070 每 SM 100 KB，所以每 SM 只能放 1 个 CTA，limiter 是 smem。4096³ 的问题有 1024 个 tile、每轮 46 个，需要 23 轮，最后一轮只有 12/46 满。

类比 · 搬家公司排班

一个 CTA 是一辆货车加一组工人，SM 是停车位。货车越大（tile 越大），每趟搬得越多、来回越少（数据复用高），但一个车位只停得下一辆（occupancy 低）。车位总数固定（46），货物分成 1024 车，就要排 23 轮；最后一轮只有 12 辆车在干活，其他车位空着——这就是 wave 的尾巴。K1 就是在出车前把这些数都算出来。

题目要求

在 `mine/k1_gemm_model.py` 里实现下面的函数（`GPU`、`TileConfig`、`RTX3070` 已给出）：

函数	返回
`gemm_flops(M, N, K)`	`2·M·N·K`
`gemm_min_bytes(M, N, K, in_bytes, out_bytes)`	最少 DRAM 字节
`arithmetic_intensity(flops, nbytes)`	FLOP / Byte
`tiled_bytes(M, N, K, cfg, ...)`	不计 L2 时 tiled kernel 的实际字节
`smem_bytes(cfg)` / `threads(cfg)` / `regs_per_thread(cfg)`	资源
`occupancy(gpu, cfg)`	`{"blocks", "limiter", "valid", "reason"}`；超过单 CTA smem 或 255 寄存器时 `valid=False`
`waves(gpu, M, N, cfg, blocks_per_sm)`	`{"tiles", "per_wave", "waves", "tail_fill"}`
`roofline(gpu, flops, nbytes)`	`{"time", "t_compute", "t_memory", "bound"}`，`bound ∈ {compute, memory, launch}`
`analyze(gpu, M, N, K, cfg)`	上面所有结果打包

题干

mine/k1_gemm_model.py 107 行 复制

```python
"""K1 · GEMM tile & resource model — your implementation (run ./check.sh k1).

Pure Python. Given a GEMM shape and a tile configuration (what Triton calls
BLOCK_M / BLOCK_N / BLOCK_K / num_warps / num_stages), predict *before running anything*:

  * how many FLOPs and how many bytes the problem needs at minimum,
  * how many bytes the tiled kernel really moves (A and B are re-read once per tile column
  * how much shared memory and roughly how many registers one CTA needs,
  * how many CTAs fit on one SM (occupancy) and which resource limits it,
  * how many waves of CTAs the grid needs and how full the last wave is,
  * a roofline estimate: is this config memory-, compute- or launch-bound?

These are exactly the numbers you compare against Nsight Compute in T3, and the numbers a
compiler's cost model (Luminal's search, Triton's autotuner pruning) reasons about.
"""
from dataclasses import dataclass
from math import ceil


@dataclass(frozen=True)
class GPU:
    name: str
    sm_count: int
    smem_per_sm: int            # bytes of shared memory per SM
    smem_per_block: int         # max bytes one CTA may use
    regs_per_sm: int            # 32-bit registers per SM
    max_threads_per_sm: int
    max_blocks_per_sm: int
    max_regs_per_thread: int
    peak_flops: float           # FLOP/s for the dtype being modelled (e.g. FP16 Tensor Co
    peak_bw: float              # bytes/s of DRAM bandwidth
    launch_overhead: float = 5e-6   # seconds; below ~this a kernel is launch-bound


# Approximate public specs. RTX 3070 = GA104, SM86 (the roadmap's starting GPU).
RTX3070 = GPU("RTX 3070 (SM86)", sm_count=46, smem_per_sm=100 * 1024, smem_per_block=99 *
              regs_per_sm=65536, max_threads_per_sm=1536, max_blocks_per_sm=16, max_regs_p
              peak_flops=40e12, peak_bw=448e9)


@dataclass(frozen=True)
class TileConfig:
    BM: int
    BN: int
    BK: int
    num_warps: int = 4
    num_stages: int = 3


def gemm_flops(M, N, K):
```

```
    """One multiply + one add per (m, n, k)."""
    raise NotImplementedError("gemm_flops")


def gemm_min_bytes(M, N, K, in_bytes=2, out_bytes=2):
    """Read A and B once, write C once: the lower bound on DRAM traffic."""
    raise NotImplementedError("gemm_min_bytes")


def arithmetic_intensity(flops, nbytes):
    raise NotImplementedError("arithmetic_intensity")


def tiled_bytes(M, N, K, cfg: TileConfig, in_bytes=2, out_bytes=2):
    """Bytes a tiled kernel loads from DRAM (ignoring L2 hits).

    Each CTA owns a BM x BN tile of C and walks all of K, so the whole of A is read once p
    column of tiles (ceil(N / BN) times) and the whole of B once per row of tiles."""
    raise NotImplementedError("tiled_bytes")


def smem_bytes(cfg: TileConfig, in_bytes=2):
    """num_stages copies of one A tile (BM x BK) and one B tile (BK x BN)."""
    raise NotImplementedError("smem_bytes")


def threads(cfg: TileConfig):
    raise NotImplementedError("threads")


def regs_per_thread(cfg: TileConfig, overhead=40):
    """Rough estimate: the fp32 accumulator tile spread over the CTA's threads, plus a fix
    overhead for addresses, loop counters and operand fragments. Real numbers come from
    `ptxas -v` / Nsight; this is the first-order term that grows with the tile."""
    raise NotImplementedError("regs_per_thread")


def occupancy(gpu: GPU, cfg: TileConfig, in_bytes=2):
    """CTAs that can be resident on one SM, and which resource is the limit.

    Returns {"blocks": int, "limiter": str, "valid": bool, "reason": str}."""
    raise NotImplementedError("occupancy")


def waves(gpu: GPU, M, N, cfg: TileConfig, blocks_per_sm):
    """How many rounds of CTAs the grid needs, and how full the last round is (0..1]."""
    raise NotImplementedError("waves")


def roofline(gpu: GPU, flops, nbytes):
    """Lower-bound time and the bound that sets it."""
```

```
    raise NotImplementedError("roofline")


def analyze(gpu: GPU, M, N, K, cfg: TileConfig, in_bytes=2, out_bytes=2):
    """Everything above for one (shape, config) pair."""
    raise NotImplementedError("analyze")
```

验收

```
./check.sh k1
```

tests/test_k1_gemm_model.py 76 行 复制

```
import math

import pytest

from drill import load

k1 = load("k1_gemm_model")
G = k1.RTX3070
C = k1.TileConfig


def test_flops_and_min_bytes():
    assert k1.gemm_flops(4096, 4096, 4096) == 2 * 4096 ** 3
    # fp16 A, B, C: three 4096x4096 matrices x 2 bytes
    assert k1.gemm_min_bytes(4096, 4096, 4096) == 3 * 4096 * 4096 * 2
    assert k1.gemm_min_bytes(8, 16, 4, in_bytes=4, out_bytes=4) == (8 * 4 + 4 * 16 + 8 * 1


def test_tiled_bytes_rereads_operands():
    # 1024^3 with 128x128 tiles: A is read 1024/128 = 8 times, B 8 times
    got = k1.tiled_bytes(1024, 1024, 1024, C(128, 128, 64))
    assert got == (1024 * 1024 * 8 * 2) * 2 + 1024 * 1024 * 2
    # partial tiles count as a full re-read
    assert k1.tiled_bytes(100, 100, 10, C(64, 64, 16), in_bytes=1, out_bytes=1) == 100 * 1


def test_smem_regs_threads():
    cfg = C(128, 128, 64, num_warps=4, num_stages=3)
    assert k1.smem_bytes(cfg) == 3 * (128 * 64 + 64 * 128) * 2 == 98304
    assert k1.threads(cfg) == 128
    assert k1.regs_per_thread(cfg) == 128 * 128 // 128 + 40


def test_occupancy_limiters():
    big = k1.occupancy(G, C(128, 128, 64, 4, 3))
    assert big["valid"] and big["blocks"] == 1 and big["limiter"] == "smem"

    small = k1.occupancy(G, C(64, 64, 32, 4, 2))
    # threads 12, regs 65536 // (72*128) = 7, smem 102400 // 16384 = 6, blocks 16
    assert small["blocks"] == 6 and small["limiter"] == "smem"

    too_much_smem = k1.occupancy(G, C(128, 256, 64, 8, 3))
    assert not too_much_smem["valid"] and too_much_smem["limiter"] == "smem"

    spills = k1.occupancy(G, C(256, 256, 32, 4, 2))
    assert not spills["valid"] and spills["limiter"] == "regs"


def test_waves_and_tail():
    w = k1.waves(G, 4096, 4096, C(128, 128, 64), blocks_per_sm=1)
```

```
    assert w["tiles"] == 1024 and w["per_wave"] == 46 and w["waves"] == 23
    assert w["tail_fill"] == pytest.approx(12 / 46)
    exact = k1.waves(G, 46 * 64, 64, C(64, 64, 32), blocks_per_sm=1)
    assert exact["waves"] == 1 and exact["tail_fill"] == 1.0


def test_roofline_bounds():
    flops = k1.gemm_flops(4096, 4096, 4096)
    r = k1.roofline(G, flops, k1.gemm_min_bytes(4096, 4096, 4096))
    assert r["bound"] == "compute" and r["time"] == pytest.approx(flops / G.peak_flops)
    # a vector add: 1 flop per 6 bytes
    n = 1 << 24
    assert k1.roofline(G, n, 6 * n)["bound"] == "memory"
    # tiny problem: below the launch overhead
    tiny = k1.roofline(G, k1.gemm_flops(64, 64, 64), k1.gemm_min_bytes(64, 64, 64))
    assert tiny["bound"] == "launch" and tiny["time"] == G.launch_overhead


def test_analyze_bundle():
    a = k1.analyze(G, 2048, 2048, 2048, C(128, 128, 32, 4, 3))
    assert a["flops"] == 2 * 2048 ** 3
    assert a["intensity"] == pytest.approx(a["flops"] / a["tiled_bytes"])
    assert a["occupancy"]["valid"] and "waves" in a
    bad = k1.analyze(G, 2048, 2048, 2048, C(256, 256, 64, 4, 4))
    assert not bad["occupancy"]["valid"] and "waves" not in bad
    assert math.isclose(a["roofline"]["t_compute"], a["flops"] / G.peak_flops)
```

提示

L1 先写最简单的四个 方向（轻度剧透） 收起

`gemm_flops`、`gemm_min_bytes`、`smem_bytes`、`threads` 都是一行公式。`tiled_bytes` 注意用 `ceil`：100 行的问题配 64 行的 tile，B 也要被读 2 遍。

L2 occupancy 的四个上限 拆解与常见坑（中度剧透） 收起

`threads: max_threads_per_sm // t`，`regs: regs_per_sm // (regs · t)`，`smem: smem_per_sm // smem`，`blocks: max_blocks_per_sm`。先检查两个“根本跑不了”的情况（smem 超过 `smem_per_block`、寄存器超过 255），再取最小值。同为最小时按名字排序，保证 limiter 确定。

L3 **roofline 和 waves** 几乎是答案（重度剧透） 收起 ▾

`roofline`：`t = max(flops/peak_flops, bytes/peak_bw)`；若 `t < launch_overhead`，`bound = "launch"`、`time = launch_overhead`。`waves`：`tiles = [M/BM]·[N/BN]`，`per_wave = sm_count · blocks_per_sm`，最后一轮的填充率 `= (tiles − (waves−1)·per_wave) / per_wave`。

参考解

解 solutions/k1_gemm_model.py 先自己写完、跑过测试再看 收起

solutions/k1_gemm_model.py 153 行 复制

```python
"""K1 · GEMM tile & resource model — reference solution.

Pure Python. Given a GEMM shape and a tile configuration (what Triton calls
BLOCK_M / BLOCK_N / BLOCK_K / num_warps / num_stages), predict *before running anythin

  * how many FLOPs and how many bytes the problem needs at minimum,
  * how many bytes the tiled kernel really moves (A and B are re-read once per tile co
  * how much shared memory and roughly how many registers one CTA needs,
  * how many CTAs fit on one SM (occupancy) and which resource limits it,
  * how many waves of CTAs the grid needs and how full the last wave is,
  * a roofline estimate: is this config memory-, compute- or launch-bound?

These are exactly the numbers you compare against Nsight Compute in T3, and the number
compiler's cost model (Luminal's search, Triton's autotuner pruning) reasons about.
"""
from dataclasses import dataclass
from math import ceil


@dataclass(frozen=True)
class GPU:
    name: str
    sm_count: int
    smem_per_sm: int            # bytes of shared memory per SM
    smem_per_block: int         # max bytes one CTA may use
    regs_per_sm: int            # 32-bit registers per SM
    max_threads_per_sm: int
    max_blocks_per_sm: int
    max_regs_per_thread: int
    peak_flops: float           # FLOP/s for the dtype being modelled (e.g. FP16 Tensc
    peak_bw: float              # bytes/s of DRAM bandwidth
    launch_overhead: float = 5e-6   # seconds; below ~this a kernel is launch-bound


# Approximate public specs. RTX 3070 = GA104, SM86 (the roadmap's starting GPU).
RTX3070 = GPU("RTX 3070 (SM86)", sm_count=46, smem_per_sm=100 * 1024, smem_per_block=9
              regs_per_sm=65536, max_threads_per_sm=1536, max_blocks_per_sm=16, max_re
              peak_flops=40e12, peak_bw=448e9)


@dataclass(frozen=True)
class TileConfig:
    BM: int
    BN: int
    BK: int
    num_warps: int = 4
    num_stages: int = 3


def gemm_flops(M, N, K):
```

```
    """One multiply + one add per (m, n, k)."""
    return 2 * M * N * K


def gemm_min_bytes(M, N, K, in_bytes=2, out_bytes=2):
    """Read A and B once, write C once: the lower bound on DRAM traffic."""
    return (M * K + K * N) * in_bytes + M * N * out_bytes


def arithmetic_intensity(flops, nbytes):
    return flops / nbytes


def tiled_bytes(M, N, K, cfg: TileConfig, in_bytes=2, out_bytes=2):
    """Bytes a tiled kernel loads from DRAM (ignoring L2 hits).

    Each CTA owns a BM x BN tile of C and walks all of K, so the whole of A is read on
    column of tiles (ceil(N / BN) times) and the whole of B once per row of tiles."""
    a_reads = M * K * ceil(N / cfg.BN)
    b_reads = K * N * ceil(M / cfg.BM)
    return (a_reads + b_reads) * in_bytes + M * N * out_bytes


def smem_bytes(cfg: TileConfig, in_bytes=2):
    """num_stages copies of one A tile (BM x BK) and one B tile (BK x BN)."""
    return cfg.num_stages * (cfg.BM * cfg.BK + cfg.BK * cfg.BN) * in_bytes


def threads(cfg: TileConfig):
    return cfg.num_warps * 32


def regs_per_thread(cfg: TileConfig, overhead=40):
    """Rough estimate: the fp32 accumulator tile spread over the CTA's threads, plus a
    overhead for addresses, loop counters and operand fragments. Real numbers come fro
    `ptxas -v` / Nsight; this is the first-order term that grows with the tile."""
    return cfg.BM * cfg.BN // threads(cfg) + overhead


def occupancy(gpu: GPU, cfg: TileConfig, in_bytes=2):
    """CTAs that can be resident on one SM, and which resource is the limit.

    Returns {"blocks": int, "limiter": str, "valid": bool, "reason": str}."""
    t = threads(cfg)
    regs = regs_per_thread(cfg)
    smem = smem_bytes(cfg, in_bytes)
    if smem > gpu.smem_per_block:
        return {"blocks": 0, "limiter": "smem", "valid": False,
                "reason": f"needs {smem} B shared memory, a CTA may use {gpu.smem_per_
    if regs > gpu.max_regs_per_thread:
        return {"blocks": 0, "limiter": "regs", "valid": False,
```

```
                "reason": f"~{regs} registers/thread > {gpu.max_regs_per_thread}: will
    limits = {
        "threads": gpu.max_threads_per_sm // t,
        "regs": gpu.regs_per_sm // (regs * t),
        "smem": gpu.smem_per_sm // smem if smem else gpu.max_blocks_per_sm,
        "blocks": gpu.max_blocks_per_sm,
    }
    limiter = min(limits, key=lambda k: (limits[k], k))
    blocks = limits[limiter]
    return {"blocks": blocks, "limiter": limiter, "valid": blocks > 0,
            "reason": "" if blocks > 0 else f"{limiter} allows 0 CTAs per SM"}


def waves(gpu: GPU, M, N, cfg: TileConfig, blocks_per_sm):
    """How many rounds of CTAs the grid needs, and how full the last round is (0..1]."
    tiles = ceil(M / cfg.BM) * ceil(N / cfg.BN)
    per_wave = gpu.sm_count * blocks_per_sm
    n = ceil(tiles / per_wave)
    last = tiles - (n - 1) * per_wave
    return {"tiles": tiles, "per_wave": per_wave, "waves": n, "tail_fill": last / per_


def roofline(gpu: GPU, flops, nbytes):
    """Lower-bound time and the bound that sets it."""
    t_compute = flops / gpu.peak_flops
    t_memory = nbytes / gpu.peak_bw
    t = max(t_compute, t_memory)
    if t < gpu.launch_overhead:
        bound = "launch"
    else:
        bound = "compute" if t_compute >= t_memory else "memory"
    return {"time": max(t, gpu.launch_overhead), "t_compute": t_compute, "t_memory": t


def analyze(gpu: GPU, M, N, K, cfg: TileConfig, in_bytes=2, out_bytes=2):
    """Everything above for one (shape, config) pair."""
    occ = occupancy(gpu, cfg, in_bytes)
    flops = gemm_flops(M, N, K)
    moved = tiled_bytes(M, N, K, cfg, in_bytes, out_bytes)
    out = {
        "flops": flops,
        "min_bytes": gemm_min_bytes(M, N, K, in_bytes, out_bytes),
        "tiled_bytes": moved,
        "intensity": arithmetic_intensity(flops, moved),
        "smem": smem_bytes(cfg, in_bytes),
        "regs": regs_per_thread(cfg),
        "occupancy": occ,
        "roofline": roofline(gpu, flops, moved),
    }
    if occ["valid"]:
```

```
        out["waves"] = waves(gpu, M, N, cfg, occ["blocks"])
    return out
```

在 RTX 3070 上对照

```
# kernel[grid](...) 返回 CompiledKernel；launch 之后它带着 ptxas 报告的资源数
#  (python/triton/compiler/compiler.py：self.n_regs, self.n_spills = driver.active.utils.lo
k = matmul_kernel[grid](a, b, c, M, N, K, a.stride(0), a.stride(1), b.stride(0), b.stride(
                        c.stride(0), c.stride(1), BLOCK_SIZE_M=128, BLOCK_SIZE_N=128, BLOC
                        GROUP_SIZE_M=8, num_warps=4, num_stages=3)   # 不带 @autotune 的版本
print(k.n_regs, k.n_spills, k.metadata.shared)   # 对照 regs_per_thread() / smem_bytes()
```

```
ncu --set full -k regex:matmul_kernel python run_matmul.py   # Registers Per Thread、Achiev
```

把 `n_regs` 和你的一阶估计比较：估计偏低多少？差的部分是地址计算、A/B 操作数片段（`ldmatrix` 的目标寄存器）和流水线状态。

▼ 自测：为什么不计 L2 的模型会说 4096^3 GEMM 是 memory-bound?

128×128 的 tile 下，A 和 B 各被重复读 32 次，`tiled_bytes` ≈ 2.2 GB，448 GB/s 要 4.9 ms，比 40 TFLOP/s 下的 3.4 ms 计算时间还长。但真实 GPU 上同一列 tile 的 CTA 几乎同时在读同一块 A，绝大部分重复读取会命中 4 MB 的 L2，真正到 DRAM 的字节接近 `gemm_min_bytes`（约 100 MB），于是变成 compute-bound。这就是 Triton tutorial 里做 **grouped ordering**（`GROUP_SIZE_M`）的原因：让同时运行的 CTA 共享更多 A/B tile，提高 L2 命中率。T3 的测试特意保留了这个“错误”的结论，提醒你模型的边界在哪里。

共同基础 · Track K · 共同知识主线 · 第 2 题

K2 可信的 benchmark：warmup、同步、中位数、compile 与 run 分离

一个“更快了 8%”的 PR 能不能被接受，取决于你的测量方法。这道题写一个可以信任的 benchmark harness：warmup、同步、重复取中位数、把编译 / 搜索时间和稳态时间分开，并且只在差异大于噪声时才说“更快”。

30 min | 难度 易 | 你写 `mine/k2_bench_harness.py` | 验收 `./check.sh k2`

参考解 `solutions/k2_bench_harness.py` | 📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 题目要求
5. 题干
6. 验收
7. 提示
8. 参考解
9. 在 RTX 3070 上：测出 launch-bound 到 memory-bound 的拐点

背景

roadmap 第 9 节对性能 PR 的要求是：“提供设备、版本、warmup、重复次数、median / variance、compile 与 runtime 分离”。这四件事在两个项目里都会遇到：

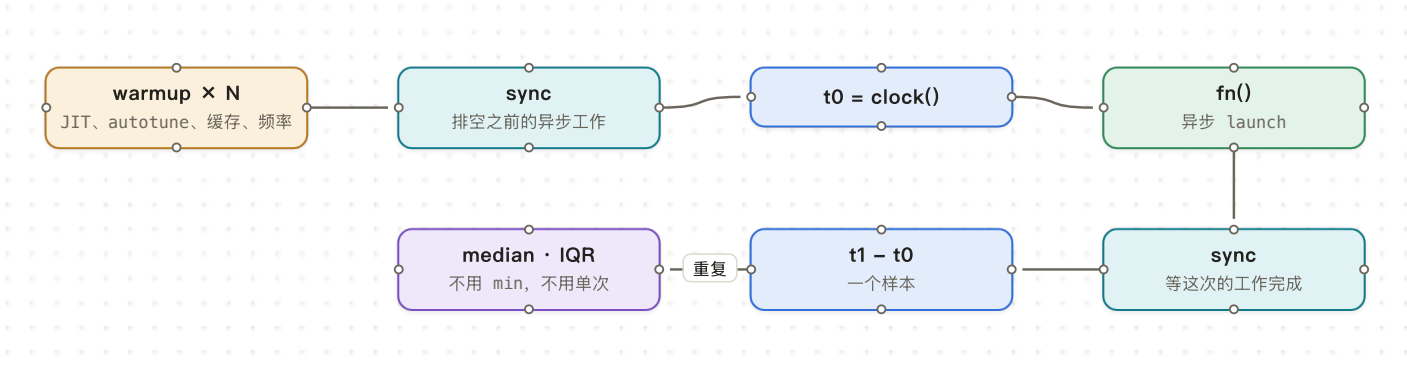
- **Triton**：第一次调用要 JIT 编译（加上 autotune，可能几秒），之后每次调用只要几微秒。
- **Luminal**：`cx.compile(...)` 里包含 e-graph 饱和和候选搜索（可能几十秒），之后的 `execute` 才是稳态。
- **GPU 是异步的**：`kernel[grid](...)` 返回时 kernel 可能还没开始跑；不先 `torch.cuda.synchronize()` 就读时钟，测到的是 launch 的时间。

源码与资料

`triton › triton.testing.do_bench`（`quantiles`、`return_mode`）· `triton › autotuner 用的 quantiles=(0.5, 0.2, 0.8)` · Luminal README 的 `LUMINAL_CUDA_PROFILE_RECAPTURE`（prepare / launch / sync 分解）· Nsight Systems

要用到的知识

一次可信的测量



两次 sync 缺一不可：开始前不 sync，样本里混进了上一轮的尾巴；结束前不 sync，测到的只是 launch 的 CPU 开销。

统计量	为什么
median	对偶发的慢样本（被别的进程抢占、频率波动）不敏感；mean 会被一个离群值拖走
IQR（q3 − q1）	噪声的尺度；两次测量的 IQR 区间重叠，就不能说谁更快
p20 / p80	Triton autotuner 用 `quantiles=(0.5, 0.2, 0.8)` 报告的就是这三个
CV（stdev / mean）	稳定性；CV 大于几个百分点时，先修环境再谈优化
Tukey fences	`[q1 − 1.5·IQR, q3 + 1.5·IQR]` 外面的样本标为离群

类比 · 量体重

早上起床量一次体重，说明不了你瘦了没有：喝没喝水、穿没穿鞋都会差一公斤。可信的做法是：每天同一时间（同步点）、先站稳（warmup）、连续量一周取中位数，再和上周的中位数比；两周的波动范围（IQR）还重叠，就别急着宣布减肥成功。“第一次量”还包括了找体重秤、开机的时间——那是 compile time，要单独记。

题目要求

在 `mine/k2_bench_harness.py` 里实现（`FakeClock` 已给出：每次 `work()` 推进时间，用来在测试里扮演 GPU）：

函数	要点
`percentile(xs, q)`	线性插值，和 numpy 默认一致
`summarize(samples)`	n、median、mean、stdev、min、max、p10、p90、q1、q3、iqr、cv
`bench(fn, warmup, repeats, timer, sync)`	warmup 不计时；每个样本前后各 sync 一次；warmup 之后也 sync 一次
`compile_vs_run(make_fn, repeats, timer, sync)`	`make_fn()` 的构造时间 + 第一次调用 = 一次性成本；之后的中位数 = 稳态
`outliers(samples, k)`	Tukey fences 之外的下标
`compare(base, new)`	`speedup = base.median / new.median`；IQR 不重叠才判 faster / slower

题干

mine/k2_bench_harness.py 72 行 复制

```python
"""K2 · A benchmark harness you can trust — your implementation (run ./check.sh k2).

The same rules apply to a Triton kernel, a Luminal graph or a cuBLAS call:

  * warm up first (JIT compile, autotune, caches, clocks) and never time the warmup;
  * synchronize before reading the clock (GPU launches are asynchronous);
  * repeat, and report the median and the spread, not the best run;
  * keep compile / search time separate from steady-state run time;
  * only call a change faster when the difference is larger than the noise.

Everything takes an injectable `timer` and `sync`, so tests drive it with a fake clock and
real run passes `time.perf_counter` and `torch.cuda.synchronize`.
"""
import statistics
import time


def percentile(xs, q):
    """Linear-interpolation percentile, q in [0, 100] (same as numpy's default)."""
    raise NotImplementedError("percentile")


def summarize(samples):
    """Robust statistics of a list of timings."""
    raise NotImplementedError("summarize")


def bench(fn, warmup=10, repeats=50, timer=time.perf_counter, sync=lambda: None):
    """Time fn() `repeats` times after `warmup` untimed calls.

    sync() runs before *starting* the clock (drain earlier async work) and before *stoppin
    (wait for this call's work). Returns summarize(samples) plus the raw samples."""
    raise NotImplementedError("bench")


def compile_vs_run(make_fn, repeats=20, timer=time.perf_counter, sync=lambda: None):
    """Separate one-time cost (building / compiling / searching + the first call) from
    steady-state cost. make_fn() returns the callable; that construction is part of the
    one-time cost, exactly like `torch.compile`, a Triton JIT or a Luminal search."""
    raise NotImplementedError("compile_vs_run")


def outliers(samples, k=1.5):
    """Indices outside Tukey's fences [q1 - k*IQR, q3 + k*IQR]."""
    raise NotImplementedError("outliers")


def compare(base, new):
    """Is `new` faster than `base`? Both are summarize() results.
```

```
    speedup = base.median / new.median. The verdict is "faster" / "slower" only when the
    interquartile ranges do not overlap; otherwise "within noise"."""
    raise NotImplementedError("compare")


class FakeClock:
    """Deterministic clock for tests and for reasoning about the harness: each fn() call
    advances time by the next duration in `durations` (cycled)."""

    def __init__(self, durations):
        self.t = 0.0
        self.durations = list(durations)
        self.i = 0
        self.calls = 0

    def __call__(self):
        return self.t

    def work(self):
        self.t += self.durations[self.i % len(self.durations)]
        self.i += 1
        self.calls += 1
```

验收

```
./check.sh k2
```

tests/test_k2_bench_harness.py 59 行 复制

```
import pytest

from drill import load

k2 = load("k2_bench_harness")


def test_percentile_matches_numpy_linear():
    xs = [5, 1, 4, 2, 3]
    assert k2.percentile(xs, 0) == 1 and k2.percentile(xs, 100) == 5 and k2.percentile(xs,
    assert k2.percentile([1, 2, 3, 4], 25) == pytest.approx(1.75)


def test_summarize_basic():
    s = k2.summarize([1.0, 2.0, 3.0, 4.0, 100.0])
    assert s["median"] == 3.0 and s["min"] == 1.0 and s["max"] == 100.0
    assert s["q1"] == 2.0 and s["q3"] == 4.0 and s["iqr"] == 2.0
    assert s["mean"] == 22.0          # the mean is dragged by the outlier, the median is


def test_bench_skips_warmup_and_syncs():
    # warmup calls take 10 s (compilation), steady calls take 1 s
    clock = k2.FakeClock([10.0] * 3 + [1.0] * 100)
    syncs = []
    r = k2.bench(clock.work, warmup=3, repeats=20, timer=clock, sync=lambda: syncs.append(
    assert r["n"] == 20 and r["median"] == 1.0 and max(r["samples"]) == 1.0
    assert clock.calls == 23
    assert len(syncs) == 1 + 2 * 20     # once after warmup, before and after every timed


def test_compile_vs_run():
    clock = k2.FakeClock([1.0])
    built = []

    def make():
        clock.t += 7.0                  # "compilation"
        built.append(1)
        return clock.work

    r = k2.compile_vs_run(make, repeats=5, timer=clock)
    assert built == [1]
    assert r["first_call"] == pytest.approx(8.0)
    assert r["steady_median"] == pytest.approx(1.0)
    assert r["one_time_cost"] == pytest.approx(7.0)


def test_outliers():
    xs = [1.0, 1.1, 0.9, 1.0, 1.05, 0.95, 5.0]
    assert k2.outliers(xs) == [6]
```

```python
def test_compare_needs_more_than_noise():
    base = k2.summarize([10, 11, 12, 10.5, 11.5])
    clearly = k2.summarize([5, 5.5, 6, 5.2, 5.8])
    noisy = k2.summarize([9.5, 11, 12.5, 10, 11.8])
    assert k2.compare(base, clearly)["verdict"] == "faster"
    assert k2.compare(base, clearly)["speedup"] == pytest.approx(11 / 5.5)
    assert k2.compare(base, noisy)["verdict"] == "within noise"
    assert k2.compare(clearly, base)["verdict"] == "slower"
```

提示

L1 percentile 方向（轻度剧透） 收起

排序，`pos = (n - 1)·q/100`，在 `s[floor(pos)]` 和 `s[floor(pos)+1]` 之间线性插值。`[1, 2, 3, 4]` 的 25 分位是 1.75。

L2 sync 的次数 拆解与常见坑（中度剧透） 收起

测试数的是：warmup 之后 1 次，每个样本前后各 1 次，共 `1 + 2·repeats` 次。

L3 compile_vs_run 与 compare 几乎是答案（重度剧透） 收起

`compile_vs_run`：`sync(); t0 = timer(); fn = make_fn(); fn(); sync(); first = timer() - t0`，然后对 `fn` 跑一次 `bench`，`one_time_cost = first - steady_median`。`compare`：区间 `[q1, q3]` 重叠 ⇔ `not (new.q3 < base.q1 or base.q3 < new.q1)`。

参考解

解 solutions/k2_bench_harness.py 先自己写完、跑过测试再看 收起 ▾

solutions/k2_bench_harness.py 114 行 复制

```
"""K2 · A benchmark harness you can trust — reference solution.

The same rules apply to a Triton kernel, a Luminal graph or a cuBLAS call:

  * warm up first (JIT compile, autotune, caches, clocks) and never time the warmup;
  * synchronize before reading the clock (GPU launches are asynchronous);
  * repeat, and report the median and the spread, not the best run;
  * keep compile / search time separate from steady-state run time;
  * only call a change faster when the difference is larger than the noise.

Everything takes an injectable `timer` and `sync`, so tests drive it with a fake clock
real run passes `time.perf_counter` and `torch.cuda.synchronize`.
"""
import statistics
import time


def percentile(xs, q):
    """Linear-interpolation percentile, q in [0, 100] (same as numpy's default)."""
    s = sorted(xs)
    if not s:
        raise ValueError("no samples")
    pos = (len(s) - 1) * q / 100
    lo = int(pos)
    hi = min(lo + 1, len(s) - 1)
    return s[lo] + (s[hi] - s[lo]) * (pos - lo)


def summarize(samples):
    """Robust statistics of a list of timings."""
    med = statistics.median(samples)
    mean = statistics.fmean(samples)
    sd = statistics.stdev(samples) if len(samples) > 1 else 0.0
    q1, q3 = percentile(samples, 25), percentile(samples, 75)
    return {
        "n": len(samples), "median": med, "mean": mean, "stdev": sd, "min": min(sample
        "p10": percentile(samples, 10), "p90": percentile(samples, 90),
        "q1": q1, "q3": q3, "iqr": q3 - q1, "cv": sd / mean if mean else 0.0,
    }


def bench(fn, warmup=10, repeats=50, timer=time.perf_counter, sync=lambda: None):
    """Time fn() `repeats` times after `warmup` untimed calls.

    sync() runs before *starting* the clock (drain earlier async work) and before *sto
    (wait for this call's work). Returns summarize(samples) plus the raw samples."""
    for _ in range(warmup):
        fn()
    sync()
    samples = []
```

```
    for _ in range(repeats):
        sync()
        t0 = timer()
        fn()
        sync()
        samples.append(timer() - t0)
    out = summarize(samples)
    out["samples"] = samples
    return out


def compile_vs_run(make_fn, repeats=20, timer=time.perf_counter, sync=lambda: None):
    """Separate one-time cost (building / compiling / searching + the first call) from
    steady-state cost. make_fn() returns the callable; that construction is part of th
    one-time cost, exactly like `torch.compile`, a Triton JIT or a Luminal search."""
    sync()
    t0 = timer()
    fn = make_fn()
    fn()
    sync()
    first = timer() - t0
    steady = bench(fn, warmup=2, repeats=repeats, timer=timer, sync=sync)
    return {"first_call": first, "steady_median": steady["median"],
            "one_time_cost": first - steady["median"], "steady": steady}


def outliers(samples, k=1.5):
    """Indices outside Tukey's fences [q1 - k*IQR, q3 + k*IQR]."""
    q1, q3 = percentile(samples, 25), percentile(samples, 75)
    lo, hi = q1 - k * (q3 - q1), q3 + k * (q3 - q1)
    return [i for i, x in enumerate(samples) if x < lo or x > hi]


def compare(base, new):
    """Is `new` faster than `base`? Both are summarize() results.

    speedup = base.median / new.median. The verdict is "faster" / "slower" only when t
    interquartile ranges do not overlap; otherwise "within noise"."""
    speedup = base["median"] / new["median"]
    overlap = not (new["q3"] < base["q1"] or base["q3"] < new["q1"])
    if overlap:
        verdict = "within noise"
    else:
        verdict = "faster" if speedup > 1 else "slower"
    return {"speedup": speedup, "verdict": verdict}


class FakeClock:
    """Deterministic clock for tests and for reasoning about the harness: each fn() ca
    advances time by the next duration in `durations` (cycled)."""
```

```
    def __init__(self, durations):
        self.t = 0.0
        self.durations = list(durations)
        self.i = 0
        self.calls = 0

    def __call__(self):
        return self.t

    def work(self):
        self.t += self.durations[self.i % len(self.durations)]
        self.i += 1
        self.calls += 1
```

在 RTX 3070 上：测出 launch-bound 到 memory-bound 的拐点

```
import torch, triton
from solutions.k2_bench_harness import bench, compare
for n in [2**k for k in range(10, 27, 2)]:
    x = torch.randn(n, device="cuda"); y = torch.randn_like(x)
    r = bench(lambda: x + y, warmup=20, repeats=100, sync=torch.cuda.synchronize)
    print(f"{n:>10} {r['median']*1e6:8.1f} us  {3*4*n/r['median']/1e9:7.1f} GB/s  cv={r['c
# 再和 triton.testing.do_bench(lambda: x + y, quantiles=(0.5, 0.2, 0.8)) 的结果对比
```

小向量时 median 基本不变（launch-bound），GB/s 随 n 线性上升；大到几百万元素后时间开始随 n 线性增长，GB/s 接近 448 GB/s 的上限（memory-bound）。注意 `time.perf_counter` + `synchronize` 测的是包括 launch 的端到端时间，`do_bench` 用 CUDA event 并在每次之间清 L2 缓存，两者在小 n 时会有差别。

▼ 自测：为什么报告 min 不如报告 median?

min 是“最幸运的一次”：缓存恰好热、频率恰好高、没有被打断。它不可复现，也不代表用户会看到的时间。median 是典型值，配上 IQR 还能说明可信度。roadmap 的 T3 也特意强调：“不要只报告最快一次；报告 median 和稳定性。”

共同基础 · Track K · 共同知识主线 · 第 3 题

K3 数值正确性：ULP、容差、重结合、边界形状

编译器总在重排浮点运算：split-K、树形归约、融合 epilogue、e-graph 里的 (a+b)+c → a+(b+c)。结果“不一样”不等于“错了”。这道题给你区分两者的工具：ULP 距离、随 dtype 和归约长度缩放的容差、专打边界的输入。

45 min | 难度 中 | 你写 `mine/k3_numerics.py` | 验收 `./check.sh k3`

参考解 `solutions/k3_numerics.py` | 📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 题目要求
5. 题干
6. 验收
7. 提示
8. 参考解
9. 在 RTX 3070 上：TF32 的坑

背景

roadmap 的共同知识表：“Numerical correctness：FP32 accumulation、rounding、reassociation、tolerance — 构造边界值测试和差分测试”。T4（Triton 最小复现）和 L5（Luminal 对拍）都会直接 import 本题的 `check_close`、`tolerance`、`edge_sizes`。

- **Triton**：同一个 `tl.sum` 换个 `num_warps`，归约顺序就变了；`tl.dot` 在 Ampere 上 fp32 输入默认走 TF32（`input_precision`），误差从 1e-7 量级变成 1e-3 量级。
- **Luminal**：L2 会证明 `(a+b)+c` 和 `a+(b+c)` 在 e-graph 里是同一个 e-class，但在 fp32 里可能差出整数；ReferenceRuntime 专门实现了窄浮点和 TF32 模拟（`luminal › narrow_float.rs` · `luminal › tf32.rs`）。

源码与资料

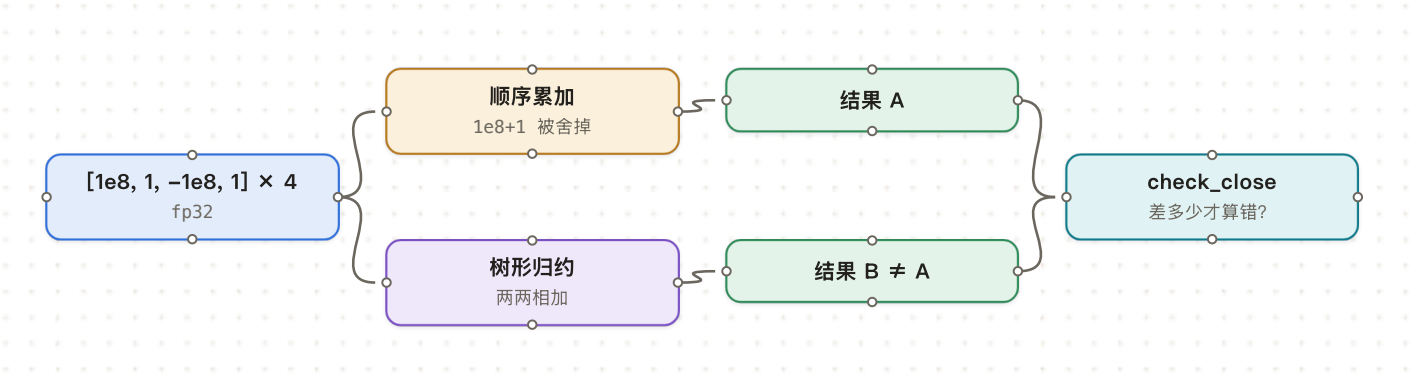
What Every Computer Scientist Should Know About Floating-Point Arithmetic · torch.testing.assert_close · `luminal › Luminal maximum 的 NaN 语义注释`

要用到的知识

dtype	尾数位	eps（1 附近的间距）	典型用途
fp32	23	$2^{-23} \approx 1.2e{-7}$	累加器
fp16	10	$2^{-10} \approx 9.8e{-4}$	存储 / Tensor Core 输入，范围小（最大 65504）
bf16	7	$2^{-7} \approx 7.8e{-3}$	存储，范围和 fp32 一样

- **ULP 距离**：把浮点数映射成“按大小排序的整数”（符号–数值 → 补码的技巧），两数之差就是中间隔了几个可表示值。`-0.0` 和 `+0.0` 相距 0 ULP。
- **容差为什么随 √k 增长**：长度 k 的求和，每一步的舍入误差大致随机，累积误差像随机游走，约 `√k · eps`。所以 `rtol = c · eps · √k`。
- **`|a − e| ≤ atol + rtol·|e|`**：接近 0 的值相对误差没有意义，靠 atol 兜底。

同一组数，两种求和顺序



两个结果都是“正确的 fp32 程序”的输出。判断编译器有没有 bug，比的是和 float64 参考值的距离是否在一个**说得出理由**的容差内，而不是两个低精度结果是否逐位相等。

类比 · 用不同的尺子量

fp32、fp16、bf16 像刻度分别是 0.1 mm、1 mm、1 cm 的尺子。用 1 cm 刻度的尺子量出来差 3 mm，不能说测量错了；但用 0.1 mm 的尺子差 3 mm 就该查一查了。容差就是“这把尺子、量这么长的东西，合理的误差范围”——尺子越粗、量的段数越多（归约越长），容差越大。

