CUDA C++ → Triton → FlashInfer → CuTe · MoE / FP8 · MLX

Kernel Dev 训练营：以 FlashInfer 为主线

核心路线是 CUDA → Triton → 从零手写推理 attention → 对着 FlashInfer 的真实 API 实现并对拍，最后进入它的源码和贡献流程；进阶部分覆盖 CuTe 布局代数、MoE 与 FP8、Apple silicon 上的 MLX。另有一张“模型 × 硬件 × 当前最优 kernel”的全景地图和面试题库。

共 **25** 题 · 约 **23 小时 20 分** · 实测： **RTX 5090** + FlashInfer 0.7.2 · **Apple M2** + MLX 0.32 · Triton 题可在 **CPU** 上验收 · 下载 PDF · GitHub

已完成 0 / 25 · 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：整套路线怎么排、每道题怎么做、在哪台机器上跑，以及做完之后你应该能做到什么。

1. 目标
2. 路线：核心 → 进阶
3. 路线图
4. 每道题的结构
5. 仓库与命令
6. 在哪跑
7. 速成路线（约 24 小时）

目标

做完这套题，你应该能做到三件事：

1. **读懂并写出主流 kernel**：用 CUDA C++、Triton、CuTe 和 MLX / Metal 各写过一遍 elementwise、reduction、GEMM、attention、MoE，知道每一层优化在解决什么问题，并能用 roofline 说清楚离硬件上限还有多远。
2. **对付面试**：说得清 FlashAttention、paged attention、split-KV、GQA、MoE grouped GEMM、FP8 block scaling 的原理和取舍，以及**不同模型、不同硬件上哪个 kernel 是当前的最佳选择**、为什么。
3. **给 FlashInfer 提 PR**：知道它的源码结构、plan/run 模式、JIT 系统、测试和 benchmark 规范，并且已经亲手实现过它的四个 API（F1–F4），能和官方实现逐数值对拍、比较性能。

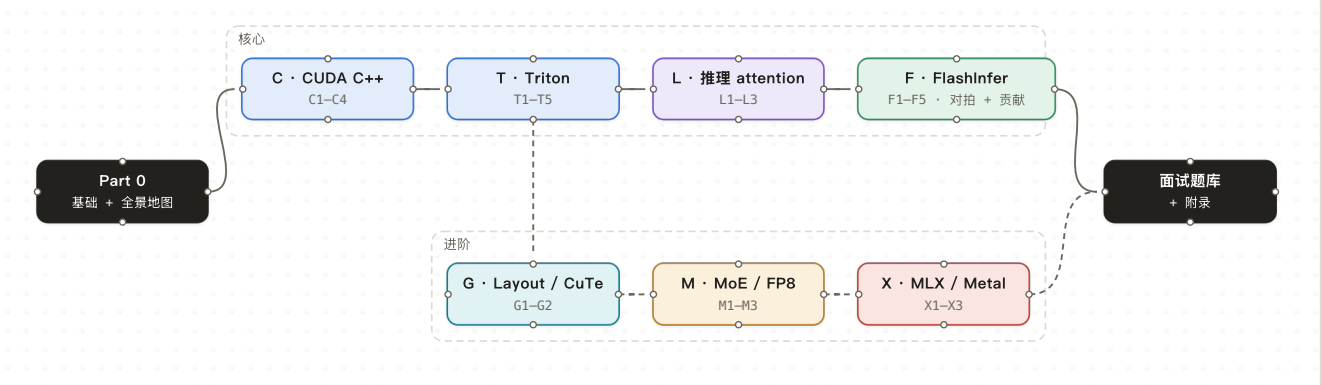
路线：核心 → 进阶

```
Part 0  基础 + 全景地图（模型 × 硬件 × 当前最优 kernel）
   │
核心 ├─ C  CUDA C++      C1–C4  线程映射、合并访存、warp 原语、GEMM tiling
     ├─ T  Triton        T1–T5  block 级编程、融合、tl.dot、FlashAttention
     ├─ L  推理 attention L1–L3  paged KV、decode + GQA、split-KV（从零手写）
     └─ F  FlashInfer    F1–F5  CSR 分页格式、merge_state、融合 norm、采样、贡献实战
   │
进阶 ├─ G  Layout / CuTe G1–G2  BlockedLayout / SliceLayout, CuTe 布局代数
     ├─ M  MoE 与量化     M1–M3  路由对齐、fused MoE grouped GEMM、FP8 block scaling (I
     └─ X  MLX / Metal   X1–X3  Apple GPU 上的 elementwise、simdgroup 归约、simdgroup
   │
面试题库 · 附录（下一步、阅读路线）
```

为什么这样排：FlashInfer 的 kernel 本身就是 CUDA C++（以及越来越多的 CuTe DSL），它的 Python 生态、vLLM / SGLang 的大量 kernel 又是 Triton。所以先用 C、T 打基础，再用 L 从零写出 paged attention，最后在 F 里对着 FlashInfer 的真实接口实现并对拍。进阶部分覆盖面试和实际工作中同样常见的其他方向：CuTe（CUTLASS 3/4、FlashAttention-4、DeepGEMM 都基于它）、MoE 与低精度、Apple 设备上的 MLX。

路线图

整套路线



实线是建议的主线顺序；虚线的进阶部分可以穿插在核心之后的任何时间做。

每道题的结构

小节	作用
背景	这个 kernel 在真实系统里出现在哪，为什么值得写
要用到的知识	只讲这题需要的新语法和硬件概念

小节	作用
题目要求	函数签名、shape / dtype、边界条件、验收标准
题干	`mine/` 里的骨架文件全文：你只填 `TODO`
验收	测试命令和测试代码。F 题额外和 FlashInfer 官方实现对拍
提示 L1 / L2 / L3	卡住 10–15 分钟再按级别展开；L3 基本就是答案
参考解	每一行都有注释的完整实现，写完再看
测完之后	实测数字（RTX 5090 / Apple M2）、可以做的实验、面试追问

仓库与命令

```
./check.sh f2                # 测你写的 mine/f2_merge_state.py
./check.sh f2 --solutions    # 测参考解（确认环境没问题）
./check.sh f2 --bench        # 性能：你的实现 vs FlashInfer / PyTorch（需要 NVIDIA GPU
./check.sh x2                # X 题自动使用 .venv-mlx（Apple silicon）
./check.sh all               # 全部
```

在哪跑

环境	能做什么
只有 CPU	T、L、F1–F4、G、M1–M2 的正确性都能验收：没有 GPU 时自动打开 `TRITON_INTERPRET=1`。macOS 上没有 triton wheel，可以用 Linux 容器。
自己的 NVIDIA GPU（比如 RTX 5090）	全部 C / T / L / F / M 题和 benchmark。已有装了 torch + triton 的 Python 即可，不需要系统级 CUDA toolkit：`./setup_env.sh /path/to/python`，以后每个新 shell 先 `source env.sh`。FlashInfer：`pip install flashinfer-python`。
Apple silicon Mac	X 题：`python3 -m venv .venv-mlx && .venv-mlx/bin/pip install mlx numpy pytest`（必须是 arm64 的 Python）。
Colab T4 / 云 GPU	T4 是 sm_75：bf16 和 FP8 测试会自动跳过（FP8 需要 sm_89+）。H100 / B200 上才能跑 FlashAttention-3/4、DeepGEMM、wgmma / tcgen05 相关内容。

页面里的“实测”数字来自：RTX 5090（sm_120）+ torch 2.13（CUDA 13.0）+ Triton 3.7 + FlashInfer 0.7.2；Apple M2（24 GB）+ MLX 0.32。

速成路线（约 24 小时）

次	内容	时长
1	Part 0 基础 + 全景地图	2 h
2	C1 + C2 + C3	2 h
3	C4	1.5 h
4	T1 + T2 + T3	1.5 h
5	T4 + T5	2.5 h
6	L1 + L2 + L3	2.5 h
7	F1 + F2 + F3	2 h
8	F4 + F5（贡献实战，可以拆成多次）	3 h
9	G1 + G2	1.5 h
10	M1 + M2 + M3	3 h
11	X1 + X2 + X3	2.5 h

只有一周时：Part 0 → T1 → T2 → T5 → L2 → L3 → F1 → F2 → F4 → M3，再读一遍面试题库。

Part 0 · 基础 1

GPU 执行模型与内存层次

写 kernel 之前必须在脑子里有的那张图：线程怎么组织、数据住在哪、一个 kernel 的速度上限由什么决定。

1. 线程的层级：grid → block → warp → thread
2. 内存层次：越近越快、越小
3. 性能上限：roofline
4. 图解与类比
5. 源码与资料

线程的层级：grid → block → warp → thread

层级	是什么	关键数字（NVIDIA）
thread	最小执行单位，有自己的寄存器	每线程最多 255 个 32-bit 寄存器

层级	是什么	关键数字（NVIDIA）
warp	32 个线程，**同一条指令同时执行**（SIMT）	永远是 32
block（CUDA）/ **program**（Triton）	一组 warp，被调度到**同一个 SM** 上，可以共享 shared memory、用 `__syncthreads()` 同步	最多 1024 线程
grid	一次 kernel launch 的所有 block，彼此之间**没有**同步手段（除了结束）	`gridDim.x` 最大 $2^{31}-1$

SM（Streaming Multiprocessor）是硬件上的“核”：A100 有 108 个，H100 SXM 有 132 个，T4 有 40 个，L4 有 58 个。一个 SM 同时驻留多个 block，每个时钟周期从就绪的 warp 里挑一个发射指令。某个 warp 在等内存时，其他 warp 顶上，这就是 GPU 隐藏延迟的方式。所以需要**足够多的 warp 在飞**：这个比例叫 **occupancy**（驻留 warp 数 / SM 上限）。寄存器和 shared memory 用得越多，能同时驻留的 block 越少。

> **warp divergence**
>
> 同一个 warp 里的线程走了 `if` 的不同分支时，两个分支会**串行**执行，各自屏蔽掉另一半线程。边界检查（`if (i < n)`）只影响最后一个 warp，代价很小；但依赖数据的分支（比如按 token 类型走不同代码）可能让效率减半。

内存层次：越近越快、越小

存储	作用域	容量（量级）	带宽 / 延迟（量级）
寄存器	线程私有	每 SM 64K × 32-bit = 256 KB	最快，约 1 cycle
shared memory / L1	block 内共享	每 SM 约 100–228 KB（A100 164 KB，H100 228 KB）	约 20–30 cycle
L2 cache	全 GPU	几十 MB（A100 40 MB，H100 50 MB）	
HBM / GDDR（global memory）	全 GPU，kernel 之间持久	16–80 GB	约 400–800 cycle；带宽见下表

合并访存（coalescing）

global memory 以 **32 字节 sector** 为单位传输。一个 warp 发出一条 load 指令时，硬件把 32 个线程的地址合并成尽量少的 sector：

- 32 个线程读 32 个**连续**的 float（128 字节）：4 个 sector，100% 有效。

- 32 个线程各读相隔 4 KB 的 float：32 个 sector，每个只用 4 字节，有效率 12.5%。

所以写 kernel 的第一条规则：**让相邻线程（`threadIdx.x` 相邻）访问相邻地址**。C3（转置）整道题都在处理这件事。

---

shared memory 的 bank

shared memory 分成 **32 个 bank**，每个 bank 每周期服务一个 4 字节字。地址 `a`（按 4 字节计）落在 bank `a % 32`。同一 warp 里多个线程访问**同一 bank 的不同地址**时会串行化（bank conflict）；访问**同一个地址**则是广播，没有冲突。经典的规避手段是给二维数组加一列 padding：`tile[32][33]`。

性能上限：roofline

一个 kernel 要么被**内存带宽**卡住，要么被**算力**卡住。判断看**算术强度**（arithmetic intensity）：

```
AI = FLOPs / 从 global memory 读写的字节数
```

和硬件的“拐点”`峰值 FLOP/s ÷ 峰值带宽` 比较：AI 低于拐点就是 memory-bound，优化方向是**少读写字节**（融合、复用、低精度）；高于拐点是 compute-bound，优化方向是**用好 Tensor Core**、减少非矩阵运算。

GPU	显存带宽	FP16/BF16 Tensor Core（稠密）	FP32（非 Tensor Core）	拐点（FP16）
T4	~320 GB/s	~65 TFLOP/s	~8 TFLOP/s	~200 FLOP/B
L4	~300 GB/s	~121 TFLOP/s	~30 TFLOP/s	~400 FLOP/B
A100 80GB SXM	~2.0 TB/s	~312 TFLOP/s	~19.5 TFLOP/s	~156 FLOP/B
H100 SXM	~3.35 TB/s	~990 TFLOP/s	~67 TFLOP/s	~295 FLOP/B
RTX 5090	~1.79 TB/s	~210 TFLOP/s（FP32 累加；实测 cuBLAS ≈ 218）	~105 TFLOP/s	~117 FLOP/B

（厂商公开峰值的约数，实际能达到的带宽一般是峰值的 80–90%。）

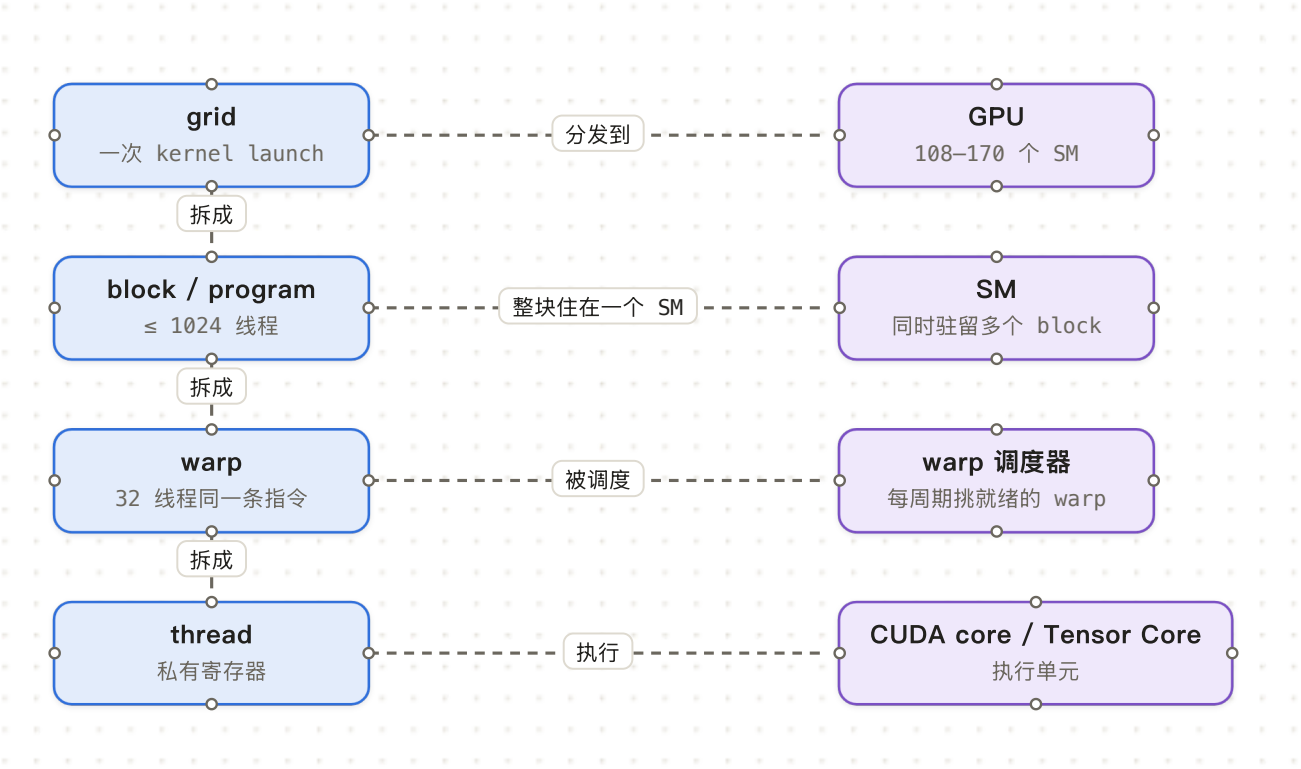
几个常见 kernel 的 AI，记住量级就够了：

kernel	AI（FLOP / byte）	结论
vector add（fp32）	1 / 12	永远 memory-bound：衡量标准是 **GB/s**
softmax、LayerNorm	~1	memory-bound：理想是“读一次、写一次”
GEMM（n×n×n，fp16）	~n / 3	n 较大时 compute-bound：衡量标准是 **TFLOP/s**
decode attention（每 token）	~1（GQA 时 ×group）	memory-bound：瓶颈是读 KV cache
prefill attention	~S	compute-bound

> **衡量一个 kernel 好不好**
>
> memory-bound 的 kernel 算 **有效带宽 = 必须读写的字节 / 时间**，和峰值带宽比；compute-bound 的算 **TFLOP/s**，和峰值算力比。只说“比 PyTorch 快 1.3 倍”是不够的：要能说出离硬件上限还差多少。`bench.py` 对每题都打印这两种数字之一。

图解与类比

软件层级 ↔ 硬件层级



左边是你写代码时的抽象，右边是它们在硬件上落在哪里。block 内的线程能共享 shared memory、能同步，正是因为整个 block 住在同一个 SM 上。

内存层次：数据从远到近



每往右一级，容量小一两个数量级、速度快一个数量级。几乎所有优化都在做一件事：让一个从 HBM 读来的数，在更近的地方被用尽可能多次。

类比 · 厨房的三层储物

HBM 是街对面的仓库（大、远），L2 是楼下储藏室，shared memory 是灶台旁的料理台，寄存器是你手里正拿着的东西。 好厨师不会每切一刀菜就跑一趟仓库：一次把一筐菜搬到料理台（tiling），在台上反复取用（复用）。kernel 优化说到底就是“少跑仓库”。

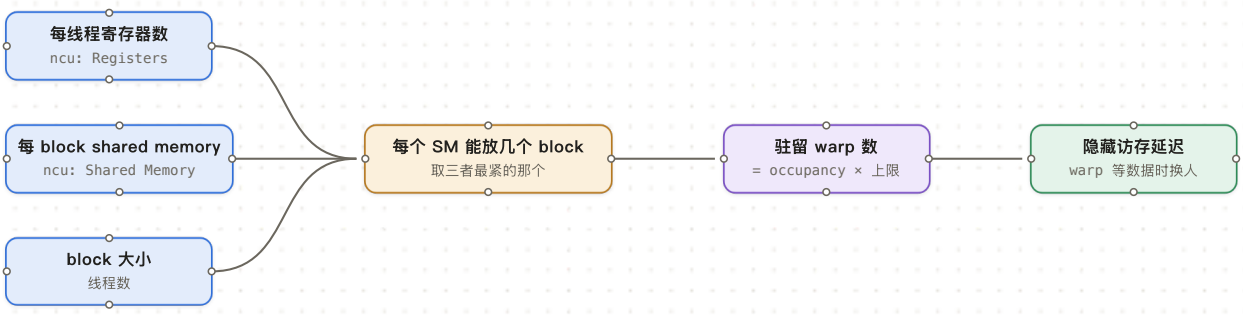
类比 · warp 是一支龙舟队

32 名桨手听同一个鼓点划桨（SIMT：同一条指令）。如果一半人要左转、一半人要右转（分支发散），只能先让左边那半划、右边那半等，再反过来，速度减半。 某支队在等补给（访存）时，调度器马上换另一支队上场：这就是为什么 SM 上要同时驻留足够多的 warp（occupancy）。

类比 · roofline 是工厂的两个上限

工厂产量受两件事限制：原料运进来的速度（带宽）和机器加工的速度（算力）。每吨原料需要的加工量就是算术强度： 加工量少（elementwise、decode）时，机器总在等原料：memory-bound；加工量大（大 GEMM、prefill）时，原料堆在门口等机器：compute-bound。先判断卡在哪边，再决定是修路还是买机器。

occupancy：资源决定能同时驻留多少 warp



类比 · 合并访存像拼车

一个 warp 发出一次 load，就像 32 个人一起叫车：如果大家住同一栋楼（地址连续），4 辆车（4 个 32 字节 sector）就能全部拉走；如果 32 个人散住在城市各处（跨步访问），就要派 32 辆车，每辆只坐一个人。车（带宽）的数量是固定的，坐不满就是浪费。

源码与资料

文档：CUDA C++ Programming Guide · CUDA C++ Best Practices Guide · How to Access Global Memory Efficiently · Using Shared Memory in CUDA C/C++ · Roofline model

白皮书：NVIDIA H100 Tensor Core GPU Architecture · NVIDIA Blackwell Architecture

Part 0 · 基础 2

CUDA C++ 语法速查

CUDA C++ 里写 kernel 只需要十几个关键字。这里按“定义 → 启动 → 索引 → 同步 → warp 原语 → 接进 PyTorch”的顺序过一遍。

1. 定义和启动 kernel
2. shared memory 与同步
3. warp 级原语
4. 原子操作
5. 接进 PyTorch：load_inline
6. 图解与类比
7. 源码与资料

定义和启动 kernel

```
// __global__：在 GPU 上执行、从 CPU 启动的函数，返回值必须是 void
__global__ void scale_kernel(const float* __restrict__ x, float* __restrict__ y, fl
    int64_t i = (int64_t)blockIdx.x * blockDim.x + threadIdx.x;   // 全局线程编号
    if (i < n) y[i] = a * x[i];                                    // 边界检查
}

// __device__：只能在 GPU 上被其他 GPU 函数调用的辅助函数
__device__ __forceinline__ float square(float v) { return v * v; }

// 启动：<<<grid, block, 动态 shared memory 字节数, stream>>>
int threads = 256;
```

```cpp
int64_t blocks = (n + threads - 1) / threads;          // 向上取整，覆盖所有元素
scale_kernel<<<blocks, threads, 0, stream>>>(x, y, 2.0f, n);
```

内建变量	含义
`threadIdx.{x,y,z}`	线程在 block 内的坐标
`blockIdx.{x,y,z}`	block 在 grid 内的坐标
`blockDim.{x,y,z}`	block 的形状（每维线程数）
`gridDim.{x,y,z}`	grid 的形状（每维 block 数）

二维 launch 用 `dim3`：`dim3 block(32, 8); dim3 grid(ceil(N/32), ceil(M/8));`。约定 **`threadIdx.x` 对应内存里最快变化的维度**（行主序矩阵的列），这样一个 warp 的 32 个线程访问连续地址。

> **索引溢出**
>
> `blockIdx.x * blockDim.x` 是 32-bit 无符号乘法，元素数超过 2^{32} 时会溢出。先转成 `int64_t` 再乘。行主序下标 `row * N + col` 同理。

---

grid-stride loop

```cpp
for (int64_t i = blockIdx.x * (int64_t)blockDim.x + threadIdx.x; i < n; i += (int64
    ...
}
```

grid 大小不再需要覆盖 n：可以固定成“每个 SM 若干个 block”，每个线程循环处理多个元素。好处是 launch 配置与数据大小无关，block 数少了调度开销也小，每个线程的累加（如 reduction）还可以留在寄存器里。

---

shared memory 与同步

```cpp
__global__ void k(...) {
    __shared__ float tile[32][33];   // 静态大小，block 内所有线程共享
    tile[threadIdx.y][threadIdx.x] = ...;
    __syncthreads();                 // block 内所有线程到达这里之后才继续
    ... = tile[threadIdx.x][threadIdx.y];
}
```

- `__syncthreads()` 必须被 block 内**所有**线程执行到。把它放进只有部分线程会进入的 `if` 里会死锁或出现未定义行为。

- 典型模式：**load → sync → compute → sync**。第二个 sync 防止下一轮迭代的 load 覆盖别人还在读的数据。

---

warp 级原语

warp 内 32 个线程可以直接交换寄存器里的值，不经过 shared memory：

```
float v = ...;
// 从 lane (laneid + offset) 读 v；超出范围的 lane 读到自己的值
v += __shfl_down_sync(0xffffffff, v, 16);   // 0xffffffff：参与的 lane 掩码（全部 32 个
v += __shfl_down_sync(0xffffffff, v, 8);
...                                          // 5 轮后 lane 0 持有 32 个值的和
float s = __shfl_xor_sync(0xffffffff, v, 1); // butterfly：与 lane ^ 1 交换，所有 lane
```

`lane = threadIdx.x % 32`，`warp = threadIdx.x / 32`（一维 block 时）。

原子操作

`atomicAdd(float* addr, float v)`：多个线程 / block 安全地加到同一地址。代价是同一地址上的原子操作会串行化，所以一般“先在 block 内归约，最后每个 block 只做一次 atomic”。浮点加法不满足结合律，**原子加的结果每次运行可能差最后几位**（非确定性）。

接进 PyTorch：`load_inline`

所有 CUDA 题都用这个模式，你只需要写 `.cu` 文件；`mine/<题>.py` 负责编译和导出：

```
#include <torch/extension.h>
#include <ATen/cuda/CUDAContext.h>       // at::cuda::getCurrentCUDAStream()
#include <c10/cuda/CUDAException.h>      // C10_CUDA_KERNEL_LAUNCH_CHECK()

__global__ void my_kernel(const float* x, float* y, int64_t n) { ... }

torch::Tensor my_op(torch::Tensor x) {                // host 函数：参数检查 + 分配 +
    TORCH_CHECK(x.is_cuda() && x.is_contiguous(), "...");
    auto y = torch::empty_like(x);
    my_kernel<<<blocks, threads, 0, at::cuda::getCurrentCUDAStream()>>>(
        x.data_ptr<float>(), y.data_ptr<float>(), x.numel());
    C10_CUDA_KERNEL_LAUNCH_CHECK();                   // launch 配置错误会在这里报出来
    return y;
}
```

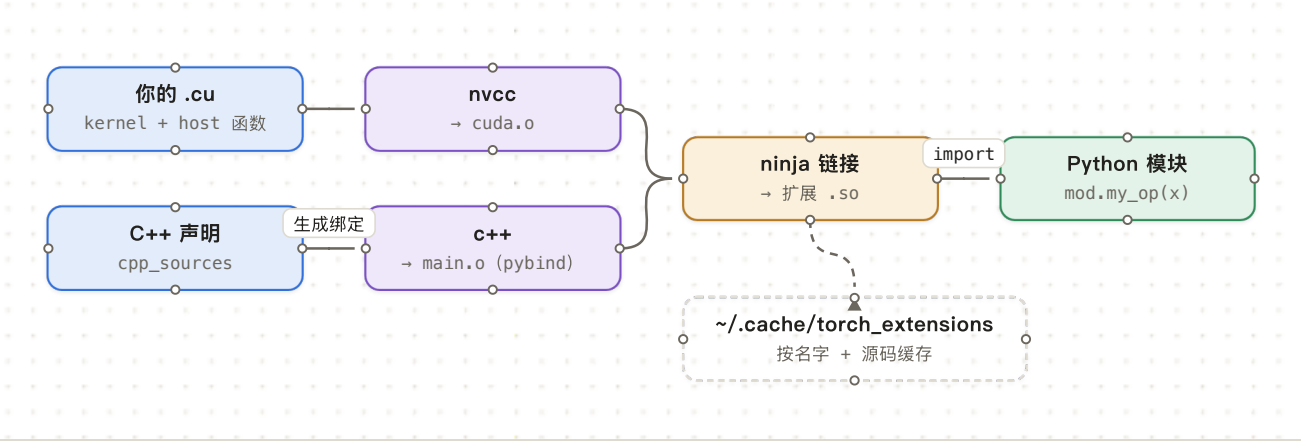
```
from torch.utils.cpp_extension import load_inline
mod = load_inline(name="my_ext", cpp_sources="torch::Tensor my_op(torch::Tensor x);"
```

```
                    cuda_sources=open("my.cu").read(), functions=["my_op"])
y = mod.my_op(x)
```

- 用 PyTorch 当前的 stream 启动，kernel 才会和前后的 torch 操作正确排序。
- 第一次编译约 30–60 秒，之后命中缓存（`~/.cache/torch_extensions`）。
- kernel 内部的越界访问通常不会立刻报错，而是在**下一个**同步点报 `illegal memory access`。调试时设 `CUDA_LAUNCH_BLOCKING=1`，或者用 `compute-sanitizer python -m pytest ...`。

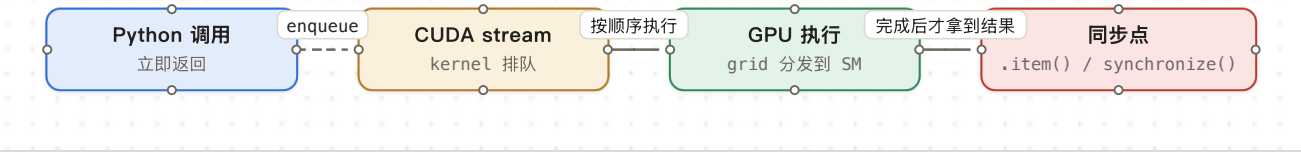
图解与类比

load_inline：一个 .cu 文件怎么变成 torch 算子



第一次调用要编译（30–60 秒），之后直接加载缓存。本仓库的 `common/cuda_ext.py` 把源码哈希放进模块名，所以改完 .cu 会自动重新编译。

kernel launch 是异步的



所以用 `time.time()` 包住一次调用，测到的只是“下单”的时间；越界访问的报错也常常在后面某个同步点才出现。

> **类比 · launch 就是给工厂下订单**
>
> `kernel<<<...>>>` 像在网上下单：点完“提交”就返回了，货（结果）还在路上。要拿到货（读结果、打印、`.item()`），就得等快递到门口（同步）。 所以计时要在“收货”时按秒表（CUDA event + synchronize），而不是在“下单”时。

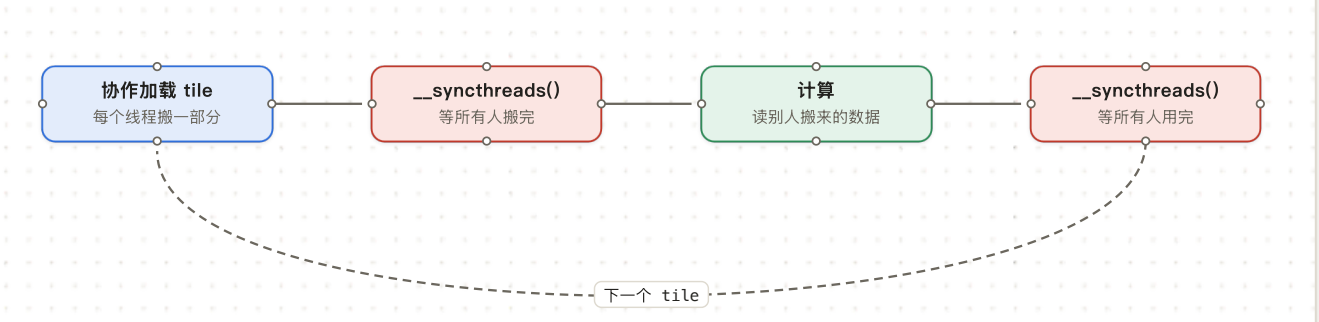
类比 · __syncthreads 是集体合影

摄影师要等所有人都站好才按快门。如果有人被安排在 `if` 分支里永远不会来，摄影师就一直等下去（死锁）。所以 `__syncthreads()` 必须放在所有线程都会执行到的地方。

类比 · grid-stride loop 是固定人数的流水线

不是“来多少箱货就雇多少工人”，而是雇固定数量的工人（SM 数 × k 个 block），每人做完一箱就去传送带上拿下一箱（`i += 总线程数`）。货多货少都用同一套班底，工人的私人账本（寄存器累加）还能一直记下去。

shared memory 的标准节奏



第二个 sync 很容易漏：没有它，跑得快的线程会在别人还在读的时候，把下一个 tile 写进同一块 shared memory。

类比 · atomicAdd 像只有一个窗口的银行

很多人往同一个账户存钱，只有一个柜员窗口（同一地址上的原子操作会串行）：人越多队越长。所以先让每个小组把钱凑齐（block 内归约），每组只派一个人去排队。 浮点数的“零头”跟存钱顺序有关，谁先到窗口不确定，所以每次结果的最后几位可能不同。

源码与资料

文档：CUDA C++ Programming Guide · An Even Easier Introduction to CUDA · Compute Sanitizer · PTX ISA

源码：`pytorch › load_inline` · `cudasamples › vectorAdd.cu` · `cudasamples › reduction_kernel.cu` · `flashinfer › vec_dtypes.cuh`（FlashInfer 的向量化加载工具）

Triton 语法速查（附 Gluon 速览）

Triton 的心智模型是“一个 program 处理一块数据”：你写 block 级的向量代码，编译器负责把它拆给线程、安排 shared memory 和流水线。

1. 与 CUDA 的对应
2. 最小的 kernel
3. 规则与常用 API
4. 性能相关的 launch 参数
5. 调试
6. Gluon 速览
7. 图解与类比
8. 源码与资料

与 CUDA 的对应

	CUDA	Triton
编程单位	一个**线程**处理一个元素	一个 **program** 处理一个 **block**（向量 / 矩阵块）
编号	`blockIdx` + `threadIdx`	`tl.program_id(axis)`；没有 threadIdx
一块数据的偏移	自己算	`pid * BLOCK + tl.arange(0, BLOCK)`
越界	`if (i < n)`	`mask=offs < n` 传给 load / store
shared memory、同步、合并访存、向量化	手写	编译器自动处理
warp 数	`blockDim`	launch 参数 `num_warps`
矩阵乘	WMMA / mma PTX	`tl.dot`

最小的 kernel

```
import torch, triton
import triton.language as tl

@triton.jit
def add_kernel(x_ptr, y_ptr, out_ptr, n, BLOCK: tl.constexpr):
    pid = tl.program_id(axis=0)
    offs = pid * BLOCK + tl.arange(0, BLOCK)      # [BLOCK] 个 int32 偏移
    mask = offs < n
    x = tl.load(x_ptr + offs, mask=mask, other=0.0)
    y = tl.load(y_ptr + offs, mask=mask, other=0.0)
    tl.store(out_ptr + offs, x + y, mask=mask)
```

```
def add(x, y):
    out = torch.empty_like(x)
    n = x.numel()
    grid = (triton.cdiv(n, 1024),)                # program 数；也可以是 lambda meta
    add_kernel[grid](x, y, out, n, BLOCK=1024)    # torch.Tensor 自动变成指针
    return out
```

规则与常用 API

- `tl.constexpr`：编译期常量。block 大小、head_dim、是否 causal 这类决定代码形状的参数都要标成 constexpr；每种取值编译出一个独立的 kernel。
- **block 形状必须是 2 的幂**：`tl.arange(0, BLOCK)` 的 BLOCK 必须是 2 的幂。行长 N 不是 2 的幂时，用 `triton.next_power_of_2(N)` 取上界，再用 mask 屏蔽多出的部分。
- **指针算术 + 广播**造二维块：`ptrs = base + rows[:, None] * stride + cols[None, :]` 得到 `[R, C]` 的指针块，`tl.load(ptrs, mask=...)` 读出 `[R, C]` 的值。
- **dtype**：`tl.load` 返回指针指向的类型；`.to(tl.float32)` 转换；存储前转回 `out_ptr.dtype.element_ty`。
- **masked load 的 `other`**：被屏蔽位置的值。做 max 时填 `-inf`，做 sum 时填 `0`。

类别	API
归约	`tl.sum(x, axis)`、`tl.max`、`tl.min`、`tl.argmax`
逐元素	`tl.exp`、`tl.log`、`tl.sqrt`、`tl.sigmoid`、`tl.maximum`、`tl.where(cond, a, b)`
构造	`tl.zeros(shape, dtype)`、`tl.full(shape, value, dtype)`、`tl.arange`
矩阵乘	`tl.dot(a, b, acc)`：`[M,K] @ [K,N]`，M、N、K 都至少 16；fp16 / bf16 输入，fp32 累加
形状	`x[:, None]`、`tl.trans(x)`、`tl.reshape`、`tl.broadcast_to`
工具	`tl.cdiv`、`triton.cdiv`、`triton.next_power_of_2`、`tl.static_assert`、`tl.device_print`

性能相关的 launch 参数

- `num_warps`：每个 program 的 warp 数（默认 4）。block 越大，需要越多 warp 来分摊。
- `num_stages`：软件流水线深度。循环里的 `tl.load` 会被提前若干轮发出，经 shared memory 缓冲，计算和访存重叠。主要影响 GEMM / attention 这类带 K 循环的

kernel。

- `@triton.autotune(configs=[triton.Config({...}, num_warps=.., num_stages=..)], key=[...])`：对每个新的 `key`（比如 M、N、K）把所有 config 各跑一遍，缓存最快的那个。

调试

- `TRITON_INTERPRET=1`：在 CPU 上逐 program 解释执行，可以在 kernel 里 `print`、打断点。本仓库没有 GPU 时自动打开。
- `tl.device_print("x", x)`：GPU 上打印；`tl.static_print`：编译期打印 constexpr 和 shape。
- 看编译产物：`kernel[grid](...)` 返回的对象有 `.asm["ttir"]`、`["ttgir"]`、`["ptx"]`。`ttgir` 里能看到编译器给每个张量选的 layout，比如 `#blocked`、`#mma`、`#shared`，这正是 G1 要讲的东西。

Gluon 速览

Gluon 是 Triton 下面一层的语言（`triton.experimental.gluon`）：语法和 Triton 很像，但 **layout 由你显式指定**，shared memory 的分配、异步拷贝、warp specialization 也由你控制。Triton 编译器替你做的那些决定，在 Gluon 里由你自己写。

```
# 示意：Gluon API 仍是 experimental，参数名以你所装版本的官方 tutorial 为准
from triton.experimental import gluon
from triton.experimental.gluon import language as gl

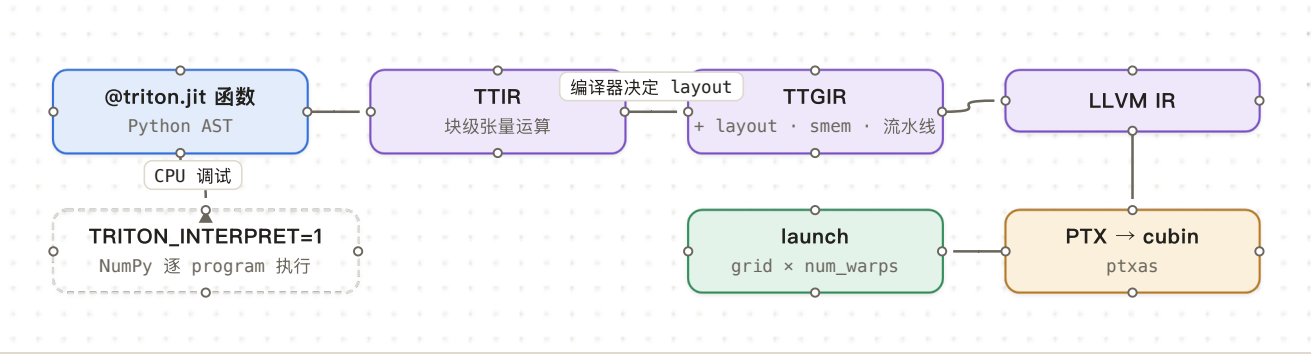
@gluon.jit
def copy_kernel(src, dst, n, BLOCK: gl.constexpr, layout: gl.constexpr):
    pid = gl.program_id(0)
    offs = pid * BLOCK + gl.arange(0, BLOCK, layout=layout)   # 偏移张量带着 layout
    mask = offs < n
    gl.store(dst + offs, gl.load(src + offs, mask=mask), mask=mask)

layout = gl.BlockedLayout(size_per_thread=[4], threads_per_warp=[32], warps_per_cta=
copy_kernel[(triton.cdiv(n, 512),)](src, dst, n, BLOCK=512, layout=layout, num_warp
```

`BlockedLayout(size_per_thread=[4], threads_per_warp=[32], warps_per_cta=[4])` 的意思是：每个线程拿 4 个连续元素（正好一次 16 字节的向量化 load），一个 warp 覆盖 128 个元素，4 个 warp 覆盖 512 个。**G1** 用纯 Python 把这个映射算清楚：哪个元素在哪个 warp、哪个 lane、第几个寄存器，以及它对合并访存和归约代价的影响。理解了这一层，读 `ttgir` 和写 Gluon 都会容易很多。

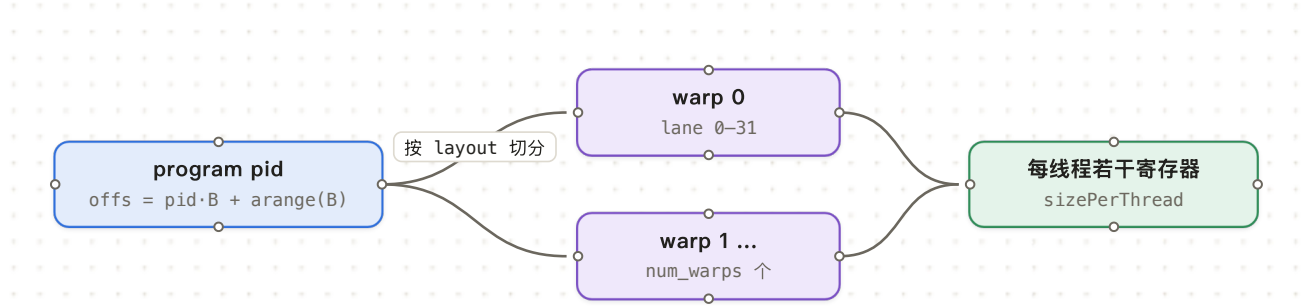
图解与类比

Triton 的编译管线



你只写到 TTIR 这一层（块级运算）；TTGIR 里出现的 `#blocked`、`#mma`、`#shared` 就是编译器替你做的决定，G1 会教你读懂它们。

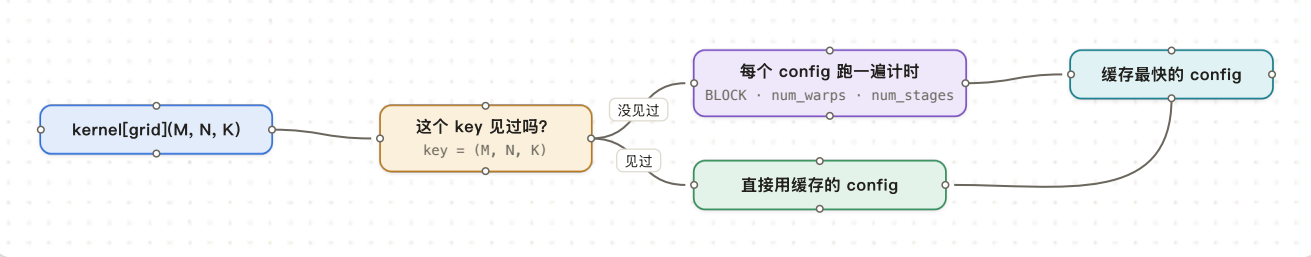
一个 program 怎么变成线程



类比 · CUDA 写给工人，Triton 写给工头

CUDA 里你给每个工人（线程）写说明书：“你是第几号，你负责第几个格子”。Triton 里你只给工头（program）写：“把这一箱（block）处理完”，工头自己决定怎么分给手下（num_warps × 32 个线程）、谁拿几个、怎么搬得最快。`mask` 就是最后一箱不满，空格子贴上“跳过”标签；`tl.constexpr` 是模具尺寸：换尺寸就要重新开模（重新编译）。

autotune：第一次调用时把每个 config 都试一遍



类比 · num_stages 是提前备料

厨师做第 k 道菜时，帮工已经在洗第 k+1、k+2 道的菜（num_stages = 3）：计算和搬运同时进行。备得越多越不会断档，但每一份备料都要占一块料理台（shared memory），台子放满了，能同时开工的厨师（occupancy）就少了。

源码与资料

文档：Triton 文档首页 · Tutorials 目录 · triton.language API

源码：`triton › 01-vector-add.py` · `triton › interpreter.py (TRITON_INTERPRET)` · `triton › Autotuner` · `triton › Gluon 01-intro` · `triton › Gluon 02-layouts` · `triton › 全部 Gluon tutorials` · `puzzles › Triton-Puzzles`

Part 0 · 基础 4

测试、计时与 profiling

每个 kernel 都走同一个循环：reference → 最简单的正确实现 → 测试 → benchmark → profile → 一次只改一个因素。

1. 正确性：和 PyTorch 对拍
2. 计时：别用 time.time() 包一次调用
3. Profiling：找到真正的瓶颈
4. 图解与类比
5. 源码与资料

正确性：和 PyTorch 对拍

```
torch.testing.assert_close(out, ref, rtol=..., atol=...)
```

- **reference 用更高精度算**：fp16 kernel 和 fp32（甚至 fp64）的 reference 比，而不是和另一个 fp16 实现比。
- **容差要和数值实际误差对应**：fp32 逐元素运算 `1e-5`；fp16 输出 `1e-2` 量级（fp16 只有约 3 位有效数字）；纯拷贝（转置、写 cache）要求**完全相等**。
- **边界 shape 一定要测**：1、block 大小 ±1、非 2 的幂、远大于 block、空张量、非连续（转置 / 切片后的 view）。大多数 bug 出在最后一个不完整的 block。
- CUDA 的越界读写不一定当场崩溃：`compute-sanitizer --tool memcheck python -m pytest tests/test_c3_transpose.py` 能抓出来。

计时：别用 `time.time()` 包一次调用

GPU 调用是**异步**的：Python 只是把 kernel 放进队列就返回了。正确的做法：

```
import triton.testing
ms = triton.testing.do_bench(lambda: my_kernel(x), warmup=25, rep=100, return_mode='
```

`do_bench` 会先 warmup（编译、autotune、缓存预热），每次测量前清空 L2，用 CUDA event 计时；默认返回平均值，传 `return_mode="median"` 取中位数，不容易被偶发的慢样本带偏。自己写的话要点相同：

```
start, end = torch.cuda.Event(enable_timing=True), torch.cuda.Event(enable_timing=T
for _ in range(10): fn()                 # warmup
start.record()
for _ in range(100): fn()
end.record(); torch.cuda.synchronize()
ms = start.elapsed_time(end) / 100
```

报告结果时要附上 GPU 型号、dtype、shape，并换算成 **GB/s 或 TFLOP/s**。小 shape 下几微秒的时间主要是 launch 开销，此时比的是 Python / launch 路径，不是 kernel 本身。

Profiling：找到真正的瓶颈

工具	回答的问题	命令
Nsight Compute（`ncu`）	单个 kernel 为什么慢：带宽用了多少、occupancy、warp 在等什么、bank conflict、寄存器数	`ncu --set full -k regex:softmax -o prof python bench.py t2`
Nsight Systems（`nsys`）	时间线：kernel 之间有没有空隙、CPU 是不是瓶颈、launch 了多少个 kernel	`nsys profile -o timeline python bench.py t2`
PyTorch Profiler	框架层哪个算子花时间	`torch.profiler.profile(...)`
compute-sanitizer	越界、竞争条件、未初始化 shared memory	`compute-sanitizer --tool racecheck ...`

`ncu` 报告里优先看三个数字：

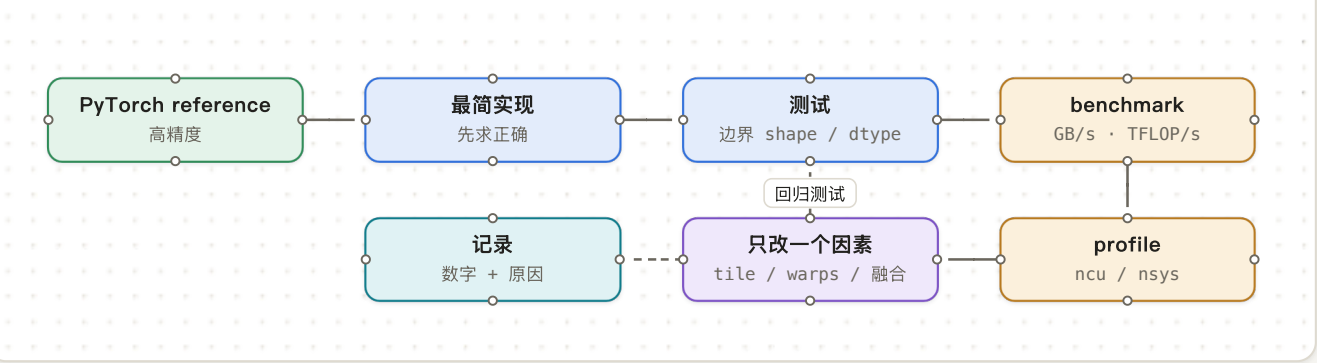
1. **Memory Throughput（% of peak）**：memory-bound 的 kernel 应该在 80% 以上。
2. **Compute (SM) Throughput**：compute-bound 的 kernel 看这里，以及 Tensor Core 利用率。

3. **Warp State Statistics**：warp 大部分时间停在哪。`Long Scoreboard` 表示在等 global memory，`Barrier` 表示在等 `__syncthreads`，`MIO Throttle` 表示 shared memory 指令排队。

> **一次只改一个因素**
>
> 每次优化只改一件事（block 大小、num_warps、加 padding、加 tiling...），测一次，记下数字和原因。做完一题，用三五行写下“瓶颈是什么、改了什么、提升多少、离上限还差多少”。这份记录就是原计划里每题的 README.md。

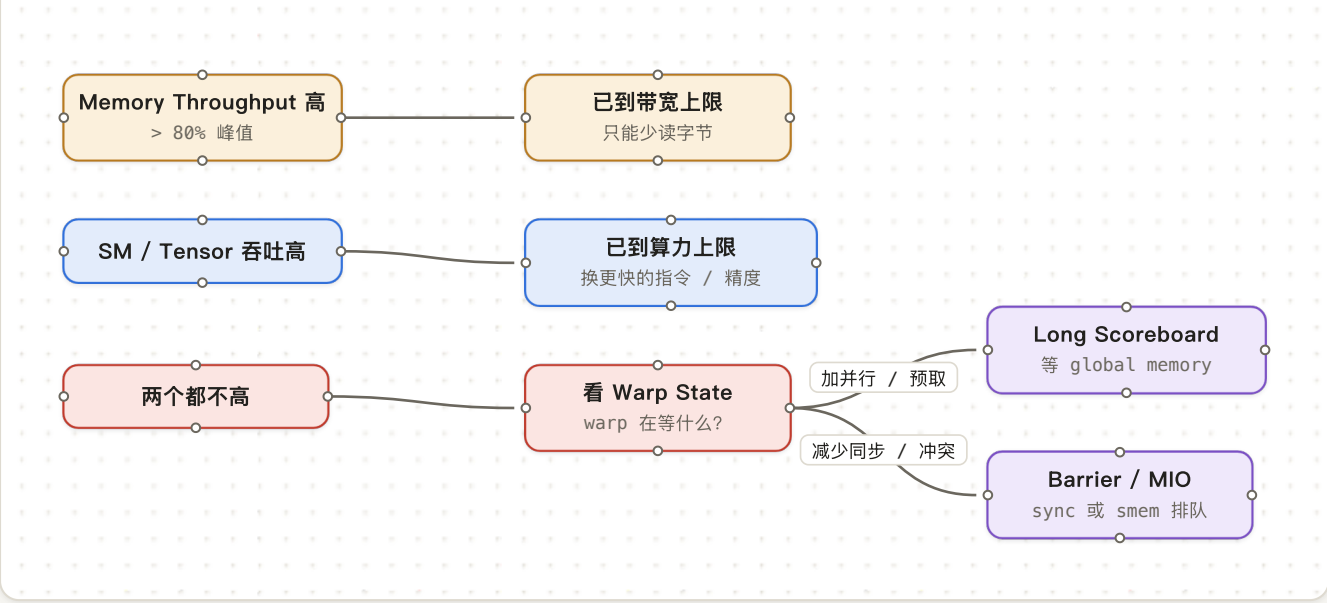
图解与类比

每个 kernel 都走的开发循环



> **类比 · 调音台一次只推一个推子**
>
> 同时动三个推子，声音变好了也说不清是哪个起了作用。kernel 调优也一样：每次只改一件事（block 大小、num_warps、padding……），测一次，记下来。

读 ncu 报告：三个数字定方向



类比 · ncu 是体检报告

不要一上来就“吃补药”（盲目调参数）。先看体检报告：带宽已经满了（血压到顶），再快的计算也没用；两项都不高，说明问题在“等待”上，就去看 warp 在等什么：是在等内存（Long Scoreboard），还是在等同伴（Barrier）。

源码与资料

文档：Nsight Compute · Nsight Systems · Compute Sanitizer · torch.testing.assert_close

源码：`triton › do_bench` · `pytorch › assert_close` · `flashinfer › bench_gpu_time`（FlashInfer 的 CUPTI 计时）

Part 0 · 基础 5

全景地图：模型 × 硬件 × 当前最优 kernel

在写任何一道题之前先建立地图：一个模型的时间花在哪类 kernel 上，这类 kernel 在每种硬件上当前用什么实现最好，以及为什么。

1. 先把 kernel 分类
2. 一个 LLM 推理 step 的时间去了哪
3. 按硬件：先看硬件给了什么
4. 按类别：当前的常见最优选择
5. 按模型领域
6. 选型的几条经验法则
7. 图解与类比
8. 源码与资料

9. 来源

先把 kernel 分类

几乎所有推理 / 训练的时间都落在下面几类 kernel 上。后面所有“谁最优”的判断，都是在问：**这个模型、这个阶段，时间主要花在哪一类上？这一类在这块硬件上是 compute-bound 还是 memory-bound?**

类别	例子	通常的瓶颈	本套题
稠密 GEMM	QKV / O / MLP 投影、LM head	prefill 和大 batch 时 compute-bound；小 batch decode 时读权重，memory-bound	C4、T4
低精度 GEMM	FP8（per-tensor / block-scaled）、NVFP4 / MXFP4、W4A16	同上，但每次 MMA 的吞吐翻倍、权重字节减半	M3
grouped GEMM（MoE）	每个 expert 各算一小块	每个 expert 的 M 很小、不均匀；路由和排序本身也有开销	M1、M2
attention：prefill	FlashAttention 类	compute-bound（S^2 的 FLOPs）	T5
attention：decode	paged decode、split-KV、MLA decode	memory-bound（读 KV cache）	L1–L3、F1、F2
线性 attention / SSM	Mamba、GDN（Qwen3-Next）、KDA（Kimi）	chunked scan；decode 是状态更新	附录
融合的小算子	RMSNorm + 残差、RoPE、SwiGLU、量化、adaLN modulate	memory-bound，次数多；不融合就被 launch 开销和多次读写拖慢	T1–T3、F3
采样	top-k / top-p / min-p、投机解码验证	词表 15 万：排序是瓶颈	F4
通信	all-reduce、MoE 的 all-to-all（DeepEP）	互联带宽；要和计算重叠	附录

一个 LLM 推理 step 的时间去了哪

- **prefill**（一次处理整段 prompt）：GEMM 占大头，attention 随 S^2 增长，长 prompt 时 attention 可以超过一半。两者都是 compute-bound，此时比拼 Tensor Core 的利用

率。

- **decode**（每步每个序列 1 个 token）：
  - 小 batch：GEMM 退化成矩阵乘向量，时间 ≈ 读一遍全部权重。量化权重（W4A16、FP8、NVFP4）直接按比例省时间。
  - 大 batch、长上下文：KV cache 的读取量超过权重，**decode attention** 变成主要开销。paged attention、GQA / MLA、KV 量化、split-KV 都在解决这件事。
  - batch = 1 的极端情况（比如 VLA、端侧）：每个 kernel 只跑几微秒，**launch 开销和 CPU 调度**反而成为瓶颈，所以要靠 CUDA graph 和更激进的融合。
- **MoE 模型**：GEMM 被拆成几十上百个 expert 的小 GEMM。专家并行（EP）时还要加上 all-to-all 通信。

按硬件：先看硬件给了什么

硬件	关键特性	这一代的代表性 kernel
A100（sm_80）	`mma.sync` Tensor Core、`cp.async`；没有 FP8	FlashAttention-2、FlashInfer（FA2 后端）、cuBLAS / CUTLASS 2.x、Triton
RTX 4090 / L40S（sm_89）	加入 FP8 `mma.sync`；没有 wgmma / TMA	FA2 类 kernel；FP8 GEMM
H100 / H200（sm_90a）	**wgmma**（warpgroup 级异步 MMA）、**TMA**（张量内存加速器）、线程块集群、warp specialization	**FlashAttention-3**、FlashInfer 的 FA3 后端、**DeepGEMM**、CUTLASS 3（CuTe）、FlashMLA
B200 / GB200 / B300（sm_100 / sm_103）	**tcgen05** MMA + **TMEM**（张量内存）、2-CTA MMA、硬件 block-scaled FP8 / FP6 / FP4（MXFP8、NVFP4）	**FlashAttention-4**（CuTe DSL）、TRT-LLM 生成的 kernel（FlashInfer 的 `trtllm-gen` 后端）、cuDNN attention、DeepGEMM（sm_100）、CUTLASS 4 / CuTe DSL
RTX 5090 / RTX PRO 6000（sm_120，消费级 Blackwell）	**没有** tcgen05 / TMEM / wgmma；`mma.sync` 扩展支持 FP8 / FP6 / FP4（含 block scale）	FA2 风格的 kernel、Triton、**SageAttention3**（FP4 attention）、FlashInfer 的 sm_120 专用 kernel（如 `nvfp4_attention_sm120`，在 FlashInfer 里属于 experimental 后端）；DeepGEMM 不支持
Jetson Thor（sm_110）/ **Orin**（sm_87）	端侧，功耗受限；统一内存	TensorRT；FlashInfer 已有 sm_110 的 GQA decode kernel
AMD MI300X / MI325X / MI355X	MFMA 矩阵指令；MI355X 加入 FP4 / FP6	**AITER**（Triton + Composable Kernel + 手写汇编：attention、fused MoE、MLA、

硬件	关键特性	这一代的代表性 kernel
（gfx942 / gfx950）		GEMM）、CK、HipKittens；vLLM / SGLang 通过 `VLLM_ROCM_USE_AITER=1` 等开关启用
Apple M 系列	统一内存（带宽约 100–800 GB/s）；simdgroup（32 线程）、`simdgroup_matrix` 8×8 MMA；M5 起 GPU 内有 neural accelerator	**MLX**：steel GEMM / attention、decode 专用的 `sdpa_vector`、分组量化的 `qmv / qmm`、MoE 用的 `gather_mm / gather_qmm`，以及面向 M5 neural accelerator 的 `*_nax` kernel；自定义 kernel 用 `mx.fast.metal_kernel`（X 系列）
TPU / Trainium	脉动阵列	Pallas（JAX）、NKI：本套题不覆盖

两个容易记错的点：

- RTX 5090 叫 Blackwell，但它的指令集（sm_120）和 B200（sm_100）**不一样**：没有 tcgen05 和 TMEM，所以 FlashAttention-4、DeepGEMM 这类为 sm_100 写的 kernel 不能直接跑。在 5090 上写 kernel，更接近“带 FP4 的 Ada”。FlashInfer 的贡献指南也把 SM12x 归为 experimental 后端。
- 越新的硬件，最优 kernel 越依赖专门的指令（wgmma、TMA、tcgen05）和 warp specialization。这也是 CUTLASS / CuTe DSL、Gluon 这类“能显式控制硬件”的工具变得重要的原因：Triton 的自动化在 Hopper / Blackwell 上更难追平手写 kernel。

按类别：当前的常见最优选择

这张表描述的是 2026 年下半年 vLLM / SGLang 这类主流 serving 框架的常见默认或推荐选择，变化很快，用之前请查对应框架的文档。

类别	H100	B200	RTX 5090	AMD MI3xx	Apple
prefill attention	FA3；FlashInfer（FA3 后端）	FA4；trtllm-gen / cuDNN（经 FlashInfer）	FA2 类；FlashInfer FA2 后端；SageAttention	AITER / CK	MLX steel attention
decode attention（GQA，分页）	FlashInfer decode / FA3	trtllm-gen（经 FlashInfer）；XQA	FlashInfer FA2 后端	AITER paged attention	MLX `sdpa_vector`
MLA（DeepSeek-V3 类）	FlashMLA；FlashInfer MLA	FlashInfer（CUTLASS / trtllm-gen MLA）	通用实现	AITER MLA	—

类别	H100	B200	RTX 5090	AMD MI3xx	Apple
稠密 GEMM	cuBLAS / CUTLASS；DeepGEMM（FP8 block）	cuBLAS / CUTLASS（NVFP4 / MXFP8）；DeepGEMM	cuBLAS；Triton	hipBLASLt；AITER	MLX steel GEMM
MoE grouped GEMM	Triton fused_moe；DeepGEMM（FP8 grouped）	FlashInfer 的 TRT-LLM / CUTLASS / CuTe-DSL MoE	Triton fused_moe	AITER fused MoE	MLX `gather_mm` / `gather_qmm`
小算子融合	FlashInfer norm / RoPE / activation；Triton	同左	同左	AITER；Triton	MLX `mx.fast.*`
采样	FlashInfer sampling	同左	同左	—	MLX

按模型领域

Dense LLM（GQA）

- prefill：attention 和 GEMM 都是 compute-bound，选这块硬件上最好的 FlashAttention 代际（A100 用 FA2，H100 用 FA3，B200 用 FA4 或 trtllm-gen）。
- decode：paged decode attention（L 系列、F1），长上下文配 split-KV（L3），GQA 打包让每个 KV 头只读一次（L2）。
- 周边：融合 RMSNorm + 残差（F3）、融合 RoPE + 写 KV cache（FlashInfer 有 `fused_qk_rmsnorm_rope`）、SwiGLU（F3）、不排序的采样（F4）。
- 量化：Hopper 上常用 FP8（per-tensor 或 block-scaled），Blackwell 上 NVFP4 / MXFP4；小 batch decode 时权重量化收益最大。

MLA 模型（DeepSeek-V3/R1、Kimi K2 等）

MLA 把 K、V 压缩到一个低秩 latent 里，KV cache 小很多。decode 时，可以把 latent 的上投影吸收进 Q，attention 直接在 latent 上做（head_dim 512 + 64 的 RoPE 部分）。这使

decode attention 的算术强度比 GQA 高，需要专门的 kernel：FlashMLA（Hopper）、FlashInfer 的 `BatchMLAPagedAttentionWrapper`。

MoE 模型（DeepSeek-V3、Qwen3-MoE、gpt-oss、Kimi）

- **路由与排列**：top-k 选专家、把 token 按专家分组并补齐到 tile 大小（M1）。
- **grouped GEMM**：一次 launch 算完所有专家（M2）。prefill 用 contiguous 布局（各专家的 token 连续排列）；decode 要配合 CUDA graph，用 masked 布局（预分配好每个专家的最大行数，按实际数量屏蔽）。DeepGEMM 同时提供这两种布局。
- **低精度**：DeepSeek-V3 的 FP8 训练和推理使用 1×128（激活）和 128×128（权重）的 block scale（M3），NVFP4 MoE 则是 Blackwell 上的主流。
- **通信**：专家并行时，dispatch / combine 的 all-to-all 用 DeepEP（基于 NVSHMEM）这类库，并尽量与计算重叠。DeepGEMM 2026 年 4 月加入了把 dispatch、两层 FP8×FP4 GEMM、combine 融合在一起的 Mega MoE kernel。

线性 attention / 混合架构（Qwen3-Next、Kimi Linear、Mamba 类）

把 attention 换成状态大小固定的递推（GDN、KDA、Mamba2），长上下文时 decode 的开销不再随长度增长。kernel 主要来自 flash-linear-attention（Triton）项目，FlashInfer 0.7 也已经加入 GDN / KDA 的 prefill 与 decode kernel。训练和 prefill 用分块并行的 chunked scan，decode 是逐 token 的状态更新，同样需要融合。

扩散模型（DiT 图像 / 视频）

- 没有 KV cache，也没有 decode：每个去噪步都是对全部 token 的一次 **compute-bound** 前向，视频模型的序列长度动辄数万到数十万，**attention 往往占一半以上的时间**。
- 当前最有效的 kernel 方向：
  - 最好的 FlashAttention 代际（H100 用 FA3，B200 用 FA4）；
  - **量化 attention**：SageAttention 系列（INT8 / FP8；SageAttention3 在 RTX 5090 上用 FP4，作者报告 attention kernel 约是 FlashAttention2 的 5 倍，HunyuanVideo 端到端约 3 倍）；
  - **稀疏 attention**：滑动 tile、radial、VSA 一类利用视频时空局部性的方法；
  - **融合 adaLN**：DiT 每层都有 `norm → scale/shift → gate → residual`，FlashInfer 已有 `fused_dit_*` 系列算子；

    - 低比特 GEMM：SVDQuant / Nunchaku 的 W4A4；
    - 多卡：xDiT 的 Ulysses / ring 序列并行；VAE 解码中的 3D 卷积依赖 cuDNN。
- 注意端到端收益：去噪步数的蒸馏、特征缓存（TeaCache 等）往往比任何单个 kernel 的提升都大。也有实际工作流中 attention 只占很小一部分的例子：kernel 快 4 倍，整体只快 2%。**先 profile，再决定优化哪个 kernel。**

Omni / 语音 / 多模态（Qwen3-Omni 类）

- 多个阶段串联：视觉 / 音频编码器（ViT、卷积 + Transformer）→ LLM（thinker）→ 语音生成（talker）→ codec 解码。
- 编码器是 compute-bound 的变长 attention（FlashAttention varlen），文本主干就是普通的 LLM 路径，外加多维 RoPE（M-RoPE）。
- talker 和 codec 每步只生成很少的数据，**小 kernel、batch 小，以 launch 开销为主**：CUDA graph、流式因果卷积、跨阶段的 chunk 流水才是关键。

VLA / 机器人

- batch = 1、对延迟有硬性要求（几十 Hz 的控制频率）。$\pi 0$ 这类模型由 VLM 主干和一个用 flow matching 多步去噪生成动作块的 action expert 组成。
- 瓶颈通常不是某一个大 kernel，而是**上百个小 kernel 的 launch 开销和 CPU 调度**。有效手段几乎总是同一个组合：用 Triton 融合小算子、预分配静态 buffer，把整条前向（包括去噪循环）捕获进一个 CUDA graph。有研究报告，仅去掉 CPU 开销就能带来约 2 倍的加速。
- 端侧部署（Jetson Thor / Orin）再加上 TensorRT 和低比特量化。

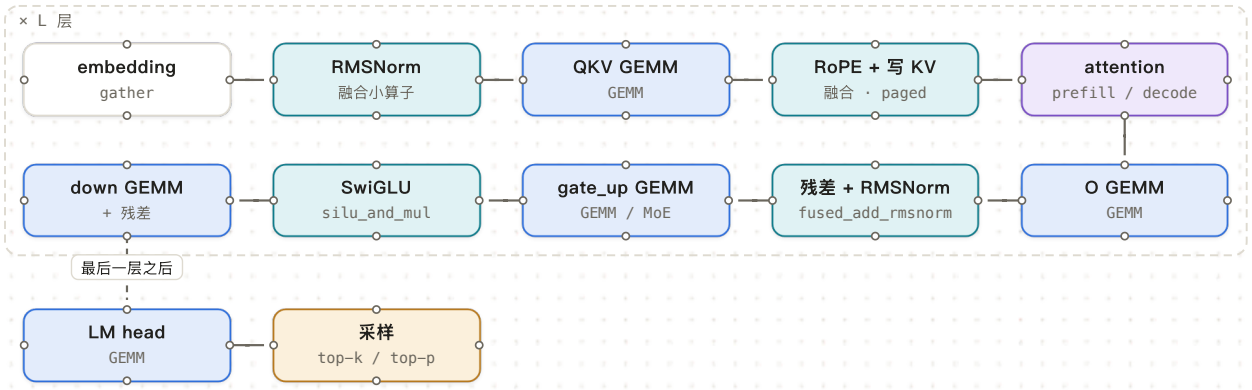
选型的几条经验法则

1. **先判断 compute-bound 还是 memory-bound**（Part 0 的 roofline）：前者比拼 Tensor Core 利用率和新指令；后者比拼少读字节（融合、量化、GQA / MLA、KV 压缩）。
2. **batch 很小时先看 launch 开销**：kernel 本身只要几微秒时，CUDA graph 和融合比再快 10% 的 kernel 更重要。
3. **硬件代际决定可选项**：B200 上最好的 kernel 在 5090 上可能根本跑不了，反过来也一样。看 sm 版本，而不是看产品名。

4. **库的选择大于手写**：FlashInfer、cuBLAS / CUTLASS、DeepGEMM、AITER、MLX 已经覆盖了绝大多数热点。值得手写的是**新模型带来的新算子**（新的 attention 变体、新的量化格式、新的融合模式），这也是向 FlashInfer 这类库贡献的主要方向。

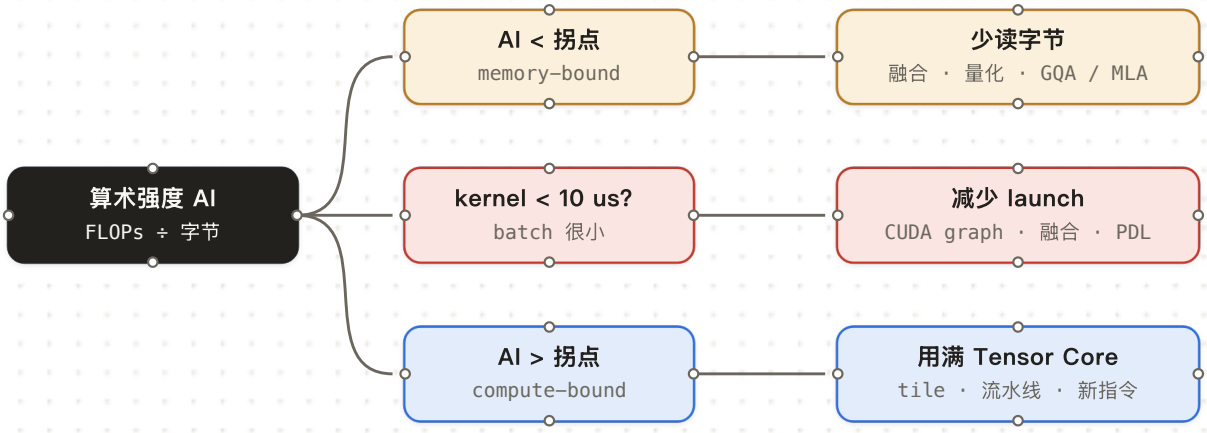
图解与类比

一个 decoder layer 里有哪些 kernel



颜色对应 kernel 类别：蓝色 GEMM、紫色 attention、青色融合小算子、橙色采样。prefill 时蓝紫两类占大头；decode 时小算子的固定开销和 KV 读取变得显著。

这个 kernel 该往哪个方向优化?