题目要求

在 `mine/k3_numerics.py` 里实现（`EPS` 和 `to_bf16` 已给出）：

函数	要点
`ulp_diff(a, b, dtype)`	fp32 / fp16 / bf16 三种；跨过 0 时 `-tiny` 到 `+tiny` 是 2
`sum_sequential(xs)` / `sum_pairwise(xs)`	每一步都在 fp32 里舍入；pairwise 奇数个时最后一个直接带到下一轮
`tolerance(dtype, k, c)`	`(rtol, atol)`，`rtol = atol = c·eps·√k`
`check_close(actual, expected, rtol, atol)`	返回报告：ok、mismatched、total、max_abs、max_rel、index；形状不同或 NaN 位置不同直接失败
`edge_sizes(block, limit)`	1、block−1、block、block+1、2·block+1、97，以及一个大的非整除尺寸
`edge_values(dtype)`	±0、最小正规数、±最大有限值、平方会溢出的值、±inf、NaN

题干

mine/k3_numerics.py 65 行 复制

```
"""K3 · Numerical correctness: ULPs, tolerances, reassociation, edge shapes — your impleme

Compilers reorder floating-point work all the time (split-K, tree reductions, fused epilog
e-graph rewrites like (a+b)+c -> a+(b+c)). The result is *different*, not necessarily *wro
To tell the two apart you need: a distance in ULPs, a tolerance that scales with the dtype
reduction length, and inputs that hit the edges (non-divisible shapes, size 1, huge/tiny v
"""
import math

import numpy as np

EPS = {"float32": 2.0 ** -23, "float16": 2.0 ** -10, "bfloat16": 2.0 ** -7}


def to_bf16(x):
    """Round float32 values to bfloat16 (round-to-nearest-even), returned as float32 value
    a = np.asarray(x, dtype=np.float32)
    bits = a.view(np.uint32).astype(np.uint64)
    rounding = ((bits >> 16) & 1) + 0x7FFF          # ties go to the even mantissa
    out = ((bits + rounding) >> 16) << 16
    return out.astype(np.uint32).view(np.float32)




def ulp_diff(a, b, dtype="float32"):
    """Number of representable `dtype` values between a and b (0 means identical)."""
    raise NotImplementedError("ulp_diff")


def sum_sequential(xs):
    raise NotImplementedError("sum_sequential")


def sum_pairwise(xs):
    """Tree reduction, the order a GPU block reduction (or tl.sum) uses."""
    raise NotImplementedError("sum_pairwise")


def tolerance(dtype, k=1, c=2.0):
    """(rtol, atol) for comparing a result computed in `dtype` with a high-precision refer

    The error of a length-k reduction grows roughly like sqrt(k) * eps when the rounding e
    are random, so rtol = c * eps * sqrt(k). atol covers values near zero, where relative
    is meaningless; using the same number is a reasonable default for O(1) data."""
    raise NotImplementedError("tolerance")


def check_close(actual, expected, rtol, atol):
    """Like torch.testing.assert_close, but returns a report instead of raising.
```

```
    passes iff |actual - expected| <= atol + rtol * |expected| everywhere (NaNs must match
    raise NotImplementedError("check_close")


def edge_sizes(block, limit=None):
    """Sizes that break tiled kernels: 1, block-1, block, block+1, 2*block+1, a prime, and
    (optionally) a large non-multiple. Sorted and de-duplicated."""
    raise NotImplementedError("edge_sizes")


def edge_values(dtype="float16"):
    """Values that stress a dtype: zeros of both signs, the smallest normal, the largest f
    values whose square overflows, and +/-inf / NaN."""
    raise NotImplementedError("edge_values")
```

验收

```
./check.sh k3
```

tests/test_k3_numerics.py 75 行 复制

```
import math

import numpy as np
import pytest

from drill import load

k3 = load("k3_numerics")


def test_bf16_rounding():
    x = np.array([1.0, 1.0 + 2 ** -8, 1.0 + 3 * 2 ** -8, 3.14159265], np.float32)
    got = k3.to_bf16(x)
    assert got[0] == 1.0
    assert got[1] == 1.0                      # exactly halfway -> even mantissa (1.0)
    assert got[2] == np.float32(1.0 + 2 ** -6)  # halfway between 1+2^-7 and 1+2^-6 -> eve
    assert abs(got[3] - 3.140625) < 1e-7      # bf16 keeps 8 significant bits


def test_ulp_diff():
    one = np.float32(1.0)
    nxt = np.nextafter(one, np.float32(2.0))
    assert k3.ulp_diff(one, nxt) == 1
    assert k3.ulp_diff(one, one) == 0
    assert k3.ulp_diff(np.float32(-0.0), np.float32(0.0)) == 0
    tiny = np.nextafter(np.float32(0), np.float32(1))
    assert k3.ulp_diff(-tiny, tiny) == 2      # across zero
    assert k3.ulp_diff(np.float16(1.0), np.float16(1.0 + 2 ** -10), "float16") == 1
    assert k3.ulp_diff(1.0, 1.0 + 2 ** -7, "bfloat16") == 1


def test_reassociation_changes_the_bits():
    xs = np.array([1e8, 1.0, -1e8, 1.0] * 4, np.float32)
    seq, tree = k3.sum_sequential(xs), k3.sum_pairwise(xs)
    assert seq != tree                          # same math, different rounding
    assert k3.sum_pairwise([1.0, 2.0, 3.0]) == 6.0 and k3.sum_pairwise([]) == 0.0


def test_tolerance_scales():
    r1, a1 = k3.tolerance("float32", 1)
    r4, _ = k3.tolerance("float32", 4)
    assert r1 == a1 == 2 * 2 ** -23
    assert r4 == pytest.approx(2 * r1)
    assert k3.tolerance("bfloat16", 1)[0] > k3.tolerance("float16", 1)[0] > r1


def test_check_close_report():
    e = np.array([1.0, 2.0, 0.0, np.nan])
    ok = k3.check_close(e + 1e-9, e, rtol=1e-6, atol=1e-6)
    assert ok["ok"] and ok["mismatched"] == 0
```

```
    bad = k3.check_close(np.array([1.0, 2.1, 0.0, np.nan]), e, rtol=1e-3, atol=1e-6)
    assert not bad["ok"] and bad["mismatched"] == 1 and bad["index"] == (1,)
    nanmis = k3.check_close(np.array([1.0, 2.0, np.nan, np.nan]), e, 1e-3, 1e-3)
    assert not nanmis["ok"] and "NaN" in nanmis["reason"]
    assert not k3.check_close(np.zeros(3), np.zeros(4), 1, 1)["ok"]


def test_fp16_matmul_needs_a_tolerance():
    rng = np.random.default_rng(0)
    k = 512
    a, b = rng.standard_normal((64, k)), rng.standard_normal((k, 64))
    ref = a @ b
    low = (a.astype(np.float16) @ b.astype(np.float16)).astype(np.float64)
    assert not k3.check_close(low, ref, *k3.tolerance("float32", k))["ok"]
    rtol, atol = k3.tolerance("float16", k, c=4.0)
    assert k3.check_close(low, ref, rtol, atol * math.sqrt(k))["ok"]


def test_edge_sizes_and_values():
    s = k3.edge_sizes(64, limit=1000)
    assert s[0] == 1 and {63, 64, 65, 129, 97}.issubset(s) and s == sorted(set(s))
    assert any(x % 64 for x in s if x > 129)
    v = k3.edge_values("float16")
    assert np.isnan(v).any() and np.isinf(v).sum() == 2
    assert np.isinf(np.float16(v[5]) * np.float16(v[5]))    # its square overflows fp16
```

提示

L1 ULP：把位模式变成有序整数 方向（轻度剧透） 收起

fp32：`i = a.view(np.int32)`；正数的位模式本身就是递增的，负数要翻转：`i < 0` 时取 `-(i & 0x7FFFFFFF)`。这样 `-0.0` 和 `+0.0` 都映射到 0。fp16 同理（`int16`、`0x7FFF`）。bf16 是 fp32 的高 16 位：`to_bf16(a).view(np.uint32) >> 16`，符号位是 `0x8000`。

L2 check_close 的顺序 拆解与常见坑（中度剧透） 收起

先比形状，再比 NaN 位置（`np.isnan(a) != np.isnan(e)`），然后在非 NaN 的位置算 `diff` 和 `allowed = atol + rtol·|e|`。`index` 取 `argmax(diff − allowed)` 并 `unravel_index`。

L3 **为什么 fp16 matmul 用 fp32 容差一定失败** 几乎是答案（重度剧透） 收起 ▾

测试里 `a.astype(float16) @ b.astype(float16)` 把输入舍入到了 fp16，单这一步的相对误差就有 2^{-11}，远大于 $2\cdot 2^{-23}\cdot\sqrt{512}$。用 `tolerance("float16", k, c=4)` 并把 atol 再乘 $\sqrt{k}$（结果的量级约为 $\sqrt{k}$）就能通过。

参考解

解 **solutions/k3_numerics.py** 先自己写完、跑过测试再看 收起 ▾

solutions/k3_numerics.py 112 行 复制

```
"""K3 · Numerical correctness: ULPs, tolerances, reassociation, edge shapes — referenc

Compilers reorder floating-point work all the time (split-K, tree reductions, fused ep
e-graph rewrites like (a+b)+c -> a+(b+c)). The result is *different*, not necessarily
To tell the two apart you need: a distance in ULPs, a tolerance that scales with the d
reduction length, and inputs that hit the edges (non-divisible shapes, size 1, huge/ti
"""
import math

import numpy as np

EPS = {"float32": 2.0 ** -23, "float16": 2.0 ** -10, "bfloat16": 2.0 ** -7}


def to_bf16(x):
    """Round float32 values to bfloat16 (round-to-nearest-even), returned as float32 v
    a = np.asarray(x, dtype=np.float32)
    bits = a.view(np.uint32).astype(np.uint64)
    rounding = ((bits >> 16) & 1) + 0x7FFF          # ties go to the even mantissa
    out = ((bits + rounding) >> 16) << 16
    return out.astype(np.uint32).view(np.float32)


def _ordered_int(a, dtype):
    """Map floats to integers whose order and spacing follow the float line (sign-magn
    two's-complement trick), so that |int(a) - int(b)| is the number of representable
    between a and b."""
    if dtype == "float32":
        i = np.asarray(a, np.float32).view(np.int32).astype(np.int64)
        return np.where(i < 0, -(i & 0x7FFFFFFF), i)
    if dtype == "float16":
        i = np.asarray(a, np.float16).view(np.int16).astype(np.int64)
        return np.where(i < 0, -(i & 0x7FFF), i)
    if dtype == "bfloat16":
        i = (to_bf16(a).view(np.uint32) >> 16).astype(np.int64)
        return np.where(i >= 0x8000, -(i & 0x7FFF), i)
    raise ValueError(dtype)


def ulp_diff(a, b, dtype="float32"):
    """Number of representable `dtype` values between a and b (0 means identical)."""
    return np.abs(_ordered_int(a, dtype) - _ordered_int(b, dtype))


def sum_sequential(xs):
    acc = np.float32(0)
    for x in np.asarray(xs, np.float32):
        acc = np.float32(acc + x)
    return acc
```

```
def sum_pairwise(xs):
    """Tree reduction, the order a GPU block reduction (or tl.sum) uses."""
    v = list(np.asarray(xs, np.float32))
    if not v:
        return np.float32(0)
    while len(v) > 1:
        nxt = [np.float32(v[i] + v[i + 1]) for i in range(0, len(v) - 1, 2)]
        if len(v) % 2:
            nxt.append(v[-1])
        v = nxt
    return v[0]


def tolerance(dtype, k=1, c=2.0):
    """(rtol, atol) for comparing a result computed in `dtype` with a high-precision r

    The error of a length-k reduction grows roughly like sqrt(k) * eps when the roundi
    are random, so rtol = c * eps * sqrt(k). atol covers values near zero, where relat
    is meaningless; using the same number is a reasonable default for O(1) data."""
    eps = EPS[dtype]
    rtol = c * eps * math.sqrt(max(k, 1))
    return rtol, rtol


def check_close(actual, expected, rtol, atol):
    """Like torch.testing.assert_close, but returns a report instead of raising.

    passes iff |actual - expected| <= atol + rtol * |expected| everywhere (NaNs must m
    a = np.asarray(actual, np.float64)
    e = np.asarray(expected, np.float64)
    if a.shape != e.shape:
        return {"ok": False, "reason": f"shape {a.shape} != {e.shape}"}
    nan_a, nan_e = np.isnan(a), np.isnan(e)
    if (nan_a != nan_e).any():
        idx = tuple(int(i) for i in np.argwhere(nan_a != nan_e)[0])
        return {"ok": False, "reason": f"NaN mismatch at {idx}", "index": idx}
    diff = np.where(nan_a, 0.0, np.abs(a - e))
    allowed = atol + rtol * np.abs(np.where(nan_e, 0.0, e))
    bad = diff > allowed
    rel = diff / np.maximum(np.abs(e), 1e-30)
    worst = tuple(int(i) for i in np.unravel_index(int(np.argmax(diff - allowed)), dif
    return {"ok": not bad.any(), "mismatched": int(bad.sum()), "total": int(diff.size)
            "max_abs": float(diff.max()) if diff.size else 0.0,
            "max_rel": float(np.where(nan_e, 0.0, rel).max()) if diff.size else 0.0, "


def edge_sizes(block, limit=None):
    """Sizes that break tiled kernels: 1, block-1, block, block+1, 2*block+1, a prime,
    (optionally) a large non-multiple. Sorted and de-duplicated."""
    s = {1, block - 1, block, block + 1, 2 * block + 1, 97}
```

```
    if limit:
        s.add(limit - (limit % block) + block // 2)
    return sorted(x for x in s if x >= 1)


def edge_values(dtype="float16"):
    """Values that stress a dtype: zeros of both signs, the smallest normal, the large
    values whose square overflows, and +/-inf / NaN."""
    fi = np.finfo(np.float16 if dtype == "float16" else np.float32)
    return np.array([0.0, -0.0, float(fi.tiny), float(fi.max), -float(fi.max), math.sq
                     1.0, -1.0, math.inf, -math.inf, math.nan], dtype=np.float32)
```

在 RTX 3070 上：TF32 的坑

```
import torch, triton, triton.language as tl
a = torch.randn(512, 512, device="cuda"); b = torch.randn(512, 512, device="cuda")
ref = (a.double() @ b.double())
# 写一个最小 tl.dot kernel，分别用 input_precision="ieee" 和 "tf32"，用 check_close + toleranc
```

你会看到 `"ieee"` 能过 `tolerance("float32", 512)`，`"tf32"` 只能过接近 `tolerance("bfloat16", 512)` 量级的容差（TF32 有 10 位尾数，比 bf16 还多 3 位，但远少于 fp32）。这就是为什么 PR 里的数值测试必须写明 dtype、精度模式和容差理由。

▼ 自测：e-graph 证明了 (a+b)+c ≡ a+(b+c)，为什么编译器还要小心使用这条规则？

e-graph 里的“等价”是**实数代数**上的等价。在浮点里，结合律不成立：`(1e8 + (-1e8)) + 1 = 1`，而 `1e8 + ((-1e8) + 1) = 0`（fp32 里 −1e8 + 1 舍入回 −1e8）。如果编译器自由使用这条规则，同一个模型在不同搜索结果下数值会不同。实际做法是：只在数值上可接受的场景使用（比如有 fp32 累加器、容差明确），并用 L5 那样的对拍测试兜底。

共同基础 · Track K · 共同知识主线 · 第 4 题

K4 Fusion 的账：launch、中间结果流量与寄存器压力

Fusion 省掉 launch、省掉中间结果在 DRAM 里的往返，但把更多东西塞进一个 kernel 的寄存器里。这道题把这笔账写成代码：给一条算子链和一个切分方案，算出字节数、launch 次数和寄存器峰值，再枚举所有切法找最快的。

30 min | 难度 中 | 你写 `mine/k4_fusion.py` | 验收 `./check.sh k4`

参考解 `solutions/k4_fusion.py` | 📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 题目要求
5. 题干
6. 验收
7. 提示
8. 参考解
9. 在 RTX 3070 上对照

背景

roadmap 的共同知识表：“Fusion：减少 launch / intermediate traffic 与 register pressure 的 trade-off — 对比 unfused 与 fused epilogue”。

- 在 **Triton** 里，fusion 就是把 bias、activation、residual 写进 GEMM kernel 的 epilogue：累加器还在寄存器里时就把后续 elementwise 做完，只写一次结果。
- 在 **Luminal** 里，fusion 是图级的选择：cuBLASLt 可以带 `CUBLASLT_EPILOGUE_RELU_BIAS` 一次完成 matmul + bias + relu（`luminal › cublaslt/device_call.rs`），也可以拆成三个 launch；搜索（L3）决定用哪种。

源码与资料

`triton › Triton tutorial 03: fused activation` · cuBLASLt epilogue · `luminal › Luminal cuBLASLt 说明`

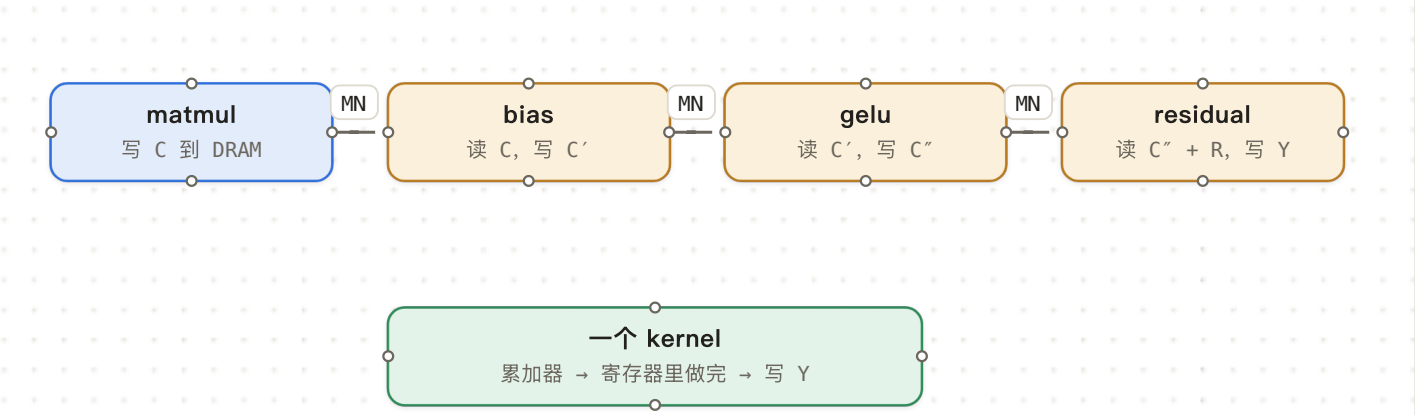
要用到的知识

把链切成若干段，每段一个 kernel：

- 一段读的字节 = 段内所有 stage 的**外部输入**（权重、bias、residual）+ 上一段的输出；写的字节 = 段内最后一个 stage 的输出。

- 段内的中间结果只在寄存器里，不进 DRAM。
- 一段的寄存器峰值 = 段内各 stage 的 `live_regs` 之和；超过 255 就会 spill，本题用“访存时间 × spill_penalty”粗略模拟。
- 每段一次 launch（约 5 μs）。n 个 stage 有 2^{n-1} 种切法。

GEMM + bias + GELU + residual：不融合 vs 融合



4096^2 的 fp16 例子：不融合时 C 在 DRAM 里往返三次，总流量 335,552,512 B，4 次 launch；全部融合后 134,225,920 B，1 次 launch，寄存器峰值从 128 升到 160。

> **类比 · 做菜的盘子**
>
> 不融合像每做完一步就把菜盛到盘子里端上桌（写 DRAM），下一步再从桌上端回来（读 DRAM）：切好的菜装盘、端回来炒、炒好装盘、端回来调味。融合就是在一口锅里连续做完再装一次盘。但锅的大小有限（寄存器）：如果每一步都要占很多锅边的位置，锅里放不下就只能临时往外盛（spill），反而更慢。

题目要求

在 `mine/k4_fusion.py` 里实现（`Stage`、`chain_mlp_epilogue`、`groups_of` 已给出）：

函数	返回
`traffic(stages, fuse, elem_bytes)`	`{"bytes", "launches", "peak_regs"}`
`time_estimate(stages, fuse, peak_flops, peak_bw, launch, elem_bytes, reg_limit, spill_penalty)`	每段 `max(访存, 计算) + launch` 之和
`best_plan(stages, peak_flops, peak_bw, **kw)`	枚举全部 2^{n-1} 种切法，返回 `(时间, 切法)` 的最小值

题干

mine/k4_fusion.py 54 行 复制

```python
"""K4 · Fusion: fewer launches and less DRAM traffic vs more register pressure — your impl

A chain of ops is a list of stages. Unfused, every stage is its own kernel: it reads its i
DRAM and writes its output back. Fused, a group of consecutive stages runs in one kernel:
group's external inputs are read and only its last output is written; the intermediates li
registers. That is the Triton view (one kernel, an epilogue) and the Luminal view (a graph
that merges nodes) of the same trade-off.
"""
from dataclasses import dataclass


@dataclass(frozen=True)
class Stage:
    name: str
    out_elems: int          # elements written by this stage
    ext_in_elems: int       # elements read from outside the chain (weights, bias, residua
    flops: int
    live_regs: int = 0      # registers per thread this stage keeps live inside a fused ke


def chain_mlp_epilogue(M, N, K):
    """GEMM followed by bias, GELU, residual add — the canonical fused epilogue."""
    return [
        Stage("matmul", M * N, M * K + K * N, 2 * M * N * K, live_regs=128),
        Stage("bias", M * N, N, M * N, live_regs=8),
        Stage("gelu", M * N, 0, 8 * M * N, live_regs=16),
        Stage("residual", M * N, M * N, M * N, live_regs=8),
    ]


def groups_of(stages, fuse):
    """fuse: list of group sizes summing to len(stages), e.g. [1, 1, 1, 1] (unfused) or [4
    assert sum(fuse) == len(stages) and all(g > 0 for g in fuse)
    out, i = [], 0
    for g in fuse:
        out.append(stages[i:i + g])
        i += g
    return out


def traffic(stages, fuse, elem_bytes=2):
    """DRAM bytes, launches, and peak live registers for a fusion plan."""
    raise NotImplementedError("traffic")


def time_estimate(stages, fuse, peak_flops, peak_bw, launch=5e-6, elem_bytes=2, reg_limit=
    """Roofline per group + launch overhead. If a fused group's live registers exceed reg_
    its memory time is multiplied by spill_penalty (local-memory spills)."""
    raise NotImplementedError("time_estimate")
```

```python
def best_plan(stages, peak_flops, peak_bw, **kw):
    """Try every way to cut the chain into consecutive groups (2^(n-1) plans)."""
    raise NotImplementedError("best_plan")
```

验收

```
./check.sh k4
```

tests/test_k4_fusion.py 51 行 复制

```
import pytest

from drill import load

k4 = load("k4_fusion")
S = 4096
PF, BW = 40e12, 448e9


def test_traffic_unfused_vs_fused():
    st = k4.chain_mlp_epilogue(S, S, S)
    mn = S * S
    un = k4.traffic(st, [1, 1, 1, 1])
    # matmul: A + B + C ; bias: C + b + C' ; gelu: 2 MN ; residual: 3 MN   (x 2 bytes)
    assert un["bytes"] == (2 * mn + mn + (mn + S + mn) + 2 * mn + 3 * mn) * 2
    assert un["launches"] == 4
    fused = k4.traffic(st, [4])
    assert fused["bytes"] == (2 * mn + S + mn + mn) * 2 and fused["launches"] == 1
    assert fused["peak_regs"] == 128 + 8 + 16 + 8
    assert k4.traffic(st, [1, 3])["bytes"] == k4.traffic(st, [2, 2])["bytes"]


def test_groups_validation():
    st = k4.chain_mlp_epilogue(8, 8, 8)
    with pytest.raises(AssertionError):
        k4.groups_of(st, [2, 1])


def test_best_plan_fuses_the_epilogue():
    t, plan = k4.best_plan(k4.chain_mlp_epilogue(S, S, S), PF, BW)
    assert plan == [4]
    t_un = k4.time_estimate(k4.chain_mlp_epilogue(S, S, S), [1, 1, 1, 1], PF, BW)
    assert t < t_un


def test_register_pressure_can_make_fusion_lose():
    n = 1 << 26
    heavy = [k4.Stage("a", n, n, n, 200), k4.Stage("b", n, 0, n, 100)]
    fused = k4.time_estimate(heavy, [2], PF, BW, spill_penalty=3.0)
    split = k4.time_estimate(heavy, [1, 1], PF, BW, spill_penalty=3.0)
    assert split < fused
    assert k4.best_plan(heavy, PF, BW, spill_penalty=3.0)[1] == [1, 1]
    light = [k4.Stage("a", n, n, n, 100), k4.Stage("b", n, 0, n, 100)]
    assert k4.best_plan(light, PF, BW, spill_penalty=3.0)[1] == [2]


def test_small_problem_is_launch_bound():
    st = k4.chain_mlp_epilogue(16, 64, 64)
    t_un = k4.time_estimate(st, [1, 1, 1, 1], PF, BW)
```

```
    t_f = k4.time_estimate(st, [4], PF, BW)
    assert t_un == pytest.approx(4 * 5e-6, rel=0.05) and t_f == pytest.approx(5e-6, rel=0.
```

提示

L1 traffic 的循环 方向（轻度剧透） 收起

维护 `prev_out`（初值 0）：每段 `reads = Σ ext_in_elems + prev_out`，`writes = grp[-1].out_elems`，然后 `prev_out = writes`。

L2 spill 拆解与常见坑（中度剧透） 收起

`regs = Σ live_regs`；`regs > reg_limit` 时 `t_mem *= spill_penalty`。

L3 枚举切法 几乎是答案（重度剧透） 收起

用一个 `n - 1` 位的掩码：第 i 位为 1 表示在 stage i 之后切一刀。把掩码翻译成段长列表，交给 `time_estimate`。

参考解

解 **solutions/k4_fusion.py** 先自己写完、跑过测试再看 收起 ▾

solutions/k4_fusion.py 83 行 复制

```python
"""K4 · Fusion: fewer launches and less DRAM traffic vs more register pressure — refer

A chain of ops is a list of stages. Unfused, every stage is its own kernel: it reads i
DRAM and writes its output back. Fused, a group of consecutive stages runs in one kern
group's external inputs are read and only its last output is written; the intermediate
registers. That is the Triton view (one kernel, an epilogue) and the Luminal view (a g
that merges nodes) of the same trade-off.
"""
from dataclasses import dataclass


@dataclass(frozen=True)
class Stage:
    name: str
    out_elems: int          # elements written by this stage
    ext_in_elems: int       # elements read from outside the chain (weights, bias, res
    flops: int
    live_regs: int = 0      # registers per thread this stage keeps live inside a fuse


def chain_mlp_epilogue(M, N, K):
    """GEMM followed by bias, GELU, residual add — the canonical fused epilogue."""
    return [
        Stage("matmul", M * N, M * K + K * N, 2 * M * N * K, live_regs=128),
        Stage("bias", M * N, N, M * N, live_regs=8),
        Stage("gelu", M * N, 0, 8 * M * N, live_regs=16),
        Stage("residual", M * N, M * N, M * N, live_regs=8),
    ]


def groups_of(stages, fuse):
    """fuse: list of group sizes summing to len(stages), e.g. [1, 1, 1, 1] (unfused) o
    assert sum(fuse) == len(stages) and all(g > 0 for g in fuse)
    out, i = [], 0
    for g in fuse:
        out.append(stages[i:i + g])
        i += g
    return out


def traffic(stages, fuse, elem_bytes=2):
    """DRAM bytes, launches, and peak live registers for a fusion plan."""
    total, regs, prev_out = 0, 0, 0
    grps = groups_of(stages, fuse)
    for grp in grps:
        reads = sum(s.ext_in_elems for s in grp) + prev_out    # + the previous group'
        total += (reads + grp[-1].out_elems) * elem_bytes
        prev_out = grp[-1].out_elems
        regs = max(regs, sum(s.live_regs for s in grp))
    return {"bytes": total, "launches": len(grps), "peak_regs": regs}
```

```
def time_estimate(stages, fuse, peak_flops, peak_bw, launch=5e-6, elem_bytes=2, reg_li
    """Roofline per group + launch overhead. If a fused group's live registers exceed
    its memory time is multiplied by spill_penalty (local-memory spills)."""
    t = 0.0
    prev_out = None
    for grp in groups_of(stages, fuse):
        reads = sum(s.ext_in_elems for s in grp) + (prev_out or 0)
        nbytes = (reads + grp[-1].out_elems) * elem_bytes
        regs = sum(s.live_regs for s in grp)
        t_mem = nbytes / peak_bw * (spill_penalty if regs > reg_limit else 1.0)
        t_cmp = sum(s.flops for s in grp) / peak_flops
        t += max(t_mem, t_cmp) + launch
        prev_out = grp[-1].out_elems
    return t


def best_plan(stages, peak_flops, peak_bw, **kw):
    """Try every way to cut the chain into consecutive groups (2^(n-1) plans)."""
    n = len(stages)
    plans = []
    for mask in range(1 << (n - 1)):
        sizes, cur = [], 1
        for i in range(n - 1):
            if mask >> i & 1:
                sizes.append(cur)
                cur = 1
            else:
                cur += 1
        sizes.append(cur)
        plans.append((time_estimate(stages, sizes, peak_flops, peak_bw, **kw), sizes))
    return min(plans)
```

在 RTX 3070 上对照

把 tutorial 03 的 matmul 改成两个版本：(a) epilogue 里做 `+ bias`、`gelu`、`+ residual`；(b) matmul 只写 C，再用三个 elementwise kernel。用 K2 的 harness 量 4096^2 和 16×4096 两个形状，并在 `ncu` 里看 DRAM bytes：大形状的差距接近流量之比，小形状的差距接近 launch 次数之比。

▼ 自测：什么时候不该融合?

当融合后的 kernel 寄存器超出上限而 spill，或者融合迫使一个 kernel 用不适合某一部分的 tile / layout 时（例如把一个需要整行数据的 softmax 塞进按 tile 切的 GEMM epilogue）。测试里的 `heavy` 链（live_regs 200 + 100）在 `spill_penalty=3` 时，拆成两个 kernel 反而更快。Luminal 不预设答案，而是把融合和不融合都放进搜索空间实测。

双 主 线

Track T · Triton（kernel compiler）

从 @triton.jit 到 PTX：IR 各层、layout 与 coalescing、autotune、最小复现、lit/FileCheck 测试。

双主线 · Track T · Triton（kernel compiler） · 第 5 题

T1 IR dump lab：同一个 kernel 的 TTIR / TTGIR / LLVM IR / PTX

同一个 kernel，在 TTIR、TTGIR、LLVM IR、PTX 四层里长什么样？哪些概念在哪一层出现、哪一层消失？这道题写一组读 IR 的小工具，用它们把 vector add 和 matmul 的四层 dump 逐层解剖，找出 layout encoding、layout conversion 和最终的 MMA 指令。

60 min | 难度 易 | 你写 `mine/t1_ir_anatomy.py` | 验收 `./check.sh t1`

参考解 `solutions/t1_ir_anatomy.py` | 📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 题目要求
5. 题干
6. 验收
7. 提示
8. 参考解
9. 在 RTX 3070 上：自己的 dump

背景

roadmap 的 T1：“实现 vector add、row reduction、matmul 三个 kernel；对 matmul dump TTIR、TTGIR、LLVM IR、PTX；标出每个 IR 阶段新增 / 消失的概念；找出 layout encoding 和 conversion；确认 dot 最终映射到哪些 MMA 指令。交付：一个可复现脚本 + IR 注释版。”

本题分两半：

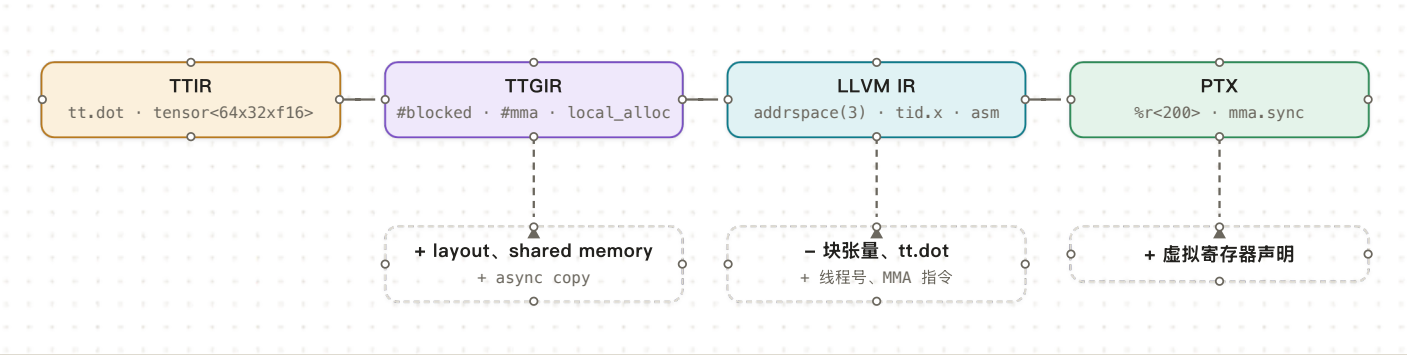
- **读**（本题的测试）：`fixtures/t1/` 里有按真实 Triton 输出格式整理的四层 dump（为了好读做了删减：省略了大部分地址计算，保留了每层的关键结构）。你写的函数要能识别阶段、统计 op、抽出 layout、找出 MMA。
- **dump**（在 GPU 上）：用 `CompiledKernel.asm` 拿到你自己 kernel 的真实四层文本，喂给同一组函数。

源码与资料

`triton › compiler.py: make_ttir / make_ttgir / make_llir / make_ptx` ·
`triton › CompiledKernel.asm` · `triton › accelerate-matmul.mlir (tt.dot → MMA layout)` ·
`triton › tritongpu_to_llvm.mlir (ldmatrix / mma.sync 的 CHECK)` · PTX ISA：mma.sync

要用到的知识

四层 IR 里，概念的出现与消失



TTIR 只描述“块级张量上的计算”；TTGIR 加上“谁拿哪个元素、数据放在哪层内存”；到 LLVM IR，块张量消失，变成每个线程的标量 / 向量代码，`tt.dot` 变成 inline asm 里的 `mma.sync`；PTX 再把它写成带虚拟寄存器的汇编。

几个要认识的东西

写法	在哪层	含义
`tt.get_program_id x`、`tt.make_range`、`tt.splat`、`tt.addptr`	TTIR / TTGIR	program instance 编号、`tl.arange`、广播、指针算术
`#blocked = #ttg.blocked<{sizePerThread = [...], ...}>`	TTGIR	线程到元素的映射（T2 会算它）
`#mma = #ttg.nvidia_mma<{versionMajor = 2, ...}>`	TTGIR	Ampere Tensor Core 累加器的 layout
`#ttg.dot_op<{opIdx = 0, parent = #mma, kWidth = 2}>`	TTGIR	`tl.dot` 的 A（opIdx 0）/ B（opIdx 1）操作数 layout
`ttg.local_alloc` / `ttg.async_copy_global_to_local` / `ttg.local_load`	TTGIR	shared memory 缓冲、`cp.async`、从 smem 读进寄存器
`ttg.convert_layout`	TTGIR	两个 layout 之间搬数据（T2 判断它的代价）
`addrspace(3)`、`@llvm.nvvm.read.ptx.sreg.tid.x`	LLVM IR	shared memory 地址空间、`threadIdx.x`
`mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32`	LLVM IR（asm）/ PTX	一个 warp 算 16×8×16 的 fp16 MMA，fp32 累加
`ldmatrix.sync.aligned.m8n8.x4[.trans].shared.b16`	PTX	从 smem 按 MMA 需要的分布读 4 个 8×8 矩阵

类比 · 同一份菜谱的四种语言

> TTIR 是**菜谱**：“把 A 和 B 搅拌在一起”。TTGIR 是**厨房分工表**：“1 号到 4 号厨师各搅一部分，先把食材从冷库搬到操作台”。LLVM IR 是**每个厨师的操作卡**：“我是 17 号，我搅第 3 行第 8–15 格”。PTX 是**把操作卡翻译成厨房设备的按钮序列**。读 IR 就是对照这四份文件，看哪一句话在哪一份里第一次出现。

题目要求

在 `mine/t1_ir_anatomy.py` 里实现（`CONCEPTS`、`_PTX_KINDS`、`_OP` 正则和 `dump_stages` 已给出）：

函数	返回
`detect_stage(text)`	`"ptx"`（有 `.version` 和 `.target sm_`）/ `"llir"`（有 `define` 和 `nvptx`）/ `"ttgir"`（有 `#ttg.` 或 `"ttg.num-warps"`）/ `"ttir"`
`mlir_ops(text)`	`Counter` 全名 op（`tt.load`、`scf.for`…），跳过注释、`#alias` 行和 `}`
`dialect_histogram(text)`	按方言汇总
`parse_attr_body(body)`	`"a = [1, 2], b = 3, c = #mma"` → dict
`layout_aliases(text)`	`{"#blocked": {"kind": "blocked", ...}, ...}`
`convert_layouts(text)`	`[(src_encoding, dst_encoding)]`
`ptx_instructions(text)`	每类（mma / ldmatrix / cp_async / barrier / global_vec）一个 `Counter`
`mma_shape(instr)`	`(16, 8, 16)`
`concepts(text)` / `concept_diff(stages)`	每层出现了哪些概念；相邻两层之间新增 / 消失了哪些

题干

mine/t1_ir_anatomy.py 106 行 复制

```
"""T1 · IR dump lab: one kernel, four representations — your implementation (run ./check.s

Triton compiles a @triton.jit function through a fixed list of stages (see
third_party/nvidia/backend/compiler.py: make_ttir -> make_ttgir -> make_llir -> make_ptx -
`kernel[grid](...)` returns a CompiledKernel whose `.asm` dict holds the text of each stag

    k = add_kernel[grid](x, y, out, n, BLOCK=1024)
    k.asm["ttir"], k.asm["ttgir"], k.asm["llir"], k.asm["ptx"]

This drill is the *reading* half: given those texts (fixtures/t1/*), tell which stage a du
count ops per dialect, pull out layout encodings and layout conversions, list the tensor-c
instructions in PTX, and diff which concepts appear / disappear between stages.
Runs anywhere — on a GPU box, `dump_stages()` produces the real texts.
"""
import re
from collections import Counter

# ---------------------------------------------------------------- stage detection
def detect_stage(text):
    """'ptx' | 'llir' | 'ttgir' | 'ttir' (checked in that order: lower stages are unambigu
    raise NotImplementedError("detect_stage")


# ---------------------------------------------------------------- MLIR ops
_OP = re.compile(r"^\s*(?:%[\w#:]+(?:\s*,\s*%[\w#:]+)*\s*=\s*)?\"?([a-z_]+\.[a-z_][\w.]*)\


def mlir_ops(text):
    """Counter of fully qualified op names ('tt.load', 'ttg.convert_layout', 'scf.for', ..
    One op per line in printed MLIR; attribute alias lines (#x = ...) and comments are ski
    raise NotImplementedError("mlir_ops")


def dialect_histogram(text):
    """Counter dialect -> number of ops ('tt', 'ttg', 'arith', 'scf', ...)."""
    raise NotImplementedError("dialect_histogram")


# ---------------------------------------------------------------- layouts
def parse_attr_body(body):
    """'sizePerThread = [1, 4], order = [1, 0], vec = 8, parent = #blocked' -> dict.
    Lists become lists of ints, ints become ints, anything else stays a string."""
    raise NotImplementedError("parse_attr_body")


def layout_aliases(text):
    """{'#blocked': {'kind': 'blocked', 'sizePerThread': [...], ...}, '#mma': {'kind': 'nv
    raise NotImplementedError("layout_aliases")
```

```python
def convert_layouts(text):
    """[(src_encoding, dst_encoding)] for every ttg.convert_layout."""
    raise NotImplementedError("convert_layouts")


# ---------------------------------------------------------------- PTX / LLVM
_PTX_KINDS = {
    "mma": r"\b((?:mma|wgmma\.mma_async)\.[\w.]+)",
    "ldmatrix": r"\b(ldmatrix\.[\w.]+)",
    "cp_async": r"\b(cp\.async\.[\w.]+|cp\.async\.bulk[\w.]*)",
    "barrier": r"\b(bar\.sync|barrier\.sync|mbarrier\.[\w.]+)",
    "global_vec": r"\b((?:ld|st)\.global(?:\.\w+)*\.v\d\.\w+)",
}


def ptx_instructions(text):
    """{'mma': Counter({'mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32': 2}), 'ldmatri
    Works on PTX and on LLVM IR (Triton emits these as inline asm strings there)."""
    raise NotImplementedError("ptx_instructions")


def mma_shape(instr):
    """'mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32' -> (16, 8, 16)."""
    raise NotImplementedError("mma_shape")


# ---------------------------------------------------------------- concepts per stage
CONCEPTS = {
    "program_id": r"tt\.get_program_id|ctaid",
    "block_tensor": r"tensor<",
    "pointer_tensor": r"tensor<[^>]*!tt\.ptr",
    "layout_encoding": r"#ttg\.(blocked|nvidia_mma|slice|dot_op)|, #\w+>",
    "shared_memory": r"ttg\.local_alloc|memdesc|addrspace\(3\)|\.shared",
    "async_copy": r"async_copy|cp\.async",
    "mma_instruction": r"mma\.sync|wgmma|ldmatrix",
    "thread_id": r"tid\.x|%tid",
    "virtual_registers": r"\.reg\s+\.",
    "high_level_dot": r"tt\.dot\b",
}


def concepts(text):
    raise NotImplementedError("concepts")


def concept_diff(stages):
    """stages: ordered list of (name, text). Returns [(from, to, appeared, disappeared)]."
    raise NotImplementedError("concept_diff")
```

```
# ---------------------------------------------------------------- the GPU half
def dump_stages(compiled_kernel):
    """On a CUDA machine: compiled = kernel[grid](...); dump_stages(compiled) -> {stage: t
    return {k: compiled_kernel.asm[k] for k in ("ttir", "ttgir", "llir", "ptx") if k in co
```