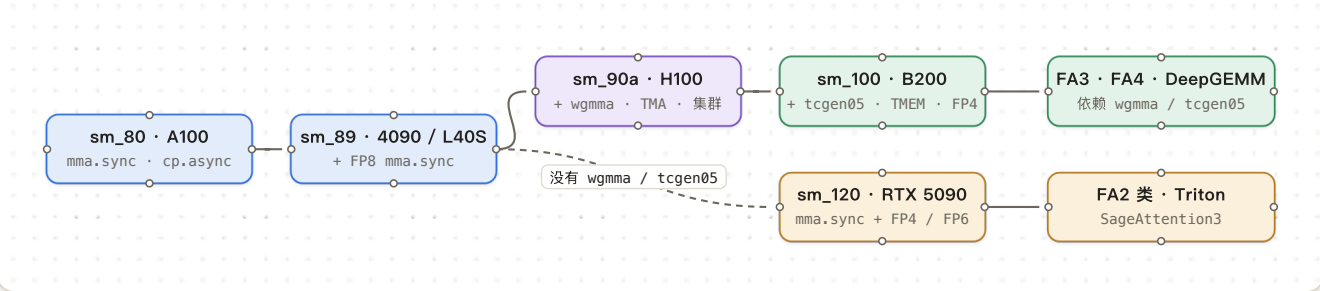
类比 · 生产线上最慢的工位决定产量

把一次推理看成一条生产线：embedding → 十几道工序 × L 层 → 采样。哪个工位最慢，就去优化哪个；把一个只占 2% 时间的工位提速 4 倍，整条线只快 1.5%（扩散模型里 attention 只占小部分时就是这样）。

类比 · 换硬件就是换工厂

B200 工厂里有专门的机械臂（tcgen05 + TMEM）和自动叉车（TMA），最好的工艺（FA4、DeepGEMM）是围绕它们设计的；RTX 5090 的工厂叫同一个品牌，却没有这台机械臂，所以那套工艺搬过去根本开不了工。选 kernel 时看 sm 版本，不看产品名。

NVIDIA 架构代际：哪些能力是新加的



类比 · 同名不同厂

“Blackwell”这个名字同时用在 B200 和 RTX 5090 上，就像同一个汽车品牌既有赛车也有家用车：品牌一样，发动机不同。B200 有 tcgen05 这台“涡轮”，为它调校的 kernel 拿到 5090 上点不着火；5090 上能用的是更接近上一代（Ada）的 `mma.sync`，只是多了 FP4。

源码与资料

库（固定 commit）： `flashinfer › flashinfer-ai/flashinfer` · `fa › flash-attention` · `deepgemm › DeepGEMM` · `deepep › DeepEP` · `flashmla › FlashMLA` · `cutlass › CUTLASS` · `sage › SageAttention` · `fla › flash-linear-attention` · `aiter › AITER` · `mlx › MLX` · `vllm › vLLM` · `sglang › SGLang`

各代 attention 的代表实现： `fa › FA2 前向` · `fa › FA3 Hopper 主循环 (TMA + wgmma + warp specialization)` · `fa › FA4 Blackwell (CuTe DSL)` · `flashinfer › FlashInfer prefill` · `flashinfer › FlashInfer decode` · `mlx › MLX sdpa_vector` · `sage › SageAttention3 (FP4, Blackwell)`

GEMM / MoE / 线性 attention： `deepgemm › DeepGEMM SM90 FP8` · `deepgemm › DeepGEMM SM100 Mega MoE` · `vllm › vLLM Triton fused_moe` · `flashinfer › FlashInfer fused_moe` · `fla › FLA GDN chunk` · `mlx › MLX 量化 matmul`

serving 框架怎么接 FlashInfer： `vllm › vLLM FlashInfer 后端` · `sglang › SGLang FlashInfer 后端`

来源

- FlashAttention-4（Blackwell，CuTe DSL）：论文 arXiv 2603.05451 · MLSys 2026
- DeepGEMM（SM90 / SM100，FP8 / FP4，grouped，Mega MoE）：文档
- SageAttention3（FP4 attention，RTX 5090 实测）：论文 arXiv 2505.11594 · GitHub · 一个端到端收益很小的反例
- AMD AITER 与 vLLM / SGLang：ROCm blog: vLLM on MI355X · MLPerf v5.1 技术解读
- VLA 实时推理：Running VLAs at Real-time Speed（arXiv 2510.26742） · FluxVLA 推理加速文档
- FlashInfer 的 API 和后端名称（trtllm-gen、cudnn、cutlass、cute_dsl、sm_110 / sm_120 kernel、GDN / KDA、fused DiT norm）：直接来自 `flashinfer-python 0.7.2` 的 Python API 和 2026-11-02 的源码。

核 心

Track C · CUDA C++

硬件映射：线程索引、合并访存、shared memory、warp 原语、数据复用。

核心 · Track C · CUDA C++ · 第 1 题

C1 Vector add + 融合 bias-GELU：索引、launch 配置、fusion

第一个 CUDA kernel：把线程编号算对、把 launch 配置配对，再体会“融合”为什么能直接省一半时间。

35 min | 难度 易 | 你写 `mine/c1_elementwise.cu` | 验收 `./check.sh c1` | 参考解 `solutions/c1_elementwise.cu`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

逐元素运算（加法、激活函数、bias、残差、类型转换）在 Transformer 里无处不在。它们的算术强度极低（vector add 每 12 字节才 1 次加法），**速度完全由搬了多少字节决定**。

PyTorch 里写 `F.gelu(x + bias)` 会启动两个 kernel：第一个读 x、写临时张量 t；第二个读 t、写 y，一共 4 趟显存读写。把它们写进一个 kernel（**fusion**），只要读 x、写 y 两趟，理论上快一倍，也少一次 launch。`torch.compile`、TensorRT、vLLM 的自定义算子做的大部分工作就是这类融合。

源码与资料

`cudasamples › vectorAdd.cu` · `vllm › vLLM activation_kernels.cu` · `flashinfer › FlashInfer activation.cuh` · `flashinfer › vec_t（16 字节向量化读写）` · An Even Easier Introduction to CUDA · Programming Guide: Kernels / Thread Hierarchy

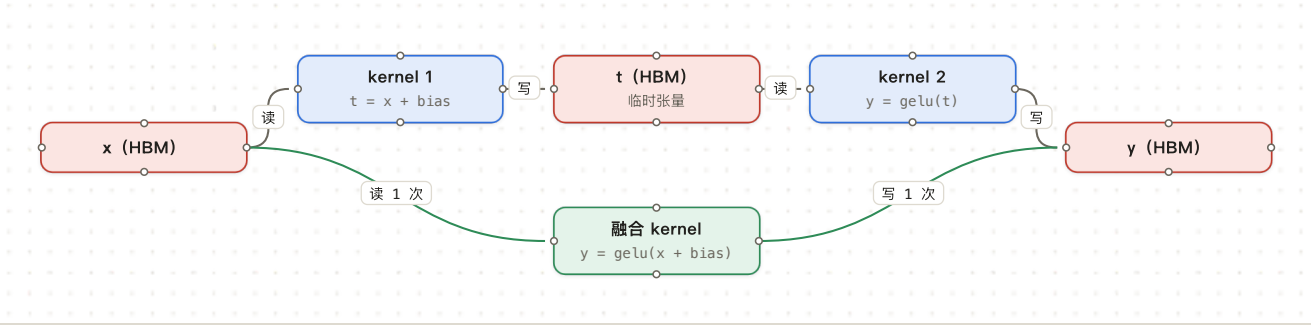
要用到的知识

- 全局线程编号：`int64_t i = (int64_t)blockIdx.x * blockDim.x + threadIdx.x;`
- launch 配置：`blocks = ceil(n / threads)`，即 `(n + threads - 1) / threads`。
- **边界检查**：最后一个 block 通常不满，`if (i < n)` 必不可少。

- **grid-stride loop**：`for (i = 起点; i < total; i += blockDim.x * gridDim.x)`，grid 大小可以与数据量无关。
- `tanhf`：CUDA 数学库的单精度 tanh。
- 行主序 `[rows, cols]` 中，扁平下标 i 对应的列号是 `i % cols`。

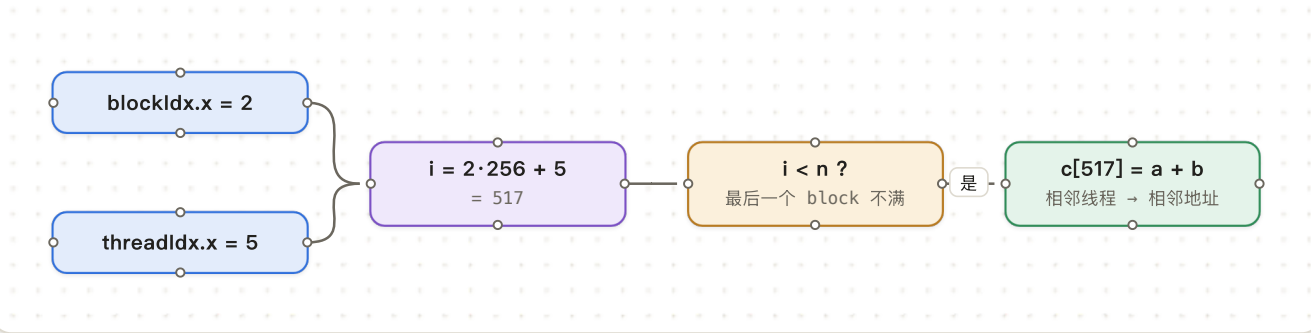
图解与类比

不融合 vs 融合：数一数 HBM 读写



上面一条路径 4 趟 HBM 读写、2 次 launch；下面 2 趟、1 次。memory-bound 的 kernel 时间正比于搬运的字节数，所以融合后约快 2 倍（RTX 5090 实测 706 → 379 us）。

一个线程怎么知道自己负责哪个元素



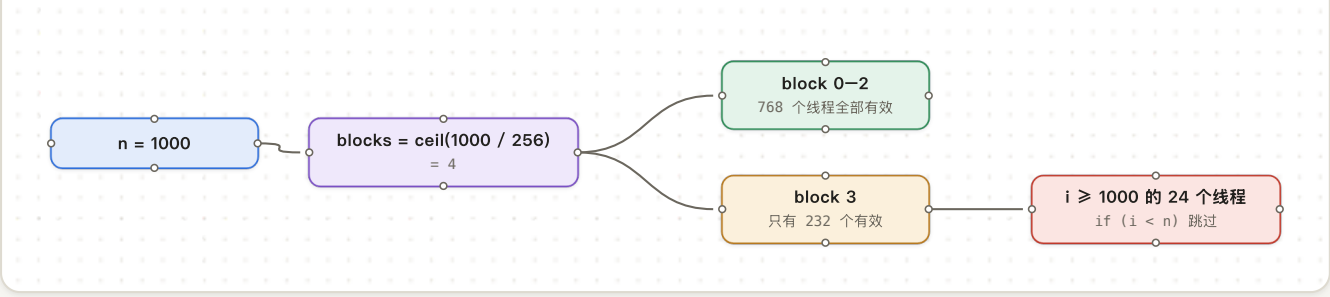
类比 · 演唱会的座位号

blockIdx 是区号，threadIdx 是区内座位号，全局编号 = 区号 × 每区座位数 + 座位号。最后一个区通常坐不满，多出来的座位不能发票（`if (i < n)`）。 相邻座位的观众去相邻的检票口（合并访存），入场才最快。

类比 · 融合 = 洗切炒一气呵成

不融合就像洗完菜放回冰箱，切菜时再拿出来，切完又放回去，炒的时候再拿：每次进出冰箱（HBM）都要时间。融合是在料理台上一次做完，只在开始拿一次、最后装盘一次。

launch 配置：n = 1000、每 block 256 线程



类比 · float4 是一次搬一箱

每个线程一次读一个 float，就像一瓶一瓶地搬水；用 `float4`（一次 16 字节）就是一次搬一箱。总重量（字节数）不变，但搬运的次数（指令数）少了 4 倍。前提是箱子摆得整齐：地址要 16 字节对齐，长度是 4 的倍数。

为什么只有 blockIdx、threadIdx，没有 warpIdx?

CUDA 只给你两样坐标：**第几个线程块（blockIdx）** 和 **块里的第几个线程（threadIdx）**。warp 不是另一套编号，它只是硬件把线程**按编号顺序每 32 个打成一捆**一起执行。既然是按顺序捆的，“第几捆”除一下就能算出来：

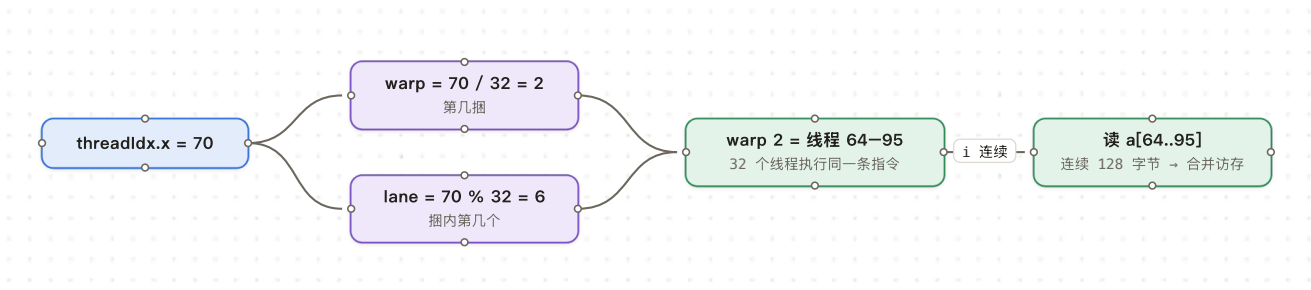
```
int warp = threadIdx.x / 32;   // 我在第几捆（warpSize 是内置常量，NVIDIA 上为 32）
int lane = threadIdx.x % 32;   // 我是这一捆里的第几个（常写成 threadIdx.x & 31）

// 2D / 3D block：先按 x 最快变化拉成一维，再除以 32
int tid = threadIdx.x + threadIdx.y * blockDim.x + threadIdx.z * blockDim.x * blockI
```

为什么不内置一个 warpIdx：

1. **你只负责分活，硬件负责怎么执行。** 编程模型里分配工作的单位是线程：“这个线程处理第几个元素”。至于怎么 32 个一捆地上机器，是硬件的事。这样代码不和某一种硬件绑死（AMD 的 wavefront 是 64 个一捆）。
2. **规则固定，可以直接算。** 线程 0–31 永远是第 0 捆，32–63 是第 1 捆……能从 threadIdx 推出来的东西，没必要再占一个内置变量。
3. **底层的 `%warpid` 不能用。** PTX 里确实有 `%warpid`，但它表示这捆线程此刻占用了 SM 上的哪个物理槽位，调度时可能改变，不是“块里的第几捆”。`%laneid` 是可靠的，不过它就等于 `tid % 32`。对比一下：Hopper 新增了 `clusterIdx`，因为 thread block cluster 是程序员要主动利用的协作层级（簇内可以访问彼此的 shared memory）；warp 一直是隐式的，所以从来没有自己的坐标。

线程 → 第几捆（warp）→ 捆内第几个（lane）



C1 的代码里没有出现 warp，但性能由它决定：同一个 warp 的 32 个线程同时发出访存，i 连续 → 地址连续 → 合并成少数几次 128 字节事务。

C1 虽然没写 warp，下面三件事都由 warp 决定：

- **合并访存**：`i = blockIdx.x * blockDim.x + threadIdx.x` 让同一 warp 的 32 个线程拿到连续的 i，也就读连续的地址。如果把 threadIdx.x 用在慢变化的维度上，同一 warp 的访问就会分散到很多段内存。
- **blockDim 取 32 的倍数**：一个 block 100 个线程时，最后一个 warp 只有 4 个线程干活，另外 28 个位置空转。
- **分支发散**：同一 warp 里有的线程走 if、有的走 else，这个 warp 就要把两条路依次各走一遍；不同 warp 之间走不同分支则没有代价。所以 C1 的 `if (i < n)` 只在最后一个 warp 里发散，几乎没有影响。

需要显式算 warp / lane 的地方在后面：C2 的 `__shfl_down_sync` 在 warp 内部交换寄存器，C4 把 block tile 再分成 warp tile，Tensor Core 的 `mma` 是一个 warp 32 个线程合作执行的指令。

> **类比 · 学号和排号**
>
> 学校只发年级号（blockIdx）和学号（threadIdx）。上课时老师按学号顺序每 32 人坐一排，同一排听同一条口令、同时动作，这一排就是一个 warp。没人需要单独发“排号”：学号 ÷ 32 是排号，余数是座位号。 排号虽然没发，但它真实存在：同一排的人同时去书架拿**相邻**的书就很快（合并访存）；一排里一半人做算术、一半人写作文，老师只能先管一半、再管另一半，就慢了（分支发散）。

文件开头的三个 #include 是什么？

题干 `.cu` 文件的前三行：

```
#include <torch/extension.h>          // 写 PyTorch C++ 扩展的“总入口”
#include <ATen/cuda/CUDAContext.h>    // 拿 PyTorch 当前的 CUDA stream、设备属性
```

```
#include <c10/cuda/CUDAException.h>   // 检查 CUDA 错误，并变成 Python 异常
```

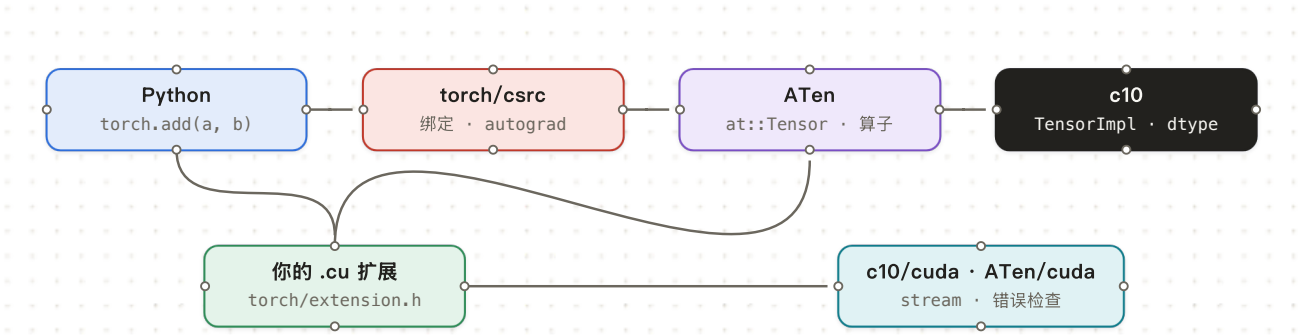
先讲两个名字，后面的头文件都建立在它们上面。

c10：PyTorch 最底层的核心库。 名字一般解释为 Caffe2 和 ATen 合并时抽出来的公共核心（“10”谐音 ten）。它只放所有地方都要用、又不涉及具体算子的基础设施：

- 张量的“身体”：`TensorImpl`（形状、stride、dtype、设备）、`Storage`（真正的那块内存）、`Allocator`。
- 基本类型：`ScalarType`（`kFloat`、`kHalf`……）、`Device` / `DeviceType`、`DispatchKey`（决定一个算子调用 CPU 版还是 CUDA 版）。
- 工具：`TORCH_CHECK` 和 `c10::Error`（报错）、`intrusive_ptr`（引用计数）、`optional`、`ArrayRef`。
- `c10/cuda/`：CUDA 相关的最小功能，比如 stream（`CUDAStream`）、设备切换（`CUDAGuard`）、错误检查（`CUDAException.h`）。

ATen：PyTorch 的 C++ 张量库（A TENsor library）。`at::Tensor` 类型和 2000 多个算子（`add`、`matmul`、`softmax`……）都在这里，每个算子按设备有 CPU、CUDA 等不同实现，由 dispatcher 按 `DispatchKey` 分发。你在 Python 里调用 `torch.add`，最终执行的就是 ATen 里的 C++ / CUDA kernel。所有算子的清单在 `native_functions.yaml`，CUDA kernel 在 `aten/src/ATen/native/cuda/`（比如 T2 引用的 `SoftMax.cu`）。

PyTorch 的分层：从 Python 到 c10



上排是 PyTorch 自己的调用链。下排是你的扩展：load_inline 生成的绑定把 Python 张量交给它，它直接使用 ATen 的 at::Tensor（torch::Tensor 只是别名），再用 c10/cuda、ATen/cuda 的工具拿 stream、检查错误，然后发射自己的 kernel。

下面分别看三个头文件。

1. **`torch/extension.h`：写 C++ 扩展的总入口。** 一行 include 带进三样东西：

- **PyTorch 的 C++ API**（`torch/all.h`）：`torch::Tensor`（就是 `at::Tensor`）、`torch::empty_like`、`a.numel()`、`a.data_ptr<float>()`、`a.is_cuda()` 等。

- **pybind11 和张量类型转换**：Python 的 `torch.Tensor` 传进来自动变成 C++ 的 `torch::Tensor`，返回时再转回去，不发生数据拷贝。我们用 `load_inline(functions=[...])` 编译时，它会自动生成 `PYBIND11_MODULE` 绑定代码，所以 `.cu` 里不需要自己写。
- **`TORCH_CHECK(cond, msg)`**（来自 c10）：条件不满足就抛 `c10::Error`，到 Python 里是一个带消息的 `RuntimeError`。文件里的 `CHECK_F32_CUDA` 宏就是三个 `TORCH_CHECK`。`at::kFloat` 也是从 c10 的 `ScalarType` 来的。

2. **`ATen/cuda/CUDAContext.h`：拿到 PyTorch 当前的 CUDA 环境。** 这里用到两个函数：

- `at::cuda::getCurrentCUDAStream()`：返回 PyTorch 当前使用的 stream，可以直接当 `cudaStream_t` 用。kernel 要发到这个 stream 上（`<<<blocks, threads, 0, stream>>>`），这样才和前后的 PyTorch 算子正确排队；如果用户写了 `with torch.cuda.stream(s):`，你的 kernel 也会跟着进 `s`。不传 stream 就会进默认 stream，可能和 PyTorch 的其他操作乱序，或者多出不必要的同步。
- `at::cuda::getCurrentDeviceProperties()`：返回当前 GPU 的 `cudaDeviceProp*`。`bias_gelu` 用它的 `multiProcessorCount`（SM 数）来定 grid 上限：最多 `SM 数 × 32` 个 block。

这个头文件还提供 cuBLAS / cuSPARSE 句柄（`getCurrentCUDABlasHandle()` 等），写 GEMM 扩展时会用到。

3. **`c10/cuda/CUDAException.h`：把 CUDA 错误变成 Python 异常。** 这里用到的是 `C10_CUDA_KERNEL_LAUNCH_CHECK()`，它等于 `C10_CUDA_CHECK(cudaGetLastError())`。为什么需要它：`kernel<<<...>>>` 这种发射**没有返回值**，发射参数出错（比如 block 数为 0、每 block 超过 1024 个线程）时不会自动报错，只会记在“最后一个错误”里。发射后立刻查一次，出错就抛异常，Python 里就能看到 `CUDA error: invalid configuration argument`。题干里 `blocks = 0` 还没填时，测试就是这样干净地失败的。

注意它只能查到**发射时**的错误。kernel **运行中**的错误（比如越界访问导致的 `illegal memory access`）是异步的，往往要到下一次同步时才报出来，而且报错位置会对不上。调试时设置 `CUDA_LAUNCH_BLOCKING=1`，让每次发射都同步执行，就能在出错的那一行报出来。

> **类比 · 插座、电表和保险丝**
>
> 写扩展就像在 PyTorch 这栋楼里接一台自己的电器。`torch/extension.h` 是**插座**：插上就能通电，Python 和 C++ 之间的张量自动转换。`CUDAContext.h` 是**电表和楼层配电信息**：告诉你这层用的是哪条线路（当前 stream），线路能扛多大负载（SM 数）。接在同一条线路上，你的电器才会和楼里其他电器按顺序用电。`CUDAException.h` 是**保险丝**：接

> 线错了（发射参数错误）当场跳闸，并告诉你哪里错了，而不是让问题悄悄留到后面。c10 和 ATen 是这栋楼的地基和主体结构，你的电器就装在它们上面。

源码：`pytorch › extension.h` · `pytorch › CUDAContext.h` · `pytorch › C10_CUDA_KERNEL_LAUNCH_CHECK` · `pytorch › getCurrentCUDAStream` · `pytorch › TORCH_CHECK` · `pytorch › TensorImpl.h` · `pytorch › ScalarType.h` · `pytorch › native_functions.yaml` · `pytorch › load_inline` · PyTorch C++ 扩展教程 · PyTorch C++ API 文档

题目要求

在 `mine/c1_elementwise.cu` 里完成：

1. **`vector_add_kernel`**：`c[i] = a[i] + b[i]`，每个线程处理一个元素；以及 host 函数里的 `blocks` 计算。
2. **`gelu_tanh`**：`0.5·x·(1 + tanh(√(2/π)·(x + 0.044715·x³)))`，与 `F.gelu(approximate="tanh")` 一致。
3. **`bias_gelu_kernel`**：`y[r][c] = gelu_tanh(x[r][c] + bias[c])`，用 grid-stride loop；host 里 `blocks = min(ceil(total/threads), SM 数 × 32)`。

输入都是连续的 float32 CUDA 张量；空张量、n = 1、n 不是 256 倍数的情况都要正确。

题干

mine/c1_elementwise.cu 72 行

```
// C1 · Vector add + fused bias-GELU — YOUR IMPLEMENTATION.
// Fill in the two kernels and the two launch configurations marked TODO.
#include <torch/extension.h>
#include <ATen/cuda/CUDAContext.h>
#include <c10/cuda/CUDAException.h>

#include <algorithm>

#define CHECK_F32_CUDA(t)                                                   \
  TORCH_CHECK((t).is_cuda(), #t " must be a CUDA tensor");                   \
  TORCH_CHECK((t).is_contiguous(), #t " must be contiguous");               \
  TORCH_CHECK((t).scalar_type() == at::kFloat, #t " must be float32")

// c[i] = a[i] + b[i] for i < n. One thread per element.
__global__ void vector_add_kernel(const float* __restrict__ a,
                                  const float* __restrict__ b,
```

```
                                float* __restrict__ c,
                                int64_t n) {
  // TODO: compute this thread's global index (as int64_t), bounds-check, add.
}

torch::Tensor vector_add(torch::Tensor a, torch::Tensor b) {
  CHECK_F32_CUDA(a);
  CHECK_F32_CUDA(b);
  TORCH_CHECK(a.numel() == b.numel(), "a and b must have the same number of eleme
nts");
  auto c = torch::empty_like(a);
  const int64_t n = a.numel();
  if (n == 0) return c;

  const int threads = 256;
  const int64_t blocks = 0;  // TODO: enough blocks of `threads` threads to cover
n elements
  cudaStream_t stream = at::cuda::getCurrentCUDAStream();
  vector_add_kernel<<<blocks, threads, 0, stream>>>(
      a.data_ptr<float>(), b.data_ptr<float>(), c.data_ptr<float>(), n);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return c;
}

// GELU, tanh approximation: 0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 x^
3)))
__device__ __forceinline__ float gelu_tanh(float x) {
  // TODO
  return x;
}

// y[r][c] = gelu_tanh(x[r][c] + bias[c]); x, y: [rows, cols] row-major; total =
rows * cols.
// Use a grid-stride loop so any grid size covers all `total` elements.
__global__ void bias_gelu_kernel(const float* __restrict__ x,
                                 const float* __restrict__ bias,
                                 float* __restrict__ y,
                                 int64_t total,
                                 int64_t cols) {
  // TODO
}

torch::Tensor bias_gelu(torch::Tensor x, torch::Tensor bias) {
  CHECK_F32_CUDA(x);
  CHECK_F32_CUDA(bias);
  TORCH_CHECK(x.dim() == 2, "x must be 2-D [rows, cols]");
  TORCH_CHECK(bias.dim() == 1 && bias.size(0) == x.size(1), "bias must be [col
s]");
  auto y = torch::empty_like(x);
  const int64_t total = x.numel();
```

```
  if (total == 0) return y;

  const int threads = 256;
  const int sms = at::cuda::getCurrentDeviceProperties()->multiProcessorCount;
  const int64_t blocks = 0;  // TODO: ceil(total / threads), but at most sms * 32
  cudaStream_t stream = at::cuda::getCurrentCUDAStream();
  bias_gelu_kernel<<<blocks, threads, 0, stream>>>(
      x.data_ptr<float>(), bias.data_ptr<float>(), y.data_ptr<float>(), total, x.
size(1));
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return y;
}
```

验收

```
./check.sh c1              # 需要 NVIDIA GPU；第一次会编译约 1 分钟
./check.sh c1 --bench      # vector_add 的 GB/s；bias_gelu 融合 vs PyTorch 两个 kern
```

tests/test_c1_elementwise.py 31 行

```
import pytest
import torch
import torch.nn.functional as F

from drill import load
from tests._cuda import requires_cuda

pytestmark = requires_cuda


@pytest.mark.parametrize("n", [1, 255, 256, 257, 1_000_003])
def test_vector_add(n):
    m = load("c1_elementwise")
    a = torch.randn(n, device="cuda")
    b = torch.randn(n, device="cuda")
    torch.testing.assert_close(m.vector_add(a, b), a + b)


def test_vector_add_empty():
    m = load("c1_elementwise")
    a = torch.empty(0, device="cuda")
    assert m.vector_add(a, a).numel() == 0


@pytest.mark.parametrize("shape", [(1, 1), (3, 7), (128, 768), (1000, 4097)])
def test_bias_gelu(shape):
    m = load("c1_elementwise")
```

```
    x = torch.randn(shape, device="cuda") * 3
    bias = torch.randn(shape[1], device="cuda")
    ref = F.gelu(x + bias, approximate="tanh")
    torch.testing.assert_close(m.bias_gelu(x, bias), ref, rtol=1e-5, atol=1e-5)
```

提示

L1 先写对，再关心性能 方向（轻度剧透） 收起

- 每个线程只做一件事：算出自己的 `i`，越界就什么都不做。
- `blocks` 是向上取整的除法；n 恰好是 256 的倍数时不能多出一个空 block，不是倍数时不能少一个。
- `bias_gelu` 的 grid 被截断到 `sms * 32`，所以线程必须**循环**：每次跳过整个 grid 的线程数。

L2 常见坑 拆解与常见坑（中度剧透） 收起

- `blockIdx.x * blockDim.x` 在 32-bit 下计算，n 超过 2^{32} 会溢出：先转 `int64_t`。
- grid-stride 的步长是 `blockDim.x * gridDim.x`（整个 grid 的线程数），不是 `blockDim.x`。
- `x * x * x` 比 `powf(x, 3)` 更快也更准。
- 忘记 `C10_CUDA_KERNEL_LAUNCH_CHECK()` 时，`blocks = 0` 这种错误配置会被静默吞掉，直到后面某个不相干的地方才报错。

L3 **核心几行** 几乎是答案（重度剧透） 收起 ▾

```
// vector_add_kernel
int64_t i = (int64_t)blockIdx.x * blockDim.x + threadIdx.x;
if (i < n) c[i] = a[i] + b[i];
// host
const int64_t blocks = (n + threads - 1) / threads;

// bias_gelu_kernel
const int64_t stride = (int64_t)blockDim.x * gridDim.x;
for (int64_t i = (int64_t)blockIdx.x * blockDim.x + threadIdx.x; i < total; i +
    y[i] = gelu_tanh(x[i] + bias[i % cols]);
```

参考解

解 **solutions/c1_elementwise.cu** 先自己写完、跑过测试再看 收起

solutions/c1_elementwise.cu 94 行

```
// C1 · Vector add + fused bias-GELU — reference solution.
//
// Two kernels, two indexing styles:
//   vector_add_kernel : one thread per element, grid sized to cover n exactl
y (+ bounds check).
//   bias_gelu_kernel  : grid-stride loop, grid capped at ~32 blocks per SM,
so one launch
//                       works for any size and each thread does several elem
ents.
#include <torch/extension.h>
#include <ATen/cuda/CUDAContext.h>
#include <c10/cuda/CUDAException.h>

#include <algorithm>

#define CHECK_F32_CUDA(t)                                                   \
  TORCH_CHECK((t).is_cuda(), #t " must be a CUDA tensor");                   \
  TORCH_CHECK((t).is_contiguous(), #t " must be contiguous");               \
  TORCH_CHECK((t).scalar_type() == at::kFloat, #t " must be float32")

// ---------------------------------------------------------------- vector ad
d
// __restrict__ promises the compiler that a, b, c never alias, so it may kee
p values in
// registers and use the read-only data path for a and b.
__global__ void vector_add_kernel(const float* __restrict__ a,
                                  const float* __restrict__ b,
                                  float* __restrict__ c,
                                  int64_t n) {
  // Global index of this thread. Cast before multiplying: blockIdx.x * block
Dim.x is
  // computed in 32-bit unsigned arithmetic and overflows once n > 2^32.
  int64_t i = (int64_t)blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) {            // the last block is usually only partly full
    c[i] = a[i] + b[i];   // consecutive threads touch consecutive floats ->
coalesced
  }
}

torch::Tensor vector_add(torch::Tensor a, torch::Tensor b) {
  CHECK_F32_CUDA(a);
  CHECK_F32_CUDA(b);
```

```
  TORCH_CHECK(a.numel() == b.numel(), "a and b must have the same number of e
lements");
  auto c = torch::empty_like(a);
  const int64_t n = a.numel();
  if (n == 0) return c;  // a launch with 0 blocks is an error

  const int threads = 256;                              // a multiple of 32
(warp size)
  const int64_t blocks = (n + threads - 1) / threads;    // ceil(n / threads)
  // Launch on PyTorch's current stream so the kernel is ordered with other t
orch ops.
  cudaStream_t stream = at::cuda::getCurrentCUDAStream();
  vector_add_kernel<<<blocks, threads, 0, stream>>>(
      a.data_ptr<float>(), b.data_ptr<float>(), c.data_ptr<float>(), n);
  C10_CUDA_KERNEL_LAUNCH_CHECK();  // surfaces bad launch configs immediately
  return c;
}

// ------------------------------------------------------------------ fused bia
s + GELU
// GELU, tanh approximation (what torch.nn.functional.gelu(approximate="tan
h") computes):
//   0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 x^3)))
__device__ __forceinline__ float gelu_tanh(float x) {
  const float k0 = 0.7978845608028654f;  // sqrt(2 / pi)
  const float k1 = 0.044715f;
  return 0.5f * x * (1.0f + tanhf(k0 * (x + k1 * x * x * x)));
}

// x: [rows, cols] row-major, bias: [cols], y: [rows, cols].
// Unfused PyTorch does `t = x + bias` (read x, write t) then `gelu(t)` (read
t, write y):
// 4 passes over memory. Fused: read x once, write y once -> about half the t
raffic.
__global__ void bias_gelu_kernel(const float* __restrict__ x,
                                 const float* __restrict__ bias,
                                 float* __restrict__ y,
                                 int64_t total,
                                 int64_t cols) {
  const int64_t stride = (int64_t)blockDim.x * gridDim.x;  // threads in the
whole grid
  for (int64_t i = (int64_t)blockIdx.x * blockDim.x + threadIdx.x; i < total;
i += stride) {
    // Row-major: element i sits in column i % cols. bias is tiny and stays i
n L1/L2.
    y[i] = gelu_tanh(x[i] + bias[i % cols]);
  }
}

torch::Tensor bias_gelu(torch::Tensor x, torch::Tensor bias) {
```

```
  CHECK_F32_CUDA(x);
  CHECK_F32_CUDA(bias);
  TORCH_CHECK(x.dim() == 2, "x must be 2-D [rows, cols]");
  TORCH_CHECK(bias.dim() == 1 && bias.size(0) == x.size(1), "bias must be [co
ls]");
  auto y = torch::empty_like(x);
  const int64_t total = x.numel();
  if (total == 0) return y;

  const int threads = 256;
  // Enough blocks to fill every SM several times over, but no more: the grid
-stride loop
  // covers the rest, and fewer blocks means less scheduling overhead for hug
e tensors.
  const int sms = at::cuda::getCurrentDeviceProperties()->multiProcessorCoun
t;
  const int64_t blocks = std::min<int64_t>((total + threads - 1) / threads,
(int64_t)sms * 32);
  cudaStream_t stream = at::cuda::getCurrentCUDAStream();
  bias_gelu_kernel<<<blocks, threads, 0, stream>>>(
      x.data_ptr<float>(), bias.data_ptr<float>(), y.data_ptr<float>(), tota
l, x.size(1));
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return y;
}
```

测完之后

参考解在 RTX 5090（峰值 ~1.79 TB/s）上的实测：

测试	PyTorch	参考解
vector_add，n = 2^{20}	10.2 us · 1229 GB/s	10.2 us · 1229 GB/s
vector_add，n = 2^{27}	1028 us · 1567 GB/s	1027 us · 1568 GB/s（峰值的 ~87%）
bias_gelu 8192×8192	706 us（两个 kernel）	**379 us**（融合，快 1.86 倍）

- **算有效带宽**：vector_add 每个元素读 8 字节、写 4 字节。大 n 时和 PyTorch 完全一样：两边都已经被显存带宽卡住了，这正是“memory-bound kernel 的上限就是带宽”。
- **n 再小一些**（比如 2^{16}）时，时间会停在几微秒不再下降：这是 launch 开销，说明此时测的不是 kernel 本身。
- **融合的收益**：字节数减半，时间也接近减半。
- 试一试：threads 改成 32、64、128、512、1024，带宽有什么变化？为什么 32 明显更慢？（提示：每个 SM 同时驻留的 block 数有上限，block 太小时驻留的 warp 不够，无

法隐藏访存延迟。）

- 进阶：用 `float4` 一次读 16 字节（要求 n 是 4 的倍数且地址 16 字节对齐），看看带宽能不能再涨一点。

▼ 面试追问：为什么 elementwise kernel 的性能和 block 大小关系不大，却对融合非常敏感?

elementwise 的算术强度极低，时间 ≈ 读写字节数 / 带宽。block 大小只要能让 SM 上驻留足够多的 warp 来隐藏延迟（通常 128–512 都可以），带宽就能跑满，再调也没有收益。融合直接减少了必须搬运的字节数和 kernel 数，相当于改变了分子，所以收益是成比例的。

核心 · Track C · CUDA C++ · 第 2 题

C2 Reduction：warp shuffle + shared memory + atomic

把一百万个数加成一个数：每一层（线程、warp、block、grid）用这一层最便宜的通信手段。softmax、LayerNorm、attention 里的 max 和 sum 都是这个结构。

45 min | 难度 中 | 你写 `mine/c2_reduce_sum.cu` | 验收 `./check.sh c2` | 参考解 `solutions/c2_reduce_sum.cu`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

reduction 是很多 kernel 的骨架：softmax 要 max 和 sum，LayerNorm 要 mean 和 var，attention 每行都在做 max / sum，loss 要对整个 batch 求和。难点在于 GPU 上成千上万个线程要把结果**汇合到一个地方**，而不同层级之间的通信代价差了几个数量级：

层级	通信手段	代价
线程内	寄存器累加	几乎免费
warp 内	`__shfl_down_sync` 直接交换寄存器	每轮几个周期，无需同步
block 内	shared memory + `__syncthreads()`	需要一次 barrier
block 之间	global memory 的 `atomicAdd`，或第二个 kernel	最贵

好的 reduction 尽量在便宜的层级做完，只把极少量数据交给更贵的层级。

源码与资料

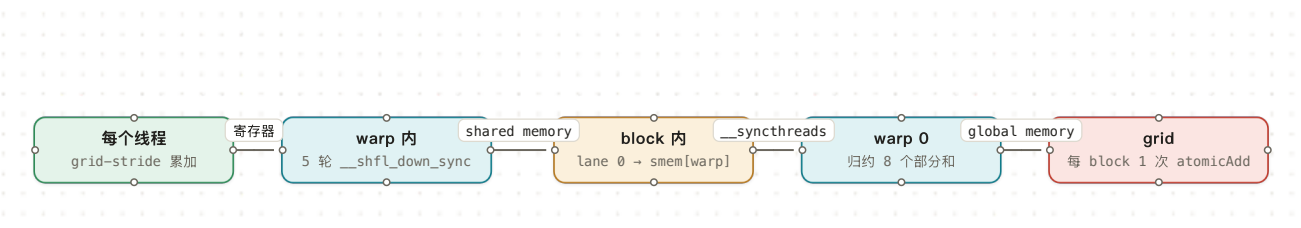
Mark Harris: Optimizing Parallel Reduction in CUDA · `cudasamples › cuda-samples reduction` · `pytorch › PyTorch Reduce.cuh` · CUB BlockReduce · Warp-level primitives

要用到的知识

- `__shfl_down_sync(0xffffffff, v, offset)`：读取 `lane + offset` 号 lane 的 v。offset 依次取 16、8、4、2、1，5 轮之后 lane 0 持有整个 warp 的和。
- `lane = threadIdx.x & 31`，`warp = threadIdx.x >> 5`。
- `__shared__ float warp_sums[WARPS];`：每个 warp 一个槽位。
- `__syncthreads()`：所有 warp 写完自己的槽位之后，warp 0 才能去读。
- `atomicAdd(out, v)`：每个 block 只做一次。
- 模板参数 `template <int BLOCK>`：让 block 大小成为编译期常量，shared memory 数组的大小才能用它。

图解与类比

reduction 的四层结构

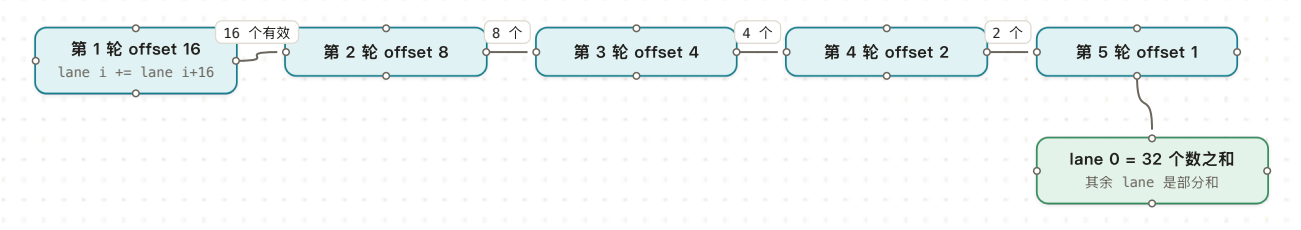


越往右通信越贵，所以越往右参与的数据越少：几百万个数在寄存器里就被压成几十万个，到 atomic 这一层只剩几百个。

类比 · 选举计票

每个投票站（线程）先自己数完；同一个镇的投票站用对讲机报数汇总（warp shuffle，不用开会）；县里把各镇代表叫到会议室开一次会（shared memory + `__syncthreads`）；最后每个县只给省里打一个电话（atomicAdd）。 电话打得越少越好，所以要让每个投票站尽量多数一些票（grid-stride loop）。atomic 的先后顺序不确定，所以最后几位小数可能每次不同。

__shfl_down_sync 的 5 轮（只看 lane 0 的视角）



类比 · 淘汰赛

32 名选手打淘汰赛：第一轮 1 号对 17 号、2 号对 18 号……（offset 16），赢家把对手的分数加到自己身上；每轮人数减半，5 轮之后冠军（lane 0）手里就是全部分数之和。整场比赛在一张桌上完成（寄存器之间直接交换），不用换场地（shared memory）。

题目要求

完成 `mine/c2_reduce_sum.cu` 中的 `warp_reduce_sum` 和 `reduce_sum_kernel<BLOCK>`：

- 输入 float32 一维连续张量 x，输出 0 维张量 `x.sum()`；host 端已经把 out 初始化为 0。
- grid 被截断为 `min(ceil(n/BLOCK), SM 数 × 4)`，所以每个线程必须用 grid-stride loop 累加多个元素。
- 每个 block 恰好执行**一次** `atomicAdd`。
- 精度：与 float64 reference 的相对误差在 `1e-4` 以内（测试用 `[0, 1)` 均匀分布）。

题干

mine/c2_reduce_sum.cu 40 行

```
// C2 · Reduction sum — YOUR IMPLEMENTATION.
// thread (grid-stride loop) -> warp (__shfl_down_sync) -> block (shared memory)
-> grid (atomicAdd)
#include <torch/extension.h>
#include <ATen/cuda/CUDAContext.h>
#include <c10/cuda/CUDAException.h>

#include <algorithm>

// Return the sum of v over the 32 lanes of the warp (only lane 0 needs the right
answer).
__device__ __forceinline__ float warp_reduce_sum(float v) {
  // TODO: 5 rounds of __shfl_down_sync(0xffffffff, v, offset), offset = 16, 8,
4, 2, 1
  return v;
}

template <int BLOCK>
__global__ void reduce_sum_kernel(const float* __restrict__ x, float* __restrict_
_ out, int64_t n) {
  static_assert(BLOCK % 32 == 0 && BLOCK <= 1024, "BLOCK must be a multiple of 3
2, at most 1024");
```

```
  constexpr int WARPS = BLOCK / 32;
  __shared__ float warp_sums[WARPS];
  // TODO:
  //   1. each thread: grid-stride loop, private running sum
  //   2. warp_reduce_sum
  //   3. lane 0 of each warp -> warp_sums[warp]; __syncthreads(); warp 0 reduces
warp_sums
  //   4. lane 0 of warp 0: atomicAdd(out, block_sum)
}

torch::Tensor reduce_sum(torch::Tensor x) {
  TORCH_CHECK(x.is_cuda() && x.is_contiguous() && x.scalar_type() == at::kFloat,
              "x must be a contiguous float32 CUDA tensor");
  auto out = torch::zeros({}, x.options());
  const int64_t n = x.numel();
  if (n == 0) return out;
  constexpr int BLOCK = 256;
  const int sms = at::cuda::getCurrentDeviceProperties()->multiProcessorCount;
  const int64_t blocks = std::min<int64_t>((n + BLOCK - 1) / BLOCK, (int64_t)sms
* 4);
  reduce_sum_kernel<BLOCK><<<blocks, BLOCK, 0, at::cuda::getCurrentCUDAStream()>>
>(
      x.data_ptr<float>(), out.data_ptr<float>(), n);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return out;
}
```

验收

```
./check.sh c2
./check.sh c2 --bench       # vs torch.sum: GB/s
```

tests/test_c2_reduce_sum.py 22 行

```
import pytest
import torch

from drill import load
from tests._cuda import requires_cuda

pytestmark = requires_cuda


@pytest.mark.parametrize("n", [1, 31, 32, 33, 255, 256, 1000, 65_537, 10_000_01
9])
def test_reduce_sum(n):
    m = load("c2_reduce_sum")
```

```
    x = torch.rand(n, device="cuda")            # positive values: relative error
is meaningful
    out = m.reduce_sum(x)
    assert out.shape == () and out.dtype == torch.float32
    ref = x.double().sum().float()
    torch.testing.assert_close(out, ref, rtol=1e-4, atol=1e-4)


def test_reduce_sum_empty():
    m = load("c2_reduce_sum")
    assert m.reduce_sum(torch.empty(0, device="cuda")).item() == 0.0
```

提示

L1 四层结构 方向（轻度剧透） 收起

1. 线程：`float v = 0; for (grid-stride) v += x[i];`
2. warp：`v = warp_reduce_sum(v);`，之后只有 lane 0 的值有意义。
3. block：lane 0 写 `warp_sums[warp]`，`__syncthreads()`，然后让 warp 0 的前 `WARPS` 个 lane 读回来，再做一次 `warp_reduce_sum`。
4. grid：warp 0 的 lane 0 做 `atomicAdd(out, v)`。

L2 常见坑 拆解与常见坑（中度剧透） 收起

- `__syncthreads()` 不能放在 `if (lane == 0)` 里面：所有线程都必须执行到它。
- 第二轮 warp 归约时，`lane >= WARPS` 的 lane 要贡献 0，不能读 `warp_sums` 越界。
- `__shfl_down_sync` 的掩码必须包含所有调用它的 lane。在 `if (warp == 0)` 里面整个 warp 0 都进入了，所以用 `0xffffffff` 是对的。
- 测试反复跑结果末位会变：这是浮点原子加的顺序不确定，不是 bug。

L3 **核心代码** 几乎是答案（重度剧透） 收起 ▾

```
for (int offset = 16; offset > 0; offset >>= 1) v += __shfl_down_sync(0xffffffff

// kernel 里
const int lane = threadIdx.x & 31, warp = threadIdx.x >> 5;
if (lane == 0) warp_sums[warp] = v;
__syncthreads();
if (warp == 0) {
  v = (lane < WARPS) ? warp_sums[lane] : 0.0f;
  v = warp_reduce_sum(v);
  if (lane == 0) atomicAdd(out, v);
}
```

参考解

解 **solutions/c2_reduce_sum.cu** 先自己写完、跑过测试再看 收起

solutions/c2_reduce_sum.cu 73 行

```cpp
// C2 · Reduction sum — reference solution.
//
// Three levels, each using the cheapest mechanism available at that level:
//   1. thread : a grid-stride loop keeps a private running sum in a registe
r;
//   2. warp   : 5 rounds of __shfl_down_sync combine 32 lanes without touchi
ng memory;
//   3. block  : each warp's lane 0 writes one partial to shared memory, warp
 0 reduces those;
//   4. grid   : one atomicAdd per block onto the single output float.
#include <torch/extension.h>
#include <ATen/cuda/CUDAContext.h>
#include <c10/cuda/CUDAException.h>

#include <algorithm>

// Sum of v across the 32 lanes of a warp; the full result ends up in lane 0.
// Round 1: lane i adds lane i+16; round 2: lane i adds lane i+8; ... 16, 8,
4, 2, 1.
// 0xffffffff = all 32 lanes take part (they must all reach this call).
__device__ __forceinline__ float warp_reduce_sum(float v) {
#pragma unroll
  for (int offset = 16; offset > 0; offset >>= 1) {
    v += __shfl_down_sync(0xffffffff, v, offset);
  }
  return v;
}

template <int BLOCK>
__global__ void reduce_sum_kernel(const float* __restrict__ x, float* __restr
ict__ out, int64_t n) {
  static_assert(BLOCK % 32 == 0 && BLOCK <= 1024, "BLOCK must be a multiple o
f 32, at most 1024");
  constexpr int WARPS = BLOCK / 32;
  __shared__ float warp_sums[WARPS];  // one slot per warp in this block

  // 1. Thread level. Consecutive threads read consecutive elements on every
iteration,
  //    so each warp-wide load is coalesced. Out-of-range threads simply add
nothing.
  float v = 0.0f;
  const int64_t stride = (int64_t)BLOCK * gridDim.x;
```

```
  for (int64_t i = (int64_t)blockIdx.x * BLOCK + threadIdx.x; i < n; i += str
ide) {
    v += x[i];
  }

  // 2. Warp level.
  v = warp_reduce_sum(v);

  // 3. Block level: lane 0 of every warp publishes its warp's sum ...
  const int lane = threadIdx.x & 31;  // threadIdx.x % 32
  const int warp = threadIdx.x >> 5;  // threadIdx.x / 32
  if (lane == 0) warp_sums[warp] = v;
  __syncthreads();  // ... and every warp must have written before warp 0 rea
ds

  // ... then the first warp reduces the WARPS partials (lanes >= WARPS contr
ibute 0).
  if (warp == 0) {
    v = (lane < WARPS) ? warp_sums[lane] : 0.0f;
    v = warp_reduce_sum(v);
    // 4. Grid level: one atomic per block. Float addition is not associativ
e, so the
    //    order in which blocks arrive can change the last bits of the resul
t.
    if (lane == 0) atomicAdd(out, v);
  }
}

torch::Tensor reduce_sum(torch::Tensor x) {
  TORCH_CHECK(x.is_cuda() && x.is_contiguous() && x.scalar_type() == at::kFlo
at,
              "x must be a contiguous float32 CUDA tensor");
  auto out = torch::zeros({}, x.options());  // atomicAdd accumulates into th
is, so start at 0
  const int64_t n = x.numel();
  if (n == 0) return out;

  constexpr int BLOCK = 256;
  // A few blocks per SM is enough to saturate bandwidth; more blocks only ad
d atomics.
  const int sms = at::cuda::getCurrentDeviceProperties()->multiProcessorCoun
t;
  const int64_t blocks = std::min<int64_t>((n + BLOCK - 1) / BLOCK, (int64_t)
sms * 4);
  cudaStream_t stream = at::cuda::getCurrentCUDAStream();
  reduce_sum_kernel<BLOCK><<<blocks, BLOCK, 0, stream>>>(x.data_ptr<float>(),
out.data_ptr<float>(), n);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return out;
}
```

测完之后

- n = 2^{26} 时应接近峰值带宽：只读 4n 字节，最终只有几百次 atomic。RTX 5090 实测：参考解 176 us · **1524 GB/s**，`torch.sum` 198 us · 1358 GB/s；n = 2^{20} 时两者都只有 ~10 us，此时主要是 launch 开销。
- **对比实验**：把 warp shuffle 换成教科书式的 shared memory 树形归约（`for (s = BLOCK/2; s > 0; s >>= 1) { if (tid < s) sm[tid] += sm[tid + s]; __syncthreads(); }`），用 `ncu` 比较 barrier 次数和时间。
- **确定性**：如果要求每次结果逐位一致（比如训练的可复现性），把 atomic 换成两趟：第一个 kernel 把每个 block 的部分和写进数组，第二个 kernel 用一个 block 按固定顺序归约它们。
- Triton 里这一整套只是一行 `tl.sum(x, axis=0)`：T2、T3 会用到。G1 会解释编译器怎么把它拆成线程内累加、shuffle 和 shared memory 这三层。

▼ 面试追问：为什么 grid 只开 SM 数 × 4 个 block，而不是 n / 256 个?

每个 block 最后都要做一次 atomicAdd，block 越多，同一地址上排队的原子操作越多；而只要每个 SM 上有几个 block 在跑，带宽就已经跑满了。让每个线程用 grid-stride loop 多累加几个元素，大部分归约都留在寄存器里完成，最便宜。

核心 · Track C · CUDA C++ · 第 3 题

C3 矩阵转置：coalescing 与 bank conflict

转置不做任何计算，却能把带宽从 30% 提到 90%：这题专门练合并访存和 shared memory 的 bank conflict。

40 min | 难度 中 | 你写 `mine/c3_transpose.cu` | 验收 `./check.sh c3` | 参考解 `solutions/c3_transpose.cu`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

转置本身很常见（layout 转换、`x.t().contiguous()`、attention 里的 K^T、各种 permute），但它真正的价值是**教学**：读和写不可能同时对行主序连续。naive 版本要么读是合并的、写是跨步的，要么反过来。解决办法是用 shared memory 当“中转站”，这和 GEMM、FlashAttention 里 staging tile 的思路完全一样。

源码与资料

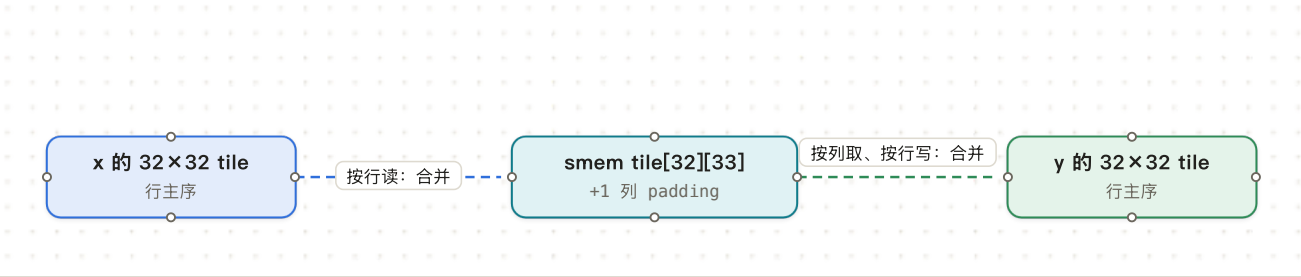
An Efficient Matrix Transpose in CUDA C/C++ · `cudasamples › cuda-samples transpose` · `cutlass › CuTe swizzle.hpp`（不用 padding、用异或消除 bank conflict） · `flashinfer › FlashInfer permuted_smem.cuh`（FlashInfer 的 swizzle 布局）

要用到的知识

- 二维 launch：`dim3 block(32, 8)`，`dim3 grid(ceil(N/32), ceil(M/32))`。
- 约定 `threadIdx.x` 走内存中连续的维度。
- `__shared__ float tile[32][33];` 以及 `__syncthreads()`。
- **bank conflict**：`tile[i][j]` 的地址是 `i·33 + j`，所在 bank 是 `(i·33 + j) % 32`。没有 padding 时是 `(i·32 + j) % 32 = j`：一个 warp 读一列（i 变、j 固定）时，32 个线程全部落在同一个 bank，访问被串行化成 32 次。

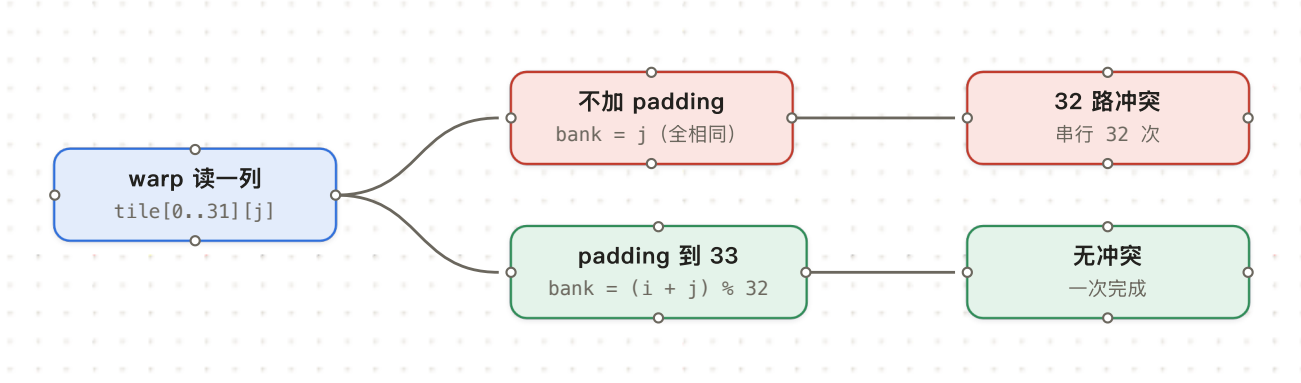
图解与类比

用 shared memory 中转的转置



naive 版本里“读”和“写”总有一个是跨步的；中转之后，跨步访问只发生在 shared memory 里，而 shared memory 不怕跨步，只怕 bank conflict。

为什么要多加一列



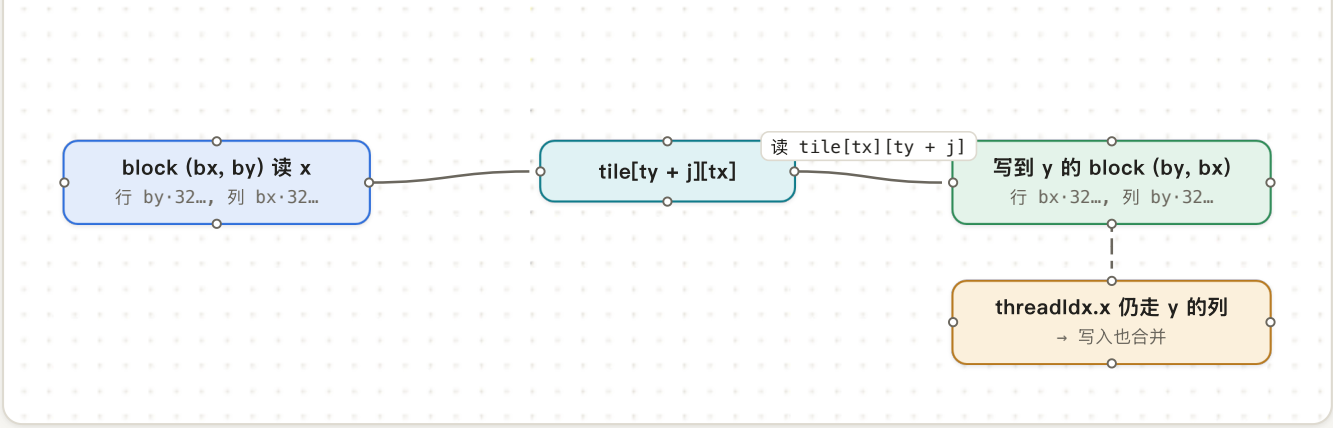
类比 · 32 个收银台

shared memory 有 32 个 bank，就像 32 个收银台。一列数据在不加 padding 时全部落在同一个台（每行正好 32 个数，绕一圈回到原位），32 个人排一条队； 每行多放一个空位后，第 i 行的第 j 个数落在第 (i + j) 个台，32 个人正好分散到 32 个台。

类比 · 搬书

naive 转置像从书架 A 按行取书、再按列塞进书架 B：每塞一本都要走到另一排。shared memory 版本是先按行把一摞书搬到桌上，再按书架 B 顺手的方向一排一排放进去：两边都是顺着走。

block 坐标在写回时对调



> **类比 · 换座位**
>
> 转置就是让 (行, 列) 互换座位。如果让每个人直接走到对面的新座位，写入是跨步的（每个人都要穿过整个礼堂）；先在小会议室（shared memory）里按新的排法站好队，再整排整排地走进新礼堂，每一排都是挨着走的。

题目要求

完成 `mine/c3_transpose.cu` 的两个 kernel（host 代码和 launch 配置已给出）：

1. **`transpose_naive_kernel`**：block (32, 8)，一个线程搬一个元素，`y[c][r] = x[r][c]`。
2. **`transpose_tiled_kernel`**：block (32, 8) 负责一个 32×32 的 tile，每个线程搬 4 行。
   - 第 1 步：合并地读 x 的 tile 到 shared memory；
   - `__syncthreads()`；
   - 第 2 步：交换 block 坐标，让 `threadIdx.x` 仍然沿 y 的行方向走，合并地写出。

x 为 `[M, N]` 连续 float32，M、N 任意（包括 1、不是 32 的倍数）。结果必须**逐位相等**。

题干

`mine/c3_transpose.cu` 51 行

```
// C3 · Matrix transpose — YOUR IMPLEMENTATION.  x: [M, N] -> y: [N, M], y[c][r] = x[r][c]
#include <torch/extension.h>
#include <ATen/cuda/CUDAContext.h>
#include <c10/cuda/CUDAException.h>
```

```cpp
constexpr int TILE = 32;
constexpr int ROWS = 8;

// block (32, 8), one thread per element of x.
__global__ void transpose_naive_kernel(const float* __restrict__ x, float* __rest
rict__ y, int M, int N) {
  // TODO
}

// block (TILE, ROWS) = 256 threads per 32x32 tile; each thread copies TILE / ROW
S rows.
// Stage the tile in shared memory so both the global read and the global write a
re coalesced.
__global__ void transpose_tiled_kernel(const float* __restrict__ x, float* __rest
rict__ y, int M, int N) {
  __shared__ float tile[TILE][TILE + 1];  // why + 1? (see hints)
  // TODO: phase 1 read x tile -> shared, __syncthreads(), phase 2 write shared -
> y tile
}

static void check_input(const torch::Tensor& x) {
  TORCH_CHECK(x.is_cuda() && x.is_contiguous() && x.scalar_type() == at::kFloat,
              "x must be a contiguous float32 CUDA tensor");
  TORCH_CHECK(x.dim() == 2, "x must be 2-D");
}

torch::Tensor transpose_naive(torch::Tensor x) {
  check_input(x);
  const int M = x.size(0), N = x.size(1);
  auto y = torch::empty({N, M}, x.options());
  if (M == 0 || N == 0) return y;
  dim3 block(32, 8);
  dim3 grid((N + block.x - 1) / block.x, (M + block.y - 1) / block.y);
  transpose_naive_kernel<<<grid, block, 0, at::cuda::getCurrentCUDAStream()>>>(
      x.data_ptr<float>(), y.data_ptr<float>(), M, N);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return y;
}

torch::Tensor transpose_tiled(torch::Tensor x) {
  check_input(x);
  const int M = x.size(0), N = x.size(1);
  auto y = torch::empty({N, M}, x.options());
  if (M == 0 || N == 0) return y;
  dim3 block(TILE, ROWS);
  dim3 grid((N + TILE - 1) / TILE, (M + TILE - 1) / TILE);
  transpose_tiled_kernel<<<grid, block, 0, at::cuda::getCurrentCUDAStream()>>>(
      x.data_ptr<float>(), y.data_ptr<float>(), M, N);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
```

```
  return y;
}
```

验收

```
./check.sh c3
./check.sh c3 --bench       # naive vs tiled vs x.t().contiguous(), 单位 GB/s
```

tests/test_c3_transpose.py 19 行

```
import pytest
import torch

from drill import load
from tests._cuda import requires_cuda

pytestmark = requires_cuda

SHAPES = [(1, 1), (1, 100), (100, 1), (32, 32), (33, 31), (257, 129), (1024, 409
6)]


@pytest.mark.parametrize("fn", ["transpose_naive", "transpose_tiled"])
@pytest.mark.parametrize("shape", SHAPES)
def test_transpose(fn, shape):
    m = load("c3_transpose")
    x = torch.randn(shape, device="cuda")
    y = getattr(m, fn)(x)
    assert y.shape == (shape[1], shape[0])
    torch.testing.assert_close(y, x.t().contiguous(), rtol=0, atol=0)   # a copy
must be exact
```

提示

L1 **画出两个坐标系** 方向（轻度剧透） 收起

block `(bx, by)` 负责 x 中行 `by·32 … by·32+31`、列 `bx·32 … bx·32+31` 的 tile。这个 tile 转置后是 y 中行 `bx·32 …`、列 `by·32 …` 的 tile。写回时把 bx 和 by 的角色对调，同时让 **`threadIdx.x` 继续对应输出的列**，这样写也是连续的。

L2 下标对照 拆解与常见坑（中度剧透） 收起

- 读：`c = bx·32 + tx`，`r = by·32 + ty + j`（j = 0, 8, 16, 24），`tile[ty + j][tx] = x[r][c]`。
- 写：`yc = by·32 + tx`（y 的列 = x 的行），`yr = bx·32 + ty + j`（y 的行 = x 的列），`y[yr][yc] = tile[tx][ty + j]`。
- 边界：读时检查 `r < M && c < N`，写时检查 `yr < N && yc < M`。
- 两个阶段之间必须 `__syncthreads()`：第 2 步读的是**别的线程**在第 1 步写进去的数据。

L3 关键几行 几乎是答案（重度剧透） 收起

```
for (int j = 0; j < TILE; j += ROWS)
  if (r0 + j < M && c < N) tile[threadIdx.y + j][threadIdx.x] = x[(int64_t)(r0
__syncthreads();
for (int j = 0; j < TILE; j += ROWS)
  if (yr0 + j < N && yc < M) y[(int64_t)(yr0 + j) * M + yc] = tile[threadIdx.x]
```

其中 `r0 = blockIdx.y*TILE + threadIdx.y`，`yc = blockIdx.y*TILE + threadIdx.x`，`yr0 = blockIdx.x*TILE + threadIdx.y`。