验收

```
./check.sh t1
```

tests/test_t1_ir_anatomy.py 83 行 复制

```
from pathlib import Path

from drill import load

t1 = load("t1_ir_anatomy")
FX = Path(__file__).resolve().parent.parent / "fixtures" / "t1"


def read(name):
    return (FX / name).read_text()


def test_detect_stage():
    assert t1.detect_stage(read("add.ttir.mlir")) == "ttir"
    assert t1.detect_stage(read("add.ttgir.mlir")) == "ttgir"
    assert t1.detect_stage(read("matmul.ttgir.mlir")) == "ttgir"
    assert t1.detect_stage(read("matmul.llir.ll")) == "llir"
    assert t1.detect_stage(read("matmul.ptx")) == "ptx"


def test_ops_and_dialects():
    ops = t1.mlir_ops(read("add.ttir.mlir"))
    assert ops["tt.load"] == 2 and ops["tt.store"] == 1 and ops["tt.splat"] == 5
    assert ops["tt.addptr"] == 3 and ops["arith.addf"] == 1 and ops["tt.return"] == 1
    assert ops["tt.func"] == 1
    d = t1.dialect_histogram(read("matmul.ttgir.mlir"))
    assert d["ttg"] >= 6 and d["scf"] == 2 and d["tt"] >= 3
    mm = t1.mlir_ops(read("matmul.ttgir.mlir"))
    assert mm["tt.dot"] == 1 and mm["ttg.local_load"] == 2 and mm["ttg.convert_layout"] ==


def test_same_ops_ttir_vs_ttgir():
    # adding layouts must not change which ops exist in this simple kernel
    assert t1.mlir_ops(read("add.ttir.mlir")) == t1.mlir_ops(read("add.ttgir.mlir"))


def test_layout_aliases():
    a = t1.layout_aliases(read("matmul.ttgir.mlir"))
    assert a["#blocked"] == {"kind": "blocked", "sizePerThread": [1, 8], "threadsPerWarp":
                             "warpsPerCTA": [4, 1], "order": [1, 0]}
    assert a["#mma"]["kind"] == "nvidia_mma" and a["#mma"]["versionMajor"] == 2
    assert a["#mma"]["instrShape"] == [16, 8] and a["#mma"]["warpsPerCTA"] == [2, 2]
    assert a["#shared"]["kind"] == "swizzled_shared" and a["#shared"]["vec"] == 8
    assert a["#smem"] == {"kind": "shared_memory"}
    assert t1.layout_aliases(read("add.ttir.mlir")) == {}


def test_parse_attr_body_nested():
    d = t1.parse_attr_body("opIdx = 0, parent = #mma, kWidth = 2")
    assert d == {"opIdx": 0, "parent": "#mma", "kWidth": 2}
```

```
def test_convert_layouts():
    cv = t1.convert_layouts(read("matmul.ttgir.mlir"))
    assert cv == [("#mma", "#blocked1")]
    assert t1.convert_layouts(read("add.ttgir.mlir")) == []


def test_ptx_instructions():
    p = t1.ptx_instructions(read("matmul.ptx"))
    mma = p["mma"]
    assert sum(mma.values()) == 2
    (name,) = mma
    assert name == "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32"
    assert t1.mma_shape(name) == (16, 8, 16)
    assert sum(p["ldmatrix"].values()) == 2 and "ldmatrix.sync.aligned.m8n8.x4.trans.share
    assert p["cp_async"]["cp.async.wait_group"] == 1 and p["barrier"]["bar.sync"] == 1
    assert p["global_vec"]["st.global.v4.b32"] == 1
    # the LLVM IR carries the same instructions as inline asm
    assert t1.ptx_instructions(read("matmul.llir.ll"))["mma"] == mma.__class__({name: 1})


def test_concept_diff_tells_the_story():
    stages = [(s, read(f)) for s, f in [("ttir", "matmul.ttir.mlir"), ("ttgir", "matmul.tt
                                        ("llir", "matmul.llir.ll"), ("ptx", "matmul.ptx")]
    diff = {(a, b): (app, gone) for a, b, app, gone in t1.concept_diff(stages)}
    app, gone = diff[("ttir", "ttgir")]
    assert "layout_encoding" in app and "shared_memory" in app and "async_copy" in app and
    app, gone = diff[("ttgir", "llir")]
    assert "mma_instruction" in app and "thread_id" in app
    assert {"block_tensor", "layout_encoding", "high_level_dot", "program_id"} <= set(gone
    app, gone = diff[("llir", "ptx")]
    assert "virtual_registers" in app
```

提示

L1 **一行一个 op** 方向（轻度剧透） 收起

MLIR 的文本形式一行一个 op：`%9 = tt.load %8, %6 : ...` 或 `tt.store ...`。给出的 `_OP` 正则已经处理了左边的 `%x =` 和多结果 `%res:2 =`，你只需要逐行 `match`，跳过以 `//`、`#`、`}` 开头的行。

L2 **layout 抽取** 拆解与常见坑（中度剧透） 收起 ▾

alias 行的形状固定：`^(#\w+)\s*=\s*#ttg\.(\w+)(?:<\{(.*)\}>)?\s*$`。`parse_attr_body` 用 `(\w+)\s*=\s*(\[[^\]]*\]|#[\w.]+|[^,]+)` 逐个抓键值，列表转成 int 列表。

L3 **convert_layout 和 concept_diff** 几乎是答案（重度剧透） 收起 ▾

`ttg.convert_layout %x : tensor<...> -> tensor<...>`：取两个 `tensor<…>`，encoding 是第一个逗号之后的部分（`_encoding_of`）。`concept_diff` 就是对相邻两层的 `concepts()` 集合做差：`sorted(b - a)`、`sorted(a - b)`。

参考解

解 solutions/t1_ir_anatomy.py 先自己写完、跑过测试再看 收起

solutions/t1_ir_anatomy.py 156 行 复制

```
"""T1 · IR dump lab: one kernel, four representations — reference solution.

Triton compiles a @triton.jit function through a fixed list of stages (see
third_party/nvidia/backend/compiler.py: make_ttir -> make_ttgir -> make_llir -> make_p
`kernel[grid](...)` returns a CompiledKernel whose `.asm` dict holds the text of each

    k = add_kernel[grid](x, y, out, n, BLOCK=1024)
    k.asm["ttir"], k.asm["ttgir"], k.asm["llir"], k.asm["ptx"]

This drill is the *reading* half: given those texts (fixtures/t1/*), tell which stage
count ops per dialect, pull out layout encodings and layout conversions, list the tens
instructions in PTX, and diff which concepts appear / disappear between stages.
Runs anywhere — on a GPU box, `dump_stages()` produces the real texts.
"""
import re
from collections import Counter

# ---------------------------------------------------------------- stage detection
def detect_stage(text):
    """'ptx' | 'llir' | 'ttgir' | 'ttir' (checked in that order: lower stages are unam
    if re.search(r"^\s*\.version\s+\d", text, re.M) and re.search(r"^\s*\.target\s+sm_
        return "ptx"
    if re.search(r"^\s*define\s", text, re.M) and "nvptx" in text:
        return "llir"
    if "#ttg." in text or '"ttg.num-warps"' in text:
        return "ttgir"
    if "tt.func" in text:
        return "ttir"
    raise ValueError("not a Triton stage dump")


# ---------------------------------------------------------------- MLIR ops
_OP = re.compile(r"^\s*(?:%[\w#:]+(?:\s*,\s*%[\w#:]+)*\s*=\s*)?\"?([a-z_]+\.[a-z_][\w.


def mlir_ops(text):
    """Counter of fully qualified op names ('tt.load', 'ttg.convert_layout', 'scf.for'
    One op per line in printed MLIR; attribute alias lines (#x = ...) and comments are
    c = Counter()
    for line in text.splitlines():
        s = line.strip()
        if not s or s.startswith(("//", "#", "}")):
            continue
        m = _OP.match(line)
        if m:
            c[m.group(1)] += 1
    return c


def dialect_histogram(text):
```

```
    """Counter dialect -> number of ops ('tt', 'ttg', 'arith', 'scf', ...)."""
    c = Counter()
    for op, n in mlir_ops(text).items():
        c[op.split(".")[0]] += n
    return c


# ---------------------------------------------------------------- layouts
def parse_attr_body(body):
    """'sizePerThread = [1, 4], order = [1, 0], vec = 8, parent = #blocked' -> dict.
    Lists become lists of ints, ints become ints, anything else stays a string."""
    out = {}
    for m in re.finditer(r"(\w+)\s*=\s*(\[[^\]]*\]|#[\w.]+(?:<\{.*?\}>)?|[^,]+)", body
        k, v = m.group(1), m.group(2).strip()
        if v.startswith("["):
            out[k] = [int(x) for x in v[1:-1].split(",") if x.strip()]
        elif re.fullmatch(r"-?\d+", v):
            out[k] = int(v)
        else:
            out[k] = v
    return out


def layout_aliases(text):
    """{'#blocked': {'kind': 'blocked', 'sizePerThread': [...], ...}, '#mma': {'kind':
    out = {}
    for m in re.finditer(r"^(#[\w]+)\s*=\s*#ttg\.(\w+)(?:<\{(.*)\}>)?\s*$", text, re.M
        d = {"kind": m.group(2)}
        if m.group(3):
            d.update(parse_attr_body(m.group(3)))
        out[m.group(1)] = d
    return out


def _encoding_of(tensor_type):
    """'tensor<64x64xf16, #mma>' -> '#mma'; nested '#ttg.dot_op<{opIdx = 0, parent = #
    inner = tensor_type[tensor_type.index("<") + 1: tensor_type.rindex(">")]
    parts = inner.split(",", 1)
    return parts[1].strip() if len(parts) > 1 else None


def convert_layouts(text):
    """[(src_encoding, dst_encoding)] for every ttg.convert_layout."""
    out = []
    for m in re.finditer(r"ttg\.convert_layout\s+%[\w#]+\s*:\s*(tensor<.*?>)\s*->\s*(t
        out.append((_encoding_of(m.group(1)), _encoding_of(m.group(2))))
    return out


# ---------------------------------------------------------------- PTX / LLVM
_PTX_KINDS = {
```

```
    "mma": r"\b((?:mma|wgmma\.mma_async)\.[\w.]+)",
    "ldmatrix": r"\b(ldmatrix\.[\w.]+)",
    "cp_async": r"\b(cp\.async\.[\w.]+|cp\.async\.bulk[\w.]*)",
    "barrier": r"\b(bar\.sync|barrier\.sync|mbarrier\.[\w.]+)",
    "global_vec": r"\b((?:ld|st)\.global(?:\.\w+)*\.v\d\.\w+)",
}


def ptx_instructions(text):
    """{'mma': Counter({'mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32': 2}), 'ldm
    Works on PTX and on LLVM IR (Triton emits these as inline asm strings there)."""
    out = {}
    for kind, pat in _PTX_KINDS.items():
        out[kind] = Counter(m.group(1).rstrip(".") for m in re.finditer(pat, text))
    return out


def mma_shape(instr):
    """'mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32' -> (16, 8, 16)."""
    m = re.search(r"\.m(\d+)n(\d+)k(\d+)", instr)
    return tuple(int(x) for x in m.groups()) if m else None


# ------------------------------------------------------------------ concepts per stage
CONCEPTS = {
    "program_id": r"tt\.get_program_id|ctaid",
    "block_tensor": r"tensor<",
    "pointer_tensor": r"tensor<[^>]*!tt\.ptr",
    "layout_encoding": r"#ttg\.(blocked|nvidia_mma|slice|dot_op)|, #\w+>",
    "shared_memory": r"ttg\.local_alloc|memdesc|addrspace\(3\)|\.shared",
    "async_copy": r"async_copy|cp\.async",
    "mma_instruction": r"mma\.sync|wgmma|ldmatrix",
    "thread_id": r"tid\.x|%tid",
    "virtual_registers": r"\.reg\s+\.",
    "high_level_dot": r"tt\.dot\b",
}


def concepts(text):
    return {k for k, pat in CONCEPTS.items() if re.search(pat, text)}


def concept_diff(stages):
    """stages: ordered list of (name, text). Returns [(from, to, appeared, disappeared
    out = []
    for (a, ta), (b, tb) in zip(stages, stages[1:]):
        ca, cb = concepts(ta), concepts(tb)
        out.append((a, b, sorted(cb - ca), sorted(ca - cb)))
    return out
```

```
# ---------------------------------------------------------------- the GPU half
def dump_stages(compiled_kernel):
    """On a CUDA machine: compiled = kernel[grid](...); dump_stages(compiled) -> {stag
    return {k: compiled_kernel.asm[k] for k in ("ttir", "ttgir", "llir", "ptx") if k i
```

在 RTX 3070 上：自己的 dump

```
import torch, triton, triton.language as tl
from solutions.t1_ir_anatomy import *

@triton.jit
def add_kernel(x, y, out, n, BLOCK: tl.constexpr):
    offs = tl.program_id(0) * BLOCK + tl.arange(0, BLOCK)
    m = offs < n
    tl.store(out + offs, tl.load(x + offs, mask=m) + tl.load(y + offs, mask=m), mask=m)

x = torch.randn(4096, device="cuda"); y = torch.randn_like(x); o = torch.empty_like(x)
k = add_kernel[(4,)](x, y, o, 4096, BLOCK=1024)
st = dump_stages(k)                          # {"ttir": ..., "ttgir": ..., "llir": ..., "ptx"
for a, b, app, gone in concept_diff(list(st.items())):
    print(f"{a} -> {b}: +{app} -{gone}")
print(layout_aliases(st["ttgir"]))
```

对 tutorial 03 的 matmul 做同样的事，把 `ptx_instructions(st["ptx"])["mma"]` 打印出来：在 SM86 上应该看到 `mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32`；把 `num_warps`、`BLOCK_SIZE_K` 改一改，数一数 `mma.sync` 的个数怎么变（每个 warp 每个 K 步需要 `(warp 的 M 块 / 16) × (warp 的 N 块 / 8) × (BK / 16)` 条）。想看每个 pass 之后的 IR，设 `MLIR_ENABLE_DUMP=1`；只想把各层文本存成文件，用 `TRITON_KERNEL_DUMP=1 TRITON_DUMP_DIR=./dump`。

▼ 自测：为什么 vector add 的 TTGIR 里没有 convert_layout，matmul 里却有?

vector add 从头到尾只有一种 layout（`#blocked`，每线程 4 个连续 fp32），load、add、store 都在它上面，不需要转换。matmul 的累加器必须是 Tensor Core 要求的 `#mma` layout，而最后的 `tt.store` 想要合并访存的 `#blocked1`，两者对“哪个线程持有哪个元素”的安排不同，所以在 store 之前要插一个 `ttg.convert_layout %9 : ...#mma -> ...#blocked1`。T2 会告诉你这次转换要不要经过 shared memory。

双主线 · Track T · Triton（kernel compiler） · 第 6 题

T2 Layout reasoning：复刻 coalesce pass，lane → 坐标，convert_layout 的代价

Triton 的 coalesce pass 为每个 load / store 挑一个 BlockedEncoding。这道题用 Python 逐行复刻它的两段真实代码，输出要和 coalesce.mlir 的 CHECK 逐字一致；然后用这个 layout 回答 T2 的三个问题：每个 lane 负责哪些坐标、load 是否合并访存、一次 convert_layout 是寄存器重命名、warp shuffle 还是要过 shared memory。

⏱ 75 min | 难度 难 | 你写 `mine/t2_coalesce_layout.py` | 验收 `./check.sh t2`

参考解 `solutions/t2_coalesce_layout.py` | 📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识：coalesce pass 的两步
4. 要用到的知识：lane 表、合并访存、convert_layout 的三档代价
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 在 RTX 3070 上：TTGIR 与 PTX 对照

背景

roadmap 的 T2：“选 2D blocked layout，改变 size-per-thread、threads-per-warp、warps-per-CTA；手算每个 lane 负责哪些坐标；分析 global load 是否 coalesced；制造一个 layout conversion，观察 shared memory / shuffle 是否出现。交付：lane-coordinate 表 + PTX / TTGIR 对照。”

与其凭感觉改参数，不如直接看编译器怎么选：`tritongpu-coalesce` 这个 pass 决定了绝大多数 load / store 的 layout，而它的核心只有两段几十行的代码。

源码与资料

- `triton › Utility.cpp: getNumElementsPerThread`：每个线程沿最快维拿几个元素
- `triton › CoalesceUtils.cpp: getEncoding`：再用 `numElems / numThreads` 封顶，构造 BlockedEncoding
- `triton › TritonGPUAttrDefs.td: BlockedEncodingAttr 的 builder`：从最快维开始分 lane、分 warp
- `triton › test/TritonGPU/coalesce.mlir`：本题的验收标准之一

- triton › Analysis/Utility.cpp：cvtReordersRegisters / cvtNeedsWarpShuffle / cvtNeedsSharedM
- Linear Layouts 论文：Triton 现在内部用线性布局统一表示所有 layout

要用到的知识：coalesce pass 的两步

第一步：每线程几个元素（perThread）

```
elem_bytes   = elem_bits / 8
max_multiple = divisibility_bytes / elem_bytes        # 指针对齐允许的最大向量
max_contig   = min(contiguity, shape[order[0]])       # axis analysis 证明的连续长度
perThread    = min(max_multiple, max_contig, 128 / elem_bits)   # 一条指令最多 128 bit
perThread    = min(perThread, max(numElems / numThreads, 1))    # 别让线程没活干
sizePerThread = [1, ..., perThread (在 order[0] 维), ..., 1]
```

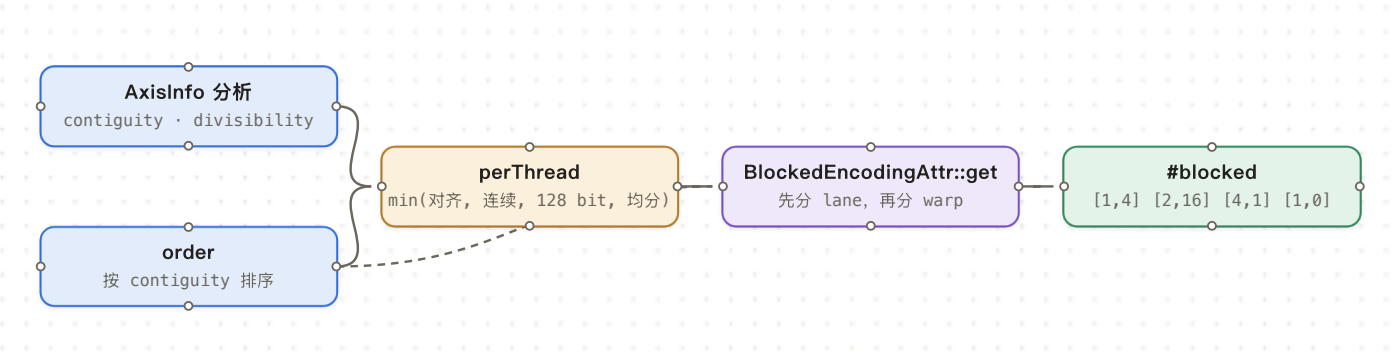
第二步：分 lane 和 warp（BlockedEncodingAttr builder）

```
rem_lanes, rem_threads, rem_warps = 32, num_warps*32, num_warps
for d in order[:-1]:                                   # 从最快维开始，最慢维除外
    threads_cta = clamp(rem_threads, 1, max(1, shape[d] / spt[d]))
    tpw[d]      = clamp(threads_cta, 1, rem_lanes)
    wpc[d]      = clamp(threads_cta / tpw[d], 1, rem_warps)
    rem_*      /= ...
tpw[order[-1]] = 32 / (前面各维 tpw 之积)               # 最慢维拿走剩下的 lane 和 warp
wpc[order[-1]] = num_warps / (前面各维 wpc 之积)
```

手算 coalesce.mlir 的 `@transpose`（64×64 fp32，`num-warps = 4`，`tt.divisibility = 16`）：

- 读：`order = [1, 0]`，`perThread = min(16/4, 64, 128/32, 4096/128) = 4` → `spt = [1, 4]`。dim 1：`threads_cta = clamp(128, 1, 64/4=16) = 16`，`tpw[1] = 16`，`wpc[1] = 1`；dim 0 拿剩下的：`tpw[0] = 32/16 = 2`，`wpc[0] = 4`。结果 `#ttg.blocked<{sizePerThread = [1, 4], threadsPerWarp = [2, 16], warpsPerCTA = [4, 1], order = [1, 0]}>`，**正是测试文件里的 `[[row_layout]]`**。
- 写（转置，`order = [0, 1]`）：同理得到 `sizePerThread = [4, 1], threadsPerWarp = [16, 2], warpsPerCTA = [1, 4]`，即 `[[col_layout]]`。

coalesce pass：从指针分析到 layout



输入只有四样：张量形状、元素位宽、指针沿最快维的 contiguity / divisibility、num_warps。所以同一个 kernel 换 dtype（fp32 → fp16）时，perThread 从 4 变成 8，layout 跟着变。

要用到的知识：lane 表、合并访存、convert_layout 的三档代价

- **lane 表**：线程 `(warp, lane)` 的第 r 个寄存器对应的坐标（和 gpu-kernel-drills G1 同一个公式）：`coord[d] = rep[d]·tile[d] + warp_c[d]·spt[d]·tpw[d] + lane_c[d]·spt[d] + s[d]`，再对 shape 取模（张量比 tile 小时会有多个线程持有同一元素）。
- **合并访存**：warp 0 的第一条向量 load，每个 lane 读 `vec = min(spt[order[0]], 16 / elem_bytes)` 个连续元素；数一下覆盖了多少个 32 B sector，效率 = 请求字节 / (sector 数 × 32)。
- **convert_layout 的三档**（Triton 用线性布局精确判断，本题用“元素归属”近似）：

档位	条件（本题）	Triton 里	代价
`registers`	每个元素的持有线程集合不变	`cvtReordersRegisters`	只是寄存器重命名，0 成本
`warp_shuffle`	每个元素仍在同一个 warp 里	`cvtNeedsWarpShuffle`（还要求置换足够简单）	若干条 `shfl.sync`
`shared_memory`	有元素要跨 warp	`cvtNeedsSharedMemory`	写 smem → barrier → 读 smem

类比 · 交接班

一次 layout 转换像两个班次交接工具。每个人手里的工具不变、只是换了个口袋（registers）——零成本；同一个小组内互相递一下（warp shuffle）——很快；工具要交给另一个小组的人（跨 warp），就得先放回公共工具房、等所有人都放好（barrier）、再各自去取（shared memory）——最慢。`remove_layout_conversions` 这个 pass 存在的意义，就是尽量别让工具进工具房。

题目要求

在 `mine/t2_coalesce_layout.py` 里实现（`Blocked`、`parse_blocked`、`clamp`、`_delinearize`、`tile` 已给出）：

函数	返回
`build_blocked(shape, spt, order, num_warps, tpw=32)`	复刻 builder
`elements_per_thread(shape, order, elem_bits, contiguity, divisibility_bytes, num_warps)`	复刻 getNumElementsPerThread + 均分上限
`coalesced_layout(shape, order, elem_bits, contiguity, divisibility_bytes=16, num_warps=4)`	上面两步合起来
`thread_elements(L, shape, warp, lane)`	按寄存器顺序的坐标列表
`owners(L, shape)`	`coord → {(warp, lane)}`
`lane_table(L, shape, warp=0)`	`[(lane, coords)]`
`first_load_sectors(L, shape, elem_bytes, strides=None)`	`(请求字节, sector 数, 效率)`
`classify_conversion(src, dst, shape)`	`"registers"` / `"warp_shuffle"` / `"shared_memory"`

题干

mine/t2_coalesce_layout.py 113 行 复制

```
"""T2 · Layout reasoning: re-derive Triton's coalesce pass, then map lanes to elements — y

Triton's `tritongpu-coalesce` pass picks a BlockedEncoding for every load/store. Two piece
real code decide it, and this drill reimplements both in Python:

  1. lib/Dialect/TritonGPU/Transforms/Utility.cpp  getNumElementsPerThread()
       perThread = min(divisibility_bytes / elem_bytes, contiguity, shape[order[0]], 128
     then CoalesceUtils.cpp also caps it by  max(numElems / numThreads, 1).
  2. include/triton/Dialect/TritonGPU/IR/TritonGPUAttrDefs.td  BlockedEncodingAttr builder
       walk dims from the fastest (order[0]) and hand out lanes, then warps; the slowest
       takes whatever is left.

With the layout in hand we answer the T2 questions: which lane owns which coordinates, is
first load of a warp coalesced, and does a ttg.convert_layout stay in registers, need warp
shuffles, or go through shared memory (lib/Analysis/Utility.cpp: cvtReordersRegisters /
cvtNeedsWarpShuffle / cvtNeedsSharedMemory).
"""
import re
from dataclasses import dataclass
from itertools import product
from math import prod


@dataclass(frozen=True)
class Blocked:
    size_per_thread: tuple
    threads_per_warp: tuple
    warps_per_cta: tuple
    order: tuple

    def __str__(self):
        f = lambda v: "[" + ", ".join(map(str, v)) + "]"
        return (f"#ttg.blocked<{{sizePerThread = {f(self.size_per_thread)}, threadsPerWarp
                f"{f(self.threads_per_warp)}, warpsPerCTA = {f(self.warps_per_cta)}, order


def parse_blocked(text):
    """Parse '#ttg.blocked<{sizePerThread = [1, 4], threadsPerWarp = [2, 16], ...}>'."""
    def grab(key):
        m = re.search(key + r"\s*=\s*\[([^\]]*)\]", text)
        if not m:
            raise ValueError(f"missing {key}")
        return tuple(int(x) for x in m.group(1).split(","))
    return Blocked(grab("sizePerThread"), grab("threadsPerWarp"), grab("warpsPerCTA"), gra


def clamp(x, lo, hi):
    return max(lo, min(x, hi))
```

```python
def build_blocked(shape, size_per_thread, order, num_warps, threads_per_warp=32):
    """Port of the BlockedEncodingAttr::get(shape, sizePerThread, order, numWarps, numThre
    raise NotImplementedError("build_blocked")


def elements_per_thread(shape, order, elem_bits, contiguity, divisibility_bytes, num_warps
    """getNumElementsPerThread + the numElems/numThreads cap from CoalesceUtils.cpp."""
    raise NotImplementedError("elements_per_thread")


def coalesced_layout(shape, order, elem_bits, contiguity, divisibility_bytes=16, num_warps
    """The layout the coalesce pass gives a load/store whose pointer has this contiguity a
    raise NotImplementedError("coalesced_layout")


# ---------------------------------------------------------------- who owns what
def _delinearize(idx, sizes, order):
    """Split a linear index into per-dim coordinates; order[0] varies fastest."""
    c = [0] * len(sizes)
    for d in order:
        c[d] = idx % sizes[d]
        idx //= sizes[d]
    return c


def tile(L: Blocked):
    return tuple(s * t * w for s, t, w in zip(L.size_per_thread, L.threads_per_warp, L.war


def thread_elements(L: Blocked, shape, warp, lane):
    """Coordinates held by (warp, lane), in register order (inside a chunk first, then rep
    both linearized along `order`)."""
    raise NotImplementedError("thread_elements")


def owners(L: Blocked, shape):
    """coord -> set of (warp, lane) holding it (more than one = broadcast / duplication)."
    raise NotImplementedError("owners")


def lane_table(L: Blocked, shape, warp=0):
    """[(lane, [coords...])] for one warp — the T2 deliverable."""
    raise NotImplementedError("lane_table")


# ---------------------------------------------------------------- coalescing
def first_load_sectors(L: Blocked, shape, elem_bytes, strides=None):
    """Simulate warp 0's first vectorized load instruction on a tensor with element stride
    `strides` (row-major contiguous by default). Each lane loads one vector of
    vec = min(spt[order[0]], 16 // elem_bytes) contiguous elements along order[0].
    Returns (bytes_requested, sectors_touched, efficiency = bytes / (sectors * 32))."""
```

```
    raise NotImplementedError("first_load_sectors")


# ---------------------------------------------------------------- convert_layout cost
def classify_conversion(src: Blocked, dst: Blocked, shape):
    """'registers'      every element stays in the same thread(s): only registers are rena
                         (Triton: cvtReordersRegisters)
       'warp_shuffle'   every element stays in the same warp: lanes exchange with shfl
                         (Triton: cvtNeedsWarpShuffle — the real check also limits the per
       'shared_memory'  some element must cross warps: write to smem, barrier, read back
                         (Triton: cvtNeedsSharedMemory)"""
    raise NotImplementedError("classify_conversion")
```

验收

```
./check.sh t2
```

tests/test_t2_coalesce_layout.py 89 行 复制

```
from drill import load

t2 = load("t2_coalesce_layout")
B = t2.Blocked


def test_parse_roundtrip():
    s = "#ttg.blocked<{sizePerThread = [1, 4], threadsPerWarp = [2, 16], warpsPerCTA = [4,
    L = t2.parse_blocked(s)
    assert L == B((1, 4), (2, 16), (4, 1), (1, 0))
    assert str(L) == s


def test_reproduces_coalesce_mlir_transpose():
    # test/TritonGPU/coalesce.mlir @transpose: 64x64 f32, num-warps 4, tt.divisibility = 1
    row = t2.coalesced_layout([64, 64], order=[1, 0], elem_bits=32, contiguity=64)
    col = t2.coalesced_layout([64, 64], order=[0, 1], elem_bits=32, contiguity=64)
    assert str(row) == ("#ttg.blocked<{sizePerThread = [1, 4], threadsPerWarp = [2, 16], "
                        "warpsPerCTA = [4, 1], order = [1, 0]}>")
    assert str(col) == ("#ttg.blocked<{sizePerThread = [4, 1], threadsPerWarp = [16, 2], "
                        "warpsPerCTA = [1, 4], order = [0, 1]}>")


def test_elements_per_thread_limits():
    # fp16: 128 bits -> 8 elements per thread
    assert t2.elements_per_thread([128, 64], [1, 0], 16, 64, 16, 4) == 8
    # alignment 8 bytes only -> 2 fp32
    assert t2.elements_per_thread([128, 64], [1, 0], 32, 64, 8, 4) == 2
    # contiguity 1 (gather-like pointer) -> 1
    assert t2.elements_per_thread([128, 64], [1, 0], 32, 1, 16, 4) == 1
    # tiny tensor: 64 elements over 128 threads -> 1
    assert t2.elements_per_thread([64], [0], 32, 64, 16, 4) == 1


def test_build_blocked_1d_and_small():
    assert t2.build_blocked([1024], [4], [0], 4) == B((4,), (32,), (4,), (0,))
    # 16 columns of fp16 with 8 per thread: only 2 lanes along dim 1, the rest go to rows
    L = t2.build_blocked([64, 16], [1, 8], [1, 0], 4)
    assert L == B((1, 8), (16, 2), (4, 1), (1, 0))


def test_thread_elements_and_lane_table():
    L = B((1, 4), (2, 16), (4, 1), (1, 0))
    assert t2.tile(L) == (8, 64)
    assert t2.thread_elements(L, [8, 64], 0, 0) == [(0, 0), (0, 1), (0, 2), (0, 3)]
    assert t2.thread_elements(L, [8, 64], 0, 17) == [(1, 4), (1, 5), (1, 6), (1, 7)]
    assert t2.thread_elements(L, [8, 64], 3, 16)[0] == (7, 0)
    # 64x64: the tile repeats 8 times along rows -> 32 registers per thread
    regs = t2.thread_elements(L, [64, 64], 0, 0)
    assert len(regs) == 32 and regs[4] == (8, 0)
```

```
    table = t2.lane_table(L, [8, 64])
    assert len(table) == 32 and table[16][1][0] == (1, 0)


def test_broadcast_when_tensor_smaller_than_tile():
    L = B((1, 4), (2, 16), (4, 1), (1, 0))
    own = t2.owners(L, [4, 64])          # tile is 8 rows, tensor has 4
    assert len(own) == 4 * 64
    assert len(own[(0, 0)]) == 2          # warp 0 lane 0 and warp 2 lane 0
    assert own[(0, 0)] == {(0, 0), (2, 0)}


def test_coalescing():
    row = B((1, 4), (2, 16), (4, 1), (1, 0))
    n, sectors, eff = t2.first_load_sectors(row, [64, 64], 4)
    assert n == 32 * 16 and sectors == 16 and eff == 1.0
    # the same row layout reading a column-major (transposed) tensor
    n, sectors, eff = t2.first_load_sectors(row, [64, 64], 4, strides=[1, 64])
    assert sectors == 64 and eff == 0.25          # rows 0 and 1 share a sector, 64 column
    # one element per lane, lanes along the slow dim -> 32 different rows
    bad = B((1, 1), (32, 1), (4, 1), (1, 0))
    assert t2.first_load_sectors(bad, [128, 64], 4)[1] == 32


def test_classify_conversion():
    shape = [64, 64]
    row = B((1, 4), (2, 16), (4, 1), (1, 0))
    assert t2.classify_conversion(row, row, shape) == "registers"
    # order also decides how lanes are numbered, so flipping it is NOT a register-only cha
    flipped = B((1, 4), (2, 16), (4, 1), (0, 1))
    assert t2.classify_conversion(row, flipped, shape) != "registers"
    # each warp keeps the same rows; only which lane holds which column changes
    narrow = B((1, 2), (2, 16), (4, 1), (1, 0))
    assert t2.classify_conversion(row, narrow, shape) == "warp_shuffle"
    # lanes stacked 16 tall: warp 0 now owns rows 0-15, which warps 0-3 shared before
    tall = B((1, 4), (16, 2), (4, 1), (1, 0))
    assert t2.classify_conversion(row, tall, shape) == "shared_memory"
    col = B((4, 1), (16, 2), (1, 4), (0, 1))
    assert t2.classify_conversion(row, col, shape) == "shared_memory"
```

提示

L1 先让 coalesce.mlir 的两个 layout 过 方向（轻度剧透） 收起

把上面的伪代码逐行翻译即可，注意所有除法都是整数除法，`clamp(x, lo, hi) = max(lo, min(x, hi))`。`elements_per_thread` 里 `max_multiple = max(divisibility_bytes // elem_bytes, 1)`。

L2 thread_elements 的循环顺序 拆解与常见坑（中度剧透） 收起

外层循环重复次数 `r in range(prod(reps))`，内层循环 chunk 内 `s in range(prod(spt))`，各自用 `_delinearize(·, sizes, L.order)` 拆坐标。lane 和 warp 也用 `_delinearize` 拆，分别按 `threads_per_warp` 和 `warps_per_cta`。

L3 sector 和分类 几乎是答案（重度剧透） 收起

`first_load_sectors`：地址 `= Σ coord[d]·stride[d]·elem_bytes`，默认行主序 `stride[d] = prod(shape[d+1:])`，sector `= addr // 32`。`classify_conversion`：先比 `owners(src)[c] == owners(dst)[c]`（全相等 → registers），再比 warp 集合 `{w for w, _ in ...}`（全相等 → warp_shuffle），否则 shared_memory。

参考解

解 solutions/t2_coalesce_layout.py 先自己写完、跑过测试再看 收起

solutions/t2_coalesce_layout.py 173 行 复制

```
"""T2 · Layout reasoning: re-derive Triton's coalesce pass, then map lanes to elements

Triton's `tritongpu-coalesce` pass picks a BlockedEncoding for every load/store. Two p
real code decide it, and this drill reimplements both in Python:

  1. lib/Dialect/TritonGPU/Transforms/Utility.cpp  getNumElementsPerThread()
       perThread = min(divisibility_bytes / elem_bytes, contiguity, shape[order[0]],
     then CoalesceUtils.cpp also caps it by  max(numElems / numThreads, 1).
  2. include/triton/Dialect/TritonGPU/IR/TritonGPUAttrDefs.td  BlockedEncodingAttr bui
       walk dims from the fastest (order[0]) and hand out lanes, then warps; the slow
       takes whatever is left.

With the layout in hand we answer the T2 questions: which lane owns which coordinates,
first load of a warp coalesced, and does a ttg.convert_layout stay in registers, need
shuffles, or go through shared memory (lib/Analysis/Utility.cpp: cvtReordersRegisters
cvtNeedsWarpShuffle / cvtNeedsSharedMemory).
"""
import re
from dataclasses import dataclass
from itertools import product
from math import prod


@dataclass(frozen=True)
class Blocked:
    size_per_thread: tuple
    threads_per_warp: tuple
    warps_per_cta: tuple
    order: tuple

    def __str__(self):
        f = lambda v: "[" + ", ".join(map(str, v)) + "]"
        return (f"#ttg.blocked<{{sizePerThread = {f(self.size_per_thread)}, threadsPer
                f"{f(self.threads_per_warp)}, warpsPerCTA = {f(self.warps_per_cta)}, o


def parse_blocked(text):
    """Parse '#ttg.blocked<{sizePerThread = [1, 4], threadsPerWarp = [2, 16], ...}>'."
    def grab(key):
        m = re.search(key + r"\s*=\s*\[([^\]]*)\]", text)
        if not m:
            raise ValueError(f"missing {key}")
        return tuple(int(x) for x in m.group(1).split(","))
    return Blocked(grab("sizePerThread"), grab("threadsPerWarp"), grab("warpsPerCTA"),


def clamp(x, lo, hi):
    return max(lo, min(x, hi))
```

```
def build_blocked(shape, size_per_thread, order, num_warps, threads_per_warp=32):
    """Port of the BlockedEncodingAttr::get(shape, sizePerThread, order, numWarps, num
    rank = len(size_per_thread)
    tpw, wpc = [0] * rank, [0] * rank
    rem_lanes, rem_threads, rem_warps = threads_per_warp, num_warps * threads_per_warp
    prev_lanes = prev_warps = 1
    for d in range(rank - 1):
        i = order[d]
        threads_cta = clamp(rem_threads, 1, max(1, shape[i] // size_per_thread[i]))
        tpw[i] = clamp(threads_cta, 1, rem_lanes)
        wpc[i] = clamp(threads_cta // tpw[i], 1, rem_warps)
        rem_warps //= wpc[i]
        rem_lanes //= tpw[i]
        rem_threads //= threads_cta
        prev_lanes *= tpw[i]
        prev_warps *= wpc[i]
    tpw[order[rank - 1]] = threads_per_warp // prev_lanes
    wpc[order[rank - 1]] = num_warps // prev_warps
    return Blocked(tuple(size_per_thread), tuple(tpw), tuple(wpc), tuple(order))


def elements_per_thread(shape, order, elem_bits, contiguity, divisibility_bytes, num_w
    """getNumElementsPerThread + the numElems/numThreads cap from CoalesceUtils.cpp."""
    elem_bytes = max(elem_bits // 8, 1)
    max_multiple = max(divisibility_bytes // elem_bytes, 1)
    max_contig = min(contiguity, shape[order[0]])
    alignment = min(max_multiple, max_contig)
    per_thread = min(alignment, 128 // elem_bits)
    return min(per_thread, max(prod(shape) // (num_warps * threads_per_warp), 1))


def coalesced_layout(shape, order, elem_bits, contiguity, divisibility_bytes=16, num_w
    """The layout the coalesce pass gives a load/store whose pointer has this contigui
    per = elements_per_thread(shape, order, elem_bits, contiguity, divisibility_bytes,
    spt = [1] * len(shape)
    spt[order[0]] = per
    return build_blocked(shape, spt, order, num_warps)


# ---------------------------------------------------------------- who owns what
def _delinearize(idx, sizes, order):
    """Split a linear index into per-dim coordinates; order[0] varies fastest."""
    c = [0] * len(sizes)
    for d in order:
        c[d] = idx % sizes[d]
        idx //= sizes[d]
    return c


def tile(L: Blocked):
    return tuple(s * t * w for s, t, w in zip(L.size_per_thread, L.threads_per_warp, L
```

```
def thread_elements(L: Blocked, shape, warp, lane):
    """Coordinates held by (warp, lane), in register order (inside a chunk first, then
    both linearized along `order`)."""
    t = tile(L)
    reps = [max(1, -(-s // td)) for s, td in zip(shape, t)]
    lc = _delinearize(lane, L.threads_per_warp, L.order)
    wc = _delinearize(warp, L.warps_per_cta, L.order)
    out = []
    for r in range(prod(reps)):
        rc = _delinearize(r, reps, L.order)
        for s in range(prod(L.size_per_thread)):
            sc = _delinearize(s, L.size_per_thread, L.order)
            coord = tuple((rc[d] * t[d] + wc[d] * L.size_per_thread[d] * L.threads_per
                           + lc[d] * L.size_per_thread[d] + sc[d]) % shape[d] for d in
            out.append(coord)
    return out


def owners(L: Blocked, shape):
    """coord -> set of (warp, lane) holding it (more than one = broadcast / duplicatio
    num_warps = prod(L.warps_per_cta)
    lanes = prod(L.threads_per_warp)
    own = {}
    for w in range(num_warps):
        for ln in range(lanes):
            for c in thread_elements(L, shape, w, ln):
                own.setdefault(c, set()).add((w, ln))
    return own


def lane_table(L: Blocked, shape, warp=0):
    """[(lane, [coords...])] for one warp — the T2 deliverable."""
    return [(ln, thread_elements(L, shape, warp, ln)) for ln in range(prod(L.threads_p


# ---------------------------------------------------------------- coalescing
def first_load_sectors(L: Blocked, shape, elem_bytes, strides=None):
    """Simulate warp 0's first vectorized load instruction on a tensor with element st
    `strides` (row-major contiguous by default). Each lane loads one vector of
    vec = min(spt[order[0]], 16 // elem_bytes) contiguous elements along order[0].
    Returns (bytes_requested, sectors_touched, efficiency = bytes / (sectors * 32))."
    if strides is None:
        strides = [prod(shape[d + 1:]) for d in range(len(shape))]
    vec = min(L.size_per_thread[L.order[0]], max(16 // elem_bytes, 1))
    sectors = set()
    nbytes = 0
    for ln in range(prod(L.threads_per_warp)):
        first = thread_elements(L, shape, 0, ln)[:vec]
        for c in first:
```

```
            addr = sum(ci * si for ci, si in zip(c, strides)) * elem_bytes
            sectors.add(addr // 32)
            nbytes += elem_bytes
    return nbytes, len(sectors), nbytes / (32 * len(sectors))


# ---------------------------------------------------------- convert_layout cost
def classify_conversion(src: Blocked, dst: Blocked, shape):
    """'registers'      every element stays in the same thread(s): only registers are
                         (Triton: cvtReordersRegisters)
       'warp_shuffle'   every element stays in the same warp: lanes exchange with shfl
                         (Triton: cvtNeedsWarpShuffle — the real check also limits the
       'shared_memory'  some element must cross warps: write to smem, barrier, read ba
                         (Triton: cvtNeedsSharedMemory)"""
    a, b = owners(src, shape), owners(dst, shape)
    if all(a[c] == b[c] for c in a):
        return "registers"
    warps = lambda s: {w for w, _ in s}
    if all(warps(a[c]) == warps(b[c]) for c in a):
        return "warp_shuffle"
    return "shared_memory"
```

在 RTX 3070 上：TTGIR 与 PTX 对照

1. 用 coalesce.mlir 本身做实验：`triton-opt test/TritonGPU/coalesce.mlir -split-input-file -tritongpu-coalesce`（源码 build 后在 `build/` 下有 `triton-opt`），把输出里的 `#blocked` 和 `coalesced_layout()` 对比；把 `%arg0` 的 `tt.divisibility` 改成 8，预测 perThread 会变成多少，再跑一次验证。
2. 写一个 64×64 的转置 kernel，看 `k.asm["ttgir"]` 里的 `ttg.convert_layout`，再在 `k.asm["ptx"]` 里找 `st.shared` / `ld.shared` / `bar.sync`：你的 `classify_conversion` 说 `shared_memory` 的那次转换，PTX 里应该正好对应一组 smem 写 + barrier + smem 读。
3. 把 `ld.global.v4.f32`（每线程 4 个 fp32 = 16 B）和 `first_load_sectors` 的效率对照：行布局读行主序张量时效率 1.0；拿同一个 layout 去读转置的张量，效率掉到 0.25。

▼ 自测：为什么 fp16 的 [128, 64] 张量 perThread 是 8 而不是 4?

一条全局访存指令最多 128 bit（`128 / elem_bits`）：fp32 是 4 个，fp16 是 8 个。只要指针 16 B 对齐（`divisibility = 16` → `16 / 2 = 8`）、连续长度至少 8、每个线程平均至少分到 8 个元素（8192 / 128 = 64），perThread 就是 8，对应 PTX 里的 `ld.global.v4.b32`（4 个 32 bit = 8 个 fp16）。

双主线 · Track T · Triton（kernel compiler） · 第 7 题

T3 GEMM tile sweep：prune → perf model → do_bench → 选最优

在 RTX 3070 上扫 BM / BN / BK / num_warps / num_stages，不能只“跑一遍取最快”。这道题按 triton.autotune 的真实流程写一个 autotuner：early_config_prune 剪掉跑不了的配置、perf_model + top_k 只留最有希望的几个、do_bench 式的分位数测量、在噪声范围内优先选资源更省的那个，并输出 CSV。

⏱ 45 min | 难度 中 | 你写 `mine/t3_autotune_sweep.py` | 验收 `./check.sh t3`

参考解 `solutions/t3_autotune_sweep.py` | 📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 题目要求
5. 题干
6. 验收
7. 提示
8. 参考解
9. 在 RTX 3070 上：真实的 sweep 和 CSV

背景

roadmap 的 T3：“扫 BM / BN / BK、num_warps、num_stages（仅选硬件和版本支持的配置）；记录 latency、registers、shared memory、occupancy、spill；区分 memory-bound、compute-bound 和 launch-bound；不要只报告最快一次，报告 median 和稳定性。交付：benchmark CSV + 性能分析。”

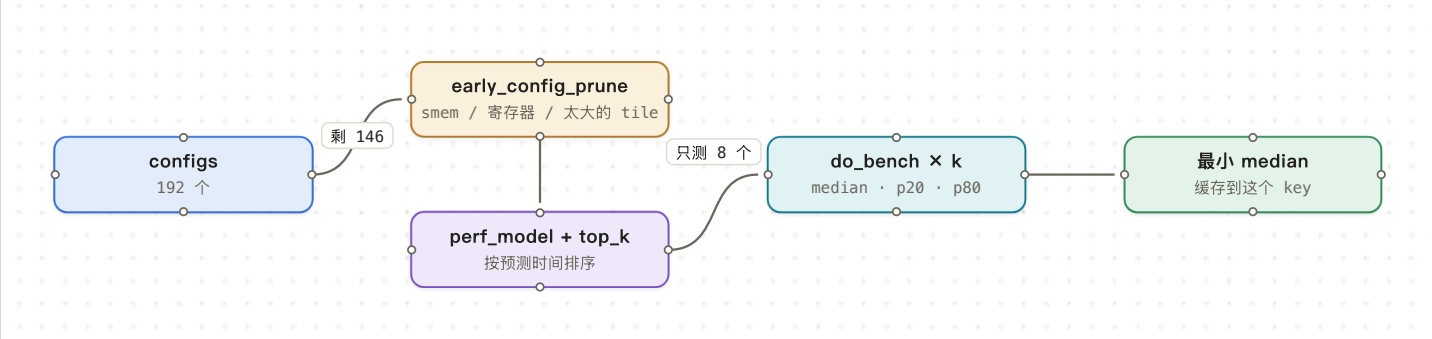
`triton.autotune` 本身就是这个流程的实现。读懂它，再自己写一遍，你就知道它省掉了哪些测量、可能错过什么。

源码与资料

`triton › autotuner.py: prune_configs_by 的三个键` · `triton › Autotuner.prune_configs` · `triton › _bench: quantiles=(0.5, 0.2, 0.8)` · `triton › testing.do_bench` · `triton › tutorial 03 的配置列表`

要用到的知识

triton.autotune 对一个新 key 做的事



key 是 `@autotune(key=[...])` 里列出的参数（通常是 M、N、K）。形状一变就重新调一次，所以生产代码要么限制 key 的取值，要么像 Luminal 一样把调优挪到编译期。

本题的 perf model（K1 的资源模型 + wave 量化）：

```
blocks/SM = occupancy(gpu, cfg).blocks
per_wave  = 46 · blocks/SM
waves     = ⌈tiles / per_wave⌉
t_compute = waves · per_wave · (2·BM·BN·K) / peak      # 最后一轮半空也按满算
t_memory  = tiled_bytes / bandwidth
t         = max(t_compute, t_memory) + launch
```

选最优：先找 median 最小的；所有 [p20, p80] 区间和它重叠的配置都算“噪声内一样快”，其中选 shared memory 最少的（给 occupancy 留余量，换 shape 时更稳）。

> **类比 · 招聘**
>
> 192 份简历（configs）先过硬性条件（early_config_prune：学历、签证——跑不了的直接淘汰），再按简历打分（perf_model）挑出前 8 名面试（do_bench）。面试官给每人打几轮分，看中位数；两位候选人分数区间重叠时，选要价更低的那个（shared memory 更少）。简历打分可能看走眼（perf model 不准），所以 top_k 不能太小。

题目要求

在 `mine/t3_autotune_sweep.py` 里实现（`config_space`、`fake_measure`、`to_csv` 已给出；K1 的资源函数从 `solutions.k1_gemm_model` 导入）：

函数	要点
`early_prune(configs, gpu, M, N, K)`	去掉 `occupancy` 无效的；`BM > max(32, 2M)`、`BN > max(32, 2N)`、`BK > max(16, K)` 的也去掉
`perf_model(gpu, M, N, K, cfg)`	上面的公式
`top_k(configs, model, k)`	按预测时间稳定排序取前 k
`quantiles(samples, qs)`	线性插值，默认 `(0.5, 0.2, 0.8)`
`sweep(configs, measure, repeats)`	每行：五个参数 + median / p20 / p80 / cv / smem
`pick_best(rows)`	噪声内选 smem 最少
`bound(gpu, M, N, K, cfg)`	K1 roofline 的 bound
`autotune(gpu, M, N, K, measure, configs, k, repeats)`	`(best_row, rows, n_pruned)`

题干

mine/t3_autotune_sweep.py 86 行 复制

```
"""T3 · GEMM tile / resource sweep, the way triton.autotune does it — your implementation

python/triton/runtime/autotuner.py runs, per new key:
    configs -> early_config_prune(configs, nargs)       # drop configs that cannot work
            -> perf_model + top_k                        # keep only the k most promising
            -> do_bench(kernel_call, quantiles=(0.5, 0.2, 0.8))  # measure the survivors
            -> pick the smallest median
We build the same pipeline in Python. The pruning and the perf model reuse K1's resource m
(solutions.k1_gemm_model, given), and `measure` is injectable: on the RTX 3070 it launches
real Triton kernel, in tests it is a seeded fake.
"""
import csv
import io
import random
import statistics
from itertools import product
from math import ceil

from solutions.k1_gemm_model import RTX3070, TileConfig, occupancy, roofline, smem_bytes,


def config_space(BM=(32, 64, 128, 256), BN=(32, 64, 128, 256), BK=(32, 64), num_warps=(4,
    return [TileConfig(*c) for c in product(BM, BN, BK, num_warps, num_stages)]


def early_prune(configs, gpu, M, N, K):
    """Drop configs that cannot launch (K1 occupancy invalid) or whose tile is pointlessly
    than the problem (BM > 2 * M rounds a 100-row problem up to a 256-row tile)."""
    raise NotImplementedError("early_prune")


def perf_model(gpu, M, N, K, c, in_bytes=2):
    """Predicted seconds = max(compute with wave quantization, DRAM traffic) + launch over

    Compute: tiles run in waves of sm_count * blocks_per_sm CTAs; a partly empty last wave
    still costs a full wave, so time = waves * per_wave * tile_flops / peak."""
    raise NotImplementedError("perf_model")


def top_k(configs, model, k):
    """The k configs with the smallest predicted time (stable on ties)."""
    raise NotImplementedError("top_k")


def quantiles(samples, qs=(0.5, 0.2, 0.8)):
    raise NotImplementedError("quantiles")


def sweep(configs, measure, repeats=25):
    """Measure each config `repeats` times. Rows: cfg fields + median/p20/p80/cv."""
```

```
    raise NotImplementedError("sweep")


def pick_best(rows):
    """Smallest median; if another row's interval [p20, p80] overlaps the winner's, prefer
    one with less shared memory (more occupancy headroom, same speed within noise)."""
    raise NotImplementedError("pick_best")


def bound(gpu, M, N, K, c):
    """'memory' | 'compute' | 'launch' from the K1 roofline of this config's real traffic.
    raise NotImplementedError("bound")


def to_csv(rows):
    buf = io.StringIO()
    w = csv.DictWriter(buf, fieldnames=list(rows[0]))
    w.writeheader()
    w.writerows(rows)
    return buf.getvalue()


def autotune(gpu, M, N, K, measure, configs=None, k=8, repeats=25):
    """The whole autotuner: prune -> perf_model top-k -> sweep -> pick. Returns (best_row,
    raise NotImplementedError("autotune")


def fake_measure(gpu, M, N, K, seed=0, noise=0.03):
    """A seeded stand-in for a real launch: the perf model times a config-specific efficie
    factor (what the model does not know) times multiplicative noise."""
    rng = random.Random(seed)

    def m(c):
        eff = 1.0 + 0.15 * ((hash((c.BM, c.BN, c.BK, c.num_warps, c.num_stages)) % 7) / 7)
        return perf_model(gpu, M, N, K, c) * eff * (1 + rng.gauss(0, noise))
    return m
```

验收

```
./check.sh t3
```

tests/test_t3_autotune_sweep.py 71 行 复制

```
import pytest

from drill import load
from solutions.k1_gemm_model import RTX3070 as G, TileConfig as C, occupancy

t3 = load("t3_autotune_sweep")
S = 4096


def test_space_and_prune():
    cs = t3.config_space()
    assert len(cs) == 4 * 4 * 2 * 2 * 3
    alive = t3.early_prune(cs, G, S, S, S)
    assert len(alive) == 146 and all(occupancy(G, c)["valid"] for c in alive)
    skinny = t3.early_prune(cs, G, 100, S, S)
    assert max(c.BM for c in skinny) == 128          # 256-row tiles are pruned for M = 10


def test_perf_model_wave_quantization():
    c = C(128, 128, 32, 4, 3)                          # 2 CTAs / SM (smem-limited) -> 92
    assert occupancy(G, c)["blocks"] == 2
    t = t3.perf_model(G, S, S, S, c)
    assert t == pytest.approx(0.004873388571428571)
    # the same tile on 92 tiles (exactly one wave) vs 93 tiles (a second, almost empty wav
    one = t3.perf_model(G, 128 * 92, 128, 4096, c)
    two = t3.perf_model(G, 128 * 93, 128, 4096, c)
    assert two > one


def test_top_k_is_sorted_by_model():
    alive = t3.early_prune(t3.config_space(), G, S, S, S)
    m = lambda c: t3.perf_model(G, S, S, S, c)
    k = t3.top_k(alive, m, 5)
    assert len(k) == 5 and [m(c) for c in k] == sorted(m(c) for c in k)
    assert all(m(k[-1]) <= m(c) for c in alive if c not in k)


def test_quantiles_match_do_bench_defaults():
    med, p20, p80 = t3.quantiles(list(range(11)))
    assert (med, p20, p80) == (5, 2, 8)


def test_sweep_pick_and_csv():
    m = t3.fake_measure(G, S, S, S, seed=0)
    cands = [C(128, 256, 32, 8, 2), C(128, 128, 32, 4, 3), C(64, 64, 32, 4, 2)]
    rows = t3.sweep(cands, m, repeats=15)
    assert len(rows) == 3 and all(r["p20"] <= r["median"] <= r["p80"] for r in rows)
    best = t3.pick_best(rows)
    assert best["median"] <= min(r["median"] for r in rows) * 1.1
    text = t3.to_csv(rows)
```

```
    assert text.splitlines()[0].startswith("BM,BN,BK,num_warps,num_stages,median")
    assert len(text.strip().splitlines()) == 4


def test_pick_best_prefers_less_smem_within_noise():
    rows = [{"median": 1.00, "p20": 0.95, "p80": 1.05, "smem": 98304},
            {"median": 1.02, "p20": 0.97, "p80": 1.07, "smem": 32768},
            {"median": 1.50, "p20": 1.45, "p80": 1.55, "smem": 8192}]
    assert t3.pick_best(rows)["smem"] == 32768


def test_autotune_end_to_end():
    best, rows, pruned = t3.autotune(G, S, S, S, t3.fake_measure(G, S, S, S, seed=0), k=8)
    assert pruned == 46 and len(rows) == 8
    assert (best["BM"], best["BN"]) in {(128, 256), (256, 128)}


def test_bound():
    assert t3.bound(G, 64, 64, 64, C(32, 32, 32)) == "launch"
    # no L2 in the model: a 4096^3 fp16 GEMM with 128x128 tiles re-reads 2 GB -> "memory"
    assert t3.bound(G, S, S, S, C(128, 128, 32)) == "memory"
```

提示

L1 **剪枝** 方向（轻度剧透） 收起

`occupancy(gpu, c)["valid"]` 一行搞定资源检查。192 个配置里有 46 个因为 shared memory 超限或估计寄存器超过 255 被剪掉。

L2 **wave 量化** 拆解与常见坑（中度剧透） 收起

128×128 的 tile、每 SM 2 个 CTA 时每轮 92 个；`M = 128·92` 是正好一轮，`M = 128·93` 要两轮，时间几乎翻倍——测试用它检查你有没有把最后一轮按满算。

L3 **pick_best** 几乎是答案（重度剧透） 收起

`best = min(rows, key=median)`；`ties = [r for r in rows if r.p20 <= best.p80 and best.p20 <= r.p80]`；返回 `min(ties, key=(smem, median))`。

参考解

解 solutions/t3_autotune_sweep.py 先自己写完、跑过测试再看 收起 ▾

solutions/t3_autotune_sweep.py 119 行 复制

```
"""T3 · GEMM tile / resource sweep, the way triton.autotune does it — reference soluti

python/triton/runtime/autotuner.py runs, per new key:
    configs -> early_config_prune(configs, nargs)       # drop configs that cannot wor
            -> perf_model + top_k                        # keep only the k most promis
            -> do_bench(kernel_call, quantiles=(0.5, 0.2, 0.8))  # measure the survivo
            -> pick the smallest median
We build the same pipeline in Python. The pruning and the perf model reuse K1's resour
(solutions.k1_gemm_model, given), and `measure` is injectable: on the RTX 3070 it laun
real Triton kernel, in tests it is a seeded fake.
"""
import csv
import io
import random
import statistics
from itertools import product
from math import ceil

from solutions.k1_gemm_model import RTX3070, TileConfig, occupancy, roofline, smem_byt


def config_space(BM=(32, 64, 128, 256), BN=(32, 64, 128, 256), BK=(32, 64), num_warps=
    return [TileConfig(*c) for c in product(BM, BN, BK, num_warps, num_stages)]


def early_prune(configs, gpu, M, N, K):
    """Drop configs that cannot launch (K1 occupancy invalid) or whose tile is pointle
    than the problem (BM > 2 * M rounds a 100-row problem up to a 256-row tile)."""
    keep = []
    for c in configs:
        if not occupancy(gpu, c)["valid"]:
            continue
        if c.BM > max(32, 2 * M) or c.BN > max(32, 2 * N) or c.BK > max(16, K):
            continue
        keep.append(c)
    return keep


def perf_model(gpu, M, N, K, c, in_bytes=2):
    """Predicted seconds = max(compute with wave quantization, DRAM traffic) + launch

    Compute: tiles run in waves of sm_count * blocks_per_sm CTAs; a partly empty last
    still costs a full wave, so time = waves * per_wave * tile_flops / peak."""
    occ = occupancy(gpu, c, in_bytes)
    tiles = ceil(M / c.BM) * ceil(N / c.BN)
    per_wave = gpu.sm_count * occ["blocks"]
    waves = ceil(tiles / per_wave)
    t_compute = waves * per_wave * (2 * c.BM * c.BN * K) / gpu.peak_flops
    t_memory = tiled_bytes(M, N, K, c, in_bytes) / gpu.peak_bw
    return max(t_compute, t_memory) + gpu.launch_overhead
```

```
def top_k(configs, model, k):
    """The k configs with the smallest predicted time (stable on ties)."""
    return sorted(configs, key=model)[:k]


def quantiles(samples, qs=(0.5, 0.2, 0.8)):
    s = sorted(samples)
    out = []
    for q in qs:
        pos = (len(s) - 1) * q
        lo = int(pos)
        hi = min(lo + 1, len(s) - 1)
        out.append(s[lo] + (s[hi] - s[lo]) * (pos - lo))
    return out


def sweep(configs, measure, repeats=25):
    """Measure each config `repeats` times. Rows: cfg fields + median/p20/p80/cv."""
    rows = []
    for c in configs:
        xs = [measure(c) for _ in range(repeats)]
        med, p20, p80 = quantiles(xs)
        mean = statistics.fmean(xs)
        rows.append({"BM": c.BM, "BN": c.BN, "BK": c.BK, "num_warps": c.num_warps, "nu
                     "median": med, "p20": p20, "p80": p80, "cv": statistics.pstdev(xs
    return rows


def pick_best(rows):
    """Smallest median; if another row's interval [p20, p80] overlaps the winner's, pr
    one with less shared memory (more occupancy headroom, same speed within noise)."""
    best = min(rows, key=lambda r: r["median"])
    ties = [r for r in rows if r["p20"] <= best["p80"] and best["p20"] <= r["p80"]]
    return min(ties, key=lambda r: (r["smem"], r["median"]))


def bound(gpu, M, N, K, c):
    """'memory' | 'compute' | 'launch' from the K1 roofline of this config's real traf
    return roofline(gpu, gemm_flops(M, N, K), tiled_bytes(M, N, K, c))["bound"]


def to_csv(rows):
    buf = io.StringIO()
    w = csv.DictWriter(buf, fieldnames=list(rows[0]))
    w.writeheader()
    w.writerows(rows)
    return buf.getvalue()
```

```
def autotune(gpu, M, N, K, measure, configs=None, k=8, repeats=25):
    """The whole autotuner: prune -> perf_model top-k -> sweep -> pick. Returns (best_
    configs = configs or config_space()
    alive = early_prune(configs, gpu, M, N, K)
    cand = top_k(alive, lambda c: perf_model(gpu, M, N, K, c), k)
    rows = sweep(cand, measure, repeats)
    return pick_best(rows), rows, len(configs) - len(alive)


def fake_measure(gpu, M, N, K, seed=0, noise=0.03):
    """A seeded stand-in for a real launch: the perf model times a config-specific eff
    factor (what the model does not know) times multiplicative noise."""
    rng = random.Random(seed)

    def m(c):
        eff = 1.0 + 0.15 * ((hash((c.BM, c.BN, c.BK, c.num_warps, c.num_stages)) % 7)
        return perf_model(gpu, M, N, K, c) * eff * (1 + rng.gauss(0, noise))
    return m
```

在 RTX 3070 上：真实的 sweep 和 CSV

```
import torch, triton
from solutions.t3_autotune_sweep import *
from solutions.k1_gemm_model import RTX3070 as G
# matmul_kernel：tutorial 03 去掉 @autotune 的版本
def real_measure(c, M=4096, N=4096, K=4096):
    a = torch.randn(M, K, device="cuda", dtype=torch.float16); b = torch.randn(K, N, devic
    out = torch.empty(M, N, device="cuda", dtype=torch.float16)
    grid = (triton.cdiv(M, c.BM) * triton.cdiv(N, c.BN),)
    run = lambda: matmul_kernel[grid](a, b, out, M, N, K, a.stride(0), a.stride(1), b.stri
                                      out.stride(0), out.stride(1), BLOCK_SIZE_M=c.BM, BLO
                                      BLOCK_SIZE_K=c.BK, GROUP_SIZE_M=8, num_warps=c.num_w
    return triton.testing.do_bench(run, warmup=10, rep=50) * 1e-3     # ms -> s
best, rows, pruned = autotune(G, 4096, 4096, 4096, real_measure, k=12, repeats=5)
open("t3_sweep.csv", "w").write(to_csv(rows))
```

在 CSV 里再加三列：`k.n_regs`、`k.n_spills`、`k.metadata.shared`（来自编译出的 kernel），以及 `ncu` 的 achieved occupancy。然后写性能分析：哪些配置的 `n_regs` 和 K1 的估计差得最多？有 spill 的配置慢了多少？`perf_model` 的排名和实测排名的 Spearman 相关系数是多少（L3 里有现成的 `spearman`）？

▼ 自测：为什么 top_k 选得太小有风险?

perf model 只看 FLOPs、字节和 occupancy，看不见 bank conflict、指令调度、L2 命中率、`num_stages` 对延迟隐藏的真实效果。测试里的 `fake_measure` 给每个配置乘了一个模型不知道的效率因子，正是在模拟这一点。如果 top_k = 1，你就完全信任了模型；top_k 太大，又失去了剪枝的意义。实践中常见做法是：top_k 取 5—10，并定期用全量 sweep 校验模型。

双主线 · Track T · Triton（kernel compiler） · 第 8 题

T4 Correctness bug minimization：fuzz → 失败 → 缩小 → 回归测试

维护者最欢迎的 issue 是一个“十行能复现”的最小例子。这道题写出产出它的完整流水线：固定 seed 的随机差分测试 → 保留第一个失败输入 → 把形状、dtype、stride 缩到最小 → 自动生成可以直接提交的 pytest 回归测试。顺带实现 delta debugging，用来缩小 IR 和图。

45 min | 难度 中 | 你写 `mine/t4_minimize.py` | 验收 `./check.sh t4`

参考解 `solutions/t4_minimize.py` | 📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 题目要求
5. 题干
6. 验收
7. 提示
8. 参考解
9. 在 RTX 3070 上：换成真的 Triton kernel

背景

roadmap 的 T4：“对一个 op / layout / dtype 构造随机差分测试，覆盖非整除 shape、mask、FP16 / BF16 边界和非连续 stride；写 PyTorch reference；固定 seed，保留失败输入；缩小到最小 shape / dtype / layout reproducer。交付：可直接转成 pytest / lit regression 的最小案例。”

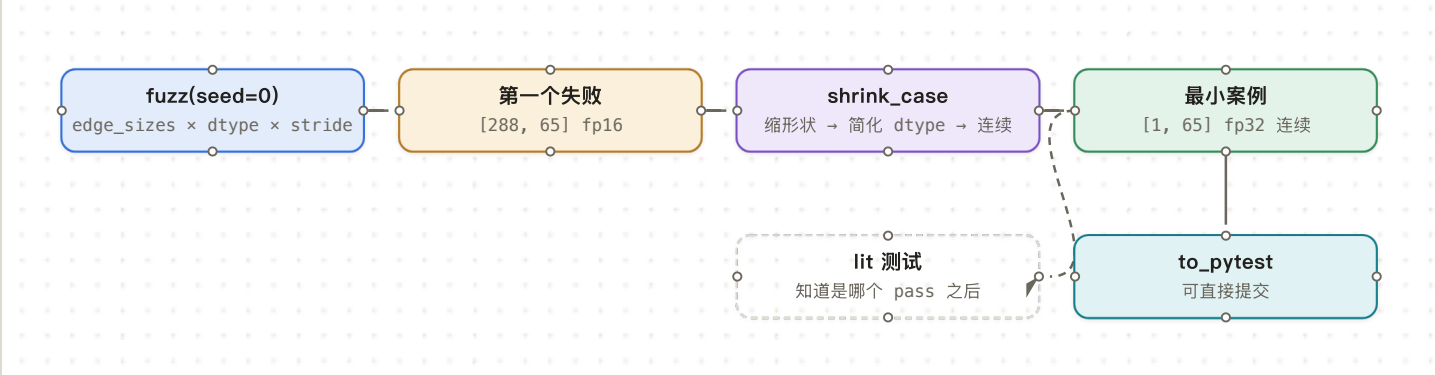
被测对象 `buggy_row_sum` 是一个“分块求和 kernel”，带着一个经典 bug：单块路径做了 mask，多块路径只走完整的块、忘了尾巴。随机测试能撞到它，但撞到的输入（比如 `[129, 255]`、bf16、转置 stride）离“最小”很远。

源码与资料

`triton › python/test/unit/language/test_core.py（Triton 的 Python 回归测试大本营）` · `triton › Triton CONTRIBUTING.md` · Zeller：Simplifying and Isolating Failure-Inducing Input（ddmin） · Hypothesis 的 shrinking

要用到的知识

从随机失败到回归测试



65 = BLOCK + 1：最小的“需要两块、第二块不满”的长度。看到这个数字，维护者立刻就知道该去查 tail 的 mask。

两种缩小

- **结构化**（`shrink_case`）：对每一维用 `shrink_int` 找“仍然失败的最小值”（先试减半，再减 1）；再按 `float32 → float16 → bfloat16` 尝试更简单的 dtype；再试连续 stride。直到一整轮都没有变化（fixpoint）。
- **列表**（`ddmin`）：输入是一串东西（IR 的行、图的节点、一组 rewrite 规则），把它切成 n 块，试“只保留某一块”或“去掉某一块”，能缩就缩、不能缩就切得更细。结果是 **1-minimal**：再删掉任何一个元素就不再失败。

> **类比 · 找出是哪道菜让人过敏**
>
> 一桌 20 道菜吃完过敏了。ddmin 的做法：先只吃前 10 道，没事？那问题在后 10 道或者两部分的组合里；再分成 4 份试……最后发现是“3 号菜 + 17 号菜”一起吃才过敏（测试里的 `[3, 17]`）。结构化缩小则像减量：先吃一半份量还过敏吗？再少一口呢？直到找到最小的那一口。

题目要求

在 `mine/t4_minimize.py` 里实现（`DTYPE_ORDER`、`random_case`、`make_input`、`row_sum_reference`、`buggy_row_sum` 已给出；容差和 `check_close` 来自 K3 的参考解）：

函数	要点
`ddmin(items, fails)`	1-minimal 子列表
`shrink_int(n, fails, lo=1)`	先减半、再减一
`shrink_case(case, fails)`	形状 → dtype → 连续性，循环到 fixpoint
`fuzz(check, n, seed, block)`	返回 `(第一个失败的 case 或 None, 试了几个)`
`differential_check(impl, case, seed)`	`impl(x)` vs float64 参考，容差 `tolerance(dtype, n, c=4)`，atol 再乘 `max(1, √n)`
`to_pytest(case, name, impl, seed)`	生成一个独立的 pytest 函数源码

题干

mine/t4_minimize.py 91 行 复制

```
"""T4 · Correctness bug minimization: fuzz -> fail -> shrink -> regression test — your imp

The loop every compiler contributor runs before opening an issue:

  1. random differential test against a reference (fixed seed, edge-heavy shapes, several
     non-contiguous strides);
  2. keep the first failing input;
  3. shrink it — smallest shape, simplest dtype, contiguous if possible — while it still f
  4. print it as a pytest regression (or a lit test once you know which pass is wrong).

`ddmin` is Zeller's delta debugging for lists (e.g. a list of IR lines or of ops in a grap
`shrink_case` is the structured version for (shape, dtype, layout) cases.
"""
import random

import numpy as np

from solutions.k3_numerics import check_close, edge_sizes, to_bf16, tolerance


# ---------------------------------------------------------------- generic list shrinking
def ddmin(items, fails):
    """Smallest sub-list (1-minimal) for which fails(sub) is still True. fails(items) must
    raise NotImplementedError("ddmin")


def shrink_int(n, fails, lo=1):
    """Smallest m in [lo, n] with fails(m) (assumes fails(n)); halve first, then step down
    raise NotImplementedError("shrink_int")


# ---------------------------------------------------------------- structured cases
DTYPE_ORDER = ["float32", "float16", "bfloat16"]       # simplest first


def shrink_case(case, fails):
    """case = {'shape': [..], 'dtype': str, 'contiguous': bool}. Shrink until nothing chan
    raise NotImplementedError("shrink_case")


def random_case(rng, block=64):
    sizes = edge_sizes(block, limit=4 * block)
    return {"shape": [rng.choice(sizes), rng.choice(sizes)], "dtype": rng.choice(DTYPE_ORD
            "contiguous": rng.random() < 0.7}


def fuzz(check, n=200, seed=0, block=64):
    """Run `check(case) -> bool ok` on n random cases; return (first failing case or None,
    raise NotImplementedError("fuzz")
```

```
# ---------------------------------------------------------------- the op under test
NP = {"float32": np.float32, "float16": np.float16, "bfloat16": np.float32}   # numpy has


def make_input(case, seed=0):
    m, n = case["shape"]
    rng = np.random.default_rng(seed)
    if case["contiguous"]:
        x = rng.standard_normal((m, n))
    else:
        x = rng.standard_normal((n, m)).T            # same values pattern, transposed str
    x = x.astype(NP[case["dtype"]])
    return to_bf16(x) if case["dtype"] == "bfloat16" else x


def row_sum_reference(x):
    return np.asarray(x, np.float64).sum(axis=1)


def differential_check(impl, case, seed=0):
    """impl(x) vs float64 reference with a K3 tolerance for this dtype and row length."""
    raise NotImplementedError("differential_check")


def buggy_row_sum(x, BLOCK=64):
    """A 'tiled kernel' with a classic bug: the single-block path is masked, the multi-blo
    walks only full blocks and forgets the tail."""
    m, n = x.shape
    acc = np.zeros(m, np.float32)
    if n <= BLOCK:
        acc += x.astype(np.float32).sum(axis=1)
    else:
        for k0 in range(0, n - BLOCK + 1, BLOCK):     # BUG: drops n % BLOCK trailing colu
            acc += x[:, k0:k0 + BLOCK].astype(np.float32).sum(axis=1)
    return acc


def to_pytest(case, name="test_row_sum_regression", impl="row_sum", seed=0):
    """Emit a standalone regression test for the minimized case."""
    raise NotImplementedError("to_pytest")
```

验收

```
./check.sh t4
```

tests/test_t4_minimize.py 57 行 复制

```
import numpy as np

from drill import load

t4 = load("t4_minimize")


def test_ddmin_finds_the_two_culprits():
    items = list(range(20))
    fails = lambda xs: 3 in xs and 17 in xs
    assert sorted(t4.ddmin(items, fails)) == [3, 17]
    assert t4.ddmin(list("abcdefgh"), lambda xs: "e" in xs) == ["e"]


def test_ddmin_ir_lines():
    ir = ["%0 = tt.get_program_id x", "%1 = tt.make_range", "%2 = tt.load %p", "%3 = tt.do
          "%4 = ttg.convert_layout %3", "tt.store %q, %4", "tt.return"]
    crash = lambda lines: any("tt.dot" in l for l in lines) and any("convert_layout" in l
    assert t4.ddmin(ir, crash) == ["%3 = tt.dot %a, %b", "%4 = ttg.convert_layout %3"]


def test_shrink_int():
    assert t4.shrink_int(1000, lambda m: m >= 65) == 65
    assert t4.shrink_int(1000, lambda m: m % 64 != 0 and m > 64) == 65
    assert t4.shrink_int(7, lambda m: True) == 1


def test_reference_impl_passes_and_bug_is_found():
    good = lambda x: x.astype(np.float32).sum(axis=1)
    for case in [{"shape": [3, 129], "dtype": "float16", "contiguous": False},
                 {"shape": [1, 1], "dtype": "float32", "contiguous": True}]:
        assert t4.differential_check(good, case)
    bad, tried = t4.fuzz(lambda c: t4.differential_check(t4.buggy_row_sum, c), n=200, seed
    assert bad is not None and bad["shape"][1] > 64 and bad["shape"][1] % 64 != 0


def test_shrink_case_to_minimal():
    check = lambda c: t4.differential_check(t4.buggy_row_sum, c)
    start = {"shape": [129, 255], "dtype": "bfloat16", "contiguous": False}
    assert not check(start)
    m = t4.shrink_case(start, lambda c: not check(c))
    assert m == {"shape": [1, 65], "dtype": "float32", "contiguous": True}


def test_to_pytest_runs():
    code = t4.to_pytest({"shape": [1, 65], "dtype": "float32", "contiguous": True}, impl="
    ns = {"np": np, "buggy": t4.buggy_row_sum}
    exec(code, ns)
    try:
        ns["test_row_sum_regression"]()
```

```
    except AssertionError:
        pass
    else:
        raise AssertionError("the emitted regression test should fail on the buggy kernel")
    ns["buggy"] = lambda x: x.astype(np.float32).sum(axis=1)
    exec(code, ns)
    ns["test_row_sum_regression"]()
```

提示

L1 shrink_int 方向（轻度剧透） 收起

```
while n > lo:
    half = max(lo, n // 2)
    if half < n and fails(half): n = half
    elif fails(n - 1): n -= 1
    else: break
```

注意它假设“失败的 n”大致单调；对 `n % 64 != 0 and n > 64` 这种不单调的条件，减半会失败，逐个减一仍然能找到 65。

L2 ddmin 的骨架 拆解与常见坑（中度剧透） 收起

`n = 2`；循环：切成 n 块；有某一块单独失败 → 换成它、`n = 2`；有某个补集失败（块数 > 2 时）→ 换成补集、`n = max(n - 1, 2)`；都不行 → 若 `n ≥ len(items)` 结束，否则 `n = min(2n, len)`。

L3 to_pytest 必须真的能跑 几乎是答案（重度剧透） 收起

测试会 `exec` 你生成的代码：用 buggy 实现时它必须失败，用正确实现时必须通过。非连续的 case 用 `rng.standard_normal((n, m)).T` 生成输入，和 `make_input` 保持一致。

参考解

解 solutions/t4_minimize.py 先自己写完、跑过测试再看 收起

solutions/t4_minimize.py 153 行 复制

```python
"""T4 · Correctness bug minimization: fuzz -> fail -> shrink -> regression test — refe

The loop every compiler contributor runs before opening an issue:

  1. random differential test against a reference (fixed seed, edge-heavy shapes, seve
     non-contiguous strides);
  2. keep the first failing input;
  3. shrink it — smallest shape, simplest dtype, contiguous if possible — while it sti
  4. print it as a pytest regression (or a lit test once you know which pass is wrong)

`ddmin` is Zeller's delta debugging for lists (e.g. a list of IR lines or of ops in a
`shrink_case` is the structured version for (shape, dtype, layout) cases.
"""
import random

import numpy as np

from solutions.k3_numerics import check_close, edge_sizes, to_bf16, tolerance


# ---------------------------------------------------------------- generic list shrink
def ddmin(items, fails):
    """Smallest sub-list (1-minimal) for which fails(sub) is still True. fails(items)
    assert fails(items), "the full input must fail"
    n = 2
    while len(items) >= 2:
        chunk = -(-len(items) // n)
        subsets = [items[i:i + chunk] for i in range(0, len(items), chunk)]
        progress = False
        for i, sub in enumerate(subsets):
            if fails(sub):                                  # reduce to a subset
                items, n, progress = sub, 2, True
                break
            rest = [x for j, s in enumerate(subsets) if j != i for x in s]
            if len(subsets) > 2 and fails(rest):            # reduce to a complement
                items, n, progress = rest, max(n - 1, 2), True
                break
        if not progress:
            if n >= len(items):
                break
            n = min(n * 2, len(items))
    return items


def shrink_int(n, fails, lo=1):
    """Smallest m in [lo, n] with fails(m) (assumes fails(n)); halve first, then step
    assert fails(n)
    while n > lo:
        half = max(lo, n // 2)
        if half < n and fails(half):
```

```
            n = half
        elif fails(n - 1):
            n -= 1
        else:
            break
    return n


# ---------------------------------------------------------------- structured cases
DTYPE_ORDER = ["float32", "float16", "bfloat16"]       # simplest first


def shrink_case(case, fails):
    """case = {'shape': [..], 'dtype': str, 'contiguous': bool}. Shrink until nothing
    case = dict(case, shape=list(case["shape"]))
    assert fails(case)
    changed = True
    while changed:
        changed = False
        for d in range(len(case["shape"])):
            def f(m, d=d):
                c = dict(case, shape=case["shape"][:d] + [m] + case["shape"][d + 1:])
                return fails(c)
            m = shrink_int(case["shape"][d], f)
            if m != case["shape"][d]:
                case["shape"][d] = m
                changed = True
        for dt in DTYPE_ORDER[:DTYPE_ORDER.index(case["dtype"])]:
            if fails(dict(case, dtype=dt)):
                case["dtype"] = dt
                changed = True
                break
        if not case["contiguous"] and fails(dict(case, contiguous=True)):
            case["contiguous"] = True
            changed = True
    return case


def random_case(rng, block=64):
    sizes = edge_sizes(block, limit=4 * block)
    return {"shape": [rng.choice(sizes), rng.choice(sizes)], "dtype": rng.choice(DTYPE
            "contiguous": rng.random() < 0.7}


def fuzz(check, n=200, seed=0, block=64):
    """Run `check(case) -> bool ok` on n random cases; return (first failing case or N
    rng = random.Random(seed)
    for i in range(n):
        c = random_case(rng, block)
        if not check(c):
            return c, i + 1
```

```
    return None, n


# ---------------------------------------------------------------- the op under test
NP = {"float32": np.float32, "float16": np.float16, "bfloat16": np.float32}   # numpy


def make_input(case, seed=0):
    m, n = case["shape"]
    rng = np.random.default_rng(seed)
    if case["contiguous"]:
        x = rng.standard_normal((m, n))
    else:
        x = rng.standard_normal((n, m)).T            # same values pattern, transposed
    x = x.astype(NP[case["dtype"]])
    return to_bf16(x) if case["dtype"] == "bfloat16" else x


def row_sum_reference(x):
    return np.asarray(x, np.float64).sum(axis=1)


def differential_check(impl, case, seed=0):
    """impl(x) vs float64 reference with a K3 tolerance for this dtype and row length.
    x = make_input(case, seed)
    rtol, atol = tolerance(case["dtype"], case["shape"][1], c=4.0)
    return check_close(impl(x), row_sum_reference(x), rtol, atol * max(1.0, case["shap


def buggy_row_sum(x, BLOCK=64):
    """A 'tiled kernel' with a classic bug: the single-block path is masked, the multi
    walks only full blocks and forgets the tail."""
    m, n = x.shape
    acc = np.zeros(m, np.float32)
    if n <= BLOCK:
        acc += x.astype(np.float32).sum(axis=1)
    else:
        for k0 in range(0, n - BLOCK + 1, BLOCK):     # BUG: drops n % BLOCK trailing
            acc += x[:, k0:k0 + BLOCK].astype(np.float32).sum(axis=1)
    return acc


def to_pytest(case, name="test_row_sum_regression", impl="row_sum", seed=0):
    """Emit a standalone regression test for the minimized case."""
    m, n = case["shape"]
    make = (f"rng.standard_normal(({m}, {n}))" if case["contiguous"]
            else f"rng.standard_normal(({n}, {m})).T")
    return (f"def {name}():\n"
            f"    # minimized by t4_minimize.shrink_case; seed={seed}\n"
            f"    rng = np.random.default_rng({seed})\n"
```

```
            f"    x = {make}.astype(np.{NP[case['dtype']].__name__})\n"
            f"    np.testing.assert_allclose({impl}(x), x.astype(np.float64).sum(axis=
```

在 RTX 3070 上：换成真的 Triton kernel

把 `buggy_row_sum` 换成一个真实的 Triton 行求和（自己故意写错尾部 mask，或者挑一个你怀疑的 op，比如某个 dtype 的 `tl.cumsum`），`impl = lambda x: triton_row_sum(torch.from_numpy(x).cuda()).cpu().numpy()`，其余代码不变。找到真 bug 时，把 `to_pytest` 的输出改写成 `test_core.py` 的风格（`@pytest.mark.parametrize` 形状和 dtype，`device` fixture），附上 `git rev-parse HEAD`、GPU 和 Toolkit 版本，就是一个合格的 issue。如果你已经定位到某个 pass，用 `MLIR_ENABLE_DUMP=1` 拿到出错前的 TTGIR，用 `ddmin` 缩小它，再写成 T5 那样的 lit 测试。

▼ 自测：为什么要先简化 dtype，再试连续 stride?