参考解

解 **solutions/c3_transpose.cu** 先自己写完、跑过测试再看 收起

solutions/c3_transpose.cu 85 行

```
// C3 · Matrix transpose — reference solution.
//
// y[c][r] = x[r][c] for x: [M, N] -> y: [N, M], both row-major float32.
//
// naive : each thread reads x[r][c] (coalesced: consecutive threads, consecutive c) and
//         writes y[c][r] (strided by M floats: every lane of the warp hits a different
//         memory segment). Writes become ~32 separate transactions per warp.
// tiled : a block stages a 32x32 tile in shared memory, then reads it back "sideways"
//         so that the global write is also coalesced. Shared memory is padded to 33
//         columns so the sideways read does not hit the same bank 32 times.
#include <torch/extension.h>
#include <ATen/cuda/CUDAContext.h>
#include <c10/cuda/CUDAException.h>

constexpr int TILE = 32;  // tile edge = warp size: one warp covers one tile row
constexpr int ROWS = 8;   // threads per block in y; each thread copies TILE / ROWS = 4 rows

__global__ void transpose_naive_kernel(const float* __restrict__ x, float* __restrict__ y, int M, int N) {
  const int c = blockIdx.x * blockDim.x + threadIdx.x;  // column in x (fast index)
  const int r = blockIdx.y * blockDim.y + threadIdx.y;  // row in x
  if (r < M && c < N) {
    y[(int64_t)c * M + r] = x[(int64_t)r * N + c];
  }
}

__global__ void transpose_tiled_kernel(const float* __restrict__ x, float* __restrict__ y, int M, int N) {
  // +1 column of padding: element tile[i][j] lives in bank (i * 33 + j) % 32 = (i + j) % 32,
  // so a warp reading a column tile[0..31][j] touches 32 different banks.
  // Without padding, (i * 32 + j) % 32 = j: all 32 lanes hit one bank -> 32-way conflict.
  __shared__ float tile[TILE][TILE + 1];
```

```
  // Phase 1: coalesced read of x's tile (blockIdx.y, blockIdx.x) into shared
memory.
  const int c = blockIdx.x * TILE + threadIdx.x;   // column in x
  const int r0 = blockIdx.y * TILE + threadIdx.y;  // first row in x handled
by this thread
#pragma unroll
  for (int j = 0; j < TILE; j += ROWS) {
    const int r = r0 + j;
    if (r < M && c < N) tile[threadIdx.y + j][threadIdx.x] = x[(int64_t)r * N
+ c];
  }

  __syncthreads();  // all of the tile must be in shared memory before anyone
reads it back

  // Phase 2: x's tile (by, bx) becomes y's tile (bx, by). Swap the block coo
rdinates and
  // let threadIdx.x run along y's rows again, so the global write is coalesc
ed too.
  const int yc = blockIdx.y * TILE + threadIdx.x;   // column in y  (= row in
x)
  const int yr0 = blockIdx.x * TILE + threadIdx.y;  // row in y     (= column
in x)
#pragma unroll
  for (int j = 0; j < TILE; j += ROWS) {
    const int yr = yr0 + j;
    // y[yr][yc] = x[yc][yr] = tile[yc - by*TILE][yr - bx*TILE] = tile[thread
Idx.x][threadIdx.y + j]
    if (yr < N && yc < M) y[(int64_t)yr * M + yc] = tile[threadIdx.x][threadI
dx.y + j];
  }
}

static void check_input(const torch::Tensor& x) {
  TORCH_CHECK(x.is_cuda() && x.is_contiguous() && x.scalar_type() == at::kFlo
at,
              "x must be a contiguous float32 CUDA tensor");
  TORCH_CHECK(x.dim() == 2, "x must be 2-D");
}

torch::Tensor transpose_naive(torch::Tensor x) {
  check_input(x);
  const int M = x.size(0), N = x.size(1);
  auto y = torch::empty({N, M}, x.options());
  if (M == 0 || N == 0) return y;
  dim3 block(32, 8);
  dim3 grid((N + block.x - 1) / block.x, (M + block.y - 1) / block.y);
  transpose_naive_kernel<<<grid, block, 0, at::cuda::getCurrentCUDAStream()>>
>(
      x.data_ptr<float>(), y.data_ptr<float>(), M, N);
```

```
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return y;
}

torch::Tensor transpose_tiled(torch::Tensor x) {
  check_input(x);
  const int M = x.size(0), N = x.size(1);
  auto y = torch::empty({N, M}, x.options());
  if (M == 0 || N == 0) return y;
  dim3 block(TILE, ROWS);                                    // 256 threads
  dim3 grid((N + TILE - 1) / TILE, (M + TILE - 1) / TILE);    // one block p
er 32x32 tile of x
  transpose_tiled_kernel<<<grid, block, 0, at::cuda::getCurrentCUDAStream()>>
>(
      x.data_ptr<float>(), y.data_ptr<float>(), M, N);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return y;
}
```

测完之后

- RTX 5090 上 8192×8192 的实测：naive 667 us · 805 GB/s，`x.t().contiguous()` 668 us · 804 GB/s（PyTorch 的通用拷贝 kernel 在这里也是跨步写），**tiled 381 us · 1409 GB/s**（峰值的 ~79%）。你在别的 GPU 上的数字会不同，但 tiled 应明显快于另外两个。
- **去掉 padding**（`tile[32][32]`）再测一次，用 `ncu` 的 `l1tex__data_bank_conflicts_pipe_lsu_mem_shared_op_ld` 指标看 bank conflict 的数量。
- 把 block 改成 (32, 32)，每个线程只搬一个元素：线程多了，速度却不一定更快。为什么？（每个线程只做一次 load / store，相对开销更大；block 越大，每个 SM 能同时驻留的 block 也越少。）
- 想一想：GEMM（C4）里 B 的 tile 读进 shared memory 之后，是按行读还是按列读？会不会冲突？

▼ 面试追问：为什么不直接让每个线程读 x[r][c]、写 y[c][r]？写入不合并到底慢在哪？

一个 warp 的 32 个线程写 y 的同一列（相邻 r），地址相隔 M·4 字节，落在 32 个不同的 32 字节 sector 上。硬件要发 32 个独立的写事务，每个事务只用 4 字节，有效带宽只剩 1/8。shared memory 中转之后，读和写都是 32 个连续 float，各需 4 个 sector。

核心 · Track C · CUDA C++ · 第 4 题

C4 GEMM 三连：naive → shared-memory tiling → register tiling

GEMM 是 GPU 性能工程的母题：每一级优化都在做同一件事，提高“每从慢存储读一个数，能做多少次乘加”。

90 min | 难度 难 | 你写 `mine/c4_gemm.cu` | 验收 `./check.sh c4` | 参考解 `solutions/c4_gemm.cu`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

Transformer 里绝大部分 FLOPs 都在矩阵乘：QKV 投影、MLP、LM head。计算 `C[M,N] = A[M,K] @ B[K,N]` 需要 `2MNK` FLOPs，而 A、B、C 一共只有 `MK + KN + MN` 个数。n = 4096 时，每个数平均被用了约 2700 次。**算法本身的复用度极高，问题是 naive 实现完全没利用这一点。**

版本	每次 FMA 需要的 global 读	关键思想
v0 naive	2 次（各读 A、B 一个数）	无复用，memory-bound
v1 shared memory tiling	2 / 32	32×32 的 tile 读进 shared memory，每个数在 block 内被复用 32 次
v2 register tiling	再减少 shared memory 读取	每个线程算 4×4 个输出：从 shared memory 读 4+4 个数，做 16 次 FMA
v3 Tensor Core（扩展）		`mma` 指令一次算一个小矩阵块；Triton 的 `tl.dot` 自动使用（T4）

源码与资料

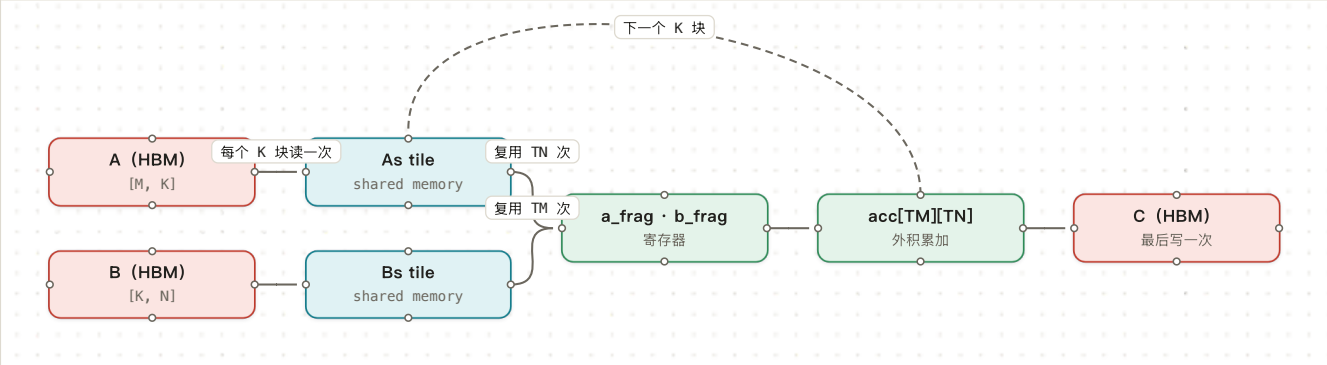
Simon Boehm: How to Optimize a CUDA Matmul Kernel · `cudasamples › cuda-samples matrixMul` · `cudasamples › WMMA Tensor Core GEMM` · `cutlass › CuTe sgemm_1` · `cutlass › CuTe sgemm_sm80` · `cutlass › CuTe wgmma + TMA (Hopper)` · `flashinfer › FlashInfer mma.cuh`

要用到的知识

- **二维 tile 的协作加载**：block 内每个线程搬一部分，`__syncthreads()` 之后所有人都能用整块。
- **越界填 0**：超出矩阵的部分装 0，对点积没有贡献，内层循环就不需要边界判断。
- **load → sync → compute → sync**：第二个 sync 防止下一轮的 load 覆盖还在被读的 tile。
- **寄存器数组**：`float acc[TM][TN]`。循环边界是编译期常量并且 `#pragma unroll` 时，数组会完全放进寄存器。
- **外积**（outer product）：`acc[i][j] += a[i] * b[j]`，读 TM + TN 个数，做 TM·TN 次 FMA。
- `__launch_bounds__(THREADS)`：告诉编译器 block 大小，便于分配寄存器。

图解与类比

GEMM 的数据流：一次读，多次用

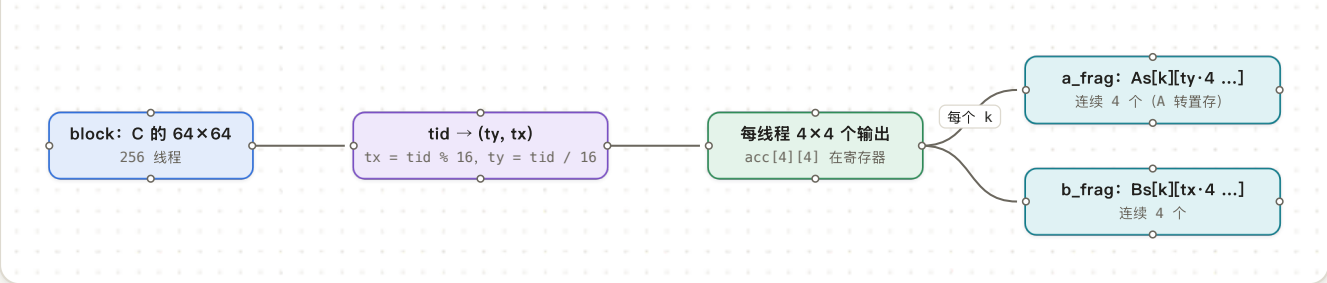


naive 每次乘加都要从 HBM 读 2 个数；shared memory tiling 让每个数在 block 内被用 32 次；register tiling 再让每个从 shared memory 读来的数被用 4 次。

类比 · 一次做 100 份菜

naive：每做一道菜就去仓库拿两样食材。shared-memory tiling：一次拿一整筐放到料理台，32 道菜共用这一筐。register tiling：每个厨师同时端着 4×4 = 16 个盘子，一次拿起来的配料直接分到 4 个盘子里。 筐越大越省路，但料理台和手的容量有限（shared memory、寄存器）：tile 太大，能同时干活的厨师就少了。

register tiling：256 个线程怎么分 64×64 的 C 块



类比 · 外积就是九九乘法表

手里有一列 4 个数（a_frag）和一行 4 个数（b_frag），把它们两两相乘就是一张 4×4 的乘法表：读 8 个数，做 16 次乘加。tile 越大，表越大，每读一个数能做的乘法越多，但每个人（线程）手里要拿的东西（寄存器）也越多，拿不下就会掉到地上（寄存器溢出到 local memory）。

题目要求

`mine/c4_gemm.cu` 里 `matmul_naive` 已作为示例完整给出。你要实现：

1. **`matmul_smem_kernel`**：block 32×32，每个线程算 C 的一个元素；沿 K 每次处理 32，A、B 的 tile 读进 shared memory。
2. **`matmul_regtile_kernel`**：BM = BN = 64，BK = 16，TM = TN = 4，每个 block 256 个线程；每个线程负责 C 中 4×4 的一块。A 的 tile 以**转置**方式存为 `As[BK][BM]`。

float32 输入、float32 累加；M、N、K 任意（包括 1 和非 tile 倍数）。与 fp64 reference 比较，容差 `1e-3`。

题干

mine/c4_gemm.cu 99 行

```
// C4 · GEMM: naive -> shared-memory tiling -> register tiling — YOUR IMPLEMENTAT
ION.
// matmul_naive is given as a worked example. Implement the two TODO kernels.
```

```
//
// C[M, N] = A[M, K] @ B[K, N], all row-major float32, fp32 accumulation.
//
// Arithmetic intensity is what changes between versions:
//   naive   : every multiply-add loads one A and one B value from global memory
(2 loads / FMA).
//   smem    : a 32x32 block loads a tile once into shared memory and reuses each
value 32 times.
//   regtile : each thread computes a 4x4 patch of C from registers; every value
read from
//             shared memory is used 4 times, so shared-memory traffic drops by 4
x again.
#include <torch/extension.h>
#include <ATen/cuda/CUDAContext.h>
#include <c10/cuda/CUDAException.h>

// ---------------------------------------------------------------- v0: naive
__global__ void matmul_naive_kernel(const float* __restrict__ A, const float* __r
estrict__ B,
                                    float* __restrict__ C, int M, int N, int K) {
  const int row = blockIdx.y * blockDim.y + threadIdx.y;
  const int col = blockIdx.x * blockDim.x + threadIdx.x;  // threadIdx.x -> colum
n: B and C reads coalesce
  if (row >= M || col >= N) return;
  float acc = 0.0f;
  for (int k = 0; k < K; ++k) acc += A[(int64_t)row * K + k] * B[(int64_t)k * N +
col];
  C[(int64_t)row * N + col] = acc;
}

// ---------------------------------------------------------------- v1: shared-me
mory tiling
constexpr int T = 32;  // block = T x T threads = one T x T tile of C

__global__ void matmul_smem_kernel(const float* __restrict__ A, const float* __re
strict__ B,
                                   float* __restrict__ C, int M, int N, int K) {
  __shared__ float As[T][T];
  __shared__ float Bs[T][T];
  // TODO: loop over K in steps of T: load one A tile and one B tile (zero-fill o
ut of range),
  //       __syncthreads(), accumulate T products, __syncthreads(). Store C if in
range.
}

// ---------------------------------------------------------------- v2: register
tiling
// Block tile BM x BN of C, K in steps of BK, each thread owns a TM x TN patch.
constexpr int BM = 64, BN = 64, BK = 16, TM = 4, TN = 4;
constexpr int THREADS = (BM / TM) * (BN / TN);  // 16 * 16 = 256
```

```
__global__ void __launch_bounds__(THREADS)
matmul_regtile_kernel(const float* __restrict__ A, const float* __restrict__ B,
                      float* __restrict__ C, int M, int N, int K) {
  __shared__ float As[BK][BM];  // A tile stored transposed: As[k][m]
  __shared__ float Bs[BK][BN];
  // TODO: tid -> (tx, ty); float acc[TM][TN] = {};
  //       for each K tile: cooperative loads (each thread loads BM*BK/THREADS va
lues of A and of B),
  //       __syncthreads(), for k in BK: load a_frag[TM], b_frag[TN] from shared,
outer product into acc,
  //       __syncthreads(). Finally store the TM x TN patch with bounds checks.
}

// ------------------------------------------------------------ host wrappers
static void check_inputs(const torch::Tensor& A, const torch::Tensor& B) {
  for (const auto* t : {&A, &B}) {
    TORCH_CHECK(t->is_cuda() && t->is_contiguous() && t->scalar_type() == at::kFl
oat,
                "inputs must be contiguous float32 CUDA tensors");
    TORCH_CHECK(t->dim() == 2, "inputs must be 2-D");
  }
  TORCH_CHECK(A.size(1) == B.size(0), "inner dimensions must match");
}

torch::Tensor matmul_naive(torch::Tensor A, torch::Tensor B) {
  check_inputs(A, B);
  const int M = A.size(0), K = A.size(1), N = B.size(1);
  auto C = torch::empty({M, N}, A.options());
  if (M == 0 || N == 0) return C;
  dim3 block(32, 8);
  dim3 grid((N + block.x - 1) / block.x, (M + block.y - 1) / block.y);
  matmul_naive_kernel<<<grid, block, 0, at::cuda::getCurrentCUDAStream()>>>(
      A.data_ptr<float>(), B.data_ptr<float>(), C.data_ptr<float>(), M, N, K);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return C;
}

torch::Tensor matmul_smem(torch::Tensor A, torch::Tensor B) {
  check_inputs(A, B);
  const int M = A.size(0), K = A.size(1), N = B.size(1);
  auto C = torch::empty({M, N}, A.options());
  if (M == 0 || N == 0) return C;
  dim3 block(T, T);  // 1024 threads: the per-block maximum
  dim3 grid((N + T - 1) / T, (M + T - 1) / T);
  matmul_smem_kernel<<<grid, block, 0, at::cuda::getCurrentCUDAStream()>>>(
      A.data_ptr<float>(), B.data_ptr<float>(), C.data_ptr<float>(), M, N, K);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return C;
}
```

```
torch::Tensor matmul_regtile(torch::Tensor A, torch::Tensor B) {
  check_inputs(A, B);
  const int M = A.size(0), K = A.size(1), N = B.size(1);
  auto C = torch::empty({M, N}, A.options());
  if (M == 0 || N == 0) return C;
  dim3 grid((N + BN - 1) / BN, (M + BM - 1) / BM);
  matmul_regtile_kernel<<<grid, THREADS, 0, at::cuda::getCurrentCUDAStream()>>>(
      A.data_ptr<float>(), B.data_ptr<float>(), C.data_ptr<float>(), M, N, K);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return C;
}
```

验收

```
./check.sh c4
./check.sh c4 --bench       # TFLOP/s：naive、smem、regtile 与 cuBLAS（FP32，关闭 TF32
```

tests/test_c4_gemm.py 20 行

```
import pytest
import torch

from drill import load
from tests._cuda import requires_cuda

pytestmark = requires_cuda

SHAPES = [(1, 1, 1), (17, 33, 9), (64, 64, 64), (100, 70, 130), (128, 512, 256),
(513, 257, 1025)]


@pytest.mark.parametrize("fn", ["matmul_naive", "matmul_smem", "matmul_regtile"])
@pytest.mark.parametrize("mnk", SHAPES)
def test_matmul(fn, mnk):
    M, N, K = mnk
    m = load("c4_gemm")
    A = torch.randn(M, K, device="cuda")
    B = torch.randn(K, N, device="cuda")
    ref = (A.double() @ B.double()).float()    # fp64 reference: no TF32 surprise
s
    torch.testing.assert_close(getattr(m, fn)(A, B), ref, rtol=1e-3, atol=1e-3)
```

提示

L1 先只做 smem 版 方向（轻度剧透） 收起

- 内层先只想“一个 tile 步”：A 的 tile 是 `A[by·32 : +32, k0 : +32]`，B 的 tile 是 `B[k0 : +32, bx·32 : +32]`。
- 线程 (ty, tx) 正好装 A tile 的 `[ty][tx]` 和 B tile 的 `[ty][tx]`。
- 输出 `C[row][col] += Σ_k As[ty][k] · Bs[k][tx]`。

L2 regtile 的线程分工 拆解与常见坑（中度剧透） 收起

- 256 个线程排成 16×16：`tx = tid % 16` 管第 tx 组 4 列，`ty = tid / 16` 管第 ty 组 4 行。
- **加载**：A 的 tile 有 64×16 = 1024 个数，256 个线程每人搬 4 个：`for (e = tid; e < BM*BK; e += THREADS)`，`m = e / BK, k = e % BK`，存成 `As[k][m]`。B 的 tile 有 16×64 个数，`k = e / BN, n = e % BN`，存成 `Bs[k][n]`。e 连续时 k（或 n）连续，所以 global 读是合并的。
- **计算**：对每个 k，`a_frag[i] = As[k][ty*4 + i]`，`b_frag[j] = Bs[k][tx*4 + j]`，然后做 4×4 外积。
- **为什么 A 要转置存**：这样 `a_frag` 读的是 `As[k][...]` 里连续的 4 个数，而不是跨行跳着读。

L3 regtile 内层 几乎是答案（重度剧透） 收起

```
for (int k = 0; k < BK; ++k) {
  for (int i = 0; i < TM; ++i) a_frag[i] = As[k][ty * TM + i];
  for (int j = 0; j < TN; ++j) b_frag[j] = Bs[k][tx * TN + j];
  for (int i = 0; i < TM; ++i)
    for (int j = 0; j < TN; ++j) acc[i][j] += a_frag[i] * b_frag[j];
}
// 写回: gm = block_row + ty*TM + i, gn = block_col + tx*TN + j, 越界跳过
```

参考解

解 **solutions/c4_gemm.cu** 先自己写完、跑过测试再看 收起

solutions/c4_gemm.cu 167 行

```
// C4 · GEMM: naive -> shared-memory tiling -> register tiling — reference so
lution.
//
// C[M, N] = A[M, K] @ B[K, N], all row-major float32, fp32 accumulation.
//
// Arithmetic intensity is what changes between versions:
//   naive   : every multiply-add loads one A and one B value from global mem
ory (2 loads / FMA).
//   smem    : a 32x32 block loads a tile once into shared memory and reuses
each value 32 times.
//   regtile : each thread computes a 4x4 patch of C from registers; every va
lue read from
//             shared memory is used 4 times, so shared-memory traffic drops
by 4x again.
#include <torch/extension.h>
#include <ATen/cuda/CUDAContext.h>
#include <c10/cuda/CUDAException.h>

// ---------------------------------------------------------------- v0: naive
__global__ void matmul_naive_kernel(const float* __restrict__ A, const float*
__restrict__ B,
                                    float* __restrict__ C, int M, int N, int
K) {
  const int row = blockIdx.y * blockDim.y + threadIdx.y;
  const int col = blockIdx.x * blockDim.x + threadIdx.x;  // threadIdx.x -> c
olumn: B and C reads coalesce
  if (row >= M || col >= N) return;
  float acc = 0.0f;
  for (int k = 0; k < K; ++k) acc += A[(int64_t)row * K + k] * B[(int64_t)k *
N + col];
  C[(int64_t)row * N + col] = acc;
}

// ---------------------------------------------------------------- v1: share
d-memory tiling
constexpr int T = 32;  // block = T x T threads = one T x T tile of C

__global__ void matmul_smem_kernel(const float* __restrict__ A, const float*
__restrict__ B,
                                   float* __restrict__ C, int M, int N, int
K) {
  __shared__ float As[T][T];
```

```
  __shared__ float Bs[T][T];
  const int tx = threadIdx.x, ty = threadIdx.y;
  const int row = blockIdx.y * T + ty;
  const int col = blockIdx.x * T + tx;

  float acc = 0.0f;
  for (int k0 = 0; k0 < K; k0 += T) {
    // Every thread loads one element of each tile. Out-of-range elements bec
ome 0, which
    // contributes nothing to the dot product, so the inner loop needs no bou
nds checks.
    As[ty][tx] = (row < M && k0 + tx < K) ? A[(int64_t)row * K + k0 + tx] :
0.0f;
    Bs[ty][tx] = (k0 + ty < K && col < N) ? B[(int64_t)(k0 + ty) * N + col] :
0.0f;
    __syncthreads();  // tiles fully loaded before anyone uses them

#pragma unroll
    for (int k = 0; k < T; ++k) {
      // As[ty][k]: same address for the whole warp (same ty) -> broadcast, n
o conflict.
      // Bs[k][tx]: 32 consecutive floats -> 32 different banks, no conflict.
      acc += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();  // everyone done with this tile before the next iterati
on overwrites it
  }
  if (row < M && col < N) C[(int64_t)row * N + col] = acc;
}

// ------------------------------------------------------------- v2: regis
ter tiling
// Block tile BM x BN of C, K in steps of BK, each thread owns a TM x TN patc
h.
constexpr int BM = 64, BN = 64, BK = 16, TM = 4, TN = 4;
constexpr int THREADS = (BM / TM) * (BN / TN);  // 16 * 16 = 256

__global__ void __launch_bounds__(THREADS)
matmul_regtile_kernel(const float* __restrict__ A, const float* __restrict__
B,
                      float* __restrict__ C, int M, int N, int K) {
  // A's tile is stored transposed (As[k][m]) so the inner loop reads TM cons
ecutive m values
  // for a fixed k; B's tile keeps its natural layout Bs[k][n].
  __shared__ float As[BK][BM];
  __shared__ float Bs[BK][BN];

  const int tid = threadIdx.x;
  const int tx = tid % (BN / TN);  // which TN-wide column strip of the block
tile
```

```
  const int ty = tid / (BN / TN);  // which TM-tall row strip
  const int block_row = blockIdx.y * BM;
  const int block_col = blockIdx.x * BN;

  float acc[TM][TN] = {};  // the thread's TM x TN results live in registers
  float a_frag[TM], b_frag[TN];

  for (int k0 = 0; k0 < K; k0 += BK) {
    // Cooperative load of A[block_row : +BM, k0 : +BK] (BM*BK = 1024 floats,
4 per thread).
    // e -> (m, k) with k fastest, so consecutive threads read consecutive ad
dresses of A.
    for (int e = tid; e < BM * BK; e += THREADS) {
      const int m = e / BK, k = e % BK;
      const int gm = block_row + m, gk = k0 + k;
      As[k][m] = (gm < M && gk < K) ? A[(int64_t)gm * K + gk] : 0.0f;
    }
    // Cooperative load of B[k0 : +BK, block_col : +BN], n fastest -> coalesc
ed.
    for (int e = tid; e < BK * BN; e += THREADS) {
      const int k = e / BN, n = e % BN;
      const int gk = k0 + k, gn = block_col + n;
      Bs[k][n] = (gk < K && gn < N) ? B[(int64_t)gk * N + gn] : 0.0f;
    }
    __syncthreads();

#pragma unroll
    for (int k = 0; k < BK; ++k) {
      // Load a column of A's tile and a row of B's tile into registers ...
#pragma unroll
      for (int i = 0; i < TM; ++i) a_frag[i] = As[k][ty * TM + i];
#pragma unroll
      for (int j = 0; j < TN; ++j) b_frag[j] = Bs[k][tx * TN + j];
      // ... and take their outer product: TM*TN FMAs for TM+TN shared-memory
reads.
#pragma unroll
      for (int i = 0; i < TM; ++i)
#pragma unroll
        for (int j = 0; j < TN; ++j) acc[i][j] += a_frag[i] * b_frag[j];
    }
    __syncthreads();
  }

#pragma unroll
  for (int i = 0; i < TM; ++i) {
    const int gm = block_row + ty * TM + i;
    if (gm >= M) continue;
#pragma unroll
    for (int j = 0; j < TN; ++j) {
      const int gn = block_col + tx * TN + j;
```

```
      if (gn < N) C[(int64_t)gm * N + gn] = acc[i][j];
    }
  }
}

// ------------------------------------------------------------ host wrap
pers
static void check_inputs(const torch::Tensor& A, const torch::Tensor& B) {
  for (const auto* t : {&A, &B}) {
    TORCH_CHECK(t->is_cuda() && t->is_contiguous() && t->scalar_type() == a
t::kFloat,
                "inputs must be contiguous float32 CUDA tensors");
    TORCH_CHECK(t->dim() == 2, "inputs must be 2-D");
  }
  TORCH_CHECK(A.size(1) == B.size(0), "inner dimensions must match");
}

torch::Tensor matmul_naive(torch::Tensor A, torch::Tensor B) {
  check_inputs(A, B);
  const int M = A.size(0), K = A.size(1), N = B.size(1);
  auto C = torch::empty({M, N}, A.options());
  if (M == 0 || N == 0) return C;
  dim3 block(32, 8);
  dim3 grid((N + block.x - 1) / block.x, (M + block.y - 1) / block.y);
  matmul_naive_kernel<<<grid, block, 0, at::cuda::getCurrentCUDAStream()>>>(
      A.data_ptr<float>(), B.data_ptr<float>(), C.data_ptr<float>(), M, N,
K);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return C;
}

torch::Tensor matmul_smem(torch::Tensor A, torch::Tensor B) {
  check_inputs(A, B);
  const int M = A.size(0), K = A.size(1), N = B.size(1);
  auto C = torch::empty({M, N}, A.options());
  if (M == 0 || N == 0) return C;
  dim3 block(T, T);  // 1024 threads: the per-block maximum
  dim3 grid((N + T - 1) / T, (M + T - 1) / T);
  matmul_smem_kernel<<<grid, block, 0, at::cuda::getCurrentCUDAStream()>>>(
      A.data_ptr<float>(), B.data_ptr<float>(), C.data_ptr<float>(), M, N,
K);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return C;
}

torch::Tensor matmul_regtile(torch::Tensor A, torch::Tensor B) {
  check_inputs(A, B);
  const int M = A.size(0), K = A.size(1), N = B.size(1);
  auto C = torch::empty({M, N}, A.options());
  if (M == 0 || N == 0) return C;
```

```
  dim3 grid((N + BN - 1) / BN, (M + BM - 1) / BM);
  matmul_regtile_kernel<<<grid, THREADS, 0, at::cuda::getCurrentCUDAStream()>
>>(
      A.data_ptr<float>(), B.data_ptr<float>(), C.data_ptr<float>(), M, N,
K);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return C;
}
```

测完之后

RTX 5090 实测（FP32，关闭 TF32，单位 TFLOP/s）：

n	naive	smem	regtile	cuBLAS
512	5.2	5.2	**2.5**	16.3
2048	7.4	8.7	22.1	63.3
4096	7.4	9.1	**33.0**	65.4

- 大矩阵时 register tiling 比 smem 快 3.6 倍，达到 cuBLAS 的一半。
- **smem 只比 naive 快一点**：5090 的 L1 / L2 很大，naive 的大量重复读取其实命中了 cache。shared memory tiling 在 L2 较小的老 GPU 上收益更明显，register tiling 才是这里真正的提升。
- **512 时 regtile 反而最慢**：64×64 的 tile 只切出 64 个 block，5090 有 170 个 SM，大部分 SM 都闲着。这就是下面面试追问里说的 wave quantization / SM 利用率问题。
- 用 `ncu` 看 regtile 的**寄存器数**和 occupancy：TM、TN 改成 8 会怎样？更多复用，但寄存器压力更大，可能溢出到 local memory（`ncu` 里叫 `spilling`）。
- `As[k][m]` 的写入有 bank conflict（m 方向跨 64）。可以改成 `float4` 向量化加载，或给 As 加 padding，看看能提升多少。
- **双缓冲**：用两组 shared memory tile，计算当前 tile 时预取下一个 tile。这就是 Triton 里 `num_stages` 做的事。
- **Tensor Core**：用 `nvcuda::wmma`（`#include <mma.h>`）把 16×16×16 的 half 精度块交给硬件，FP16 下可以再快一个数量级。T4 里的 `tl.dot` 会自动用上 Tensor Core。

▼ 面试追问：tile 越大越好吗?

不是。tile 越大，每个数的复用度越高（算术强度 ∝ tile 边长），但一个 block 占用的 shared memory 和寄存器也越多，每个 SM 能驻留的 block 和 warp 就越少，访存延迟更难隐藏。矩阵本身较小时，大 tile 还会导致 block 数不足，喂不满所有 SM（wave quantization）。所以实际做法是对每个 shape 做 autotune，T4 就是这么做的。

Track T · Triton

block 级编程、mask、融合、reduction、tl.dot、autotune、FlashAttention。

核心 · Track T · Triton · 第 5 题

T1 Triton 入门：融合 add + GELU

用 Triton 重写 C1 的融合激活：同样的事，代码少一半，你开始用“一块数据”而不是“一个元素”思考。

25 min | 难度 易 | 你写 `mine/t1_add_gelu.py` | 验收 `./check.sh t1` | 参考解 `solutions/t1_add_gelu.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

和 C1 一样，这是一个 memory-bound 的融合 kernel：`gelu(x + y)`（残差加 + 激活）。它在这里的意义是**熟悉 Triton 的基本动作**：program_id、arange、mask、load、store、dtype 转换、grid。后面每一道 Triton 题都建立在这几个动作上。

源码与资料

`triton › 官方 tutorial 01-vector-add` · triton.language API · `flashinfer › FlashInfer 自带的 Triton activation` · `triton › interpreter.py`

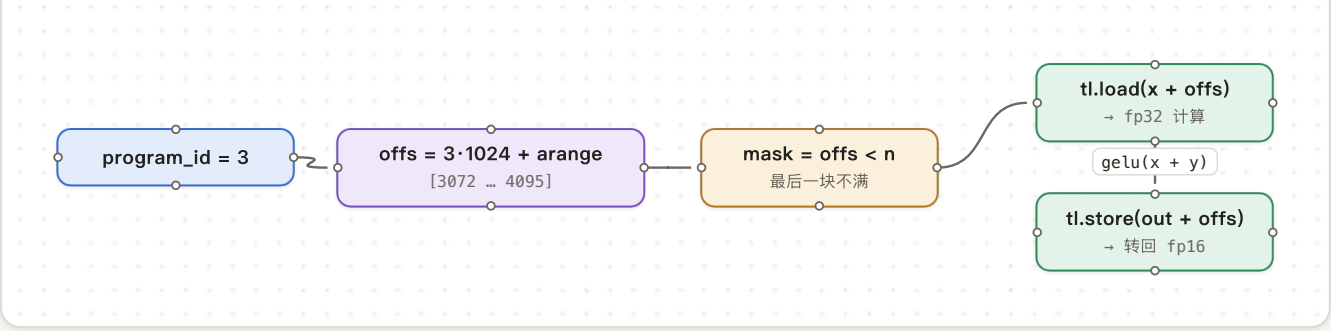
要用到的知识

- `pid = tl.program_id(axis=0)`：当前 program 的编号（对应 CUDA 的 blockIdx）。
- `offs = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)`：这个 program 负责的 BLOCK_SIZE 个下标（一个向量）。
- `mask = offs < n_elements`，`tl.load(ptr + offs, mask=mask, other=0.0)`，`tl.store(ptr + offs, val, mask=mask)`。
- `.to(tl.float32)` 和 `out_ptr.dtype.element_ty`：fp16 输入时先升到 fp32 计算，存回时转回原类型。
- `tl.sigmoid`；`tanh(u) = 2·sigmoid(2u) − 1`。

- launch：`kernel[grid](args..., BLOCK_SIZE=1024)`，`grid = (triton.cdiv(n, BLOCK_SIZE),)`。

图解与类比

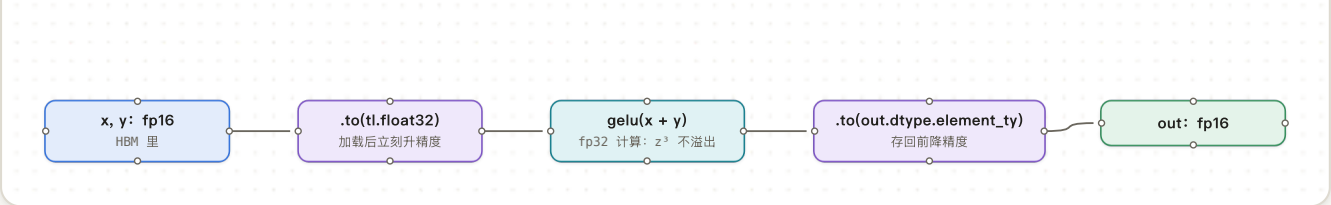
一个 program 做了什么



类比 · 工头领箱子

每个工头（program）领到一个箱号（program_id），箱子里有 1024 个格子（arange）。最后一箱可能只有半箱货：没货的格子贴“跳过”（mask），load 时给它填一个无害的默认值（`other`），store 时直接不写。

dtype 的一进一出



类比 · 用大锅炒，用小碗装

fp16 是小碗（最大约 65504，精度 3 位），fp32 是大锅。食材（数据）用小碗运进厨房省地方，但炒的时候（z^3 这种会变大的中间结果）要倒进大锅，否则会溢出；炒完再盛回小碗端出去。

题目要求

在 `mine/t1_add_gelu.py` 中：

1. 实现 `add_gelu_kernel`。

2. 实现 `add_gelu(x, y, block_size=1024)`：把输入转成连续张量，分配输出，计算 grid 并 launch。

要求：任意 shape（按扁平数组处理）；fp32 / fp16 / bf16；内部用 fp32 计算，输出 dtype 与输入相同；非连续输入（如转置后的 view）也要正确。

题干

mine/t1_add_gelu.py 21 行

```
"""T1 · Fused add + GELU in Triton — YOUR IMPLEMENTATION.

out = gelu_tanh(x + y), any shape, fp32 / fp16 / bf16; compute in fp32, store in
x.dtype.
gelu_tanh(z) = 0.5 * z * (1 + tanh(0.7978845608028654 * (z + 0.044715 * z^3)))
Hint for tanh: tanh(u) = 2 * sigmoid(2u) - 1  (tl.sigmoid).
"""
import torch
import triton
import triton.language as tl


@triton.jit
def add_gelu_kernel(x_ptr, y_ptr, out_ptr, n_elements, BLOCK_SIZE: tl.constexpr):
    # TODO: pid -> offsets -> mask -> load x, y (as fp32) -> gelu(x + y) -> store
(cast to out dtype)
    pass


def add_gelu(x: torch.Tensor, y: torch.Tensor, block_size: int = 1024) -> torch.T
ensor:
    assert x.shape == y.shape and x.dtype == y.dtype and x.device == y.device
    # TODO: make inputs contiguous, allocate out, compute the grid, launch add_ge
lu_kernel
    raise NotImplementedError
```

验收

```
./check.sh t1               # 没有 GPU 也能跑（解释器模式）
./check.sh t1 --bench       # GPU：GB/s，vs PyTorch 两个 kernel
```

tests/test_t1_add_gelu.py 32 行

```
import pytest
import torch
```

```python
import torch.nn.functional as F

from common.env import DEVICE
from drill import load


@pytest.mark.parametrize("n", [1, 1023, 1024, 1025, 100_003])
def test_add_gelu_fp32(n):
    m = load("t1_add_gelu")
    x = torch.randn(n, device=DEVICE) * 3
    y = torch.randn(n, device=DEVICE)
    torch.testing.assert_close(m.add_gelu(x, y), F.gelu(x + y, approximate="tanh"), rtol=1e-5, atol=1e-5)


@pytest.mark.parametrize("dtype", [torch.float16, torch.bfloat16])
def test_add_gelu_half(dtype):
    m = load("t1_add_gelu")
    x = torch.randn(7, 333, device=DEVICE).to(dtype)
    y = torch.randn(7, 333, device=DEVICE).to(dtype)
    out = m.add_gelu(x, y)
    assert out.dtype == dtype and out.shape == x.shape
    ref = F.gelu(x.float() + y.float(), approximate="tanh").to(dtype)
    torch.testing.assert_close(out, ref, rtol=2e-2, atol=2e-2)


def test_add_gelu_noncontiguous_input():
    m = load("t1_add_gelu")
    x = torch.randn(64, 48, device=DEVICE).t()          # a transposed view
    y = torch.randn(48, 64, device=DEVICE)
    torch.testing.assert_close(m.add_gelu(x, y), F.gelu(x + y, approximate="tanh"), rtol=1e-5, atol=1e-5)
```

提示

L1 **结构** 方向（轻度剧透） 收起 ▾

这个 kernel 只有 5 步：算 offs → 算 mask → 两次 load（转 fp32）→ 计算 → store（转回输出类型）。wrapper 也是：`contiguous()` → `empty_like` → grid → launch。

L2 常见坑 拆解与常见坑（中度剧透） 收起

- 忘了 `mask`：最后一个 program 会读写越界。解释器模式下可能直接报错，GPU 上可能悄悄写坏别的内存。
- 没有转 fp32：fp16 下 z^3 在 $|z| > 40$ 左右就会溢出成 inf。
- 存储时没转回：fp32 的值写进 fp16 指针会报类型错误，或者在某些版本上被隐式截断。
- 非连续输入：kernel 把指针当扁平数组用，`x.t()` 这类 view 直接传进去会读错位置，所以 wrapper 里要先 `.contiguous()`。

L3 kernel 主体 几乎是答案（重度剧透） 收起

```
pid = tl.program_id(axis=0)
offs = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
mask = offs < n_elements
x = tl.load(x_ptr + offs, mask=mask, other=0.0).to(tl.float32)
y = tl.load(y_ptr + offs, mask=mask, other=0.0).to(tl.float32)
z = x + y
t = 2.0 * tl.sigmoid(2.0 * 0.7978845608028654 * (z + 0.044715 * z * z * z)) - 1
tl.store(out_ptr + offs, (0.5 * z * (1.0 + t)).to(out_ptr.dtype.element_ty), ma
```

参考解

解 **solutions/t1_add_gelu.py** 先自己写完、跑过测试再看 收起

solutions/t1_add_gelu.py 49 行

```
"""T1 · Fused add + GELU in Triton — reference solution.

out = gelu_tanh(x + y), any shape, fp32 / fp16 / bf16. Math in fp32, result i
n x.dtype.

Triton's programming model in one kernel:
  * a "program" (≈ a CUDA thread block) handles BLOCK_SIZE consecutive elemen
ts;
  * tl.arange gives the program a *vector* of offsets — you write block-level
code,
    the compiler spreads the vector across the block's threads;
  * every load/store past the end is disabled with a mask.
"""
import torch
import triton
import triton.language as tl


@triton.jit
def add_gelu_kernel(x_ptr, y_ptr, out_ptr, n_elements, BLOCK_SIZE: tl.constex
pr):
    pid = tl.program_id(axis=0)                       # which block of elemen
ts this program owns
    offs = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)  # [BLOCK_SIZE] elemen
t offsets
    mask = offs < n_elements                          # the last program is u
sually partial

    # Load, upcast to fp32 (fp16 x^3 overflows/loses precision quickly).
    x = tl.load(x_ptr + offs, mask=mask, other=0.0).to(tl.float32)
    y = tl.load(y_ptr + offs, mask=mask, other=0.0).to(tl.float32)
    z = x + y

    # GELU (tanh approximation): 0.5 z (1 + tanh(sqrt(2/pi) (z + 0.044715 z^
3))).
    # tanh(u) = 2 sigmoid(2u) - 1; tl.sigmoid is available on every backend.
    inner = 0.7978845608028654 * (z + 0.044715 * z * z * z)
    t = 2.0 * tl.sigmoid(2.0 * inner) - 1.0
    out = 0.5 * z * (1.0 + t)

    # Cast back to the output's element type before storing.
    tl.store(out_ptr + offs, out.to(out_ptr.dtype.element_ty), mask=mask)
```

```python
def add_gelu(x: torch.Tensor, y: torch.Tensor, block_size: int = 1024) -> tor
ch.Tensor:
    assert x.shape == y.shape and x.dtype == y.dtype and x.device == y.device
    # The kernel indexes memory as a flat array, so it needs contiguous input
s.
    x, y = x.contiguous(), y.contiguous()
    out = torch.empty_like(x)
    n = x.numel()
    if n == 0:
        return out
    # One program per BLOCK_SIZE elements. grid can also be a lambda over the
meta-params:
    #   grid = lambda meta: (triton.cdiv(n, meta["BLOCK_SIZE"]),)
    grid = (triton.cdiv(n, block_size),)
    add_gelu_kernel[grid](x, y, out, n, BLOCK_SIZE=block_size)
    return out
```

测完之后

- RTX 5090 实测（fp16，n = 2^{26}）：PyTorch 两个 kernel 434 us · 927 GB/s，参考解 **260 us · 1548 GB/s**。和 C1 的 CUDA 版一样到了带宽上限：编译器自动做了向量化，每个线程一次读 16 字节（fp16 就是 8 个元素）。
- 试试 `block_size` 取 128、256、…、8192，以及 `num_warps=2/4/8`，结果有什么变化？
- 看编译产物：`k = add_gelu_kernel[grid](...)`，然后 `print(k.asm["ttgir"])`，找到 `#blocked = #ttg.blocked<{sizePerThread = [...], threadsPerWarp = [32], warpsPerCTA = [...]}>`。这就是 G1 要讲的 layout。

▼ 面试追问：Triton 的 program 和 CUDA 的 block 是什么关系？

一个 program 在 GPU 上就是一个 CUDA block（CTA），由 `num_warps × 32` 个线程组成。你写的块级操作（比如长度为 BLOCK 的向量加法）会被编译器按 layout 拆给这些线程，每个线程负责若干个连续元素。因此 Triton 里没有 threadIdx：线程这一层由编译器管理，你管理的是 program 这一层的数据分块。

核心 · Track T · Triton · 第 6 题

T2 融合 row softmax：数值稳定 + 一次读写

一行一个 program，整行放进一个 block：数值稳定的 softmax 只读一次、写一次。这是“融合 reduction”的标准写法。

35 min | 难度 中 | 你写 `mine/t2_softmax.py` | 验收 `./check.sh t2` | 参考解 `solutions/t2_softmax.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

softmax 出现在 attention、MoE 路由、采样（logits → 概率）里。PyTorch 的通用实现要把同一行读好几次（求 max、求 exp 的和、归一化）。行不太长（比如词表维度之外的大多数情况，N ⩽ 几千）时，整行可以放进一个 program 的寄存器里：**读一次 → 在片上完成 max / exp / sum / 除法 → 写一次。**

源码与资料

`triton › 官方 tutorial 02-fused-softmax` · `pytorch › PyTorch SoftMax.cu` · `cutlass › CuTe DSL softmax notebook` · Online normalizer calculation for softmax (Milakov & Gimelshein)

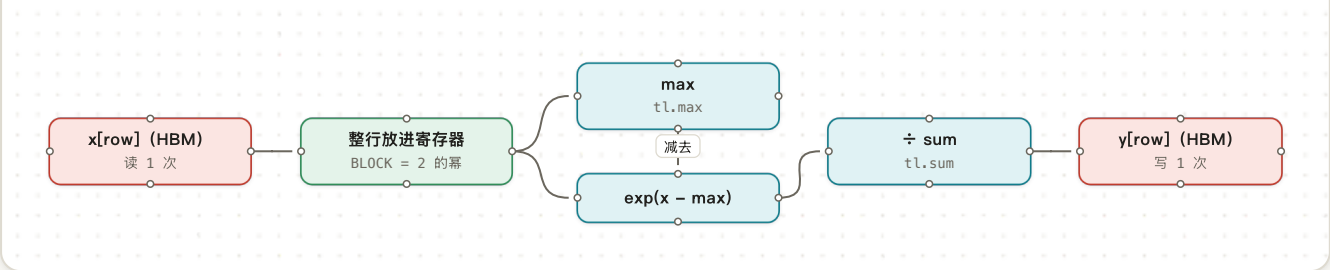
要用到的知识

- **数值稳定**：`softmax(x) = softmax(x - max(x))`。减去最大值后最大的指数是 $e^0 = 1$，不会溢出。不减的话 x = 100 时 fp32 的 $e^{100} \approx 2.7e43$ 已经接近上限，fp16 在 x > 11 时 e^x 就溢出了。
- **行内归约**：`tl.max(x, axis=0)`、`tl.sum(x, axis=0)` 作用在整个向量上，返回标量。
- **mask 的填充值要服务于运算**：做 max 时，padding 位置应该是 −inf，这样它不会成为最大值，exp(−inf) = 0 也不会计入 sum。

- **stride**：行与行之间的距离是 `x.stride(0)`，切片后可能不等于 N。
- `BLOCK_SIZE = triton.next_power_of_2(N)`；行越长，`num_warps` 越大。
- `tl.program_id(0).to(tl.int64)`：大张量下 `row * stride` 可能超过 int32。

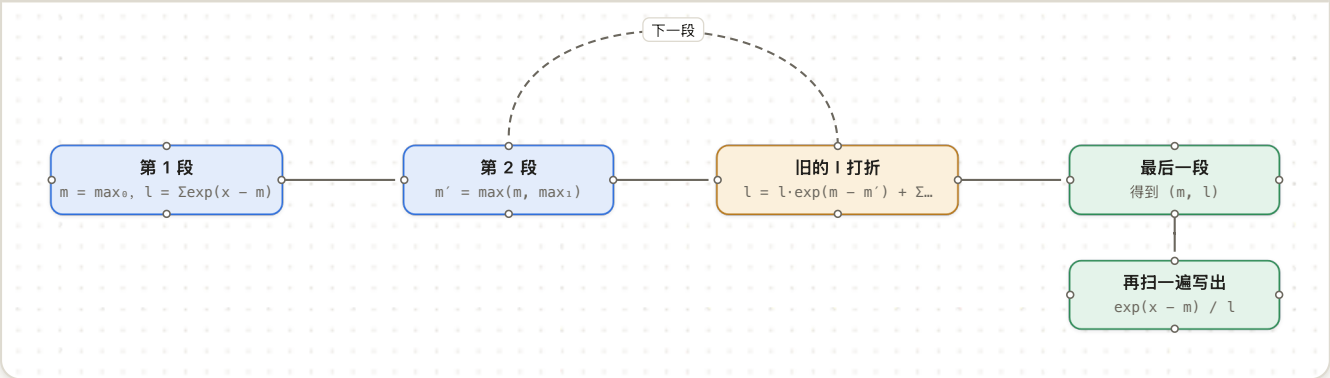
图解与类比

一行一个 program：全部在片上完成



> **类比 · 以最高的人为基准量身高**
>
> 直接算 e^x 就像用海拔量身高：数字大得离谱（溢出）。先减去最大值，相当于以全班最高的人为 0 点，大家的“相对身高”都 ⩽ 0，e 的幂都在 (0, 1] 之间，怎么都不会溢出；最后除以总和，就是按比例分蛋糕。 padding 位置填 −∞，等于给不存在的同学记一个“无限矮”，他既不会成为最高的人，分到的蛋糕也是 0。

行太长时：online softmax 分段扫描（T5 的预告）



> **类比 · 换汇率**
>
> 你在不同国家记账，一开始用美元当基准（当前最大值 m），后来发现日元更大（出现了更大的 m′），就要把之前所有账目按新汇率折算一次：旧的总和乘以 exp(m − m′)。折算只需要一次乘法，不需要回去重新翻每一笔账。

题目要求

`mine/t2_softmax.py` 的 wrapper 已经写好（一行一个 program，`grid = (M,)`）。实现 `softmax_kernel`：

- 输入 `[M, N]`（行可以有 stride，列连续），输出连续的 `[M, N]`，dtype 与输入相同。
- 内部用 fp32 计算。
- 对很大的输入值（±1e4）必须给出有限且正确的结果。

题干

mine/t2_softmax.py 28 行

```
"""T2 · Fused row softmax in Triton — YOUR IMPLEMENTATION.

softmax over the last dim of x: [M, N]; one program per row, whole row in one blo
ck.
"""
import torch
import triton
import triton.language as tl


@triton.jit
def softmax_kernel(out_ptr, in_ptr, in_row_stride, out_row_stride, n_cols, BLOCK_
SIZE: tl.constexpr):
    # TODO: row = program_id (as int64); cols = arange(0, BLOCK_SIZE); mask;
    #       load with other=-inf, subtract max, exp, divide by sum, store
    pass


def softmax(x: torch.Tensor) -> torch.Tensor:
    assert x.dim() == 2, "expects [M, N]"
    M, N = x.shape
    if x.stride(1) != 1:
        x = x.contiguous()
    y = torch.empty((M, N), dtype=x.dtype, device=x.device)
    if M == 0 or N == 0:
        return y
    BLOCK_SIZE = triton.next_power_of_2(N)
    num_warps = 4 if BLOCK_SIZE <= 2048 else (8 if BLOCK_SIZE <= 8192 else 16)
    softmax_kernel[(M,)](y, x, x.stride(0), y.stride(0), N, BLOCK_SIZE=BLOCK_SIZ
E, num_warps=num_warps)
    return y
```

验收

```
./check.sh t2
./check.sh t2 --bench       # N = 256 … 16384 的 GB/s, vs torch.softmax
```

tests/test_t2_softmax.py 29 行

```
import pytest
import torch

from common.env import DEVICE
from drill import load


@pytest.mark.parametrize("shape", [(1, 1), (4, 7), (33, 128), (17, 1000), (8, 4097)])
def test_softmax_fp32(shape):
    m = load("t2_softmax")
    x = torch.randn(shape, device=DEVICE) * 5
    torch.testing.assert_close(m.softmax(x), torch.softmax(x, dim=-1), rtol=1e-5, atol=1e-6)


def test_softmax_large_values_are_stable():
    m = load("t2_softmax")
    x = torch.tensor([[1000.0, 1001.0, 1002.0], [-1e4, 0.0, 1e4]], device=DEVICE)
    out = m.softmax(x)
    assert torch.isfinite(out).all()
    torch.testing.assert_close(out, torch.softmax(x, dim=-1))


def test_softmax_fp16_and_row_stride():
    m = load("t2_softmax")
    big = torch.randn(10, 300, device=DEVICE).half()
    x = big[:, :257]                                    # rows are strided (stride 300)
    out = m.softmax(x)
    assert out.dtype == torch.float16
    torch.testing.assert_close(out.float(), torch.softmax(x.float(), dim=-1), rtol=1e-2, atol=1e-3)
```

提示

L1 五行 方向（轻度剧透） 收起

row 起点 → load（other=−inf，转 fp32）→ 减 max → exp → 除以 sum → store。

L2 常见坑 拆解与常见坑（中度剧透） 收起

- `other=0.0` 而不是 `-inf`：某一行全为负数时，padding 的 0 会成为 max，而且 exp(0 − max) 也会被算进分母。
- 用 `n_cols` 代替 stride 来定位行：切片得到的输入（stride 300、N 257）会读错。
- 输出 stride 与输入 stride 混用：输出是新分配的连续张量，用 `out_row_stride`。

L3 kernel 主体 几乎是答案（重度剧透） 收起

```
row = tl.program_id(0).to(tl.int64)
cols = tl.arange(0, BLOCK_SIZE)
mask = cols < n_cols
x = tl.load(in_ptr + row * in_row_stride + cols, mask=mask, other=-float("inf")
x = x - tl.max(x, axis=0)
num = tl.exp(x)
y = num / tl.sum(num, axis=0)
tl.store(out_ptr + row * out_row_stride + cols, y.to(out_ptr.dtype.element_ty),
```

参考解

解 **solutions/t2_softmax.py** 先自己写完、跑过测试再看 收起

solutions/t2_softmax.py 46 行

```
"""T2 · Fused row softmax in Triton — reference solution.

softmax(x) over the last dim of a 2-D tensor [M, N]. One program per row; the whole row
fits in one block (BLOCK_SIZE = next_power_of_2(N)), so x is read once and y written once.
PyTorch's unfused version reads x for max, again for exp/sum, again to normalise.
"""
import torch
import triton
import triton.language as tl


@triton.jit
def softmax_kernel(out_ptr, in_ptr, in_row_stride, out_row_stride, n_cols, BLOCK_SIZE: tl.constexpr):
    # program_id is int32; promote before multiplying by a stride so row * stride cannot
    # overflow for tensors with more than 2^31 elements.
    row = tl.program_id(0).to(tl.int64)
    cols = tl.arange(0, BLOCK_SIZE)         # BLOCK_SIZE is a power of two >= n_cols
    mask = cols < n_cols

    # Padding lanes load -inf: they lose the max, and exp(-inf) = 0 adds nothing to the sum.
    x = tl.load(in_ptr + row * in_row_stride + cols, mask=mask, other=-float("inf")).to(tl.float32)

    # Numerically stable softmax: softmax(x) = softmax(x - max(x)); the largest exponent is 0,
    # so exp never overflows even for fp16 inputs around 1e4.
    x = x - tl.max(x, axis=0)
    num = tl.exp(x)
    den = tl.sum(num, axis=0)
    y = num / den

    tl.store(out_ptr + row * out_row_stride + cols, y.to(out_ptr.dtype.element_ty), mask=mask)
```

```python
def softmax(x: torch.Tensor) -> torch.Tensor:
    assert x.dim() == 2, "expects [M, N]"
    M, N = x.shape
    # The kernel walks a row with unit stride; the rows themselves may be str
ided.
    if x.stride(1) != 1:
        x = x.contiguous()
    y = torch.empty((M, N), dtype=x.dtype, device=x.device)
    if M == 0 or N == 0:
        return y
    BLOCK_SIZE = triton.next_power_of_2(N)
    # More warps for wider rows, so each thread holds a reasonable number of
elements.
    num_warps = 4 if BLOCK_SIZE <= 2048 else (8 if BLOCK_SIZE <= 8192 else 1
6)
    softmax_kernel[(M,)](y, x, x.stride(0), y.stride(0), N, BLOCK_SIZE=BLOCK_
SIZE, num_warps=num_warps)
    return y
```

测完之后

- 有效带宽按 `2 × M × N × 2 字节`（fp16 读一次、写一次）计算。RTX 5090 上 M = 4096 的实测：

N	torch.softmax	参考解
256	6.8 us	8.2 us（更慢）
1024	16.4 us	**11.9 us**
4096	65.5 us · 1025 GB/s	**43.6 us · 1540 GB/s**
16384	180 us	178 us

N = 256 时每行只有 512 字节，一行一个 program 太碎，调度开销占主导（见下面“一个 program 处理多行”）。

- N = 16384 时，一行 16K 个 fp32 值要放进寄存器，每个线程要持有很多值。观察寄存器溢出和性能下降。
- **超长行怎么办**：把行分成若干段，用 **online softmax** 在一次遍历中维护 `(m, l)`：遇到新的一段，`m_new = max(m, max(段))`，`l = l·exp(m − m_new) + Σexp(段 − m_new)`。T5 的 FlashAttention 用的就是这个公式。
- 一个 program 处理多行（grid 只开 SM 数 × k 个 program，用 `tl.range` 循环），可以减少短行时的调度开销：这是官方 tutorial 的进阶写法。

▼ 面试追问：fused softmax 为什么比 PyTorch 快？快的上限在哪？

softmax 是 memory-bound 的，时间 ≈ 搬运字节数 / 带宽。融合后每个元素只读写各一次，这就是下限，所以上限就是“读一次 + 写一次”在峰值带宽下所需的时间。行变得非常长、一个 block 放不下时，要么分两遍（多读一次），要么用 online softmax 在一次遍历中完成。

核心 · Track T · Triton · 第 7 题

T3 LayerNorm / RMSNorm forward

每个 Transformer 层至少两次 norm：一行之内做两个归约（mean、var），再加一个仿射变换，全部融合在一个 kernel 里。

40 min | 难度 中 | 你写 `mine/t3_layernorm.py` | 验收 `./check.sh t3` | 参考解 `solutions/t3_layernorm.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

LayerNorm（GPT-2、BERT）和 RMSNorm（LLaMA、Qwen、DeepSeek）在每一层出现两次。它们和 softmax 一样是 memory-bound 的逐行运算：一行一个 program，读一次、写一次。forward 还要把每行的 `mean` 和 `rstd`（$1/\sigma$）存下来，backward 要用。

- LayerNorm：`y = (x - μ) / sqrt(σ² + ε) · w + b`，σ^2 是**有偏**方差（除以 N）。
- RMSNorm：`y = x / sqrt(mean(x²) + ε) · w`，不减均值、没有 bias，少一次归约。

源码与资料

`triton` › 官方 `tutorial 05-layer-norm` · `flashinfer` › `FlashInfer Triton norm` · `flashinfer` › `FlashInfer RMSNormKernel (CUDA)` · `vllm` › `vLLM layernorm_kernels.cu` · `pytorch` › `PyTorch layer_norm_kernel.cu` · RMSNorm 论文

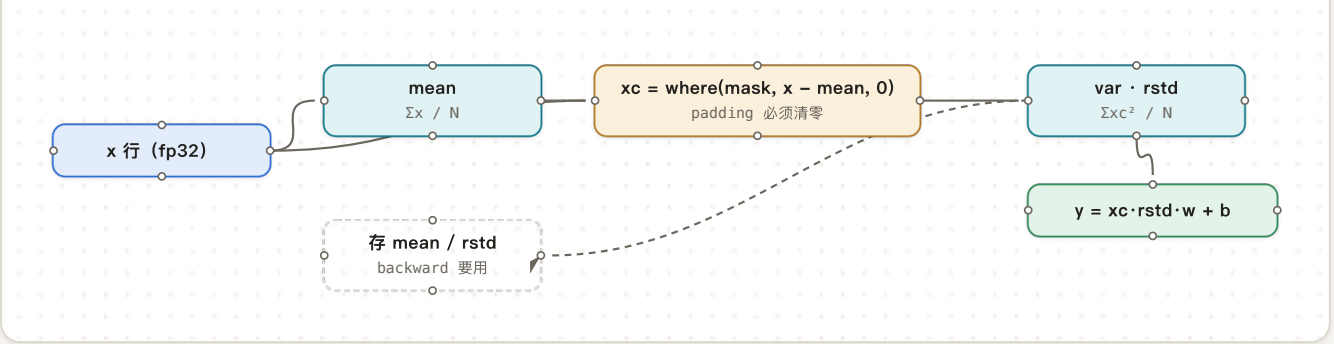
要用到的知识

- 同一行内的**两次归约**：先求 μ，再求 `Σ(x - μ)²`。两遍法比 `E[x²] - E[x]²` 数值更稳定：后者在均值很大时会发生灾难性抵消。
- **mask 之后的值要再检查**：padding 处 load 的是 0，求和时没问题；但 `x - μ` 在 padding 处等于 $-\mu$，不是 0，平方之前要用 `tl.where(mask, x - μ, 0)` 清零。
- 标量 store：`tl.store(mean_ptr + row, mean)`。

- `1.0 / tl.sqrt(v)` 或 `tl.rsqrt(v)`。

图解与类比

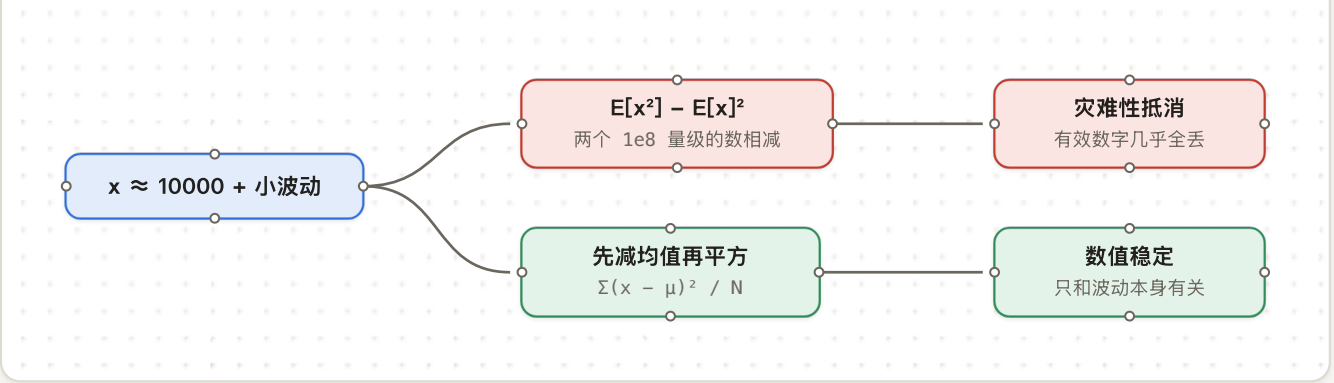
LayerNorm 的一行



类比 · 成绩标准化

先算全班平均分，每个人减去平均分，再除以标准差。缺考的同学（padding）在求和时按 0 分算没问题，但“减去平均分”之后他会变成 −平均分，平方后就把方差拉大了：必须在平方之前把他重新清零。

为什么用两遍法求方差



类比 · 称体重差

想知道两个人体重差多少，最差的办法是各自连同一辆 10 吨卡车一起称，再相减：秤的误差比他们的体重差还大（E[x²] − E[x]² 就是这样）。两遍法是先把卡车开走（减去均值），再称剩下的差别。

题目要求

wrapper 已写好（多维输入会被 reshape 成 `[M, N]`；N ⩽ 16384）。实现两个 kernel：

1. `layer_norm_kernel`：输出 y，并写 `mean[M]`、`rstd[M]`（fp32）。
2. `rms_norm_kernel`：输出 y。

要求在 fp32 下与 `F.layer_norm` 的误差 ⩽ 1e-4，fp16 输入时输出 fp16，内部用 fp32 计算。

题干

mine/t3_layernorm.py 59 行

```
"""T3 · LayerNorm / RMSNorm forward in Triton — YOUR IMPLEMENTATION.

layer_norm(x[M, N], w, b, eps) -> (y, mean[M], rstd[M]); rms_norm(x[M, N], w, ep
s) -> y.
The wrappers are done; implement the two kernels.
"""
import torch
import triton
import triton.language as tl


@triton.jit
def layer_norm_kernel(x_ptr, w_ptr, b_ptr, y_ptr, mean_ptr, rstd_ptr,
                      x_row_stride, y_row_stride, N, eps, BLOCK_N: tl.constexpr):
    # TODO: row, cols, mask; load x as fp32; mean; xc = where(mask, x - mean, 0);
    # var (biased); rstd = 1 / sqrt(var + eps); y = xc * rstd * w + b; store y, m
ean, rstd
    pass


@triton.jit
def rms_norm_kernel(x_ptr, w_ptr, y_ptr, x_row_stride, y_row_stride, N, eps, BLOC
K_N: tl.constexpr):
    # TODO: rstd = 1 / sqrt(mean(x^2) + eps); y = x * rstd * w
    pass


def _launch_params(N: int) -> tuple[int, int]:
    BLOCK_N = triton.next_power_of_2(N)
    if BLOCK_N > 16384:
        raise ValueError("this single-block kernel supports N <= 16384; loop over
the row for more")
    num_warps = min(max(BLOCK_N // 256, 1), 8)   # ~256 elements per warp, 1..8 w
```

```
arps
    return BLOCK_N, num_warps


def layer_norm(x: torch.Tensor, w: torch.Tensor, b: torch.Tensor, eps: float = 1e
-5):
    shape = x.shape
    x2 = x.reshape(-1, shape[-1])                # [..., N] -> [M, N]
    if x2.stride(-1) != 1:
        x2 = x2.contiguous()
    M, N = x2.shape
    y = torch.empty_like(x2)
    mean = torch.empty(M, dtype=torch.float32, device=x.device)
    rstd = torch.empty(M, dtype=torch.float32, device=x.device)
    if M > 0:
        BLOCK_N, num_warps = _launch_params(N)
        layer_norm_kernel[(M,)](x2, w, b, y, mean, rstd, x2.stride(0), y.stride
(0), N, eps,
                                BLOCK_N=BLOCK_N, num_warps=num_warps)
    return y.reshape(shape), mean, rstd


def rms_norm(x: torch.Tensor, w: torch.Tensor, eps: float = 1e-6) -> torch.Tenso
r:
    shape = x.shape
    x2 = x.reshape(-1, shape[-1])
    if x2.stride(-1) != 1:
        x2 = x2.contiguous()
    M, N = x2.shape
    y = torch.empty_like(x2)
    if M > 0:
        BLOCK_N, num_warps = _launch_params(N)
        rms_norm_kernel[(M,)](x2, w, y, x2.stride(0), y.stride(0), N, eps, BLOCK_
N=BLOCK_N, num_warps=num_warps)
    return y.reshape(shape)
```

验收

```
./check.sh t3
./check.sh t3 --bench
```

tests/test_t3_layernorm.py 40 行

```
import pytest
import torch
import torch.nn.functional as F
```

```
from common.env import DEVICE
from drill import load


@pytest.mark.parametrize("shape", [(1, 8), (5, 7), (64, 768), (3, 4, 1000)])
def test_layer_norm(shape):
    m = load("t3_layernorm")
    N = shape[-1]
    x = torch.randn(shape, device=DEVICE) * 2 + 1
    w = torch.randn(N, device=DEVICE)
    b = torch.randn(N, device=DEVICE)
    y, mean, rstd = m.layer_norm(x, w, b, 1e-5)
    torch.testing.assert_close(y, F.layer_norm(x, (N,), w, b, 1e-5), rtol=1e-4, a
tol=1e-4)
    x2 = x.reshape(-1, N)
    torch.testing.assert_close(mean, x2.mean(-1), rtol=1e-5, atol=1e-5)
    torch.testing.assert_close(rstd, torch.rsqrt(x2.var(-1, unbiased=False) + 1e-
5), rtol=1e-4, atol=1e-4)


def test_layer_norm_fp16():
    m = load("t3_layernorm")
    x = torch.randn(16, 512, device=DEVICE).half()
    w = torch.ones(512, device=DEVICE).half()
    b = torch.zeros(512, device=DEVICE).half()
    y, _, _ = m.layer_norm(x, w, b, 1e-5)
    assert y.dtype == torch.float16
    ref = F.layer_norm(x.float(), (512,), w.float(), b.float(), 1e-5)
    torch.testing.assert_close(y.float(), ref, rtol=1e-2, atol=1e-2)


@pytest.mark.parametrize("shape", [(1, 16), (7, 333), (32, 4096)])
def test_rms_norm(shape):
    m = load("t3_layernorm")
    x = torch.randn(shape, device=DEVICE)
    w = torch.randn(shape[-1], device=DEVICE)
    ref = x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + 1e-6) * w
    torch.testing.assert_close(m.rms_norm(x, w, 1e-6), ref, rtol=1e-4, atol=1e-4)
```

提示

L1 顺序 方向（轻度剧透） 收起

load x（fp32）→ `mean = sum(x) / N` → `xc = where(mask, x − mean, 0)` → `var = sum(xc²) / N` → `rstd` → load w、b → `y = xc · rstd · w + b` → 存 y、mean、rstd。

L2 常见坑 拆解与常见坑（中度剧透） 收起

- 平方前没清零 padding：N 不是 2 的幂时（7、333、1000），方差会偏大。
- 除以 `BLOCK_N` 而不是 `N`。
- 用了无偏方差（除以 N − 1）：`F.layer_norm` 用的是有偏的。
- mean / rstd 存成了 fp16：wrapper 分配的是 fp32 张量。

L3 LayerNorm 主体 几乎是答案（重度剧透） 收起

```
x = tl.load(x_ptr + row * x_row_stride + cols, mask=mask, other=0.0).to(tl.floa
mean = tl.sum(x, axis=0) / N
xc = tl.where(mask, x - mean, 0.0)
rstd = 1.0 / tl.sqrt(tl.sum(xc * xc, axis=0) / N + eps)
w = tl.load(w_ptr + cols, mask=mask, other=0.0).to(tl.float32)
b = tl.load(b_ptr + cols, mask=mask, other=0.0).to(tl.float32)
tl.store(y_ptr + row * y_row_stride + cols, (xc * rstd * w + b).to(y_ptr.dtype.
tl.store(mean_ptr + row, mean)
tl.store(rstd_ptr + row, rstd)
```

参考解

解 **solutions/t3_layernorm.py** 先自己写完、跑过测试再看 收起

solutions/t3_layernorm.py 86 行

```
"""T3 · LayerNorm / RMSNorm forward in Triton — reference solution.

layer_norm(x[M, N], w, b, eps) -> (y, mean[M], rstd[M])
    y = (x - mean) * rstd * w + b,  rstd = 1 / sqrt(var + eps),  var is the b
iased variance.
rms_norm(x[M, N], w, eps) -> y
    y = x * rsqrt(mean(x^2) + eps) * w

One program per row, the full row in one block (N <= 16384). mean and rstd ar
e returned
because the backward pass needs exactly these two numbers per row.
"""
import torch
import triton
import triton.language as tl


@triton.jit
def layer_norm_kernel(x_ptr, w_ptr, b_ptr, y_ptr, mean_ptr, rstd_ptr,
                      x_row_stride, y_row_stride, N, eps, BLOCK_N: tl.constex
pr):
    row = tl.program_id(0).to(tl.int64)
    cols = tl.arange(0, BLOCK_N)
    mask = cols < N

    x = tl.load(x_ptr + row * x_row_stride + cols, mask=mask, other=0.0).to(t
l.float32)

    # Statistics in fp32. Padding lanes loaded 0, so they do not disturb the
sum ...
    mean = tl.sum(x, axis=0) / N
    # ... but x - mean is NOT zero there: zero it again before squaring.
    xc = tl.where(mask, x - mean, 0.0)
    var = tl.sum(xc * xc, axis=0) / N          # two-pass variance: stabler t
han E[x^2] - E[x]^2
    rstd = 1.0 / tl.sqrt(var + eps)

    w = tl.load(w_ptr + cols, mask=mask, other=0.0).to(tl.float32)
    b = tl.load(b_ptr + cols, mask=mask, other=0.0).to(tl.float32)
    y = xc * rstd * w + b

    tl.store(y_ptr + row * y_row_stride + cols, y.to(y_ptr.dtype.element_ty),
```

```
mask=mask)
    tl.store(mean_ptr + row, mean)             # scalar stores: one value per
program
    tl.store(rstd_ptr + row, rstd)


@triton.jit
def rms_norm_kernel(x_ptr, w_ptr, y_ptr, x_row_stride, y_row_stride, N, eps,
BLOCK_N: tl.constexpr):
    row = tl.program_id(0).to(tl.int64)
    cols = tl.arange(0, BLOCK_N)
    mask = cols < N
    x = tl.load(x_ptr + row * x_row_stride + cols, mask=mask, other=0.0).to(t
l.float32)
    rstd = 1.0 / tl.sqrt(tl.sum(x * x, axis=0) / N + eps)   # no mean subtrac
tion, no bias
    w = tl.load(w_ptr + cols, mask=mask, other=0.0).to(tl.float32)
    tl.store(y_ptr + row * y_row_stride + cols, (x * rstd * w).to(y_ptr.dtyp
e.element_ty), mask=mask)


def _launch_params(N: int) -> tuple[int, int]:
    BLOCK_N = triton.next_power_of_2(N)
    if BLOCK_N > 16384:
        raise ValueError("this single-block kernel supports N <= 16384; loop
over the row for more")
    num_warps = min(max(BLOCK_N // 256, 1), 8)   # ~256 elements per warp,
1..8 warps
    return BLOCK_N, num_warps


def layer_norm(x: torch.Tensor, w: torch.Tensor, b: torch.Tensor, eps: float
= 1e-5):
    shape = x.shape
    x2 = x.reshape(-1, shape[-1])                # [..., N] -> [M, N]
    if x2.stride(-1) != 1:
        x2 = x2.contiguous()
    M, N = x2.shape
    y = torch.empty_like(x2)
    mean = torch.empty(M, dtype=torch.float32, device=x.device)
    rstd = torch.empty(M, dtype=torch.float32, device=x.device)
    if M > 0:
        BLOCK_N, num_warps = _launch_params(N)
        layer_norm_kernel[(M,)](x2, w, b, y, mean, rstd, x2.stride(0), y.stri
de(0), N, eps,
                                BLOCK_N=BLOCK_N, num_warps=num_warps)
    return y.reshape(shape), mean, rstd


def rms_norm(x: torch.Tensor, w: torch.Tensor, eps: float = 1e-6) -> torch.Te
```

```
nsor:
    shape = x.shape
    x2 = x.reshape(-1, shape[-1])
    if x2.stride(-1) != 1:
        x2 = x2.contiguous()
    M, N = x2.shape
    y = torch.empty_like(x2)
    if M > 0:
        BLOCK_N, num_warps = _launch_params(N)
        rms_norm_kernel[(M,)](x2, w, y, x2.stride(0), y.stride(0), N, eps, BL
OCK_N=BLOCK_N, num_warps=num_warps)
    return y.reshape(shape)
```

测完之后

- RTX 5090 实测（fp16，8192 行）：N = 8192 时 `F.layer_norm` 199 us，参考解 **176 us · 1524 GB/s**；N = 1024 / 4096 时快 5–8%。
- **RMSNorm 和 LayerNorm 一样快**（176 us 对 176 us）：两者读写的都是 M×N 的 x 和 y，多出的 bias 只有 N 个数，可以忽略；少的那次归约在寄存器里完成，也不花多少时间。memory-bound 的 kernel 里，只有减少**主要数据**的字节数才有用。
- **再融合一层**：vLLM 的 `fused_add_rms_norm` 把残差加也并进来：`x = x + residual; residual = x; y = rms_norm(x)`。改写你的 kernel 实现它，并算算省了多少字节。
- **N 更大怎么办**：官方 Triton LayerNorm tutorial 用 `for off in range(0, N, BLOCK_N)` 循环分段累加，再做第二遍归一化。
- backward：`dx` 需要 `mean`、`rstd` 以及两个行内归约（`Σ dy·w·x̂` 和 `Σ dy·w`）；`dw`、`db` 要跨行求和，需要第二阶段归约。这是很好的进阶练习。

▼ 面试追问：为什么 LLM 普遍从 LayerNorm 换成了 RMSNorm?

RMSNorm 去掉了减均值和 bias，经验上对模型质量几乎没有影响，参数和计算更少、实现也更简单，更容易和残差加、量化等相邻操作融合。但单看 norm kernel 本身，它并不会明显更快：两者都是 memory-bound 的，时间由读 x、写 y 决定（本题实测两者耗时相同）。真正的收益来自融合：一个更简单的 norm 更容易和前后的操作合成一个 kernel，从而省掉整趟读写。

核心 · Track T · Triton · 第 8 题

T4 Triton GEMM：tl.dot、grouped ordering、autotune

30 行 Triton 达到 cuBLAS 的 80–100%：tl.dot 用上 Tensor Core，二维指针块描述 tile，grouped ordering 提高 L2 命中，autotune 为每个 shape 挑配置。

60 min | 难度 中 | 你写 `mine/t4_matmul.py` | 验收 `./check.sh t4` | 参考解 `solutions/t4_matmul.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. grouped ordering 是什么
5. 图解与类比
6. 题目要求
7. 题干
8. 验收
9. 提示
10. 参考解
11. 测完之后

背景

C4 用 CUDA 手写了 tiling 和 register tiling，但没有用 Tensor Core。Triton 把 C4 的一整套（shared memory staging、寄存器分块、Tensor Core 指令、软件流水线）都交给编译器，你只需要描述 **“一个 program 算 C 的哪一块、沿 K 怎么走”**。官方 Matrix Multiplication tutorial 就是这道题的原型。真实系统里，vLLM 的 MoE `fused_moe` kernel、各种量化 GEMM 也都是在这个骨架上改出来的。

源码与资料

`triton › 官方 tutorial 03-matrix-multiplication` · `triton › 08-grouped-gemm` · `triton › 09-persistent-matmul` · `triton › 10-block-scaled-matmul` · `triton › Autotuner`

要用到的知识

- **二维指针块**：`a_ptrs = a_ptr + offs_m[:, None] * stride_am + offs_k[None, :] * stride_ak`，得到 `[BLOCK_M, BLOCK_K]` 的指针。用 stride 而不是假设行主序，所以转置后的输入（stride (1, M)）也能用。
- **`tl.dot(a, b, acc)`**：`acc += a @ b`。a、b 是 fp16 / bf16，acc 是 fp32。三个维度都至少 16。

- **沿 K 循环**：`for k in range(0, tl.cdiv(K, BLOCK_K))`，每轮 `a_ptrs += BLOCK_K * stride_ak`。最后一段 K 不满时用 mask 填 0。
- **grouped ordering**：把一维的 `pid` 映射成 `(pid_m, pid_n)` 时，不按行主序排，而是把 `GROUP_M` 行 tile 编成一组，在组内先沿列方向（m 方向）走。
- **autotune**：`triton.autotune(configs=[...], key=["M", "N", "K"])`。

---

grouped ordering 是什么

C 被切成 `num_pid_m × num_pid_n` 个 tile。同一时刻 GPU 上同时在跑的 program 大约有 SM 数 × 每 SM 驻留数这么多。如果按行主序编号，同时在跑的是一整行 tile：它们共用 A 的同一个行条带，却各需要 B 的一个不同列条带，B 的工作集很大，L2 装不下。分组之后，同时在跑的是 `GROUP_M` 行 × 若干列的一块方形区域：A 和 B 的条带都被多个 program 复用。

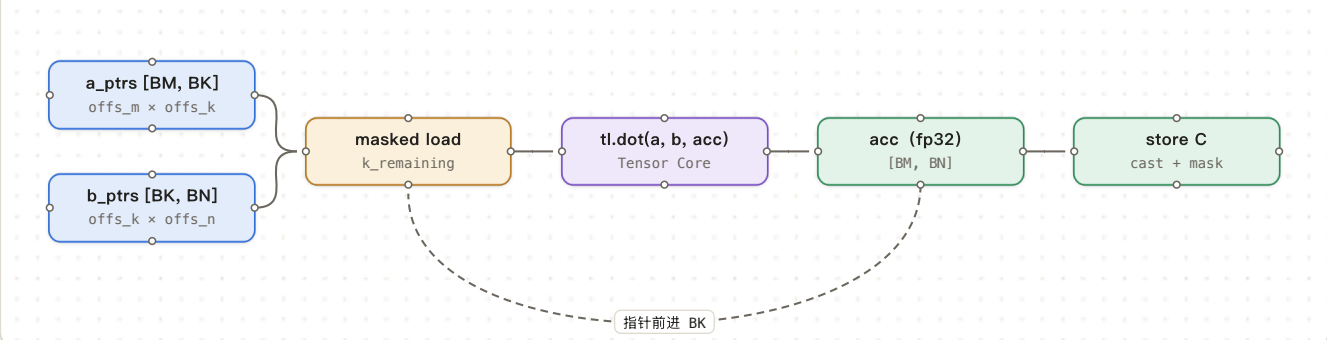
```
pid 的编号顺序（num_pid_m = 4, num_pid_n = 3）

行主序：            GROUP_M = 2：
 0  1  2             0  2  4
 3  4  5             1  3  5
 6  7  8             6  8 10
 9 10 11             7  9 11
```

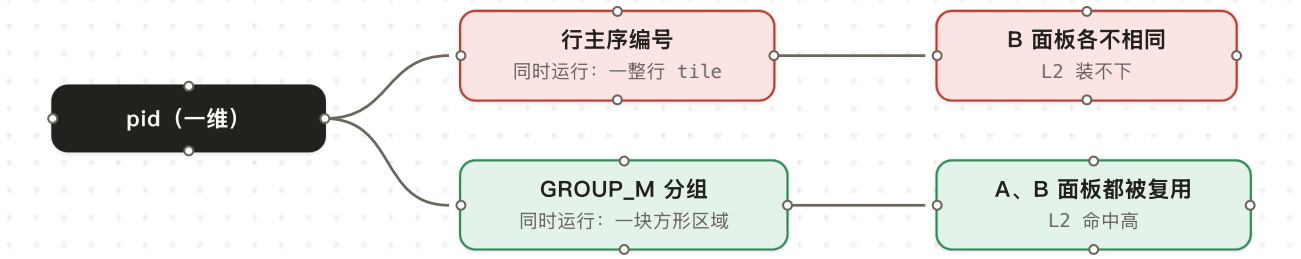
```
num_pid_in_group = GROUP_M * num_pid_n
group_id = pid // num_pid_in_group
first_pid_m = group_id * GROUP_M
group_size_m = min(num_pid_m - first_pid_m, GROUP_M)      # 最后一组可能不满
pid_m = first_pid_m + ((pid % num_pid_in_group) % group_size_m)
pid_n = (pid % num_pid_in_group) // group_size_m
```

图解与类比

K 循环：两个指针块 + 一次 tl.dot



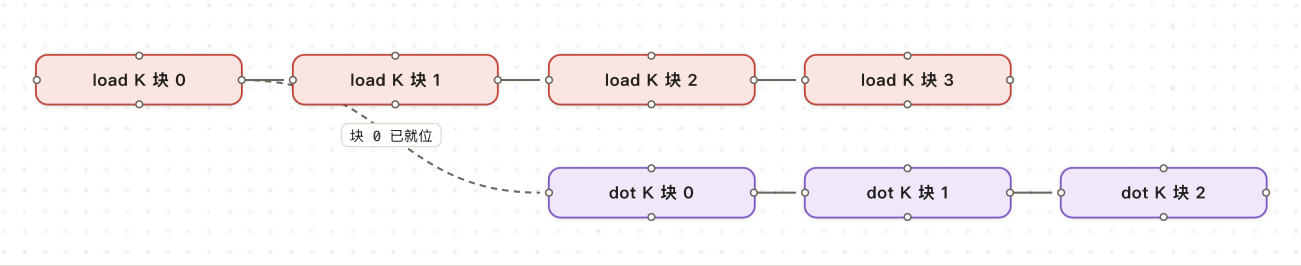
grouped ordering 为什么能提高 L2 命中



类比 · 图书馆的服务台

同一时刻在读书的人越是集中在少数几排书架，管理员（L2）越能把这些书都摆在服务台上；如果每个人要的都是不同书架上的书，服务台堆不下，只能一趟趟回库房（HBM）。grouped ordering 就是安排大家“先把这几排书架读完再换”。

num_stages = 3：搬运和计算重叠



计算第 k 块时，第 k+1、k+2 块已经在搬运（cp.async / TMA → shared memory）。每多一个 stage 就多占一份 shared memory。

类比 · 洗衣、烘干、叠衣服

只有一台洗衣机和一台烘干机：第一桶在烘干时，第二桶已经在洗了。软件流水线就是让“搬数据”（洗）和“算”（烘）同时进行，总时间接近两者中较慢的那个，而不是两者之和。

题目要求

wrapper 和 autotune 配置已给出。实现 `matmul_kernel`：

- `C = A @ B`，A、B 为 fp16 或 bf16，任意 stride，任意 M、N、K（不必是 block 的倍数）；fp32 累加，输出与输入同 dtype。

- 必须使用 grouped ordering（测试用 5×3 个 tile、`GROUP_M=2` 检查每个 tile 都恰好被算一次）。
- CPU 解释器下用固定 config；GPU 上 `config=None` 时走 autotune。

题干

mine/t4_matmul.py 62 行

```
"""T4 · Tiled GEMM with tl.dot, grouped ordering and autotuning — YOUR IMPLEMENTATION.

C[M, N] = A[M, K] @ B[K, N]; fp16/bf16 in, fp32 accumulate, C in the input dtype.
The wrapper and autotune configs are done; implement matmul_kernel.
"""
import os

import torch
import triton
import triton.language as tl


@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr,
                  GROUP_M: tl.constexpr):
    # TODO:
    # 1. pid -> (pid_m, pid_n) with grouped ordering (GROUP_M rows of tiles per group)
    # 2. offs_m, offs_n, offs_k; a_ptrs [BLOCK_M, BLOCK_K], b_ptrs [BLOCK_K, BLOCK_N]
    # 3. acc = zeros fp32; loop over K tiles: masked loads, acc = tl.dot(a, b, acc), advance pointers
    # 4. cast and masked store of the C tile
    pass


# Candidate configs for the autotuner. num_stages = software-pipelining depth (how many K tiles
# are prefetched into shared memory ahead of use); num_warps = threads per program / 32.
AUTOTUNE_CONFIGS = [
    triton.Config({"BLOCK_M": 128, "BLOCK_N": 256, "BLOCK_K": 64, "GROUP_M": 8}, num_stages=3, num_warps=8),
    triton.Config({"BLOCK_M": 128, "BLOCK_N": 128, "BLOCK_K": 64, "GROUP_M": 8},
```

```
num_stages=4, num_warps=4),
    triton.Config({"BLOCK_M": 128, "BLOCK_N": 64, "BLOCK_K": 32, "GROUP_M": 8}, n
um_stages=4, num_warps=4),
    triton.Config({"BLOCK_M": 64, "BLOCK_N": 128, "BLOCK_K": 32, "GROUP_M": 8}, n
um_stages=4, num_warps=4),
    triton.Config({"BLOCK_M": 64, "BLOCK_N": 64, "BLOCK_K": 32, "GROUP_M": 8}, nu
m_stages=5, num_warps=2),
]

# autotune wraps the same jitted kernel: the first call for each new (M, N, K) be
nchmarks all
# configs and caches the fastest one; later calls with that key reuse it.
matmul_kernel_autotuned = triton.autotune(configs=AUTOTUNE_CONFIGS, key=["M",
"N", "K"])(matmul_kernel)


def matmul(a: torch.Tensor, b: torch.Tensor, config: dict | None = None) -> torc
h.Tensor:
    """config=None -> autotuned on GPU; pass e.g. {"BLOCK_M": 64, ...} to force o
ne config."""
    assert a.dim() == 2 and b.dim() == 2 and a.shape[1] == b.shape[0], "shape mis
match"
    assert a.dtype == b.dtype and a.dtype in (torch.float16, torch.bfloat16)
    M, K = a.shape
    _, N = b.shape
    c = torch.empty((M, N), dtype=a.dtype, device=a.device)
    if M == 0 or N == 0:
        return c
    args = (a, b, c, M, N, K, a.stride(0), a.stride(1), b.stride(0), b.stride(1),
c.stride(0), c.stride(1))

    # The CPU interpreter cannot time configs, so it always uses one fixed confi
g.
    if config is None and os.environ.get("TRITON_INTERPRET") == "1":
        config = {"BLOCK_M": 64, "BLOCK_N": 64, "BLOCK_K": 32, "GROUP_M": 8}
    if config is not None:
        grid = (triton.cdiv(M, config["BLOCK_M"]) * triton.cdiv(N, config["BLOCK_
N"]),)
        matmul_kernel[grid](*args, **config)
    else:
        # A 1-D grid sized from whichever config the autotuner is trying.
        grid = lambda meta: (triton.cdiv(M, meta["BLOCK_M"]) * triton.cdiv(N, met
a["BLOCK_N"]),)
        matmul_kernel_autotuned[grid](*args)
    return c
```

验收

```
./check.sh t4
./check.sh t4 --bench       # fp16, 512 … 8192 的方阵, vs cuBLAS 的 TFLOP/s
```

tests/test_t4_matmul.py 47 行

```
import pytest
import torch

from common.env import BF16_OK, DEVICE, HAS_CUDA
from drill import load

FIXED = {"BLOCK_M": 64, "BLOCK_N": 64, "BLOCK_K": 32, "GROUP_M": 8}


@pytest.mark.parametrize("mnk", [(16, 16, 16), (64, 64, 64), (100, 70, 130), (25
7, 129, 96)])
def test_matmul_fixed_config(mnk):
    M, N, K = mnk
    m = load("t4_matmul")
    a = torch.randn(M, K, device=DEVICE).half()
    b = torch.randn(K, N, device=DEVICE).half()
    out = m.matmul(a, b, config=FIXED)
    assert out.dtype == torch.float16 and out.shape == (M, N)
    torch.testing.assert_close(out.float(), a.float() @ b.float(), rtol=1e-2, ato
l=5e-2)


def test_matmul_transposed_input():
    m = load("t4_matmul")
    a = torch.randn(96, 80, device=DEVICE).half().t()     # [80, 96], strides (1,
80)
    b = torch.randn(96, 72, device=DEVICE).half()
    torch.testing.assert_close(m.matmul(a, b, config=FIXED).float(), a.float() @
b.float(), rtol=1e-2, atol=5e-2)


def test_grouped_order_covers_every_tile_once():
    m = load("t4_matmul")
    # 5 x 3 tiles with GROUP_M=2: if the pid -> (pid_m, pid_n) mapping skipped or
repeated a
    # tile, some output tile would be left unwritten (garbage from torch.empty) a
nd the check fails.
    cfg = {"BLOCK_M": 16, "BLOCK_N": 16, "BLOCK_K": 16, "GROUP_M": 2}
    a = torch.randn(80, 32, device=DEVICE).half()
    b = torch.randn(32, 48, device=DEVICE).half()
    out = m.matmul(a, b, config=cfg)
    torch.testing.assert_close(out.float(), a.float() @ b.float(), rtol=1e-2, ato
```

```
l=5e-2)


@pytest.mark.skipif(not HAS_CUDA, reason="autotuning needs a GPU")
@pytest.mark.parametrize("dtype", [torch.float16, torch.bfloat16])
def test_matmul_autotuned(dtype):
    if dtype == torch.bfloat16 and not BF16_OK:
        pytest.skip("bf16 needs sm_80+")
    m = load("t4_matmul")
    a = torch.randn(1000, 777, device=DEVICE, dtype=dtype)
    b = torch.randn(777, 1500, device=DEVICE, dtype=dtype)
    torch.testing.assert_close(m.matmul(a, b).float(), (a.float() @ b.float()), r
tol=2e-2, atol=1e-1)
```

提示

L1 四段式 方向（轻度剧透） 收起

1. pid → (pid_m, pid_n)（上面的公式）
2. 造 `a_ptrs`、`b_ptrs`，`acc = tl.zeros((BLOCK_M, BLOCK_N), tl.float32)`
3. K 循环：masked load → `acc = tl.dot(a, b, acc)` → 指针前进
4. `acc.to(c 的类型)`，按 `(offs_m < M) & (offs_n < N)` 做 masked store

L2 常见坑 拆解与常见坑（中度剧透） 收起

- K 方向的 mask：用 `k_remaining = K - k * BLOCK_K` 和 `offs_k < k_remaining`，而不是 `offs_k < K`（offs_k 每轮都是 0…BLOCK_K-1）。
- 只对 a 的 m 方向和 b 的 n 方向做 mask，忘了 K 方向：最后一段会读到下一行的数据，结果错得不明显。
- 指针前进写成 `a_ptrs += BLOCK_K`：输入转置后 stride_ak 不是 1。
- `tl.dot` 的输入如果被提前转成 fp32，就用不上 fp16 Tensor Core 了（在 Ampere 上会走 TF32，慢很多，精度也不同）。

L3 **主循环** 几乎是答案（重度剧透） 收起 ▾

```
acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)
for k in range(0, tl.cdiv(K, BLOCK_K)):
    k_remaining = K - k * BLOCK_K
    a = tl.load(a_ptrs, mask=(offs_m[:, None] < M) & (offs_k[None, :] < k_remai
    b = tl.load(b_ptrs, mask=(offs_k[:, None] < k_remaining) & (offs_n[None, :]
    acc = tl.dot(a, b, acc)
    a_ptrs += BLOCK_K * stride_ak
    b_ptrs += BLOCK_K * stride_bk
```

参考解

解 **solutions/t4_matmul.py** 先自己写完、跑过测试再看 收起

solutions/t4_matmul.py 94 行

```
"""T4 · Tiled GEMM with tl.dot, grouped ordering and autotuning — reference solution.

C[M, N] = A[M, K] @ B[K, N]; A, B fp16 or bf16, fp32 accumulation, C in the input dtype.
Each program computes one BLOCK_M x BLOCK_N tile of C, walking K in BLOCK_K steps.
tl.dot lowers to Tensor Core instructions (mma / wgmma) on NVIDIA GPUs.
"""
import os

import torch
import triton
import triton.language as tl


@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn,
                  BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr,
                  GROUP_M: tl.constexpr):
    # ---- 1. which C tile does this program own? (grouped / "super-grouping" order)
    # Plain row-major order (pid_m = pid // num_pid_n) makes consecutive programs sweep a
    # whole row of tiles: they share one A row-panel but each needs a different B column-panel.
    # Grouping GROUP_M rows of tiles together and going down the columns first means programs
    # running at the same time reuse both A and B panels while they are still in L2.
    pid = tl.program_id(0)
    num_pid_m = tl.cdiv(M, BLOCK_M)
    num_pid_n = tl.cdiv(N, BLOCK_N)
    num_pid_in_group = GROUP_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_M
    group_size_m = min(num_pid_m - first_pid_m, GROUP_M)   # the last group may be shorter
    pid_m = first_pid_m + ((pid % num_pid_in_group) % group_size_m)
```

```
    pid_n = (pid % num_pid_in_group) // group_size_m

    # ---- 2. pointers to the first A and B tiles (2-D blocks of pointers via
broadcasting)
    offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    offs_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
    offs_k = tl.arange(0, BLOCK_K)
    a_ptrs = a_ptr + offs_m[:, None] * stride_am + offs_k[None, :] * stride_a
k   # [BLOCK_M, BLOCK_K]
    b_ptrs = b_ptr + offs_k[:, None] * stride_bk + offs_n[None, :] * stride_b
n   # [BLOCK_K, BLOCK_N]

    # ---- 3. main loop over K: accumulate in fp32 registers
    acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_K)):
        k_remaining = K - k * BLOCK_K
        # Masked loads return 0 outside the matrix; zeros add nothing to the
dot product.
        a = tl.load(a_ptrs, mask=(offs_m[:, None] < M) & (offs_k[None, :] < k
_remaining), other=0.0)
        b = tl.load(b_ptrs, mask=(offs_k[:, None] < k_remaining) & (offs_n[No
ne, :] < N), other=0.0)
        acc = tl.dot(a, b, acc)          # acc += a @ b on Tensor Cores
        a_ptrs += BLOCK_K * stride_ak    # slide both tiles along K
        b_ptrs += BLOCK_K * stride_bk

    # ---- 4. epilogue: cast and store the C tile
    c = acc.to(c_ptr.dtype.element_ty)
    c_ptrs = c_ptr + offs_m[:, None] * stride_cm + offs_n[None, :] * stride_c
n
    tl.store(c_ptrs, c, mask=(offs_m[:, None] < M) & (offs_n[None, :] < N))


# Candidate configs for the autotuner. num_stages = software-pipelining depth
(how many K tiles
# are prefetched into shared memory ahead of use); num_warps = threads per pr
ogram / 32.
AUTOTUNE_CONFIGS = [
    triton.Config({"BLOCK_M": 128, "BLOCK_N": 256, "BLOCK_K": 64, "GROUP_M":
8}, num_stages=3, num_warps=8),
    triton.Config({"BLOCK_M": 128, "BLOCK_N": 128, "BLOCK_K": 64, "GROUP_M":
8}, num_stages=4, num_warps=4),
    triton.Config({"BLOCK_M": 128, "BLOCK_N": 64, "BLOCK_K": 32, "GROUP_M":
8}, num_stages=4, num_warps=4),
    triton.Config({"BLOCK_M": 64, "BLOCK_N": 128, "BLOCK_K": 32, "GROUP_M":
8}, num_stages=4, num_warps=4),
    triton.Config({"BLOCK_M": 64, "BLOCK_N": 64, "BLOCK_K": 32, "GROUP_M":
8}, num_stages=5, num_warps=2),
]
```

```
# autotune wraps the same jitted kernel: the first call for each new (M, N,
K) benchmarks all
# configs and caches the fastest one; later calls with that key reuse it.
matmul_kernel_autotuned = triton.autotune(configs=AUTOTUNE_CONFIGS, key=["M",
"N", "K"])(matmul_kernel)


def matmul(a: torch.Tensor, b: torch.Tensor, config: dict | None = None) -> t
orch.Tensor:
    """config=None -> autotuned on GPU; pass e.g. {"BLOCK_M": 64, ...} to for
ce one config."""
    assert a.dim() == 2 and b.dim() == 2 and a.shape[1] == b.shape[0], "shape
mismatch"
    assert a.dtype == b.dtype and a.dtype in (torch.float16, torch.bfloat16)
    M, K = a.shape
    _, N = b.shape
    c = torch.empty((M, N), dtype=a.dtype, device=a.device)
    if M == 0 or N == 0:
        return c
    args = (a, b, c, M, N, K, a.stride(0), a.stride(1), b.stride(0), b.stride
(1), c.stride(0), c.stride(1))

    # The CPU interpreter cannot time configs, so it always uses one fixed co
nfig.
    if config is None and os.environ.get("TRITON_INTERPRET") == "1":
        config = {"BLOCK_M": 64, "BLOCK_N": 64, "BLOCK_K": 32, "GROUP_M": 8}
    if config is not None:
        grid = (triton.cdiv(M, config["BLOCK_M"]) * triton.cdiv(N, config["BL
OCK_N"]),)
        matmul_kernel[grid](*args, **config)
    else:
        # A 1-D grid sized from whichever config the autotuner is trying.
        grid = lambda meta: (triton.cdiv(M, meta["BLOCK_M"]) * triton.cdiv(N,
meta["BLOCK_N"]),)
        matmul_kernel_autotuned[grid](*args)
    return c
```

测完之后

- RTX 5090 实测（fp16，单位 TFLOP/s）：512³ 时 21.7 vs cuBLAS 26.8（81%）；1024³ 115.5 vs 116.9；2048³ 164.4 vs 164.6；4096³ 209.3 vs 211.0；8192³ **217.8 vs 217.7**。从 1024 开始基本追平 cuBLAS。小矩阵时 tile 数不够填满 170 个 SM，差距更大。
- 把 `GROUP_M` 改成 1（退化为行主序），在 8192³ 上对比：L2 命中率（`ncu` 的 `lts__t_sector_hit_rate`）和时间都会变差。

- 记录 autotune 为每个 shape 选中的 config（`matmul_kernel_autotuned.best_config`），观察大 shape 和小 shape 的选择有什么不同。这正是原计划 Week 4 要求的实验记录。
- `num_stages` 从 2 调到 5：stage 越多预取越深，但每个 stage 都要占一份 shared memory，超过上限会编译失败或降低 occupancy。
- 进阶：在 epilogue 里融合 bias + 激活（`c = gelu(acc + bias)`），这是 GEMM 融合最常见的形式。

▼ 面试追问：Triton GEMM 和 cuBLAS 差在哪里?

大 shape 下主要差在一些底层细节：Hopper 上的 TMA 和 wgmma 异步流水线、warp specialization（一部分 warp 只负责搬数据，一部分只负责算）、persistent kernel（program 数等于 SM 数，循环处理多个 tile），以及为特定 shape 手工调过的 tile 配置。新版 Triton 和 Gluon 已经能显式表达其中大部分优化；小 shape 下差距主要来自启动开销和 SM 利用率（split-K、stream-K 能缓解）。

核心 · Track T · Triton · 第 9 题

T5 FlashAttention forward：tiling + online softmax

把 T2 的 online softmax 和 T4 的 tl.dot 合在一起：attention 不再生成 S×S 的分数矩阵，显存从 O(S²) 降到 O(S)，速度也因为少搬数据而大幅提升。

90 min | 难度 难 | 你写 `mine/t5_flash_attn.py` | 验收 `./check.sh t5` | 参考解 `solutions/t5_flash_attn.py`

1. 背景
2. 源码与资料
3. 要用到的知识：online softmax
4. 要用到的知识：形状与掩码
5. 图解与类比
6. 题目要求
7. 题干
8. 验收
9. 提示
10. 参考解
11. 测完之后

背景

标准 attention：`P = softmax(QKᵀ · scale)`，`O = P V`。naive 实现要把 `[S, S]` 的 P 写回显存再读出来。S = 8192、32 个 head 时，P 在 fp16 下有 4 GB。FlashAttention 的观察是：**softmax 可以分块、增量地算**，所以只要让一个 program 负责 BLOCK_M 个 query，把 K、V 按 BLOCK_N 一块块流过来，所有中间结果都留在片上。vLLM / SGLang 的 prefill、训练框架的 attention 都基于这个算法。

源码与资料

`triton › 官方 tutorial 06-fused-attention` · FlashAttention · FlashAttention-2 · FlashAttention-3 · FlashAttention-4 · `fa › FA2 flash_fwd_kernel.h` · `fa › FA2 online softmax` · `fa › FA3 主循环` · `fa › FA4 (CuTe DSL)` · `flashinfer › FlashInfer prefill.cuh`

要用到的知识：online softmax

对 query 行 i，维护三个量（fp32）：

- `m`：目前见过的最大分数
- `l`：`Σ exp(s − m)`

- `acc`：`Σ exp(s − m) · v`

来了新的一块分数 `s_blk`（以及对应的 `v_blk`）：

```
m_new = max(m, max(s_blk))
p     = exp(s_blk − m_new)            # 新块的权重
alpha = exp(m − m_new)                # 旧状态的缩放系数（最大值变大时，旧的指数要整体变小）
l     = l · alpha + Σ p
acc   = acc · alpha + p @ v_blk
m     = m_new
```

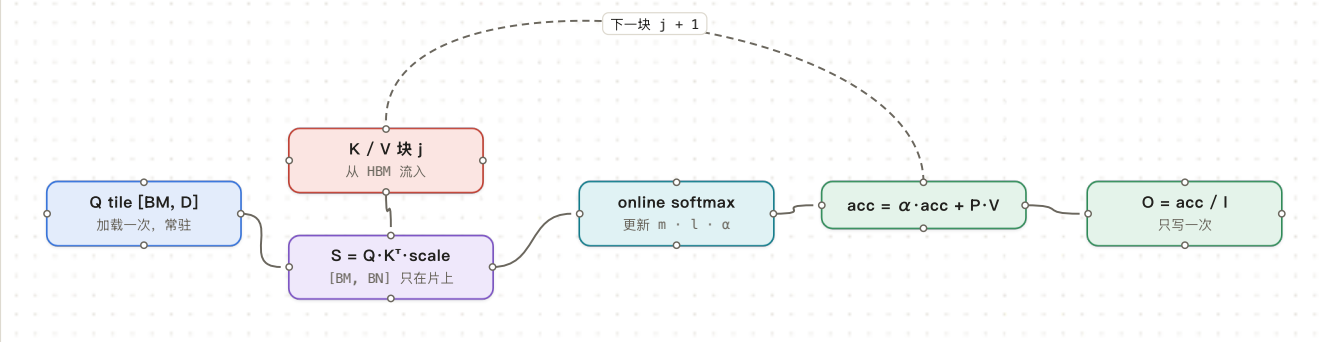
全部块处理完之后，`o = acc / l`。这和一次性算完全 softmax 在数学上**完全等价**。

要用到的知识：形状与掩码

- q tile：`[BLOCK_M, D]`，循环外加载一次。
- 直接把 K **按转置的形状**加载成 `[D, BLOCK_N]`：`K + cols[None, :] * stride_s + offs_d[:, None]`，这样 `tl.dot(q, k)` 就是 `[BLOCK_M, BLOCK_N]`。
- 无效位置的分数设为 −inf：超出序列长度的 key（`cols >= S`），以及 causal 时 `cols > query 位置`。
- causal 时，query 块 `start_m` 只需要看前 `(start_m + 1) · BLOCK_M` 个 key，循环上界直接截断，省一半计算。
- `p.to(v.dtype)` 后再做 `tl.dot(p, v, acc)`：Tensor Core 需要 fp16 / bf16 输入。

图解与类比

FlashAttention：Q 常驻，K/V 流过

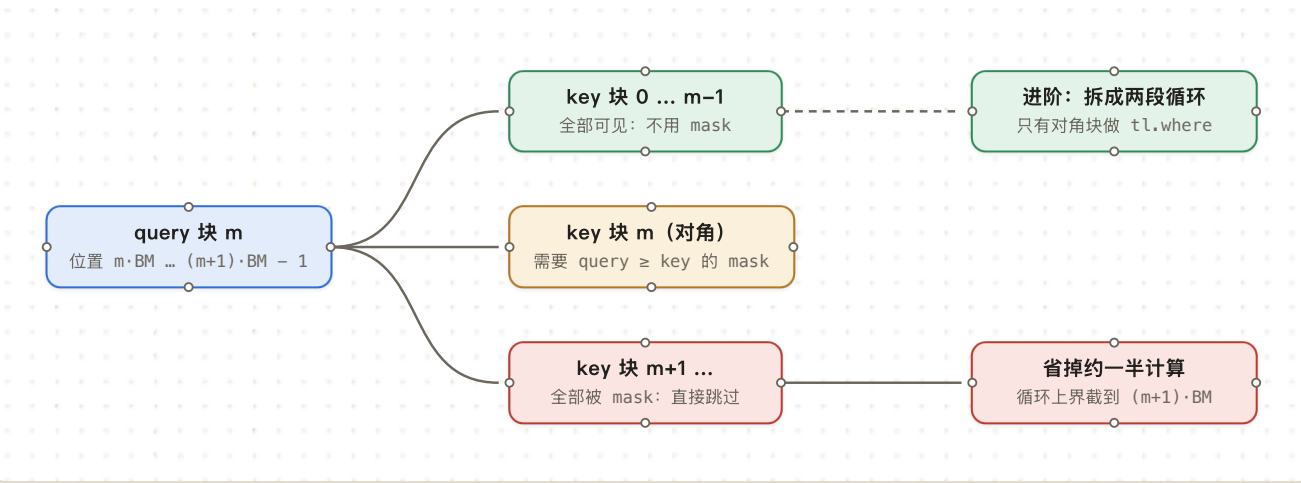


从头到尾不存在 S×S 的矩阵：每一时刻片上只有 BM×BN 的一小块分数。HBM 读写从 $O(S^2)$ 降到 $O(S \cdot D)$。

> **类比 · 比赛记分：只记最高分和折算后的总分**

你不需要保存每位选手的成绩，只要记住“目前的最高分 m”和“按这个最高分折算后的总分 l”。来了一位更高分的选手，最高分从 m_old 变成 m_new，之前的总分要整体打个折：乘以 α = exp(m_old − m_new)。 加权平均（acc）也同样打折。全部选手看完，用 acc / l 就得到和一次性看完全部成绩完全相同的结果。

causal：query 块只需要看左下三角



类比 · 只看已经发生的事

causal attention 里每个词只能看见它前面的词，就像写日记时只能回忆过去、不能偷看未来。query 块 m 之后的 key 块全是“未来”，整块跳过；只有对角线那一块是“今天”，一部分已经发生、一部分还没有，需要逐个判断（mask）。

题目要求

wrapper 已写好：`grid = (cdiv(S, BLOCK_M), B·H)`，q、k、v、o 形状相同 `[B, H, S, D]` 且连续，D ∈ {16, 32, 64, 128}，S 任意。实现 `flash_attn_fwd_kernel`：

- 支持 `CAUSAL`（constexpr）和自定义 `sm_scale`。
- 与 `F.scaled_dot_product_attention` 在 fp16 下误差 ⩽ 1e−2。
- 不能分配或写出任何 `[S, S]` 大小的中间结果。

题干

mine/t5_flash_attn.py 45 行

```
"""T5 · FlashAttention forward (tiled + online softmax) in Triton — YOUR IMPLEMEN
TATION.
```

```
o = softmax(q k^T * sm_scale [+ causal mask]) v, q/k/v/o: [B, H, S, D] fp16/bf16.
The wrapper is done; implement flash_attn_fwd_kernel.
"""
import math

import torch
import triton
import triton.language as tl


@triton.jit
def flash_attn_fwd_kernel(Q, K, V, O, sm_scale,
                          stride_b, stride_h, stride_s,       # same strides for
q, k, v, o; stride_d == 1
                          H, S,
                          CAUSAL: tl.constexpr, HEAD_DIM: tl.constexpr,
                          BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr):
    # TODO:
    # start_m, bh -> off_b, off_h -> base offset
    # load the q tile [BLOCK_M, HEAD_DIM]
    # m_i = -inf, l_i = 0, acc = 0 (fp32)
    # hi = min((start_m + 1) * BLOCK_M, S) if CAUSAL else S
    # for start_n in range(0, hi, BLOCK_N):
    #     load k^T [HEAD_DIM, BLOCK_N]; s = dot(q, k) * sm_scale; mask invalid ke
ys with -inf
    #     online softmax update of m_i, l_i, acc (acc += p @ v)
    # store acc / l_i
    pass


def flash_attention(q: torch.Tensor, k: torch.Tensor, v: torch.Tensor, causal: bo
ol = False,
                    sm_scale: float | None = None, block_m: int = 64, block_n: in
t = 64) -> torch.Tensor:
    assert q.shape == k.shape == v.shape and q.dim() == 4, "q, k, v: [B, H, S,
D]"
    assert q.dtype == k.dtype == v.dtype and q.dtype in (torch.float16, torch.bfl
oat16)
    B, H, S, D = q.shape
    assert D in (16, 32, 64, 128), "HEAD_DIM must be a power of two in [16, 128]"
    q, k, v = q.contiguous(), k.contiguous(), v.contiguous()
    o = torch.empty_like(q)
    if sm_scale is None:
        sm_scale = 1.0 / math.sqrt(D)
    grid = (triton.cdiv(S, block_m), B * H)
    flash_attn_fwd_kernel[grid](q, k, v, o, sm_scale, q.stride(0), q.stride(1),
q.stride(2), H, S,
                                CAUSAL=causal, HEAD_DIM=D, BLOCK_M=block_m, BLOCK
_N=block_n,
```

```
                                num_warps=4, num_stages=2)
    return o
```

验收

```
./check.sh t5               # CPU 上只跑小 shape；GPU 上额外测 S=2048, D=128
./check.sh t5 --bench       # vs torch SDPA, TFLOP/s
```

tests/test_t5_flash_attn.py 38 行

```
import pytest
import torch
import torch.nn.functional as F

from common.env import BF16_OK, DEVICE, HAS_CUDA
from drill import load


def reference(q, k, v, causal):
    return F.scaled_dot_product_attention(q.float(), k.float(), v.float(), is_cau
sal=causal)


@pytest.mark.parametrize("causal", [False, True])
@pytest.mark.parametrize("B,H,S,D", [(1, 1, 16, 16), (2, 3, 100, 64), (1, 2, 257,
32)])
def test_flash_attention(B, H, S, D, causal):
    m = load("t5_flash_attn")
    q, k, v = (torch.randn(B, H, S, D, device=DEVICE).half() for _ in range(3))
    out = m.flash_attention(q, k, v, causal=causal)
    assert out.shape == q.shape and out.dtype == q.dtype
    torch.testing.assert_close(out.float(), reference(q, k, v, causal), rtol=1e-
2, atol=1e-2)


def test_custom_scale():
    m = load("t5_flash_attn")
    q, k, v = (torch.randn(1, 2, 64, 32, device=DEVICE).half() for _ in range(3))
    out = m.flash_attention(q, k, v, sm_scale=0.05)
    ref = F.scaled_dot_product_attention(q.float(), k.float(), v.float(), scale=
0.05)
    torch.testing.assert_close(out.float(), ref, rtol=1e-2, atol=1e-2)


@pytest.mark.skipif(not HAS_CUDA, reason="larger shapes are slow in the CPU inter
preter")
@pytest.mark.parametrize("causal", [False, True])
```

```python
def test_flash_attention_large(causal):
    m = load("t5_flash_attn")
    dtype = torch.bfloat16 if BF16_OK else torch.float16
    q, k, v = (torch.randn(2, 8, 2048, 128, device=DEVICE, dtype=dtype) for _ in
range(3))
    torch.testing.assert_close(m.flash_attention(q, k, v, causal=causal).float(),
reference(q, k, v, causal),
                               rtol=2e-2, atol=2e-2)
```

提示

L1 骨架 方向（轻度剧透） 收起

```
定位 (b, h) 和 query 块 → load q
m = -inf, l = 0, acc = 0
for 每个 key 块 (causal 时截断):
    s = q @ kᵀ · scale，无效位置填 -inf
    online softmax 更新（上面 6 行）
o = acc / l → store
```

L2 常见坑 拆解与常见坑（中度剧透） 收起

- 第一轮 `m = -inf`：`alpha = exp(-inf - m_new) = 0`，这是正确的（旧状态本来就是空的）。但如果某一行在**整块**里全是 -inf，`m_new` 也是 -inf，`exp(-inf - (-inf))` 会得到 NaN。本题里每个真实 query 在第一个块就能看到 key 0，所以不会发生；L3 的空分片就会遇到这种情况。
- causal 的比较方向：query 位置 ≥ key 位置才有效（`offs_m[:, None] >= cols[None, :]`）。
- `alpha` 要同时乘到 `l` 和 `acc` 上，漏乘任何一个，结果都会偏。
- 超出 S 的 query 行（最后一个块）要在 store 时 mask 掉。
- `range(0, hi, BLOCK_N)` 的 hi 可以是运行时的值，Triton 支持。

L3 **循环体** 几乎是答案（重度剧透） 收起 ▾

```
for start_n in range(0, hi, BLOCK_N):
    cols = start_n + offs_n
    k = tl.load(K + base + cols[None, :] * stride_s + offs_d[:, None], mask=col
    s = tl.dot(q, k) * sm_scale
    valid = cols[None, :] < S
    if CAUSAL:
        valid = valid & (offs_m[:, None] >= cols[None, :])
    s = tl.where(valid, s, float("-inf"))
    m_new = tl.maximum(m_i, tl.max(s, axis=1))
    p = tl.exp(s - m_new[:, None])
    alpha = tl.exp(m_i - m_new)
    l_i = l_i * alpha + tl.sum(p, axis=1)
    v = tl.load(V + base + cols[:, None] * stride_s + offs_d[None, :], mask=col
    acc = tl.dot(p.to(v.dtype), v, acc * alpha[:, None])
    m_i = m_new
```

参考解

解 **solutions/t5_flash_attn.py** 先自己写完、跑过测试再看 收起

solutions/t5_flash_attn.py 97 行

```
"""T5 · FlashAttention forward (tiled + online softmax) in Triton — reference solution.

o = softmax(q k^T * sm_scale [+ causal mask]) v, q/k/v/o: [B, H, S, D] fp16 or bf16.
One program = BLOCK_M queries of one (batch, head). It streams K/V in BLOCK_N chunks and
never materialises the S x S score matrix: only a BLOCK_M x BLOCK_N tile exists at a time.

Online softmax state per query row i (all fp32):
    m_i  running max of the scores seen so far
    l_i  running sum of exp(score - m_i)
    acc  running sum of exp(score - m_i) * v
When a new chunk raises the max to m_new, old terms are rescaled by alpha = exp(m_i - m_new).
At the end o_i = acc_i / l_i.
"""
import math

import torch
import triton
import triton.language as tl


@triton.jit
def flash_attn_fwd_kernel(Q, K, V, O, sm_scale,
                          stride_b, stride_h, stride_s,       # same strides for q, k, v, o; stride_d == 1
                          H, S,
                          CAUSAL: tl.constexpr, HEAD_DIM: tl.constexpr,
                          BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr):
    start_m = tl.program_id(0)                # which block of queries
    bh = tl.program_id(1)                     # which (batch, head)
    off_b = bh // H
    off_h = bh % H
    base = off_b.to(tl.int64) * stride_b + off_h.to(tl.int64) * stride_h

    offs_m = start_m * BLOCK_M + tl.arange(0, BLOCK_M)    # query positions
    offs_n = tl.arange(0, BLOCK_N)                        # key positions within a chunk
    offs_d = tl.arange(0, HEAD_DIM)
```

```
    # The query tile is loaded once and stays in registers for the whole loo
p.
    q = tl.load(Q + base + offs_m[:, None] * stride_s + offs_d[None, :],
                mask=offs_m[:, None] < S, other=0.0)

    m_i = tl.full([BLOCK_M], float("-inf"), dtype=tl.float32)
    l_i = tl.zeros([BLOCK_M], dtype=tl.float32)
    acc = tl.zeros([BLOCK_M, HEAD_DIM], dtype=tl.float32)

    # Causal: queries in this block never look past their own last position,
so key chunks
    # beyond (start_m + 1) * BLOCK_M are skipped entirely (about half the wor
k).
    if CAUSAL:
        hi = tl.minimum((start_m + 1) * BLOCK_M, S)
    else:
        hi = S

    for start_n in range(0, hi, BLOCK_N):
        cols = start_n + offs_n
        # Load K already transposed: a [HEAD_DIM, BLOCK_N] tile, so q @ k is
a plain tl.dot.
        k = tl.load(K + base + cols[None, :] * stride_s + offs_d[:, None],
                    mask=cols[None, :] < S, other=0.0)
        s = tl.dot(q, k) * sm_scale                          # [BLOCK_M, BLOCK_
N] fp32 scores

        valid = cols[None, :] < S                            # keys past the en
d of the sequence
        if CAUSAL:
            valid = valid & (offs_m[:, None] >= cols[None, :])
        s = tl.where(valid, s, float("-inf"))

        # Online softmax update. Every real query row has at least one valid
key (key 0) in
        # the first chunk, so m_new is finite from the first iteration on and
no
        # (-inf) - (-inf) = NaN can appear.
        m_new = tl.maximum(m_i, tl.max(s, axis=1))
        p = tl.exp(s - m_new[:, None])                       # [BLOCK_M, BLOCK_
N]
        alpha = tl.exp(m_i - m_new)                          # rescales the old
state; 0 on the first chunk
        l_i = l_i * alpha + tl.sum(p, axis=1)

        v = tl.load(V + base + cols[:, None] * stride_s + offs_d[None, :],
                    mask=cols[:, None] < S, other=0.0)
        acc = acc * alpha[:, None]
        acc = tl.dot(p.to(v.dtype), v, acc)                  # acc += p @ v (p
```

```
in fp16/bf16 for Tensor Cores)
        m_i = m_new

    o = acc / l_i[:, None]
    tl.store(O + base + offs_m[:, None] * stride_s + offs_d[None, :],
             o.to(O.dtype.element_ty), mask=offs_m[:, None] < S)


def flash_attention(q: torch.Tensor, k: torch.Tensor, v: torch.Tensor, causa
l: bool = False,
                    sm_scale: float | None = None, block_m: int = 64, block_
n: int = 64) -> torch.Tensor:
    assert q.shape == k.shape == v.shape and q.dim() == 4, "q, k, v: [B, H,
S, D]"
    assert q.dtype == k.dtype == v.dtype and q.dtype in (torch.float16, torc
h.bfloat16)
    B, H, S, D = q.shape
    assert D in (16, 32, 64, 128), "HEAD_DIM must be a power of two in [16, 1
28]"
    q, k, v = q.contiguous(), k.contiguous(), v.contiguous()
    o = torch.empty_like(q)
    if sm_scale is None:
        sm_scale = 1.0 / math.sqrt(D)
    grid = (triton.cdiv(S, block_m), B * H)
    flash_attn_fwd_kernel[grid](q, k, v, o, sm_scale, q.stride(0), q.stride
(1), q.stride(2), H, S,
                                CAUSAL=causal, HEAD_DIM=D, BLOCK_M=block_m, B
LOCK_N=block_n,
                                num_warps=4, num_stages=2)
    return o
```

测完之后

- FLOPs 按 $4 \cdot B \cdot H \cdot S^2 \cdot D$（$QK^T$ 和 PV 各 $2 \cdot S^2 \cdot D$，causal 时减半）计算。RTX 5090 实测（B = 4、H = 16、D = 64，fp16，单位 TFLOP/s）：

S	非 causal：SDPA / 参考解	causal：SDPA / 参考解
512	116.6 / 116.5	58.1 / 68.7
1024	155.2 / 161.3	104.5 / 116.5
2048	189.6 / 190.3	149.0 / 150.8
4096	211.2 / 203.0	186.0 / 175.9

这个只有几十行的版本已经和 PyTorch SDPA 处于同一水平（95–118%）。S = 4096 时略慢，下面几条是继续优化的方向。在 H100 上差距会大一些：那里的 FlashAttention-3 用了 wgmma、TMA 和 warp specialization。

- **exp2 技巧**：把 `sm_scale` 乘上 `log2(e)`，用 `tl.exp2` 代替 `tl.exp`，硬件的 `ex2` 指令更快。官方 tutorial 就是这么做的。
- **causal 的两段循环**：对角线之前的块完全不需要 mask，只有对角块需要。拆成两段循环可以省掉大部分 `tl.where`。
- **把 LSE 存下来**：`lse = m + log(l)`，backward 需要它。L3 会用它来合并多个分片。
- 不同的 `BLOCK_M / BLOCK_N / num_warps / num_stages`：在 D = 64 和 D = 128 下最优值不同。用 autotune 试试。
- 你刚写的就是 **prefill** attention。decode（每次只有 1 个 query）为什么不能直接用它？见 L2。

▼ 面试追问：FlashAttention 的 FLOPs 没有减少，为什么会更快？

attention 的瓶颈不在 FLOPs，而在 HBM 读写：naive 实现要把 S×S 的分数矩阵和 softmax 结果写出、再读回，字节数是 $O(S^2)$。FlashAttention 把这部分留在 SRAM / 寄存器里，HBM 读写降到 $O(S \cdot D)$（Q、K、V、O 各读写一次，严格说 K、V 会被每个 query 块重读一遍）。它没有减少计算，减少的是 IO，属于 IO-aware 算法；反而多出了一点重缩放的计算。

Track L · 推理 attention 从零写

Paged KV cache、decode attention、GQA、split-KV：真实 serving 的形状。

核心 · Track L · 推理 attention 从零写 · 第 10 题

L1 Paged KV cache：slot mapping + 写 cache

vLLM 每一步 forward 的第一个自定义 kernel：把新 token 的 K、V 按 block table 写进分页的 KV cache。先用纯 PyTorch 算出 slot，再用 Triton 搬数据。

35 min | 难度 易 | 你写 `mine/l1_paged_kv_write.py` | 验收 `./check.sh l1` | 参考解 `solutions/l1_paged_kv_write.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

PagedAttention 把 KV cache 切成固定大小的 block（page，通常 16 个 token），像操作系统的虚拟内存一样按需分配：每个序列有一张 **block table**，记录它的第 i 个逻辑 block 存在哪个物理 block。这样可以消除碎片，还能让多个序列共享前缀 block。

每一步 forward，新算出的 K、V（prefill 时是整段 prompt，decode 时每个序列一个 token）都要先写进 cache。调度器会为每个 token 预先算好一个 **slot**：

```
slot = block_table[seq][pos // block_size] * block_size + pos % block_size
```

cache 布局为 `[num_blocks, block_size, H_kv, D]`，并且是连续的，所以 slot s 的数据就在扁平偏移 `s · H_kv · D` 处。vLLM 里对应的是 `reshape_and_cache` / `reshape_and_cache_flash`（CUDA），以及 `slot_mapping` 的计算。

源码与资料

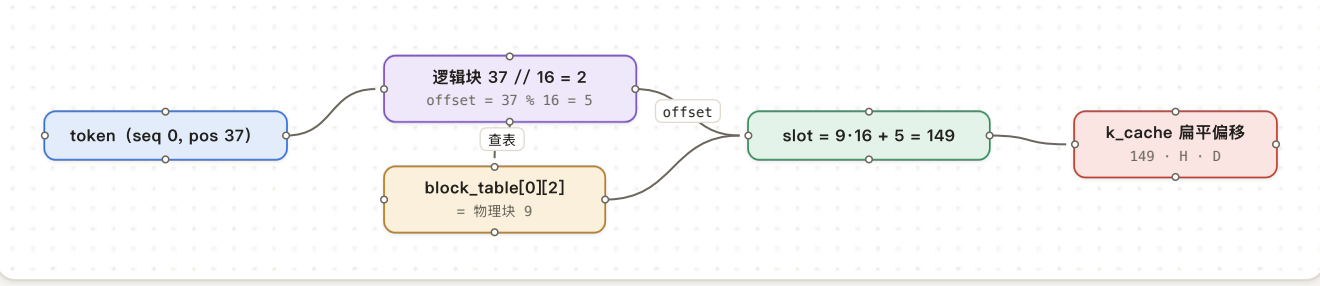
PagedAttention / vLLM 论文 · `vllm › vLLM reshape_and_cache_flash_kernel` · `flashinfer › FlashInfer page.cuh` · `flashinfer › FlashInfer append_paged_kv_cache` · `flashinfer › test_page.py`

要用到的知识

- PyTorch 的高级索引：`block_table[seq_idx, logical_block]` 一次性 gather 出 T 个物理 block 号。
- 每个 token 一个 program；token 的 `H·D` 个值当作一维向量搬运（`BLOCK = next_power_of_2(H·D)`）。
- **padding token**：slot = −1 表示跳过（CUDA graph 的 batch 被补齐时会出现）。用 mask 让整个 program 什么都不写。
- key / value 可能是 fused qkv 投影结果的 view：行 stride 是 `3·H·D` 而不是 `H·D`，但每行内部的 `[H, D]` 是连续的。

图解与类比

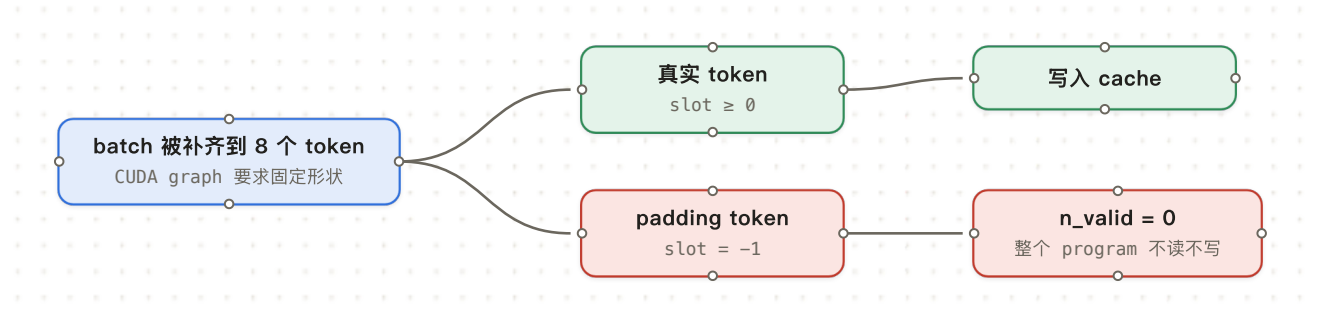
一个 token 的 KV 写到哪里



类比 · 酒店分房

每个旅行团（序列）有一张房间清单（block table）：“团里第 2 间房 = 905 号房”。客人（token）只知道自己是团里第 37 位：第 37 // 16 = 2 间房、第 37 % 16 = 5 张床，于是去 905 号房 5 号床（slot）。 房间按需分配、不必连号，团员增加时再开新房间：这就是 paged KV cache 没有碎片的原因。

padding token 怎么被跳过



> **类比 · 空房卡**
>
> 旅行团人数不固定，但大巴（CUDA graph）的座位数是固定的，空座位上放一张写着“-1”的空房卡。前台（kernel）看到空房卡就什么也不做，不会把任何东西放进 -1 号房间。

题目要求

在 `mine/l1_paged_kv_write.py` 中：

1. **`slot_mapping(block_table, seq_idx, positions, block_size)`**：纯 PyTorch、向量化（不准写 Python 循环），返回 int64 的 `[T]`。
2. **`reshape_and_cache_kernel`**：把 `key[t]`、`value[t]` 写到 `k_cache`、`v_cache` 的 slot 处；slot < 0 的 token 跳过。wrapper 已写好。

写 cache 是纯拷贝，结果必须**逐位相等**。

题干

mine/l1_paged_kv_write.py 47 行

```
"""L1 · Paged KV cache: slot mapping + cache write (vLLM's reshape_and_cache) — Y
OUR IMPLEMENTATION.

Cache layout (one tensor each for K and V):
    k_cache, v_cache : [num_blocks, block_size, num_kv_heads, head_dim]
A sequence's token at logical position p lives in physical block block_table[seq]
[p // block_size]
at offset p % block_size. Flattening (block, offset) gives a "slot":
    slot = physical_block * block_size + offset
and because the cache is contiguous, slot s starts at element s * num_kv_heads *
head_dim.
"""
import torch
import triton
import triton.language as tl


def slot_mapping(block_table: torch.Tensor, seq_idx: torch.Tensor, positions: tor
ch.Tensor,
                 block_size: int) -> torch.Tensor:
    """block_table [B, max_blocks] int32, seq_idx [T], positions [T] -> slots [T]
int64.
```

```
    Pure PyTorch, vectorised: no Python loop over tokens.
    """
    # TODO: logical block = pos // block_size; physical = block_table[seq, logica
l];
    #       slot = physical * block_size + pos % block_size  (int64, no Python lo
op)
    raise NotImplementedError


@triton.jit
def reshape_and_cache_kernel(key_ptr, value_ptr, k_cache_ptr, v_cache_ptr, slot_p
tr,
                             key_stride, value_stride, HD, BLOCK: tl.constexpr):
    # TODO: one program per token: load its slot (-1 = skip), copy the token's H*
D key and value
    #       values to k_cache / v_cache at element offset slot * HD
    pass


def reshape_and_cache(key: torch.Tensor, value: torch.Tensor, k_cache: torch.Tens
or,
                      v_cache: torch.Tensor, slots: torch.Tensor) -> None:
    """key/value [T, H, D] -> written in place into k_cache/v_cache [NB, BS, H,
D] at `slots`."""
    T, H, D = key.shape
    assert value.shape == key.shape
    assert k_cache.is_contiguous() and v_cache.is_contiguous()
    assert k_cache.shape[2:] == (H, D) and k_cache.dtype == key.dtype
    for t in (key, value):
        assert t.stride(2) == 1 and t.stride(1) == D, "each token's [H, D] must b
e contiguous"
    if T == 0:
        return
    HD = H * D
    reshape_and_cache_kernel[(T,)](key, value, k_cache, v_cache, slots.long(),
                                   key.stride(0), value.stride(0), HD, BLOCK=trit
on.next_power_of_2(HD))
```

验收

```
./check.sh l1
./check.sh l1 --bench
```

tests/test_l1_paged_kv_write.py 52 行

```
import torch
```

```
from common.env import DEVICE
from drill import load


def test_slot_mapping():
    m = load("l1_paged_kv_write")
    block_table = torch.tensor([[7, 2, 5], [0, 3, 0]], dtype=torch.int32)
    seq_idx = torch.tensor([0, 0, 0, 1, 1])
    positions = torch.tensor([0, 3, 4, 0, 5])
    # block_size 4: seq0 pos0 -> block 7 off 0; pos3 -> 7*4+3; pos4 -> block 2 of
f 0;
    #               seq1 pos0 -> block 0 off 0; pos5 -> block 3 off 1
    out = m.slot_mapping(block_table, seq_idx, positions, 4)
    assert out.dtype == torch.int64
    assert out.tolist() == [28, 31, 8, 0, 13]


def _reference_write(key, value, k_cache, v_cache, slots):
    kc, vc = k_cache.clone(), v_cache.clone()
    NB, BS, H, D = kc.shape
    valid = slots >= 0
    kc.view(NB * BS, H, D)[slots[valid]] = key[valid]
    vc.view(NB * BS, H, D)[slots[valid]] = value[valid]
    return kc, vc


def test_reshape_and_cache():
    m = load("l1_paged_kv_write")
    NB, BS, H, D, T = 6, 16, 2, 64, 37
    k_cache = torch.zeros(NB, BS, H, D, device=DEVICE, dtype=torch.float16)
    v_cache = torch.zeros_like(k_cache)
    key = torch.randn(T, H, D, device=DEVICE).half()
    value = torch.randn(T, H, D, device=DEVICE).half()
    slots = torch.randperm(NB * BS)[:T].to(DEVICE)
    slots[[3, 10]] = -1                                  # padding tokens must b
e skipped
    ref_k, ref_v = _reference_write(key, value, k_cache, v_cache, slots)
    m.reshape_and_cache(key, value, k_cache, v_cache, slots)
    assert torch.equal(k_cache, ref_k) and torch.equal(v_cache, ref_v)


def test_reshape_and_cache_strided_rows():
    m = load("l1_paged_kv_write")
    T, H, D = 9, 4, 32
    qkv = torch.randn(T, 3 * H, D, device=DEVICE)          # fused projection out
put
    key, value = qkv[:, H:2 * H], qkv[:, 2 * H:]            # views with row stri
de 3*H*D
    k_cache = torch.zeros(2, 16, H, D, device=DEVICE)
    v_cache = torch.zeros_like(k_cache)
```

```
    slots = torch.arange(T, device=DEVICE) * 3
    ref_k, ref_v = _reference_write(key, value, k_cache, v_cache, slots)
    m.reshape_and_cache(key, value, k_cache, v_cache, slots)
    assert torch.equal(k_cache, ref_k) and torch.equal(v_cache, ref_v)
```

提示

L1 slot_mapping 三行 方向（轻度剧透） 收起

`logical = positions // block_size` → `physical = block_table[seq_idx, logical]` → `physical * block_size + positions % block_size`。记得都转成 `long`。

L2 kernel 拆解与常见坑（中度剧透） 收起

- `t = tl.program_id(0)`，`slot = tl.load(slot_ptr + t)`。
- 源地址：`key_ptr + t * key_stride + offs`；目标地址：`k_cache_ptr + slot * HD + offs`。
- 跳过 padding：最直观的写法是 `mask = (offs < HD) & (slot >= 0)`。它在 GPU 上没问题，但 CPU 解释器不接受由标量和块组合出来的 mask。可以换一种等价写法：`n_valid = tl.where(slot >= 0, HD, 0)`，`mask = offs < n_valid`。

L3 kernel 全文 几乎是答案（重度剧透） 收起

```
t = tl.program_id(0).to(tl.int64)
slot = tl.load(slot_ptr + t)
offs = tl.arange(0, BLOCK)
mask = offs < tl.where(slot >= 0, HD, 0)
k = tl.load(key_ptr + t * key_stride + offs, mask=mask)
v = tl.load(value_ptr + t * value_stride + offs, mask=mask)
tl.store(k_cache_ptr + slot * HD + offs, k, mask=mask)
tl.store(v_cache_ptr + slot * HD + offs, v, mask=mask)
```

参考解

解 **solutions/l1_paged_kv_write.py** 先自己写完、跑过测试再看 收起

solutions/l1_paged_kv_write.py 62 行

```
"""L1 · Paged KV cache: slot mapping + cache write (vLLM's reshape_and_cache)
— reference solution.

Cache layout (one tensor each for K and V):
    k_cache, v_cache : [num_blocks, block_size, num_kv_heads, head_dim]
A sequence's token at logical position p lives in physical block block_table
[seq][p // block_size]
at offset p % block_size. Flattening (block, offset) gives a "slot":
    slot = physical_block * block_size + offset
and because the cache is contiguous, slot s starts at element s * num_kv_head
s * head_dim.
"""
import torch
import triton
import triton.language as tl


def slot_mapping(block_table: torch.Tensor, seq_idx: torch.Tensor, positions:
torch.Tensor,
                 block_size: int) -> torch.Tensor:
    """block_table [B, max_blocks] int32, seq_idx [T], positions [T] -> slots
[T] int64.

    Pure PyTorch, vectorised: no Python loop over tokens.
    """
    seq_idx = seq_idx.long()
    positions = positions.long()
    logical_block = positions // block_size
# which page of the sequence
    physical_block = block_table[seq_idx, logical_block].long()
# gather: [T]
    return physical_block * block_size + positions % block_size


@triton.jit
def reshape_and_cache_kernel(key_ptr, value_ptr, k_cache_ptr, v_cache_ptr, sl
ot_ptr,
                             key_stride, value_stride, HD, BLOCK: tl.constexp
r):
    t = tl.program_id(0).to(tl.int64)          # one program per new token
    slot = tl.load(slot_ptr + t)               # int64; -1 marks padding toke
ns to skip
```

```
    offs = tl.arange(0, BLOCK)                  # the token's H*D values, flat
tened
    # Padding token -> 0 valid elements, every lane masked off. (Written as o
ne comparison
    # rather than `(offs < HD) & (slot >= 0)`: same meaning, and the CPU inte
rpreter rejects
    # a mask built by and-ing a scalar with a block.)
    n_valid = tl.where(slot >= 0, HD, 0)
    mask = offs < n_valid

    # key/value rows may be strided (e.g. a view into a fused qkv tensor), bu
t each row's
    # [H, D] part is contiguous.
    k = tl.load(key_ptr + t * key_stride + offs, mask=mask)
    v = tl.load(value_ptr + t * value_stride + offs, mask=mask)
    # The cache is contiguous: slot s occupies elements [s * HD, (s + 1) * H
D).
    tl.store(k_cache_ptr + slot * HD + offs, k, mask=mask)
    tl.store(v_cache_ptr + slot * HD + offs, v, mask=mask)


def reshape_and_cache(key: torch.Tensor, value: torch.Tensor, k_cache: torch.
Tensor,
                      v_cache: torch.Tensor, slots: torch.Tensor) -> None:
    """key/value [T, H, D] -> written in place into k_cache/v_cache [NB, BS,
H, D] at `slots`."""
    T, H, D = key.shape
    assert value.shape == key.shape
    assert k_cache.is_contiguous() and v_cache.is_contiguous()
    assert k_cache.shape[2:] == (H, D) and k_cache.dtype == key.dtype
    for t in (key, value):
        assert t.stride(2) == 1 and t.stride(1) == D, "each token's [H, D] mu
st be contiguous"
    if T == 0:
        return
    HD = H * D
    reshape_and_cache_kernel[(T,)](key, value, k_cache, v_cache, slots.long
(),
                                   key.stride(0), value.stride(0), HD, BLOCK=
triton.next_power_of_2(HD))
```