每一种简化都去掉一个“可能的原因”。如果 fp32 + 连续 stride 还能复现，维护者马上知道问题与低精度、与非连续访存无关，只与形状有关；反过来，如果只有 bf16 才失败，那 bug 很可能在类型转换或累加精度上。最小案例的价值不仅是短，更是它**排除了多少假设**。

双主线 · Track T · Triton（kernel compiler） · 第 9 题

T5 lit / FileCheck：自己写一个 FileCheck，读懂 Triton 的编译器测试

Triton 的编译器测试几乎都是“一个 .mlir 文件 + 一行 RUN + 一串 CHECK 注释”。这道题自己实现一个 FileCheck（CHECK / NEXT / SAME / NOT / LABEL / DAG / COUNT、{{regex}}、[[VAR:regex]] 捕获与引用），然后你就能读懂、也能写出 Triton 的 lit 测试——这是最稳妥的第一笔贡献。

⏱ 60 min | 难度 难 | 你写 `mine/t5_filecheck.py` | 验收 `./check.sh t5` | 参考解 `solutions/t5_filecheck.py` | 📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 题目要求
5. 题干
6. 验收
7. 提示
8. 参考解
9. 在 Triton 仓库里：读一个、改一个、写一个

背景

roadmap 对 Triton 贡献的建议：“最稳妥的第一步是补一个能稳定失败 / 通过的 regression test，再提交针对性修复。”Week 7 的任务是“读 lit / FileCheck 与 Python tests”。

一个 lit 测试的执行过程：

```
// RUN: triton-opt %s -split-input-file -tritongpu-coalesce | FileCheck %s
```

1. lit 把 `%s` 换成这个文件的路径，执行 RUN 行：`triton-opt` 对文件跑 `tritongpu-coalesce` 这个 pass，输出变换后的 IR。
2. 输出通过管道交给 `FileCheck %s`：FileCheck 从**同一个文件**里读出所有 `// CHECK...` 注释，按规则去匹配输出。
3. 全部匹配 → 测试通过。

源码与资料

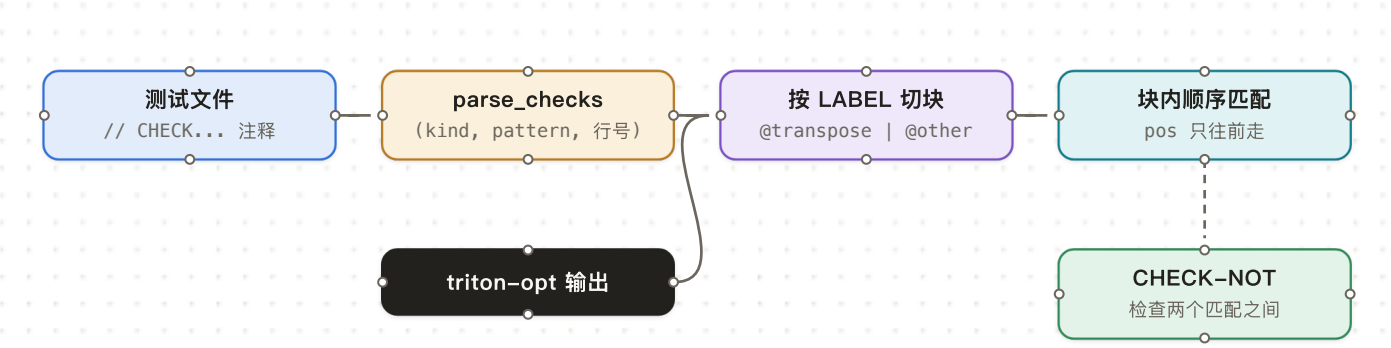
`triton › test/TritonGPU/coalesce.mlir` ·
`triton › accelerate-matmul.mlir（CHECK-LABEL 与 MMA 变量捕获）` ·
`triton › tritongpu_to_llvm.mlir（CHECK-COUNT、CHECK-NOT）` · LLVM FileCheck 文档 · lit 文档

要用到的知识

指令	语义
`CHECK: pat`	从当前位置往后找 pat
`CHECK-NEXT: pat`	必须在上一个匹配的**下一行**
`CHECK-SAME: pat`	必须在上一个匹配的**同一行**、它之后
`CHECK-NOT: pat`	在上一个匹配和下一个正向匹配之间**不能出现**
`CHECK-LABEL: pat`	先用所有 LABEL 把输入切成块，其他 CHECK 不能跨块（一个函数一个块）
`CHECK-DAG: pat`	连续的一组 DAG 可以任意顺序匹配（但不能重叠）
`CHECK-COUNT-n: pat`	连续匹配 n 次
`{{regex}}`	一段正则
`[[NAME:regex]]` / `[[NAME]]`	捕获 / 引用变量；`[[$NAME]]` 在 `--enable-var-scope` 下也能跨 LABEL

模式里的空白序列匹配输入里的任意空白序列，其余字符都按字面匹配（所以 `#ttg.blocked<{` 不用转义）。

FileCheck 怎么扫描输出



关键是“当前位置”pos：每个正向匹配都把它推到匹配末尾，NOT 被挂起，直到下一个正向匹配确定了检查区间的终点。

类比 · 对答案的方式

FileCheck 不是逐字比较整份输出（那样任何无关改动都会打破测试），而是像改卷老师只看几个**关键步骤**有没有、顺序对不对。LABEL 是“第几题”，每道题只在自己的范围里找；NEXT 是“这一步必须紧接着上一步”；NOT 是“这两步之间不许出现某个错误写法”；`[[row_layout]]` 是“你第一问算出来的数，第三问要用同一个”。

题目要求

在 `mine/t5_filecheck.py` 里实现（`DIRECTIVE` 正则、`CheckFailure`、`split_input_file` 已给出）：

函数	要点
`parse_checks(source, prefix)`	`[(kind, pattern, line_no)]`；`COUNT-n` 展开成 n 个 `CHECK`；支持 `--check-prefix` 式的自定义前缀
`compile_pattern(pat, env)`	转成 Python 正则，返回 `(regex, [捕获的变量名])`；引用未定义变量要报 `undefined`
`filecheck(check_source, output, prefix, enable_var_scope)`	通过返回 True，否则抛 `CheckFailure`，消息里带上失败的 CHECK 行号和种类

内部怎么组织随你；参考解用了 `_check_block`（处理一个 LABEL 块）和 `_check_nots`。

题干

mine/t5_filecheck.py 54 行 复制

```
"""T5 · A mini FileCheck: how Triton's lit tests decide pass / fail — your implementation

Every file under triton/test/ starts with a RUN line such as
    // RUN: triton-opt %s -split-input-file -tritongpu-coalesce | FileCheck %s
lit runs the pass, pipes the output to LLVM's FileCheck, and FileCheck matches the `// CHE
comments of the same file against that output. Supported here (the subset Triton tests use

  CHECK:        match somewhere at or after the current position
  CHECK-NEXT:   match on the line right after the previous match
  CHECK-SAME:   match on the same line as the previous match, after it
  CHECK-NOT:    must not appear between the previous match and the next positive match
  CHECK-LABEL:  splits input into blocks first; checks never cross a label
  CHECK-DAG:    a group of consecutive DAG checks may match in any order
  CHECK-COUNT-n: match n times in a row (each after the previous)

Patterns: literal text, except {{regex}} and [[VAR:regex]] (capture) / [[VAR]] (use), with
the `$` prefix ([[$VAR]]) meaning "global": with --enable-var-scope only those survive a C
match any whitespace run in the input (FileCheck's default canonicalization).
"""
import re

DIRECTIVE = re.compile(r"(?://|;|#)\s*(CHECK(?:-NEXT|-SAME|-NOT|-LABEL|-DAG|-COUNT-\d+)?):


class CheckFailure(AssertionError):
    pass


def parse_checks(source, prefix="CHECK"):
    """[(kind, pattern, line_no)] from a test file. kind: 'CHECK', 'NEXT', 'SAME', 'NOT',
    raise NotImplementedError("parse_checks")


def compile_pattern(pat, env):
    """Turn a FileCheck pattern into a Python regex. Returns (regex, [capture variable nam
    raise NotImplementedError("compile_pattern")









def filecheck(check_source, output, prefix="CHECK", enable_var_scope=False):
    """Raise CheckFailure (an AssertionError) if `output` does not satisfy the CHECKs in c
    raise NotImplementedError("filecheck")
```

```
def split_input_file(text):
    """triton-opt -split-input-file: chunks separated by a line `// -----`."""
    return [c.strip("\n") for c in re.split(r"^// -----\s*$", text, flags=re.M)]
```

验收

```
./check.sh t5
```

tests/test_t5_filecheck.py 96 行 复制

```
import pytest

from drill import load

t5 = load("t5_filecheck")
FC = t5.filecheck
Fail = t5.CheckFailure

OUT = """#blocked = #ttg.blocked<{sizePerThread = [1, 4], threadsPerWarp = [2, 16], warpsP
#blocked1 = #ttg.blocked<{sizePerThread = [4, 1], threadsPerWarp = [16, 2], warpsPerCTA =
module {
  tt.func @transpose(%arg0: !tt.ptr<f32>) {
    %8 = ttg.convert_layout %7 : tensor<64x64x!tt.ptr<f32>, #blocked2> -> tensor<64x64x!tt
    %9 = tt.load %8 : tensor<64x64x!tt.ptr<f32>, #blocked>
    %10 = ttg.convert_layout %9 : tensor<64x64xf32, #blocked> -> tensor<64x64xf32, #blocke
    tt.store %11, %10 : tensor<64x64x!tt.ptr<f32>, #blocked1>
    tt.return
  }
  tt.func @other() {
    %0 = arith.constant 0 : i32
    tt.return
  }
}"""


def test_parse_checks():
    src = "// CHECK: a\n// CHECK-NEXT: b\n  // CHECK-COUNT-2: c\n// CHECK-NOT: d\n// CHECK
    kinds = [k for k, _, _ in t5.parse_checks(src)]
    assert kinds == ["CHECK", "NEXT", "CHECK", "CHECK", "NOT", "LABEL", "SAME"]
    assert t5.parse_checks("// MMA: x", prefix="MMA") == [("CHECK", "x", 1)]


def test_compile_pattern():
    rx, names = t5.compile_pattern("[[L:#.*]] = #ttg.blocked<{{.*}}order = [1, 0]", {})
    assert names == ["L"] and rx.search(OUT).group("L") == "#blocked"
    rx, _ = t5.compile_pattern("tt.load   [[P]]", {"P": "%8"})
    assert rx.search("%9 = tt.load %8 : x")


def test_real_coalesce_style_checks_pass():
    checks = """
// CHECK: [[row:#.*]] = #ttg.blocked<{sizePerThread = [1, 4], threadsPerWarp = [2, 16]
// CHECK: [[col:#.*]] = #ttg.blocked<{sizePerThread = [4, 1]
// CHECK-LABEL: @transpose
// CHECK: [[ptr:%.*]] = ttg.convert_layout {{.*}} -> tensor<64x64x!tt.ptr<f32>, [[row]]>
// CHECK-NEXT: [[val:%[0-9]+]] = tt.load [[ptr]]
// CHECK-SAME: [[row]]>
// CHECK-NOT: tt.dot
// CHECK: tt.store {{.*}}, [[col]]>
"""
```

```
    assert FC(checks, OUT)


def test_failures_are_reported():
    with pytest.raises(Fail, match="not found"):
        FC("// CHECK: tt.dot", OUT)
    with pytest.raises(Fail, match="CHECK-NEXT"):
        FC("// CHECK: tt.func @transpose\n// CHECK-NEXT: tt.load", OUT)
    with pytest.raises(Fail, match="CHECK-NOT"):
        FC("// CHECK: @transpose\n// CHECK-NOT: tt.load\n// CHECK: tt.store", OUT)
    with pytest.raises(Fail, match="no check strings"):
        FC("// nothing here", OUT)


def test_order_matters_unless_dag():
    with pytest.raises(Fail):
        FC("// CHECK: tt.store\n// CHECK: tt.load", OUT)
    assert FC("// CHECK-DAG: tt.store\n// CHECK-DAG: tt.load", OUT)


def test_labels_bound_the_search():
    # arith.constant exists, but only in @other: it must not satisfy a check in the @trans
    with pytest.raises(Fail):
        FC("// CHECK-LABEL: @transpose\n// CHECK: arith.constant\n// CHECK-LABEL: @other",
    assert FC("// CHECK-LABEL: @transpose\n// CHECK: tt.return\n// CHECK-LABEL: @other\n//


def test_count_and_variables():
    assert FC("// CHECK-COUNT-2: ttg.convert_layout", OUT)
    with pytest.raises(Fail):
        FC("// CHECK-COUNT-3: ttg.convert_layout", OUT)
    with pytest.raises(Fail, match="undefined"):
        FC("// CHECK: tt.load [[NOPE]]", OUT)


def test_var_scope():
    src = "// CHECK: [[B:#blocked1]] =\n// CHECK-LABEL: @transpose\n// CHECK: tt.store {{.
    assert FC(src, OUT)
    with pytest.raises(Fail, match="undefined"):
        FC(src, OUT, enable_var_scope=True)
    assert FC(src.replace("[[B", "[[$B"), OUT, enable_var_scope=True)


def test_split_input_file():
    parts = t5.split_input_file("a\n// -----\nb\nc\n// -----\nd")
    assert parts == ["a", "b\nc", "d"]
```

提示

L1 compile_pattern 方向（轻度剧透） 收起

逐字符扫描：遇到 `{{` 找到 `}}`，中间原样放进 `(?:...)`；遇到 `[[`：带 `:` 是捕获，变成命名组 `(?P<name>...)`（名字里的 `$` 要换成合法字符），否则用 `re.escape(env[name])`；空白序列 → `\s+`；其他字符 `re.escape`。

L2 NEXT 和 SAME 用“行边界” 拆解与常见坑（中度剧透） 收起

写一个 `_line_bounds(text, pos)` 返回 pos 所在行的 `[start, end)`。SAME：在上一个匹配的结尾到行尾之间搜。NEXT：取上一个匹配所在行的下一行的边界，在那一行里搜。

L3 NOT 的区间 几乎是答案（重度剧透） 收起

遇到 NOT 先挂起；下一个正向匹配（或 DAG 组的最早匹配）确定后，检查 `[上一个匹配的结尾, 这个匹配的开头)`；块结束时，剩下的 NOT 检查到块尾。LABEL：先按顺序找到所有 LABEL 的位置，再对每个块单独跑。

参考解

解 **solutions/t5_filecheck.py** 先自己写完、跑过测试再看 收起 ▾

solutions/t5_filecheck.py 193 行 复制

```
"""T5 · A mini FileCheck: how Triton's lit tests decide pass / fail — reference soluti

Every file under triton/test/ starts with a RUN line such as
    // RUN: triton-opt %s -split-input-file -tritongpu-coalesce | FileCheck %s
lit runs the pass, pipes the output to LLVM's FileCheck, and FileCheck matches the `//
comments of the same file against that output. Supported here (the subset Triton tests

  CHECK:        match somewhere at or after the current position
  CHECK-NEXT:   match on the line right after the previous match
  CHECK-SAME:   match on the same line as the previous match, after it
  CHECK-NOT:    must not appear between the previous match and the next positive match
  CHECK-LABEL:  splits input into blocks first; checks never cross a label
  CHECK-DAG:    a group of consecutive DAG checks may match in any order
  CHECK-COUNT-n: match n times in a row (each after the previous)

Patterns: literal text, except {{regex}} and [[VAR:regex]] (capture) / [[VAR]] (use),
the `$` prefix ([[$VAR]]) meaning "global": with --enable-var-scope only those survive
match any whitespace run in the input (FileCheck's default canonicalization).
"""
import re

DIRECTIVE = re.compile(r"(?://|;|#)\s*(CHECK(?:-NEXT|-SAME|-NOT|-LABEL|-DAG|-COUNT-\d+


class CheckFailure(AssertionError):
    pass


def parse_checks(source, prefix="CHECK"):
    """[(kind, pattern, line_no)] from a test file. kind: 'CHECK', 'NEXT', 'SAME', 'NO
    out = []
    rx = re.compile(DIRECTIVE.pattern.replace("CHECK", re.escape(prefix)))
    for i, line in enumerate(source.splitlines(), 1):
        m = rx.search(line)
        if not m:
            continue
        tag, pat = m.group(1), m.group(2).strip()
        kind = tag[len(prefix):].lstrip("-") or "CHECK"
        if kind.startswith("COUNT-"):
            out.extend([("CHECK", pat, i)] * int(kind[6:]))
        else:
            out.append((kind, pat, i))
    return out


def compile_pattern(pat, env):
    """Turn a FileCheck pattern into a Python regex. Returns (regex, [capture variable
    rx, names, i = "", [], 0
    while i < len(pat):
        if pat.startswith("{{", i):
```

```
            j = pat.index("}}", i + 2)
            rx += "(?:" + pat[i + 2:j] + ")"
            i = j + 2
        elif pat.startswith("[[", i):
            j = pat.index("]]", i + 2)
            body = pat[i + 2:j]
            if ":" in body:
                name, sub = body.split(":", 1)
                key = re.sub(r"\W", "_", name)
                rx += f"(?P<{key}>{sub})"
                names.append(name)
            else:
                if body not in env:
                    raise CheckFailure(f"undefined variable [[{body}]]")
                rx += re.escape(env[body])
            i = j + 2
        elif pat[i].isspace():
            while i < len(pat) and pat[i].isspace():
                i += 1
            rx += r"\s+"
        else:
            rx += re.escape(pat[i])
            i += 1
    return re.compile(rx), names


def _search(pat, text, pos, end, env):
    rx, names = compile_pattern(pat, env)
    m = rx.search(text, pos, end)
    if m:
        for n in names:
            env[n] = m.group(re.sub(r"\W", "_", n))
    return m


def _line_bounds(text, pos):
    s = text.rfind("\n", 0, pos) + 1
    e = text.find("\n", pos)
    return s, (len(text) if e < 0 else e)


def _check_block(checks, text, start, end, env):
    pos = start
    last = None             # (match_start, match_end) of the previous positive match
    pending_not = []
    i = 0
    while i < len(checks):
        kind, pat, ln = checks[i]
        if kind == "NOT":
            pending_not.append((pat, ln))
            i += 1
```

```
            continue
        if kind == "DAG":
            group = []
            while i < len(checks) and checks[i][0] == "DAG":
                group.append(checks[i])
                i += 1
            used, hi = [], pos
            for _, p, l in group:
                rx, names = compile_pattern(p, env)
                found = None
                for m in rx.finditer(text, pos, end):
                    if all(m.end() <= a or m.start() >= b for a, b in used):
                        found = m
                        break
                if not found:
                    raise CheckFailure(f"line {l}: CHECK-DAG: {p!r} not found")
                for n in names:
                    env[n] = found.group(re.sub(r"\W", "_", n))
                used.append((found.start(), found.end()))
                hi = max(hi, found.end())
            _check_nots(pending_not, text, pos, min(a for a, _ in used), env)
            pending_not, pos, last = [], hi, (min(a for a, _ in used), hi)
            continue
        if kind == "SAME":
            if last is None:
                raise CheckFailure(f"line {ln}: CHECK-SAME with no previous match")
            _, le = _line_bounds(text, last[1] - 1 if last[1] > last[0] else last[1])
            m = _search(pat, text, last[1], le, env)
            if not m:
                raise CheckFailure(f"line {ln}: CHECK-SAME: {pat!r} not on the same li
        elif kind == "NEXT":
            if last is None:
                raise CheckFailure(f"line {ln}: CHECK-NEXT with no previous match")
            _, le = _line_bounds(text, max(last[1] - 1, last[0]))
            if le >= end:
                raise CheckFailure(f"line {ln}: CHECK-NEXT: no next line")
            ns, ne = _line_bounds(text, le + 1)
            m = _search(pat, text, ns, ne, env)
            if not m:
                raise CheckFailure(f"line {ln}: CHECK-NEXT: {pat!r} is not on the next
        else:
            m = _search(pat, text, pos, end, env)
            if not m:
                raise CheckFailure(f"line {ln}: CHECK: {pat!r} not found")
        _check_nots(pending_not, text, pos, m.start(), env)
        pending_not = []
        last, pos = (m.start(), m.end()), m.end()
        i += 1
    _check_nots(pending_not, text, pos, end, env)
```

```
def _check_nots(nots, text, lo, hi, env):
    for pat, ln in nots:
        rx, _ = compile_pattern(pat, env)
        m = rx.search(text, lo, hi)
        if m:
            raise CheckFailure(f"line {ln}: CHECK-NOT: {pat!r} found: {m.group(0)!r}")


def filecheck(check_source, output, prefix="CHECK", enable_var_scope=False):
    """Raise CheckFailure (an AssertionError) if `output` does not satisfy the CHECKs
    checks = parse_checks(check_source, prefix)
    if not checks:
        raise CheckFailure("no check strings found")
    env = {}
    # 1. labels cut the input into blocks
    labels = [(i, c) for i, c in enumerate(checks) if c[0] == "LABEL"]
    if not labels:
        _check_block(checks, output, 0, len(output), env)
        return True
    starts = []
    pos = 0
    for _, (_, pat, ln) in labels:
        m = _search(pat, output, pos, len(output), env)
        if not m:
            raise CheckFailure(f"line {ln}: CHECK-LABEL: {pat!r} not found")
        starts.append((m.start(), m.end()))
        pos = m.end()
    idx = [i for i, _ in labels] + [len(checks)]
    if labels[0][0] > 0:
        _check_block(checks[:labels[0][0]], output, 0, starts[0][0], env)
    for b, (s, e) in enumerate(starts):
        block_end = starts[b + 1][0] if b + 1 < len(starts) else len(output)
        if enable_var_scope:     # FileCheck --enable-var-scope: only [[$VAR]] survive
            env = {k: v for k, v in env.items() if k.startswith("$")}
        _check_block(checks[idx[b] + 1: idx[b + 1]], output, e, block_end, env)
    return True


def split_input_file(text):
    """triton-opt -split-input-file: chunks separated by a line `// -----`."""
    return [c.strip("\n") for c in re.split(r"^// -----\s*$", text, flags=re.M)]
```

在 Triton 仓库里：读一个、改一个、写一个

1. **读**：打开 `coalesce.mlir`，对着 T2 的手算解释每一条 CHECK；用你的 `filecheck()` 跑一遍：`filecheck(open("coalesce.mlir").read(), subprocess.check_output(["triton-opt", ...]))`，和真 FileCheck 结果一致。

2. **改**：故意改坏 `CoalesceUtils.cpp` 里的 `perThread` 计算（比如去掉 `128 / elemNumBits` 的上限），重新 build，用 `lit test/TritonGPU/coalesce.mlir -v` 看哪条 CHECK 先失败、失败信息长什么样。
3. **写**：为 T4 找到的问题（或者 coalesce 的一个没被覆盖的情况，比如 `tt.divisibility = 8`）写一个新的 `// -----` 分段测试：输入 IR + RUN 行 + 最少但足够的 CHECK。好的 CHECK 只检查你关心的东西，别把整个输出抄进去。

```
cd triton && pip install -e . --no-build-isolation      # 第一次 build
lit -v build/cmake.*/test/TritonGPU/coalesce.mlir        # 或：cd build/cmake.* && ninja ch
```

▼ 自测：为什么 Triton 的 CHECK 里到处是 `[[row_layout:#.*]]` 这种捕获，而不是直接写 `#blocked`？

layout alias 的名字（`#blocked`、`#blocked1`、`#blocked2`）是打印器按出现顺序分配的，一个无关的改动（多出一个 layout）就会让编号整体移位。捕获“那个内容是 `sizePerThread = [1, 4], ...` 的 alias”，后面再用 `[[row_layout]]` 引用，测试只依赖**语义**（layout 的内容和它被用在哪里），不依赖名字。这也是写新测试时最常被 reviewer 指出的问题之一。

Track L · Luminal（graph compiler）

primitive graph、e-graph 与 equality saturation、实测驱动的搜索、执行计划与 arena、Reference vs CUDA 对拍。

双主线 · Track L · Luminal（graph compiler） · 第 10 题

L1 Graph semantics：把 relu(x @ w + b) 拆成 primitive ops

把 y = relu(x @ w + b) 交给 Luminal，它记录下来的不是三个“大算子”，而是十几个 primitive 节点：broadcast 视图、一个 [M, N, K] 的 Mul、一个 ReduceSum、一个 LessThan 和一个 Select。这道题用 Python 写一个 Luminal 式的“只记录、不计算”的图，给出节点语义表，再写一个按拓扑序执行的 reference runtime。

⏱ 40 min | 难度 易 | 你写 `mine/l1_graph.py` | 验收 `./check.sh l1` | 参考解 `solutions/l1_graph.py`

📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 题目要求
5. 题干
6. 验收
7. 提示
8. 参考解
9. 在 Luminal 里：看真正的图

背景

roadmap 的 L1：“构建 y = relu(x @ w + b)，标出 shape、dtype、broadcast 和每个 primitive。交付：graph DAG + 节点语义表。”

真实 Luminal 里这件事由 `GraphTensor` 的方法完成：

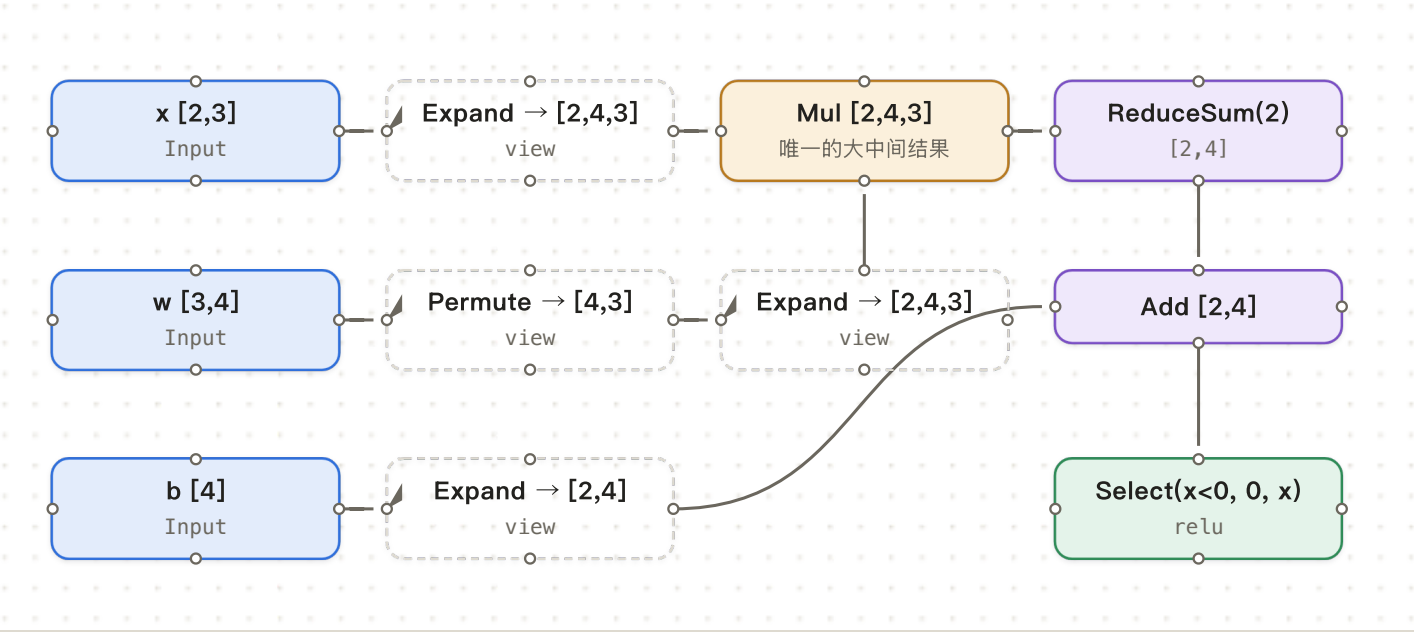
- `luminal › matmul`：“broadcast + reduce in the operands' shared dtype, NOTHING else”——把两个操作数映射到 `[..., M, N, K]`，相乘，沿 K 求和；从不插入 cast。
- `luminal › relu` = `maximum_f32(0.)`；`luminal › maximum` = `lt(...).select(...)`，外加一个 NaN 覆盖（注释解释了为什么不用 `fmax`）。
- `luminal › exp` = `(x · log2(e)).exp2()`：primitive 集里只有 `Exp2`。

源码与资料

`luminal › README: RISC-style architecture (15 个 primitive)` ·
`luminal › src/logical/ir.rs: LogicalOp` · `luminal › src/graph.rs` ·
`luminal › frontend/movement.rs: expand / permute`

要用到的知识

relu(x @ w + b) 记录成的 DAG（M=2, K=3, N=4）



灰色的 Expand / Permute 只改变索引方式，不搬数据。真正“贵”的是 Mul：如果每个节点都物化成一个 buffer，`[M, N, K]` 的乘积会占 M·N·K·4 字节——这正是 Luminal 需要 e-graph 和搜索，把 Mul + ReduceSum 合成一个 GEMM（或者交给 cuBLASLt）的原因。

规则	说明
broadcast 是显式的	二元 op 要求两边 shape 完全一致；`b[N]` 加到 `[M, N]` 上前要先 `Expand`
记录不计算	`x + y` 只往 `graph.nodes` 里追加一个节点，返回一个只含 id 的句柄
dtype 随节点走	`LessThan` 产出 `bool`；`Select` 的 dtype 跟随两个分支
拓扑序执行	reference runtime 从输出反向 DFS，得到“每个输入都在使用者之前”的顺序

类比 · 录音而不是现场演奏

PyTorch eager 像现场演奏：每写一行，乐手就弹一个音（`x + y` 立刻算出结果）。Luminal 像录音棚里先把整首曲子写成**分轨乐谱**（DAG），而且所有乐器都被拆成几种基本音色（primitive）。写完之后编曲（e-graph + 搜索）可以重新安排：把两条轨合并成一个和弦（fusion），用采样代替真乐手（cuBLASLt）。reference runtime 就是按乐谱老老实实一轨一轨弹一遍，用来核对编曲之后的版本有没有弹错（L5）。

题目要求

在 `mine/l1_graph.py` 里实现（`Graph.add / tensor / constant`、`Tensor` 的 movement 和二元 op、`sum`、`exp2` 已给出）：

函数	要点
`Graph.topo_order()`	从 `outputs`（没有就用最后一个节点）反向 DFS，后序
`Graph.op_counts()`	只数拓扑序里可达的节点
`Graph.semantics_table()`	`(id, op, inputs, shape, dtype, note)`；Expand / Permute 的 note 里写上 `view`
`Graph.materialized_bytes(elem_bytes)`	非 view、非 Input / Constant 节点的 buffer 字节数之和
`Tensor.matmul(w)`	`self.expand(1, N)`；`w.permute([1, 0]).expand(0, M)`；`Mul`；`sum(2)`
`Tensor.relu()`	`self.lt(zero).select(zero, self)`
`Tensor.exp()`	`(self · log2(e)).exp2()`
`build_mlp(M, K, N)`	返回 `(g, x, w, b, y)`
`run(g, feeds)`	numpy 执行，返回 `{node id: array}`

题干

mine/l1_graph.py 152 行 复制

```
"""L1 · Graph semantics: y = relu(x @ w + b) as a Luminal-style primitive graph — your imp

Luminal records every tensor operation into a static DAG instead of running it (lazy, ahea
time). High-level ops are *spelled* with a small primitive set (README: "Everything in Lum
boils down to 15 primitive ops"; src/logical/ir.rs `LogicalOp` has the current full list):

  matmul(a, b)  = broadcast both to [M, N, K]  ->  Mul  ->  ReduceSum(K)     (src/frontend
  relu(x)       = maximum(x, 0) = Select(LessThan(x, 0), 0, x)                (src/fronten
  exp(x)        = Exp2(x * log2(e))                                           (src/fronten
  a + b[N]      = Add(a, Expand(b))  — broadcasting is a *view*, not a copy

This file is a ~150-line Python model of that design: a Graph of nodes with shapes and dty
a GraphTensor API that records, a node-semantics table, and a reference runtime (numpy) th
executes the DAG in topological order — the analogue of Luminal's ReferenceRuntime.
"""
from dataclasses import dataclass, field

import numpy as np

PRIMITIVES = {"Input", "Constant", "Add", "Mul", "Recip", "Exp2", "Log2", "Sqrt", "Sin", "
              "Select", "ReduceSum", "ReduceMax", "Expand", "Permute", "Cast"}
MOVEMENT = {"Expand", "Permute"}          # views: change indexing, move no data


@dataclass
class Node:
    id: int
    op: str
    inputs: list
    shape: tuple
    dtype: str = "f32"
    attrs: dict = field(default_factory=dict)


class Graph:
    def __init__(self):
        self.nodes = []
        self.outputs = []

    def add(self, op, inputs, shape, dtype="f32", **attrs):
        assert op in PRIMITIVES, op
        n = Node(len(self.nodes), op, [t.id for t in inputs], tuple(shape), dtype, attrs)
        self.nodes.append(n)
        return Tensor(self, n.id)

    def tensor(self, shape, name, dtype="f32"):
        return self.add("Input", [], shape, dtype, name=name)

    def constant(self, value, dtype="f32"):
        return self.add("Constant", [], (), dtype, value=float(value))
```

```
    # ---- inspection
    def topo_order(self):
        """Node ids such that every input comes before its user (ids already are; we verif
        raise NotImplementedError("Graph.topo_order")

    def op_counts(self):
        raise NotImplementedError("Graph.op_counts")

    def semantics_table(self):
        """[(id, op, input ids, shape, dtype, note)] — the L1 deliverable."""
        raise NotImplementedError("Graph.semantics_table")

    def materialized_bytes(self, elem_bytes=4):
        """Bytes a naive executor would allocate: every non-view, non-input node gets a bu
        raise NotImplementedError("Graph.materialized_bytes")


@dataclass(frozen=True)
class Tensor:
    g: Graph
    id: int

    @property
    def shape(self):
        return self.g.nodes[self.id].shape

    @property
    def dtype(self):
        return self.g.nodes[self.id].dtype

    def output(self):
        self.g.outputs.append(self.id)
        return self

    # ---- movement
    def expand(self, axis, size):
        s = list(self.shape)
        s.insert(axis, size)
        return self.g.add("Expand", [self], s, self.dtype, axis=axis, size=size)

    def expand_to(self, shape):
        """Right-aligned numpy-style broadcast by inserting leading / size-1-free axes."""
        t = self
        for axis in range(len(shape) - len(self.shape)):
            t = t.expand(axis, shape[axis])
        assert t.shape == tuple(shape), (t.shape, shape)
        return t

    def permute(self, axes):
        return self.g.add("Permute", [self], [self.shape[a] for a in axes], self.dtype, ax
```

```
    # ---- binary (shapes must already agree, like Luminal: broadcasting is explicit)
    def _bin(self, op, other, dtype=None):
        if not isinstance(other, Tensor):
            other = self.g.constant(other).expand_to(self.shape) if self.shape else self.g
        assert self.shape == other.shape, f"{op}: {self.shape} vs {other.shape}"
        return self.g.add(op, [self, other], self.shape, dtype or self.dtype)

    def __add__(self, o):
        return self._bin("Add", o)

    def __mul__(self, o):
        return self._bin("Mul", o)

    def lt(self, o):
        return self._bin("LessThan", o, "bool")

    def select(self, a, b):
        """self is the boolean condition."""
        assert self.shape == a.shape == b.shape
        return self.g.add("Select", [self, a, b], a.shape, a.dtype)

    def sum(self, axis):
        s = list(self.shape)
        s.pop(axis)
        return self.g.add("ReduceSum", [self], s, self.dtype, axis=axis)

    # ---- high-level ops, spelled with primitives
    def matmul(self, w):
        """[M, K] @ [K, N]: both operands become [M, N, K] views, then Mul and ReduceSum(K
        raise NotImplementedError("Tensor.matmul")

    def relu(self):
        raise NotImplementedError("Tensor.relu")

    def exp(self):
        raise NotImplementedError("Tensor.exp")

    def exp2(self):
        return self.g.add("Exp2", [self], self.shape, self.dtype)


def build_mlp(M, K, N):
    """y = relu(x @ w + b). Returns (graph, x, w, b, y)."""
    raise NotImplementedError("build_mlp")


# ------------------------------------------------------------ reference runtime
def run(g, feeds):
```

```
    """Execute in topological order with numpy. feeds: {input name: array}. Returns {node
    raise NotImplementedError("run")
```

验收

```
./check.sh l1
```

tests/test_l1_graph.py 66 行 复制

```
import numpy as np
import pytest

from drill import load

l1 = load("l1_graph")


def test_recording_does_not_compute():
    g = l1.Graph()
    x = g.tensor((2, 3), "x")
    y = x + x
    assert isinstance(y, l1.Tensor) and len(g.nodes) == 2 and g.nodes[1].op == "Add"


def test_matmul_is_broadcast_mul_reduce():
    g = l1.Graph()
    a, b = g.tensor((5, 3), "a"), g.tensor((3, 7), "b")
    c = a.matmul(b).output()
    assert c.shape == (5, 7)
    ops = [g.nodes[i].op for i in g.topo_order()]
    assert ops.count("Mul") == 1 and ops.count("ReduceSum") == 1 and ops.count("Permute")
    mul = next(n for n in g.nodes if n.op == "Mul")
    assert mul.shape == (5, 7, 3)                    # the [M, N, K] product
    with pytest.raises(AssertionError):
        a.matmul(g.tensor((4, 7), "bad"))


def test_mlp_graph_shape_and_counts():
    g, x, w, b, y = l1.build_mlp(2, 3, 4)
    assert y.shape == (2, 4) and y.dtype == "f32"
    assert g.op_counts() == {"Input": 3, "Expand": 5, "Permute": 1, "Mul": 1, "ReduceSum":
                             "Constant": 1, "LessThan": 1, "Select": 1}
    table = g.semantics_table()
    lt = next(r for r in table if r[1] == "LessThan")
    assert lt[4] == "bool"
    assert any("view" in r[5] for r in table if r[1] == "Expand")
    assert all(set(g.nodes[r[0]].op for r in table) <= l1.PRIMITIVES for _ in [0])


def test_reference_runtime_matches_numpy():
    rng = np.random.default_rng(0)
    for M, K, N in [(2, 3, 4), (1, 1, 1), (7, 13, 5)]:
        g, x, w, b, y = l1.build_mlp(M, K, N)
        X, W, B = rng.standard_normal((M, K)), rng.standard_normal((K, N)), rng.standard_n
        v = l1.run(g, {"x": X, "w": W, "b": B})
        np.testing.assert_allclose(v[y.id], np.maximum(X @ W + B, 0), rtol=1e-5, atol=1e-5


def test_relu_is_select_and_exp_is_exp2():
```

```
    g = l1.Graph()
    x = g.tensor((3,), "x")
    r = x.relu().output()
    assert g.nodes[r.id].op == "Select"
    e = x.exp().output()
    assert g.nodes[e.id].op == "Exp2"
    v = l1.run(g, {"x": np.array([-1.0, 0.0, 2.0])})
    np.testing.assert_allclose(v[r.id], [0, 0, 2])
    np.testing.assert_allclose(v[e.id], np.exp([-1.0, 0.0, 2.0]), rtol=1e-6)


def test_naive_execution_materializes_the_product():
    g, *_ = l1.build_mlp(64, 256, 64)
    # Mul [64, 64, 256] alone is 4 MiB of fp32 — that's what fusion must avoid
    assert g.materialized_bytes() >= 64 * 64 * 256 * 4
    assert g.materialized_bytes() == (64 * 64 * 256 + 4 * 64 * 64) * 4
```

提示

L1 matmul 的两个 view 方向（轻度剧透） 收起

A 是 `[M, K]`，在第 1 维插入 N：`[M, N, K]`。B 是 `[K, N]`，先转置成 `[N, K]`，再在第 0 维插入 M：`[M, N, K]`。两者逐元素相乘，沿最后一维求和得到 `[M, N]`。

L2 run 里的 Expand 拆解与常见坑（中度剧透） 收起

`np.broadcast_to(np.expand_dims(a, axis), node.shape)`：先插一个长度 1 的轴，再广播成目标形状。`broadcast_to` 返回的是 view，不复制数据，和 Luminal 的语义一致。

L3 materialized_bytes 的数 几乎是答案（重度剧透） 收起

`build_mlp(64, 256, 64)`：Mul 是 64·64·256 个元素，ReduceSum、Add、LessThan、Select 各 64·64 个，乘 4 字节，测试里写成 `(64·64·256 + 4·64·64)·4`。