测完之后

- 这是纯拷贝 kernel：有效带宽按“读 key + value、写两个 cache”计算。RTX 5090 实测：T = 8192、H = 8、D = 128 时 53.5 us · **1254 GB/s**（slot 是随机排列的，写入比顺序拷贝分散一些）。

- **FP8 KV cache**：在写入时量化（`cache = (k / scale).to(fp8)`），cache 容量翻倍，decode 读 KV 的字节数减半。改写 kernel 加一个 per-tensor 的 scale。
- vLLM 的 CUDA 版本区分了两种 cache layout：`[num_blocks, block_size, H, D]`（flash 布局，本题用的）和把 D 再切成 `x = 16 字节` 小块的旧布局（为 paged attention v1/v2 的向量化读取设计）。读源码时留意 `reshape_and_cache` 和 `reshape_and_cache_flash` 的区别。
- 为什么 block_size 一般是 16？太小：block table 变长，attention kernel 的间接寻址更多；太大：内部碎片（最后一个 block 的浪费）变多，前缀共享的粒度也更粗。

▼ 面试追问：paged KV cache 相比连续 KV cache 的代价是什么？

attention kernel 每读一个 page 都要先查 block table，多一次间接寻址；物理上不连续，也不能用一次大的连续拷贝。好处是几乎没有碎片、可以动态增长、可以在序列之间共享前缀 block（copy-on-write），并发量因此能提高好几倍。page 大小是这两者之间的权衡。

核心 · Track L · 推理 attention 从零写 · 第 11 题

L2 Paged decode attention + GQA

decode 阶段每个序列只有 1 个新 query，却要读完它整个历史的 KV：这是一个纯 memory-bound 的问题。GQA 让多个 query head 共用一份 KV，kernel 要利用这一点，把每个 page 只读一次。

60 min | 难度 难 | 你写 `mine/l2_paged_decode.py` | 验收 `./check.sh l2` | 参考解 `solutions/l2_paged_decode.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

prefill（T5）是 compute-bound 的：Q 有很多行，QK^T 是真正的矩阵乘。decode 时每个序列每个 head 只有一个 query 向量：

- 计算量：每个 KV token 约 $4 \cdot D$ FLOPs（点积 + 加权求和）
- 读取量：每个 KV token 要读 $2 \cdot D$ 个数（K 和 V）

算术强度约为 1，远低于硬件拐点，**时间 ≈ KV cache 字节数 / 带宽**。所以 decode 优化的核心问题是：**怎样把 KV 读得尽量少、尽量快**。

GQA（Grouped-Query Attention，LLaMA-2 70B、LLaMA-3、Qwen 等）：Hq 个 query head 共享 Hkv 个 KV head，$GROUP = Hq / Hkv$（常见是 4 或 8）。如果每个 query head 各开一个 program，同一份 KV 会被读 GROUP 次；如果一个 program 负责一个 KV head 下的全部 GROUP 个 query head，KV 只读一次，算术强度也随之提高 GROUP 倍。

源码与资料

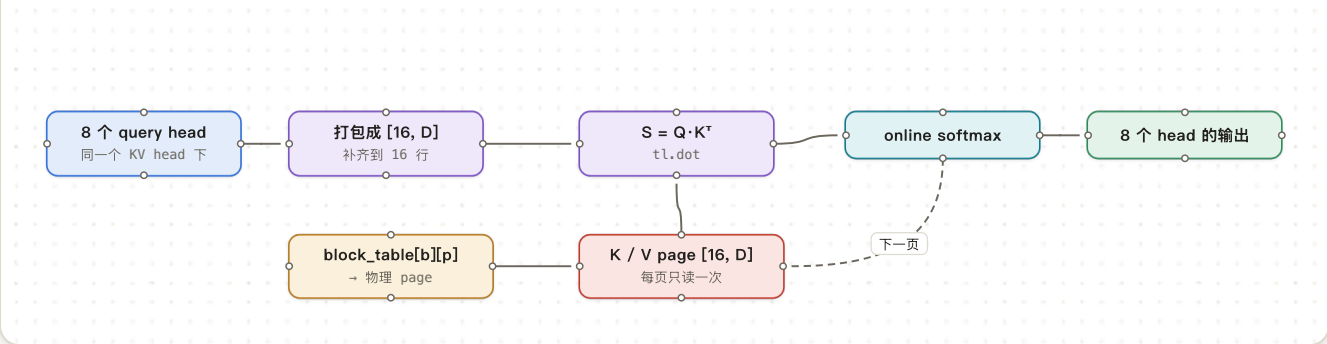
`flashinfer` › FlashInfer decode.cuh · `vllm` › vLLM Triton decode attention · `sglang` › SGLang Triton decode attention · GQA 论文

要用到的知识

- 输入：`q [B, Hq, D]`；`k_cache, v_cache [num_blocks, PAGE, Hkv, D]`；`block_tables [B, max_blocks]`；`context_lens [B]`。
- 一个 program = (序列 b, kv head)：`grid = (B, Hkv)`。
- 这个 program 的 query：`q_heads = kvh·GROUP + [0, GROUP)`，组成一个 `[GROUP, D]` 的小矩阵。`tl.dot` 要求 M ⩾ 16，所以补到 `GROUP_PAD = max(16, next_pow2(GROUP))` 行，多出的行填 0，最后不存。
- 按 page 循环：`phys = block_tables[b, p]`，读 `[PAGE, D]` 的 K、V。
- `tl.trans(k)`：`[PAGE, D] → [D, PAGE]`，然后 `s = tl.dot(q, tl.trans(k))`，得到 `[GROUP_PAD, PAGE]`。
- 最后一个 page 不满：`tok = p·PAGE + offs_t`，`tok >= ctx` 的位置分数设为 −inf。
- 跨 page 的 softmax：与 T5 完全相同的 online softmax。

图解与类比

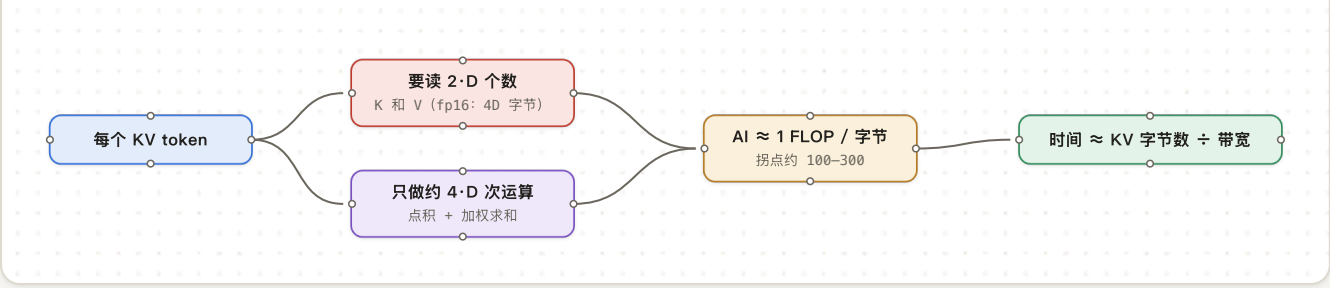
GQA decode：一个 KV head 服务一组 query head



类比 · 组团借书

8 个人（query head）要看同一本参考书（KV head）。各自去图书馆借一次，就要搬 8 趟同一本书；组团借一次、大家围着一起看（GQA 打包），只搬 1 趟。decode 的瓶颈正是“搬书”（读 KV），所以这一招直接省下 7/8 的时间。

decode 为什么是 memory-bound



类比 · 每页只读一个字

decode 时每读一个 KV token，只做几次乘加就扔掉了，就像为了查一个字把整本字典从书架上搬下来又放回去：时间都花在搬书上。所以一切优化都在减少搬书：GQA（几个人一起查）、KV 量化（字典印得更小）、paged（字典不必连号放）。

题目要求

wrapper 已写好（要求 PAGE 和 D 是 ⩾ 16 的 2 的幂，`context_len ≥ 1`）。实现 `paged_decode_kernel`：

- 支持 MHA（`Hq = Hkv`）、GQA（`8/2`）、MQA（`6/1`，GROUP 不是 2 的幂）。
- 支持 PAGE = 16、32；context 长度任意（1、17、100）。
- 每个 K / V page 在每个 program 中只读一次。
- fp16 下与 reference 的误差 ⩽ 1e-2。

题干

mine/l2_paged_decode.py 52 行

```
"""L2 · Paged decode attention with GQA — YOUR IMPLEMENTATION.

q [B, Hq, D], k/v_cache [num_blocks, PAGE, Hkv, D], block_tables [B, max_blocks]
int32,
context_lens [B] int32 -> out [B, Hq, D]. One program per (sequence, kv head).
The wrapper is done; implement paged_decode_kernel.
"""
import math

import torch
import triton
import triton.language as tl
```

```
@triton.jit
def paged_decode_kernel(Q, K_cache, V_cache, block_tables, context_lens, O, scal
e,
                        stride_qb, stride_qh,
                        stride_cb, stride_ct, stride_ch,          # cache stride
s: block, token, head (dim stride 1)
                        stride_btb,                               # block_tables
row stride
                        stride_ob, stride_oh,
                        GROUP: tl.constexpr, GROUP_PAD: tl.constexpr,
                        PAGE: tl.constexpr, HEAD_DIM: tl.constexpr):
    # TODO:
    # b, kvh, ctx; offs_g (GROUP_PAD), offs_t (PAGE), offs_d; q_heads = kvh * GRO
UP + offs_g
    # load q [GROUP_PAD, D] (padded rows 0)
    # for p in range(cdiv(ctx, PAGE)): phys = block_tables[b, p]; load k, v pages
[PAGE, D];
    #     s = dot(q, trans(k)) * scale; mask tokens >= ctx; online softmax update
    # store acc / l_i for the real heads
    pass


def paged_decode_attention(q: torch.Tensor, k_cache: torch.Tensor, v_cache: torc
h.Tensor,
                           block_tables: torch.Tensor, context_lens: torch.Tenso
r,
                           scale: float | None = None) -> torch.Tensor:
    B, Hq, D = q.shape
    NB, PAGE, Hkv, D2 = k_cache.shape
    assert D == D2 and v_cache.shape == k_cache.shape and Hq % Hkv == 0
    assert k_cache.is_contiguous() and v_cache.is_contiguous() and q.stride(-1) =
= 1
    assert PAGE >= 16 and PAGE & (PAGE - 1) == 0, "page size: a power of two >= 1
6 (tl.dot)"
    assert D >= 16 and D & (D - 1) == 0, "head_dim: a power of two >= 16"
    GROUP = Hq // Hkv
    GROUP_PAD = max(16, triton.next_power_of_2(GROUP))
    if scale is None:
        scale = 1.0 / math.sqrt(D)
    out = torch.empty_like(q)
    paged_decode_kernel[(B, Hkv)](
        q, k_cache, v_cache, block_tables, context_lens, out, scale,
        q.stride(0), q.stride(1),
        k_cache.stride(0), k_cache.stride(1), k_cache.stride(2),
        block_tables.stride(0),
        out.stride(0), out.stride(1),
        GROUP=GROUP, GROUP_PAD=GROUP_PAD, PAGE=PAGE, HEAD_DIM=D, num_warps=4)
    return out
```

验收

```
./check.sh l2
./check.sh l2 --bench       # 有效带宽：KV 字节数 / 时间
```

reference（`tests/_paged.py`）按 block table 把每个序列的 KV 拼回连续张量，用 `repeat_interleave` 展开 GQA，再做普通 softmax：

tests/_paged.py 35 行

```
"""Helpers shared by the paged-attention tests: build a random paged KV cache + a
reference."""
import torch


def make_paged(context_lens, num_kv_heads, head_dim, page, dtype, device, extra_b
locks=3, seed=0):
    g = torch.Generator().manual_seed(seed)
    pages_per_seq = [(c + page - 1) // page for c in context_lens]
    max_blocks = max(pages_per_seq)
    num_blocks = sum(pages_per_seq) + extra_blocks
    perm = torch.randperm(num_blocks, generator=g)          # physical blocks in
random order
    block_tables = torch.zeros(len(context_lens), max_blocks, dtype=torch.int32)
    used = 0
    for b, n in enumerate(pages_per_seq):
        block_tables[b, :n] = perm[used:used + n].to(torch.int32)
        used += n
    k_cache = torch.randn(num_blocks, page, num_kv_heads, head_dim, generator=g).
to(dtype)
    v_cache = torch.randn(num_blocks, page, num_kv_heads, head_dim, generator=g).
to(dtype)
    return (k_cache.to(device), v_cache.to(device), block_tables.to(device),
            torch.tensor(context_lens, dtype=torch.int32, device=device))


def reference_decode(q, k_cache, v_cache, block_tables, context_lens, scale):
    B, Hq, D = q.shape
    _, page, Hkv, _ = k_cache.shape
    out = torch.empty(B, Hq, D, dtype=torch.float32, device=q.device)
    for b in range(B):
        ctx = int(context_lens[b])
        blocks = block_tables[b, : (ctx + page - 1) // page].long()
        k = k_cache[blocks].reshape(-1, Hkv, D)[:ctx].float()          # [ctx, Hk
v, D]
        v = v_cache[blocks].reshape(-1, Hkv, D)[:ctx].float()
        k = k.repeat_interleave(Hq // Hkv, dim=1)                       # GQA: sh
are kv heads
```

```
        v = v.repeat_interleave(Hq // Hkv, dim=1)
        s = torch.einsum("hd,thd->ht", q[b].float(), k) * scale          # [Hq, ct
x]
        out[b] = torch.einsum("ht,thd->hd", s.softmax(-1), v)
    return out
```

tests/test_l2_paged_decode.py 22 行

```
import math

import pytest
import torch

from common.env import DEVICE
from drill import load
from tests._paged import make_paged, reference_decode


@pytest.mark.parametrize("Hq,Hkv", [(4, 4), (8, 2), (6, 1)])
@pytest.mark.parametrize("page", [16, 32])
def test_paged_decode(Hq, Hkv, page):
    m = load("l2_paged_decode")
    D = 64
    ctx = [1, 17, 100]
    k_cache, v_cache, bt, cl = make_paged(ctx, Hkv, D, page, torch.float16, DEVIC
E)
    q = torch.randn(len(ctx), Hq, D, device=DEVICE).half()
    out = m.paged_decode_attention(q, k_cache, v_cache, bt, cl)
    assert out.shape == q.shape and out.dtype == q.dtype
    ref = reference_decode(q, k_cache, v_cache, bt, cl, 1 / math.sqrt(D))
    torch.testing.assert_close(out.float(), ref, rtol=1e-2, atol=1e-2)
```

提示

L1 **和 T5 比，只有三处不同** 方向（轻度剧透） 收起

1. query 不是 `[BLOCK_M, D]` 的连续一块，而是同一个 kv head 下的 GROUP 个 head（补齐到 GROUP_PAD 行）。
2. key 的位置不是 `start_n + arange`，而是先查 block table 找到物理 page，再在 page 内偏移。
3. 每个序列的长度不同：循环上界是 `cdiv(context_lens[b], PAGE)`，最后一个 page 用 mask。

L2 **地址** 拆解与常见坑（中度剧透） 收起 ▾

```
q:      Q + b·stride_qb + q_heads[:, None]·stride_qh + offs_d[None, :]
k / v:  CACHE + phys·stride_cb + offs_t[:, None]·stride_ct + kvh·stride_ch + of
out:    O + b·stride_ob + q_heads[:, None]·stride_oh + offs_d[None, :]   (mask:
```

`phys` 要转成 int64：`num_blocks · stride_cb` 很容易超过 2^{31}。

L3 **循环体** 几乎是答案（重度剧透） 收起 ▾

```python
for p in range(0, tl.cdiv(ctx, PAGE)):
    phys = tl.load(block_tables + b * stride_btb + p).to(tl.int64)
    t_mask = (p * PAGE + offs_t) < ctx
    kv_off = phys * stride_cb + offs_t[:, None] * stride_ct + kvh * stride_ch +
    k = tl.load(K_cache + kv_off, mask=t_mask[:, None], other=0.0)
    v = tl.load(V_cache + kv_off, mask=t_mask[:, None], other=0.0)
    s = tl.where(t_mask[None, :], tl.dot(q, tl.trans(k)) * scale, float("-inf")
    m_new = tl.maximum(m_i, tl.max(s, axis=1))
    p_ = tl.exp(s - m_new[:, None])
    alpha = tl.exp(m_i - m_new)
    l_i = l_i * alpha + tl.sum(p_, axis=1)
    acc = tl.dot(p_.to(v.dtype), v, acc * alpha[:, None])
    m_i = m_new
```

参考解

解 **solutions/l2_paged_decode.py** 先自己写完、跑过测试再看 收起

solutions/l2_paged_decode.py 97 行

```
"""L2 · Paged decode attention with GQA — reference solution.

Decode step: each sequence has ONE new query token per head, and attends over its whole
history stored in the paged KV cache.
    q            [B, Hq, D]
    k/v_cache    [num_blocks, PAGE, Hkv, D]       (Hq = GROUP * Hkv)
    block_tables [B, max_blocks] int32, context_lens [B] int32 (>= 1)
    out          [B, Hq, D]

One program per (sequence, kv head). It handles the GROUP query heads that share this kv head
together, so every K/V page is read from memory once for the whole group — decode is
memory-bound, and this is the main saving GQA offers. The group is padded to >= 16 rows so
q @ k^T can use tl.dot (Tensor Cores need M >= 16).
"""
import math

import torch
import triton
import triton.language as tl


@triton.jit
def paged_decode_kernel(Q, K_cache, V_cache, block_tables, context_lens, O, scale,
                        stride_qb, stride_qh,
                        stride_cb, stride_ct, stride_ch,          # cache strides: block, token, head (dim stride 1)
                        stride_btb,                               # block_tables row stride
                        stride_ob, stride_oh,
                        GROUP: tl.constexpr, GROUP_PAD: tl.constexpr,
                        PAGE: tl.constexpr, HEAD_DIM: tl.constexpr):
    b = tl.program_id(0).to(tl.int64)
    kvh = tl.program_id(1).to(tl.int64)
    ctx = tl.load(context_lens + b)

    offs_g = tl.arange(0, GROUP_PAD)                  # query heads within the group (padded)
```

```
    offs_t = tl.arange(0, PAGE)                        # token offsets within a
page
    offs_d = tl.arange(0, HEAD_DIM)
    g_mask = offs_g < GROUP
    q_heads = kvh * GROUP + offs_g                     # query heads served by
this kv head

    q = tl.load(Q + b * stride_qb + q_heads[:, None] * stride_qh + offs_d[Non
e, :],
                mask=g_mask[:, None], other=0.0)       # [GROUP_PAD, D]; padded
rows are 0

    m_i = tl.full([GROUP_PAD], float("-inf"), dtype=tl.float32)
    l_i = tl.zeros([GROUP_PAD], dtype=tl.float32)
    acc = tl.zeros([GROUP_PAD, HEAD_DIM], dtype=tl.float32)

    num_pages = tl.cdiv(ctx, PAGE)
    for p in range(0, num_pages):
        # Indirection: logical page p of this sequence -> physical block in t
he cache.
        phys = tl.load(block_tables + b * stride_btb + p).to(tl.int64)
        tok = p * PAGE + offs_t
        t_mask = tok < ctx                             # the last page is usual
ly partial

        kv_off = phys * stride_cb + offs_t[:, None] * stride_ct + kvh * strid
e_ch + offs_d[None, :]
        k = tl.load(K_cache + kv_off, mask=t_mask[:, None], other=0.0)   # [P
AGE, D]
        v = tl.load(V_cache + kv_off, mask=t_mask[:, None], other=0.0)   # [P
AGE, D]

        s = tl.dot(q, tl.trans(k)) * scale             # [GROUP_PAD, PAGE] fp32
        s = tl.where(t_mask[None, :], s, float("-inf"))

        # Same online-softmax update as FlashAttention (T5). Page 0 always ha
s token 0, so
        # m_new is finite from the first iteration.
        m_new = tl.maximum(m_i, tl.max(s, axis=1))
        p_ = tl.exp(s - m_new[:, None])
        alpha = tl.exp(m_i - m_new)
        l_i = l_i * alpha + tl.sum(p_, axis=1)
        acc = acc * alpha[:, None]
        acc = tl.dot(p_.to(v.dtype), v, acc)
        m_i = m_new

    out = acc / l_i[:, None]
    tl.store(O + b * stride_ob + q_heads[:, None] * stride_oh + offs_d[None,
:],
             out.to(O.dtype.element_ty), mask=g_mask[:, None])
```

```
def paged_decode_attention(q: torch.Tensor, k_cache: torch.Tensor, v_cache: t
orch.Tensor,
                           block_tables: torch.Tensor, context_lens: torch.Te
nsor,
                           scale: float | None = None) -> torch.Tensor:
    B, Hq, D = q.shape
    NB, PAGE, Hkv, D2 = k_cache.shape
    assert D == D2 and v_cache.shape == k_cache.shape and Hq % Hkv == 0
    assert k_cache.is_contiguous() and v_cache.is_contiguous() and q.stride(-
1) == 1
    assert PAGE >= 16 and PAGE & (PAGE - 1) == 0, "page size: a power of two
>= 16 (tl.dot)"
    assert D >= 16 and D & (D - 1) == 0, "head_dim: a power of two >= 16"
    GROUP = Hq // Hkv
    GROUP_PAD = max(16, triton.next_power_of_2(GROUP))
    if scale is None:
        scale = 1.0 / math.sqrt(D)
    out = torch.empty_like(q)
    paged_decode_kernel[(B, Hkv)](
        q, k_cache, v_cache, block_tables, context_lens, out, scale,
        q.stride(0), q.stride(1),
        k_cache.stride(0), k_cache.stride(1), k_cache.stride(2),
        block_tables.stride(0),
        out.stride(0), out.stride(1),
        GROUP=GROUP, GROUP_PAD=GROUP_PAD, PAGE=PAGE, HEAD_DIM=D, num_warps=4)
    return out
```

测完之后

- 有效带宽 = `2 · B · ctx · Hkv · D · 2 字节 / 时间`。RTX 5090 实测（Hq = 32、Hkv = 8、D = 128、page 16）：

B · ctx	时间	有效带宽
64 × 1024	176 us	**1520 GB/s**（峰值的 85%）
8 × 4096	261 us	514 GB/s
1 × 32768	1839 us	**73 GB/s**（峰值的 4%）

- **B = 1、ctx = 32768** 时非常慢：只有 `B · Hkv = 8` 个 program，5090 的 170 个 SM 里只有 8 个在工作，每个 program 还要串行走 2048 个 page。这正是 L3 要解决的问题。

- 对比“每个 query head 一个 program”的写法（`grid = (B, Hq)`）：KV 读取量变成 GROUP 倍，看看带宽和时间的变化。
- vLLM 的 paged attention v1 正是这个结构（CUDA 手写，每个 warp 负责一部分 token）；v2 加上了分片，FlashInfer 的 `BatchDecodeWithPagedKVCacheWrapper` 则会根据 batch 和长度自动决定是否分片。

▼ 面试追问：为什么 decode attention 用不满 Tensor Core？GQA 有什么帮助？

每个 head 只有 1 个 query，$q @ k^T$ 是矩阵乘向量，Tensor Core 需要至少 16 行的矩阵块。把同一 KV head 下的 GROUP 个 query 打包成一个矩阵，可以凑出更多行（GROUP = 8 时补到 16，有效利用率是 50%），更重要的是 KV 只读一次，算术强度提升 GROUP 倍。投机解码的验证阶段每个序列有多个 query，也会利用同样的打包思路。

核心 · Track L · 推理 attention 从零写 · 第 12 题

L3 Split-KV decode（Flash-Decoding）+ LSE 合并

batch 小、上下文长时，L2 只能启动个位数的 program，GPU 大部分时间在空转。把每个序列的上下文再切成 NUM_SPLITS 段并行算，最后用 log-sum-exp 精确合并：这就是 Flash-Decoding。

60 min | 难度 难 | 你写 `mine/l3_split_kv_decode.py` | 验收 `./check.sh l3` | 参考解 `solutions/l3_split_kv_decode.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

L2 的 program 数 = `B · Hkv`。B = 1、Hkv = 8 时只有 8 个 program，而 H100 有 132 个 SM。每个 program 还要串行走完 32K 个 token：带宽利用率可能只有 10%。

Flash-Decoding / split-KV：沿上下文维度再切一刀。

- **第一步**（`grid = (B, Hkv, NUM_SPLITS)`）：每个 program 只处理自己那一段 page，得到这一段内归一化好的输出 `o_s`，以及这一段的 **log-sum-exp** `lse_s = m_s + log(l_s)`。
- **第二步**（`grid = (B, Hq)`）：合并。

```
M   = max_s lse_s
w_s = exp(lse_s - M)
out = Σ_s w_s · o_s / Σ_s w_s
```

为什么这样合并是精确的：`lse_s = log Σ_{j∈s} exp(score_j)` 正是这一段的 softmax 分母取对数，所以 `exp(lse_s)` 就是这一段在整体分母中所占的权重。减去 M 只是为了防止溢出，和 T2 中减去 max 的原理相同。

这个合并操作在很多地方都会出现：FlashInfer 的 `merge_state`、vLLM 的 `merge_attn_states`（前缀缓存 / cascade attention 中，把对共享前缀和私有后缀分别做的 attention 合并起来）、ring attention 等。

源码与资料

Flash-Decoding for long-context inference ·

`flashinfer › FlashInfer scheduler.cuh（分片决策）` ·

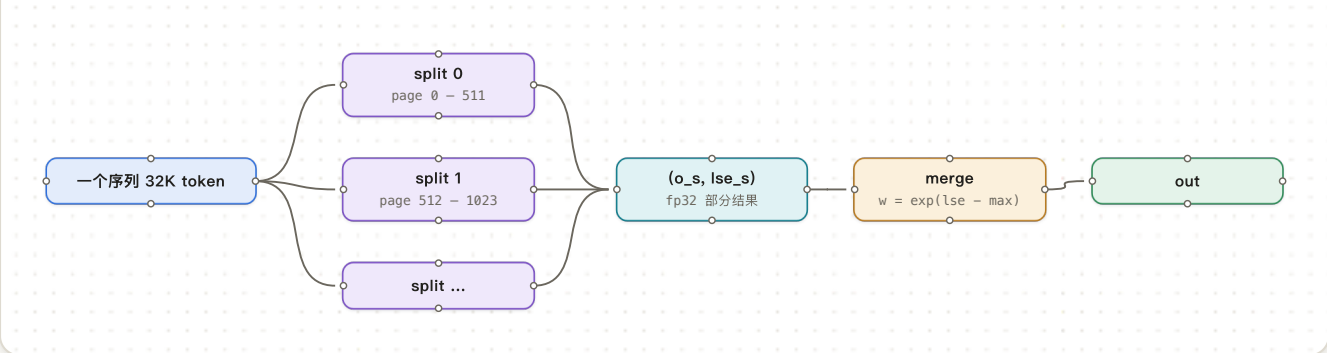
`flashinfer › FlashInfer cascade.cuh（合并）` · `vllm › vLLM Triton decode（stage1 / stage2）`

要用到的知识

- 每个 split 负责的 page 范围：`pages_per_split = cdiv(num_pages, NUM_SPLITS)`，`[split·pps, min((split+1)·pps, num_pages))`。
- **空分片**：短序列的后几个 split 可能分不到任何 page。此时 `l = 0`、`m = -inf`，必须写出 `o = 0`、`lse = -inf`，合并时权重为 `exp(-inf) = 0`。不能做 `0/0`，也不能做 `log(0)` 再相加。
- 中间结果用 fp32 存储：`o_part [B, Hq, NUM_SPLITS, D]`，`lse [B, Hq, NUM_SPLITS]`。
- 合并 kernel 一次读入所有分片：`SPLIT_PAD = next_pow2(NUM_SPLITS)`，多出的分片 `lse` 填 −inf。

图解与类比

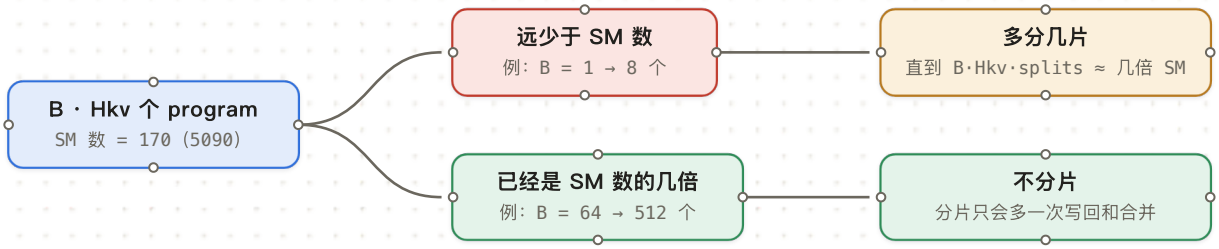
split-KV：沿上下文切开，再精确合并



类比 · 64 个人分头读一本书

一个人读 1000 页太慢，就分给 64 个人各读一段。每人交两样东西：这一段的摘要（o_s），以及“这一段占全书多少分量”（lse_s）。主编按分量加权合并摘要。 只交摘要不交分量，主编无法知道哪段更重要，也就没法正确合并；空的分段（短序列分不到页）分量是 0（lse = −∞），直接忽略。

分几片合适?



> **类比 · 搬家公司派几个人**
>
> 只搬一个书柜（batch 小），派一个人慢慢搬，其他 169 个工人在旁边看着；把书柜拆成 64 格分给 64 个人同时搬，再在新家组装（合并），快得多。可如果本来就要搬 512 个书柜（batch 大），每人已经有活干，再拆只会多出组装的功夫。

题目要求

wrapper 已写好。实现两个 kernel：

1. **`split_kv_partial_kernel`**：和 L2 相同的循环，但只处理本 split 的 page 范围；写出 `o_part` 和 `lse`，并正确处理空分片。
2. **`merge_kernel`**：按上面的公式合并，输出 `[B, Hq, D]`。

`num_splits` 取 1、3、4、8 时结果都要与 reference 一致，并且不能出现 NaN（测试中有 `context_len = 1` 的序列，它的大部分分片都是空的）。

题干

mine/l3_split_kv_decode.py 69 行

```
"""L3 · Split-KV decode attention (Flash-Decoding) + log-sum-exp merge — YOUR IMP
LEMENTATION.

Pass 1: each (seq, kv head, split) program -> partial output + LSE. Pass 2: merg
e.
The wrapper is done; implement both kernels.
"""
import math

import torch
import triton
```

```
import triton.language as tl


@triton.jit
def split_kv_partial_kernel(Q, K_cache, V_cache, block_tables, context_lens, O_pa
rt, LSE, scale,
                            stride_qb, stride_qh,
                            stride_cb, stride_ct, stride_ch,
                            stride_btb,
                            stride_pb, stride_ph, stride_ps,      # O_part [B, H
q, NUM_SPLITS, D] fp32
                            stride_lb, stride_lh,                 # LSE    [B, H
q, NUM_SPLITS]    fp32
                            NUM_SPLITS: tl.constexpr,
                            GROUP: tl.constexpr, GROUP_PAD: tl.constexpr,
                            PAGE: tl.constexpr, HEAD_DIM: tl.constexpr):
    # TODO: like L2, but only over pages [p_start, p_end) of this split;
    # write o = acc / l_i (0 for an empty split) and lse = m_i + log(l_i) (-inf f
or an empty split)
    pass


@triton.jit
def merge_kernel(O_part, LSE, O,
                 stride_pb, stride_ph, stride_ps, stride_lb, stride_lh, stride_o
b, stride_oh,
                 NUM_SPLITS: tl.constexpr, SPLIT_PAD: tl.constexpr, HEAD_DIM: tl.
constexpr):
    # TODO: load the NUM_SPLITS lse values (pad with -inf) and partial outputs;
    # w = exp(lse - max(lse)); out = sum(w * o) / sum(w)
    pass


def split_kv_decode_attention(q: torch.Tensor, k_cache: torch.Tensor, v_cache: to
rch.Tensor,
                              block_tables: torch.Tensor, context_lens: torch.Ten
sor,
                              num_splits: int = 4, scale: float | None = None) ->
torch.Tensor:
    B, Hq, D = q.shape
    NB, PAGE, Hkv, D2 = k_cache.shape
    assert D == D2 and v_cache.shape == k_cache.shape and Hq % Hkv == 0
    assert k_cache.is_contiguous() and v_cache.is_contiguous() and q.stride(-1) =
= 1
    assert PAGE >= 16 and PAGE & (PAGE - 1) == 0 and D >= 16 and D & (D - 1) == 0
    assert num_splits >= 1
    GROUP = Hq // Hkv
    GROUP_PAD = max(16, triton.next_power_of_2(GROUP))
    if scale is None:
        scale = 1.0 / math.sqrt(D)
```

```
    # Partial results stay in fp32: they are combined again, so keep the extra pr
ecision.
    o_part = torch.empty((B, Hq, num_splits, D), dtype=torch.float32, device=q.de
vice)
    lse = torch.empty((B, Hq, num_splits), dtype=torch.float32, device=q.device)
    split_kv_partial_kernel[(B, Hkv, num_splits)](
        q, k_cache, v_cache, block_tables, context_lens, o_part, lse, scale,
        q.stride(0), q.stride(1),
        k_cache.stride(0), k_cache.stride(1), k_cache.stride(2),
        block_tables.stride(0),
        o_part.stride(0), o_part.stride(1), o_part.stride(2),
        lse.stride(0), lse.stride(1),
        NUM_SPLITS=num_splits, GROUP=GROUP, GROUP_PAD=GROUP_PAD, PAGE=PAGE, HEAD_
DIM=D, num_warps=4)

    out = torch.empty_like(q)
    merge_kernel[(B, Hq)](
        o_part, lse, out,
        o_part.stride(0), o_part.stride(1), o_part.stride(2), lse.stride(0), lse.
stride(1),
        out.stride(0), out.stride(1),
        NUM_SPLITS=num_splits, SPLIT_PAD=max(2, triton.next_power_of_2(num_split
s)), HEAD_DIM=D, num_warps=1)
    return out
```

验收

```
./check.sh l3
./check.sh l3 --bench       # L2 vs L3 (num_splits = 4 / 16 / 64)，三种 (B, ctx) 组合
```

tests/test_l3_split_kv_decode.py 22 行

```
import math

import pytest
import torch

from common.env import DEVICE
from drill import load
from tests._paged import make_paged, reference_decode


@pytest.mark.parametrize("num_splits", [1, 3, 4, 8])
def test_split_kv_decode(num_splits):
    m = load("l3_split_kv_decode")
    Hq, Hkv, D, page = 8, 2, 64, 16
```

```
    ctx = [1, 20, 150]                  # seq 0 has 1 page: most of its split
s are empty
    k_cache, v_cache, bt, cl = make_paged(ctx, Hkv, D, page, torch.float16, DEVIC
E)
    q = torch.randn(len(ctx), Hq, D, device=DEVICE).half()
    out = m.split_kv_decode_attention(q, k_cache, v_cache, bt, cl, num_splits=num
_splits)
    assert out.shape == q.shape and out.dtype == q.dtype
    assert torch.isfinite(out.float()).all(), "empty splits must not produce NaN"
    ref = reference_decode(q, k_cache, v_cache, bt, cl, 1 / math.sqrt(D))
    torch.testing.assert_close(out.float(), ref, rtol=1e-2, atol=1e-2)
```

提示

L1 先写 merge，再改 L2 方向（轻度剧透） 收起

merge 是一个独立的小 kernel，可以先用 PyTorch 写出等价代码，验证合并公式：

```
M = lse.max(-1, keepdim=True)                       # [B, Hq, 1]
w = torch.exp(lse - M)                              # [B, Hq, S]
out = (w[..., None] * o_part).sum(-2) / w.sum(-1, keepdim=True)
```

然后把 L2 的 kernel 复制过来，改循环范围和结尾部分。

L2 空分片的处理 拆解与常见坑（中度剧透） 收起

```
has = l_i > 0
o   = acc / tl.where(has, l_i, 1.0)[:, None]                    # 空分片:
lse = tl.where(has, m_i + tl.log(tl.where(has, l_i, 1.0)), float("-inf"))
```

`range(p_start, p_end)` 在 `p_start >= p_end` 时一次也不执行，所以 `m_i`、`l_i`、`acc` 保持初始值。

L3 **merge kernel** 几乎是答案（重度剧透） 收起 ▾

```
lse = tl.load(LSE + b * stride_lb + h * stride_lh + offs_s, mask=s_mask, other=
w = tl.exp(lse - tl.max(lse, axis=0))
o = tl.load(O_part + b * stride_pb + h * stride_ph + offs_s[:, None] * stride_p
            mask=s_mask[:, None], other=0.0)
out = tl.sum(w[:, None] * o, axis=0) / tl.sum(w, axis=0)
```

参考解

解 **solutions/l3_split_kv_decode.py** 先自己写完、跑过测试再看 收起

solutions/l3_split_kv_decode.py 132 行

```
"""L3 · Split-KV decode attention (Flash-Decoding) + log-sum-exp merge — refe
rence solution.

L2 launches B * Hkv programs. With a small batch and a long context that is f
ar fewer
programs than the GPU has SMs, and each one loops over thousands of tokens: t
he GPU sits
mostly idle. Split-KV also splits the context of each sequence into NUM_SPLIT
S ranges:

  pass 1 (grid B x Hkv x NUM_SPLITS): attention over one range of pages -> a
partial output
          o_s (already normalised within the range) and its log-sum-exp lse_s
= m_s + log(l_s)
  pass 2 (grid B x Hq): merge the partials exactly:
          w_s = exp(lse_s - max_s lse_s),  out = sum_s w_s * o_s / sum_s w_s
An empty range (short sequences) gets lse = -inf and weight 0.
"""
import math

import torch
import triton
import triton.language as tl


@triton.jit
def split_kv_partial_kernel(Q, K_cache, V_cache, block_tables, context_lens,
O_part, LSE, scale,
                            stride_qb, stride_qh,
                            stride_cb, stride_ct, stride_ch,
                            stride_btb,
                            stride_pb, stride_ph, stride_ps,      # O_part
[B, Hq, NUM_SPLITS, D] fp32
                            stride_lb, stride_lh,                 # LSE
[B, Hq, NUM_SPLITS]    fp32
                            NUM_SPLITS: tl.constexpr,
                            GROUP: tl.constexpr, GROUP_PAD: tl.constexpr,
                            PAGE: tl.constexpr, HEAD_DIM: tl.constexpr):
    b = tl.program_id(0).to(tl.int64)
    kvh = tl.program_id(1).to(tl.int64)
    split = tl.program_id(2)
    ctx = tl.load(context_lens + b)
```

```
    # This program's contiguous range of pages: [p_start, p_end).
    num_pages = tl.cdiv(ctx, PAGE)
    pages_per_split = tl.cdiv(num_pages, NUM_SPLITS)
    p_start = split * pages_per_split
    p_end = tl.minimum(p_start + pages_per_split, num_pages)   # may be <= p_
start: empty split

    offs_g = tl.arange(0, GROUP_PAD)
    offs_t = tl.arange(0, PAGE)
    offs_d = tl.arange(0, HEAD_DIM)
    g_mask = offs_g < GROUP
    q_heads = kvh * GROUP + offs_g
    q = tl.load(Q + b * stride_qb + q_heads[:, None] * stride_qh + offs_d[Non
e, :],
                mask=g_mask[:, None], other=0.0)

    m_i = tl.full([GROUP_PAD], float("-inf"), dtype=tl.float32)
    l_i = tl.zeros([GROUP_PAD], dtype=tl.float32)
    acc = tl.zeros([GROUP_PAD, HEAD_DIM], dtype=tl.float32)

    for p in range(p_start, p_end):                  # identical body to L2
        phys = tl.load(block_tables + b * stride_btb + p).to(tl.int64)
        tok = p * PAGE + offs_t
        t_mask = tok < ctx
        kv_off = phys * stride_cb + offs_t[:, None] * stride_ct + kvh * strid
e_ch + offs_d[None, :]
        k = tl.load(K_cache + kv_off, mask=t_mask[:, None], other=0.0)
        v = tl.load(V_cache + kv_off, mask=t_mask[:, None], other=0.0)
        s = tl.dot(q, tl.trans(k)) * scale
        s = tl.where(t_mask[None, :], s, float("-inf"))
        m_new = tl.maximum(m_i, tl.max(s, axis=1))
        p_ = tl.exp(s - m_new[:, None])
        alpha = tl.exp(m_i - m_new)
        l_i = l_i * alpha + tl.sum(p_, axis=1)
        acc = acc * alpha[:, None]
        acc = tl.dot(p_.to(v.dtype), v, acc)
        m_i = m_new

    # Finalise this range. An empty range has l_i == 0: write o = 0 and lse =
-inf so the
    # merge gives it weight exp(-inf) = 0 (and never divide by zero).
    has = l_i > 0
    o = acc / tl.where(has, l_i, 1.0)[:, None]
    lse = tl.where(has, m_i + tl.log(tl.where(has, l_i, 1.0)), float("-inf"))

    tl.store(O_part + b * stride_pb + q_heads[:, None] * stride_ph + split *
stride_ps + offs_d[None, :],
             o, mask=g_mask[:, None])
    tl.store(LSE + b * stride_lb + q_heads * stride_lh + split, lse, mask=g_m
```

```
ask)


@triton.jit
def merge_kernel(O_part, LSE, O,
                 stride_pb, stride_ph, stride_ps, stride_lb, stride_lh, strid
e_ob, stride_oh,
                 NUM_SPLITS: tl.constexpr, SPLIT_PAD: tl.constexpr, HEAD_DIM:
tl.constexpr):
    b = tl.program_id(0).to(tl.int64)
    h = tl.program_id(1).to(tl.int64)
    offs_s = tl.arange(0, SPLIT_PAD)
    offs_d = tl.arange(0, HEAD_DIM)
    s_mask = offs_s < NUM_SPLITS

    lse = tl.load(LSE + b * stride_lb + h * stride_lh + offs_s, mask=s_mask,
other=float("-inf"))
    m = tl.max(lse, axis=0)                  # finite: context_len >= 1, so s
ome split is non-empty
    w = tl.exp(lse - m)                      # [SPLIT_PAD]; empty or padded s
plits -> 0
    o = tl.load(O_part + b * stride_pb + h * stride_ph + offs_s[:, None] * st
ride_ps + offs_d[None, :],
                mask=s_mask[:, None], other=0.0)                      # [SPL
IT_PAD, D]
    out = tl.sum(w[:, None] * o, axis=0) / tl.sum(w, axis=0)
    tl.store(O + b * stride_ob + h * stride_oh + offs_d, out.to(O.dtype.eleme
nt_ty))


def split_kv_decode_attention(q: torch.Tensor, k_cache: torch.Tensor, v_cach
e: torch.Tensor,
                              block_tables: torch.Tensor, context_lens: torc
h.Tensor,
                              num_splits: int = 4, scale: float | None = Non
e) -> torch.Tensor:
    B, Hq, D = q.shape
    NB, PAGE, Hkv, D2 = k_cache.shape
    assert D == D2 and v_cache.shape == k_cache.shape and Hq % Hkv == 0
    assert k_cache.is_contiguous() and v_cache.is_contiguous() and q.stride(-
1) == 1
    assert PAGE >= 16 and PAGE & (PAGE - 1) == 0 and D >= 16 and D & (D - 1)
== 0
    assert num_splits >= 1
    GROUP = Hq // Hkv
    GROUP_PAD = max(16, triton.next_power_of_2(GROUP))
    if scale is None:
        scale = 1.0 / math.sqrt(D)

    # Partial results stay in fp32: they are combined again, so keep the extr
```

```
a precision.
    o_part = torch.empty((B, Hq, num_splits, D), dtype=torch.float32, device=
q.device)
    lse = torch.empty((B, Hq, num_splits), dtype=torch.float32, device=q.devi
ce)
    split_kv_partial_kernel[(B, Hkv, num_splits)](
        q, k_cache, v_cache, block_tables, context_lens, o_part, lse, scale,
        q.stride(0), q.stride(1),
        k_cache.stride(0), k_cache.stride(1), k_cache.stride(2),
        block_tables.stride(0),
        o_part.stride(0), o_part.stride(1), o_part.stride(2),
        lse.stride(0), lse.stride(1),
        NUM_SPLITS=num_splits, GROUP=GROUP, GROUP_PAD=GROUP_PAD, PAGE=PAGE, H
EAD_DIM=D, num_warps=4)

    out = torch.empty_like(q)
    merge_kernel[(B, Hq)](
        o_part, lse, out,
        o_part.stride(0), o_part.stride(1), o_part.stride(2), lse.stride(0),
lse.stride(1),
        out.stride(0), out.stride(1),
        NUM_SPLITS=num_splits, SPLIT_PAD=max(2, triton.next_power_of_2(num_sp
lits)), HEAD_DIM=D, num_warps=1)
    return out
```

测完之后

RTX 5090 实测（Hq = 32、Hkv = 8、D = 128，单位 us，括号内为有效带宽 GB/s）：

B × ctx	L2（不分片）	splits = 4	splits = 16	splits = 64
1 × 32768	1838（73）	492（273）	155（865）	**96（1394）**
8 × 4096	261（514）	103（1309）	**95（1420）**	107（1257）
64 × 1024	**176（1521）**	186（1440）	204（1315）	262（1024）

- **B = 1、ctx = 32K**：分 64 片后快了 **19 倍**，带宽从峰值的 4% 升到 78%。
- **B = 64、ctx = 1K**：分片反而更慢：program 数本来就够多，额外的中间结果写回和第二个 kernel 都是纯开销。
- 每一行的最佳分片数都不一样，最优点大约在 `B · Hkv · splits` 达到 SM 数的几倍时。
- 所以真实系统会**根据 B、ctx、SM 数来决定 NUM_SPLITS**：目标是 `B · Hkv · NUM_SPLITS` 大约是 SM 数的若干倍，并且每个分片至少有若干个 page。FlashInfer

decode 的 `plan()` 阶段做的就是这件事（它还能让不同长度的序列分到不同数量的分片）。

- 把 merge 融合进第一个 kernel：最后完成的那个分片负责合并（需要一个原子计数器）。省掉一次 launch，代价是实现更复杂。

▼ 面试追问：只给你每个分片的输出 o_s，不给 lse，能合并吗?

不能。合并需要知道每个分片的 softmax 分母（`exp(lse_s)`）在整体中所占的比例。只有 o_s 时，分片之间的相对权重已经丢失了：两个分片的 o_s 完全相同，背后的分母可能差几个数量级。所以凡是需要“分段计算 attention 再合并”的场景（split-KV、cascade、ring attention、chunked prefill 的跨 chunk 合并），中间结果都要同时保存 o 和 lse（或者等价地保存 m 和 l）。

Track F · FlashInfer

对着 FlashInfer 的真实 API 实现并逐数值对拍，最后进入它的源码和贡献流程。

核心 · Track F · FlashInfer · 第 13 题

F1 FlashInfer 分页格式（CSR）+ plan / run decode

第一次用 FlashInfer：把 L 系列的 block table 换成 FlashInfer 的 CSR 分页格式，理解 plan / run 两段式，再拿官方 decode 和你自己的 L2 / L3 比一比。

40 min | 难度 易 | 你写 `mine/f1_flashinfer_paged.py` | 验收 `./check.sh f1` | 参考解 `solutions/f1_flashinfer_paged.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

FlashInfer 是 vLLM、SGLang、TensorRT-LLM、MLC 等框架共用的 attention / sampling / GEMM / MoE kernel 库。它的大部分接口都遵循两个约定，这一题先把它们吃透：

1. **分页 KV 用 CSR 格式描述**，不是 vLLM 那种补齐到最长序列的二维 block table。不同序列的 page 数相差很大（1 个对 2000 个）时，CSR 不浪费空间，也不需要关心 max_blocks。
2. **plan / run 分离**：`plan()` 在 CPU 上根据这一批序列的长度做调度，决定要不要 split-KV、分几段、每个 CTA 处理哪些 page，然后把结果拷到 GPU 的 workspace 里；`run()` 只负责启动 kernel。一个 batch 的形状确定后，plan 只做一次，所有层的 `run()` 都复用它，而且 `run()` 可以被 CUDA graph 捕获。

源码与资料

FlashInfer 论文（MLSys 2025） · FlashInfer 文档 · attention / decode API 文档 · `flashinfer › BatchDecodeWithPagedKVCacheWrapper` · `flashinfer › batch_decode.cu` · `flashinfer › scheduler.cuh` · `flashinfer › decode.cuh` · `flashinfer › test_batch_decode_kernels.py` · `flashinfer › bench_batch_decode.py` · `vllm › vLLM 如何调用 plan / run` · `sglang › SGLang 如何调用`

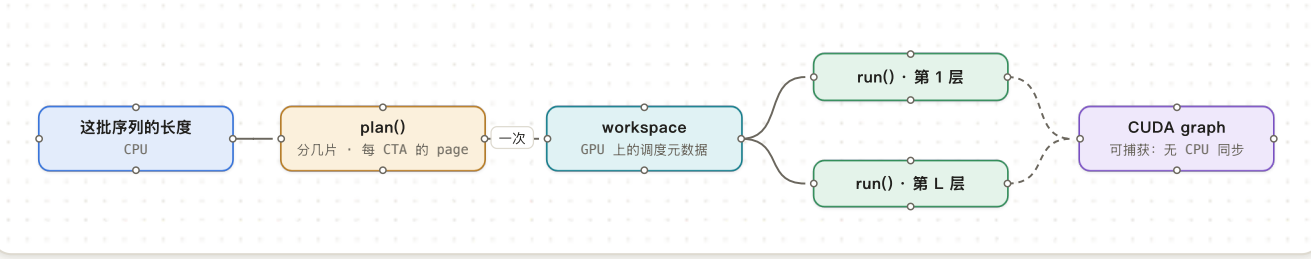
要用到的知识

```
indptr        [B + 1] int32   序列 b 的 page 是 indices[indptr[b] : indptr[b+1]]
indices       [nnz]   int32   各序列的物理 page 号，按序列依次拼接
last_page_len [B]     int32   每个序列最后一个 page 用了几个 token，取值 1 … page_size
context_len[b] = (indptr[b+1] − indptr[b] − 1) · page_size + last_page_len[b]
```

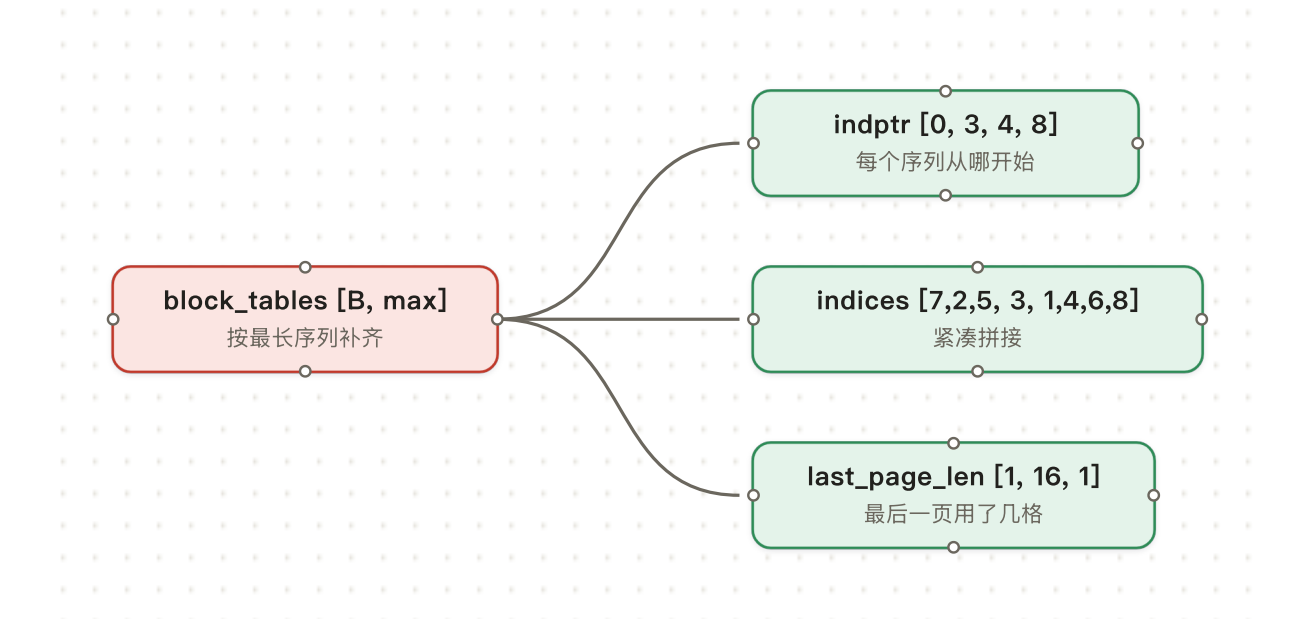
- 布尔索引 `block_tables[mask]` 按行主序取元素，正好保持“每个序列的 page 连续且有序”。
- 本仓库的 cache 形状 `[num_pages, page_size, H_kv, D]` 就是 FlashInfer 的 **NHD** 布局（另一种是 HND：`[num_pages, H_kv, page_size, D]`）。把 `(k_cache, v_cache)` 作为一个 tuple 传给 `run()`。
- `run(..., return_lse=True)` 会同时返回 LSE。**FlashInfer 的 LSE 以 2 为底**：`lse = log₂ Σ exp(score)`，等于自然对数 LSE ÷ ln 2。F2 要用到这一点。

图解与类比

plan 一次，run 很多次



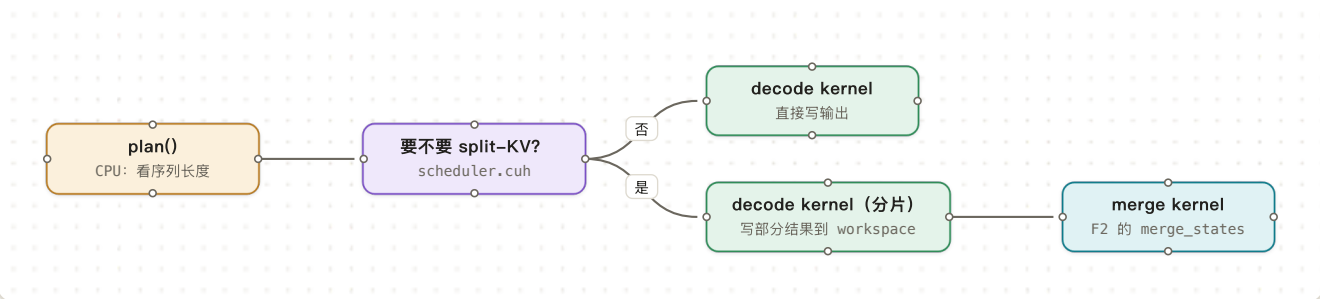
block table → CSR



类比 · 婚宴排座

开席前把座位表排好一次（`plan()`），之后每上一道菜（每一层的 `run()`）服务员都照同一张表走，不用每道菜重新排座。 CSR 是不给每桌都摆 20 把椅子（按最长序列补齐），而是按实际人数紧凑排成一长排，再附一张“第几桌从第几号椅子开始”的索引（indptr）。

BatchDecodeWithPagedKVCacheWrapper 内部



类比 · 把座位表交给服务员

`plan()` 排好的座位表不是放在前台（CPU），而是复印一份交给每位服务员（拷到 GPU 的 workspace）。这样上菜时（`run()`）服务员不用回前台问，整个上菜过程可以被提前“录像”（CUDA graph），以后每一桌照着录像回放。

题目要求

在 `mine/f1_flashinfer_paged.py` 中实现：

1. `block_table_to_csr(block_tables, context_lens, page_size) -> (indptr, indices, last_page_len)`：向量化，不准对序列写 Python 循环；全部是 int32。
2. `csr_to_block_table(indptr, indices, last_page_len, page_size) -> (block_tables, context_lens)`：它的逆变换，block table 用 0 补齐。
3. `flashinfer_decode(q, k_cache, v_cache, block_tables, context_lens, workspace=None, return_lse=False)`：用 `BatchDecodeWithPagedKVCacheWrapper` 完成 decode，输入与 L2 相同。

前两个函数在 CPU 上就能验收；第三个需要 GPU 和 `pip install flashinfer-python`。测试会把它的输出和 L 系列的 reference 对拍，并检查 LSE 确实以 2 为底。

题干

mine/f1_flashinfer_paged.py 34 行

```
"""F1 · FlashInfer paged-KV CSR + plan/run decode — YOUR IMPLEMENTATION.

See the drill page for the spec; the reference lives in solutions/."""
import torch


def block_table_to_csr(block_tables: torch.Tensor, context_lens: torch.Tensor, pa
ge_size: int):
    """Vectorised (no Python loop over sequences). Requires context_lens >= 1."""
    # TODO: num_pages = ceil(ctx / page_size); indptr = [0, cumsum(num_pages)] (i
nt32)
    # TODO: indices = block_tables entries with column < num_pages[b], row-major
(boolean mask)
    # TODO: last_page_len = ctx - (num_pages - 1) * page_size
    # TODO: return indptr, indices, last_page_len
    raise NotImplementedError


def csr_to_block_table(indptr: torch.Tensor, indices: torch.Tensor, last_page_le
n: torch.Tensor,
                       page_size: int):
    """Inverse of block_table_to_csr -> (block_tables padded with 0, context_len
s)."""
    # TODO: num_pages = indptr[1:] - indptr[:-1]; scatter indices back into a zer
o-padded [B, max] table
    # TODO: context_lens = (num_pages - 1) * page_size + last_page_len
    raise NotImplementedError


def flashinfer_decode(q, k_cache, v_cache, block_tables, context_lens, workspace=
None, return_lse=False):
    """Run FlashInfer's batch decode on the L2-style inputs.

    q [B, Hq, D]; k_cache/v_cache [num_pages, page_size, Hkv, D] ("NHD" layout, p
assed as a tuple).
    plan() does the CPU-side scheduling once per batch shape; run() launches the
kernel and can
    be called again (e.g. once per layer) with the same plan.
    """
    # TODO: workspace (128 MB uint8) -> flashinfer.BatchDecodeWithPagedKVCacheWra
pper(workspace, kv_layout='NHD')
    # TODO: wrapper.plan(indptr, indices, last, Hq, Hkv, D, page_size, q_data_typ
e=..., kv_data_type=...)
    # TODO: return wrapper.run(q, (k_cache, v_cache), return_lse=return_lse)
    raise NotImplementedError
```

验收

```
./check.sh f1               # CSR 部分 CPU 可测；FlashInfer 部分需要 GPU
./check.sh f1 --bench       # FlashInfer decode vs 你的 L2
```

tests/test_f1_flashinfer_paged.py 56 行

```python
import math

import pytest
import torch

from common.env import DEVICE, HAS_FLASHINFER
from drill import load
from tests._paged import make_paged, reference_decode


def test_block_table_to_csr_by_hand():
    m = load("f1_flashinfer_paged")
    bt = torch.tensor([[7, 2, 5, 0], [3, 0, 0, 0], [1, 4, 6, 8]], dtype=torch.int
32)
    ctx = torch.tensor([33, 16, 49], dtype=torch.int32)        # page_size 16: 3,
1, 4 pages
    indptr, indices, last = m.block_table_to_csr(bt, ctx, 16)
    assert indptr.dtype == indices.dtype == last.dtype == torch.int32
    assert indptr.tolist() == [0, 3, 4, 8]
    assert indices.tolist() == [7, 2, 5, 3, 1, 4, 6, 8]
    assert last.tolist() == [1, 16, 1]                         # 33 = 2*16 + 1; 16
fills one page


def test_roundtrip():
    m = load("f1_flashinfer_paged")
    g = torch.Generator().manual_seed(0)
    ctx = torch.randint(1, 200, (9,), generator=g).int()
    P = 16
    pages = (ctx + P - 1) // P
    bt = torch.zeros(9, int(pages.max()), dtype=torch.int32)
    perm = torch.randperm(int(pages.sum()), generator=g).int()
    used = 0
    for b, n in enumerate(pages.tolist()):
        bt[b, :n] = perm[used:used + n]
        used += n
    bt2, ctx2 = m.csr_to_block_table(*m.block_table_to_csr(bt, ctx, P), P)
    assert torch.equal(ctx2, ctx)
    for b, n in enumerate(pages.tolist()):
        assert torch.equal(bt2[b, :n], bt[b, :n])
```

```python
@pytest.mark.skipif(not HAS_FLASHINFER, reason="needs a GPU with flashinfer insta
lled")
@pytest.mark.parametrize("Hq,Hkv", [(32, 8), (8, 8)])
def test_flashinfer_decode_matches_reference_and_lse_is_base2(Hq, Hkv):
    m = load("f1_flashinfer_paged")
    D = 128
    ctx = [1, 37, 300, 1000]
    k_cache, v_cache, bt, cl = make_paged(ctx, Hkv, D, 16, torch.float16, DEVICE)
    q = torch.randn(len(ctx), Hq, D, device=DEVICE).half()
    out, lse = m.flashinfer_decode(q, k_cache, v_cache, bt, cl, return_lse=True)
    ref = reference_decode(q, k_cache, v_cache, bt, cl, 1 / math.sqrt(D))
    torch.testing.assert_close(out.float(), ref, rtol=1e-2, atol=1e-2)
    # natural-log LSE for one head, computed directly
    b, h = 2, 0
    blocks = bt[b, : (ctx[b] + 15) // 16].long()
    k = k_cache[blocks].reshape(-1, Hkv, D)[: ctx[b], h // (Hq // Hkv)].float()
    lse_ln = torch.logsumexp(k @ q[b, h].float() / math.sqrt(D), 0)
    assert abs(lse[b, h].item() - lse_ln.item() / math.log(2)) < 1e-2, "FlashInfe
r LSE is log2-based"
```

提示

L1 先算 num_pages 方向（轻度剧透） 收起

`num_pages = ceil(ctx / page_size)`，`indptr` 是它的前缀和（前面补一个 0），`last_page_len = ctx - (num_pages - 1)·page_size`。indices 就是 block table 中每行前 `num_pages[b]` 个元素拼起来。

L2 FlashInfer 调用顺序 拆解与常见坑（中度剧透） 收起

```python
ws = torch.empty(128 << 20, dtype=torch.uint8, device=q.device)       # workspa
w = flashinfer.BatchDecodeWithPagedKVCacheWrapper(ws, kv_layout="NHD")
w.plan(indptr, indices, last, num_qo_heads, num_kv_heads, head_dim, page_size,
       q_data_type=q.dtype, kv_data_type=k_cache.dtype)
out = w.run(q, (k_cache, v_cache))
```

workspace 用来存放 split-KV 的中间结果。第一次调用会 JIT 编译对应 dtype / head_dim 的 kernel，需要几十秒。

L3 **两个转换函数** 几乎是答案（重度剧透） 收起 ▾

```
keep = torch.arange(max_blocks)[None, :] < num_pages[:, None]
indices = block_tables[keep].int()              # 行主序 = 每个序列依次排列
# 逆变换
block_tables = torch.zeros(B, max_blocks, dtype=torch.int32); block_tables[keep
```

参考解

解 **solutions/f1_flashinfer_paged.py** 先自己写完、跑过测试再看 收起

solutions/f1_flashinfer_paged.py 60 行

```
"""F1 · FlashInfer's paged-KV format (CSR) + plan/run decode — reference solu
tion.

vLLM-style block table (what L1–L3 used):
    block_tables [B, max_blocks] int32 (padded), context_lens [B] int32
FlashInfer's CSR format (no padding; each sequence's pages are contiguous in
`indices`):
    indptr        [B + 1] int32   pages of sequence b are indices[indptr[b] :
indptr[b+1]]
    indices       [nnz]   int32   physical page ids, concatenated
    last_page_len [B]     int32   tokens used in each sequence's last page, i
n [1, page_size]
so  context_len[b] = (indptr[b+1] - indptr[b] - 1) * page_size + last_page_le
n[b].
"""
import torch


def block_table_to_csr(block_tables: torch.Tensor, context_lens: torch.Tenso
r, page_size: int):
    """Vectorised (no Python loop over sequences). Requires context_lens >=
1."""
    B, max_blocks = block_tables.shape
    ctx = context_lens.long()
    num_pages = (ctx + page_size - 1) // page_size                          #
[B]
    indptr = torch.zeros(B + 1, dtype=torch.int32, device=block_tables.devic
e)
    indptr[1:] = torch.cumsum(num_pages, 0).to(torch.int32)
    # Keep the first num_pages[b] entries of row b. Boolean indexing walks th
e tensor in
    # row-major order, so each sequence's pages stay together and in order.
    keep = torch.arange(max_blocks, device=block_tables.device)[None, :] < nu
m_pages[:, None]
    indices = block_tables[keep].to(torch.int32)
    last_page_len = (ctx - (num_pages - 1) * page_size).to(torch.int32)   # =
= (ctx - 1) % P + 1
    return indptr, indices, last_page_len


def csr_to_block_table(indptr: torch.Tensor, indices: torch.Tensor, last_page
_len: torch.Tensor,
```

```
                      page_size: int):
    """Inverse of block_table_to_csr -> (block_tables padded with 0, context_
lens)."""
    num_pages = (indptr[1:] - indptr[:-1]).long()                          #
[B]
    B = num_pages.numel()
    max_blocks = int(num_pages.max().item()) if B else 0
    col = torch.arange(max_blocks, device=indices.device)[None, :]
    keep = col < num_pages[:, None]
    block_tables = torch.zeros(B, max_blocks, dtype=torch.int32, device=indic
es.device)
    block_tables[keep] = indices                                           # s
ame row-major order
    context_lens = ((num_pages - 1) * page_size + last_page_len.long()).to(to
rch.int32)
    return block_tables, context_lens


def flashinfer_decode(q, k_cache, v_cache, block_tables, context_lens, worksp
ace=None, return_lse=False):
    """Run FlashInfer's batch decode on the L2-style inputs.

    q [B, Hq, D]; k_cache/v_cache [num_pages, page_size, Hkv, D] ("NHD" layou
t, passed as a tuple).
    plan() does the CPU-side scheduling once per batch shape; run() launches
the kernel and can
    be called again (e.g. once per layer) with the same plan.
    """
    import flashinfer

    B, Hq, D = q.shape
    _, page_size, Hkv, _ = k_cache.shape
    indptr, indices, last = block_table_to_csr(block_tables, context_lens, pa
ge_size)
    if workspace is None:
        workspace = torch.empty(128 << 20, dtype=torch.uint8, device=q.devic
e)   # scratch for split-KV partials
    wrapper = flashinfer.BatchDecodeWithPagedKVCacheWrapper(workspace, kv_lay
out="NHD")
    wrapper.plan(indptr, indices, last, Hq, Hkv, D, page_size,
                 q_data_type=q.dtype, kv_data_type=k_cache.dtype)
    return wrapper.run(q, (k_cache, v_cache), return_lse=return_lse)
```

测完之后

RTX 5090 实测（Hq = 32、Hkv = 8、D = 128、page 16，fp16）：

B × ctx	你的 L2（不分片）	你的 L3（最佳分片）	FlashInfer `run()`
1 × 32768	1844 us	96 us（64 片）	**98 us**
8 × 4096	260 us	95 us（16 片）	**98 us**
64 × 1024	**174 us**	176 us（不分）	180 us

- FlashInfer 在每种形状下都**自动**选到了接近最优的分片方式：这正是 `plan()` 的工作。你的 L3 需要手动为每种形状挑选 `num_splits`。
- 把 `plan()` 也计入时间试试：它在 CPU 上运行并要同步，所以真实系统里每个 batch 只调用一次，再被所有层复用。
- 读源码：`flashinfer/decode.py` 里的 `BatchDecodeWithPagedKVCacheWrapper.plan` → `csrc/batch_decode.cu` → `include/flashinfer/attention/scheduler.cuh`（决定分片的启发式）。把它的分片规则和你在 L3 里观察到的最优分片对照一下。
- `use_tensor_cores=True`：GQA 的 group 较大时，用 prefill kernel 走 Tensor Core 来做 decode。试试对比 group = 8 时的性能。

▼ 面试追问：为什么 FlashInfer 要把 plan 和 run 分开?

调度决策（分几片、每个 CTA 处理哪些 page、中间结果放在 workspace 的哪里）只取决于这一批序列的长度，对所有层都一样。放在 `plan()` 里在 CPU 上算一次，`run()` 就只剩启动 kernel，形状固定、没有 CPU 同步，可以被 CUDA graph 捕获，每层只付一次 launch 的代价。如果每次调用都现场决定，要么每层都在 CPU 上重复计算，要么只能在 GPU 上用一个保守的固定策略。

核心 · Track F · FlashInfer · 第 14 题

F2 merge_state / merge_states：以 2 为底的 LSE 合并

FlashInfer 的 cascade / split-KV / 前缀共享都建立在同一个原语上：两段 attention 状态 (v, lse) 的精确合并。用 Triton 实现它，并和 flashinfer.merge_state 逐数值对拍。

40 min | 难度 中 | 你写 `mine/f2_merge_state.py` | 验收 `./check.sh f2` | 参考解 `solutions/f2_merge_state.py`

1. 背景
2. 源码与资料
3. 要用到的知识：以 2 为底的 LSE
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

L3 里你已经写过一次合并：多个 split 的部分结果用 LSE 加权求和。FlashInfer 把它做成了独立的公开 API，因为它在很多地方都会用到：

- **split-KV decode**：各分片的结果合并（L3）。
- **cascade / 共享前缀 attention**：一批请求共用一个很长的系统 prompt 时，对共享前缀只做一次 attention（batch 内的所有 query 一起读一份 KV），对各自的后缀单独做，最后把两部分合并（`MultiLevelCascadeAttentionWrapper`）。
- **chunked prefill、ring / context parallel**：分块或分卡算出的部分 attention 结果，都要这样合并。

源码与资料

cascade API 文档 · `flashinfer › merge_state` · `flashinfer › merge_states` · `flashinfer › MultiLevelCascadeAttentionWrapper` · `flashinfer › cascade.cuh` · `flashinfer › state.cuh（m / d / o 状态）` · `flashinfer › cascade.cu` · `flashinfer › FlashInfer 自带的 Triton merge` · `flashinfer › test_shared_prefix_kernels.py` · Cascade Inference 博客

要用到的知识：以 2 为底的 LSE

FlashInfer 的 kernel 内部用 `exp2` 和 `log2`（硬件上有对应的快速指令），所以对外暴露的 LSE 也是以 2 为底：

```
s = log₂ Σ_j exp(score_j)            (自然对数版 = s · ln 2)
合并 (v_a, s_a) 和 (v_b, s_b):
  m = max(s_a, s_b),  w_a = 2^(s_a − m),  w_b = 2^(s_b − m)
  v = (w_a v_a + w_b v_b) / (w_a + w_b),  s = m + log₂(w_a + w_b)
```

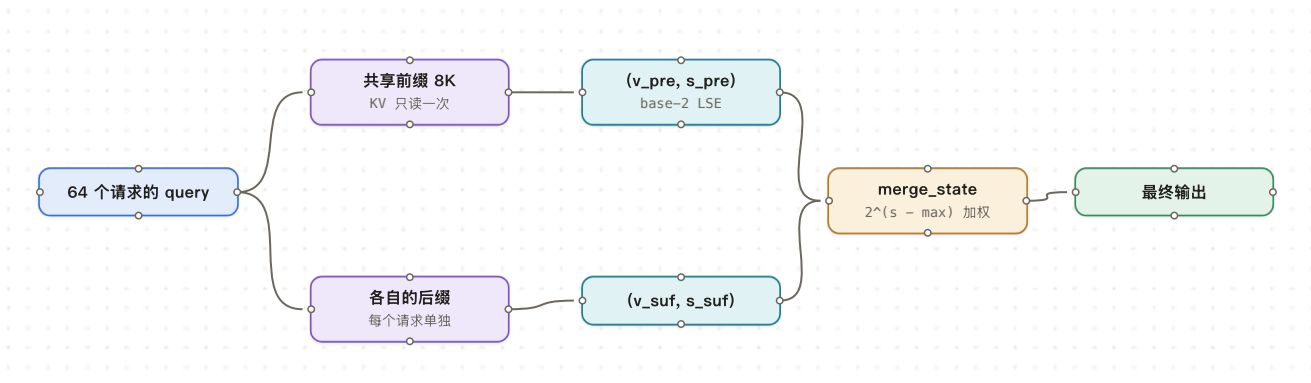
`2^(s_a − m) = exp((s_a − m)·ln 2)`，就是这一段在总分母中的相对权重；底数只要前后一致就行。**混用底数是常见 bug**：L3 的 LSE 是自然对数，直接拿去和 FlashInfer 的 LSE 合并会得到错误的权重。

形状与 FlashInfer 一致：

函数	输入	输出
`merge_state`	`v_a, v_b [S, H, D]`，`s_a, s_b [S, H]` fp32	`v [S, H, D]`，`s [S, H]`
`merge_states`	`v [S, N, H, D]`，`s [S, N, H]`	`v [S, H, D]`，`s [S, H]`

图解与类比

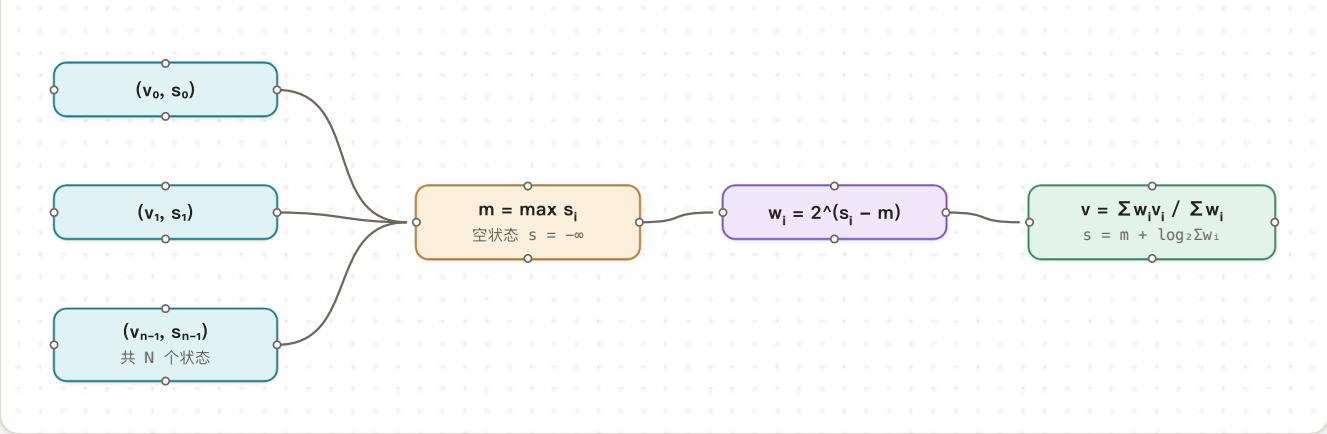
cascade：共享前缀只读一次



类比 · 两个班合并算平均分

两个班各自算出平均分（v），还要知道各班的“总权重”（lse 编码的就是 softmax 的分母），按权重合并才是全年级的平均分。 底数就像单位：FlashInfer 用“以 2 为底”，你用“以 e 为底”，就像一边写斤一边写公斤，直接相加就错了。

merge_states：N 个状态一次合并



类比 · 合并是可以交换、结合的

两两合并的顺序无所谓：先合并 A 和 B 再合并 C，与先合并 B 和 C 再合并 A 结果相同（除了浮点舍入）。所以 split-KV、cascade、ring attention 都可以随意切分，最后再按任意顺序合并；merge_states 只是把 N 个状态在一个 kernel 里一次合并完。

题目要求

wrapper 已写好。在 `mine/f2_merge_state.py` 中实现两个 Triton kernel：每个 program 负责一个 (位置, head)，加载 D 维向量和标量 LSE，按上面的公式合并。

- 空状态（`s = -inf`）的权重是 0，合并结果应等于另一个状态。
- 输出 v 与输入 dtype 相同，s 为 fp32。
- 测试：把一次完整的 attention 拆成两半（或 N 段）分别计算，合并后必须等于整体计算的结果；GPU 上再和 `flashinfer.merge_state / merge_states` 对拍。

题干

mine/f2_merge_state.py 46 行

```
"""F2 · merge_state / merge_states (base-2 LSE) — YOUR IMPLEMENTATION.

See the drill page for the spec; the reference lives in solutions/."""
import torch
import triton
import triton.language as tl


@triton.jit
```

```python
def merge_state_kernel(va_ptr, sa_ptr, vb_ptr, sb_ptr, vo_ptr, so_ptr, H, D,
                       BLOCK_D: tl.constexpr):
    # TODO: pid = (position, head); load s_a, s_b and the two D-vectors
    # TODO: m = max; w = exp2(s - m); v = (wa va + wb vb) / (wa + wb); s = m + lo
g2(wa + wb)
    pass


@triton.jit
def merge_states_kernel(v_ptr, s_ptr, vo_ptr, so_ptr, N, H, D,
                        BLOCK_N: tl.constexpr, BLOCK_D: tl.constexpr):
    # TODO: load the N lse values of (pos, h) (pad with -inf) and the [N, D] outp
uts
    # TODO: w = exp2(s - max); v = sum(w v) / sum(w); s = max + log2(sum(w))
    pass


def merge_state(v_a, s_a, v_b, s_b):
    S, H, D = v_a.shape
    v_a, v_b = v_a.contiguous(), v_b.contiguous()
    s_a, s_b = s_a.float().contiguous(), s_b.float().contiguous()
    v_out = torch.empty_like(v_a)
    s_out = torch.empty((S, H), dtype=torch.float32, device=v_a.device)
    if S * H:
        merge_state_kernel[(S * H,)](v_a, s_a, v_b, s_b, v_out, s_out, H, D,
                                     BLOCK_D=triton.next_power_of_2(D))
    return v_out, s_out


def merge_states(v, s):
    S, N, H, D = v.shape
    v, s = v.contiguous(), s.float().contiguous()
    v_out = torch.empty((S, H, D), dtype=v.dtype, device=v.device)
    s_out = torch.empty((S, H), dtype=torch.float32, device=v.device)
    if S * H:
        merge_states_kernel[(S * H,)](v, s, v_out, s_out, N, H, D,
                                      BLOCK_N=max(2, triton.next_power_of_2(N)),
                                      BLOCK_D=triton.next_power_of_2(D))
    return v_out, s_out
```

验收

```
./check.sh f2
./check.sh f2 --bench      # vs flashinfer.merge_state
```

tests/test_f2_merge_state.py 70 行

```
import math

import pytest
import torch

from common.env import DEVICE, HAS_FLASHINFER
from drill import load

LN2 = math.log(2)


def attn_state(q, k, v):
    """Ground truth: (softmax(q k^T) v, log2-sum-exp) for q [S, H, D], k/v [T, H, D]."""
    s = torch.einsum("shd,thd->sht", q.float(), k.float()) / math.sqrt(q.shape[-1])
    return torch.einsum("sht,thd->shd", s.softmax(-1), v.float()), torch.logsumexp(s, -1) / LN2


def test_merge_two_halves_equals_full_attention():
    m = load("f2_merge_state")
    S, T, H, D = 5, 90, 4, 64
    q, k, v = torch.randn(S, H, D, device=DEVICE), torch.randn(T, H, D, device=DEVICE), torch.randn(T, H, D, device=DEVICE)
    va, sa = attn_state(q, k[:40], v[:40])
    vb, sb = attn_state(q, k[40:], v[40:])
    vf, sf = attn_state(q, k, v)
    vm, sm = m.merge_state(va.half(), sa, vb.half(), sb)
    assert vm.dtype == torch.float16 and sm.dtype == torch.float32
    torch.testing.assert_close(vm.float(), vf, rtol=2e-3, atol=2e-3)
    torch.testing.assert_close(sm, sf, rtol=1e-4, atol=1e-4)


def test_merge_with_empty_state():
    m = load("f2_merge_state")
    va = torch.randn(3, 2, 32, device=DEVICE)
    sa = torch.randn(3, 2, device=DEVICE)
    vb = torch.zeros_like(va)
    sb = torch.full_like(sa, float("-inf"))           # an empty key set has weight 2^-inf = 0
    vm, sm = m.merge_state(va, sa, vb, sb)
    torch.testing.assert_close(vm, va)
    torch.testing.assert_close(sm, sa)


@pytest.mark.parametrize("N", [1, 3, 8])
def test_merge_states(N):
    m = load("f2_merge_state")
    S, T, H, D = 4, 16 * N, 3, 64
```

```
    q, k, v = torch.randn(S, H, D, device=DEVICE), torch.randn(T, H, D, device=DE
VICE), torch.randn(T, H, D, device=DEVICE)
    parts = [attn_state(q, k[i * 16:(i + 1) * 16], v[i * 16:(i + 1) * 16]) for i
in range(N)]
    V = torch.stack([p[0] for p in parts], 1).half()          # [S, N, H, D]
    Sx = torch.stack([p[1] for p in parts], 1)                # [S, N, H]
    vm, sm = m.merge_states(V, Sx)
    vf, sf = attn_state(q, k, v)
    torch.testing.assert_close(vm.float(), vf, rtol=2e-3, atol=2e-3)
    torch.testing.assert_close(sm, sf, rtol=1e-4, atol=1e-4)


@pytest.mark.skipif(not HAS_FLASHINFER, reason="needs a GPU with flashinfer insta
lled")
def test_matches_flashinfer():
    import flashinfer
    m = load("f2_merge_state")
    va, vb = (torch.randn(64, 32, 128, device="cuda").half() for _ in range(2))
    sa, sb = (torch.randn(64, 32, device="cuda") * 4 for _ in range(2))
    v1, s1 = m.merge_state(va, sa, vb, sb)
    v2, s2 = flashinfer.merge_state(va, sa, vb, sb)
    torch.testing.assert_close(v1, v2, rtol=2e-3, atol=2e-3)
    torch.testing.assert_close(s1, s2, rtol=1e-5, atol=1e-5)
    V, Sx = torch.randn(64, 5, 32, 128, device="cuda").half(), torch.randn(64, 5,
32, device="cuda")
    v1, s1 = m.merge_states(V, Sx)
    v2, s2 = flashinfer.merge_states(V, Sx)
    torch.testing.assert_close(v1, v2, rtol=2e-3, atol=2e-3)
    torch.testing.assert_close(s1, s2, rtol=1e-5, atol=1e-5)
```

提示

L1 地址 方向（轻度剧透） 收起

`v_a` 连续存放、形状 `[S, H, D]`，所以 (位置, head) 的线性编号 `pid = pos·H + h` 对应的向量从 `pid·D` 开始，标量 LSE 在 `pid`。merge_states 中 `s[pos, n, h]` 的位置是 `(pos·N + n)·H + h`。

L2 一定用 exp2 / log2 拆解与常见坑（中度剧透） 收起

`tl.exp2(s - m)` 和 `tl.log2(w_a + w_b)`。用 `tl.exp` 不会报错，但数值会和 FlashInfer 对不上（相当于把以 2 为底的 LSE 当成了自然对数）。

L3 **merge_states 的核心** 几乎是答案（重度剧透） 收起 ▾

```
s = tl.load(s_ptr + (pos * N + offs_n) * H + h, mask=n_mask, other=float("-inf"
m = tl.max(s, axis=0)
w = tl.exp2(s - m)                                          # 补齐的状态权重为 0
v = tl.load(v_ptr + ((pos * N + offs_n[:, None]) * H + h) * D + offs_d[None, :]
out = tl.sum(w[:, None] * v, axis=0) / tl.sum(w, axis=0)
s_out = m + tl.log2(tl.sum(w, axis=0))
```

参考解

解 **solutions/f2_merge_state.py** 先自己写完、跑过测试再看 收起

solutions/f2_merge_state.py 79 行

```
"""F2 · merge_state / merge_states with FlashInfer's base-2 LSE — reference s
olution.

An attention "state" over a subset of keys is (v, s): v is the softmax-normal
ised output over
that subset, s the log-sum-exp of its scores. FlashInfer stores s in BASE 2:
    s = log2( sum_j exp(score_j) )          (natural-log LSE = s * ln 2)
Merging two states (same queries, disjoint key sets):
    m   = max(s_a, s_b)
    w_a = 2^(s_a - m),  w_b = 2^(s_b - m)
    v   = (w_a v_a + w_b v_b) / (w_a + w_b),   s = m + log2(w_a + w_b)
Shapes follow flashinfer.merge_state / merge_states:
    merge_state : v_a, v_b [S, H, D]; s_a, s_b [S, H] fp32  -> (v [S, H, D],
s [S, H])
    merge_states: v [S, N, H, D];     s [S, N, H]   fp32    -> (v [S, H, D],
s [S, H])
"""
import torch
import triton
import triton.language as tl


@triton.jit
def merge_state_kernel(va_ptr, sa_ptr, vb_ptr, sb_ptr, vo_ptr, so_ptr, H, D,
                       BLOCK_D: tl.constexpr):
    pid = tl.program_id(0).to(tl.int64)       # one program per (position, he
ad) = pid
    offs = tl.arange(0, BLOCK_D)
    mask = offs < D
    sa = tl.load(sa_ptr + pid)
    sb = tl.load(sb_ptr + pid)
    m = tl.maximum(sa, sb)
    wa = tl.exp2(sa - m)                      # base 2 throughout: exp2 / log
2
    wb = tl.exp2(sb - m)
    va = tl.load(va_ptr + pid * D + offs, mask=mask, other=0.0).to(tl.float3
2)
    vb = tl.load(vb_ptr + pid * D + offs, mask=mask, other=0.0).to(tl.float3
2)
    v = (wa * va + wb * vb) / (wa + wb)
    tl.store(vo_ptr + pid * D + offs, v.to(vo_ptr.dtype.element_ty), mask=mas
k)
```

```
    tl.store(so_ptr + pid, m + tl.log2(wa + wb))


@triton.jit
def merge_states_kernel(v_ptr, s_ptr, vo_ptr, so_ptr, N, H, D,
                        BLOCK_N: tl.constexpr, BLOCK_D: tl.constexpr):
    pid = tl.program_id(0).to(tl.int64)       # (position, head)
    pos = pid // H
    h = pid % H
    offs_n = tl.arange(0, BLOCK_N)
    offs_d = tl.arange(0, BLOCK_D)
    n_mask = offs_n < N
    # s[pos, n, h] lives at (pos * N + n) * H + h
    s = tl.load(s_ptr + (pos * N + offs_n) * H + h, mask=n_mask, other=float
("-inf"))
    m = tl.max(s, axis=0)
    w = tl.exp2(s - m)                        # padded states -> 2^-inf = 0
    v = tl.load(v_ptr + ((pos * N + offs_n[:, None]) * H + h) * D + offs_d[No
ne, :],
                mask=n_mask[:, None] & (offs_d[None, :] < D), other=0.0).to(t
l.float32)
    w_sum = tl.sum(w, axis=0)
    out = tl.sum(w[:, None] * v, axis=0) / w_sum
    tl.store(vo_ptr + pid * D + offs_d, out.to(vo_ptr.dtype.element_ty), mask
=offs_d < D)
    tl.store(so_ptr + pid, m + tl.log2(w_sum))


def merge_state(v_a, s_a, v_b, s_b):
    S, H, D = v_a.shape
    v_a, v_b = v_a.contiguous(), v_b.contiguous()
    s_a, s_b = s_a.float().contiguous(), s_b.float().contiguous()
    v_out = torch.empty_like(v_a)
    s_out = torch.empty((S, H), dtype=torch.float32, device=v_a.device)
    if S * H:
        merge_state_kernel[(S * H,)](v_a, s_a, v_b, s_b, v_out, s_out, H, D,
                                     BLOCK_D=triton.next_power_of_2(D))
    return v_out, s_out


def merge_states(v, s):
    S, N, H, D = v.shape
    v, s = v.contiguous(), s.float().contiguous()
    v_out = torch.empty((S, H, D), dtype=v.dtype, device=v.device)
    s_out = torch.empty((S, H), dtype=torch.float32, device=v.device)
    if S * H:
        merge_states_kernel[(S * H,)](v, s, v_out, s_out, N, H, D,
                                      BLOCK_N=max(2, triton.next_power_of_2
(N)),
```

```
                                        BLOCK_D=triton.next_power_of_2(D))
    return v_out, s_out
```

测完之后

- RTX 5090 实测（H = 32、D = 128，fp16）：S = 128 时你的实现和 FlashInfer 都是 6.2 us（纯 launch 开销）；S = 8192 时你的 156 us · 1293 GB/s，FlashInfer 135 us · 1490 GB/s。差距来自向量化宽度和每个 program 的工作量：试试每个 program 处理多个 head。
- 读源码：`flashinfer/cascade.py` 的 `merge_state` → `include/flashinfer/attention/cascade.cuh`。注意它用 `vec_t` 一次读 16 字节，每个线程处理多个元素。
- 用 `MultiLevelCascadeAttentionWrapper` 做一个实验：64 个请求共享一个 8K token 的前缀，比较“cascade”与“每个请求各自读一遍前缀”的耗时，体会共享前缀省下了多少 KV 读取。

▼ 面试追问：cascade attention 为什么能加速共享前缀?

不用 cascade 时，batch 里每个请求都要从 HBM 读一遍完整的前缀 KV，读取量是 B × 前缀长度。cascade 先把所有请求的 query 一起对共享前缀做一次 attention：前缀的 KV 只读一次，而且多个 query 共用同一块 KV，可以组成矩阵乘、用上 Tensor Core。各请求的后缀再单独计算，最后用 merge_state 合并。合并本身只读写 O(B·H·D) 的数据，相对前缀的 KV 读取可以忽略。

核心 · Track F · FlashInfer · 第 15 题

F3 fused_add_rmsnorm（原地）+ silu_and_mul

每个 decoder layer 都要做两次“残差加 + RMSNorm”和一次 SwiGLU：按 FlashInfer 的语义（包括原地写回两个张量）用 Triton 实现它们，并和官方 kernel 比带宽。

35 min | 难度 中 | 你写 `mine/f3_fused_norm_act.py` | 验收 `./check.sh f3` | 参考解 `solutions/f3_fused_norm_act.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

pre-norm Transformer 的每一层：

```
h = x + attn(rmsnorm(x));   x = h + mlp(rmsnorm(h))
```

serving 框架会把“残差加”和“下一个 RMSNorm”融合成一个 kernel：`fused_add_rmsnorm(x, residual, w)`。它读 x 和 residual 各一次，然后**原地**写回两个张量：residual 更新为 `x + residual`，作为残差流继续往下传；x 更新为归一化后的结果，作为下一个子层的输入。MLP 里的 SwiGLU 同样融合成一个 kernel：`silu_and_mul(gate_up) = silu(gate) · up`。

这两个 kernel 都在 FlashInfer 的 `flashinfer/norm` 和 `flashinfer/activation.py` 里。它们是 FlashInfer 中最简单的一类实现，也是 F5 推荐的第一条源码阅读路线。

源码与资料

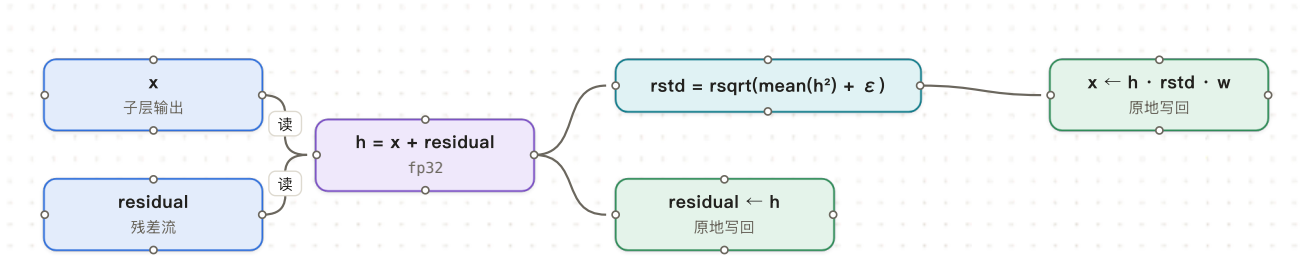
norm API 文档 · activation API 文档 · `flashinfer › fused_add_rmsnorm` · `flashinfer › gen_norm_module` · `flashinfer › norm.cu` · `flashinfer › FusedAddRMSNorm` · `flashinfer › RMSNormKernel` · `flashinfer › silu_and_mul` · `flashinfer › activation.cuh` · `flashinfer › test_norm.py` · `flashinfer › bench_fused_add_rmsnorm.py` · `vllm › vLLM fused_add_rms_norm_kernel`

要用到的知识

- **原地语义**：函数返回 None，测试会检查两个张量的 data_ptr 没有变。wrapper 里要用 `view` 而不是 `reshape`：张量不连续时 `reshape` 会悄悄拷贝一份，写回就丢了。
- **精度约定**（和 FlashInfer 一致）：`h = x + residual` 在 fp32 里计算；存回 residual 时转成 residual 的 dtype；归一化使用 fp32 的 h。
- `silu(g) = g · sigmoid(g)`；输入 `[..., 2d]` 的前一半是 gate，后一半是 up（这个顺序已在 FlashInfer 上验证过）。
- silu_and_mul 用二维 grid：`(行数, cdiv(d, BLOCK))`。d 很大（比如 14336）时，一个 program 只处理一段，program 数也更多。

图解与类比

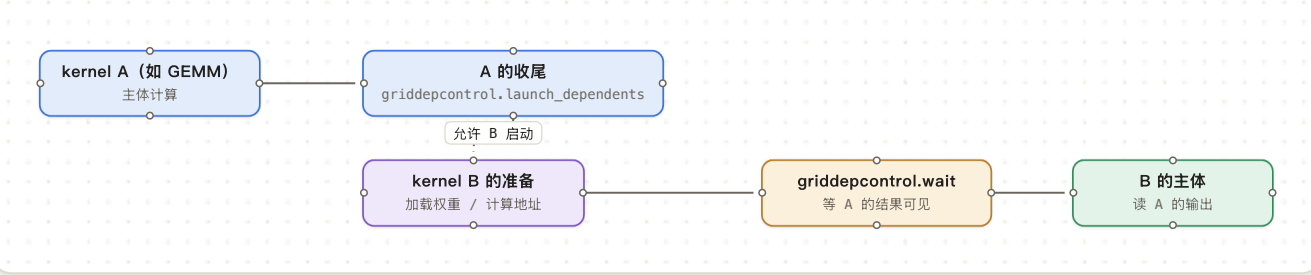
fused_add_rmsnorm：读两个张量，原地写回两个张量



类比 · 记账：一次翻开账本做完两件事

把今天的收入（x）记进账户余额（residual），这是给后面所有人看的总账；再把余额“标准化”一下交给下一个部门（归一化后的 x）。 不融合就是翻开账本记一笔、合上，再翻开做标准化：每翻一次账本都是一趟 HBM 读写。

PDL：下一个 kernel 提前开始准备



类比 · 接力赛提前起跑

> 接力赛里，下一棒不会站着等上一棒跑到面前才起步，而是在交接区提前助跑（PDL：上一个 kernel 快结束时，下一个 kernel 就开始做不依赖上一棒结果的准备），真正交接棒（`griddepcontrol.wait`）时已经在全速奔跑。decode 阶段每个 kernel 只有几微秒，省下的这段启动时间很可观。

题目要求

wrapper 已写好。实现 `fused_add_rmsnorm_kernel` 和 `silu_and_mul_kernel`：

- fp32 / fp16 / bf16；N 任意（⩽ 16384）。
- GPU 上与 `flashinfer.fused_add_rmsnorm`、`flashinfer.silu_and_mul` 对拍（bf16，H = 4096、I = 11008）。

题干

mine/f3_fused_norm_act.py 42 行

```
"""F3 · fused_add_rmsnorm (in place) + silu_and_mul — YOUR IMPLEMENTATION.

See the drill page for the spec; the reference lives in solutions/."""
import torch
import triton
import triton.language as tl


@triton.jit
def fused_add_rmsnorm_kernel(x_ptr, r_ptr, w_ptr, x_stride, r_stride, N, eps, BLO
CK_N: tl.constexpr):
    # TODO: h = x + residual (fp32); store h into residual; x = h * rsqrt(mean(h^
2) + eps) * w
    pass


@triton.jit
def silu_and_mul_kernel(x_ptr, out_ptr, x_stride, out_stride, d, BLOCK: tl.conste
xpr):
    # TODO: gate = x[row, offs], up = x[row, d + offs]; out = gate * sigmoid(gat
e) * up
    pass


def fused_add_rmsnorm(x: torch.Tensor, residual: torch.Tensor, weight: torch.Tens
or, eps: float = 1e-6) -> None:
    assert x.shape == residual.shape and x.shape[-1] == weight.shape[0]
```

```
    assert x.stride(-1) == 1 and residual.stride(-1) == 1
    N = x.shape[-1]
    x2, r2 = x.view(-1, N), residual.view(-1, N)       # view, not reshape: we mu
st write in place
    M = x2.shape[0]
    if M == 0:
        return
    BLOCK_N = triton.next_power_of_2(N)
    fused_add_rmsnorm_kernel[(M,)](x2, r2, weight, x2.stride(0), r2.stride(0), N,
eps,
                                   BLOCK_N=BLOCK_N, num_warps=min(max(BLOCK_N //
256, 1), 8))


def silu_and_mul(x: torch.Tensor) -> torch.Tensor:
    assert x.shape[-1] % 2 == 0 and x.stride(-1) == 1
    d = x.shape[-1] // 2
    x2 = x.reshape(-1, 2 * d)
    out = torch.empty((x2.shape[0], d), dtype=x.dtype, device=x.device)
    if x2.shape[0]:
        BLOCK = min(1024, triton.next_power_of_2(d))
        silu_and_mul_kernel[(x2.shape[0], triton.cdiv(d, BLOCK))](x2, out, x2.str
ide(0), out.stride(0), d, BLOCK=BLOCK)
    return out.reshape(*x.shape[:-1], d)
```

验收

```
./check.sh f3
./check.sh f3 --bench       # vs FlashInfer, GB/s
```

tests/test_f3_fused_norm_act.py 52 行

```
import pytest
import torch
import torch.nn.functional as F

from common.env import DEVICE, HAS_FLASHINFER
from drill import load


def ref_fused_add_rmsnorm(x, r, w, eps):
    h = x.float() + r.float()
    return (h * torch.rsqrt(h.pow(2).mean(-1, keepdim=True) + eps) * w.float()).t
o(x.dtype), h.to(r.dtype)


@pytest.mark.parametrize("shape", [(1, 64), (7, 333), (4, 3, 1024)])
```

```
@pytest.mark.parametrize("dtype", [torch.float32, torch.float16])
def test_fused_add_rmsnorm_in_place(shape, dtype):
    m = load("f3_fused_norm_act")
    x = torch.randn(shape, device=DEVICE).to(dtype)
    r = torch.randn(shape, device=DEVICE).to(dtype)
    w = torch.randn(shape[-1], device=DEVICE).to(dtype)
    ref_x, ref_r = ref_fused_add_rmsnorm(x, r, w, 1e-6)
    x_ptr, r_ptr = x.data_ptr(), r.data_ptr()
    assert m.fused_add_rmsnorm(x, r, w, 1e-6) is None
    assert x.data_ptr() == x_ptr and r.data_ptr() == r_ptr, "must write in place"
    tol = 1e-5 if dtype == torch.float32 else 1e-2
    torch.testing.assert_close(r, ref_r, rtol=tol, atol=tol)
    torch.testing.assert_close(x, ref_x, rtol=tol, atol=tol)


@pytest.mark.parametrize("shape", [(1, 2), (5, 2 * 333), (2, 3, 2 * 4096)])
def test_silu_and_mul(shape):
    m = load("f3_fused_norm_act")
    x = torch.randn(shape, device=DEVICE)
    d = shape[-1] // 2
    out = m.silu_and_mul(x)
    assert out.shape == (*shape[:-1], d)
    torch.testing.assert_close(out, F.silu(x[..., :d]) * x[..., d:], rtol=1e-5, a
tol=1e-5)


@pytest.mark.skipif(not HAS_FLASHINFER, reason="needs a GPU with flashinfer insta
lled")
def test_matches_flashinfer():
    import flashinfer
    m = load("f3_fused_norm_act")
    x, r = (torch.randn(64, 4096, device="cuda", dtype=torch.bfloat16) for _ in r
ange(2))
    w = torch.randn(4096, device="cuda", dtype=torch.bfloat16)
    x1, r1, x2, r2 = x.clone(), r.clone(), x.clone(), r.clone()
    m.fused_add_rmsnorm(x1, r1, w)
    flashinfer.fused_add_rmsnorm(x2, r2, w)
    torch.testing.assert_close(r1, r2)
    torch.testing.assert_close(x1.float(), x2.float(), rtol=2e-2, atol=2e-2)
    y = torch.randn(64, 2 * 11008, device="cuda", dtype=torch.bfloat16)
    torch.testing.assert_close(m.silu_and_mul(y).float(), flashinfer.silu_and_mul
(y).float(), rtol=2e-2, atol=2e-2)
```

提示

L1 fused_add_rmsnorm 的顺序 方向（轻度剧透） 收起

load x、r（fp32）→ `h = x + r` → **先**把 h 存进 residual → 用 h 算 rstd → 存 `h·rstd·w` 进 x。两个 store 写的是不同的张量，先后顺序不影响正确性；但都要发生在 load 之后。

L2 常见坑 拆解与常见坑（中度剧透） 收起

- 用 T3 的 `rms_norm` 再加一次残差加：结果对，但多读写了一遍，就失去了融合的意义。
- wrapper 里写成 `x.reshape(-1, N)`：非连续输入时写回不到原张量。
- silu_and_mul 把前后两半弄反：FlashInfer 是 `silu(前一半) · 后一半`。

L3 kernel 主体 几乎是答案（重度剧透） 收起

```
x = tl.load(x_ptr + row * x_stride + cols, mask=mask, other=0.0).to(tl.float32)
r = tl.load(r_ptr + row * r_stride + cols, mask=mask, other=0.0).to(tl.float32)
h = x + r
tl.store(r_ptr + row * r_stride + cols, h.to(r_ptr.dtype.element_ty), mask=mask
rstd = 1.0 / tl.sqrt(tl.sum(h * h, axis=0) / N + eps)
w = tl.load(w_ptr + cols, mask=mask, other=0.0).to(tl.float32)
tl.store(x_ptr + row * x_stride + cols, (h * rstd * w).to(x_ptr.dtype.element_t
```

参考解

解 **solutions/f3_fused_norm_act.py** 先自己写完、跑过测试再看 收起

solutions/f3_fused_norm_act.py 63 行

```
"""F3 · fused_add_rmsnorm (in place) + silu_and_mul, with FlashInfer's semant
ics — reference solution.

fused_add_rmsnorm(x, residual, w, eps)   -> None, both tensors updated IN PLA
CE:
    h = x + residual          (computed in fp32)
    residual <- h             (stored in residual's dtype: the next layer's r
esidual stream)
    x        <- h * rsqrt(mean(h^2) + eps) * w
This is the pre-norm Transformer step "add the sublayer output to the residua
l, then normalise
for the next sublayer" in one pass: read x, residual once; write both once.

silu_and_mul(x[..., 2d]) -> [..., d]:  silu(x[..., :d]) * x[..., d:]   (the S
wiGLU MLP gate)
"""
import torch
import triton
import triton.language as tl


@triton.jit
def fused_add_rmsnorm_kernel(x_ptr, r_ptr, w_ptr, x_stride, r_stride, N, eps,
BLOCK_N: tl.constexpr):
    row = tl.program_id(0).to(tl.int64)
    cols = tl.arange(0, BLOCK_N)
    mask = cols < N
    x = tl.load(x_ptr + row * x_stride + cols, mask=mask, other=0.0).to(tl.fl
oat32)
    r = tl.load(r_ptr + row * r_stride + cols, mask=mask, other=0.0).to(tl.fl
oat32)
    h = x + r
    tl.store(r_ptr + row * r_stride + cols, h.to(r_ptr.dtype.element_ty), mas
k=mask)   # new residual
    rstd = 1.0 / tl.sqrt(tl.sum(h * h, axis=0) / N + eps)
    w = tl.load(w_ptr + cols, mask=mask, other=0.0).to(tl.float32)
    tl.store(x_ptr + row * x_stride + cols, (h * rstd * w).to(x_ptr.dtype.ele
ment_ty), mask=mask)


@triton.jit
def silu_and_mul_kernel(x_ptr, out_ptr, x_stride, out_stride, d, BLOCK: tl.co
```

```
nstexpr):
    row = tl.program_id(0).to(tl.int64)
    offs = tl.program_id(1) * BLOCK + tl.arange(0, BLOCK)   # 2-D grid: rows
x column blocks
    mask = offs < d
    gate = tl.load(x_ptr + row * x_stride + offs, mask=mask, other=0.0).to(t
l.float32)
    up = tl.load(x_ptr + row * x_stride + d + offs, mask=mask, other=0.0).to
(tl.float32)
    out = gate * tl.sigmoid(gate) * up                          # silu(g) = g *
sigmoid(g)
    tl.store(out_ptr + row * out_stride + offs, out.to(out_ptr.dtype.element_
ty), mask=mask)


def fused_add_rmsnorm(x: torch.Tensor, residual: torch.Tensor, weight: torch.
Tensor, eps: float = 1e-6) -> None:
    assert x.shape == residual.shape and x.shape[-1] == weight.shape[0]
    assert x.stride(-1) == 1 and residual.stride(-1) == 1
    N = x.shape[-1]
    x2, r2 = x.view(-1, N), residual.view(-1, N)       # view, not reshape: w
e must write in place
    M = x2.shape[0]
    if M == 0:
        return
    BLOCK_N = triton.next_power_of_2(N)
    fused_add_rmsnorm_kernel[(M,)](x2, r2, weight, x2.stride(0), r2.stride
(0), N, eps,
                                   BLOCK_N=BLOCK_N, num_warps=min(max(BLOCK_N
// 256, 1), 8))


def silu_and_mul(x: torch.Tensor) -> torch.Tensor:
    assert x.shape[-1] % 2 == 0 and x.stride(-1) == 1
    d = x.shape[-1] // 2
    x2 = x.reshape(-1, 2 * d)
    out = torch.empty((x2.shape[0], d), dtype=x.dtype, device=x.device)
    if x2.shape[0]:
        BLOCK = min(1024, triton.next_power_of_2(d))
        silu_and_mul_kernel[(x2.shape[0], triton.cdiv(d, BLOCK))](x2, out, x
2.stride(0), out.stride(0), d, BLOCK=BLOCK)
    return out.reshape(*x.shape[:-1], d)
```

测完之后

RTX 5090 实测（bf16）：

算子	T = 16（decode）	T = 4096（prefill）
fused_add_rmsnorm，H = 4096	你的 6.2 us / FlashInfer 6.2 us	你的 **88 us · 1522 GB/s** / FlashInfer 90 us
silu_and_mul，I = 14336	你的 **6.1 us** / FlashInfer 8.2 us	你的 244 us / FlashInfer 242 us

- 大 T 时两者都到了带宽上限：这类 memory-bound 的 kernel，Triton 写起来容易，性能也够好。
- 小 T 时时间都在 6 us 左右：这是 launch 和调度的下限。decode 阶段这类 kernel 的优化方向是进一步**融合**（和前后的 GEMM、量化合并）以及 **PDL**（Programmatic Dependent Launch，Hopper 起支持，允许下一个 kernel 提前启动）。FlashInfer 的 API 带 `enable_pdl` 参数，RMSNorm 源码中的 `griddepcontrol` 指令就是它。
- 读源码：`flashinfer/norm/__init__.py` 的 `rmsnorm` → `flashinfer/jit/norm.py`（`gen_norm_module`）→ `csrc/norm.cu` 和 `csrc/flashinfer_norm_binding.cu` → `include/flashinfer/norm.cuh` 里的 `RMSNorm` / `FusedAddRMSNorm`。关注 `vec_size = gcd(16 / sizeof(T), d)` 和 `block_size = min(1024, d / vec_size)` 这两行：它是怎么决定向量化宽度和线程数的?
- 进阶：FlashInfer 有 `fused_add_rmsnorm_quant`、`fused_add_rmsnorm_fp8_block_quant`，把 FP8 量化也并进来。结合 M3，自己实现一个“残差加 + RMSNorm + 1×128 FP8 量化”的融合 kernel。

▼ 面试追问：为什么这些小算子在 decode 时值得花力气优化?

decode 时每层的 GEMM 也很小（矩阵乘向量），一层 attention + MLP 的总耗时可能只有几十微秒，而每个小算子都有几微秒的固定开销。一个 decoder layer 里有 RMSNorm × 2、残差加 × 2、RoPE、SwiGLU、量化等十来个小算子，不融合时它们的开销加起来可以和 GEMM 相当。所以 decode 优化的重点是减少 kernel 数量（融合、CUDA graph、PDL），而不是把单个小 kernel 再提速 10%。

核心 · Track F · FlashInfer · 第 16 题

F4 不排序的 top-k / top-p：对浮点位模式二分

15 万词表上的 top-k / top-p：不排序，只做若干次全词表的归约来逼近阈值。你会写出一个正确、在大 batch 时比排序快 3 倍的版本，再从它和 FlashInfer 之间 3–10 倍的差距里看出好算法长什么样。

60 min | 难度 难 | 你写 `mine/f4_topk_topp_renorm.py` | 验收 `./check.sh f4` | 参考解 `solutions/f4_topk_topp_renorm.py`

1. 背景
2. 源码与资料
3. 要用到的知识：对浮点位模式二分
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

每个 decode step、每个请求都要从 logits 里采样一个 token。采样参数（temperature、top-k、top-p、min-p）各不相同，词表大小 V 约为 3 万到 25 万。教科书做法是先**排序**再求累积和：对 64 个请求、15 万的词表，单是 `torch.sort` 就要约 1 ms，和一整层 decode 的时间相当。

FlashInfer 的采样和 renorm kernel **从不排序**：它反复对“概率大于某个 pivot 的元素”做（质量, 个数）的归约，根据结果移动 pivot，直到找到阈值。每一轮都是一次在片上完成的全词表规约，不需要全局排序，也不需要额外的显存。

源码与资料

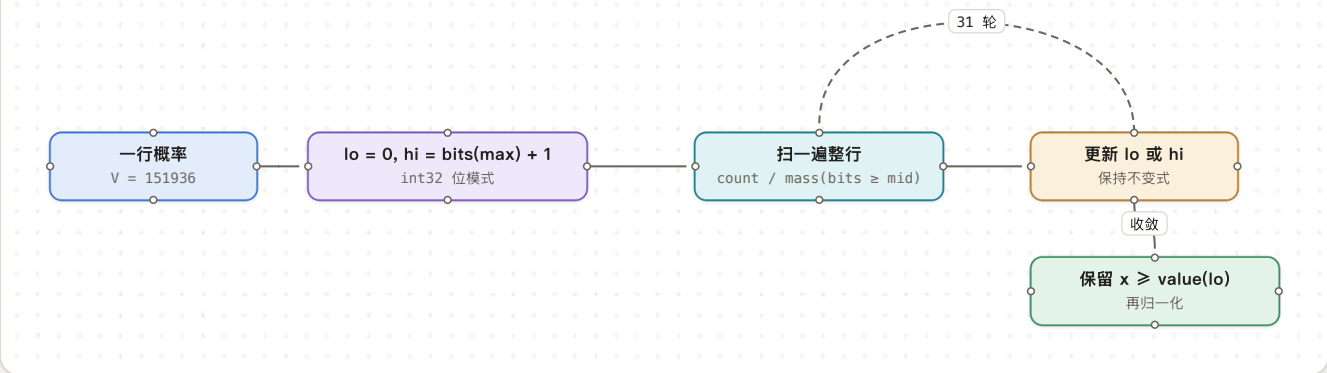
sampling API 文档 · Sorting-Free GPU Kernels for LLM Sampling（FlashInfer 博客） · `flashinfer › top_p_renorm_probs` · `flashinfer › top_k_renorm_probs` · `flashinfer › TopPRenormProb` · `flashinfer › topk.cuh（radix / multi-CTA）` · `flashinfer › renorm.cu` · `flashinfer › test_sampling.py` · `flashinfer › bench_sampling.py` · `vllm › vLLM 的 top-k / top-p 采样` · `sglang › SGLang sampler`

要用到的知识：对浮点位模式二分

- 对非负的 fp32，**把它的位模式当成 int32 解释后，大小顺序与浮点值一致**。所以阈值可以在整数区间 `[0, bits(max) + 1)` 上二分，31 次就能**精确**定位到某一个 fp32 值，不会有浮点二分的精度问题。
- 不变式：`f(lo)` 满足条件，`f(hi)` 不满足：
  - top-k：条件是 `count(x ≥ t) ≥ k`；
  - top-p：条件是 `mass(x ≥ t) ≥ p`。
- 结束时保留 `x ≥ value(lo)` 的元素再归一化。没有并列值时，top-k 正好留下 k 个；top-p 留下的是“按概率从大到小累加，刚好达到 p 的最小集合”（与 FlashInfer 的语义一致，已在 RTX 5090 上验证）。
- Triton：`x.to(tl.int32, bitcast=True)` 按位重新解释；一个 program 负责一行，每一轮都在 `for start in range(0, V, BLOCK)` 循环里扫一遍整行。

图解与类比

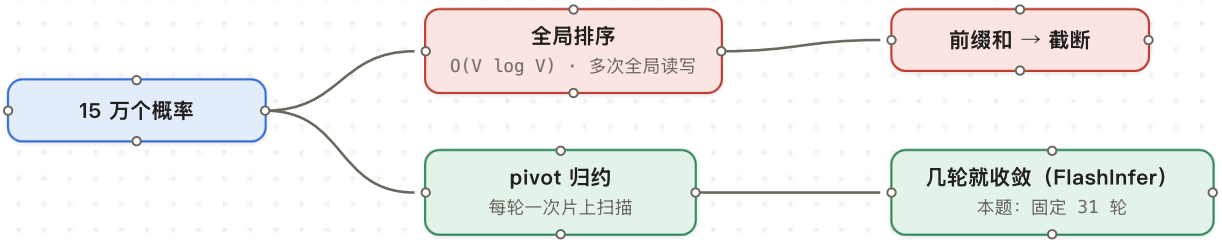
对浮点位模式二分：31 轮找到阈值



类比 · 猜门槛游戏

每轮问一个问题：“概率 ⩾ 这个门槛的有多少个（或者加起来多少）？”太多就抬高门槛，太少就降低。 在浮点值上二分像在一条刻度不均匀的尺子上找点，越找越慢；在位模式上二分像在字典里按页码翻：fp32 只有 2^{31} 个非负的值，31 次必然翻到那一页。

排序 vs 不排序



类比 · 找全班第 20 高的人

想找出全班最高的 20 个人，不需要让全班按身高排好队。拿一根标杆问“比这根高的站出来”：站出来的太多就把标杆调高，太少就调低。几轮之后正好站出来 20 个人，整个过程大家都没有挪动位置（不需要排序和额外的显存）。

题目要求

wrapper 已写好（`k` 和 `p` 可以是标量，也可以是每行一个值的张量）。实现 `_renorm_kernel`：

- `IS_TOP_K` 为 constexpr，同一个 kernel 处理两种情况。
- 输出 fp32 概率，保留的元素和为 1，其余为 0。
- 测试与排序 reference 逐元素对拍；GPU 上和 FlashInfer 对拍（V = 151936，Qwen 的词表大小）。

题干

mine/f4_topk_topp_renorm.py 39 行

```
"""F4 · sorting-free top-k / top-p renorm — YOUR IMPLEMENTATION.

See the drill page for the spec; the reference lives in solutions/."""
import torch
import triton
import triton.language as tl


@triton.jit
def _renorm_kernel(probs_ptr, out_ptr, k_ptr, p_ptr, V, IS_TOP_K: tl.constexpr, B
LOCK: tl.constexpr):
```

```
    # TODO: 1. row max -> hi = bits(max) + 1, lo = 0 (int32 bit patterns of the f
p32 probabilities)
    # TODO: 2. 31 rounds: mid = lo + (hi - lo) // 2; one pass over the row: count
(top-k) or mass (top-p) of x with bits >= mid
    # TODO:    keep the invariant f(lo) satisfies the condition, f(hi) does not
    # TODO: 3. kept = sum of x with bits >= lo; write x / kept for those, 0 elsew
here
    pass


def _per_row(val, B, dtype, device):
    if isinstance(val, torch.Tensor):
        return val.to(device=device, dtype=dtype).expand(B).contiguous()
    return torch.full((B,), val, dtype=dtype, device=device)


def top_k_renorm_probs(probs: torch.Tensor, top_k) -> torch.Tensor:
    probs = probs.float().contiguous()
    B, V = probs.shape
    out = torch.empty_like(probs)
    k = _per_row(top_k, B, torch.float32, probs.device)       # compared with a f
loat count
    _renorm_kernel[(B,)](probs, out, k, k, V, IS_TOP_K=True, BLOCK=4096, num_warp
s=16)
    return out


def top_p_renorm_probs(probs: torch.Tensor, top_p) -> torch.Tensor:
    probs = probs.float().contiguous()
    B, V = probs.shape
    out = torch.empty_like(probs)
    p = _per_row(top_p, B, torch.float32, probs.device)
    _renorm_kernel[(B,)](probs, out, p, p, V, IS_TOP_K=False, BLOCK=4096, num_war
ps=16)
    return out
```

验收

```
./check.sh f4
./check.sh f4 --bench       # vs FlashInfer 和 torch.sort
```

tests/test_f4_topk_topp_renorm.py 70 行

```
import pytest
import torch

from common.env import DEVICE, HAS_CUDA, HAS_FLASHINFER
```

```
from drill import load


def make_probs(B, V, temp=1.0, seed=0):
    g = torch.Generator().manual_seed(seed)
    return torch.softmax(torch.randn(B, V, generator=g) / temp, -1).to(DEVICE)


def ref_top_k(p, k):
    kth = torch.topk(p, k, dim=-1).values[:, -1:]
    out = torch.where(p >= kth, p, 0.0)
    return out / out.sum(-1, keepdim=True)


def ref_top_p(p, top_p):
    sp, idx = torch.sort(p, -1, descending=True)
    keep_sorted = (sp.cumsum(-1) - sp) < top_p            # mass strictly before
this entry < p
    keep = torch.zeros_like(keep_sorted).scatter(-1, idx, keep_sorted)
    out = torch.where(keep, p, 0.0)
    return out / out.sum(-1, keepdim=True)


V_SIZES = [1000, 32000] if HAS_CUDA else [1000, 5000]


@pytest.mark.parametrize("V", V_SIZES)
@pytest.mark.parametrize("k", [1, 10, 50])
def test_top_k(V, k):
    m = load("f4_topk_topp_renorm")
    p = make_probs(4, V)
    out = m.top_k_renorm_probs(p, k)
    assert ((out > 0).sum(-1) == k).all()
    torch.testing.assert_close(out, ref_top_k(p, k), rtol=1e-5, atol=1e-7)


@pytest.mark.parametrize("V", V_SIZES)
@pytest.mark.parametrize("top_p", [0.1, 0.5, 0.9])
def test_top_p(V, top_p):
    m = load("f4_topk_topp_renorm")
    p = make_probs(4, V, temp=0.5)
    out = m.top_p_renorm_probs(p, top_p)
    torch.testing.assert_close(out, ref_top_p(p, top_p), rtol=1e-5, atol=1e-7)


def test_per_row_parameters():
    m = load("f4_topk_topp_renorm")
    p = make_probs(3, 2000)
    ks = torch.tensor([1, 7, 100], device=DEVICE)
    out = m.top_k_renorm_probs(p, ks)
```

```
    assert (out > 0).sum(-1).tolist() == [1, 7, 100]


@pytest.mark.skipif(not HAS_FLASHINFER, reason="needs a GPU with flashinfer instalLed")
def test_matches_flashinfer():
    import flashinfer
    m = load("f4_topk_topp_renorm")
    p = make_probs(32, 151936, temp=0.7)                    # Qwen-sized vocabulary
    torch.testing.assert_close(m.top_k_renorm_probs(p, 20), flashinfer.top_k_renorm_probs(p, 20), rtol=1e-4, atol=1e-7)
    # top-p: where the nucleus boundary lands within ~1e-6 of p, fp32 accumulation order decides
    # whether the boundary token is in. Allow a 1-token difference there; elsewhere be identical.
    a, b = m.top_p_renorm_probs(p, 0.8), flashinfer.top_p_renorm_probs(p, 0.8)
    na, nb = (a > 0).sum(-1), (b > 0).sum(-1)
    assert ((na - nb).abs() <= 1).all()
    same = na == nb
    assert same.float().mean() > 0.5
    torch.testing.assert_close(a[same], b[same], rtol=1e-4, atol=1e-7)
```

提示

L1 三个阶段 方向（轻度剧透） 收起

1. 扫一遍求行最大值 → `hi = bits(max) + 1`，`lo = 0`。
2. 31 轮二分：每轮扫一遍，统计 `bits(x) ≥ mid` 的个数（top-k）或质量（top-p），据此更新 lo 或 hi。
3. 扫一遍求保留部分的总和，再扫一遍写出归一化后的结果。

L2 常见坑 拆解与常见坑（中度剧透） 收起

- 在浮点值上二分（`mid = (lo + hi) / 2`）：几十次之后才能收敛，而且相邻的两个概率值可能非常接近，导致结果差一个元素。在位模式上二分，31 次就是精确的。
- 用 `>` 而不是 `≥`：不变式和最终的保留条件必须用同一个比较。
- `hi` 初值设为 `bits(max)`：此时 `count(x ≥ hi) ≥ 1`，不满足“hi 不满足条件”的不变式（k = 1 时会出错）。

L3 **二分循环** 几乎是答案（重度剧透） 收起 ▾

```
for _ in range(31):
    mid = lo + (hi - lo) // 2
    acc = tl.zeros([BLOCK], dtype=tl.float32)
    for start in range(0, V, BLOCK):
        x = tl.load(base + start + offs, mask=start + offs < V, other=0.0)
        above = x.to(tl.int32, bitcast=True) >= mid
        acc += above.to(tl.float32) if IS_TOP_K else tl.where(above, x, 0.0)
    ok = tl.sum(acc, axis=0) >= (k if IS_TOP_K else p)
    lo = tl.where(ok, mid, lo)
    hi = tl.where(ok, hi, mid)
```

参考解

解 **solutions/f4_topk_topp_renorm.py** 先自己写完、跑过测试再看 收起

solutions/f4_topk_topp_renorm.py 91 行

```
"""F4 · Sorting-free top-k / top-p renormalisation — reference solution.

top_k_renorm_probs(probs [B, V], k)  : keep the k largest probabilities per r
ow, renormalise.
top_p_renorm_probs(probs [B, V], p)  : keep the smallest set of largest proba
bilities whose
                                       mass reaches p (nucleus), renormalise.
Same semantics as flashinfer.top_k_renorm_probs / top_p_renorm_probs (for row
s without ties).

Sorting a 150k-entry row per request per step is the slow part of naive sampl
ers. FlashInfer
never sorts: it repeatedly reduces (mass, count) of the entries above a pivot
and moves the
pivot. Here the pivot search is an exact bisection over the float's bit patte
rn: for x >= 0,
the int32 view of an fp32 value increases with the value, so 31 halvings of
[0, bits(max)+1)
pin down the exact threshold. Each round is one pass over the row.
"""
import torch
import triton
import triton.language as tl


@triton.jit
def _renorm_kernel(probs_ptr, out_ptr, k_ptr, p_ptr, V, IS_TOP_K: tl.constexp
r, BLOCK: tl.constexpr):
    row = tl.program_id(0).to(tl.int64)
    base = probs_ptr + row * V
    offs = tl.arange(0, BLOCK)

    # hi = bits(max) + 1 so that "count(x >= hi) == 0"; lo = 0 so that "count
(x >= lo) == V".
    mx = tl.zeros([BLOCK], dtype=tl.float32)
    for start in range(0, V, BLOCK):
        x = tl.load(base + start + offs, mask=start + offs < V, other=0.0)
        mx = tl.maximum(mx, x)
    lo = tl.zeros([], dtype=tl.int32)
    hi = tl.max(mx, axis=0).to(tl.int32, bitcast=True) + 1

    if IS_TOP_K:
```

```
        k = tl.load(k_ptr + row)
    else:
        p = tl.load(p_ptr + row)

    # Invariant: f(lo) satisfies the condition, f(hi) does not. 31 rounds shr
ink hi - lo to 1.
    for _ in range(31):
        mid = lo + (hi - lo) // 2
        acc = tl.zeros([BLOCK], dtype=tl.float32)
        for start in range(0, V, BLOCK):
            x = tl.load(base + start + offs, mask=start + offs < V, other=0.
0)
            above = x.to(tl.int32, bitcast=True) >= mid      # x >= value(mi
d), compared as bits
            if IS_TOP_K:
                acc += above.to(tl.float32)                  # count above th
e pivot
            else:
                acc += tl.where(above, x, 0.0)               # mass above the
pivot
        total = tl.sum(acc, axis=0)
        if IS_TOP_K:
            ok = total >= k
        else:
            ok = total >= p
        lo = tl.where(ok, mid, lo)
        hi = tl.where(ok, hi, mid)

    # Keep x >= value(lo): exactly k entries (top-k) / the nucleus (top-p). R
enormalise.
    kept = tl.zeros([BLOCK], dtype=tl.float32)
    for start in range(0, V, BLOCK):
        x = tl.load(base + start + offs, mask=start + offs < V, other=0.0)
        kept += tl.where(x.to(tl.int32, bitcast=True) >= lo, x, 0.0)
    denom = tl.sum(kept, axis=0)
    for start in range(0, V, BLOCK):
        m = start + offs < V
        x = tl.load(base + start + offs, mask=m, other=0.0)
        y = tl.where(x.to(tl.int32, bitcast=True) >= lo, x / denom, 0.0)
        tl.store(out_ptr + row * V + start + offs, y, mask=m)


def _per_row(val, B, dtype, device):
    if isinstance(val, torch.Tensor):
        return val.to(device=device, dtype=dtype).expand(B).contiguous()
    return torch.full((B,), val, dtype=dtype, device=device)


def top_k_renorm_probs(probs: torch.Tensor, top_k) -> torch.Tensor:
    probs = probs.float().contiguous()
```

```
    B, V = probs.shape
    out = torch.empty_like(probs)
    k = _per_row(top_k, B, torch.float32, probs.device)       # compared with
a float count
    _renorm_kernel[(B,)](probs, out, k, k, V, IS_TOP_K=True, BLOCK=4096, num_
warps=16)
    return out


def top_p_renorm_probs(probs: torch.Tensor, top_p) -> torch.Tensor:
    probs = probs.float().contiguous()
    B, V = probs.shape
    out = torch.empty_like(probs)
    p = _per_row(top_p, B, torch.float32, probs.device)
    _renorm_kernel[(B,)](probs, out, p, p, V, IS_TOP_K=False, BLOCK=4096, num
_warps=16)
    return out
```

测完之后

RTX 5090 实测（V = 151936）：

	B = 1	B = 64
你的 top-k = 20	268 us	312 us
FlashInfer top-k = 20	**25 us**	**88 us**
你的 top-p = 0.9	268 us	311 us
FlashInfer top-p = 0.9	**76 us**	**164 us**
`torch.sort`（只是排序）	59 us	1020 us

（`BLOCK = 4096, num_warps = 16`；最初用 `BLOCK = 1024, num_warps = 4` 时是 887 us，只调这两个参数就快了 3.3 倍。）

为什么还差 3–10 倍（B = 1 的 top-k 差距最大）：你的版本固定扫 33 遍整行，而且每行只有一个 program，B 很小时只用到一个 SM；B = 1 时甚至比 `torch.sort` 还慢，B = 64 时才快 3 倍。FlashInfer 的做法：

- **用数据决定 pivot**：从一个采样得到的值或当前的上下界出发，每轮同时检查两个 pivot（源码里的 `pivot_0`、`pivot_1`），通常几轮就能收敛，而不是固定 31 轮；
- 每轮的 `(sum, count)` 用 CUB 的 BlockReduce 完成，每个线程一次读 16 字节；
- top-k 还有专门的基数选择（radix select）实现，并且支持**多个 CTA 协作处理同一行**（`include/flashinfer/topk.cuh` 中的 multi-CTA radix top-k），正好解决 B 很小时

只能用一个 SM 的问题。

对拍时的一个发现：top-p 在 32 行中有 4 行和 FlashInfer 差一个 token，而这 4 行在边界处累积质量与 p 的差距都在 1e-6 以内。用 float64 排序的 reference 检查，你的结果是对的：FlashInfer 的 fp32 归约顺序在这种边界情况下会多留或少留一个 token。测试因此允许边界处有 ±1 个 token 的差异。真实采样中这无关紧要，但这是 **kernel 对拍时“容差该怎么定”**的一个好例子。

进阶：

- 让一行由多个 program 协作完成（每个 program 负责一段，每轮做一次跨 program 的归约），解决 B = 1 时只用一个 SM 的问题。
- 读 `include/flashinfer/sampling.cuh` 中的 `TopPRenormProb`（launcher）及其调用的 kernel，以及 `flashinfer/sampling.py` 中 `top_k_top_p_sampling_from_probs` 的 `filter_apply_order` 参数：先 top-k 还是先 top-p，结果为什么会不一样？

▼ 面试追问：为什么采样不用排序？用了会怎样？

排序是 O(V log V) 的全局操作，需要额外的显存和多次全局读写；而 top-k / top-p 只需要知道一个阈值。不排序的做法是让每一轮都在片上完成一次全词表归约，代价是 O(V × 轮数)，轮数很小，数据还在 L2 里。batch 增大后排序的代价迅速上升（本题 B = 64 时 `torch.sort` 要 1 ms，FlashInfer 的 top-k 只要 88 us）。另外，不排序的实现更容易做到每个请求使用不同的 k / p，并且可以把采样直接融合进同一个 kernel，不必先把过滤后的分布写回显存。

核心 · Track F · FlashInfer · 第 17 题

F5 FlashInfer 贡献实战：源码地图、JIT、测试、benchmark、PR 规范

从使用者变成贡献者：搭好 FlashInfer 的开发环境，沿着一个算子从 Python 追到 CUDA，改一行看 JIT 重新编译，跑官方 benchmark，然后按项目规范准备第一个 PR。

180 min | 难度 中 | 实践 · 在 FlashInfer 仓库里动手，没有 check.sh

1. 背景：为什么是 FlashInfer
2. 源码与资料
3. 源码地图
4. 图解与类比
5. Lab 1：开发环境
6. Lab 2：沿一个算子追到底（RMSNorm）
7. Lab 3：用官方工具做 benchmark
8. Lab 4：选一个贡献方向
9. 新算子的标准流程（来自项目的 CLAUDE.md）
10. PR 规范（最常被忽略的几条）
11. 自检清单

背景：为什么是 FlashInfer

FlashInfer 被 vLLM、SGLang、TensorRT-LLM 等主流 serving 框架使用，并且在持续加入新模型需要的算子：MLA、GDN / KDA 等线性 attention、NVFP4 / MXFP8 GEMM 和 MoE、DiT 的融合 norm、sm_110 / sm_120 的专用 kernel。它的代码同时包含 CUDA C++（`include/`）、CuTe DSL（`flashinfer/cute_dsl`）、Triton（`flashinfer/triton`），以及对 cuDNN、TRT-LLM、CUTLASS 后端的封装，所以对想做 kernel 工作的人来说，它几乎覆盖了整个技术栈。

本页的内容都核对过 FlashInfer 主分支 2026-11-02 的源码和它自带的 `CONTRIBUTING.md` / `CLAUDE.md`。项目变化很快，动手前以仓库里的最新版本为准。

源码与资料

规范：flashinfer › CONTRIBUTING.md · flashinfer › CLAUDE.md · flashinfer › CLAUDE.md · Adding a New Operation · flashinfer › CLAUDE.md · Opening a Pull Request · flashinfer › PR 模板 · flashinfer › code_review_guidance.md · flashinfer › experimental/README.md · issues · 文档站

目录：flashinfer › flashinfer/ · flashinfer › csrc/ · flashinfer › jit/ · flashinfer › triton/ · flashinfer › cute_dsl/ · flashinfer › experimental/ ·

flashinfer › tests/ · flashinfer › benchmarks/

基础设施： flashinfer › JitSpec · flashinfer › gen_jit_spec · flashinfer › aot.py · flashinfer › flashinfer_api · flashinfer › flashinfer_experimental_api · flashinfer › is_sm90a_supported · flashinfer › flashinfer_benchmark.py · flashinfer › bench_gpu_time

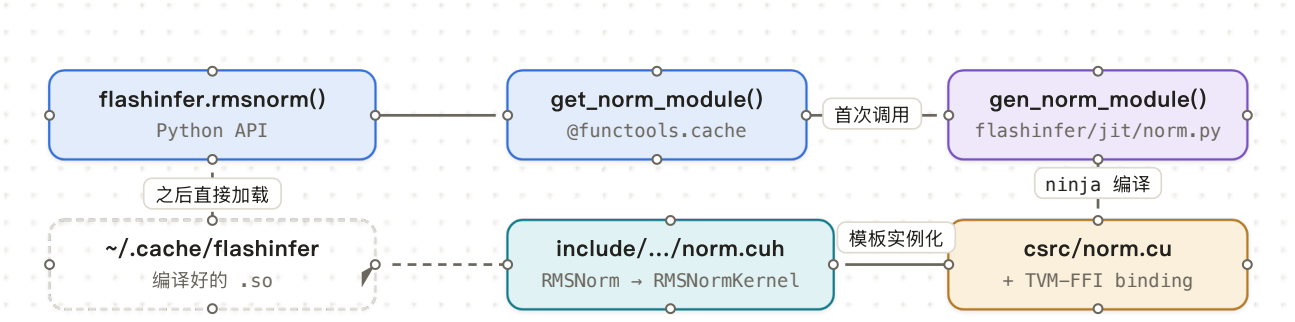
源码地图

目录	放什么	规则
`include/flashinfer/`	kernel 定义（`.cuh`）：只接收裸指针，与框架无关	**不能 include Torch 头文件**
`csrc/`	launcher 和绑定：把张量转成指针、选模板实例、调用 kernel	TVM-FFI 绑定在 `*_jit_binding.cu` / `*_binding.cu`
`flashinfer/jit/`	JIT 模块生成器：列出要编译的源文件和 nvcc 参数	例如 `jit/norm.py` 的 `gen_norm_module()`
`flashinfer/*.py`、`flashinfer/<模块>/`	Python API：参数检查、选择后端、`@functools.cache` 缓存编译好的模块	公开 API 用 `@flashinfer_api` 装饰
`flashinfer/triton/`、`flashinfer/cute_dsl/`	Triton 和 CuTe DSL 写的 kernel	其中有 `norm.py`、`activation.py`、`cascade.py`，正好对应 F2、F3
`flashinfer/experimental/`	实验性后端（比如 SM12x 消费级 GPU 的 kernel）	需要 owner 和 tracking issue
`tests/<领域>/`、`benchmarks/`	pytest 测试；`flashinfer_benchmark.py` 统一 benchmark	按 GPU 架构跳过用 `flashinfer.utils` 中的检测函数

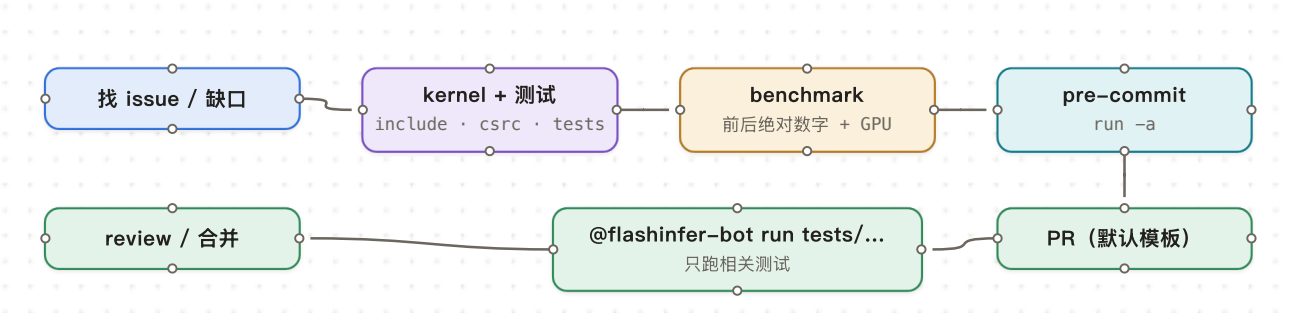
JIT 是怎么工作的：第一次调用某个 API 时，FlashInfer 按 dtype、head_dim 等参数生成专门的 CUDA 代码，用 ninja 编译，把 `.so` 缓存到 `~/.cache/flashinfer/`；之后的调用直接加载。修改 `.cuh` 后不需要手动重新构建，再次运行时它会检测到变化并重新编译。

图解与类比

rmsnorm 在 FlashInfer 里的调用链



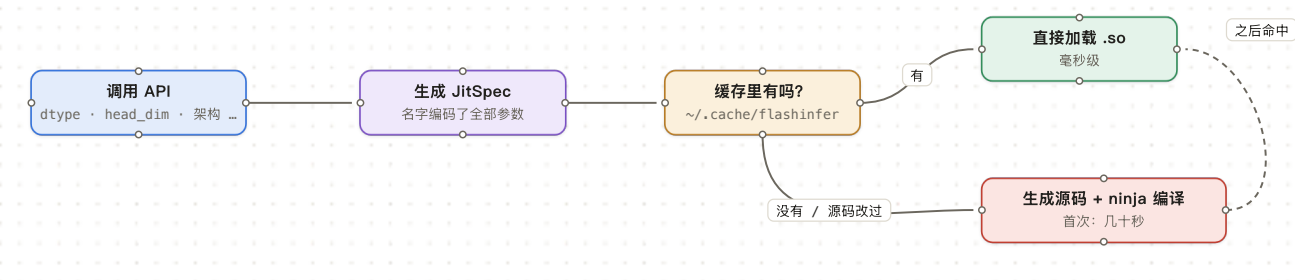
一个贡献的完整流程



类比 · JIT 是现点现做的餐厅

FlashInfer 的 kernel 组合（dtype × head_dim × 架构 × 各种开关）多到没法全部预先做好。JIT 是有人点哪道菜才做哪道，做过的菜谱冻在冰箱里（`~/.cache/flashinfer`），下次直接热；AOT 和 jit-cache 包则是开业前把最常点的菜提前做好，避免第一位客人等半分钟。