参考解

解 solutions/l1_graph.py 先自己写完、跑过测试再看 收起 ▾

solutions/l1_graph.py 226 行 复制

```
"""L1 · Graph semantics: y = relu(x @ w + b) as a Luminal-style primitive graph — refe

Luminal records every tensor operation into a static DAG instead of running it (lazy,
time). High-level ops are *spelled* with a small primitive set (README: "Everything in
boils down to 15 primitive ops"; src/logical/ir.rs `LogicalOp` has the current full li

  matmul(a, b)  = broadcast both to [M, N, K]  ->  Mul  ->  ReduceSum(K)     (src/fron
  relu(x)       = maximum(x, 0) = Select(LessThan(x, 0), 0, x)                (src/fro
  exp(x)        = Exp2(x * log2(e))                                           (src/fro
  a + b[N]      = Add(a, Expand(b))  — broadcasting is a *view*, not a copy

This file is a ~150-line Python model of that design: a Graph of nodes with shapes and
a GraphTensor API that records, a node-semantics table, and a reference runtime (numpy
executes the DAG in topological order — the analogue of Luminal's ReferenceRuntime.
"""
from dataclasses import dataclass, field

import numpy as np

PRIMITIVES = {"Input", "Constant", "Add", "Mul", "Recip", "Exp2", "Log2", "Sqrt", "Sin
              "Select", "ReduceSum", "ReduceMax", "Expand", "Permute", "Cast"}
MOVEMENT = {"Expand", "Permute"}          # views: change indexing, move no data


@dataclass
class Node:
    id: int
    op: str
    inputs: list
    shape: tuple
    dtype: str = "f32"
    attrs: dict = field(default_factory=dict)


class Graph:
    def __init__(self):
        self.nodes = []
        self.outputs = []

    def add(self, op, inputs, shape, dtype="f32", **attrs):
        assert op in PRIMITIVES, op
        n = Node(len(self.nodes), op, [t.id for t in inputs], tuple(shape), dtype, att
        self.nodes.append(n)
        return Tensor(self, n.id)

    def tensor(self, shape, name, dtype="f32"):
        return self.add("Input", [], shape, dtype, name=name)

    def constant(self, value, dtype="f32"):
        return self.add("Constant", [], (), dtype, value=float(value))
```

```
    # ---- inspection
    def topo_order(self):
        """Node ids such that every input comes before its user (ids already are; we v
        seen, order = set(), []

        def visit(i):
            if i in seen:
                return
            seen.add(i)
            for j in self.nodes[i].inputs:
                visit(j)
            order.append(i)
        for o in (self.outputs or [len(self.nodes) - 1]):
            visit(o)
        return order

    def op_counts(self):
        c = {}
        for i in self.topo_order():
            c[self.nodes[i].op] = c.get(self.nodes[i].op, 0) + 1
        return c

    def semantics_table(self):
        """[(id, op, input ids, shape, dtype, note)] — the L1 deliverable."""
        rows = []
        for i in self.topo_order():
            n = self.nodes[i]
            note = ""
            if n.op == "Expand":
                note = f"broadcast axis {n.attrs['axis']} to {n.attrs['size']} (view)"
            elif n.op == "Permute":
                note = f"axes {n.attrs['axes']} (view)"
            elif n.op in ("ReduceSum", "ReduceMax"):
                note = f"reduce axis {n.attrs['axis']}"
            elif n.op == "Input":
                note = n.attrs["name"]
            elif n.op == "Constant":
                note = repr(n.attrs["value"])
            rows.append((n.id, n.op, list(n.inputs), n.shape, n.dtype, note))
        return rows

    def materialized_bytes(self, elem_bytes=4):
        """Bytes a naive executor would allocate: every non-view, non-input node gets
        total = 0
        for i in self.topo_order():
            n = self.nodes[i]
            if n.op not in MOVEMENT and n.op not in ("Input", "Constant"):
                total += int(np.prod(n.shape)) * elem_bytes
        return total
```

```python
@dataclass(frozen=True)
class Tensor:
    g: Graph
    id: int

    @property
    def shape(self):
        return self.g.nodes[self.id].shape

    @property
    def dtype(self):
        return self.g.nodes[self.id].dtype

    def output(self):
        self.g.outputs.append(self.id)
        return self

    # ---- movement
    def expand(self, axis, size):
        s = list(self.shape)
        s.insert(axis, size)
        return self.g.add("Expand", [self], s, self.dtype, axis=axis, size=size)

    def expand_to(self, shape):
        """Right-aligned numpy-style broadcast by inserting leading / size-1-free axes
        t = self
        for axis in range(len(shape) - len(self.shape)):
            t = t.expand(axis, shape[axis])
        assert t.shape == tuple(shape), (t.shape, shape)
        return t

    def permute(self, axes):
        return self.g.add("Permute", [self], [self.shape[a] for a in axes], self.dtype

    # ---- binary (shapes must already agree, like Luminal: broadcasting is explicit)
    def _bin(self, op, other, dtype=None):
        if not isinstance(other, Tensor):
            other = self.g.constant(other).expand_to(self.shape) if self.shape else se
        assert self.shape == other.shape, f"{op}: {self.shape} vs {other.shape}"
        return self.g.add(op, [self, other], self.shape, dtype or self.dtype)

    def __add__(self, o):
        return self._bin("Add", o)

    def __mul__(self, o):
        return self._bin("Mul", o)

    def lt(self, o):
        return self._bin("LessThan", o, "bool")
```

```
    def select(self, a, b):
        """self is the boolean condition."""
        assert self.shape == a.shape == b.shape
        return self.g.add("Select", [self, a, b], a.shape, a.dtype)

    def sum(self, axis):
        s = list(self.shape)
        s.pop(axis)
        return self.g.add("ReduceSum", [self], s, self.dtype, axis=axis)

    # ---- high-level ops, spelled with primitives
    def matmul(self, w):
        """[M, K] @ [K, N]: both operands become [M, N, K] views, then Mul and ReduceS
        (m, k), (k2, n) = self.shape, w.shape
        assert k == k2, "contract dim mismatch"
        a = self.expand(1, n)                       # [M, N, K]
        b = w.permute([1, 0]).expand(0, m)          # [K, N] -> [N, K] -> [M, N, K]
        return (a * b).sum(2)

    def relu(self):
        zero = self.g.constant(0.0).expand_to(self.shape)
        return self.lt(zero).select(zero, self)

    def exp(self):
        return (self * float(np.log2(np.e))).exp2()

    def exp2(self):
        return self.g.add("Exp2", [self], self.shape, self.dtype)


def build_mlp(M, K, N):
    """y = relu(x @ w + b). Returns (graph, x, w, b, y)."""
    g = Graph()
    x, w, b = g.tensor((M, K), "x"), g.tensor((K, N), "w"), g.tensor((N,), "b")
    y = (x.matmul(w) + b.expand_to((M, N))).relu().output()
    return g, x, w, b, y


# ------------------------------------------------------------ reference runtime
def run(g, feeds):
    """Execute in topological order with numpy. feeds: {input name: array}. Returns {n
    vals = {}
    for i in g.topo_order():
        n = g.nodes[i]
        a = [vals[j] for j in n.inputs]
        if n.op == "Input":
            v = np.asarray(feeds[n.attrs["name"]], np.float32)
            assert v.shape == n.shape, (n.attrs["name"], v.shape, n.shape)
        elif n.op == "Constant":
            v = np.float32(n.attrs["value"])
        elif n.op == "Expand":
```

```
            v = np.broadcast_to(np.expand_dims(a[0], n.attrs["axis"]), n.shape)
        elif n.op == "Permute":
            v = np.transpose(a[0], n.attrs["axes"])
        elif n.op == "Add":
            v = a[0] + a[1]
        elif n.op == "Mul":
            v = a[0] * a[1]
        elif n.op == "LessThan":
            v = a[0] < a[1]
        elif n.op == "Select":
            v = np.where(a[0], a[1], a[2])
        elif n.op == "ReduceSum":
            v = a[0].sum(axis=n.attrs["axis"], dtype=np.float32)
        elif n.op == "ReduceMax":
            v = a[0].max(axis=n.attrs["axis"])
        elif n.op == "Exp2":
            v = np.exp2(a[0])
        elif n.op == "Recip":
            v = 1 / a[0]
        else:
            raise NotImplementedError(n.op)
        vals[i] = v
    return vals
```

在 Luminal 里：看真正的图

```
use luminal::prelude::*;
let mut cx = Graph::new();
let x = cx.tensor(vec![2, 3], DType::F32);
let w = cx.tensor(vec![3, 4], DType::F32);
let b = cx.tensor(vec![4], DType::F32);
let y = (x.matmul(w) + b.expand_rhs(vec![2, 4])).relu().output();
// 编译前后都可以导出可视化：src/visualization.rs；编译时 EGGLOG_DEBUG=1 会把 egglog 程序也打印出来
```

对照你的语义表：真实图里每个操作数的 broadcast 是一个 view map 节点（matmul.rs 调用 `luminal › record_view_map`，注释说“Record ONE view apply ... no intermediate movement scaffolding”），而不是一串 Expand / Permute；`relu` 会多出 NaN 覆盖那一个 `Select`。这些差异正好说明：primitive 集合的设计（README 的 15 个 vs `LogicalOp` 的完整列表）本身也在演进，读代码时以 pin 住的 commit 为准。

▼ 自测：为什么 broadcast 不复制数据也能被“搜索”？

broadcast 只是索引映射：`b_expanded[m, n] = b[n]`。它在图里是一个节点，所以 e-graph 可以对它做改写（比如把 `Add(ReduceSum(Mul(...)), Expand(b))` 识别成“带 bias 的 GEMM”，匹配到 cuBLASLt 的 `EPILOGUE_BIAS`）；而它在执行时可以不分配任何内存。把“改变索引”和“搬数据”分开，是 Luminal 能自动发现 fusion 的前提。

双主线 · Track L · Luminal（graph compiler） · 第 11 题

L2 E-graph 从零写：union-find、rebuild、rewrite、saturation、extraction

Luminal 的搜索空间是一个 e-graph：所有等价程序同时存在，改写只增不删。这道题从零写一个 e-graph（hashcons、union-find、congruence rebuild、e-matching、equality saturation、带预算的停止、基于代价的 extraction），用 (a+b)+c、a+(b+c)、b+a 手算验证，再看 Luminal 为什么只对整数开放交换律。

⏱ 75 min | 难度 难 | 你写 `mine/l2_egraph.py` | 验收 `./check.sh l2`

参考解 `solutions/l2_egraph.py` | 📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 浮点下哪些 rewrite 不安全
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. egglog 小实验（roadmap 的交付物）

背景

roadmap 的 L2：“用 (a+b)+c、a+(b+c)、a+b、b+a 画 e-class；讨论浮点下哪些 rewrite 不一定安全。交付：e-graph 草图 + 一个 egglog 小实验。”

Luminal 的规则写在 egglog 里：核心规则在 `luminal › egglog_preamble.egg`，每个 primitive 自己的规则在 `src/logical/op/<op>/*.egg`。例如 `luminal › add/rewrite.egg` 开头就是一段值得背下来的注释：

```
; Encoding bits are observable. Float operand permutation can change a NaN
; payload, so unqualified floating commutativity is not a bit-exact equality.
; Retain the law for exact integer encodings only (fail closed on unknown dtype).
(rule ((= ?out (LogicalAdd ?lhs ?rhs)) (= (dtype-of ?out) (Int)))
 ((union ?out (LogicalAdd ?rhs ?lhs)))
 :ruleset main_ruleset :name "add/rewrite: commutativity, Int")
```

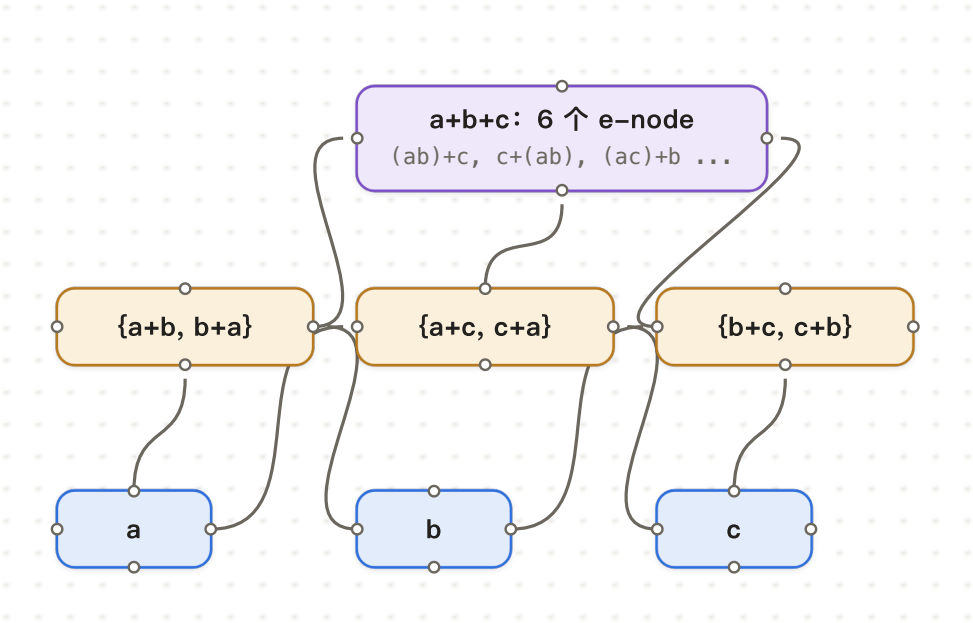
连 `a + b → b + a` 都只对整数开放——更不用说结合律。

源码与资料

egg 论文：Fast and Extensible Equality Saturation · egglog 论文 · `egglog › egglog README` · `egglogpy › egglog-python README` · `luminal › preamble 里的整数恒等式规则` ·

要用到的知识

(a+b)+c 饱和之后的 7 个 e-class



3 个叶子 + 3 个两项和（每个 2 个 e-node）+ 1 个三项和（3 种“先加哪两个”× 2 种左右顺序 = 6 个 e-node），共 7 个 e-class、15 个 e-node。n 项时是 $2^n - 1$ 个 e-class、$3^n - 2^{n+1} + 1 + n$ 个 e-node：4 项 54，5 项 185，6 项 608。

部件	做什么
hashcons	规范化后的 e-node → e-class；同一个 `("+", c1, c2)` 只存一份
union-find	`find(c)` 给出代表元；`union` 合并两个类
rebuild	合并之后，子节点的代表元变了，原本不同的 e-node 可能变得相同（`f(a)` 和 `f(b)` 在 `a = b` 之后），要把它们所在的类也合并（congruence）
e-matching	在某个类里找所有能匹配模式 `("+", ("+", "?x", "?y"), "?z")` 的替换
saturation	每轮：先收集所有规则的所有匹配，再统一 instantiate + union，然后 rebuild；一轮下来类数和节点数都没变 → 饱和
budget	`iter_limit`、`node_limit`：没饱和也要停（`"iter_limit"` / `"node_limit"`）
extraction	自底向上求每个类的最小代价（要迭代到不动点，因为类可以有环：`x·1 = x`）

类比 · 族谱和同名的人

> e-class 像“同一个人的所有称呼”：张三、老张、张经理都指向同一个人（union-find 记住“谁是谁”）。hashcons 保证“张经理的儿子”这个称呼只登记一次。rebuild 处理连锁反应：一旦发现“老张”和“张经理”是同一个人，那“老张的儿子”和“张经理的儿子”也一定是同一个人。extraction 就是在所有称呼里挑最短的那个写进正式文件。

浮点下哪些 rewrite 不安全

规则	实数上	浮点上
`a + b → b + a`	成立	数值上成立，但 NaN 的 payload 可能不同 → Luminal 只对整数开放
`(a + b) + c → a + (b + c)`	成立	**不成立**：`(1e8 + -1e8) + 1 = 1`，`1e8 + (-1e8 + 1) = 0`（测试 `test_reassociation_is_not_bit_exact`）
`x · 1 → x`、`x + 0 → x`	成立	`-0.0 + 0.0 = +0.0`，所以 `x + 0 → x` 对 `x = -0.0` 改变了符号位
`exp2(log2(x)) → x`	x > 0 时成立	x ⩽ 0 时左边是 NaN，右边不是
`a·b + a·c → a·(b + c)`	成立	舍入次数不同；也可能溢出 / 不溢出

题目要求

在 `mine/l2_egraph.py` 里实现（`find`、`canon`、`add_enode`、`add`、`num_classes`、`num_nodes`、规则集、两个代价函数和 `evaluate` 已给出）：

方法	要点
`union(a, b)`	小类挂到大类下面，合并 e-node 集合；返回是否真的合并了
`rebuild()`	重新规范化所有 e-node；两个类里出现了同一个 e-node 就合并；直到不再变化；重建 hashcons
`equiv(t1, t2)`	两个项加进来，比较代表元
`ematch(pattern, cid, subst)`	变量（`?x`）绑定或检查一致；叶子比对 `(leaf,)`；复合模式对类里每个同名同元数的 e-node 递归
`instantiate(pattern, subst)`	按替换把模式加进图里
`run(rules, iter_limit, node_limit)`	返回 `{"stop": "saturated" / "iter_limit" / "node_limit", "history": [...]}`
`extract(cid, cost)`	`(最小代价, 项)`；`cost(op, child_costs)`

题干

mine/l2_egraph.py 120 行 复制

```
"""L2 · E-graphs and equality saturation from scratch — your implementation (run ./check.s

Luminal's search space is an e-graph built by egglog (src/egglog_core/egglog_preamble.egg
the rules). An e-graph stores *many equivalent programs at once*:

  e-node   an operator applied to e-classes      ("+", c1, c2)
  e-class  a set of e-nodes known to be equal    {a+b, b+a}
  rewrite  lhs -> rhs: when lhs matches in class C, add rhs and union it into C (nothing i
  rebuild  restore congruence: if a ≡ b then f(a) ≡ f(b)
  saturate apply all rules until nothing new appears (or a node / iteration budget runs ou
  extract  pick one e-node per class to minimize a cost -> one concrete program

Terms are tuples: ("+", x, y), leaves are strings or numbers. Pattern variables start with
"""
from itertools import count


class EGraph:
    def __init__(self):
        self.parent = {}            # union-find over class ids
        self.classes = {}           # canonical id -> set of canonical e-nodes
        self.hashcons = {}          # canonical e-node -> class id
        self._ids = count()
        self.pending = []

    # ------------------------------------------------------------ union-find
    def find(self, a):
        while self.parent[a] != a:
            self.parent[a] = self.parent[self.parent[a]]
            a = self.parent[a]
        return a

    def canon(self, enode):
        op, *kids = enode
        return (op, *(self.find(k) for k in kids))

    # ------------------------------------------------------------ building
    def add_enode(self, enode):
        enode = self.canon(enode)
        if enode in self.hashcons:
            return self.find(self.hashcons[enode])
        cid = next(self._ids)
        self.parent[cid] = cid
        self.classes[cid] = {enode}
        self.hashcons[enode] = cid
        return cid

    def add(self, term):
        if isinstance(term, tuple):
            return self.add_enode((term[0], *(self.add(t) for t in term[1:])))
```

```
        return self.add_enode((term,))

    def union(self, a, b):
        raise NotImplementedError("EGraph.union")

    def rebuild(self):
        """Re-canonicalize every e-node; merge classes whose e-nodes became identical (con
        raise NotImplementedError("EGraph.rebuild")

    # ------------------------------------------------------------ queries
    def equiv(self, t1, t2):
        raise NotImplementedError("EGraph.equiv")

    def num_classes(self):
        return len(self.classes)

    def num_nodes(self):
        return sum(len(v) for v in self.classes.values())

    # ------------------------------------------------------------ e-matching
    def ematch(self, pattern, cid, subst=None):
        """All substitutions {?var: class id} under which `pattern` matches something in c
        raise NotImplementedError("EGraph.ematch")

    def instantiate(self, pattern, subst):
        raise NotImplementedError("EGraph.instantiate")

    # ------------------------------------------------------------ saturation
    def run(self, rules, iter_limit=20, node_limit=10_000):
        """Equality saturation. rules: [(name, lhs, rhs)]. Returns a report dict."""
        raise NotImplementedError("EGraph.run")

    # ------------------------------------------------------------ extraction
    def extract(self, cid, cost):
        """Cheapest term in class cid. cost(op, child_costs) -> number. Bottom-up fixpoint
        (classes may be cyclic, e.g. x*1 = x)."""
        raise NotImplementedError("EGraph.extract")


# ------------------------------------------------------------ rule sets
COMM_ASSOC = [
    ("comm-add", ("+", "?a", "?b"), ("+", "?b", "?a")),
    ("assoc-add", ("+", ("+", "?a", "?b"), "?c"), ("+", "?a", ("+", "?b", "?c"))),
]
ALGEBRA = COMM_ASSOC + [
    ("mul-one", ("*", "?a", 1), "?a"),
    ("add-zero", ("+", "?a", 0), "?a"),
    ("factor", ("+", ("*", "?a", "?b"), ("*", "?a", "?c")), ("*", "?a", ("+", "?b", "?c"))
    ("exp2-log2", ("exp2", ("log2", "?a")), "?a"),
]
```

```
def ast_size(op, kids):
    return 1 + sum(kids)


def op_cost(op, kids):
    """Rough kernel cost: multiplies cost more than adds; leaves are free."""
    return {"*": 4, "+": 1, "exp2": 8, "log2": 8}.get(op, 0) + sum(kids)


def evaluate(term, env, dtype=float):
    """Evaluate a term with Python floats or numpy scalars — to show which rewrites change
    import math
    if isinstance(term, tuple):
        op, *k = term
        v = [evaluate(t, env, dtype) for t in k]
        return {"+": lambda a, b: dtype(a + b), "*": lambda a, b: dtype(a * b),
                "exp2": lambda a: dtype(2.0 ** a), "log2": lambda a: dtype(math.log2(a))}[
    return dtype(env[term]) if isinstance(term, str) else dtype(term)
```

验收

```
./check.sh l2
```

tests/test_l2_egraph.py 90 行 复制

```
import numpy as np

from drill import load

l2 = load("l2_egraph")
E = l2.EGraph


def test_hashcons_shares_subterms():
    g = E()
    a = g.add(("+", "a", "b"))
    b = g.add(("+", "a", "b"))
    assert a == b and g.num_classes() == 3 and g.num_nodes() == 3


def test_union_and_congruence():
    g = E()
    fa, fb = g.add(("f", "a")), g.add(("f", "b"))
    assert g.find(fa) != g.find(fb)
    g.union(g.add("a"), g.add("b"))
    g.rebuild()
    assert g.find(fa) == g.find(fb)          # a = b  =>  f(a) = f(b)


def test_ematch():
    g = E()
    root = g.add(("+", ("+", "a", "b"), "c"))
    subs = g.ematch(("+", ("+", "?x", "?y"), "?z"), root)
    assert len(subs) == 1 and g.find(subs[0]["?x"]) == g.find(g.add("a"))
    assert g.ematch(("+", "?x", "?x"), root) == []
    assert len(g.ematch(("+", "?x", "?x"), g.add(("+", "a", "a")))) == 1


def test_three_term_saturation_counts():
    g = E()
    g.add(("+", ("+", "a", "b"), "c"))
    rep = g.run(l2.COMM_ASSOC)
    assert rep["stop"] == "saturated"
    # classes: a, b, c, a+b, a+c, b+c, a+b+c ; e-nodes: 3 leaves + 2 per pair + 6 at the t
    assert g.num_classes() == 7 and g.num_nodes() == 15
    assert g.equiv(("+", "a", ("+", "b", "c")), ("+", ("+", "a", "b"), "c"))
    assert g.equiv(("+", ("+", "c", "a"), "b"), ("+", "b", ("+", "c", "a")))
    assert not g.equiv(("+", "a", "b"), ("+", "a", "c"))


def test_blowup_is_exponential():
    sizes = []
    for n in (3, 4, 5, 6):
        g = E()
        t = "x0"
```

```
        for i in range(1, n):
            t = ("+", t, f"x{i}")
        g.add(t)
        assert g.run(l2.COMM_ASSOC, iter_limit=30, node_limit=10 ** 6)["stop"] == "saturat
        assert g.num_classes() == 2 ** n - 1
        assert g.num_nodes() == 3 ** n - 2 ** (n + 1) + 1 + n
        sizes.append(g.num_nodes())
    assert sizes == [15, 54, 185, 608]


def test_budgets_stop_the_search():
    g = E()
    t = "x0"
    for i in range(1, 8):
        t = ("+", t, f"x{i}")
    g.add(t)
    assert g.run(l2.COMM_ASSOC, iter_limit=2)["stop"] == "iter_limit"
    g2 = E()
    g2.add(t)
    assert g2.run(l2.COMM_ASSOC, iter_limit=50, node_limit=500)["stop"] == "node_limit"


def test_extraction_picks_the_cheaper_program():
    g = E()
    root = g.add(("+", ("*", "x", "y"), ("*", "x", "z")))
    g.run(l2.ALGEBRA)
    cost, term = g.extract(root, l2.op_cost)
    assert cost == 5 and term[0] == "*" and g.equiv(term, ("*", "x", ("+", "y", "z")))
    g = E()
    root = g.add(("*", ("exp2", ("log2", "x")), 1))
    g.run(l2.ALGEBRA)
    assert g.extract(root, l2.op_cost) == (0, "x")
    assert g.extract(root, l2.ast_size) == (1, "x")


def test_reassociation_is_not_bit_exact():
    env = {"a": 1e8, "b": -1e8, "c": 1.0}
    left = l2.evaluate(("+", ("+", "a", "b"), "c"), env, np.float32)
    right = l2.evaluate(("+", "a", ("+", "b", "c")), env, np.float32)
    assert left == 1.0 and right == 0.0      # equal in the e-graph, different in fp32
```

提示

L1 先过 congruence 测试 方向（轻度剧透） 收起

`union` 之后不要立刻修 hashcons；全部 union 完再调一次 `rebuild()`。rebuild 的循环：把每个类的 e-node 集合重新 `canon`；用一个新字典 `enode → 类`，遇到已经属于另一个类的同一个 e-node 就 `union`，并记下“这一轮有变化”。

L2 一轮饱和的顺序 拆解与常见坑（中度剧透） 收起

先对所有规则、所有类收集匹配（`matches`），再统一 `instantiate` + `union`，最后 `rebuild`。边匹配边修改会让同一轮的结果依赖遍历顺序。停止条件：这一轮 `unions == 0` 且类数、节点数都和这一轮开始时一样。

L3 extraction 的不动点 几乎是答案（重度剧透） 收起

`best = {}`；反复扫描所有类的所有 e-node：子类都已有代价时，算出这个 e-node 的代价，比当前更小就更新、并标记 changed。没有变化时停止。最后从根类开始，按 `best` 里记下的 e-node 递归构造项。

参考解

解 solutions/l2_egraph.py 先自己写完、跑过测试再看 收起

solutions/l2_egraph.py 201 行 复制

```
"""L2 · E-graphs and equality saturation from scratch — reference solution.

Luminal's search space is an e-graph built by egglog (src/egglog_core/egglog_preamble.
the rules). An e-graph stores *many equivalent programs at once*:

  e-node   an operator applied to e-classes      ("+", c1, c2)
  e-class  a set of e-nodes known to be equal    {a+b, b+a}
  rewrite  lhs -> rhs: when lhs matches in class C, add rhs and union it into C (nothi
  rebuild  restore congruence: if a ≡ b then f(a) ≡ f(b)
  saturate apply all rules until nothing new appears (or a node / iteration budget run
  extract  pick one e-node per class to minimize a cost -> one concrete program

Terms are tuples: ("+", x, y), leaves are strings or numbers. Pattern variables start
"""
from itertools import count


class EGraph:
    def __init__(self):
        self.parent = {}            # union-find over class ids
        self.classes = {}           # canonical id -> set of canonical e-nodes
        self.hashcons = {}          # canonical e-node -> class id
        self._ids = count()
        self.pending = []

    # ------------------------------------------------------------ union-find
    def find(self, a):
        while self.parent[a] != a:
            self.parent[a] = self.parent[self.parent[a]]
            a = self.parent[a]
        return a

    def canon(self, enode):
        op, *kids = enode
        return (op, *(self.find(k) for k in kids))

    # ------------------------------------------------------------ building
    def add_enode(self, enode):
        enode = self.canon(enode)
        if enode in self.hashcons:
            return self.find(self.hashcons[enode])
        cid = next(self._ids)
        self.parent[cid] = cid
        self.classes[cid] = {enode}
        self.hashcons[enode] = cid
        return cid

    def add(self, term):
        if isinstance(term, tuple):
            return self.add_enode((term[0], *(self.add(t) for t in term[1:])))
```

```
        return self.add_enode((term,))

    def union(self, a, b):
        a, b = self.find(a), self.find(b)
        if a == b:
            return False
        if len(self.classes[a]) < len(self.classes[b]):
            a, b = b, a
        self.parent[b] = a
        self.classes[a] |= self.classes.pop(b)
        self.pending.append(a)
        return True

    def rebuild(self):
        """Re-canonicalize every e-node; merge classes whose e-nodes became identical
        changed = True
        while changed:
            changed = False
            new_hash = {}
            for cid in list(self.classes):
                cid = self.find(cid)
                if cid not in self.classes:
                    continue
                self.classes[cid] = {self.canon(n) for n in self.classes[cid]}
            for cid, nodes in list(self.classes.items()):
                for n in nodes:
                    other = new_hash.get(n)
                    if other is not None and self.find(other) != self.find(cid):
                        self.union(other, cid)
                        changed = True
                    new_hash[n] = self.find(cid)
            self.hashcons = {self.canon(n): self.find(c) for n, c in new_hash.items()}
        self.pending.clear()

    # ------------------------------------------------------------ queries
    def equiv(self, t1, t2):
        return self.find(self.add(t1)) == self.find(self.add(t2))

    def num_classes(self):
        return len(self.classes)

    def num_nodes(self):
        return sum(len(v) for v in self.classes.values())

    # ------------------------------------------------------------ e-matching
    def ematch(self, pattern, cid, subst=None):
        """All substitutions {?var: class id} under which `pattern` matches something
        subst = dict(subst or {})
        cid = self.find(cid)
        if isinstance(pattern, str) and pattern.startswith("?"):
            if pattern in subst:
```

```
                return [subst] if self.find(subst[pattern]) == cid else []
            subst[pattern] = cid
            return [subst]
        if not isinstance(pattern, tuple):
            return [subst] if (pattern,) in self.classes[cid] else []
        out = []
        op, *pkids = pattern
        for n in self.classes[cid]:
            if n[0] != op or len(n) - 1 != len(pkids):
                continue
            partial = [subst]
            for p, k in zip(pkids, n[1:]):
                partial = [s2 for s in partial for s2 in self.ematch(p, k, s)]
            out.extend(partial)
        return out

    def instantiate(self, pattern, subst):
        if isinstance(pattern, str) and pattern.startswith("?"):
            return subst[pattern]
        if not isinstance(pattern, tuple):
            return self.add_enode((pattern,))
        return self.add_enode((pattern[0], *(self.instantiate(p, subst) for p in patte

    # ------------------------------------------------------------ saturation
    def run(self, rules, iter_limit=20, node_limit=10_000):
        """Equality saturation. rules: [(name, lhs, rhs)]. Returns a report dict."""
        history = []
        for it in range(iter_limit):
            before = (self.num_classes(), self.num_nodes())
            matches = []
            for name, lhs, rhs in rules:
                for cid in list(self.classes):
                    for s in self.ematch(lhs, cid):
                        matches.append((name, cid, rhs, s))
            unions = 0
            for name, cid, rhs, s in matches:
                unions += self.union(cid, self.instantiate(rhs, s))
            self.rebuild()
            history.append({"iter": it, "matches": len(matches), "unions": unions,
                            "classes": self.num_classes(), "nodes": self.num_nodes()})
            if unions == 0 and (self.num_classes(), self.num_nodes()) == before:
                return {"stop": "saturated", "history": history}
            if self.num_nodes() > node_limit:
                return {"stop": "node_limit", "history": history}
        return {"stop": "iter_limit", "history": history}

    # ------------------------------------------------------------ extraction
    def extract(self, cid, cost):
        """Cheapest term in class cid. cost(op, child_costs) -> number. Bottom-up fixp
        (classes may be cyclic, e.g. x*1 = x)."""
        best = {}
```

```
        changed = True
        while changed:
            changed = False
            for c, nodes in self.classes.items():
                for n in nodes:
                    if all(self.find(k) in best for k in n[1:]):
                        cc = cost(n[0], [best[self.find(k)][0] for k in n[1:]])
                        if c not in best or cc < best[c][0]:
                            best[c] = (cc, n)
                            changed = True

        def build(c):
            _, n = best[self.find(c)]
            return n[0] if len(n) == 1 else (n[0], *(build(k) for k in n[1:]))
        c = self.find(cid)
        return best[c][0], build(c)


# ---------------------------------------------------------------- rule sets
COMM_ASSOC = [
    ("comm-add", ("+", "?a", "?b"), ("+", "?b", "?a")),
    ("assoc-add", ("+", ("+", "?a", "?b"), "?c"), ("+", "?a", ("+", "?b", "?c"))),
]
ALGEBRA = COMM_ASSOC + [
    ("mul-one", ("*", "?a", 1), "?a"),
    ("add-zero", ("+", "?a", 0), "?a"),
    ("factor", ("+", ("*", "?a", "?b"), ("*", "?a", "?c")), ("*", "?a", ("+", "?b", "?
    ("exp2-log2", ("exp2", ("log2", "?a")), "?a"),
]


def ast_size(op, kids):
    return 1 + sum(kids)


def op_cost(op, kids):
    """Rough kernel cost: multiplies cost more than adds; leaves are free."""
    return {"*": 4, "+": 1, "exp2": 8, "log2": 8}.get(op, 0) + sum(kids)


def evaluate(term, env, dtype=float):
    """Evaluate a term with Python floats or numpy scalars — to show which rewrites ch
    import math
    if isinstance(term, tuple):
        op, *k = term
        v = [evaluate(t, env, dtype) for t in k]
        return {"+": lambda a, b: dtype(a + b), "*": lambda a, b: dtype(a * b),
                "exp2": lambda a: dtype(2.0 ** a), "log2": lambda a: dtype(math.log2(a
    return dtype(env[term]) if isinstance(term, str) else dtype(term)
```

egglog 小实验（roadmap 的交付物）

仓库里的 examples/l2_egglog_experiment.py 用 egglog 的 Python 绑定做同一件事：声明 `Num` 类型和 `+`，注册交换律、结合律，`run(10)`，再 `check(eq(left).to(right))`。

```
.venv/bin/pip install egglog && .venv/bin/python examples/l2_egglog_experiment.py
# equal after saturation
# Num.var("a") + Num.var("b") + Num.var("c")
# 4 iterations
```

4 轮之后饱和，和你自己的 e-graph 的 `history` 长度一样。接着试两件事：(1) 把项数加到 6，观察 egglog 的运行时间和你的 `num_nodes()` 是否同样按 3^n 增长；(2) 仿照 Luminal 的 `add/rewrite.egg`，给规则加一个条件（只在 dtype 是整数时触发），体会“fail closed”。

▼ 自测：既然浮点结合律不成立，Luminal 怎么发现 split-K 之类需要重排归约的优化？

不靠一条通用的“浮点结合律”规则，而是：(1) 把归约本身做成带语义的节点（`ReduceSum`、`UnspecifiedOrderScanSum`——名字里就写着“求值顺序未指定”），让“顺序可以变”成为这个 op 的契约，而不是任意 `Add` 链的性质；(2) 由用户在图里显式写出精度（matmul 不自动 cast）；(3) 用 L5 那样的 Reference vs device 对拍，在容差内验证结果。规则的合法性在图的语义层面保证，而不是对所有浮点表达式开放。

双主线 · Track L · Luminal（graph compiler） · 第 12 题

L3 Search：genome、GA、实测排名、finalists 与代理成本

饱和之后 e-graph 里有成千上万个等价程序，选哪一个？Luminal 的 CUDA backend 不信成本公式，而是编译、运行、计时。这道题实现它的搜索骨架：genome（每个 e-class 选一个 e-node）、遗传算法、按实测纳秒的 rank_insert、带验证的 finalists 遍历；再把“代理成本”和“实测”放在一起，量出代理的相关性和它会犯的错。

50 min | 难度 中 | 你写 `mine/l3_search.py` | 验收 `./check.sh l3` | 参考解 `solutions/l3_search.py` | 📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 题目要求
5. 题干
6. 验收
7. 提示
8. 参考解
9. 在 Luminal 里：search trade-off 报告

背景

roadmap 的 L3：“对一个小 graph 改变可用 rewrite / 搜索预算 / timeout，记录候选数、搜索时间、最终 latency。交付：search trade-off 报告。”读源码要区分三样东西：**cost proxy**、**compile / search time**、**steady-state runtime**。

真实代码里的对应物：

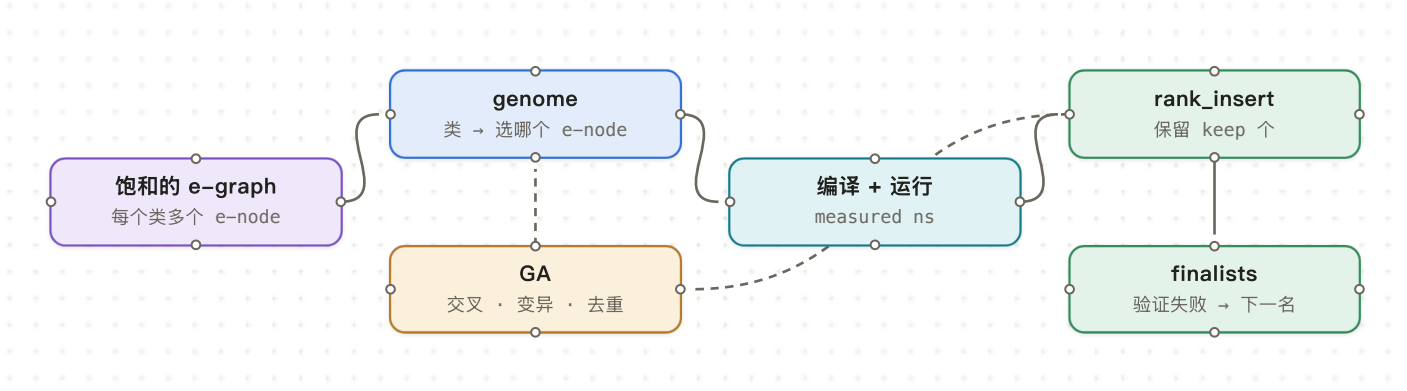
- `luminal › Genome`：“per-LayoutTensor-class producer selection”，一个 `ClassId → ProducerChoice` 的映射。
- `luminal › rank_insert`：只保留 `keep` 个最快的 `(nanos, genome)`，按纳秒升序。
- `luminal › PendingFinalist`：“1-BASED rank in the GA's ordering”，metric 是“measured nanoseconds”；一个 finalist 被拒绝（例如 arena 超过设备内存预算），就走到下一名。
- `luminal › search_implementations`：整个流程，开头就检查“candidate search requires the `device` feature and a CUDA GPU”。

源码与资料

`luminal › CompileOptions` · README：`SEARCH_LOG`、`LUMINAL_SEARCH_SEED`、`SEARCH_MEMORY_MIB`、`LUMINAL_SEARCH_OP_NAMES` · `luminal › cuBLASLt 进入搜索空间的方式`

要用到的知识

实测驱动的搜索



本题的空间是 `mlp_space()`：matmul（primitive_loop / tiled_kernel / cublaslt）× epilogue（separate / fused）× w_layout × accumulate（fp32 / fp16），共 24 个 genome。真实的空间是每个 e-class 选项数的乘积，大到无法穷举。

概念	本题	要点
预算	`budget`：最多测几个不同的 genome	重复的 genome 不再测（`fingerprint` 去重）
best-so-far 曲线	`curve`	单调不增；它的形状告诉你“再多搜一会儿值不值”
代理成本	`mlp_proxy`	FLOP / 字节模型，分不清手写 tile 和 cuBLASLt，看不见 layout
相关性	`spearman(proxy, measured)`	本例约 0.82：大方向对，细节错
finalists	`select_finalist`	fp16 累加在 K=256 时数值不合格，primitive_loop 的 arena 超预算
break-even	`break_even_runs`	多花的搜索时间要执行多少次才能赚回来

类比 · 选餐厅

代理成本像看点评网站的评分：大致靠谱（米其林不会比路边摊差），但分不清两家都是 4.5 分的店哪家今天更好吃。实测搜索像真的去吃：准，但每家都要花一顿饭的时间（预算）。finalists 像订位：排名第一的那家今天满座（验证失败，内存超预算），那就去第二名。break-even 是在问：为了每顿饭好吃一点点，花一个月去试遍全城的餐厅，你要在这个城市住多久才划算？

题目要求

在 `mine/l3_search.py` 里实现（空间工具 `space_size / all_genomes / fingerprint / random_genome / mutate / crossover`，以及 `mlp_*` 工作负载已给出）：

函数	要点
`rank_insert(ranked, nanos, genome, keep)`	移植 search.rs：满了且不比最差的快就不插；插入位置是第一个比它慢的位置；截断到 keep
`genetic_search(space, measure, budget, population, keep, seed, mutation)`	先测 `population` 个随机 genome，然后循环：从 ranked 里选两个父代 → crossover → mutate → 测（已测过的跳过）；连续 50 次没有新 genome 就停
`proxy_search(space, proxy, measure, k)`	按代理排序全部 genome，只测前 k 个
`spearman(xs, ys)`	秩相关，并列取平均秩
`select_finalist(ranked, validate)`	`(rank, nanos, genome, rejected)`；全部被拒抛 `RuntimeError("no finalist ...")`
`break_even_runs(search_a, lat_a, search_b, lat_b)`	`(search_a - search_b) / (lat_b - lat_a)`；a 不更快时返回 inf