JIT 缓存命中的判断



类比 · 读源码像拆快递

> FlashInfer 的一个 API 是层层包装的快递：Python 函数（外箱）→ JIT 生成器（填充物，列出要编译什么）→ csrc 绑定（内盒，把张量换成指针）→ include 里的 kernel（真正的货物）。读源码时不要从中间撕开，按这个顺序一层层拆，每层只问一个问题：这一层负责什么？

Lab 1：开发环境

在 GPU 机器上执行（本仓库的 `setup_env.sh` 已经准备好 nvcc；FlashInfer 本身需要 CUDA 12.x 或 13.x 的 toolkit）：

```
git clone --recursive https://github.com/flashinfer-ai/flashinfer.git && cd flashin
python -m pip install --upgrade pip setuptools
pip install --no-build-isolation -e . -v        # --no-build-isolation：用你环境里已有
pip install pre-commit && pre-commit install
python -m flashinfer.collect_env                # 报 bug 时附上这份环境信息
```

`--no-build-isolation` 很重要：不加的话，pip 可能另外下载一个用不同 CUDA 版本构建的 torch。

Lab 2：沿一个算子追到底（RMSNorm）

1. `flashinfer › norm/ init .py · rmsnorm()`：参数检查，然后调用 `get_norm_module().rmsnorm(out, input, weight, eps, enable_pdl)`。
2. `flashinfer › get_norm_module()` → `flashinfer › jit/norm.py · gen_norm_module()`：列出 `flashinfer › norm.cu` 和 `flashinfer › flashinfer_norm_binding.cu`，加上 `-DENABLE_BF16 -DENABLE_FP8`。
3. `flashinfer › norm.cu`：把张量转成指针，按 dtype 分发，调用 `flashinfer::norm::RMSNorm<T>(...)`。
4. `flashinfer › norm.cuh · RMSNorm (launcher)`：`vec_size = gcd(16 / sizeof(T), d)`，每行一个 block，`cudaLaunchKernelEx` 加上 PDL 属性。kernel 本体是 `flashinfer › RMSNormKernel`：向量化加载、warp 归约、shared memory 汇总。把它和你的 T3、F3 对照着读。

动手：

```
export FLASHINFER_JIT_VERBOSE=1
pytest tests/norm -k "rmsnorm and not cake" -x -q    # 第一次运行会看到 JIT 编译日志
# 在 RMSNormKernel 里随便改一处（比如加一行注释），再运行一次：会看到它重新编译
```

Lab 3：用官方工具做 benchmark

```
pip install -U cupti-python        # CUPTI 计时，比 CUDA event 更准；没有时自动回退
python benchmarks/flashinfer_benchmark.py \
    --routine BatchDecodeWithPagedKVCacheWrapper --backends fa2 \
    --batch_size 32 --s_kv 2048 --num_qo_heads 32 --num_kv_heads 8 \
    --head_dim_qk 128 --head_dim_vo 128 --page_size 16 --refcheck -vv
```

- 在 RTX 5090 上只能使用 `fa2` 等通用后端；`trtllm-gen`、`cutlass` 的部分后端需要 sm_100。先看清楚哪些后端支持你的 GPU。
- 写 benchmark 代码时用 `flashinfer.testing.bench_gpu_time`（返回每次迭代的耗时列表，取中位数）。
- 把这里的数字和你的 F1 / L3 对照。

Lab 4：选一个贡献方向

按入门难度由低到高：

方向	例子	你在本套题里已有的准备
文档与示例	补全 docstring 中缺少的形状 / dtype 说明；写一个 CSR 分页格式的示例	F1
测试	给某个 API 加上边界情况（非 2 的幂的 head_dim、空序列、非连续输入）；补全 sm_120 上的测试覆盖	所有题的测试写法
benchmark	给某个还没有 benchmark 的 API 写 routine，比较不同后端	每题的“测完之后”
消费级 GPU 支持	找出在 RTX 5090（sm_120）上不能运行或运行很慢的 API，按 experimental 规范补一个后端	你有 5090；L、F、M 系列
Triton 实现	`flashinfer/triton/` 中算子的性能或功能改进（如 merge_state 的向量化）	T 系列、F2、F3
新算子	新模型需要的融合算子（新的 norm 变体、量化格式、attention 变体）	全部

寻找切入点：在 GitHub issues 中搜索与你的硬件或熟悉的算子有关的内容；读 `docs/design_docs/`；跑一遍 `pytest tests/<你关心的目录>`，看看在你的 GPU 上哪些测试被跳过、哪些失败。

新算子的标准流程（来自项目的 CLAUDE.md）

1. 在 `include/flashinfer/new_op.cuh` 中写 kernel（框架无关，只用裸指针）。
2. 在 `csrc/new_op.cu` 中写 launcher（处理 PyTorch 张量）。
3. 在 `csrc/new_op_jit_binding.cu` 中写 TVM-FFI 绑定。
4. （可选）需要按类型特化时，写 Jinja 模板。
5. 在 `flashinfer/jit/new_op.py` 中写 JIT 模块生成器。
6. 在 `flashinfer/new_op.py` 中写 Python API，用 `@functools.cache` 缓存模块。
7. 在 `tests/` 中写测试。
8. 在 `flashinfer/aot.py` 中注册 AOT 编译。
9. 在 `flashinfer/__init__.py` 中导出。
10. 在 `flashinfer/trace/templates/` 中加 `TraceTemplate`，用 `@flashinfer_api(trace=...)` 接上。
11. 在 `tests/trace/example.py` 中加调用示例，重新生成并提交 `fi_trace_out/` 下的 JSON。

官方推荐的参考实现：简单的看 `flashinfer/norm/__init__.py`（RMSNorm，不用 Jinja），中等的看 `flashinfer/sampling.py`（用了 Jinja），复杂的看 `flashinfer/decode.py`（plan / run 模式，workspace 管理）。

PR 规范（最常被忽略的几条）

- **保留默认的 PR 模板**（`.github/pull_request_template.md`）：Description、Related Issues、Checklist、Tests、Reviewer Notes。不要换成自己的格式：squash merge 之后，标题和描述就是 commit message。
- **性能 PR 必须给出改动前后的绝对数字**：用可复现的 benchmark（比如 `flashinfer_benchmark.py`），写明 GPU 型号和问题规模。只给加速比、或者不说 GPU 型号，都不够。
- **提交前运行** `pre-commit run -a`。
- **CI 默认只跑相关的测试**：在 PR 中评论 `@flashinfer-bot run tests/<目录或文件>`，只写与改动有关的路径；不带路径会跑全部测试，代价很高。
- **experimental 规范**：消费级 GPU（SM12x）的 kernel、针对最新模型的新算子，走 experimental 路线：后端放在 `flashinfer/experimental/`，API 用 `@flashinfer_experimental_api` 标记，需要一个写明 owner、原因和转正计划的 tracking issue。`backend="auto"` 只有在设置 `FLASHINFER_ALLOW_EXPERIMENTAL_AUTO_BACKENDS=1` 时才会选到实验性后端。

自检清单

- [ ] 能画出 `rmsnorm` 从 Python 到 kernel 的调用链，并说出每一层的职责
- [ ] 知道 JIT 缓存在哪、怎么看生成的代码（`FLASHINFER_JIT_VERBOSE=1`）、怎么清缓存
- [ ] 在自己的 GPU 上跑通了一个测试目录和一个 benchmark routine
- [ ] 能用一句话说清 plan / run、CSR 分页格式、base-2 LSE
- [ ] 选好了第一个贡献方向，并在 issues 里确认没人在做

▼ 面试追问：FlashInfer 为什么用 JIT，而不是把所有 kernel 预先编译好?

attention 这类 kernel 的模板参数组合非常多：dtype（q、kv、o 各自不同）× head_dim × 位置编码 × 是否滑动窗口 × 是否 soft cap × 架构（sm_80 / 90 / 100 / 120）……全部预编译，wheel 会非常大，编译时间也难以接受，而大部分组合没有人用。JIT 只编译实际用到的组合并缓存，新架构、新变体也可以很快加入。对部署来说，可以用 AOT（`flashinfer/aot.py`）或预编译的 cubin / JIT cache 包，提前把需要的组合编译好，避免线上第一次调用时卡几十秒。

进 阶

Track G · Layout / CuTe

数据在 warp / lane / register 上怎么分布；CuTe 的布局代数。

进阶 · Track G · Layout / CuTe · 第 18 题

G1 Layout 计算器：BlockedLayout / SliceLayout 到线程的映射

Triton 替你决定、Gluon 让你自己决定的那件事：一个块级张量的每个元素到底在哪个 warp、哪个 lane、哪个寄存器里。用纯 Python 把它算出来。

45 min | 难度 中 | 你写 `mine/g1_layouts.py` | 验收 `./check.sh g1` | 参考解 `solutions/g1_layouts.py`

1. 背景
2. 源码与资料
3. 要用到的知识：BlockedLayout
4. 要用到的知识：SliceLayout、归约与合并访存
5. 图解与类比
6. 题目要求
7. 题干
8. 验收
9. 提示
10. 参考解
11. 测完之后：把它和真实编译结果对上

背景

Triton 代码里，一个 `[64, 64]` 的张量看起来像一个整体；但在硬件上，它被拆给 4 个 warp × 32 个 lane，每个线程持有若干寄存器。这种拆法叫 **layout**，它决定了：

- 一次 load 能不能向量化、是否合并访存；
- `tl.sum(x, axis=1)` 是在线程内完成、需要 warp shuffle，还是需要经过 shared memory；
- 两个张量 layout 不同时，必须做一次 layout 转换（通常要经过 shared memory，很贵）。

原计划 Week 6–7 的 Gluon 学习检查点，第一条就是“能解释一个 tensor 的元素如何映射到不同线程”。这道题把这件事变成可以测试的代码，不需要 GPU，也不依赖某个 Gluon 版本的 API。

源码与资料

`triton › Gluon 01-intro` · `triton › Gluon 02-layouts` · `triton › BlockedEncodingAttr 定义` · `triton › LinearLayoutConversions.cpp（layout → 线性布局）` · Linear Layouts 论文

要用到的知识：BlockedLayout

```
BlockedLayout(size_per_thread  = [1, 4],   # 每个线程拿 1 行 × 4 列连续元素（一个 chunk）
              threads_per_warp = [4, 8],   # 一个 warp 的 32 个 lane 排成 4 × 8
              warps_per_cta    = [4, 1],   # 4 个 warp 沿行方向叠起来
              order            = [1, 0])   # 编号时维度 1（列）变化最快
```

一个 CTA tile 覆盖 `tile[d] = spt[d] · tpw[d] · wpc[d]` 个元素：这里是 `[1·4·4, 4·8·1] = [16, 32]`。

线程 (warp, lane) 的第 s 个 chunk 元素、第 rep 次重复，对应的坐标：

```
lane_c = 按 order 把 lane 拆成各维坐标（threads_per_warp 为各维大小）
warp_c = 按 order 把 warp 拆成各维坐标（warps_per_cta 为各维大小）
coord[d] = rep[d] · tile[d] + warp_c[d] · (spt[d] · tpw[d]) + lane_c[d] · spt[d] + 
coord[d] %= shape[d]          # 张量比 tile 小时取模：多个线程持有同一元素（broadcast）
```

- 张量比 tile **大**：tile 沿该维重复 `shape[d] / tile[d]` 次，每个线程多出相应倍数的寄存器。
- **寄存器编号**（本题约定，和 Triton 的编号方式一致）：先编一个 chunk 内的元素，再编重复次数，两者都按 `order` 线性化。
- “按 order 拆分”：`order[0]` 是最快变化的维度。比如 `tpw = [4, 8]`、`order = [1, 0]` 时，lane 9 → 维度 1 取 `9 % 8 = 1`，维度 0 取 `9 // 8 = 1`，即 lane 坐标 (1, 1)。

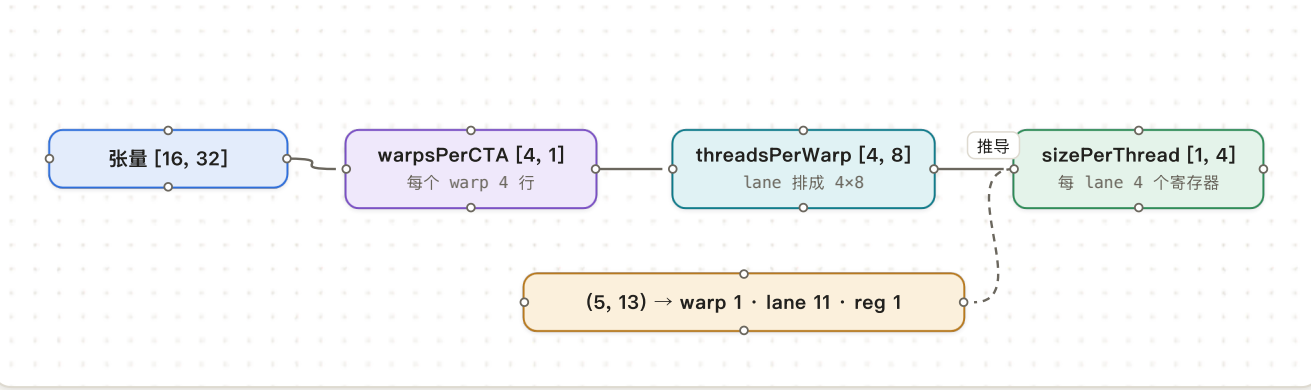
例子（shape `[16, 32]`）：warp 0 lane 0 持有 `(0,0) (0,1) (0,2) (0,3)`；lane 1 持有 `(0,4)…(0,7)`；lane 8 持有 `(1,0)…(1,3)`；warp 1 lane 0 持有 `(4,0)…(4,3)`。

要用到的知识：SliceLayout、归约与合并访存

- **SliceLayout(dim, parent)**：从 parent 中去掉 dim 这一维得到的一维 layout，`tl.sum(x, axis=dim)` 的结果就是这种 layout。线程持有的是它在 parent 中元素坐标去掉 dim 之后的值（去重）。例如沿 axis=1 求和后，warp 0 的 lane 0–7 都持有第 0 行的和：同一个值复制了 8 份。
- **归约的代价**沿 `axis` 分三层：线程内 `spt[axis] · reps[axis]` 个值先相加；然后在 lane 之间做 `log2(tpw[axis])` 轮 butterfly shuffle；如果 `wpc[axis] > 1`，还要把各 warp 的部分和经过 shared memory 和一次 barrier 汇合，这一步最贵。
- **合并访存**：一个线程一条指令最多读 16 字节，即 `vec = min(spt[order[0]], 16 / dtype_bytes)` 个元素。数一下一个 warp 的第一条 load 用到了多少个 32 字节 sector：理想情况下 `sector 数 × 32 = 请求的字节数`。

图解与类比

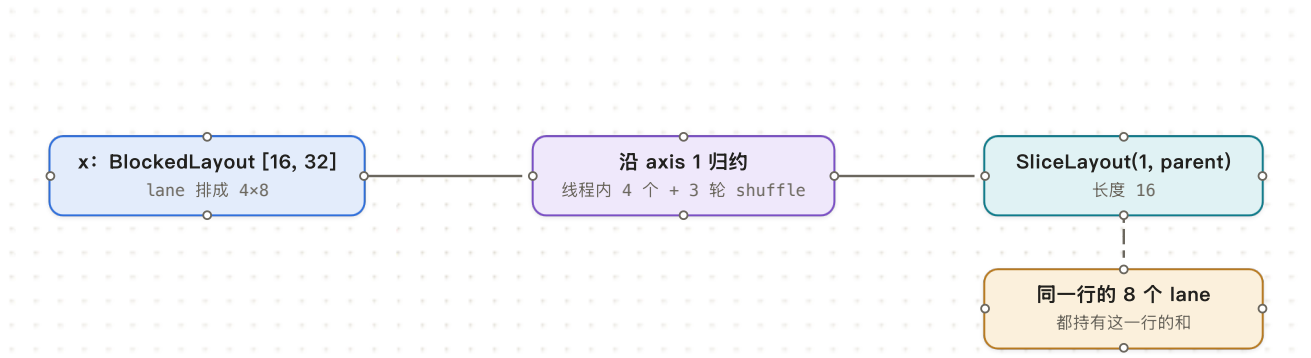
BlockedLayout：元素 → warp → lane → 寄存器



类比 · 发扑克牌

先把牌桌分成几个区（warpsPerCTA），每个区里按座位（threadsPerWarp）发牌，每人一次拿几张连号的牌（sizePerThread）；`order` 决定是先横着发还是先竖着发。 连号的牌正好是内存里相邻的地址时，一个人一次就能拿起一整摞（向量化），一桌人拿的牌也挨在一起（合并访存）。

tl.sum(x, axis=1) 之后：SliceLayout



类比 · 开会之后每人都有一份纪要

一排 8 个人（同一行的 8 个 lane）各自算了一部分，开完会（butterfly 归约）之后，每个人手里都有一份完全相同的会议纪要（这一行的和）。这就是 SliceLayout 里的“复制”：同一个值存在多个线程里，后续每个线程都能直接用，不必再广播。

题目要求

在 `mine/g1_layouts.py` 里实现（`BlockedLayout` 数据类和 `tile()` 已给出）：

函数	返回
`_delinearize(index, sizes, order)`	把编号拆成各维坐标，`order[0]` 最快
`reps(layout, shape)`	每维的重复次数（至少 1）
`thread_elements(layout, shape, warp, lane)`	按寄存器编号排列的坐标列表
`owners(layout, shape, coord)`	持有该元素的所有 `(warp, lane, reg)`
`slice_elements(parent, parent_shape, dim, warp, lane)`	SliceLayout 下该线程持有的坐标（去重、保持首次出现的顺序）
`reduce_plan(layout, shape, axis)`	`{"in_thread", "shuffle_rounds", "cross_warp"}`
`warp_load_sectors(layout, shape, dtype_bytes, warp=0)`	`(sector 数, 请求字节数)`，按行主序计算地址

题干

mine/g1_layouts.py 114 行

```
"""G1 · Layout calculator for Triton / Gluon BlockedLayout and SliceLayout — YOUR IMPLEMENTATION.

Pure Python, runs anywhere. A BlockedLayout says how the elements of a block-level tensor are
spread over the registers, lanes (threads of a warp) and warps of one program:

    size_per_thread  [d] : contiguous elements a thread owns along dim d (one "chunk")
    threads_per_warp [d] : lanes laid out along dim d          (product must be 32)
    warps_per_cta    [d] : warps laid out along dim d          (product = num_warps)
    order                : dims from fastest- to slowest-varying when numbering lanes,
                           warps and registers (order[0] is the fastest)

One "CTA tile" covers tile[d] = spt[d] * tpw[d] * wpc[d] elements along each dim:
    coord[d] = rep[d] * tile[d] + warp_c[d] * (spt[d] * tpw[d]) + lane_c[d] * spt[d] + s[d]
If the tensor is larger than the tile, the pattern repeats (each thread gets more registers);
```

```
if it is smaller, coordinates wrap around (several threads hold the same element:
broadcast).

Register numbering used in this drill (it matches how Triton numbers registers fo
r blocked
layouts): the elements of one size_per_thread chunk come first (linearised by `or
der`),
then the repetitions (also linearised by `order`).
"""
from dataclasses import dataclass
from itertools import product
from math import log2, prod


@dataclass(frozen=True)
class BlockedLayout:
    size_per_thread: tuple[int, ...]
    threads_per_warp: tuple[int, ...]
    warps_per_cta: tuple[int, ...]
    order: tuple[int, ...]

    def __post_init__(self):
        r = len(self.size_per_thread)
        assert len(self.threads_per_warp) == len(self.warps_per_cta) == len(self.
order) == r
        assert sorted(self.order) == list(range(r)), "order must be a permutation
of the dims"
        assert prod(self.threads_per_warp) == 32, "a warp has 32 lanes"

    @property
    def rank(self) -> int:
        return len(self.size_per_thread)

    @property
    def num_warps(self) -> int:
        return prod(self.warps_per_cta)

    def tile(self) -> tuple[int, ...]:
        """Elements covered by one CTA tile along each dim."""
        return tuple(s * t * w for s, t, w in zip(self.size_per_thread, self.thre
ads_per_warp, self.warps_per_cta))


def _delinearize(index: int, sizes: tuple[int, ...], order: tuple[int, ...]) -> l
ist[int]:
    """Inverse of numbering a multi-index with order[0] as the fastest-varying di
m."""
    # TODO: split `index` into per-dim coords, order[0] varying fastest
    raise NotImplementedError
```

```
def _linearize(coords, sizes, order) -> int:
    # TODO: inverse of _delinearize (optional helper)
    raise NotImplementedError


def reps(layout: BlockedLayout, shape: tuple[int, ...]) -> tuple[int, ...]:
    """How many times the CTA tile repeats along each dim (1 if the tensor is sma
ller)."""
    # TODO: max(1, shape[d] // tile[d]) per dim
    raise NotImplementedError


def thread_elements(layout: BlockedLayout, shape: tuple[int, ...], warp: int, lan
e: int) -> list[tuple[int, ...]]:
    """Coordinates held by (warp, lane), indexed by register number."""
    # TODO: loop repetitions (outer) x chunk elements (inner); coord formula in t
he module docstring; wrap with % shape
    raise NotImplementedError


def owners(layout: BlockedLayout, shape: tuple[int, ...], coord: tuple[int, ...])
-> list[tuple[int, int, int]]:
    """Every (warp, lane, register) holding element `coord` (more than one when b
roadcast)."""
    # TODO: search every (warp, lane, reg)
    raise NotImplementedError


def slice_elements(parent: BlockedLayout, parent_shape: tuple[int, ...], dim: in
t,
                   warp: int, lane: int) -> list[tuple[int, ...]]:
    """SliceLayout(dim, parent): the layout of a tensor obtained by removing `dim
`, e.g. the
    result of tl.sum(x, axis=dim). A thread holds the remaining coordinates of it
s parent
    elements; duplicates collapse, first occurrence keeps its register order."""
    # TODO: drop `dim` from each parent coordinate, de-duplicate keeping first oc
currence
    raise NotImplementedError


def reduce_plan(layout: BlockedLayout, shape: tuple[int, ...], axis: int) -> dic
t:
    """What tl.sum / tl.max along `axis` costs with this layout.

    in_thread      values each thread combines in its own registers for one outpu
t
    shuffle_rounds butterfly __shfl_xor rounds across the lanes laid out along `a
xis`
```

```
    cross_warp     True if warps are laid out along `axis`: partials go through s
hared
                   memory plus a barrier, the expensive part
    """
    # TODO: in_thread = spt[axis] * reps[axis]; shuffle_rounds = log2(tpw[axis]);
cross_warp = wpc[axis] > 1
    raise NotImplementedError


def warp_load_sectors(layout: BlockedLayout, shape: tuple[int, ...], dtype_bytes:
int, warp: int = 0) -> tuple[int, int]:
    """Memory efficiency of the first vectorised load a warp issues on a row-majo
r tensor.

    Each lane loads the first `vec` registers of its chunk in one instruction, wh
ere
    vec = min(spt[order[0]], 16 // dtype_bytes) (at most 16 bytes per lane per in
struction).
    Returns (32-byte sectors touched, bytes actually requested). Ideal: sectors *
32 == bytes.
    """
    # TODO: vec = min(spt[order[0]], 16 // dtype_bytes); collect 32-byte sector i
ds of every lane's first vec elements
    raise NotImplementedError


def all_coords(shape):
    return list(product(*(range(n) for n in shape)))
```

验收

```
./check.sh g1                # 纯 Python，几毫秒
```

测试里的期望值是**按定义手算**的。每个断言旁边的注释说明了推导过程，卡住时可以先看测试。

tests/test_g1_layouts.py 89 行

```
"""Expected values below are worked out by hand from the layout definition (see t
he drill page)."""
from drill import load

# 4 warps stacked along rows; inside a warp 4 x 8 lanes; each lane owns 1 x 4 con
tiguous elements.
A = dict(size_per_thread=(1, 4), threads_per_warp=(4, 8), warps_per_cta=(4, 1), o
rder=(1, 0))
# The "wrong" orientation for a row-major tensor: lanes run down the columns.
BAD = dict(size_per_thread=(4, 1), threads_per_warp=(8, 4), warps_per_cta=(4, 1),
order=(0, 1))
```

```
def L(m, d):
    return m.BlockedLayout(**d)


def test_tile_and_reps():
    m = load("g1_layouts")
    lay = L(m, A)
    assert lay.tile() == (16, 32)
    assert m.reps(lay, (16, 32)) == (1, 1)
    assert m.reps(lay, (64, 64)) == (4, 2)
    assert m.reps(lay, (8, 32)) == (1, 1)      # smaller than the tile -> broadca
st, no repetition


def test_thread_elements_basic():
    m = load("g1_layouts")
    lay = L(m, A)
    assert m.thread_elements(lay, (16, 32), warp=0, lane=0) == [(0, 0), (0, 1),
(0, 2), (0, 3)]
    assert m.thread_elements(lay, (16, 32), warp=0, lane=1) == [(0, 4), (0, 5),
(0, 6), (0, 7)]
    assert m.thread_elements(lay, (16, 32), warp=0, lane=8) == [(1, 0), (1, 1),
(1, 2), (1, 3)]
    assert m.thread_elements(lay, (16, 32), warp=1, lane=0) == [(4, 0), (4, 1),
(4, 2), (4, 3)]
    assert m.thread_elements(lay, (16, 32), warp=3, lane=31) == [(15, 28), (15, 2
9), (15, 30), (15, 31)]


def test_thread_elements_with_repetition():
    m = load("g1_layouts")
    lay = L(m, A)
    # (32, 32): the tile repeats twice along rows -> 8 registers, second chunk 16
rows lower.
    assert m.thread_elements(lay, (32, 32), warp=0, lane=0) == [
        (0, 0), (0, 1), (0, 2), (0, 3), (16, 0), (16, 1), (16, 2), (16, 3)]


def test_every_element_has_exactly_one_owner_without_broadcast():
    m = load("g1_layouts")
    lay = L(m, A)
    shape = (32, 64)
    seen = {}
    for w in range(lay.num_warps):
        for ln in range(32):
            for c in m.thread_elements(lay, shape, w, ln):
                seen[c] = seen.get(c, 0) + 1
    assert len(seen) == 32 * 64 and set(seen.values()) == {1}
```

```
def test_owners():
    m = load("g1_layouts")
    lay = L(m, A)
    # row 5 -> warp 1 (rows 4..7), lane row 1; col 13 -> lane col 3, element 1 of
the chunk
    assert m.owners(lay, (16, 32), (5, 13)) == [(1, 11, 1)]
    # (8, 32) is half the tile tall: rows wrap, so warps 2 and 3 duplicate warps
0 and 1.
    assert m.owners(lay, (8, 32), (0, 0)) == [(0, 0, 0), (2, 0, 0)]


def test_slice_layout():
    m = load("g1_layouts")
    lay = L(m, A)
    # tl.sum(x, axis=1) keeps rows: lanes 0..7 of warp 0 all end up holding row
0.
    assert m.slice_elements(lay, (16, 32), dim=1, warp=0, lane=0) == [(0,)]
    assert m.slice_elements(lay, (16, 32), dim=1, warp=0, lane=7) == [(0,)]
    assert m.slice_elements(lay, (16, 32), dim=1, warp=0, lane=8) == [(1,)]
    # tl.sum(x, axis=0) keeps columns: lane 0 holds columns 0..3.
    assert m.slice_elements(lay, (16, 32), dim=0, warp=0, lane=0) == [(0,), (1,),
(2,), (3,)]


def test_reduce_plan():
    m = load("g1_layouts")
    lay = L(m, A)
    assert m.reduce_plan(lay, (16, 32), axis=1) == {"in_thread": 4, "shuffle_roun
ds": 3, "cross_warp": False}
    assert m.reduce_plan(lay, (16, 32), axis=0) == {"in_thread": 1, "shuffle_roun
ds": 2, "cross_warp": True}
    assert m.reduce_plan(lay, (16, 128), axis=1)["in_thread"] == 16    # 4 per ch
unk x 4 repetitions


def test_warp_load_sectors():
    m = load("g1_layouts")
    # Good layout, fp32: 4 rows x 128 B = 16 sectors for 512 B -> 100% efficient.
    assert m.warp_load_sectors(L(m, A), (16, 32), 4) == (16, 512)
    # fp16: 4 halves per lane (8 B): 4 rows x 64 B = 8 sectors for 256 B.
    assert m.warp_load_sectors(L(m, A), (16, 32), 2) == (8, 256)
    # Bad orientation: each lane's 4 "contiguous" registers sit in 4 different ro
ws.
    # 32 rows x 4 columns (16 B each) -> 32 sectors (1024 B moved) for 512 B use
d: 50%.
    assert m.warp_load_sectors(L(m, BAD), (128, 32), 4) == (32, 512)
```

提示

L1 先写 _delinearize 方向（轻度剧透） 收起

`for d in order: coords[d] = index % sizes[d]; index //= sizes[d]`。lane 坐标、warp 坐标、chunk 内坐标、重复坐标全都用它。

L2 thread_elements 的两层循环 拆解与常见坑（中度剧透） 收起

外层循环重复次数 `rep_i in range(prod(reps))`，内层循环 chunk 内元素 `s_i in range(prod(spt))`，各自用 `_delinearize` 拆成坐标，然后套用坐标公式并对 shape 取模。这个循环顺序就是寄存器编号的顺序。

L3 其余函数都是 thread_elements 的组合 几乎是答案（重度剧透） 收起

- `owners`：枚举所有 `(warp, lane)` 和寄存器，比较坐标。
- `slice_elements`：`rest = c[:dim] + c[dim+1:]`，用 set 去重。
- `reduce_plan`：直接套公式（`int(log2(tpw[axis]))`）。
- `warp_load_sectors`：对 32 个 lane 取 `thread_elements(...)[:vec]`，地址 = `Σ coord[d] · stride[d] · bytes`，行主序 `stride[d] = prod(shape[d+1:])`，sector 编号是 `addr // 32`，统计集合大小。

参考解

解 **solutions/g1_layouts.py** 先自己写完、跑过测试再看 收起

solutions/g1_layouts.py 157 行

```
"""G1 · Layout calculator for Triton / Gluon BlockedLayout and SliceLayout —
reference solution.

Pure Python, runs anywhere. A BlockedLayout says how the elements of a block-
level tensor are
spread over the registers, lanes (threads of a warp) and warps of one progra
m:

    size_per_thread  [d] : contiguous elements a thread owns along dim d (one
"chunk")
    threads_per_warp [d] : lanes laid out along dim d          (product must
be 32)
    warps_per_cta    [d] : warps laid out along dim d          (product = num
_warps)
    order                : dims from fastest- to slowest-varying when numberi
ng lanes,
                           warps and registers (order[0] is the fastest)

One "CTA tile" covers tile[d] = spt[d] * tpw[d] * wpc[d] elements along each
dim:
    coord[d] = rep[d] * tile[d] + warp_c[d] * (spt[d] * tpw[d]) + lane_c[d] *
spt[d] + s[d]
If the tensor is larger than the tile, the pattern repeats (each thread gets
more registers);
if it is smaller, coordinates wrap around (several threads hold the same elem
ent: broadcast).

Register numbering used in this drill (it matches how Triton numbers register
s for blocked
layouts): the elements of one size_per_thread chunk come first (linearised by
`order`),
then the repetitions (also linearised by `order`).
"""
from dataclasses import dataclass
from itertools import product
from math import log2, prod


@dataclass(frozen=True)
class BlockedLayout:
    size_per_thread: tuple[int, ...]
    threads_per_warp: tuple[int, ...]
```

```
    warps_per_cta: tuple[int, ...]
    order: tuple[int, ...]

    def __post_init__(self):
        r = len(self.size_per_thread)
        assert len(self.threads_per_warp) == len(self.warps_per_cta) == len(s
elf.order) == r
        assert sorted(self.order) == list(range(r)), "order must be a permuta
tion of the dims"
        assert prod(self.threads_per_warp) == 32, "a warp has 32 lanes"

    @property
    def rank(self) -> int:
        return len(self.size_per_thread)

    @property
    def num_warps(self) -> int:
        return prod(self.warps_per_cta)

    def tile(self) -> tuple[int, ...]:
        """Elements covered by one CTA tile along each dim."""
        return tuple(s * t * w for s, t, w in zip(self.size_per_thread, self.
threads_per_warp, self.warps_per_cta))


def _delinearize(index: int, sizes: tuple[int, ...], order: tuple[int, ...])
-> list[int]:
    """Inverse of numbering a multi-index with order[0] as the fastest-varyin
g dim."""
    coords = [0] * len(sizes)
    for d in order:
        coords[d] = index % sizes[d]
        index //= sizes[d]
    return coords


def _linearize(coords, sizes, order) -> int:
    index, mult = 0, 1
    for d in order:
        index += coords[d] * mult
        mult *= sizes[d]
    return index


def reps(layout: BlockedLayout, shape: tuple[int, ...]) -> tuple[int, ...]:
    """How many times the CTA tile repeats along each dim (1 if the tensor is
smaller)."""
    out = []
    for n, t in zip(shape, layout.tile()):
        assert n % t == 0 or t % n == 0, "shape and tile must divide each oth
```

```
er (powers of two)"
        out.append(max(1, n // t))
    return tuple(out)


def thread_elements(layout: BlockedLayout, shape: tuple[int, ...], warp: int,
lane: int) -> list[tuple[int, ...]]:
    """Coordinates held by (warp, lane), indexed by register number."""
    spt, tpw, wpc, order = layout.size_per_thread, layout.threads_per_warp, l
ayout.warps_per_cta, layout.order
    lane_c = _delinearize(lane, tpw, order)    # this lane's position inside
the warp
    warp_c = _delinearize(warp, wpc, order)    # this warp's position inside
the CTA
    tile = layout.tile()
    r = reps(layout, shape)
    out = []
    for rep_i in range(prod(r)):                        # repetitions: outer
        rep_c = _delinearize(rep_i, r, order)
        for s_i in range(prod(spt)):                    # elements of one chun
k: inner
            s_c = _delinearize(s_i, spt, order)
            coord = tuple(
                (rep_c[d] * tile[d] + warp_c[d] * spt[d] * tpw[d] + lane_c[d]
* spt[d] + s_c[d]) % shape[d]
                for d in range(layout.rank)
            )
            out.append(coord)
    return out


def owners(layout: BlockedLayout, shape: tuple[int, ...], coord: tuple[int,
...]) -> list[tuple[int, int, int]]:
    """Every (warp, lane, register) holding element `coord` (more than one wh
en broadcast)."""
    found = []
    for warp in range(layout.num_warps):
        for lane in range(32):
            for reg, c in enumerate(thread_elements(layout, shape, warp, lan
e)):
                if c == tuple(coord):
                    found.append((warp, lane, reg))
    return found


def slice_elements(parent: BlockedLayout, parent_shape: tuple[int, ...], dim:
int,
                   warp: int, lane: int) -> list[tuple[int, ...]]:
    """SliceLayout(dim, parent): the layout of a tensor obtained by removing
`dim`, e.g. the
```

```
    result of tl.sum(x, axis=dim). A thread holds the remaining coordinates o
f its parent
    elements; duplicates collapse, first occurrence keeps its register orde
r."""
    seen, out = set(), []
    for c in thread_elements(parent, parent_shape, warp, lane):
        rest = c[:dim] + c[dim + 1:]
        if rest not in seen:
            seen.add(rest)
            out.append(rest)
    return out


def reduce_plan(layout: BlockedLayout, shape: tuple[int, ...], axis: int) ->
dict:
    """What tl.sum / tl.max along `axis` costs with this layout.

    in_thread      values each thread combines in its own registers for one o
utput
    shuffle_rounds butterfly __shfl_xor rounds across the lanes laid out alon
g `axis`
    cross_warp     True if warps are laid out along `axis`: partials go throu
gh shared
                   memory plus a barrier, the expensive part
    """
    r = reps(layout, shape)
    return {
        "in_thread": layout.size_per_thread[axis] * r[axis],
        "shuffle_rounds": int(log2(layout.threads_per_warp[axis])),
        "cross_warp": layout.warps_per_cta[axis] > 1,
    }


def warp_load_sectors(layout: BlockedLayout, shape: tuple[int, ...], dtype_by
tes: int, warp: int = 0) -> tuple[int, int]:
    """Memory efficiency of the first vectorised load a warp issues on a row-
major tensor.

    Each lane loads the first `vec` registers of its chunk in one instructio
n, where
    vec = min(spt[order[0]], 16 // dtype_bytes) (at most 16 bytes per lane pe
r instruction).
    Returns (32-byte sectors touched, bytes actually requested). Ideal: secto
rs * 32 == bytes.
    """
    vec = max(1, min(layout.size_per_thread[layout.order[0]], 16 // dtype_byt
es))
    strides = [prod(shape[d + 1:]) for d in range(len(shape))]     # row-majo
r element strides
    sectors = set()
```

```
    for lane in range(32):
        for c in thread_elements(layout, shape, warp, lane)[:vec]:
            addr = sum(ci * si for ci, si in zip(c, strides)) * dtype_bytes
            sectors.add(addr // 32)
    return len(sectors), 32 * vec * dtype_bytes


def all_coords(shape):
    return list(product(*(range(n) for n in shape)))
```

测完之后：把它和真实编译结果对上

在 GPU 上编译 T2 的 softmax，看 `ttgir`：

```
k = softmax_kernel[(M,)](y, x, x.stride(0), y.stride(0), N, BLOCK_SIZE=1024, num_wa
print(k.asm["ttgir"])   # 找 #blocked = #ttg.blocked<{sizePerThread = [...], threads
```

把编译器选的参数代进你的 `reduce_plan`：`tl.max` 和 `tl.sum` 是否需要跨 warp？`num_warps` 改成 1 呢？再看 T4 的 GEMM：`#mma`（Tensor Core 的累加器 layout）、`#dot_op`、`#shared` 这几种 layout 分别出现在哪，它们之间的 `ttg.convert_layout` 就是编译器替你安排的数据搬运。

接下来如果想写 Gluon，就从官方 Gluon tutorial 的第一课开始：那里的 `BlockedLayout`、`SliceLayout`、shared memory layout，现在你都能直接算出它们的含义。

▼ 面试追问：为什么 `order` 对行主序张量很重要?

`order[0]` 决定哪一维在 lane 和寄存器编号中变化最快。行主序张量里列方向在内存中连续，所以 `order = [1, 0]` 时，一个线程的 chunk 和相邻的 lane 都落在连续地址上，可以向量化、也能合并访存。反过来用 `[0, 1]`，一个线程的“连续”寄存器会落在不同的行上，同样的数据需要更多 sector（测试里的 BAD layout 正好浪费一半带宽）。

进阶 · Track G · Layout / CuTe · 第 19 题

G2 CuTe 布局代数：coalesce、composition、complement、divide

CUTLASS 3/4、CuTe DSL、FlashAttention-3/4、DeepGEMM 和 FlashInfer 的 CuTe-DSL kernel 都建立在同一套“布局代数”上。用纯 Python 实现它的六个核心操作，以后读 CuTe 代码时就能把每个 layout 当成一个函数来读。

60 min | 难度 难 | 你写 `mine/g2_cute_layout.py` | 验收 `./check.sh g2` | 参考解 `solutions/g2_cute_layout.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

G1 的 BlockedLayout 描述“张量元素如何分给线程”。CuTe 把这件事推广成一种统一的数学对象：**Layout = (Shape, Stride)，是一个从坐标到偏移的函数**。线程到数据的映射（TV layout）、MMA 指令要求的寄存器排列、shared memory 的 swizzle、一个 tile 怎么切成更小的 tile，全部用 layout 和它们之间的运算表达。写 Hopper / Blackwell 上的高性能 kernel（wgmma、TMA、tcgen05）基本绕不开 CuTe，而 CuTe DSL（CUTLASS 4 的 Python 前端）让这件事可以直接在 Python 里完成：FlashAttention-4 全部用它写成，FlashInfer 0.7 也有大量 `cute_dsl` 后端。

源码与资料

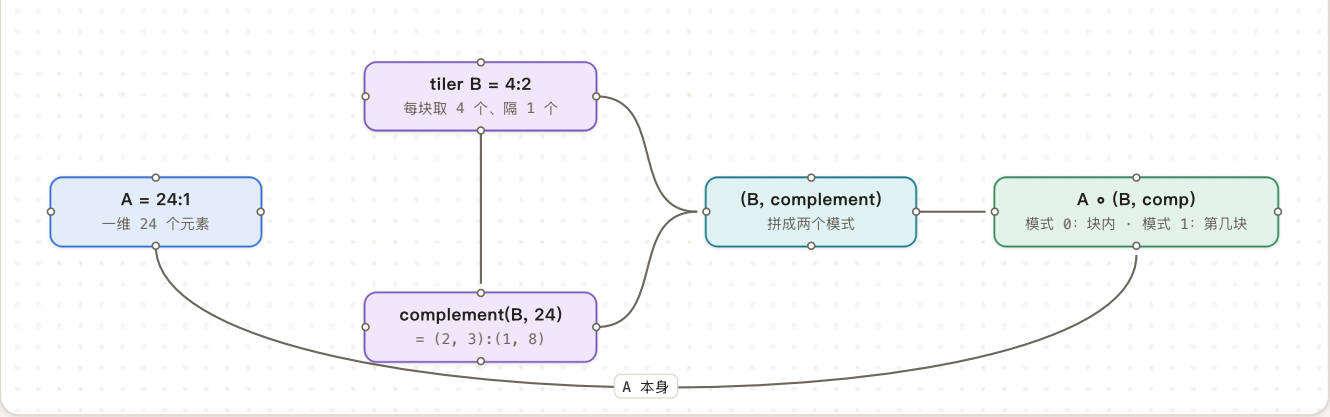
`cutlass › CuTe 01_layout.md` · `cutlass › CuTe 02_layout_algebra.md` · `cutlass › pycute composition` · `cutlass › pycute complement` · `cutlass › layout.hpp` · `cutlass › swizzle.hpp` · `cutlass › CuTe DSL 文档` · `cutlass › CuTe DSL tiled GEMM notebook` · `cutlass › CuTe DSL elementwise_add` · `flashinfer › FlashInfer cute_dsl/` · `fa › FA4 flash_attn/cute/`

要用到的知识

- **记号**：`(4,2):(1,4)` 表示 shape (4,2)、stride (1,4)，即 4×2 的列主序矩阵；`(4,2):(2,1)` 是行主序。
- **调用**：坐标 `(i, j)` → `i·1 + j·4`。整数下标先按**列字典序**（第一个模式变化最快）拆成坐标：下标 5 在 shape (4,2) 中对应 (1, 1)。shape 可以嵌套：`((2,2),2):((1,4),2)`。
- **size** = shape 中所有数的乘积；**cosize** = 最大偏移 + 1。
- **coalesce**：函数不变，模式数最少。去掉大小为 1 的模式；若 `stride_{i+1} == shape_i · stride_i`，两个模式可以合并成一个。
- **composition** `A ∘ B`：`R(i) = A(B(i))`。用 B 的布局去“选取” A 中的元素：tiling、对数据重新排列都是 composition。
- **complement** `complement(A, M)`：A 在 `[0, M)` 中留下的“空隙”的布局。A 和它的 complement 合起来恰好铺满 `[0, M)`。
- **logical_divide** `A ⊘ B = A ∘ (B, complement(B, size(A)))`：把 A 切成“每块由 B 选出的元素”（模式 0）×“块的重复”（模式 1）。CuTe 里的 `zipped_divide`、`local_tile` 都建立在它上面。

图解与类比

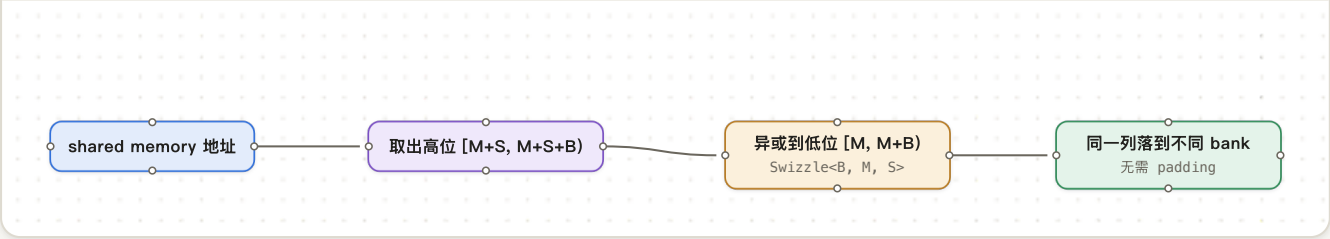
logical_divide：用布局代数切 tile



类比 · layout 是一张地址公式

Layout 就是“坐标 → 地址”的公式；composition 是先查楼层表、再查房间表；complement 是这栋楼里你还没占用的房间；logical_divide 把一栋楼按“每 4 间一组”重新编号，编号的方式由公式推出来，而不是手写下标。

swizzle：用异或打散 bank



> **类比 · 按楼层错开房号**
>
> 每层楼都有 32 个房间，大家都住 5 号房（同一列）就会挤在同一个电梯口（同一个 bank）。swizzle 规定“第 i 层的房号要和楼层号做一次异或”，于是每层的“5 号房”实际落在不同的位置，电梯口就不拥挤了；而且和 padding 不同，它一个房间都不浪费。

题目要求

在 `mine/g2_cute_layout.py` 中实现（`Layout` 数据类、`flatten` 等辅助函数已给出）：

函数	说明
`_idx2crd(idx, shape)`	列字典序，递归处理嵌套 shape
`_inner(crd, shape, stride)`	坐标与 stride 的内积；嵌套模式接收整数时先拆分
`make_layout_left / right(shape)`	紧凑的列主序 / 行主序
`coalesce(layout)`	
`composition(a, b)`	B 逐个模式处理；对 B 的一个模式 s:d，在 coalesce 后的 A 上“先跳过 d 个元素，再取 s 个”
`complement(a, cotarget)`	按 stride 排序 A 的模式，每段空隙成为一个模式
`logical_divide(a, tiler)`	

测试用例取自 CuTe 文档中的例子，并且都按**函数相等**来检查（逐点比较取值），不要求和某种特定的写法完全一致。

题干

`mine/g2_cute_layout.py` 93 行

```python
"""G2 · CuTe layout algebra — YOUR IMPLEMENTATION.

See the drill page for the spec; the reference lives in solutions/."""
from dataclasses import dataclass
from math import prod


def flatten(t):
    if isinstance(t, tuple):
        return tuple(x for e in t for x in flatten(e))
    return (t,)


def _product(shape):
    return prod(flatten(shape))


def _idx2crd(idx, shape):
    """Colexicographic: the first mode varies fastest. Follows the nesting of `sh
ape`."""
    # TODO: colexicographic: first mode fastest, recurse into nested modes
    raise NotImplementedError


def _inner(crd, shape, stride):
    # TODO: sum of coord * stride over (nested) modes; unflatten an int coord for
a nested mode first
    raise NotImplementedError


@dataclass(frozen=True)
class Layout:
    shape: object
    stride: object

    def __call__(self, crd):
        return _inner(crd, self.shape, self.stride)

    def size(self):                     # number of coordinates (the domain)
        return _product(self.shape)

    def cosize(self):                   # 1 + largest offset (the codomain the la
yout touches)
        return self(self.size() - 1) + 1 if self.size() else 0

    def __repr__(self):
        return f"{self.shape}:{self.stride}"


def make_layout_right(shape):
```

```
    """Compact row-major (LayoutRight): the LAST mode has stride 1."""
    # TODO: last mode stride 1, earlier strides = suffix products
    raise NotImplementedError


def _unflatten(shape, it):
    if isinstance(shape, tuple):
        return tuple(_unflatten(s, it) for s in shape)
    return next(it)


def make_layout_left(shape):
    """Compact column-major (LayoutLeft): stride = exclusive prefix product of th
e flattened shape."""
    # TODO: stride = exclusive prefix product of the flattened shape
    raise NotImplementedError


def coalesce(layout: Layout) -> Layout:
    """Same function, fewest modes: drop size-1 modes, merge (s0:d0, s1:d1) when
d1 == s0 * d0."""
    # TODO: drop size-1 modes; merge consecutive modes when stride_next == shape_
prev * stride_prev
    raise NotImplementedError


def composition(a: Layout, b: Layout) -> Layout:
    """R = A o B, i.e. R(i) = A(B(i)) for i < size(B). B is applied mode by mode.

    For one mode s:d of B, walk A's (coalesced) modes: skip the first d elements
("divide"),
    then keep the next s ("mod"). Requires the usual CuTe divisibility condition
s."""
    # TODO: by-mode over B; for one mode s:d walk coalesced A: skip d elements, k
eep s
    raise NotImplementedError


def complement(a: Layout, cotarget: int) -> Layout:
    """The layout of the "holes" of A in [0, cotarget): (A, complement(A)) togeth
er cover it.

    Sort A's modes by stride; each gap between where one mode ends and the next b
egins becomes
    a mode of the complement, plus a final mode up to cotarget."""
    # TODO: sort A's modes by stride; each gap becomes a mode; last mode reaches
cotarget
    raise NotImplementedError
```

```python
def logical_divide(a: Layout, tiler: Layout) -> Layout:
    """Split A into (tile, rest): mode 0 = the elements the tiler picks, mode 1 = repetitions.
    logical_divide(A, B) = A o (B, complement(B, size(A)))."""
    # TODO: A o (tiler, complement(tiler, size(A)))
    raise NotImplementedError
```

验收

```
./check.sh g2              # 纯 Python
```

tests/test_g2_cute_layout.py 71 行

```python
"""Examples are the ones used in the CuTe documentation (layout algebra chapter), checked as functions."""
from drill import load


def L(m, shape, stride):
    return m.Layout(shape, stride)


def same_function(a, b, n):
    return all(a(i) == b(i) for i in range(n))


def test_call_and_size():
    m = load("g2_cute_layout")
    col = L(m, (4, 2), (1, 4))
    row = L(m, (4, 2), (2, 1))
    assert col.size() == 8 and col.cosize() == 8
    assert [col(i) for i in range(8)] == list(range(8))
    assert [row(i) for i in range(8)] == [0, 2, 4, 6, 1, 3, 5, 7]   # index -> (i % 4, i // 4)
    assert row((1, 1)) == 3                                          # coordinate form
    nested = L(m, ((2, 2), 2), ((1, 4), 2))
    assert nested((3, 1)) == 1 * 1 + 1 * 4 + 1 * 2                   # 3 -> (1, 1) inside mode 0
    assert L(m, 8, 2).cosize() == 15


def test_left_right():
    m = load("g2_cute_layout")
    assert m.make_layout_left((4, 3, 2)) == m.Layout((4, 3, 2), (1, 4, 12))
    assert m.make_layout_right((4, 3, 2)) == m.Layout((4, 3, 2), (6, 2, 1))
```

```
def test_coalesce():
    m = load("g2_cute_layout")
    assert m.coalesce(L(m, (2, (1, 6)), (1, (6, 2)))) == m.Layout(12, 1)
    assert m.coalesce(L(m, (4, 2), (2, 1))) == m.Layout((4, 2), (2, 1))     # row
-major: nothing merges
    a = L(m, ((2, 4), 3), ((1, 2), 16))
    c = m.coalesce(a)
    assert c == m.Layout((8, 3), (1, 16)) and same_function(a, c, a.size())


def test_composition():
    m = load("g2_cute_layout")
    a = L(m, (6, 2), (8, 2))
    b = L(m, (4, 3), (3, 1))
    r = m.composition(a, b)
    assert same_function(r, lambda i: a(b(i)), b.size())
    assert m.coalesce(r) == m.coalesce(m.Layout(((2, 2), 3), ((24, 2), 8)))
    # picking every other element of a 1-D layout
    assert m.composition(L(m, 20, 2), L(m, (5, 4), (4, 1))) == m.Layout((5, 4),
(8, 2))


def test_complement():
    m = load("g2_cute_layout")
    assert m.complement(L(m, 4, 1), 24) == m.Layout(6, 4)
    assert m.complement(L(m, 6, 4), 24) == m.Layout(4, 1)
    c = m.complement(L(m, (2, 2), (1, 6)), 24)
    assert c == m.Layout((3, 2), (2, 12))
    a = L(m, (2, 2), (1, 6))
    offsets = sorted(a(i) + c(j) for i in range(a.size()) for j in range(c.size
()))
    assert offsets == list(range(24))                    # A and its complement
tile [0, 24) exactly


def test_logical_divide():
    m = load("g2_cute_layout")
    a = L(m, 16, 1)
    d = m.logical_divide(a, L(m, 4, 1))                  # 16 elements -> 4 tile
s of 4 contiguous
    assert [d((i, j)) for j in range(4) for i in range(4)] == list(range(16))
    a = L(m, 24, 1)
    d = m.logical_divide(a, L(m, 4, 2))                  # tile = 4 elements wit
h stride 2
    assert [d((i, 0)) for i in range(4)] == [0, 2, 4, 6]
    assert sorted(d((i, j)) for i in range(4) for j in range(6)) == list(range(2
4))
```

提示

L1 先写调用，再写 coalesce 方向（轻度剧透） 收起

`_idx2crd` 对每个模式依次做 `idx % size(mode)`、`idx //= size(mode)`；嵌套模式就递归。`coalesce` 只需要遍历 flatten 之后的 (shape, stride) 对：维护结果列表，能合并就合并到最后一个模式上。

L2 composition 的单模式算法 拆解与常见坑（中度剧透） 收起

对 B 的一个模式 `s:d`（rest_shape = s，rest_stride = d），遍历 coalesce 后 A 的每个模式 `(a_s, a_d)`（最后一个模式除外）：

```
new_shape  = min(max(1, a_s // rest_stride), rest_shape)
若 new_shape != 1：结果加入模式 new_shape : rest_stride · a_d
rest_shape  //= new_shape
rest_stride = ceil(rest_stride / a_s)
```

剩下的 rest_shape 放在 A 的最后一个模式上：stride 为 `rest_stride · A 最后一个模式的 stride`。

L3 complement 几乎是答案（重度剧透） 收起

```
modes = sorted((stride, shape) for 每个 shape != 1 的模式)
current = 1
for (d, s) in modes:  结果加入 (d // current) : current;  current = s · d
结果加入 ceil(cotarget / current) : current，最后 coalesce
```

例：`complement((2,2):(1,6), 24)` → `(1,3,2):(1,2,12)` → coalesce 得到 `(3,2):(2,12)`。

参考解

解 **solutions/g2_cute_layout.py** 先自己写完、跑过测试再看 收起

solutions/g2_cute_layout.py 159 行

```
"""G2 · CuTe layout algebra in pure Python — reference solution.

A CuTe Layout is a pair (Shape, Stride) of (nested) integer tuples. It is a F
UNCTION from
coordinates to offsets:   offset = sum_i coord_i * stride_i.
A plain integer index is first turned into a coordinate colexicographically
(first mode fastest).
Everything in CUTLASS 3 / CuTe DSL — thread-value layouts, tiled MMA/copy, sm
em swizzles,
FlashAttention-4 and FlashInfer's CuTe-DSL kernels — is built from these few
operations.

    Layout((4, 2), (1, 4))   "4x2 column-major"   written (4,2):(1,4)
    Layout((4, 2), (2, 1))   "4x2 row-major"      written (4,2):(2,1)
"""
from dataclasses import dataclass
from math import prod


def flatten(t):
    if isinstance(t, tuple):
        return tuple(x for e in t for x in flatten(e))
    return (t,)


def _product(shape):
    return prod(flatten(shape))


def _idx2crd(idx, shape):
    """Colexicographic: the first mode varies fastest. Follows the nesting of
`shape`."""
    if isinstance(shape, tuple):
        crd = []
        for s in shape:
            n = _product(s)
            crd.append(_idx2crd(idx % n, s))
            idx //= n
        return tuple(crd)
    return idx
```

```
def _inner(crd, shape, stride):
    if isinstance(shape, tuple):
        if not isinstance(crd, tuple):            # an integer for a nested m
ode: unflatten it first
            crd = _idx2crd(crd, shape)
        return sum(_inner(c, s, d) for c, s, d in zip(crd, shape, stride))
    return crd * stride


@dataclass(frozen=True)
class Layout:
    shape: object
    stride: object

    def __call__(self, crd):
        return _inner(crd, self.shape, self.stride)

    def size(self):                       # number of coordinates (the domain)
        return _product(self.shape)

    def cosize(self):                     # 1 + largest offset (the codomain th
e layout touches)
        return self(self.size() - 1) + 1 if self.size() else 0

    def __repr__(self):
        return f"{self.shape}:{self.stride}"


def make_layout_right(shape):
    """Compact row-major (LayoutRight): the LAST mode has stride 1."""
    fl = flatten(shape)
    strides, acc = [0] * len(fl), 1
    for i in reversed(range(len(fl))):
        strides[i] = acc
        acc *= fl[i]
    return Layout(shape, _unflatten(shape, iter(strides)))


def _unflatten(shape, it):
    if isinstance(shape, tuple):
        return tuple(_unflatten(s, it) for s in shape)
    return next(it)


def make_layout_left(shape):
    """Compact column-major (LayoutLeft): stride = exclusive prefix product o
f the flattened shape."""
    strides, acc = [], 1
    for s in flatten(shape):
        strides.append(acc)
```

```
        acc *= s
    return Layout(shape, _unflatten(shape, iter(strides)))


def coalesce(layout: Layout) -> Layout:
    """Same function, fewest modes: drop size-1 modes, merge (s0:d0, s1:d1) w
hen d1 == s0 * d0."""
    shapes, strides = [], []
    for s, d in zip(flatten(layout.shape), flatten(layout.stride)):
        if s == 1:
            continue
        if shapes and strides[-1] * shapes[-1] == d:
            shapes[-1] *= s                      # the new mode continues th
e previous one
        else:
            shapes.append(s)
            strides.append(d)
    if not shapes:
        return Layout(1, 0)
    if len(shapes) == 1:
        return Layout(shapes[0], strides[0])
    return Layout(tuple(shapes), tuple(strides))


def composition(a: Layout, b: Layout) -> Layout:
    """R = A o B, i.e. R(i) = A(B(i)) for i < size(B). B is applied mode by m
ode.

    For one mode s:d of B, walk A's (coalesced) modes: skip the first d eleme
nts ("divide"),
    then keep the next s ("mod"). Requires the usual CuTe divisibility condit
ions."""
    if isinstance(b.shape, tuple):
        modes = [composition(a, Layout(s, d)) for s, d in zip(b.shape, b.stri
de)]
        return Layout(tuple(m.shape for m in modes), tuple(m.stride for m in
modes))
    if b.stride == 0:
        return Layout(b.shape, 0)
    fa = coalesce(a)
    a_shapes, a_strides = flatten(fa.shape), flatten(fa.stride)
    rest_shape, rest_stride = b.shape, b.stride
    shapes, strides = [], []
    for s, d in zip(a_shapes[:-1], a_strides[:-1]):
        assert s % rest_stride == 0 or rest_stride % s == 0, "stride divisibi
lity"
        new_shape = min(max(1, s // rest_stride), rest_shape)
        if new_shape != 1:
            shapes.append(new_shape)
            strides.append(rest_stride * d)
```

```
            assert rest_shape % new_shape == 0, "shape divisibility"
            rest_shape //= new_shape
            rest_stride = -(-rest_stride // s)        # ceil division
        if rest_shape != 1 or not shapes:             # whatever is left runs alo
ng A's last mode
            shapes.append(rest_shape)
            strides.append(rest_stride * a_strides[-1])
        if len(shapes) == 1:
            return Layout(shapes[0], strides[0])
        return Layout(tuple(shapes), tuple(strides))


def complement(a: Layout, cotarget: int) -> Layout:
    """The layout of the "holes" of A in [0, cotarget): (A, complement(A)) to
gether cover it.

    Sort A's modes by stride; each gap between where one mode ends and the ne
xt begins becomes
    a mode of the complement, plus a final mode up to cotarget."""
    modes = sorted((d, s) for s, d in zip(flatten(a.shape), flatten(a.strid
e)) if s != 1 and d != 0)
    shapes, strides, current = [], [], 1
    for d, s in modes:
        assert d % current == 0, "A must be injective with divisible strides"
        shapes.append(d // current)
        strides.append(current)
        current = s * d
    shapes.append(-(-cotarget // current))
    strides.append(current)
    return coalesce(Layout(tuple(shapes), tuple(strides)))


def logical_divide(a: Layout, tiler: Layout) -> Layout:
    """Split A into (tile, rest): mode 0 = the elements the tiler picks, mode
1 = repetitions.
    logical_divide(A, B) = A o (B, complement(B, size(A)))."""
    c = complement(tiler, a.size())
    return composition(a, Layout((tiler.shape, c.shape), (tiler.stride, c.str
ide)))
```

测完之后

- 用你的实现算一算：一个 128×64 的行主序 tile，按 `(32, 8)` 分块之后，第 3 块的第 5 个元素在原 tile 中的偏移是多少？这正是 `local_tile` 做的事。

- 读 CuTe DSL 的官方示例（CUTLASS 仓库 `examples/python/CuTeDSL/`），从 elementwise add 开始：它用 `zipped_divide` 把张量切成每个线程负责的小块，你现在可以手算出每个线程拿到哪些元素。
- **swizzle**：shared memory 的 bank conflict 通常不是靠 padding（C3 的做法）解决，而是对地址做异或：`Swizzle<B, M, S>` 把地址的第 [M+S, M+S+B) 位异或到第 [M, M+B) 位上。它也是一个“布局”，可以和别的 layout 组合。
- 进阶：在 RTX 5090 上安装 `nvidia-cutlass-dsl`（FlashInfer 0.7 已经依赖它），跑通一个 CuTe DSL 的 elementwise kernel，再读 FlashInfer 中 `flashinfer/cute_dsl/` 下的一个 kernel。

▼ 面试追问：CuTe 的 layout 抽象解决了什么问题?