题干

mine/l3_search.py 127 行 复制

```
"""L3 · Search cost vs measured latency: a Luminal-style genome search — your implementati

After saturation the e-graph holds many programs. Luminal's CUDA backend does not trust a
formula to choose between them; it *measures*. The pieces (crates/luminal_cuda_lite/src/se
finalists.rs, src/extraction.rs):

  Genome      one producer choice per e-class (which e-node implements this class)
  GA          a genetic search over genomes; each candidate is compiled + run, metric = me
  rank_insert keep only the best `keep` genomes, sorted by measured ns
  finalists   walk the ranking; a candidate that fails validation (e.g. its arena exceeds
              device memory budget) is rejected and the next rank is tried

Here the "e-graph" is a dict {class: [option, ...]} and `measure(genome)` is injectable.
`mlp_space()` is a small but honest model of y = relu(x @ w + b) on a GPU.
"""
import random
from itertools import product
from math import prod


def space_size(space):
    return prod(len(v) for v in space.values())


def all_genomes(space):
    keys = sorted(space)
    return [dict(zip(keys, combo)) for combo in product(*(space[k] for k in keys))]


def fingerprint(genome):
    return tuple(sorted(genome.items()))


def random_genome(space, rng):
    return {k: rng.choice(v) for k, v in sorted(space.items())}


def mutate(genome, space, rng, p=0.3):
    g = dict(genome)
    for k in sorted(space):
        if rng.random() < p:
            g[k] = rng.choice(space[k])
    return g


def crossover(a, b, rng):
    return {k: (a if rng.random() < 0.5 else b)[k] for k in sorted(a)}


def rank_insert(ranked, nanos, genome, keep):
```

```
    """Port of search.rs rank_insert: ranked is [(nanos, genome)] ascending, at most `keep
    raise NotImplementedError("rank_insert")


def genetic_search(space, measure, budget, population=6, keep=4, seed=0, mutation=0.3):
    """Measure at most `budget` distinct genomes. Returns ranked finalists, the number of
    measurements, and the best-so-far curve (one entry per measurement)."""
    raise NotImplementedError("genetic_search")


def proxy_search(space, proxy, measure, k):
    """The cheap alternative: rank *every* genome by a cost proxy, measure only the top k.
    raise NotImplementedError("proxy_search")


def spearman(xs, ys):
    """Rank correlation (no tie correction needed for the tests; ties get average rank)."""
    raise NotImplementedError("spearman")


def select_finalist(ranked, validate):
    """finalists.rs: walk ranks 1, 2, ...; return (rank, nanos, genome, rejected[]) of the
    candidate that validates. validate(genome) -> None if OK, else a reason string."""
    raise NotImplementedError("select_finalist")


def break_even_runs(search_seconds_a, latency_a, search_seconds_b, latency_b):
    """How many executions until spending more search time (a) pays off over b. inf if nev
    raise NotImplementedError("break_even_runs")


# ---------------------------------------------------------------- the workload
def mlp_space():
    """Decision points for y = relu(x @ w + b) after saturation (simplified)."""
    return {
        "matmul": ["primitive_loop", "tiled_kernel", "cublaslt"],
        "epilogue": ["separate", "fused"],
        "w_layout": ["row_major", "col_major"],
        "accumulate": ["fp32", "fp16"],
    }


MATMUL_NS = {"primitive_loop": 2400, "tiled_kernel": 130, "cublaslt": 95}
FUSE_SAVING = {"primitive_loop": 0, "tiled_kernel": 25, "cublaslt": 22}     # cuBLASLt: EP


def mlp_measure(noise=0.0, seed=0):
    """'Measured' ns: what a GPU would show, including effects a FLOP proxy cannot see."""
    rng = random.Random(seed)

    def m(g):
```

```
        ns = MATMUL_NS[g["matmul"]]
        ns += 30 if g["epilogue"] == "separate" else 30 - FUSE_SAVING[g["matmul"]]
        if g["matmul"] == "tiled_kernel" and g["w_layout"] == "row_major":
            ns += 20                        # B tile loads along K are strided for this kern
        if g["accumulate"] == "fp16":
            ns -= 8
        return ns * (1 + rng.gauss(0, noise)) if noise else ns
    return m


def mlp_proxy(g):
    """A FLOP/byte proxy: knows the primitive loop is bad and fusion saves a pass, but can
    tell a hand-written tile from cuBLASLt, nor see layout effects."""
    base = {"primitive_loop": 2000, "tiled_kernel": 100, "cublaslt": 100}[g["matmul"]]
    return base + (30 if g["epilogue"] == "separate" else 10) + (-5 if g["accumulate"] ==


def mlp_validate(budget_bytes=8 << 20, M=64, N=64, K=256, max_err_ulps=None):
    """Reject plans whose intermediates exceed the arena budget, and fp16 accumulation (pr
    def v(g):
        if g["matmul"] == "primitive_loop" and M * N * K * 4 > budget_bytes:
            return f"arena {M * N * K * 4} B > budget {budget_bytes} B"
        if g["accumulate"] == "fp16":
            return "fp16 accumulation fails the K=%d tolerance" % K
        return None
    return v
```

验收

```
./check.sh l3
```

tests/test_l3_search.py 72 行 复制

```
import pytest

from drill import load

l3 = load("l3_search")
S = l3.mlp_space()


def test_space():
    assert l3.space_size(S) == 3 * 2 * 2 * 2
    gs = l3.all_genomes(S)
    assert len(gs) == 24 and len({l3.fingerprint(g) for g in gs}) == 24


def test_rank_insert_keeps_best_k_sorted():
    r = []
    for ns, name in [(50, "a"), (30, "b"), (70, "c"), (10, "d"), (60, "e")]:
        l3.rank_insert(r, ns, name, keep=3)
    assert r == [(10, "d"), (30, "b"), (50, "a")]
    l3.rank_insert(r, 50, "tie", keep=3)        # cannot displace an equal worst
    assert r[-1] == (50, "a")


def test_true_optimum():
    m = l3.mlp_measure()
    best = min(l3.all_genomes(S), key=m)
    assert m(best) == 95 and best["matmul"] == "cublaslt" and best["epilogue"] == "fused"


def test_genetic_search_budget_and_curve():
    m = l3.mlp_measure()
    r = l3.genetic_search(S, m, budget=10, seed=0)
    assert r["measured"] == 10 and len(r["curve"]) == 10
    assert all(a >= b for a, b in zip(r["curve"], r["curve"][1:]))     # best-so-far never
    assert r["ranked"][0][0] == 95 and len(r["ranked"]) == 4
    small = l3.genetic_search(S, m, budget=3, seed=1)
    assert small["measured"] == 3


def test_proxy_is_correlated_but_not_exact():
    m = l3.mlp_measure()
    gs = l3.all_genomes(S)
    rho = l3.spearman([l3.mlp_proxy(g) for g in gs], [m(g) for g in gs])
    assert 0.7 < rho < 0.95
    assert l3.spearman([1, 2, 3], [10, 20, 30]) == pytest.approx(1.0)
    assert l3.spearman([1, 2, 3], [3, 2, 1]) == pytest.approx(-1.0)


def test_finalists_walk_past_invalid_candidates():
    m = l3.mlp_measure()
```

```
    r = l3.genetic_search(S, m, budget=24, seed=0)
    rank, ns, g, rejected = l3.select_finalist(r["ranked"], l3.mlp_validate())
    assert rank == 3 and ns == 103 and g["accumulate"] == "fp32" and len(rejected) == 2
    assert all("fp16" in why for _, why in rejected)


def test_proxy_top_k_can_all_be_invalid():
    ranked = l3.proxy_search(S, l3.mlp_proxy, l3.mlp_measure(), k=3)
    with pytest.raises(RuntimeError, match="no finalist"):
        l3.select_finalist(ranked, l3.mlp_validate())


def test_primitive_loop_breaks_the_arena_budget():
    v = l3.mlp_validate(budget_bytes=1 << 20)
    g = {"matmul": "primitive_loop", "epilogue": "separate", "w_layout": "row_major", "acc
    assert "arena" in v(g)


def test_break_even():
    # 2 s of search to save 30 ns per call vs 0.1 s of search
    assert l3.break_even_runs(2.0, 87e-9, 0.1, 117e-9) == pytest.approx(1.9 / 30e-9)
    assert l3.break_even_runs(1.0, 120e-9, 0.1, 117e-9) == float("inf")
```

提示

L1 rank_insert 的等号 方向（轻度剧透） 收起

`nanos >= ranked[-1][0]` 时不插（和 Rust 版一致：相等也不能挤掉别人）；插入位置用 `nanos < metric` 找第一个，所以相等的新来者排在已有者之后。

L2 genetic_search 的计数 拆解与常见坑（中度剧透） 收起

用字典 `seen[fingerprint] = ns` 同时做缓存和计数；`evaluate` 在 `len(seen) >= budget` 时直接返回。每测一个新 genome 就 `rank_insert` 并把 `ranked[0][0]` 追加到 `curve`。测试用 `seed=0, budget=10` 检查你恰好测了 10 个、曲线单调、找到了 95 ns 的最优。

参考解

解 solutions/l3_search.py 先自己写完、跑过测试再看 收起

solutions/l3_search.py 186 行 复制

```
"""L3 · Search cost vs measured latency: a Luminal-style genome search — reference sol

After saturation the e-graph holds many programs. Luminal's CUDA backend does not trus
formula to choose between them; it *measures*. The pieces (crates/luminal_cuda_lite/sr
finalists.rs, src/extraction.rs):

  Genome      one producer choice per e-class (which e-node implements this class)
  GA          a genetic search over genomes; each candidate is compiled + run, metric
  rank_insert keep only the best `keep` genomes, sorted by measured ns
  finalists   walk the ranking; a candidate that fails validation (e.g. its arena exce
              device memory budget) is rejected and the next rank is tried

Here the "e-graph" is a dict {class: [option, ...]} and `measure(genome)` is injectabl
`mlp_space()` is a small but honest model of y = relu(x @ w + b) on a GPU.
"""
import random
from itertools import product
from math import prod


def space_size(space):
    return prod(len(v) for v in space.values())


def all_genomes(space):
    keys = sorted(space)
    return [dict(zip(keys, combo)) for combo in product(*(space[k] for k in keys))]


def fingerprint(genome):
    return tuple(sorted(genome.items()))


def random_genome(space, rng):
    return {k: rng.choice(v) for k, v in sorted(space.items())}


def mutate(genome, space, rng, p=0.3):
    g = dict(genome)
    for k in sorted(space):
        if rng.random() < p:
            g[k] = rng.choice(space[k])
    return g


def crossover(a, b, rng):
    return {k: (a if rng.random() < 0.5 else b)[k] for k in sorted(a)}


def rank_insert(ranked, nanos, genome, keep):
```

```
    """Port of search.rs rank_insert: ranked is [(nanos, genome)] ascending, at most `
    keep = max(keep, 1)
    if len(ranked) >= keep and nanos >= ranked[-1][0]:
        return
    pos = next((i for i, (m, _) in enumerate(ranked) if nanos < m), len(ranked))
    ranked.insert(pos, (nanos, genome))
    del ranked[keep:]


def genetic_search(space, measure, budget, population=6, keep=4, seed=0, mutation=0.3)
    """Measure at most `budget` distinct genomes. Returns ranked finalists, the number
    measurements, and the best-so-far curve (one entry per measurement)."""
    rng = random.Random(seed)
    seen, ranked, curve = {}, [], []

    def evaluate(g):
        fp = fingerprint(g)
        if fp in seen or len(seen) >= budget:
            return seen.get(fp)
        ns = measure(g)
        seen[fp] = ns
        rank_insert(ranked, ns, g, keep)
        curve.append(ranked[0][0])
        return ns

    pop = [random_genome(space, rng) for _ in range(population)]
    for g in pop:
        evaluate(g)
    stale = 0
    while len(seen) < min(budget, space_size(space)) and stale < 50:
        parents = [g for _, g in ranked] or pop
        a, b = rng.choice(parents), rng.choice(parents)
        child = mutate(crossover(a, b, rng), space, rng, mutation)
        before = len(seen)
        evaluate(child)
        stale = stale + 1 if len(seen) == before else 0
    return {"ranked": ranked, "measured": len(seen), "curve": curve}


def proxy_search(space, proxy, measure, k):
    """The cheap alternative: rank *every* genome by a cost proxy, measure only the to
    cand = sorted(all_genomes(space), key=proxy)[:k]
    ranked = []
    for g in cand:
        rank_insert(ranked, measure(g), g, k)
    return ranked


def spearman(xs, ys):
    """Rank correlation (no tie correction needed for the tests; ties get average rank
    def ranks(v):
```

```
        order = sorted(range(len(v)), key=lambda i: v[i])
        r = [0.0] * len(v)
        i = 0
        while i < len(order):
            j = i
            while j + 1 < len(order) and v[order[j + 1]] == v[order[i]]:
                j += 1
            for t in range(i, j + 1):
                r[order[t]] = (i + j) / 2
            i = j + 1
        return r
    rx, ry = ranks(xs), ranks(ys)
    n = len(xs)
    mx, my = sum(rx) / n, sum(ry) / n
    cov = sum((a - mx) * (b - my) for a, b in zip(rx, ry))
    vx = sum((a - mx) ** 2 for a in rx) ** 0.5
    vy = sum((b - my) ** 2 for b in ry) ** 0.5
    return cov / (vx * vy)


def select_finalist(ranked, validate):
    """finalists.rs: walk ranks 1, 2, ...; return (rank, nanos, genome, rejected[]) of
    candidate that validates. validate(genome) -> None if OK, else a reason string."""
    rejected = []
    for rank, (ns, g) in enumerate(ranked, 1):
        why = validate(g)
        if why is None:
            return rank, ns, g, rejected
        rejected.append((rank, why))
    raise RuntimeError(f"no finalist validated: {rejected}")


def break_even_runs(search_seconds_a, latency_a, search_seconds_b, latency_b):
    """How many executions until spending more search time (a) pays off over b. inf if
    if latency_a >= latency_b:
        return float("inf")
    return (search_seconds_a - search_seconds_b) / (latency_b - latency_a)


# ---------------------------------------------------------------- the workload
def mlp_space():
    """Decision points for y = relu(x @ w + b) after saturation (simplified)."""
    return {
        "matmul": ["primitive_loop", "tiled_kernel", "cublaslt"],
        "epilogue": ["separate", "fused"],
        "w_layout": ["row_major", "col_major"],
        "accumulate": ["fp32", "fp16"],
    }


MATMUL_NS = {"primitive_loop": 2400, "tiled_kernel": 130, "cublaslt": 95}
```

```
FUSE_SAVING = {"primitive_loop": 0, "tiled_kernel": 25, "cublaslt": 22}     # cuBLASLt


def mlp_measure(noise=0.0, seed=0):
    """'Measured' ns: what a GPU would show, including effects a FLOP proxy cannot see
    rng = random.Random(seed)

    def m(g):
        ns = MATMUL_NS[g["matmul"]]
        ns += 30 if g["epilogue"] == "separate" else 30 - FUSE_SAVING[g["matmul"]]
        if g["matmul"] == "tiled_kernel" and g["w_layout"] == "row_major":
            ns += 20                        # B tile loads along K are strided for this
        if g["accumulate"] == "fp16":
            ns -= 8
        return ns * (1 + rng.gauss(0, noise)) if noise else ns
    return m


def mlp_proxy(g):
    """A FLOP/byte proxy: knows the primitive loop is bad and fusion saves a pass, but
    tell a hand-written tile from cuBLASLt, nor see layout effects."""
    base = {"primitive_loop": 2000, "tiled_kernel": 100, "cublaslt": 100}[g["matmul"]]
    return base + (30 if g["epilogue"] == "separate" else 10) + (-5 if g["accumulate"]


def mlp_validate(budget_bytes=8 << 20, M=64, N=64, K=256, max_err_ulps=None):
    """Reject plans whose intermediates exceed the arena budget, and fp16 accumulation
    def v(g):
        if g["matmul"] == "primitive_loop" and M * N * K * 4 > budget_bytes:
            return f"arena {M * N * K * 4} B > budget {budget_bytes} B"
        if g["accumulate"] == "fp16":
            return "fp16 accumulation fails the K=%d tolerance" % K
        return None
    return v
```

在 Luminal 里：search trade-off 报告

```
# 在 CUDA 机器上，对一个小 graph（例如 crates/luminal_cuda_lite/examples 里最小的那个，或你自己的
SEARCH_LOG=1 LUMINAL_SEARCH_OP_NAMES=1 LUMINAL_SEARCH_SEED=0 cargo run --release -p lumina
SEARCH_LOG=1 LUMINAL_SEARCH_SEED=1 ...     # 换种子
SEARCH_MEMORY_MIB=256 ...                  # 收紧内存预算，看 finalist 拒绝 / fallback 行
```

报告里要有：每次运行的候选数、搜索时间、最终 latency、best-so-far 曲线（从 `SEARCH_LOG` 的 best-so-far 行整理）、被拒的 finalist 及原因，以及一个 break-even 估计。以当前 README 和 `CompileOptions` 实际支持的开关为准——如果某个参数（比如搜索预算）只能在代码里改，就在报告里注明改了哪一行。

▼ 自测：为什么 proxy_search 的前 3 名全军覆没，而 GA 找到了合格的第 3 名?

代理成本认为 fp16 累加更便宜（−5），所以它的前 3 名全是 fp16 累加；但 validation 认为 fp16 在 K=256 时不满足容差，全部拒绝。GA 实测了更多样的 genome，ranked 里保留了 fp32 的候选，finalist 遍历到第 3 名时找到了合格的 103 ns 方案。教训：**排名和验证要分开，且排名之后要留足够多的候选**——这正是 finalists.rs 存在的原因。

双主线 · Track L · Luminal（graph compiler） · 第 13 题

L4 Backend trace：选中计划 → launch 列表 → arena 分配（plan_slab）

搜索选中了一个计划之后，backend 还要做两件事：决定每个 buffer 放在一块设备内存的哪个偏移上，以及按依赖顺序发出 launch。这道题移植 Luminal 的 plan_slab（DAG 可达性 + first fit + 256 B 对齐），把 relu(x @ w + b) 的融合 / 不融合两个计划都排一遍，再把一次执行拆成 graph compile / search / prepare / execute 四个阶段。

⏱ 45 min | 难度 中 | 你写 `mine/l4_plan_trace.py` | 验收 `./check.sh l4`

参考解 `solutions/l4_plan_trace.py` | 📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 题目要求
5. 题干
6. 验收
7. 提示
8. 参考解
9. 在 Luminal 里：一条端到端 trace

背景

roadmap 的 L4：“打开 Luminal logging / LLIR dump / CUDA profiling 相关环境变量，跟踪一个 matmul graph；区分 graph compile、candidate evaluation、prepare、execute；识别实际调用的 kernel / library；比较理论 FLOPs 与真实 GPU 时间。交付：一条端到端 trace 注释。”

trace 里最难读懂的一段是内存：一个 finalist 的 `arena.slab_bytes` 就是它的设备内存高水位，finalists.rs 拿它和设备预算比较。这个数来自 `luminal › src/buffer/arena.rs 的 plan_slab`，文档注释写得很清楚：

> Two buffers may alias only if every use of one strictly precedes every use of the other. Unordered branches and operands of the same node cannot alias.

源码与资料

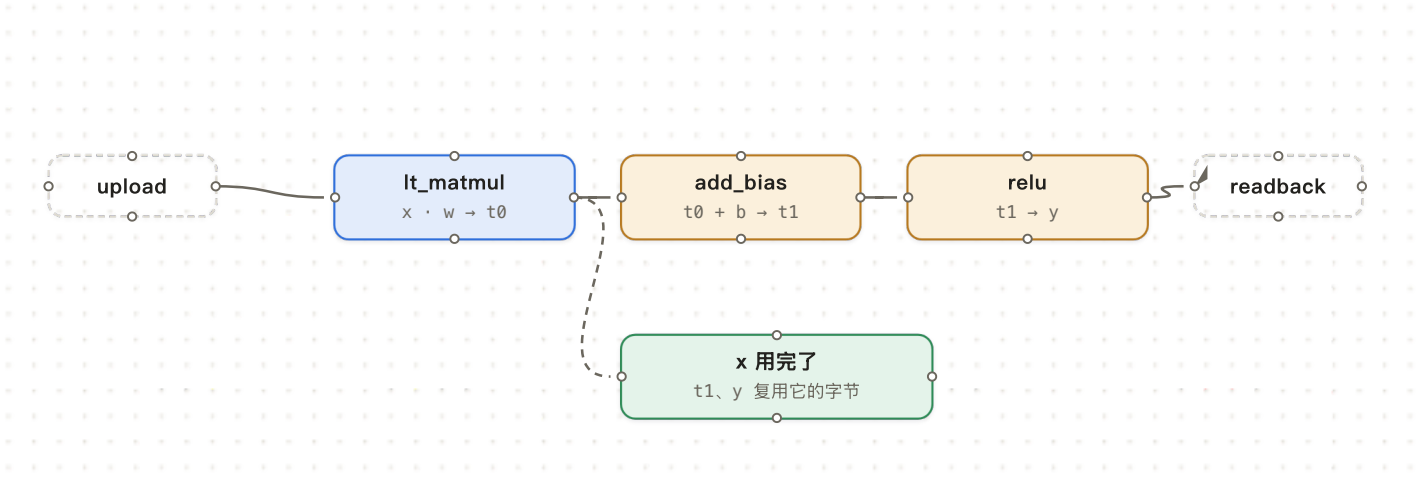
`luminal › SlabBuffer` · `luminal › aligned_size` · `luminal › cuda_lite/arena.rs：buffer_bytes` · `luminal › finalists.rs：arena 与 slab_bytes` · `luminal › cuda_graph.rs` · README：`LUMINAL_LOG_LLIR`、`LLIR_DUMP_DIR`、`LUMINAL_CUDA_PROFILE_RECAPTURE`、`LUMINAL_CUDA_MEMORY_DEBUG`

要用到的知识

plan_slab 的三步

1. 对 DAG 做拓扑排序，倒着求传递闭包：`after[u]` = 从 u 出发能到达的所有节点（真实代码用 64 位 bitset）。
2. `precedes(a, b)` ⇔ a 的**每一次**使用都能到达 b 的**每一次**使用。
3. 按请求顺序逐个放：只有和当前 buffer “互不先后”的已放 buffer 才算占位；把占位区间排序，从 0 开始找第一个放得下的空隙（first fit）。大小先向上对齐到 256 B；空 buffer 也占一个对齐单位。

不融合的计划：谁能和谁共用内存



x 的使用是 {upload, lt_matmul}，t1 的使用是 {add_bias, relu}：每一个 x 的使用都在每一个 t1 的使用之前，所以 t1 可以放在 offset 0。t0 的使用 {lt_matmul, add_bias} 和 x 在 lt_matmul 处重叠，不能共用。

类比 · 会议室排期

slab 是一层楼的会议室走廊，每个 buffer 是一场会，它的 uses 是开会的时间段。两场会能用同一间会议室，当且仅当一场**完全结束**之后另一场才开始（precedes）。两场会只要可能同时进行（DAG 上没有先后：并行的分支），就必须用不同的房间。first fit 就是“从走廊最左边开始，找第一间空着的”。它简单、确定，但不保证最省——文档原话：“A capacity error means this algorithm did not find a placement, not that no possible placement exists.”

题目要求

在 `mine/l4_plan_trace.py` 里实现（`SlabBuffer`、`aligned`、`mlp_plan`、`PHASES` 已给出）：

函数	要点
`toposort(dag)`	Kahn 算法，ready 集合按名字排序（结果确定）；有环抛 `ValueError`
`reachability(dag)`	`{node: 从它可达的节点集合}`（不含自身）
`plan_slab(dag, buffers, capacity, alignment=256)`	`{id: offset}`；放不下抛 `MemoryError("slab capacity ...")`；alignment ⩽ 0 抛 `ValueError`
`slab_bytes(buffers, offsets)`	高水位 = max(offset + 对齐后大小)
`naive_bytes(buffers)`	不复用时的总量
`check_no_overlap(dag, buffers, offsets)`	互不先后的 buffer 字节区间不相交
`phase_breakdown(events)`	四个阶段的总时间和占比；未知阶段抛 `ValueError`
`achieved(flops, seconds, peak)`	`(FLOP/s, 占峰值比例)`

题干

mine/l4_plan_trace.py 97 行 复制

```
"""L4 · Backend trace: from the selected plan to GPU launches and one memory slab — your i

Once search picked a genome, the backend turns the plan into:

  1. a DAG of kernels / library calls (the LLIR the README's LUMINAL_LOG_LLIR prints),
  2. buffer lifetimes: the DAG nodes at which each buffer is read or written,
  3. one device slab with an offset per buffer — src/buffer/arena.rs `plan_slab`:
       * transitive closure of the DAG ("u happens-before v"),
       * two buffers may share bytes only if *every* use of one precedes *every* use of th
       * first fit in request order, sizes rounded up to the alignment (256 B on CUDA),
  4. an execution timeline, split into phases (graph compile / search / prepare / execute
     what LUMINAL_CUDA_PROFILE_RECAPTURE breaks TTFT/TPOT into).
"""
from dataclasses import dataclass


@dataclass
class SlabBuffer:
    id: str
    bytes: int
    uses: list          # DAG node names


def toposort(dag):
    """dag: {node: [successors]}. Kahn's algorithm with sorted tie-break (deterministic)."
    raise NotImplementedError("toposort")


def reachability(dag):
    """after[u] = set of nodes strictly reachable from u."""
    raise NotImplementedError("reachability")


def aligned(n, a):
    return -(-max(n, 1) // a) * a          # empty buffers still reserve one aligned unit


def plan_slab(dag, buffers, capacity, alignment=256):
    """Port of arena.rs plan_slab (FirstFit). Returns {buffer id: offset}; raises on overf
    raise NotImplementedError("plan_slab")


def slab_bytes(buffers, offsets, alignment=256):
    """High-water mark: what finalists.rs compares to the device budget."""
    raise NotImplementedError("slab_bytes")


def naive_bytes(buffers, alignment=256):
    """One allocation per buffer, no reuse."""
    raise NotImplementedError("naive_bytes")
```

```
def check_no_overlap(dag, buffers, offsets, alignment=256):
    """Invariant: buffers whose lifetimes are not ordered never share a byte."""
    raise NotImplementedError("check_no_overlap")


# ------------------------------------------------------------------ the MLP plan
def mlp_plan(M=64, K=256, N=64, fused=True, elem=4):
    """The selected plan for relu(x @ w + b) as (dag, buffers, launches).
    fused=True : one cuBLASLt call with the RELU_BIAS epilogue.
    fused=False: cuBLASLt matmul, then an add-bias kernel, then a relu kernel."""
    if fused:
        dag = {"upload": ["lt_matmul_relu_bias"], "lt_matmul_relu_bias": ["readback"], "re
        bufs = [SlabBuffer("x", M * K * elem, ["upload", "lt_matmul_relu_bias"]),
                SlabBuffer("w", K * N * elem, ["upload", "lt_matmul_relu_bias"]),
                SlabBuffer("b", N * elem, ["upload", "lt_matmul_relu_bias"]),
                SlabBuffer("workspace", 1 << 20, ["lt_matmul_relu_bias"]),
                SlabBuffer("y", M * N * elem, ["lt_matmul_relu_bias", "readback"])]
        launches = [("lt_matmul_relu_bias", "cublasLtMatmul", 2 * M * N * K)]
    else:
        dag = {"upload": ["lt_matmul"], "lt_matmul": ["add_bias"], "add_bias": ["relu"],
               "relu": ["readback"], "readback": []}
        bufs = [SlabBuffer("x", M * K * elem, ["upload", "lt_matmul"]),
                SlabBuffer("w", K * N * elem, ["upload", "lt_matmul"]),
                SlabBuffer("b", N * elem, ["upload", "add_bias"]),
                SlabBuffer("workspace", 1 << 20, ["lt_matmul"]),
                SlabBuffer("t0", M * N * elem, ["lt_matmul", "add_bias"]),
                SlabBuffer("t1", M * N * elem, ["add_bias", "relu"]),
                SlabBuffer("y", M * N * elem, ["relu", "readback"])]
        launches = [("lt_matmul", "cublasLtMatmul", 2 * M * N * K),
                    ("add_bias", "kernel", M * N), ("relu", "kernel", M * N)]
    return dag, bufs, launches


# ------------------------------------------------------------------ phases
PHASES = ("graph_compile", "search", "prepare", "execute")


def phase_breakdown(events):
    """events: [(phase, seconds)] possibly repeated. Returns {phase: total, 'share': {phas
    raise NotImplementedError("phase_breakdown")


def achieved(flops, seconds, peak_flops):
    """Achieved FLOP/s and fraction of peak — compare theory with the profiled kernel time
    raise NotImplementedError("achieved")
```

验收

```
./check.sh l4
```

tests/test_l4_plan_trace.py 66 行 复制

```
import pytest

from drill import load

l4 = load("l4_plan_trace")
B = l4.SlabBuffer


def test_toposort_and_reachability():
    dag = {"a": ["b", "c"], "b": ["d"], "c": ["d"], "d": []}
    assert l4.toposort(dag) == ["a", "b", "c", "d"]
    after = l4.reachability(dag)
    assert after["a"] == {"b", "c", "d"} and after["b"] == {"d"} and after["d"] == set()
    with pytest.raises(ValueError):
        l4.toposort({"a": ["b"], "b": ["a"]})


def test_chain_reuses_memory():
    dag = {"a": ["b"], "b": ["c"], "c": []}
    bufs = [B("p", 1000, ["a"]), B("q", 100, ["b"]), B("r", 300, ["c"]), B("s", 10, ["a",
    off = l4.plan_slab(dag, bufs, 10_000)
    assert off == {"p": 0, "q": 0, "r": 0, "s": 1024}       # p, q, r never live together
    assert l4.slab_bytes(bufs, off) == 1280
    assert l4.naive_bytes(bufs) == 1024 + 256 + 512 + 256


def test_unordered_branches_cannot_alias():
    dag = {"a": ["b", "c"], "b": ["d"], "c": ["d"], "d": []}
    bufs = [B("p", 256, ["b"]), B("q", 256, ["c"]), B("r", 256, ["d"])]
    off = l4.plan_slab(dag, bufs, 10_000)
    assert off["p"] != off["q"]                                  # b and c may run concurrentl
    assert off["r"] == 0                                         # d is after both
    assert l4.check_no_overlap(dag, bufs, off)


def test_alignment_and_capacity():
    dag = {"a": []}
    assert l4.aligned(1, 256) == 256 and l4.aligned(0, 256) == 256 and l4.aligned(257, 256
    with pytest.raises(MemoryError, match="capacity"):
        l4.plan_slab(dag, [B("x", 300, ["a"]), B("y", 300, ["a"])], capacity=768)
    with pytest.raises(ValueError):
        l4.plan_slab(dag, [B("x", 1, ["a"])], 100, alignment=0)


def test_mlp_plans():
    dag, bufs, launches = l4.mlp_plan(fused=True)
    off = l4.plan_slab(dag, bufs, 64 << 20)
    assert len(launches) == 1 and launches[0][1] == "cublasLtMatmul"
    assert l4.check_no_overlap(dag, bufs, off)
    dag2, bufs2, launches2 = l4.mlp_plan(fused=False)
```

```
    off2 = l4.plan_slab(dag2, bufs2, 64 << 20)
    assert [l[0] for l in launches2] == ["lt_matmul", "add_bias", "relu"]
    assert off2["t1"] == 0 and off2["y"] == 16384           # x is dead by then: its byte
    assert l4.slab_bytes(bufs2, off2) < l4.naive_bytes(bufs2)
    assert l4.check_no_overlap(dag2, bufs2, off2)


def test_phases_and_achieved():
    ph = l4.phase_breakdown([("graph_compile", 0.4), ("search", 3.0), ("prepare", 0.1),
                             ("execute", 0.0005), ("execute", 0.0005)])
    assert ph["execute"] == pytest.approx(0.001)
    assert ph["share"]["search"] == pytest.approx(3.0 / 3.501)
    with pytest.raises(ValueError):
        l4.phase_breakdown([("lunch", 1.0)])
    f, frac = l4.achieved(2 * 64 * 64 * 256, 10e-6, 40e12)
    assert f == pytest.approx(2 * 64 * 64 * 256 / 10e-6) and frac < 0.01   # tiny GEMM: la
```

提示

L1 reachability 要倒着算 方向（轻度剧透） 收起

按拓扑序的**逆序**处理节点：`after[n] = ∪ ({s} ∪ after[s]) for s in dag[n]`。这样处理 n 时，它的所有后继都已经算好了。

L2 first fit 的循环 拆解与常见坑（中度剧透） 收起

```
off = 0
for start, n in sorted(occupied):
    if off + size <= start: break      # 放进这个空隙
    off = max(off, start + n)          # 跳过这个占位
```

L3 测试里的两个关键数 几乎是答案（重度剧透） 收起

链 `a → b → c` 上，p / q / r 各只在一个节点使用，彼此严格先后，所以都在 offset 0；s 跨越 a 和 c，和三者都重叠，放到 `aligned(1000) = 1024`。菱形 DAG 里 b、c 无先后，p、q 必须错开。

参考解

解 **solutions/l4_plan_trace.py** 先自己写完、跑过测试再看 收起 ▾

`solutions/l4_plan_trace.py` 156 行 复制

```
"""L4 · Backend trace: from the selected plan to GPU launches and one memory slab — re

Once search picked a genome, the backend turns the plan into:

  1. a DAG of kernels / library calls (the LLIR the README's LUMINAL_LOG_LLIR prints),
  2. buffer lifetimes: the DAG nodes at which each buffer is read or written,
  3. one device slab with an offset per buffer — src/buffer/arena.rs `plan_slab`:
       * transitive closure of the DAG ("u happens-before v"),
       * two buffers may share bytes only if *every* use of one precedes *every* use o
       * first fit in request order, sizes rounded up to the alignment (256 B on CUDA)
  4. an execution timeline, split into phases (graph compile / search / prepare / exec
     what LUMINAL_CUDA_PROFILE_RECAPTURE breaks TTFT/TPOT into).
"""
from dataclasses import dataclass


@dataclass
class SlabBuffer:
    id: str
    bytes: int
    uses: list          # DAG node names


def toposort(dag):
    """dag: {node: [successors]}. Kahn's algorithm with sorted tie-break (deterministi
    nodes = set(dag) | {s for v in dag.values() for s in v}
    indeg = {n: 0 for n in nodes}
    for v in dag.values():
        for s in v:
            indeg[s] += 1
    ready = sorted(n for n in nodes if indeg[n] == 0)
    out = []
    while ready:
        n = ready.pop(0)
        out.append(n)
        for s in sorted(dag.get(n, [])):
            indeg[s] -= 1
            if indeg[s] == 0:
                ready.append(s)
                ready.sort()
    if len(out) != len(nodes):
        raise ValueError("slab lifetime graph must be acyclic")
    return out


def reachability(dag):
    """after[u] = set of nodes strictly reachable from u."""
    after = {}
    for n in reversed(toposort(dag)):
        acc = set()
```

```
        for s in dag.get(n, []):
            acc.add(s)
            acc |= after[s]
        after[n] = acc
    return after


def aligned(n, a):
    return -(-max(n, 1) // a) * a          # empty buffers still reserve one aligned u


def plan_slab(dag, buffers, capacity, alignment=256):
    """Port of arena.rs plan_slab (FirstFit). Returns {buffer id: offset}; raises on c
    if alignment <= 0:
        raise ValueError("slab alignment must be positive")
    after = reachability(dag)
    precedes = lambda a, b: all(v in after[u] for u in a.uses for v in b.uses)
    sizes = [aligned(b.bytes, alignment) for b in buffers]
    offsets = []
    for i, b in enumerate(buffers):
        occupied = sorted((offsets[j], sizes[j]) for j in range(i)
                          if not precedes(b, buffers[j]) and not precedes(buffers[j],
        off = 0
        for start, n in occupied:
            if off + sizes[i] <= start:
                break
            off = max(off, start + n)
        if off + sizes[i] > capacity:
            raise MemoryError(f"slab capacity {capacity} exceeded while placing reques
        offsets.append(off)
    return {b.id: o for b, o in zip(buffers, offsets)}


def slab_bytes(buffers, offsets, alignment=256):
    """High-water mark: what finalists.rs compares to the device budget."""
    return max((offsets[b.id] + aligned(b.bytes, alignment) for b in buffers), default


def naive_bytes(buffers, alignment=256):
    """One allocation per buffer, no reuse."""
    return sum(aligned(b.bytes, alignment) for b in buffers)


def check_no_overlap(dag, buffers, offsets, alignment=256):
    """Invariant: buffers whose lifetimes are not ordered never share a byte."""
    after = reachability(dag)
    precedes = lambda a, b: all(v in after[u] for u in a.uses for v in b.uses)
    for i, a in enumerate(buffers):
        for b in buffers[i + 1:]:
            if precedes(a, b) or precedes(b, a):
                continue
```

```
            a0, a1 = offsets[a.id], offsets[a.id] + aligned(a.bytes, alignment)
            b0, b1 = offsets[b.id], offsets[b.id] + aligned(b.bytes, alignment)
            if a0 < b1 and b0 < a1:
                return False
    return True


# ---------------------------------------------------------------- the MLP plan
def mlp_plan(M=64, K=256, N=64, fused=True, elem=4):
    """The selected plan for relu(x @ w + b) as (dag, buffers, launches).
    fused=True : one cuBLASLt call with the RELU_BIAS epilogue.
    fused=False: cuBLASLt matmul, then an add-bias kernel, then a relu kernel."""
    if fused:
        dag = {"upload": ["lt_matmul_relu_bias"], "lt_matmul_relu_bias": ["readback"],
        bufs = [SlabBuffer("x", M * K * elem, ["upload", "lt_matmul_relu_bias"]),
                SlabBuffer("w", K * N * elem, ["upload", "lt_matmul_relu_bias"]),
                SlabBuffer("b", N * elem, ["upload", "lt_matmul_relu_bias"]),
                SlabBuffer("workspace", 1 << 20, ["lt_matmul_relu_bias"]),
                SlabBuffer("y", M * N * elem, ["lt_matmul_relu_bias", "readback"])]
        launches = [("lt_matmul_relu_bias", "cublasLtMatmul", 2 * M * N * K)]
    else:
        dag = {"upload": ["lt_matmul"], "lt_matmul": ["add_bias"], "add_bias": ["relu"
               "relu": ["readback"], "readback": []}
        bufs = [SlabBuffer("x", M * K * elem, ["upload", "lt_matmul"]),
                SlabBuffer("w", K * N * elem, ["upload", "lt_matmul"]),
                SlabBuffer("b", N * elem, ["upload", "add_bias"]),
                SlabBuffer("workspace", 1 << 20, ["lt_matmul"]),
                SlabBuffer("t0", M * N * elem, ["lt_matmul", "add_bias"]),
                SlabBuffer("t1", M * N * elem, ["add_bias", "relu"]),
                SlabBuffer("y", M * N * elem, ["relu", "readback"])]
        launches = [("lt_matmul", "cublasLtMatmul", 2 * M * N * K),
                    ("add_bias", "kernel", M * N), ("relu", "kernel", M * N)]
    return dag, bufs, launches


# ---------------------------------------------------------------- phases
PHASES = ("graph_compile", "search", "prepare", "execute")


def phase_breakdown(events):
    """events: [(phase, seconds)] possibly repeated. Returns {phase: total, 'share': {
    tot = {p: 0.0 for p in PHASES}
    for p, s in events:
        if p not in tot:
            raise ValueError(f"unknown phase {p}")
        tot[p] += s
    all_s = sum(tot.values())
    tot["share"] = {p: (tot[p] / all_s if all_s else 0.0) for p in PHASES}
    return tot
```

```
def achieved(flops, seconds, peak_flops):
    """Achieved FLOP/s and fraction of peak — compare theory with the profiled kernel
    f = flops / seconds
    return f, f / peak_flops
```

在 Luminal 里：一条端到端 trace

```
LUMINAL_LOG=1 LUMINAL_LOG_LLIR=1 LUMINAL_SEARCH_OP_NAMES=1 \
LLIR_DUMP_DIR=./llir LUMINAL_CUDA_PROFILE_RECAPTURE=1 LUMINAL_CUDA_MEMORY_DEBUG=1 \
cargo run --release -p luminal_cuda_lite --example <your_matmul_example> --features device
```

在 trace.log 上做注释：

1. **graph compile**：`EGGLOG_LOG` 的 e-graph 构建 / schedule 行。
2. **candidate evaluation**：`SEARCH_LOG` 的进度和 best-so-far 行；`[Kernels: …] [Hosts: …]` 告诉你候选由哪些 kernel / 库组成（是否出现 cuBLASLt）。
3. **prepare / execute**：`CUDA_PREPARE_PROFILE`、`CUDA_EXEC_PROFILE`、`CUDA_ALLOC_PROFILE` 行，填进 `phase_breakdown`。
4. **内存**：`llir/` 里选中 finalist 的 `.txt` / `.dot`，和 `LUMINAL_CUDA_MEMORY_DEBUG` 的 arena 统计；用 `plan_slab` 复算一个小图的 slab，对上 `slab_bytes`。
5. **理论 vs 实测**：`achieved(2·M·N·K, kernel_seconds, peak)`。64×256×64 这种小 GEMM 只能达到峰值的零点几个百分点——launch-bound；换成 4096³ 再看。

▼ 自测：为什么“同一个节点的两个操作数”不能共用内存，即使它们都只在这个节点被用到?

对同一个节点 v，`precedes(a, b)` 要求 a 的使用 v 严格先于 b 的使用 v，也就是 v 能到达 v——在 DAG 里不可能。所以两者互不先后，必须放在不同的地址。这符合直觉：kernel 运行时要同时读两个操作数，它们当然不能占同一块内存。

双主线 · Track L · Luminal（graph compiler） · 第 14 题

L5 Differential test：Reference vs CUDA，带理由的容差

Luminal 同一张图可以在 ReferenceRuntime（CPU，定义语义）和 CUDA 上跑。backend 贡献最有说服力的形式，就是一个“CUDA 和 Reference 在随机形状、dtype、边界尺寸下一致”的回归测试，外加一句说得出理由的容差。这道题围绕 softmax 写出整套差分测试，并用三个真实风格的 bug 检验它能不能抓到。

45 min | 难度 中 | 你写 `mine/l5_differential.py` | 验收 `./check.sh l5`

参考解 `solutions/l5_differential.py` | 📖 单独讲解页

1. 背景
2. 源码与资料
3. 要用到的知识
4. 题目要求
5. 题干
6. 验收
7. 提示
8. 参考解
9. 在 Luminal 里：写一个真的

背景

roadmap 的 L5：“选一个 primitive / high-level op，在随机 shape、dtype、边界维度下比较 ReferenceRuntime 与 CUDA 输出。交付：一个 regression test，附 tolerance rationale。”Week 11 的建议也是“补 reference-vs-CUDA test 或最小 build regression；避免没有数据支持的性能优化”。

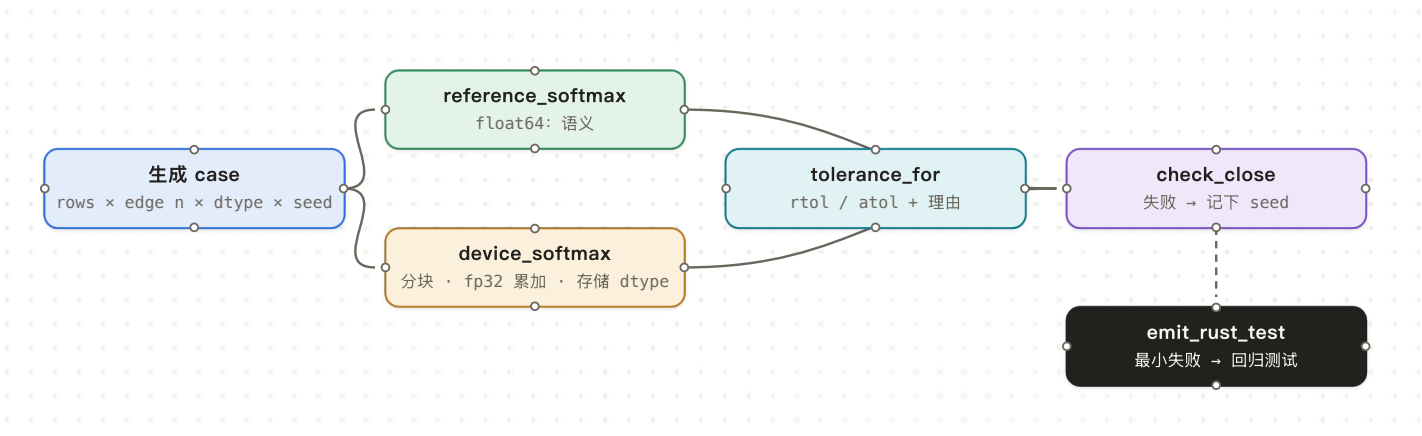
Luminal 仓库里已经有大量这类测试，可以学它们的写法：`luminal › cublaslt_precision_matches.rs`、`luminal › cublaslt_tf32.rs`、`crates/luminal_cuda_lite/tests/` 下的其他文件，以及 reference 一侧的 `luminal › harness.rs`、`luminal › narrow_float.rs`。

源码与资料

`luminal › Luminal softmax` · `luminal › ReferenceRuntime` · README：`LUMINAL_CUDA_CHECK_NONFINITE_INTERNAL`（检查中间 buffer 的非有限值）· K3 的 `check_close` / `tolerance` / `edge_sizes`

要用到的知识

差分测试的结构



输入先舍入到存储 dtype，再分别交给两边，这样两边看到的是同一组数；差异只来自计算本身。

容差怎么定（要写进测试的那句理由）

- 输出要舍入到存储 dtype：相对误差最多半个 ulp，取 `2·eps(dtype)` 留余量。
- 长度 n 的 fp32 求和：约 `√n · eps32`，取 `4·eps32·√n`。
- `rtol = 2·eps(dtype) + 4·eps32·√n`；softmax 的输出在 [0, 1]，很多概率接近 0，相对误差没有意义，所以 `atol = max(eps(dtype) / n, 1e-7)`。

三个 bug，三种教训

bug	现象	什么输入能抓到
`tail`：最后一个不满的块没进求和	结果整体偏大	n 不是 64 的倍数且 > 64（`edge_sizes` 里的 65、97、129…）
`fp16_acc`：在存储 dtype 里累加	误差 1e−5 量级，fp32 容差能抓到	任何长度；fp32 输入最明显
`no_max_sub`：没有先减最大值	exp 溢出成 inf，输出 NaN	**只有大输入**：N(0, 3) 永远抓不到，scale = 40 才抓到

> **类比 · 质检抽样**
>
> 差分测试是工厂质检：同一批原料（同一组舍入后的输入）同时送进“标准样机”（Reference）和“新生产线”（CUDA），比较产品。只抽“正常尺寸”的零件（随机正态输入），可能永远发现不了“超大零件会卡住机器”（溢出）的问题；质检方案必须专门放进边界件：刚好多出一点的尺寸（n = 65）、极端的数值（大 logits）。容差理由就是质检标准上那句“允许误差 ±x，因为……”。

题目要求

在 `mine/l5_differential.py` 里实现（`store`、`reference_softmax`、`device_softmax`（含三个 bug 开关）、`make_input` 已给出）：

函数	要点
`tolerance_for(dtype, n)`	`(rtol, atol, why)`；`why` 里写明 dtype 和 n
`run_differential(impl, cases, seed, block, dtypes, scale)`	用 `random.Random(seed)` 依次抽 rows ∈ {1, 2, 7}、n ∈ `edge_sizes(block, 4·block)`、dtype、输入 seed；返回 `(failures, cases)`
`smallest_failure(fails)`	元素数最少、dtype 最简单的那个
`emit_rust_test(f, name)`	以 `#[test]` 开头，含 `softmax(1)`、`DType::F32` 等、容差注释；花括号配对

题干

mine/l5_differential.py 95 行 复制

```
"""L5 · Differential test: ReferenceRuntime vs CUDA, written as a regression test — your i

Luminal ships two runtimes for the same compiled graph: `luminal_reference` (a CPU interpr
that defines the semantics, with explicit narrow-float and tf32 emulation: narrow_float.rs
tf32.rs) and `luminal_cuda_lite` (the real device). A backend contribution is judged by
"does CUDA agree with Reference within a tolerance I can justify?".

This drill builds that harness in Python around one op (softmax over the last axis, a clas
place for backend bugs: max-subtraction, masked tails, fp16 overflow):

  reference_softmax   float64, the semantics
  device_softmax      a 'device' implementation: tiled in blocks, fp32 accumulation, outpu
                      the storage dtype — with switchable, realistic bugs
  tolerance_for       rtol/atol from the dtype's eps and the reduction length (K3), plus a
                      documented rationale string that goes into the test
  run_differential    random shapes (edge-heavy), dtypes, seeds -> list of failures
  emit_rust_test      the regression test in the style of crates/luminal_cuda_lite/tests
"""
import math
import random

import numpy as np

from solutions.k3_numerics import EPS, check_close, edge_sizes, to_bf16

STORAGE = {"float32": np.float32, "float16": np.float16, "bfloat16": np.float32}


def store(x, dtype):
    x = np.asarray(x, np.float32).astype(STORAGE[dtype])
    return to_bf16(x) if dtype == "bfloat16" else x


def reference_softmax(x):
    x = np.asarray(x, np.float64)
    m = x.max(axis=-1, keepdims=True)
    e = np.exp(x - m)
    return e / e.sum(axis=-1, keepdims=True)


def device_softmax(x, dtype, block=64, bug=None):
    """Two passes over the row in blocks of `block`: online max+sum, then normalize.
    bug: None | 'no_max_sub' (overflows for large inputs) | 'tail' (last partial block ign
    the sum) | 'fp16_acc' (accumulate in the storage dtype)."""
    x = store(x, dtype).astype(np.float32)
    rows, n = x.shape
    acc_t = np.float16 if bug == "fp16_acc" else np.float32
    out = np.empty_like(x)
    for r in range(rows):
        m = np.float32(-np.inf) if bug != "no_max_sub" else np.float32(0)
```

```
        s = acc_t(0)
        stop = n if bug != "tail" else (n // block) * block or n
        for k0 in range(0, stop, block):
            blk = x[r, k0:min(k0 + block, stop)]
            if bug != "no_max_sub":
                m_new = max(m, blk.max())
                s = acc_t(s * acc_t(np.exp(m - m_new)) + acc_t(np.exp(blk - m_new).sum()))
                m = m_new
            else:
                with np.errstate(over="ignore"):
                    s = acc_t(s + acc_t(np.exp(blk).sum()))
        with np.errstate(over="ignore", invalid="ignore"):
            out[r] = np.exp(x[r] - m) / np.float32(s)
    return store(out, dtype)


def tolerance_for(dtype, n):
    """(rtol, atol, rationale). Softmax outputs are in [0, 1]; the error comes from (a) ro
    the *output* to the storage dtype (half an ulp: eps/2 relative) and (b) the length-n s
    fp32 (~sqrt(n) * eps32). atol guards the tiny probabilities where relative error is no
    raise NotImplementedError("tolerance_for")


def make_input(rows, n, dtype, seed, scale=3.0):
    rng = np.random.default_rng(seed)
    x = rng.standard_normal((rows, n)) * scale
    return store(x, dtype)


def run_differential(impl, cases=60, seed=0, block=64, dtypes=("float32", "float16", "bflo
    """Returns (failures, tried). failures: [{'rows','n','dtype','seed','report'}]."""
    raise NotImplementedError("run_differential")


def smallest_failure(fails):
    raise NotImplementedError("smallest_failure")


RUST_DTYPE = {"float32": "F32", "float16": "F16", "bfloat16": "Bf16"}


def emit_rust_test(f, name="softmax_matches_reference"):
    """A Rust regression test sketch: same graph compiled for both runtimes, compared with
    justified tolerance. (Adapt the API calls to the current crate before submitting.)"""
    raise NotImplementedError("emit_rust_test")
```

验收

```
./check.sh l5
```

tests/test_l5_differential.py 57 行 复制

```
import numpy as np

from drill import load

l5 = load("l5_differential")


def impl(bug=None):
    return lambda x, d: l5.device_softmax(x, d, bug=bug)


def test_reference_is_stable():
    y = l5.reference_softmax(np.array([[1000.0, 1000.0], [0.0, -np.inf]]))
    np.testing.assert_allclose(y, [[0.5, 0.5], [1.0, 0.0]])


def test_correct_device_passes_everywhere():
    fails, tried = l5.run_differential(impl(None), cases=60, seed=0)
    assert tried == 60 and fails == []
    fails, _ = l5.run_differential(impl(None), cases=30, seed=1, scale=40.0)
    assert fails == []


def test_tail_bug_found_and_minimized():
    fails, _ = l5.run_differential(impl("tail"), cases=60, seed=0)
    assert len(fails) == 26
    f = l5.smallest_failure(fails)
    assert f["rows"] == 1 and f["n"] % 64 != 0 and f["n"] > 64


def test_fp16_accumulation_bug_found():
    fails, _ = l5.run_differential(impl("fp16_acc"), cases=60, seed=0)
    assert len(fails) == 11 and all(f["report"]["max_abs"] < 1e-2 for f in fails)   # smal


def test_overflow_bug_needs_large_inputs():
    # N(0, 3) inputs never overflow exp(): the missing max-subtraction is invisible ...
    fails, _ = l5.run_differential(impl("no_max_sub"), cases=60, seed=0)
    assert fails == []
    # ... until the test includes large logits
    fails, _ = l5.run_differential(impl("no_max_sub"), cases=60, seed=0, scale=40.0)
    assert len(fails) == 44


def test_tolerance_rationale():
    rtol, atol, why = l5.tolerance_for("float16", 65)
    assert rtol > 2 * 2 ** -10 and atol < rtol
    assert "float16" in why and "n=65" in why
    r32, _, _ = l5.tolerance_for("float32", 65)
    assert r32 < rtol < l5.tolerance_for("bfloat16", 65)[0]
```

```
def test_emit_rust_test():
    fails, _ = l5.run_differential(impl("tail"), cases=60, seed=0)
    src = l5.emit_rust_test(l5.smallest_failure(fails))
    assert src.startswith("#[test]") and "softmax(1)" in src and "DType::F32" in src
    assert "tolerance:" in src and src.count("{") == src.count("}")
```

提示

L1 抽样顺序决定测试里的数字 方向（轻度剧透） 收起

测试断言 `tail` 恰好有 26 个失败、`no_max_sub` 在 scale=40 时 44 个——这些数依赖于你抽样的**顺序**：每个 case 依次 `rng.choice([1, 2, 7])`、`rng.choice(sizes)`、`rng.choice(dtypes)`、`rng.randrange(1 << 30)`。

L2 emit_rust_test 用 f-string 拆解与常见坑（中度剧透） 收起

Rust 的花括号在 Python f-string 里要写成 `{{` / `}}`。容差用 `:.3e` 格式化。`RUST_DTYPE` 已给出 dtype 名的映射。

参考解

解 solutions/l5_differential.py 先自己写完、跑过测试再看 收起

solutions/l5_differential.py 124 行 复制

```
"""L5 · Differential test: ReferenceRuntime vs CUDA, written as a regression test — re

Luminal ships two runtimes for the same compiled graph: `luminal_reference` (a CPU int
that defines the semantics, with explicit narrow-float and tf32 emulation: narrow_floa
tf32.rs) and `luminal_cuda_lite` (the real device). A backend contribution is judged b
"does CUDA agree with Reference within a tolerance I can justify?".

This drill builds that harness in Python around one op (softmax over the last axis, a
place for backend bugs: max-subtraction, masked tails, fp16 overflow):

  reference_softmax   float64, the semantics
  device_softmax      a 'device' implementation: tiled in blocks, fp32 accumulation, c
                      the storage dtype — with switchable, realistic bugs
  tolerance_for       rtol/atol from the dtype's eps and the reduction length (K3), pl
                      documented rationale string that goes into the test
  run_differential    random shapes (edge-heavy), dtypes, seeds -> list of failures
  emit_rust_test      the regression test in the style of crates/luminal_cuda_lite/tes
"""
import math
import random

import numpy as np

from solutions.k3_numerics import EPS, check_close, edge_sizes, to_bf16

STORAGE = {"float32": np.float32, "float16": np.float16, "bfloat16": np.float32}


def store(x, dtype):
    x = np.asarray(x, np.float32).astype(STORAGE[dtype])
    return to_bf16(x) if dtype == "bfloat16" else x


def reference_softmax(x):
    x = np.asarray(x, np.float64)
    m = x.max(axis=-1, keepdims=True)
    e = np.exp(x - m)
    return e / e.sum(axis=-1, keepdims=True)


def device_softmax(x, dtype, block=64, bug=None):
    """Two passes over the row in blocks of `block`: online max+sum, then normalize.
    bug: None | 'no_max_sub' (overflows for large inputs) | 'tail' (last partial block
    the sum) | 'fp16_acc' (accumulate in the storage dtype)."""
    x = store(x, dtype).astype(np.float32)
    rows, n = x.shape
    acc_t = np.float16 if bug == "fp16_acc" else np.float32
    out = np.empty_like(x)
    for r in range(rows):
        m = np.float32(-np.inf) if bug != "no_max_sub" else np.float32(0)
```

```
        s = acc_t(0)
        stop = n if bug != "tail" else (n // block) * block or n
        for k0 in range(0, stop, block):
            blk = x[r, k0:min(k0 + block, stop)]
            if bug != "no_max_sub":
                m_new = max(m, blk.max())
                s = acc_t(s * acc_t(np.exp(m - m_new)) + acc_t(np.exp(blk - m_new).sum
                m = m_new
            else:
                with np.errstate(over="ignore"):
                    s = acc_t(s + acc_t(np.exp(blk).sum()))
        with np.errstate(over="ignore", invalid="ignore"):
            out[r] = np.exp(x[r] - m) / np.float32(s)
    return store(out, dtype)


def tolerance_for(dtype, n):
    """(rtol, atol, rationale). Softmax outputs are in [0, 1]; the error comes from (a
    the *output* to the storage dtype (half an ulp: eps/2 relative) and (b) the length
    fp32 (~sqrt(n) * eps32). atol guards the tiny probabilities where relative error i
    out_rel = EPS[dtype]
    sum_rel = 4 * EPS["float32"] * math.sqrt(n)
    rtol = 2 * out_rel + sum_rel
    atol = max(out_rel / max(n, 1), 1e-7)
    why = (f"{dtype}: output rounding 2*eps={2 * out_rel:.2e} + fp32 sum over n={n}: "
           f"4*eps32*sqrt(n)={sum_rel:.2e}; atol=eps/n={atol:.2e} for near-zero probab
    return rtol, atol, why


def make_input(rows, n, dtype, seed, scale=3.0):
    rng = np.random.default_rng(seed)
    x = rng.standard_normal((rows, n)) * scale
    return store(x, dtype)


def run_differential(impl, cases=60, seed=0, block=64, dtypes=("float32", "float16", "
    """Returns (failures, tried). failures: [{'rows','n','dtype','seed','report'}]."""
    rng = random.Random(seed)
    sizes = edge_sizes(block, limit=4 * block)
    fails = []
    for i in range(cases):
        rows, n, dt, s = rng.choice([1, 2, 7]), rng.choice(sizes), rng.choice(dtypes),
        x = make_input(rows, n, dt, s, scale)
        rtol, atol, _ = tolerance_for(dt, n)
        rep = check_close(impl(x, dt), reference_softmax(x), rtol, atol)
        if not rep["ok"]:
            fails.append({"rows": rows, "n": n, "dtype": dt, "seed": s, "report": rep}
    return fails, cases


def smallest_failure(fails):
```

```
    return min(fails, key=lambda f: (f["rows"] * f["n"], ["float32", "float16", "bfloa


RUST_DTYPE = {"float32": "F32", "float16": "F16", "bfloat16": "Bf16"}


def emit_rust_test(f, name="softmax_matches_reference"):
    """A Rust regression test sketch: same graph compiled for both runtimes, compared
    justified tolerance. (Adapt the API calls to the current crate before submitting.)
    rtol, atol, why = tolerance_for(f["dtype"], f["n"])
    return f"""#[test]
fn {name}_{f['dtype']}_{f['rows']}x{f['n']}() {{
    // minimized by l5_differential; input seed = {f['seed']}
    // tolerance: {why}
    let mut cx = Graph::new();
    let x = cx.tensor(vec![{f['rows']}, {f['n']}], DType::{RUST_DTYPE[f['dtype']]});
    let y = x.softmax(1).output();
    let data = random_vec_seeded({f['rows'] * f['n']}, {f['seed']});
    let reference = run_reference(&mut cx, x, y, &data);
    let device = run_cuda(&mut cx, x, y, &data);
    assert_close(&device, &reference, /*rtol=*/{rtol:.3e}, /*atol=*/{atol:.3e});
}}
"""
```

在 Luminal 里：写一个真的

1. 选一个 op：softmax、`gelu`、`log_softmax`，或任何在 `src/logical/op/` 下的 primitive。
2. 在 `crates/luminal_cuda_lite/tests/` 下照着已有文件的写法：同一张图分别 compile 到 `ReferenceRuntime` 和 CUDA runtime，同样的数据，比较输出。`emit_rust_test` 生成的是骨架，其中 `random_vec_seeded`、`run_reference`、`run_cuda`、`assert_close` 要替换成仓库里实际的 helper。
3. 参数覆盖：`edge_sizes(64)` 那组长度、F32 / F16 / Bf16、一个大 scale 的用例。
4. PR 描述里写上：GPU、driver、Toolkit、commit、容差理由（直接用 `tolerance_for` 的 `why`）。

▼ 自测：如果 CUDA 和 Reference 差了 2 个 ulp，你会提 bug 吗?

先看容差理由。softmax 输出舍入到 fp16 就可能差 1 个 ulp，fp32 求和顺序不同又会带来 `√n·eps32` 量级的差异，所以 2 ulp 通常在合理范围内，不是 bug。真正值得报告的是：超出有理由的容差、NaN / inf 位置不同（`check_close` 的 NaN mismatch），或者误差随 n、随 scale 明显超线性增长——那往往意味着累加精度或数值稳定性有问题，就像 `fp16_acc` 和 `no_max_sub`。

实 战

Track X · 互补实验

同一个 workload 两边各走一遍：kernel 怎样生成和跑快，graph 怎样选对实现。

实战 · Track X · 互补实验 · 第 15 题

X1 共同实验：同一个 fused MLP，在 Triton 和 Luminal 两边量一遍

roadmap 说“最好的共同实验”是同一组 GEMM / fused MLP workload 两边各走一遍：先用 Triton 写出并 profile 一个 kernel，理解它的硬件成本；再看 Luminal 的 graph / search 如何为同一个计算选择执行实现。这是一个实践页，没有 check.sh，产出是一张对照表和一页分析。

120 min | 难度 中 | 实践 · 在 Triton / Luminal 仓库里动手，没有 check.sh | 单独讲解页

1. 实验设计
2. Triton 一侧
3. Luminal 一侧
4. 对照表模板
5. 分析要回答的问题

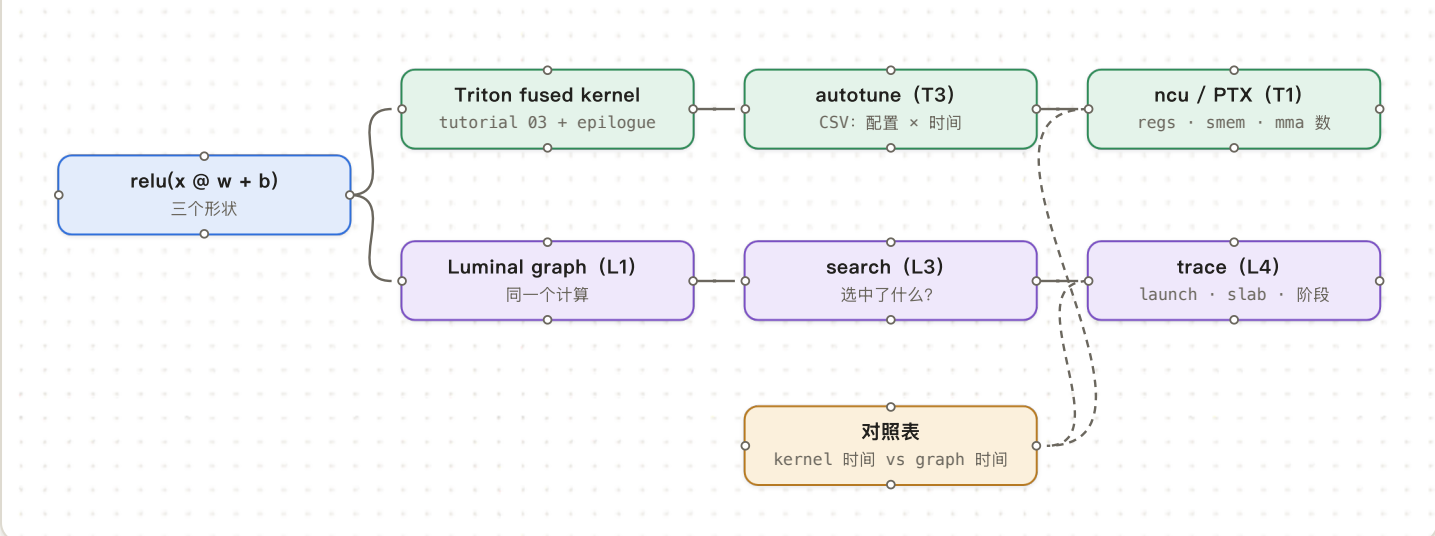
实验设计

workload：`y = relu(x @ w + b)`，fp16 输入、fp32 累加、fp16 输出；三个形状：

名字	M × K × N	代表什么
decode	16 × 4096 × 4096	小 batch 推理：memory / launch-bound
prefill	2048 × 4096 × 4096	大 batch：compute-bound
odd	1000 × 4095 × 1000	非整除：mask、tail、对齐

同一台机器、同一份环境快照（basics_env 的脚本），每个数都用 K2 的 harness：warmup、median、IQR，compile / search 时间单独列。

一个 workload，两条路



Triton 一侧

1. 在 tutorial 03 的 `matmul_kernel` 里加 `+ bias` 和 `relu` 的 epilogue（tutorial 已有 `ACTIVATION="leaky_relu"` 的例子）。
2. 正确性：K3 的 `check_close`，容差 `tolerance("float16", K)`。
3. 用 T3 的 `autotune` 扫三个形状，存 CSV；记录最佳配置的 `n_regs`、`n_spills`、`metadata.shared`。
4. 用 T1 的工具数 PTX 里的 `mma.sync` 和 `ldmatrix`，看 TTGIR 里 `ttg.convert_layout` 出现在哪。
5. 用 K1 的 `analyze` 算理论时间，和实测比较：decode 形状离 roofline 有多远？为什么？

Luminal 一侧

1. 写同一个图（L1 的 Rust 片段），在 ReferenceRuntime 上和 Triton 的结果对拍。
2. CUDA 编译，打开 `SEARCH_LOG=1 LUMINAL_SEARCH_OP_NAMES=1 LUMINAL_LOG_LLIR=1`：搜索选中的是 cuBLASLt（带 `RELU_BIAS` epilogue？）、自己生成的 kernel，还是多个 launch？
3. `LUMINAL_CUDA_PROFILE_RECAPTURE=1`：execute 时间里，kernel 本身占多少，runtime 开销占多少？
4. 把编译 + 搜索时间单独记下，算 L3 的 break-even：执行多少次之后，Luminal 的搜索成本被摊平？

对照表模板

形状	Triton 最佳配置	Triton kernel (µs)	Luminal 选中的实现	Luminal execute (µs)	Luminal compile+search (s)	cuBLAS 参考 (µs)	理论下界 (µs)
decode							
prefill							
odd							

类比 · 同一道菜，两种考法

Triton 一侧像考厨师的刀工：给你一道菜（一个 kernel），看你切得多细、火候多准（tile、layout、MMA）。Luminal 一侧像考主厨的统筹：同一道菜，是自己做还是用中央厨房的半成品（cuBLASLt），和别的菜怎么排（执行计划），整桌菜多久上齐（端到端）。两张成绩单放在一起，才知道这道菜的瓶颈在“切”还是在“排”。

分析要回答的问题

- 三个形状下，Luminal 选中的实现和 Triton 最佳 kernel 谁快？差距来自 kernel 本身，还是 launch / runtime 开销？
- decode 形状下，两边是不是都被 launch 开销主导？CUDA graph（Luminal 的 `cuda_graph.rs`）有没有起作用？
- odd 形状下，Triton 的 mask 和 Luminal 的选择各付出了多少代价？
- 如果 Luminal 选了 cuBLASLt：如果你的 Triton kernel 更快，那 Luminal 的搜索空间里缺了什么？这可能就是一个 issue（Week 13 的“深水区”候选）。

▼ 自测：为什么要把 compile / search 时间单独一列？

Triton 的 JIT + autotune 和 Luminal 的 e-graph + 搜索都是一次性成本，量级可能是秒到分钟；execute 是稳态成本，量级是微秒。混在一起报告，小 workload 上会得出“Luminal 慢一万倍”这种无意义的结论。分开报告，再用 break-even 把两者联系起来，才是 roadmap 要求的“compile 与 runtime 分离”。

Track P · 第一笔贡献

从 issue 到 merged PR：复现、定位、最小改动、测试、benchmark 证据。

实战 · Track P · 第一笔贡献 · 第 16 题

P1 贡献实战：issue 筛选、#370 复现、PR 模板、review 节奏

从“读懂了”到“合进去了”中间隔着一套流程：挑一个小而明确的 issue、先在 issue 里给出环境和最小复现、和维护者对齐方向、最小改动、补测试、给出可复现的证据、回应 review。这一页把这套流程落到 Triton 和 Luminal 的具体入口上，并用 Luminal #370 做一次完整演练。

180 min | 难度 中 | 实践 · 在 Triton / Luminal 仓库里动手，没有 check.sh | 单独讲解页

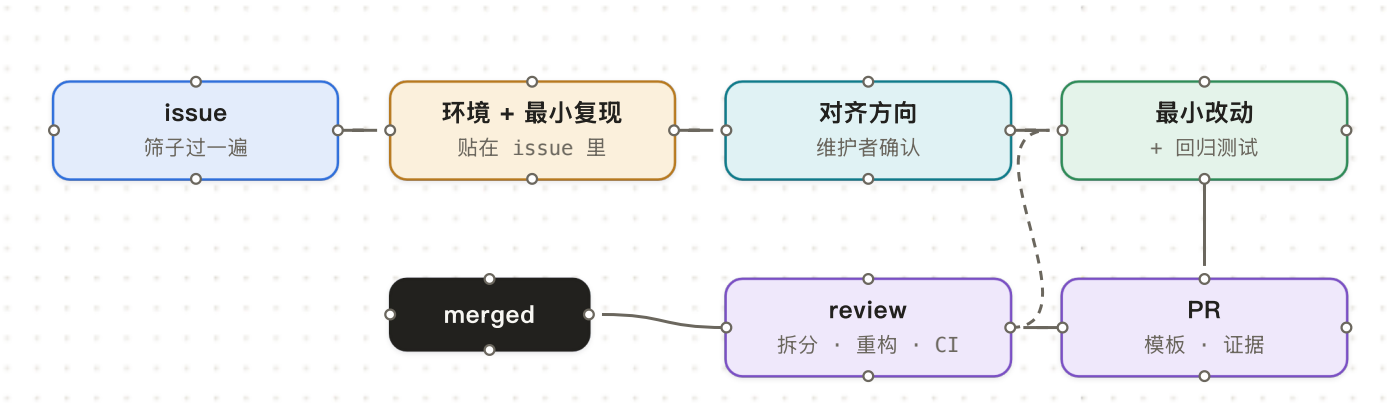
1. 挑任务：五个筛子
2. Triton 入口
3. Luminal 入口与 #370 演练
4. PR 应该长什么样
5. review 节奏（Week 12）

挑任务：五个筛子

筛子	问自己
模块	是我练过的部分吗（coalesce / layout / 测试 / CUDA backend / 对拍）？
硬件	能在 RTX 3070（SM86）上复现吗？需要 Hopper / Blackwell 的先放一放
复现性	我能写出一个 T4 / L5 那样的最小复现吗？
已有 PR	有没有人已经在做？搜 issue 编号和关键词
维护者响应	近期类似的 issue / PR 有人回复吗？

不要同时认领两个大规模 feature。 先在 issue 里提供环境与最小复现，问清预期方向，再动手。

一笔贡献的生命周期



Triton 入口

入口	建议
open issues	筛 bug、compiler、good first issue；优先能在 SM86 复现的 correctness / lowering / test 问题
`triton › CONTRIBUTING.md`	先确认测试规范、PR 流程；NVIDIA backend 由 core maintainers 负责，较大的设计改动应先讨论
`test/` 下的 lit 测试	最稳妥的第一步：补一个能稳定失败 / 通过的 regression test（T5），再提交针对性修复

Luminal 入口与 #370 演练

luminal #370：“luminal_cuda_lite fails to compile on CUDA 13.x (v1 graph dependency APIs missing)”，写这一页时仍是 open。roadmap 的建议是先用它做 CUDA 环境验证：仍然存在且范围小，就作为第一笔贡献候选；已经修复或与你的环境无关，就从 open issues 里挑一个 reference-vs-CUDA correctness test（L5）。

演练步骤：

1. **读 issue**：报告的是 CUDA 13 移除 / 改名了某些 v1 的 CUDA graph 依赖 API。Luminal 的 graph 捕获代码在 `luminal › cuda_graph.rs`；CUDA 绑定来自 cudarc 的 fork（`luminal › Cargo.toml 里的 cudarc 依赖`，feature 包含 `cuda-version-from-build-system`）。
2. **记录环境**：`nvcc --version`（Toolkit 是不是真的 13.x，而不是 `nvidia-smi` 显示的 13.x）、`CUDA_HOME`、`cargo -V`、`git rev-parse HEAD`。
3. **在当前 main 上复现**：`cargo build -p luminal_cuda_lite --features device`，保留完整日志。
4. **定位**：编译错误指向哪个符号？它在 cudarc 里还是在 Luminal 自己的代码里？CUDA 13 的 release notes 里这个 API 的替代是什么？
5. **评估范围**：是改 Luminal 一处调用（小），还是要改 cudarc fork（要和维护者确认）？
6. **在 issue 里回复**：环境、复现命令、错误摘录、你的定位和打算的修法，问维护者是否认可方向。

PR 应该长什么样

无论哪个项目，PR 描述固定包含：**Problem**（实际行为、预期、影响）、**Minimal reproducer**（代码、shape、dtype、GPU、commit SHA）、**Root cause**（哪个 pass / backend / rewrite / dispatch）、**Fix**（最小改动，为何不破坏其他路径）、**Tests**（regression、reference

comparison、相关 suite）、**Performance evidence**（如果宣称更快：设备、版本、warmup、重复次数、median / variance、compile 与 runtime 分离）。

```
## Summary
- What is broken or inefficient?
- Which compiler/backend stage is responsible?

## Reproducer
- Commit:
- GPU / driver / CUDA Toolkit:
- Shape / dtype:
- Minimal command:

## Root cause
- Relevant IR / pass / backend path:

## Fix
- Why this change is correct:
- Other backends / dtypes affected:

## Validation
- Tests:
- Before / after benchmark (if applicable):
```

> **类比 · 看病**
>
> 好的 issue 像一份清楚的病历：症状（实际 vs 预期）、什么时候发作（最小复现）、病人的基本情况（环境快照）。好的 PR 像一份手术方案：病因在哪（root cause）、只切该切的地方（最小改动）、术后怎么确认好了（tests）、有没有副作用（其他 backend / dtype）。维护者是主任医师：先让他们看病历、认可方案，再上手术台。

---

review 节奏（Week 12）

- 回应每一条评论；同意就改，不同意就给出数据或理由。
- reviewer 要求拆分，就拆：一个 PR 只做一件事。
- 补齐 CI、format（Triton 用 pre-commit；Luminal 用 `cargo fmt` / `cargo clippy`）。
- 性能 PR 给出稳定的 before / after（K2 的 `compare` 判定“faster”而不是“within noise”）。

▼ 自测：为什么建议第一笔贡献是“测试”而不是“优化”？

测试的价值容易验证（失败 → 修复后通过），改动小、风险低，维护者审起来快；它还迫使你搞懂相关代码路径和测试基础设施。优化需要稳定的 benchmark 证据、要说明对其他 backend / 形状 / dtype 没有回退，审查成本高，也更容易和维护者的规划冲突。roadmap 原话：“避免没有数据支持的性能优化”。

计划

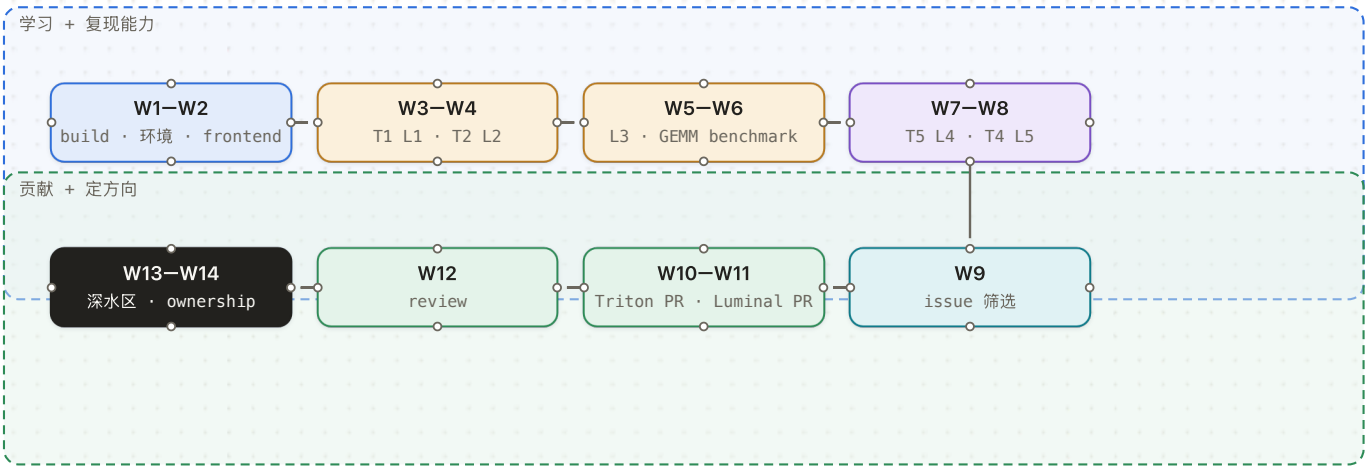
14 周计划与进度

14 周的双项目计划：前 8 周学习并建立复现能力，第 9–12 周各争取一笔小而真实的贡献，第 13 周起选一个深水区、形成长期 ownership。每周都对应这里的题目和一个验收标准。

1. 一张图
2. 每周
3. 进度追踪
4. 最后的执行建议

一张图

14 周



每周

周	Triton	Luminal	本仓库	验收
1	fork，记录 SHA / 环境，跑通简单 kernel	fork，跑通 ReferenceRuntime 示例；尝试 CUDA build，失败保留日志	basics_env，K2	两仓库都有可复现的 setup notes
2	vector add → TTIR	matmul graph → HLIR / logical graph	basics_triton、basics_luminal	两张编译流程图
3	operation、type、shape、canonicalization	primitive ops、shape tracker、graph traversal	**T1**、**L1**、K1	解释两边 IR 分别表达什么
4	blocked layout、layout conversion	e-class、rewrite、saturation、extraction	**T2**、**L2**	低层数据布局 vs 高层等价搜索的区别
5	`tl.dot` → MMA / PTX	候选生成、cost、search budget	**L3**、K4	GEMM codegen 与 graph search 对照笔记
6	GEMM benchmark	同硬件、同 dtype / shape 的 graph / execute 时间	**T3**、**X1**	benchmark 表 + 测量方法说明
7	lit / FileCheck、Python tests	selected plan → CUDA launch	**T5**、**L4**	两个项目各一条完整调用链
8	最小复现	differential test	**T4**、**L5**、K3	两边各一个可复现 correctness case
9	按模块、硬件、复现性、已有 PR、维护者响应筛 issue	同左；#370 环境验证	**P1**	每个项目一个小而明确的任务
10	第一笔贡献：correctness / test / lowering bug，IR diff + 最小复现	—	T4、T5	PR 或已获反馈的 draft
11	—	第一笔贡献：CUDA build 兼容、backend correctness 或 benchmark test	L5、P1	PR 或维护者确认的任务
12	回应 review，拆分 / 重构，补 CI / format / test	同左；性能 PR 给稳定 before / after	K2	至少一个 PR 进入实质 review
13	深水区候选：layout conversion、dot lowering、register / smem 行为	kernel selection、search cost、execution planning	—	一页 design note
14	总结 review、理解缺口、benchmark 结果	同左	—	明确的长期 ownership 方向

进度追踪

勾选状态保存在这个浏览器里（和题目的“标记完成”一样）。

- [ ] 记录 GPU、driver、CUDA Toolkit、Python、PyTorch、Rust 版本
- [ ] Triton 本地 build / kernel execution 成功
- [ ] Luminal ReferenceRuntime 示例成功
- [ ] Luminal CUDA backend build / execution 已验证或记录阻塞原因
- [ ] 画出 Triton frontend → TTIR → TTGIR → LLVM → PTX
- [ ] 画出 Luminal graph → logical IR → e-graph / search → LLIR → backend
- [ ] 完成 Triton IR dump 与 layout 练习（T1、T2）
- [ ] 完成 Luminal e-graph 与 search cost 练习（L2、L3）
- [ ] 建立同一 workload 的 benchmark harness（K2、X1）
- [ ] 两边各完成一个 correctness reproducer（T4、L5）
- [ ] 分别确认 Triton / Luminal 的贡献任务
- [ ] 提交 Triton PR 或 draft PR
- [ ] 提交 Luminal PR 或 draft PR
- [ ] 至少一个 PR 进入实质 review

> **类比 · 先学开车，再上高速，最后选一条常跑的路线**
>
> 前 8 周是驾校：每样操作（IR、layout、e-graph、搜索、测试）都要亲手做一遍，能稳定复现。第 9–12 周是上路：挑一条短而熟悉的路（小 issue）开一趟，接受教练（reviewer）的纠正。第 13 周以后，选一条你会天天跑的路线（一个模块），把它跑熟——长期 ownership 就是“这条路上的事大家会来问你”。

最后的执行建议

前 8 周以学习和建立复现能力为主；第 9–12 周分别争取一笔小而真实的贡献；第 13 周之后，依据维护者反馈和实际兴趣，选一个项目作为主攻，另一个维持每月一次的 issue / PR 参与。

附录：资源、PR 模板与参考资料

官方仓库与文档、论文、profiler 工具，以及本仓库的构建方式。所有源码链接都固定在 build/sources/pins.txt 里的 commit 上。

官方与核心资源

项目	链接
Triton	源码 · 文档 · `triton › CONTRIBUTING.md` · `triton › Gluon 入门` · Triton Puzzles
Luminal	源码 · `luminal › README` · Issues · PRs · 文档
egglog	egglog · egglog-python · egglog 论文 · egg 论文
NVIDIA	Nsight Compute · Nsight Systems · CUDA Programming Guide · PTX ISA
LLVM	FileCheck · lit · MLIR
论文	Triton（MAPL 2019） · Linear Layouts · ddmin

本仓库

```
content/        index.html 的各节（Markdown + ::: 容器 + flow 图 DSL）
hints/src/      每道题的单独讲解页源文件 → hints/*.html
mine/           你的实现（由 build/gen_stubs.py 从 solutions/ 生成）
solutions/      参考解
tests/          验收测试（./check.sh <题号>）
fixtures/t1/    T1 用的四层 IR 样例
examples/       egglog 小实验（L2）
build/          build.py · build_hints.py · sources.py（固定 commit 的源码链接） · gen_stubs.p
```

```
python3 -m venv .venv && .venv/bin/pip install pytest numpy markdown-it-py
./check.sh all --solutions
python3 build/sources.py fetch            # 下载被引用的源码文件并校验所有链接（需要 gh）
.venv/bin/python build/build.py           # index.html
.venv/bin/python build/build_hints.py     # hints/
.venv/bin/python build/gen_stubs.py       # 重新生成 mine/ 里缺失的桩文件（已有的不会覆盖）
```

和 gpu-kernel-drills 的关系

gpu-kernel-drills 练的是**写 kernel**（CUDA、Triton、FlashInfer、CuTe）；这里练的是**读和改编译器**。两边有几处直接相通：那边的 G1（BlockedLayout 计算器）是这边 T2 的前置；那边的 T4

（Triton GEMM）是这边 T3 / X1 的 kernel；那边的 FlashInfer 题对应 Luminal 搜索空间里的 FlashInfer 候选。