高性能 kernel 里有很多种“坐标 → 地址”的映射：全局内存中的张量、shared memory 中带 swizzle 的 tile、MMA 指令要求的寄存器排列、每个线程负责的数据片段。传统写法把这些映射都写成手工的下标公式，每换一种指令或 tile 大小就要重写，也很容易出错。CuTe 把它们统一成 (Shape, Stride) 函数，并提供 composition、divide、product 等运算来推导映射，例如“这块 tile 中线程 t 的第 v 个值在哪里”就是几个 layout 运算的组合。这样，同一份 kernel 代码可以换 tile 大小和 MMA 指令，编译器在编译期算出全部下标。

Track M · MoE 与低精度

路由与对齐、grouped GEMM、FP8 block scaling：DeepSeek-V3 / DeepGEMM 的配方。

进阶 · Track M · MoE 与低精度 · 第 20 题

M1 MoE 路由 + moe_align_block_size

MoE 层的第一步不是矩阵乘，而是“排队”：给每个 token 选专家，把 (token, 专家) 对按专家分组并补齐到 tile 大小，这样后面一次 launch 就能算完所有专家。

40 min | 难度 中 | 你写 `mine/m1_moe_align.py` | 验收 `./check.sh m1` | 参考解 `solutions/m1_moe_align.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

MoE（Mixtral、Qwen3-MoE、DeepSeek-V3、gpt-oss、Kimi K2）把 MLP 换成 E 个专家，每个 token 只送到 top-k 个专家（E = 64—256，k = 2—8）。计算上就变成：对每个专家 e，用分到它的那些 token 做一次小 GEMM。

直接写成 `for e in experts: y[idx_e] = x[idx_e] @ W_e` 会启动 E 个 kernel，而且每个都很小。高效的做法是先把 (token, slot) 对**按专家排序**，并把每个专家的那一组**补齐到 BLOCK_M 的整数倍**。这样每个 BLOCK_M 行的块只属于一个专家，一个 grouped GEMM kernel（M2）就能一次算完。vLLM 的 `moe_align_block_size` 做的就是这件事。

源码与资料

`vllm › vLLM moe_align_block_size` · `vllm › vLLM moe_align_sum_kernels.cu` · `vllm › vLLM topk_softmax_kernels.cu（路由）` · `vllm › test_moe_align_block_size.py` · `vllm › benchmark_moe_align_block_size.py` · DeepSeek-V3 技术报告

要用到的知识

- **路由**（Mixtral / Qwen-MoE 风格）：对专家做 softmax，取 top-k，再把这 k 个权重重新归一化。DeepSeek-V3 用 sigmoid 打分加分组限制的路由，原理相同。

- **扁平编号**：`f = t·K + k`（第 t 个 token 的第 k 个槽位）。补齐位置填 `T·K`，这是一个越界的编号，GEMM 会把它屏蔽掉。
- PyTorch：`torch.bincount`（每个专家的 token 数）、`torch.argsort(stable=True)`（稳定排序，保持组内 token 的原始顺序）、`torch.repeat_interleave`。
- 不准写对 token 或专家的 Python 循环：真实系统里这一步在 GPU 上完成，T·K 可以达到几十万。

图解与类比

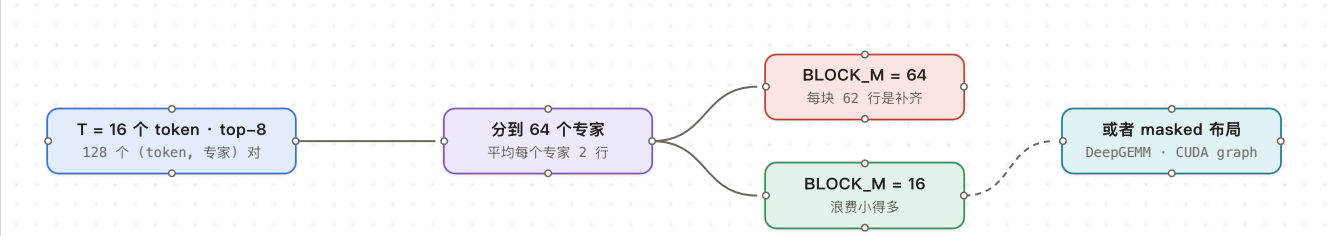
从 router logits 到“每个块属于哪个专家”



类比 · 医院分诊

每位病人（token）挂号时选 k 个科室（top-k 专家）。按科室把挂号单排好队（稳定排序），每个诊室一次叫 BLOCK_M 个号；人数凑不满一批时，用空号（padding = T·K）补齐。这样每个诊室（M2 里的一个 program）只看自己科室的病人，一次叫号就能做完一整批。

decode 时补齐的浪费



类比 · 一辆大巴只坐两个人

每个诊室一次叫 64 个号（BLOCK_M = 64），可 decode 时每个科室平均只有 2 位病人，诊室大部分时间都在给空号看病。要么把每批改小（BLOCK_M = 16），要么换一种调度：每个诊室预留固定的号段，按实际人数开诊（masked 布局）。

题目要求

在 `mine/m1_moe_align.py` 中实现：

1. `route(router_logits [T, E], top_k) -> (topk_weights [T, K] fp32, topk_ids [T, K] int32)`。
2. `moe_align_block_size(topk_ids [T, K], num_experts, block_m) -> (sorted_token_ids [P] int32, expert_ids [P / block_m] int32, P)`：
   - 按专家编号从小到大分组，组内按扁平编号升序；
   - 每组补齐到 block_m 的整数倍，补齐值为 `T·K`；
   - `expert_ids[b]` 是第 b 个块所属的专家；没有 token 的专家不占任何块。

题干

mine/m1_moe_align.py 17 行

```
"""M1 · MoE routing + moe_align_block_size — YOUR IMPLEMENTATION.

See the drill page for the spec; the reference lives in solutions/."""
import torch


def route(router_logits: torch.Tensor, top_k: int):
    # TODO: softmax (fp32) -> topk -> renormalise the kept weights; ids as int32
    raise NotImplementedError


def moe_align_block_size(topk_ids: torch.Tensor, num_experts: int, block_m: int):
    # TODO: counts = bincount; padded = round each count up to block_m; pad_start
= exclusive cumsum
    # TODO: order = stable argsort of the flat expert ids; rank within group; des
t = pad_start[e] + rank
    # TODO: sorted_token_ids = full(total, T*K); sorted_token_ids[dest] = order
    # TODO: expert_ids = repeat_interleave(arange(E), padded // block_m)
    raise NotImplementedError
```

验收

```
./check.sh m1              # CPU 即可
```

tests/test_m1_moe_align.py 41 行

```python
import torch

from drill import load


def test_route():
    m = load("m1_moe_align")
    logits = torch.tensor([[0.0, 3.0, 1.0, 2.0], [5.0, 0.0, 0.0, 4.0]])
    w, ids = m.route(logits, 2)
    assert ids.dtype == torch.int32 and ids.tolist() == [[1, 3], [0, 3]]
    torch.testing.assert_close(w.sum(-1), torch.ones(2))
    p = torch.softmax(logits[0], 0)
    torch.testing.assert_close(w[0], torch.stack([p[1], p[3]]) / (p[1] + p[3]))


def test_align_by_hand():
    m = load("m1_moe_align")
    # T=3 tokens, K=2 -> flat ids 0..5; experts: f0->2, f1->0, f2->2, f3->1, f4->
0, f5->2
    topk_ids = torch.tensor([[2, 0], [2, 1], [0, 2]], dtype=torch.int32)
    sorted_ids, expert_ids, total = m.moe_align_block_size(topk_ids, num_experts=
4, block_m=2)
    # expert 0: [1, 4] | expert 1: [3, pad] | expert 2: [0, 2, 5, pad] | expert
3: none
    assert total == 8
    assert sorted_ids.tolist() == [1, 4, 3, 6, 0, 2, 5, 6]          # pad value =
T*K = 6
    assert expert_ids.tolist() == [0, 1, 2, 2]


def test_align_invariants_random():
    m = load("m1_moe_align")
    g = torch.Generator().manual_seed(0)
    T, K, E, BM = 257, 4, 16, 16
    topk_ids = torch.stack([torch.randperm(E, generator=g)[:K] for _ in range
(T)]).int()
    sorted_ids, expert_ids, total = m.moe_align_block_size(topk_ids, E, BM)
    assert total % BM == 0 and sorted_ids.numel() == total and expert_ids.numel()
== total // BM
    flat = topk_ids.reshape(-1)
    real = sorted_ids[sorted_ids < T * K].long()
    assert sorted(real.tolist()) == list(range(T * K))              # every pair
exactly once
    blocks = sorted_ids.view(-1, BM)
    for b in range(blocks.shape[0]):                                # each bloc
k: one expert, in order
        ids = blocks[b][blocks[b] < T * K].long()
        assert (flat[ids] == expert_ids[b]).all()
        assert (ids[1:] > ids[:-1]).all()
```

提示

L1 先算每个专家的组从哪里开始 方向（轻度剧透） 收起

`counts = bincount(flat)` → `padded = ceil(counts / block_m) · block_m` → `pad_start = cumsum(padded) − padded`（不含自身的前缀和）。P 就是 `padded.sum()`。

L2 每个元素要去哪 拆解与常见坑（中度剧透） 收起

`order = argsort(flat, stable=True)` 给出按专家分组后的扁平编号。它在自己组内的名次 `rank = arange(n) − group_start[expert]`（`group_start` 是未补齐的 counts 的前缀和），目标位置 `dest = pad_start[expert] + rank`。

L3 收尾 几乎是答案（重度剧透） 收起

```
sorted_ids = torch.full((P,), T * K, dtype=torch.int32)
sorted_ids[dest] = order.int()
expert_ids = torch.repeat_interleave(torch.arange(E), padded // block_m).int()
```

参考解

解 **solutions/m1_moe_align.py** 先自己写完、跑过测试再看 收起

solutions/m1_moe_align.py 44 行

```
"""M1 · MoE routing + moe_align_block_size — reference solution.

route(logits [T, E], top_k)  -> (topk_weights [T, K] fp32, topk_ids [T, K] int32)
    softmax over experts, keep the top_k, renormalise the kept weights to sum to 1
    (Mixtral / Qwen-MoE style; DeepSeek-V3 uses sigmoid scores + group-limited routing).

moe_align_block_size(topk_ids [T, K], num_experts, block_m)
    -> sorted_token_ids [P] int32, expert_ids [P // block_m] int32, num_tokens_post_padded int
The (token, slot) pairs are numbered flat: f = t * K + k. Grouping them by expert and padding each
expert's group to a multiple of block_m lets ONE grouped GEMM launch (M2) process every expert:
each block of block_m rows belongs to exactly one expert. Padding rows hold the value T * K
(an out-of-range id the GEMM masks off). Same contract as vLLM's moe_align_block_size, except
that P here is the exact padded length instead of a preallocated upper bound.
"""
import torch


def route(router_logits: torch.Tensor, top_k: int):
    probs = torch.softmax(router_logits.float(), dim=-1)
    w, ids = torch.topk(probs, top_k, dim=-1)
    w = w / w.sum(-1, keepdim=True)
    return w, ids.to(torch.int32)


def moe_align_block_size(topk_ids: torch.Tensor, num_experts: int, block_m: int):
    flat = topk_ids.reshape(-1).long()                      # [T*K] expert of each (token, slot)
    n = flat.numel()
    dev = flat.device
    counts = torch.bincount(flat, minlength=num_experts)    # tokens per expert
    padded = (counts + block_m - 1) // block_m * block_m    # each group rounded up to block_m
```

```
    pad_start = torch.cumsum(padded, 0) - padded             # where each exp
ert's group begins
    total = int(padded.sum().item())

    # Stable sort by expert keeps the original token order inside each exper
t's group.
    order = torch.argsort(flat, stable=True)                 # flat ids, grou
ped by expert
    sorted_experts = flat[order]
    group_start = torch.cumsum(counts, 0) - counts           # start of each
group in `order`
    rank = torch.arange(n, device=dev) - group_start[sorted_experts]   # posi
tion within the group
    dest = pad_start[sorted_experts] + rank                  # position in th
e padded layout

    sorted_token_ids = torch.full((total,), n, dtype=torch.int32, device=dev)
# padding = T*K
    sorted_token_ids[dest] = order.to(torch.int32)
    expert_ids = torch.repeat_interleave(torch.arange(num_experts, device=de
v), padded // block_m).to(torch.int32)
    return sorted_token_ids, expert_ids, total
```

测完之后

- 补齐带来的浪费：T = 16（decode）、E = 64、k = 8 时，平均每个专家只有 2 个 token，block_m = 64 会让大部分行都是补齐行。这就是为什么 decode 阶段的 MoE kernel 要用更小的 block_m，或者换一种布局（DeepGEMM 的 masked grouped GEMM：为每个专家预留固定的最大行数，配合 CUDA graph 使用）。
- vLLM 的 `moe_align_block_size` 是一个 CUDA kernel（单个 block 内用 shared memory 做计数和前缀和）。用 Triton 或 CUDA 把你的 PyTorch 版本改写成一个 kernel，看看能比这几次 PyTorch 调用快多少。
- **负载不均衡**：热门专家的 token 数可能是平均值的好几倍，grouped GEMM 的时间取决于最慢的那个专家。专家并行（EP）时还有跨卡不均衡的问题，DeepSeek 的 EPLB 通过复制热门专家来缓解。

▼ 面试追问：为什么要把 token 按专家排序？直接对每个 token 做 gather 不行吗？

GEMM 的效率来自数据复用：同一个专家的权重被多个 token 共用，才能组成矩阵乘、用上 Tensor Core。按专家分组之后，一个块里的所有行都乘以同一块权重，权重只需要读一次。按 token 逐个 gather 权重，就退化成矩阵乘向量，每个 token 都要读一遍 k 个专家的权重，变成严重的 memory-bound。补齐到 block_m 是为了让每个块只对应一个专家，代价是少量补齐行的浪费。

进阶 · Track M · MoE 与低精度 · 第 21 题

M2 Fused MoE grouped GEMM（vLLM fused_moe 结构）

在 T4 的 GEMM 上加两处改动，就得到 vLLM 默认的 MoE kernel：A 的行通过 sorted_token_ids 间接读取，B 的权重块由这一块所属的专家决定。再用它拼出完整的 SwiGLU MoE 层。

75 min | 难度 难 | 你写 `mine/m2_fused_moe.py` | 验收 `./check.sh m2` | 参考解 `solutions/m2_fused_moe.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

vLLM 的 `fused_moe` 是一个 Triton kernel，它在大多数 GPU 上都是 MoE 的通用默认实现（Hopper / Blackwell 上还有 DeepGEMM、CUTLASS、TRT-LLM 等更专门的后端）。它的结构和 T4 几乎一样，只多了两处间接寻址：

1. 这个 program 负责的 BLOCK_M 行不是连续的行，而是 `sorted_token_ids[块]` 指向的 (token, slot) 对。第一个 GEMM 里，多个 slot 共用同一个 token 的输入行（`token = f // K`）。
2. B 是 `W[expert_ids[块]]`：同一个块里所有行属于同一个专家（M1 保证了这一点）。

源码与资料

`vllm` › vLLM fused_moe_kernel · `vllm` › vLLM MoE 调优配置说明 · `sglang` › SGLang fused_moe_triton_kernels · `flashinfer` › FlashInfer fused_moe/ · `triton` › Triton grouped GEMM tutorial · `deepgemm` › DeepGEMM (grouped contiguous / masked) · `aiter` › AITER (AMD fused MoE)

要用到的知识

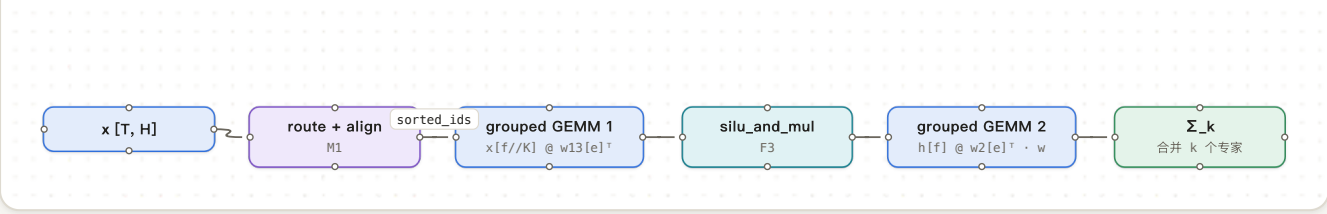
- 完整的 MoE MLP（SwiGLU）：

```
w13 [E, 2I, H]：每个专家的 gate 和 up 投影拼在一起；w2 [E, H, I]：down 投影
h  = x[t] @ w13[e]^T          -> [2I]       第一个 grouped GEMM（A 行 = token）
h  = silu(h[:I]) · h[I:]      -> [I]        F3 的 silu_and_mul
y  = h @ w2[e]^T · weight     -> [H]        第二个 grouped GEMM（A 行 = (token, slot)）
out[t] = Σ_k y[t, k]                         把 k 个专家的结果加起来
```

- 地址：`a_ptrs = a + a_rows[:, None] · stride_am + offs_k[None, :]`，其中 a_rows 是 gather 来的行号；W[e] 是 `[N, Kdim]` 的行主序矩阵，加载它的转置块 `[BLOCK_K, BLOCK_N]`：`w + e·stride_we + offs_n[None, :]·stride_wn + offs_k[:, None]`。
- 行 mask：`flat_ids < num_valid`（补齐行不读不写）。
- 在 epilogue 里把路由权重乘进去（`MUL_WEIGHT`），省掉一次单独的逐元素 kernel。

图解与类比

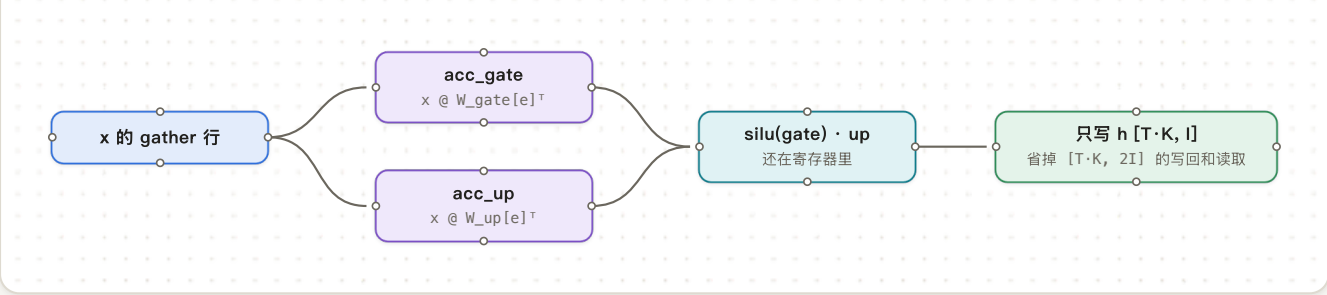
一个 MoE 层的完整流水



类比 · 快递分拣中心

先按目的地（专家）把包裹分拣成整车（BLOCK_M），每辆车（program）只跑一个目的地，路线（权重）只需要装载一次。每件包裹的面单（sorted_token_ids）告诉司机去哪个货架取货（gather 读 A 的行），送到后按原来的订单号放回（写 out[flat_id]）。

把 silu_and_mul 融合进 GEMM1 的 epilogue



类比 · 顺路办两件事

不融合时，GEMM1 把 gate 和 up 写回仓库（HBM），激活函数再把它们取出来、算完再写回去。融合就是在同一个 program 里同时算出 gate 和 up，趁热（还在寄存器里）把激活做了，只把结果送出门：少跑一趟仓库，还少占一半的仓库空间。

题目要求

`fused_moe_gemm` 的 wrapper 和 `moe_layer` 已写好。实现 `fused_moe_kernel`：

- grid = (补齐后的块数, cdiv(N, BLOCK_N))。
- `A_ROWS_ARE_TOKENS`（constexpr）：True 时 A 的行号是 `flat_id // top_k`，False 时就是 `flat_id`。
- 输出写到 `out[flat_id]`（形状 `[T·K, N]`）。
- `moe_layer` 的结果与逐 token、逐专家计算的 reference 一致（fp16，误差 2e-2）。

M2 会通过 `drill.load` 使用**你自己写的 M1**（用 `--solutions` 时使用参考解）。

题干

mine/m2_fused_moe.py 54 行

```
"""M2 · fused MoE grouped GEMM — YOUR IMPLEMENTATION.

See the drill page for the spec; the reference lives in solutions/."""
import torch
import torch.nn.functional as F
import triton
import triton.language as tl

from drill import load

_m1 = load("m1_moe_align")   # your M1 (or the reference with DRILL_IMPL=solution
s)
route, moe_align_block_size = _m1.route, _m1.moe_align_block_size


@triton.jit
def fused_moe_kernel(a_ptr, w_ptr, out_ptr, topk_w_ptr, sorted_ids_ptr, expert_id
s_ptr,
                     N, Kdim, num_valid, top_k,
                     stride_am, stride_we, stride_wn, stride_om,
                     A_ROWS_ARE_TOKENS: tl.constexpr, MUL_WEIGHT: tl.constexpr,
```

```
                     BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.co
nstexpr):
    # TODO: flat_ids = sorted_ids[block]; row_mask = flat_ids < num_valid; expert
= expert_ids[pid_m]
    # TODO: A rows = flat_ids // top_k (GEMM1) or flat_ids (GEMM2); B tile = W[ex
pert]^T [BLOCK_K, BLOCK_N]
    # TODO: K loop with masked loads + tl.dot; optionally * topk_weight[flat_id
s]; store to out[flat_ids]
    pass


def fused_moe_gemm(a, w, sorted_ids, expert_ids, num_valid, top_k, topk_weights=N
one,
                   a_rows_are_tokens=True, block_m=64, block_n=64, block_k=32):
    """a [T, Kdim] (tokens) or [T*K, Kdim]; w [E, N, Kdim] -> out [T*K, N] in a.d
type."""
    E, N, Kdim = w.shape
    assert a.stride(-1) == 1 and w.is_contiguous()
    out = torch.zeros((num_valid, N), dtype=a.dtype, device=a.device)
    tw = topk_weights.reshape(-1).float().contiguous() if topk_weights is not Non
e else out
    grid = (sorted_ids.numel() // block_m, triton.cdiv(N, block_n))
    fused_moe_kernel[grid](a, w, out, tw, sorted_ids, expert_ids, N, Kdim, num_va
lid, top_k,
                           a.stride(0), w.stride(0), w.stride(1), out.stride(0),
                           A_ROWS_ARE_TOKENS=a_rows_are_tokens, MUL_WEIGHT=topk_w
eights is not None,
                           BLOCK_M=block_m, BLOCK_N=block_n, BLOCK_K=block_k)
    return out


def moe_layer(x, router_logits, w13, w2, top_k, block_m=64):
    """x [T, H]; w13 [E, 2I, H] (gate rows then up rows); w2 [E, H, I] -> [T,
H]."""
    T, H = x.shape
    E = w13.shape[0]
    topk_w, topk_ids = route(router_logits, top_k)
    sorted_ids, expert_ids, _ = moe_align_block_size(topk_ids, E, block_m)
    n = T * top_k
    h = fused_moe_gemm(x, w13, sorted_ids, expert_ids, n, top_k, block_m=block_m)
# [T*K, 2I]
    I = h.shape[1] // 2
    h = (F.silu(h[:, :I].float()) * h[:, I:].float()).to(x.dtype)
# F3's silu_and_mul
    y = fused_moe_gemm(h, w2, sorted_ids, expert_ids, n, top_k, topk_weights=topk
_w,
                       a_rows_are_tokens=False, block_m=block_m)
# [T*K, H]
    return y.view(T, top_k, H).float().sum(1).to(x.dtype)
# combine slots
```

验收

```
./check.sh m2
./check.sh m2 --bench       # 完整 MoE 层 vs 同等 FLOPs 的稠密 cuBLAS
```

tests/test_m2_fused_moe.py 47 行

```
import pytest
import torch
import torch.nn.functional as F

from common.env import DEVICE, HAS_CUDA
from drill import load


def ref_moe(x, logits, w13, w2, top_k):
    p = torch.softmax(logits.float(), -1)
    w, ids = torch.topk(p, top_k, -1)
    w = w / w.sum(-1, keepdim=True)
    out = torch.zeros(x.shape, dtype=torch.float32, device=x.device)
    I = w13.shape[1] // 2
    for t in range(x.shape[0]):
        for k in range(top_k):
            e = int(ids[t, k])
            h = x[t].float() @ w13[e].float().t()
            h = F.silu(h[:I]) * h[I:]
            out[t] += w[t, k] * (h @ w2[e].float().t())
    return out


@pytest.mark.parametrize("T,E,K,H,I", [(1, 4, 2, 64, 32), (37, 8, 2, 128, 96), (6
4, 16, 4, 64, 64)])
def test_moe_layer(T, E, K, H, I):
    m = load("m2_fused_moe")
    torch.manual_seed(0)
    x = (torch.randn(T, H, device=DEVICE) * 0.5).half()
    logits = torch.randn(T, E, device=DEVICE)
    w13 = (torch.randn(E, 2 * I, H, device=DEVICE) / H ** 0.5).half()
    w2 = (torch.randn(E, H, I, device=DEVICE) / I ** 0.5).half()
    out = m.moe_layer(x, logits, w13, w2, K, block_m=16 if not HAS_CUDA else 64)
    assert out.shape == (T, H) and out.dtype == torch.float16
    torch.testing.assert_close(out.float(), ref_moe(x, logits, w13, w2, K), rtol=
2e-2, atol=2e-2)


def test_single_gemm_matches_gather():
    m = load("m2_fused_moe")
    from solutions.m1_moe_align import moe_align_block_size
    T, K, E, N, Kd = 20, 2, 5, 48, 64
```

```
    x = torch.randn(T, Kd, device=DEVICE).half()
    w = torch.randn(E, N, Kd, device=DEVICE).half()
    ids = torch.stack([torch.randperm(E)[:K] for _ in range(T)]).int().to(DEVICE)
    sorted_ids, expert_ids, _ = moe_align_block_size(ids, E, 16)
    out = m.fused_moe_gemm(x, w, sorted_ids, expert_ids, T * K, K, block_m=16, bl
ock_n=32, block_k=32)
    ref = torch.einsum("tkd,tknd->tkn", x.float()[:, None].expand(T, K, Kd), w.fl
oat()[ids.long()])
    torch.testing.assert_close(out.view(T, K, N).float(), ref, rtol=1e-2, atol=5e
-2)
```

提示

L1 和 T4 只差在开头 方向（轻度剧透） 收起

T4 里的 `offs_m` 直接就是行号；这里先 `flat_ids = load(sorted_ids + offs_m)`，再由 flat_ids 推出 A 的行号和输出的行号。专家号是一个标量：`expert = load(expert_ids + pid_m)`。

L2 常见坑 拆解与常见坑（中度剧透） 收起

- 补齐行的 flat_id 等于 T·K，`T·K // top_k = T` 也越界：A 的加载一定要带上 row_mask。
- 输出要写到 `out[flat_id]`，而不是 `out[offs_m]`：后者是排序后的位置，结果的顺序会全部打乱。
- 第二个 GEMM 的 A 是中间结果 h，它的行本身就是 `[T·K, I]`，不能再除以 top_k。
- 64 位地址：`expert · stride_we` 和 `a_rows · stride_am` 都可能超过 2^{31}，先转成 int64。

L3 **kernel 主体** 几乎是答案（重度剧透） 收起 ▾

```
offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
flat_ids = tl.load(sorted_ids_ptr + offs_m)
row_mask = flat_ids < num_valid
expert = tl.load(expert_ids_ptr + pid_m).to(tl.int64)
a_rows = flat_ids // top_k if A_ROWS_ARE_TOKENS else flat_ids          # 写成 co
a_ptrs = a_ptr + a_rows[:, None].to(tl.int64) * stride_am + offs_k[None, :]
w_ptrs = w_ptr + expert * stride_we + offs_n[None, :] * stride_wn + offs_k[:, N
# ... 与 T4 相同的 K 循环 ...
if MUL_WEIGHT:
    acc = acc * tl.load(topk_w_ptr + flat_ids, mask=row_mask, other=0.0)[:, Non
tl.store(out_ptr + flat_ids[:, None].to(tl.int64) * stride_om + offs_n[None, :]
```

参考解

解 **solutions/m2_fused_moe.py** 先自己写完、跑过测试再看 收起

solutions/m2_fused_moe.py 91 行

```
"""M2 · Fused MoE grouped GEMM in Triton (the vLLM fused_moe_kernel structur
e) — reference solution.

One kernel launch computes the expert matmul for every (token, slot) pair:
    out[f, :] = x[f // K, :] @ W[expert(f)]^T  (* topk_weight[f] if MUL_WEIGH
T)
using the padded, expert-grouped order from M1. Grid = (padded blocks of BLOC
K_M rows) x (N tiles);
each M-block belongs to exactly one expert, so its B operand is a single expe
rt's weight tile.

moe_layer() is a full SwiGLU MoE MLP on top of it:
    route -> align -> GEMM1 (x @ W13^T, [.., 2I]) -> silu_and_mul -> GEMM2 (h
@ W2^T, * weight) -> sum over K
"""
import torch
import torch.nn.functional as F
import triton
import triton.language as tl

from drill import load

_m1 = load("m1_moe_align")   # your M1 (or the reference with DRILL_IMPL=solu
tions)
route, moe_align_block_size = _m1.route, _m1.moe_align_block_size


@triton.jit
def fused_moe_kernel(a_ptr, w_ptr, out_ptr, topk_w_ptr, sorted_ids_ptr, exper
t_ids_ptr,
                     N, Kdim, num_valid, top_k,
                     stride_am, stride_we, stride_wn, stride_om,
                     A_ROWS_ARE_TOKENS: tl.constexpr, MUL_WEIGHT: tl.constexp
r,
                     BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: t
l.constexpr):
    pid_m = tl.program_id(0)                        # which padded block of BL
OCK_M (token, slot) rows
    pid_n = tl.program_id(1)                        # which N tile

    offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    flat_ids = tl.load(sorted_ids_ptr + offs_m)     # (token, slot) ids; paddi
```

```
ng rows = num_valid
    row_mask = flat_ids < num_valid
    expert = tl.load(expert_ids_ptr + pid_m).to(tl.int64)

    # GEMM1 reads x[token]: several slots of one token share the same input r
ow (gather).
    # GEMM2 reads h[flat_id]: every (token, slot) has its own intermediate ro
w.
    if A_ROWS_ARE_TOKENS:
        a_rows = flat_ids // top_k
    else:
        a_rows = flat_ids
    offs_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
    offs_k = tl.arange(0, BLOCK_K)
    a_ptrs = a_ptr + a_rows[:, None].to(tl.int64) * stride_am + offs_k[None,
:]
    # W[e] is [N, Kdim] row-major; load a [BLOCK_K, BLOCK_N] tile of W[e]^T.
    w_ptrs = w_ptr + expert * stride_we + offs_n[None, :] * stride_wn + offs_
k[:, None]

    acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)
    for k0 in range(0, tl.cdiv(Kdim, BLOCK_K)):
        k_left = Kdim - k0 * BLOCK_K
        a = tl.load(a_ptrs, mask=row_mask[:, None] & (offs_k[None, :] < k_lef
t), other=0.0)
        w = tl.load(w_ptrs, mask=(offs_k[:, None] < k_left) & (offs_n[None,
:] < N), other=0.0)
        acc = tl.dot(a, w, acc)
        a_ptrs += BLOCK_K
        w_ptrs += BLOCK_K

    if MUL_WEIGHT:                                  # fold the router weight i
nto the epilogue
        tw = tl.load(topk_w_ptr + flat_ids, mask=row_mask, other=0.0)
        acc = acc * tw[:, None]
    out_ptrs = out_ptr + flat_ids[:, None].to(tl.int64) * stride_om + offs_n
[None, :]
    tl.store(out_ptrs, acc.to(out_ptr.dtype.element_ty), mask=row_mask[:, Non
e] & (offs_n[None, :] < N))


def fused_moe_gemm(a, w, sorted_ids, expert_ids, num_valid, top_k, topk_weigh
ts=None,
                   a_rows_are_tokens=True, block_m=64, block_n=64, block_k=3
2):
    """a [T, Kdim] (tokens) or [T*K, Kdim]; w [E, N, Kdim] -> out [T*K, N] in
a.dtype."""
    E, N, Kdim = w.shape
    assert a.stride(-1) == 1 and w.is_contiguous()
    out = torch.zeros((num_valid, N), dtype=a.dtype, device=a.device)
```

```
    tw = topk_weights.reshape(-1).float().contiguous() if topk_weights is not
None else out
    grid = (sorted_ids.numel() // block_m, triton.cdiv(N, block_n))
    fused_moe_kernel[grid](a, w, out, tw, sorted_ids, expert_ids, N, Kdim, nu
m_valid, top_k,
                           a.stride(0), w.stride(0), w.stride(1), out.stride
(0),
                           A_ROWS_ARE_TOKENS=a_rows_are_tokens, MUL_WEIGHT=to
pk_weights is not None,
                           BLOCK_M=block_m, BLOCK_N=block_n, BLOCK_K=block_k)
    return out


def moe_layer(x, router_logits, w13, w2, top_k, block_m=64):
    """x [T, H]; w13 [E, 2I, H] (gate rows then up rows); w2 [E, H, I] -> [T,
H]."""
    T, H = x.shape
    E = w13.shape[0]
    topk_w, topk_ids = route(router_logits, top_k)
    sorted_ids, expert_ids, _ = moe_align_block_size(topk_ids, E, block_m)
    n = T * top_k
    h = fused_moe_gemm(x, w13, sorted_ids, expert_ids, n, top_k, block_m=bloc
k_m)          # [T*K, 2I]
    I = h.shape[1] // 2
    h = (F.silu(h[:, :I].float()) * h[:, I:].float()).to(x.dtype)
# F3's silu_and_mul
    y = fused_moe_gemm(h, w2, sorted_ids, expert_ids, n, top_k, topk_weights=
topk_w,
                       a_rows_are_tokens=False, block_m=block_m)
# [T*K, H]
    return y.view(T, top_k, H).float().sum(1).to(x.dtype)
# combine slots
```

测完之后

RTX 5090 实测：T = 4096、E = 64、top-8、H = 2048、I = 768（接近 Qwen3-30B-A3B 的专家形状），bf16：

	时间	TFLOP/s
你的 MoE 层（两次 grouped GEMM，block 64×64×32）	3245 us	95
同等 FLOPs 的两次稠密 cuBLAS GEMM	1461 us	212

- 差距的来源：固定的 64×64×32 tile 没有针对这个形状调优；补齐行带来的浪费；gather 读取 A 不如连续读取高效；silu_and_mul 和最后的求和是单独的 PyTorch 操

作。vLLM 为每种 (E, N, K, GPU) 组合准备了调好的 JSON 配置文件，加上 autotune 往往就能拿回一大部分性能。用 T4 的 autotune 试试。

- 融合：把 silu_and_mul 放进第一个 GEMM 的 epilogue（每个 program 同时计算 gate 和 up 两个 N 块），省掉一次中间结果的写回和读取。
- 和 FlashInfer 对比：`flashinfer.cutlass_fused_moe`、`flashinfer.fused_moe` 等 API 提供了 CUTLASS / TRT-LLM 后端的 MoE，读一读它们的接口，看看都融合了哪些步骤（路由、排列、两次 GEMM、激活、合并）。

▼ 面试追问：MoE 在 decode 和 prefill 时的瓶颈有什么不同?

prefill 时 token 多，每个专家能分到足够多的行，grouped GEMM 是 compute-bound 的，优化重点是 tile 配置、负载均衡和 FP8 / FP4 量化。decode 时 token 很少，每个专家只有几行，GEMM 退化成读一遍被激活专家的权重，是 memory-bound 的；而且一个 batch 激活的专家数可能接近全部专家，读的权重比稠密模型还多。这时的优化手段是更小的 block_m、权重量化（每读一个字节就算得更多）、配合 CUDA graph 的 masked 布局，以及专家并行时的通信与计算重叠。

进阶 · Track M · MoE 与低精度 · 第 22 题

M3 FP8 细粒度量化 + block-scaled GEMM（DeepSeek-V3 / DeepGEMM 配方）

DeepSeek-V3 的 FP8 配方：激活每 1×128 一个缩放系数，权重每 128×128 一个，每个 K 块做一次 FP8 Tensor Core 矩阵乘再乘上缩放。在 RTX 5090 上，这样写出的 Triton kernel 比 bf16 cuBLAS 快 1.7 倍。

60 min | 难度 难 | 你写 `mine/m3_fp8_block_gemm.py` | 验收 `./check.sh m3` | 参考解 `solutions/m3_fp8_block_gemm.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

FP8（e4m3：最大值 448，只有 3 位尾数）让 Tensor Core 的吞吐翻倍、权重的字节数减半。难点在于动态范围：用一个 per-tensor 的缩放系数，激活里的离群值会把大部分数压成 0。DeepSeek-V3 的做法是**细粒度缩放**：

- 激活：每个 token 的每 128 个通道一个 scale（per-token-group，1×128）；
- 权重：每 128×128 的块一个 scale（per-block）。

每个 128 宽的 K 块恰好对应一组 scale，所以 kernel 可以每算完一个 K 块就乘上这一块的 `sa · sb`，全程保持 fp32 累加。DeepGEMM（SM90 / SM100）就是这个配方的高度优化实现；vLLM 的 `w8a8_block_fp8_matmul` 是它的 Triton 版本。Blackwell 的 MXFP8 / NVFP4 把“块缩放”做进了硬件指令（1×32 或 1×16 一个 scale），思路相同。

源码与资料

DeepSeek-V3 技术报告 · `deepgemm › DeepGEMM README` · `deepgemm › DeepGEMM SM90 FP8 1d2d` · `deepgemm › DeepGEMM SM100 FP8 / FP4` · `vllm › vLLM per_token_group_quant_fp8` · `vllm › vLLM block-scaled FP8 matmul` · `flashinfer › FlashInfer deep_gemm.py` · `triton › Triton block-scaled matmul (MXFP / NVFP4)` · OCP Microscaling (MX) Formats Spec

要用到的知识

- 量化：`scale = max(|x_group|) / 448`，`q = clamp(x / scale, -448, 448).to(fp8)`；scale 存为 fp32。全零的组要用一个很小的 eps，避免除以 0。
- `torch.float8_e4m3fn`；Triton 里 `tl.dot(a_fp8, b_fp8)` 直接得到 fp32 结果（需要 sm_89+）。
- block-scaled GEMM 的主循环：

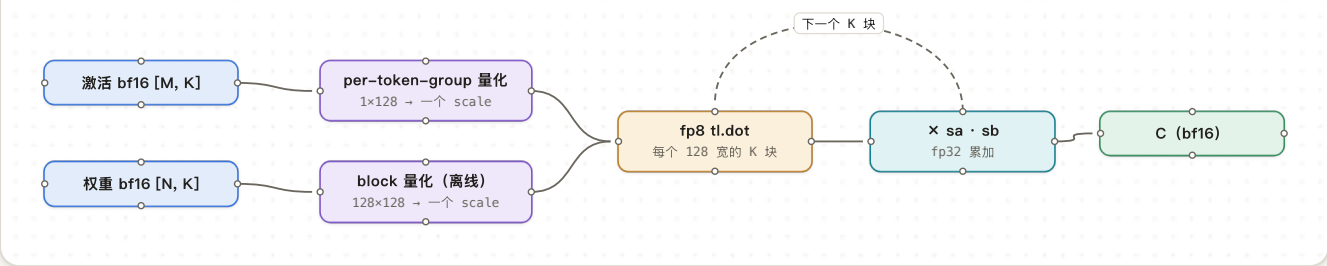
```
for kb in K 块（BLOCK_K = 128）：
    acc += tl.dot(A_q[m, kb], B_q[n, kb]^T) · sa[m, kb] · sb[n // 128, kb]
```

BLOCK_N 取 128 的约数时，一个 tile 里所有列共享同一个 `sb`（一个标量）；`sa` 是 BLOCK_M 个值组成的向量。

- 布局：A `[M, K]`，B `[N, K]`（即 `C = A @ B^T`，和 `nn.Linear` 的权重布局相同）。

图解与类比

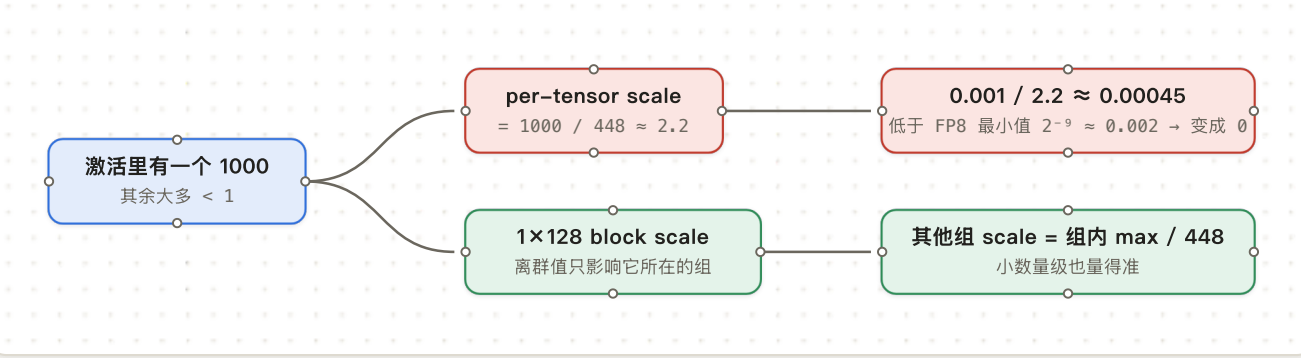
FP8 block-scaled GEMM 的数据流



类比 · 每 128 个数一把尺子

FP8 只有 3 位尾数，像一把刻度很粗的尺子。用一把尺子（per-tensor scale）同时量蚂蚁和大象，蚂蚁全变成 0；给每 128 个数各配一把合适量程的尺子（block scale），大数小数都量得准。 尺子的切换正好落在每个 K 块的边界上，所以 kernel 每算完一块乘一次 scale 就行，几乎没有额外开销。

一个离群值对 per-tensor scale 的影响



类比 · 一个巨人站进合影

集体照里有一个 3 米高的巨人，为了把他拍进去，摄影师只能把镜头拉远（大 scale），结果其他人都只剩几个像素（精度丢失）。分组拍照（block scale）：巨人那一组单独拉远，其他组照常拍特写。

题目要求

`block_quant_fp8`（离线的权重量化，纯 PyTorch）和两个 wrapper 已写好。实现：

1. `per_token_group_quant_kernel`：每个 program 负责 (行, 组)。
2. `fp8_block_gemm_kernel`：如上，BLOCK_K = 128。

验收：量化后每组的最大绝对值约为 448；GEMM 结果与“先反量化再用 fp32 相乘”的结果一致，与 bf16 原始矩阵乘的相对误差 < 5%。需要 sm_89+（RTX 40 / 50 系、H100、B200）。

题干

mine/m3_fp8_block_gemm.py 61 行

```
"""M3 · FP8 per-token-group quant + block-scaled GEMM — YOUR IMPLEMENTATION.

See the drill page for the spec; the reference lives in solutions/."""
import torch
import triton
import triton.language as tl

FP8 = torch.float8_e4m3fn
FP8_MAX = 448.0
GROUP = 128
```

```
@triton.jit
def per_token_group_quant_kernel(x_ptr, q_ptr, s_ptr, K, eps, GROUP_SIZE: tl.cons
texpr):
    # TODO: program (row, group): amax = max(|x|, eps); scale = amax / 448; q = c
lamp(x / scale); store q (fp8) and scale
    pass


def per_token_group_quant_fp8(x: torch.Tensor, group_size: int = GROUP, eps: floa
t = 1e-10):
    """x [M, K] -> (x_q [M, K] fp8 e4m3, scales [M, ceil(K/group)] fp32)."""
    x = x.contiguous()
    M, K = x.shape
    q = torch.empty((M, K), dtype=FP8, device=x.device)
    s = torch.empty((M, triton.cdiv(K, group_size)), dtype=torch.float32, device=
x.device)
    per_token_group_quant_kernel[(M, triton.cdiv(K, group_size))](x, q, s, K, ep
s, GROUP_SIZE=group_size)
    return q, s


def block_quant_fp8(w: torch.Tensor, block: int = GROUP):
    """Weights [N, K] -> (w_q fp8 [N, K], scales [ceil(N/128), ceil(K/128)]). Off
line, so plain PyTorch."""
    N, K = w.shape
    nb, kb = triton.cdiv(N, block), triton.cdiv(K, block)
    wp = torch.zeros(nb * block, kb * block, dtype=torch.float32, device=w.devic
e)
    wp[:N, :K] = w.float()
    blocks = wp.view(nb, block, kb, block)
    amax = blocks.abs().amax(dim=(1, 3)).clamp(min=1e-10)             # [nb, kb]
    scale = amax / FP8_MAX
    q = (blocks / scale[:, None, :, None]).clamp(-FP8_MAX, FP8_MAX).to(FP8)
    return q.view(nb * block, kb * block)[:N, :K].contiguous(), scale


@triton.jit
def fp8_block_gemm_kernel(a_ptr, b_ptr, c_ptr, sa_ptr, sb_ptr, M, N, K,
                          stride_am, stride_bn, stride_cm, stride_sam, stride_sb
n,
                          BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K:
tl.constexpr):
    # TODO: for each 128-wide K block: acc += tl.dot(a_fp8, b_fp8) * sa[:, None]
* sb
    # TODO: sa = a_s[offs_m, kb] (one per row); sb = b_s[(pid_n * BLOCK_N) // 12
8, kb] (one per tile)
    pass
```

```
def fp8_block_gemm(a_q, a_s, b_q, b_s, out_dtype=torch.bfloat16, block_m=64, bloc
k_n=128):
    """a_q [M, K] fp8 + a_s [M, K/128]; b_q [N, K] fp8 + b_s [N/128, K/128] -> C
[M, N] = A @ B^T."""
    M, K = a_q.shape
    N = b_q.shape[0]
    assert 128 % block_n == 0 and a_q.is_contiguous() and b_q.is_contiguous()
    c = torch.empty((M, N), dtype=out_dtype, device=a_q.device)
    grid = (triton.cdiv(M, block_m), triton.cdiv(N, block_n))
    fp8_block_gemm_kernel[grid](a_q, b_q, c, a_s, b_s, M, N, K,
                                a_q.stride(0), b_q.stride(0), c.stride(0), a_s.st
ride(0), b_s.stride(0),
                                BLOCK_M=block_m, BLOCK_N=block_n, BLOCK_K=128, nu
m_warps=4, num_stages=3)
    return c
```

验收

```
./check.sh m3              # 需要 sm_89+ 的 GPU
./check.sh m3 --bench      # vs bf16 cuBLAS
```

tests/test_m3_fp8_block_gemm.py 41 行

```
import pytest
import torch

from common.env import HAS_CUDA
from drill import load

FP8_OK = HAS_CUDA and torch.cuda.get_device_capability() >= (8, 9)
pytestmark = pytest.mark.skipif(not FP8_OK, reason="FP8 tensor cores need sm_89+
(Ada / Hopper / Blackwell)")


def dequant_tokens(q, s, group=128):
    M, K = q.shape
    return (q.float().view(M, -1, group) * s[:, :, None]).view(M, K)


@pytest.mark.parametrize("M,K", [(1, 128), (33, 512), (256, 7168)])
def test_per_token_group_quant(M, K):
    m = load("m3_fp8_block_gemm")
    x = torch.randn(M, K, device="cuda", dtype=torch.bfloat16) * 3
    q, s = m.per_token_group_quant_fp8(x)
    assert q.dtype == torch.float8_e4m3fn and s.shape == (M, K // 128) and s.dtyp
e == torch.float32
    # each group's largest |value| maps to (almost exactly) 448
```

```
    assert torch.allclose(q.float().view(M, -1, 128).abs().amax(-1), torch.full_l
ike(s, 448.0), rtol=1e-2)
    torch.testing.assert_close(dequant_tokens(q, s), x.float(), rtol=7e-2, atol=7
e-2 * s.max().item() * 32)


@pytest.mark.parametrize("M,N,K", [(16, 128, 256), (100, 384, 1024), (512, 1024,
2048)])
def test_fp8_block_gemm(M, N, K):
    m = load("m3_fp8_block_gemm")
    a = torch.randn(M, K, device="cuda", dtype=torch.bfloat16)
    b = torch.randn(N, K, device="cuda", dtype=torch.bfloat16) / K ** 0.5
    aq, as_ = m.per_token_group_quant_fp8(a)
    bq, bs = m.block_quant_fp8(b)
    c = m.fp8_block_gemm(aq, as_, bq, bs)
    # exact reference for the quantised operands: dequantise, then matmul in fp32
    b_deq = (bq.float().view(N // 128, 128, K // 128, 128) * bs[:, None, :, Non
e]).view(N, K)
    ref_q = dequant_tokens(aq, as_) @ b_deq.t()
    torch.testing.assert_close(c.float(), ref_q, rtol=2e-2, atol=2e-2)
    # and the quantisation error against the bf16 matmul stays small
    rel = (c.float() - a.float() @ b.float().t()).norm() / (a.float() @ b.float
().t()).norm()
    assert rel < 0.05
```

提示

L1 量化 kernel 只有几行 方向（轻度剧透） 收起

加载 128 个值 → `amax = max(max(|x|), eps)` → `scale = amax / 448` → 存 `clamp(x / scale)` 和 scale。scale 的位置是 `row · cdiv(K, 128) + g`。

L2 GEMM 的两个 scale 拆解与常见坑（中度剧透） 收起

- `sa = load(sa_ptr + offs_m · stride_sam + kb)`：形状 `[BLOCK_M]`。
- `sb = load(sb_ptr + ((pid_n · BLOCK_N) // 128) · stride_sbn + kb)`：一个标量。
- `acc += tl.dot(a, b) * (sa[:, None] * sb)`：先把 FP8 的点积在 fp32 里算完，再乘 scale。不能把 scale 乘进 FP8 的输入里（那样就失去了量化的意义），也不要只在最后乘一次（每个 K 块的 scale 不同）。

L3 **主循环** 几乎是答案（重度剧透） 收起 ▾

```
for kb in range(0, tl.cdiv(K, BLOCK_K)):
    k_left = K - kb * BLOCK_K
    a = tl.load(a_ptrs, mask=m_mask[:, None] & (offs_k[None, :] < k_left), othe
    b = tl.load(b_ptrs, mask=(offs_k[:, None] < k_left) & n_mask[None, :], othe
    sa = tl.load(sa_ptr + offs_m * stride_sam + kb, mask=m_mask, other=0.0)
    sb = tl.load(sb_ptr + sb_row * stride_sbn + kb)
    acc += tl.dot(a, b) * (sa[:, None] * sb)
    a_ptrs += BLOCK_K
    b_ptrs += BLOCK_K
```

参考解

解 **solutions/m3_fp8_block_gemm.py** 先自己写完、跑过测试再看 收起

solutions/m3_fp8_block_gemm.py 99 行

```
"""M3 · FP8 fine-grained (block-scaled) quantisation + GEMM — reference solut
ion.

The DeepSeek-V3 / DeepGEMM recipe, written in Triton:
  activations: per-token-group quantisation, one fp32 scale per 1 x 128 group
  weights:     per-block quantisation,       one fp32 scale per 128 x 128 blo
ck
  C[m, n] = sum_kb  (A_q[m, kb] @ B_q[n, kb]^T) * sa[m, kb] * sb[n // 128, k
b]
Each 128-wide K step is one fp8 tl.dot (Tensor Cores, fp32 accumulation) foll
owed by a scale
multiply, so the scales change every K block without leaving fp32.
FP8 e4m3: largest finite value 448. Needs sm_89+ (Ada, Hopper, Blackwell inc
l. RTX 50xx).
"""
import torch
import triton
import triton.language as tl

FP8 = torch.float8_e4m3fn
FP8_MAX = 448.0
GROUP = 128


@triton.jit
def per_token_group_quant_kernel(x_ptr, q_ptr, s_ptr, K, eps, GROUP_SIZE: tl.
constexpr):
    # One program per (row, group): the scale makes the group's max |x| map t
o 448.
    row = tl.program_id(0).to(tl.int64)
    g = tl.program_id(1)
    offs = g * GROUP_SIZE + tl.arange(0, GROUP_SIZE)
    mask = offs < K
    x = tl.load(x_ptr + row * K + offs, mask=mask, other=0.0).to(tl.float32)
    amax = tl.maximum(tl.max(tl.abs(x), axis=0), eps)         # eps: an all-z
ero group gets a tiny scale, not 0
    scale = amax / 448.0
    q = tl.clamp(x / scale, -448.0, 448.0)
    tl.store(q_ptr + row * K + offs, q.to(q_ptr.dtype.element_ty), mask=mask)
    tl.store(s_ptr + row * tl.cdiv(K, GROUP_SIZE) + g, scale)
```

```
def per_token_group_quant_fp8(x: torch.Tensor, group_size: int = GROUP, eps:
float = 1e-10):
    """x [M, K] -> (x_q [M, K] fp8 e4m3, scales [M, ceil(K/group)] fp32)."""
    x = x.contiguous()
    M, K = x.shape
    q = torch.empty((M, K), dtype=FP8, device=x.device)
    s = torch.empty((M, triton.cdiv(K, group_size)), dtype=torch.float32, dev
ice=x.device)
    per_token_group_quant_kernel[(M, triton.cdiv(K, group_size))](x, q, s, K,
eps, GROUP_SIZE=group_size)
    return q, s


def block_quant_fp8(w: torch.Tensor, block: int = GROUP):
    """Weights [N, K] -> (w_q fp8 [N, K], scales [ceil(N/128), ceil(K/128)]).
Offline, so plain PyTorch."""
    N, K = w.shape
    nb, kb = triton.cdiv(N, block), triton.cdiv(K, block)
    wp = torch.zeros(nb * block, kb * block, dtype=torch.float32, device=w.de
vice)
    wp[:N, :K] = w.float()
    blocks = wp.view(nb, block, kb, block)
    amax = blocks.abs().amax(dim=(1, 3)).clamp(min=1e-10)             # [nb,
kb]
    scale = amax / FP8_MAX
    q = (blocks / scale[:, None, :, None]).clamp(-FP8_MAX, FP8_MAX).to(FP8)
    return q.view(nb * block, kb * block)[:N, :K].contiguous(), scale


@triton.jit
def fp8_block_gemm_kernel(a_ptr, b_ptr, c_ptr, sa_ptr, sb_ptr, M, N, K,
                          stride_am, stride_bn, stride_cm, stride_sam, stride
_sbn,
                          BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK
_K: tl.constexpr):
    pid_m = tl.program_id(0)
    pid_n = tl.program_id(1)
    offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    offs_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
    offs_k = tl.arange(0, BLOCK_K)
    a_ptrs = a_ptr + offs_m[:, None].to(tl.int64) * stride_am + offs_k[None,
:]
    b_ptrs = b_ptr + offs_n[None, :].to(tl.int64) * stride_bn + offs_k[:, Non
e]      # B^T tile [BK, BN]
    m_mask = offs_m < M
    n_mask = offs_n < N
    # BLOCK_N divides 128, so all columns of this tile share one weight-scale
row: n // 128.
    sb_row = (pid_n * BLOCK_N) // 128
```

```
    acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)
    for kb in range(0, tl.cdiv(K, BLOCK_K)):                    # BLOCK_K == 12
8 == one scale group
        k_left = K - kb * BLOCK_K
        a = tl.load(a_ptrs, mask=m_mask[:, None] & (offs_k[None, :] < k_lef
t), other=0.0)
        b = tl.load(b_ptrs, mask=(offs_k[:, None] < k_left) & n_mask[None,
:], other=0.0)
        sa = tl.load(sa_ptr + offs_m * stride_sam + kb, mask=m_mask, other=0.
0)   # [BLOCK_M]
        sb = tl.load(sb_ptr + sb_row * stride_sbn + kb)
# scalar
        acc += tl.dot(a, b) * (sa[:, None] * sb)                # fp8 x fp8 ->
fp32, then rescale
        a_ptrs += BLOCK_K
        b_ptrs += BLOCK_K

    c_ptrs = c_ptr + offs_m[:, None].to(tl.int64) * stride_cm + offs_n[None,
:]
    tl.store(c_ptrs, acc.to(c_ptr.dtype.element_ty), mask=m_mask[:, None] & n
_mask[None, :])


def fp8_block_gemm(a_q, a_s, b_q, b_s, out_dtype=torch.bfloat16, block_m=64,
block_n=128):
    """a_q [M, K] fp8 + a_s [M, K/128]; b_q [N, K] fp8 + b_s [N/128, K/128] -
> C [M, N] = A @ B^T."""
    M, K = a_q.shape
    N = b_q.shape[0]
    assert 128 % block_n == 0 and a_q.is_contiguous() and b_q.is_contiguous()
    c = torch.empty((M, N), dtype=out_dtype, device=a_q.device)
    grid = (triton.cdiv(M, block_m), triton.cdiv(N, block_n))
    fp8_block_gemm_kernel[grid](a_q, b_q, c, a_s, b_s, M, N, K,
                                a_q.stride(0), b_q.stride(0), c.stride(0), a_
s.stride(0), b_s.stride(0),
                                BLOCK_M=block_m, BLOCK_N=block_n, BLOCK_K=12
8, num_warps=4, num_stages=3)
    return c
```

测完之后

RTX 5090 实测（单位 TFLOP/s）：

M × N × K	bf16 cuBLAS	你的 FP8 block-scaled	激活量化本身
128 × 7168 × 2048	108	**151**	6 us

M × N × K	bf16 cuBLAS	你的 FP8 block-scaled	激活量化本身
4096 × 7168 × 2048	219	**364**	35 us
4096 × 4096 × 7168	218	**372**	112 us

- 大约是 bf16 的 1.7 倍；FP8 在 5090 上的理论峰值约是 bf16 的 2 倍，剩下的差距来自每个 K 块的缩放乘法和流水线深度。
- **激活量化不是免费的**：4096×7168 时要 112 us，占 GEMM 时间的 17%。所以实际系统会把它融合进前一个算子：FlashInfer 的 `fused_add_rmsnorm_fp8_block_quant` 就是“残差加 + RMSNorm + 1×128 FP8 量化”一次完成。把你的 F3 和 M3 合并，实现这个融合算子。
- 在 H100 上，DeepGEMM 用 wgmma + TMA + warp specialization 把同样的计算做到接近 FP8 峰值；它也提供 grouped 版本（contiguous / masked 布局），供 MoE 使用（M1、M2）。
- 进阶：把 M2 的 grouped GEMM 改成 FP8 block-scaled 版本，即 DeepSeek-V3 MoE 的推理配置。

▼ 面试追问：为什么 DeepSeek-V3 选 1×128 和 128×128 这样的粒度?

粒度越细，离群值的影响范围越小，精度越好；但每个 scale 都要存储、加载，还要在主循环里做一次乘法。128 正好等于一个 K 块，所以 kernel 每个 K 块只需要乘一次 scale，几乎没有额外开销。激活的离群值通常集中在少数通道上，per-token × 每 128 个通道一组可以把它们隔离开；权重的分布比较平滑，128×128 的块就够了，还能让一个 tile 内所有输出列共享同一个 scale。硬件层面，Blackwell 的 MXFP8（每 32 个元素）和 NVFP4（每 16 个元素）把更细的块缩放做进了 MMA 指令。

Track X · MLX / Metal

Apple silicon 上的自定义 kernel：elementwise、simdgroup 归约、simdgroup_matrix GEMM。

进阶 · Track X · MLX / Metal · 第 23 题

X1 MLX 自定义 Metal kernel 入门：融合 add + GELU

在 Mac 上写第一个 GPU kernel：MLX 的 mx.fast.metal_kernel 只要你写 Metal kernel 的函数体，其余（签名、编译、缓存、调度）它都替你做。顺便弄清 Metal 和 CUDA 在 grid 语义上的关键差别。

30 min | 难度 易 | 你写 `mine/x1_metal_elementwise.py` | 验收 `./check.sh x1` | 参考解 `solutions/x1_metal_elementwise.py`

1. 背景
2. 源码与资料
3. 要用到的知识：CUDA ↔ Metal 对照
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

MLX 是 Apple 为 Apple silicon 做的数组框架，`mlx-lm`、`mlx-vlm` 等项目在 Mac 上运行 LLM 都靠它。它的 GPU 后端是 Metal：GEMM、attention、量化 matmul、各种融合算子都是 MLX 仓库里手写的 Metal kernel（`mlx/backend/metal/kernels/`）。`mx.fast.metal_kernel` 让你不改 MLX 源码，就在 Python 里写出一个自定义 kernel，和 MLX 的其他操作混合使用，也能被 `mx.compile` 处理。

Apple GPU 的特点：**统一内存**（CPU 和 GPU 共用同一块内存，不需要拷贝），带宽从 M2 的约 100 GB/s 到 Ultra 芯片的约 800 GB/s；没有独立显存，所以“模型能不能放下”取决于整机内存。

源码与资料

MLX: Custom Metal Kernels · `mlx › custom_metal_kernels.rst` · `mlx › custom_kernel.cpp（MLX 怎么生成签名）` · `mlx › Metal debugger（抓 GPU trace）` · Metal Shading Language Specification · `mlx › kernels/`

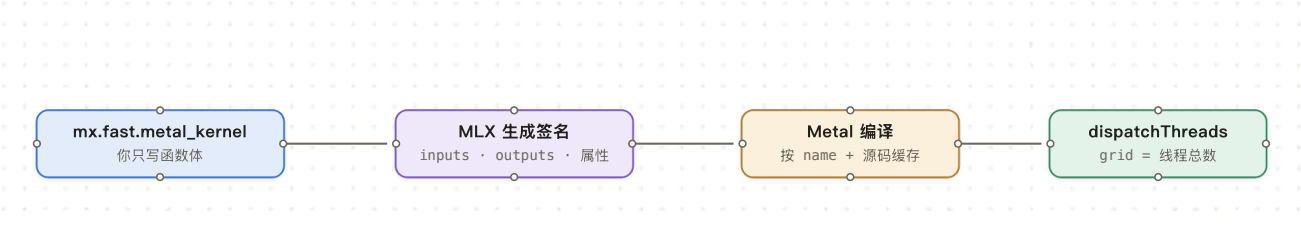
要用到的知识：CUDA ↔ Metal 对照

CUDA	Metal / MLX
`__global__ void k(...)`	MLX 根据 `input_names`、`output_names` 和函数体中用到的属性**自动生成签名**；你只写函数体
block / thread	threadgroup / thread
warp（32）	simdgroup（Apple GPU 上是 32）
`blockIdx`、`threadIdx`	`threadgroup_position_in_grid`、`thread_position_in_threadgroup`
`blockIdx.x * blockDim.x + threadIdx.x`	`thread_position_in_grid` 直接给出全局编号
`<<<grid（block 数）, block>>>`	`grid=`**线程总数**、`threadgroup=` 每组线程数（Metal 的 dispatchThreads；grid 不需要是 threadgroup 的整数倍）
`__shared__`	`threadgroup` 内存
`template <typename T>`	`template=[("T", mx.float16)]`：模板参数在调用时传入

- 输入数组变成 `const device T* x`（很小的输入可能被放在 `constant` 地址空间，见 X2），Python 标量变成 `constant int& n` 这样的参数。
- 默认 `ensure_row_contiguous=True`：输入不连续时，MLX 会先复制一份连续的再交给 kernel。
- `header=` 放辅助函数；数学函数在 `metal::` 命名空间下，`metal::precise::tanh` 是精确版本，`metal::fast::` 是快速近似版本。

图解与类比

mx.fast.metal_kernel 做了什么



类比 · 报名人数，而不是班级数

CUDA 的 `<<<grid, block>>>` 说的是“开几个班、每班几个人”；Metal / MLX 的 `grid=` 说的是“总共多少个学生”，`threadgroup=` 才是每班人数，最后一个班可以不满。用 CUDA 的习惯把班级数填进 grid，就会多开出 256 倍的学生。

统一内存：不需要 host ↔ device 拷贝



类比 · 同一张桌子 vs 两个房间

独立显卡像两个房间：CPU 在客厅，GPU 在书房，资料要通过走廊（PCIe）来回搬。Apple 的统一内存是两人共用一张桌子，不用搬资料；代价是这张桌子比 NVIDIA 书房里的桌子（HBM）小一些、取东西也慢一些（带宽低一个数量级）。

题目要求

在 `mine/x1_metal_elementwise.py` 中补全三段 Metal 源码：

1. `_HEADER`：`inline float gelu_tanh(float z)`。
2. `_SRC`：每个线程处理一个元素，`out[i] = T(gelu_tanh(float(x[i]) + float(y[i])))`。
3. `_SRC_VEC`：每个线程处理 `N_PER` 个连续元素（模板参数）。

要求支持 fp32 / fp16 / bf16、任意形状、转置过的输入。

题干

mine/x1_metal_elementwise.py 42 行

```python
"""X1 · fused add + GELU as an MLX custom Metal kernel — YOUR IMPLEMENTATION.

See the drill page for the spec; the reference lives in solutions/."""
import mlx.core as mx
```

```
_HEADER = """
    // TODO: write `inline float gelu_tanh(float z)` (metal::precise::tanh)
inline float gelu_tanh(float z) { return z; }
"""

# One element per thread. T is a template parameter, so one source serves fp32 /
fp16 / bf16.
_SRC = """
    // TODO: i = thread_position_in_grid.x; bounds check against n; out[i] = T(ge
lu_tanh(float(x[i]) + float(y[i])))
"""

# N_PER elements per thread, contiguous: fewer threads, each issuing wider loads.
_SRC_VEC = """
    // TODO: each thread handles N_PER consecutive elements starting at thread_po
sition_in_grid.x * N_PER
"""

_kernel = mx.fast.metal_kernel(name="add_gelu", input_names=["x", "y", "n"], outp
ut_names=["out"],
                               source=_SRC, header=_HEADER)
_kernel_vec = mx.fast.metal_kernel(name="add_gelu_vec", input_names=["x", "y",
"n"], output_names=["out"],
                                   source=_SRC_VEC, header=_HEADER)


def add_gelu(x: mx.array, y: mx.array, threadgroup: int = 256) -> mx.array:
    assert x.shape == y.shape and x.dtype == y.dtype
    n = x.size
    # ensure_row_contiguous (the default) makes MLX hand the kernel contiguous co
pies when needed.
    return _kernel(inputs=[x, y, n], template=[("T", x.dtype)],
                   grid=(max(n, 1), 1, 1), threadgroup=(threadgroup, 1, 1),
                   output_shapes=[x.shape], output_dtypes=[x.dtype])[0]


def add_gelu_vec(x: mx.array, y: mx.array, n_per: int = 4, threadgroup: int = 25
6) -> mx.array:
    assert x.shape == y.shape and x.dtype == y.dtype
    n = x.size
    threads = max((n + n_per - 1) // n_per, 1)
    return _kernel_vec(inputs=[x, y, n], template=[("T", x.dtype), ("N_PER", n_pe
r)],
                       grid=(threads, 1, 1), threadgroup=(threadgroup, 1, 1),
                       output_shapes=[x.shape], output_dtypes=[x.dtype])[0]
```

验收

```
./check.sh x1               # 自动使用 .venv-mlx (arm64 Python + mlx)
.venv-mlx/bin/python bench_mlx.py x1
```

tests/test_x1_metal_elementwise.py 40 行

```
import math

import pytest

mx = pytest.importorskip("mlx.core", reason="X track needs MLX on an Apple-silico
n Mac")
from drill import load  # noqa: E402


def ref(x, y):
    z = (x + y).astype(mx.float32)
    return 0.5 * z * (1 + mx.tanh(math.sqrt(2 / math.pi) * (z + 0.044715 * z **
3)))


@pytest.mark.parametrize("n", [1, 255, 256, 257, 1_000_003])
def test_add_gelu_fp32(n):
    m = load("x1_metal_elementwise")
    x, y = mx.random.normal((n,)) * 3, mx.random.normal((n,))
    assert mx.allclose(m.add_gelu(x, y), ref(x, y), atol=1e-5, rtol=1e-5).item()


@pytest.mark.parametrize("dtype", [mx.float16, mx.bfloat16])
def test_add_gelu_half_2d(dtype):
    m = load("x1_metal_elementwise")
    x, y = mx.random.normal((7, 333)).astype(dtype), mx.random.normal((7, 333)).a
stype(dtype)
    out = m.add_gelu(x, y)
    assert out.dtype == dtype and out.shape == (7, 333)
    assert mx.allclose(out.astype(mx.float32), ref(x, y), atol=2e-2, rtol=2e-2).i
tem()


def test_transposed_input():
    m = load("x1_metal_elementwise")
    x, y = mx.random.normal((48, 64)).T, mx.random.normal((64, 48))
    assert mx.allclose(m.add_gelu(x, y), ref(x, y), atol=1e-5).item()


@pytest.mark.parametrize("n", [1, 5, 1023, 4096, 100_001])
def test_add_gelu_vec(n):
    m = load("x1_metal_elementwise")
```

```
    x, y = mx.random.normal((n,)), mx.random.normal((n,))
    assert mx.allclose(m.add_gelu_vec(x, y, n_per=4), ref(x, y), atol=1e-5).item
()
```

提示

L1 函数体只有三行 方向（轻度剧透） 收起

`uint i = thread_position_in_grid.x;`，检查 `i < n`，写回。grid 传的是线程总数 n，所以不用再乘 threadgroup 大小。

L2 常见坑 拆解与常见坑（中度剧透） 收起

- 把 grid 当成 threadgroup 的个数（CUDA 习惯）：会启动 256 倍的线程，越界写坏内存，或者直接报错。
- 用 `half` 做中间计算：`z³` 很快就溢出了。全程用 `float`，存回时转成 `T`。
- 在 `_SRC_VEC` 里忘了对最后一个线程做边界检查。
- 改了 Metal 源码却没有生效：MLX 按 `name` 和源码缓存编译结果，在同一个 Python 进程里重新创建 kernel 时注意 name 的唯一性。

L3 两段源码 几乎是答案（重度剧透） 收起

```
// _SRC
uint i = thread_position_in_grid.x;
if (i < n) out[i] = T(gelu_tanh(float(x[i]) + float(y[i])));
// _SRC_VEC
uint base = thread_position_in_grid.x * N_PER;
for (uint j = 0; j < N_PER; ++j) { uint i = base + j; if (i < n) out[i] = T(gel
```

参考解

解 **solutions/x1_metal_elementwise.py** 先自己写完、跑过测试再看 收起

solutions/x1_metal_elementwise.py 58 行

```
"""X1 · First custom Metal kernel in MLX: fused add + GELU — reference soluti
on.

mx.fast.metal_kernel(name, input_names, output_names, source, header) compile
s the BODY of a Metal
kernel: MLX writes the signature for you. Inputs arrive as `const device T* x
` (or `constant`
scalars), outputs as `device T* out`, and the Metal attributes you mention in
the body
(thread_position_in_grid, threadgroup_position_in_grid, ...) are added as par
ameters.

Grid semantics differ from CUDA: `grid` is the TOTAL number of threads per di
mension (Metal's
dispatchThreads), not the number of threadgroups, and it need not be a multip
le of `threadgroup`.
"""
import mlx.core as mx

_HEADER = """
inline float gelu_tanh(float z) {
    // 0.5 z (1 + tanh(sqrt(2/pi) (z + 0.044715 z^3)))
    return 0.5f * z * (1.0f + metal::precise::tanh(0.7978845608028654f * (z +
0.044715f * z * z * z)));
}
"""

# One element per thread. T is a template parameter, so one source serves fp3
2 / fp16 / bf16.
_SRC = """
    uint i = thread_position_in_grid.x;
    if (i < n) {
        out[i] = T(gelu_tanh(float(x[i]) + float(y[i])));   // math in fp32,
store in T
    }
"""

# N_PER elements per thread, contiguous: fewer threads, each issuing wider lo
ads.
_SRC_VEC = """
    uint base = thread_position_in_grid.x * N_PER;
    for (uint j = 0; j < N_PER; ++j) {
```

```
        uint i = base + j;
        if (i < n) out[i] = T(gelu_tanh(float(x[i]) + float(y[i])));
    }
"""

_kernel = mx.fast.metal_kernel(name="add_gelu", input_names=["x", "y", "n"],
output_names=["out"],
                               source=_SRC, header=_HEADER)
_kernel_vec = mx.fast.metal_kernel(name="add_gelu_vec", input_names=["x",
"y", "n"], output_names=["out"],
                                   source=_SRC_VEC, header=_HEADER)


def add_gelu(x: mx.array, y: mx.array, threadgroup: int = 256) -> mx.array:
    assert x.shape == y.shape and x.dtype == y.dtype
    n = x.size
    # ensure_row_contiguous (the default) makes MLX hand the kernel contiguou
s copies when needed.
    return _kernel(inputs=[x, y, n], template=[("T", x.dtype)],
                   grid=(max(n, 1), 1, 1), threadgroup=(threadgroup, 1, 1),
                   output_shapes=[x.shape], output_dtypes=[x.dtype])[0]


def add_gelu_vec(x: mx.array, y: mx.array, n_per: int = 4, threadgroup: int =
256) -> mx.array:
    assert x.shape == y.shape and x.dtype == y.dtype
    n = x.size
    threads = max((n + n_per - 1) // n_per, 1)
    return _kernel_vec(inputs=[x, y, n], template=[("T", x.dtype), ("N_PER",
n_per)],
                       grid=(threads, 1, 1), threadgroup=(threadgroup, 1, 1),
                       output_shapes=[x.shape], output_dtypes=[x.dtype])[0]
```

测完之后

Apple M2（24 GB）实测，fp16，n = 2^{24}：MLX 自带操作（`mx.compile` 融合后）2191 us · 46 GB/s，你的单元素版本 2381 us · 42 GB/s，4 元素版本 2365 us · 43 GB/s。

- 先测这台机器实际能达到的带宽：同一时刻 `x + 1`（fp32，2^{24} 个元素）是 59 GB/s。M2 的标称带宽约 100 GB/s，测试时机器负载较高（load average ≈ 17）。memory-bound kernel 的性能要和**实测的上限**比较，而不是和标称值比较。
- 和 CUDA 一样，elementwise kernel 的速度由带宽决定，你的实现和 MLX 内置操作在同一水平。

- 用 Xcode 的 Metal System Trace（Instruments）或 `mx.metal.start_capture()` 抓一次 GPU trace，看 kernel 的实际时间和占用率。
- MLX 的 `mx.compile` 也能把多个逐元素操作自动融合成一个 kernel。什么时候还需要手写？（当融合需要 threadgroup 内存、跨线程归约或特殊的数据布局时：X2、X3。）

▼ 面试追问：Apple GPU 和 NVIDIA GPU 写 kernel 时最大的区别是什么？

编程模型很接近（threadgroup ≈ block，simdgroup ≈ warp，threadgroup 内存 ≈ shared memory），主要区别在于：统一内存，没有 host 与 device 之间的拷贝，但带宽比数据中心 GPU 低一个数量级，所以 LLM 推理几乎总是 memory-bound，量化（4-bit / 8-bit 分组量化）是最有效的手段；矩阵单元是 simdgroup 级别的 8×8 `simdgroup_matrix`（M5 起还有 GPU 内的 neural accelerator），而不是 Tensor Core 那样更大的 MMA；工具链是 Metal Shading Language（C++14 方言）和 Xcode / Instruments，而不是 nvcc / Nsight。

进阶 · Track X · MLX / Metal · 第 24 题

X2 Metal 归约：simdgroup + threadgroup 内存的 softmax / RMSNorm

把 C2 的归约结构搬到 Apple GPU：simd_sum / simd_max 负责 simdgroup 内，threadgroup 内存加 barrier 负责组内汇总。用它写出逐行 softmax 和 RMSNorm，并踩一个 MLX 特有的坑。

45 min | 难度 中 | 你写 `mine/x2_metal_softmax.py` | 验收 `./check.sh x2` | 参考解 `solutions/x2_metal_softmax.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

softmax、RMSNorm、LayerNorm 在 MLX 里都有内置的融合实现（`mx.softmax`、`mx.fast.rms_norm`、`mx.fast.layer_norm`）。自己写一遍，是掌握 Metal 归约写法最快的方式，也是今后写新融合算子（比如带量化的 norm、DiT 的 adaLN）的基础。

源码与资料

`mlx › MLX softmax_single_row` · `mlx › MLX softmax_looped` · `mlx › MLX rms_norm.metal` · `mlx › MLX layer_norm.metal` · Metal Shading Language Specification（SIMD-group functions 一节）

要用到的知识

- **simdgroup 归约**：`simd_sum(v)`、`simd_max(v)` 直接返回 32 个 lane 的和 / 最大值，**每个 lane 都能拿到结果**（相当于 CUDA 里用 `__shfl_xor_sync` 做完一整轮 butterfly）。
- 内建属性：`thread_index_in_simdgroup`（lane）、`simdgroup_index_in_threadgroup`、`threads_per_threadgroup`、

`threadgroup_position_in_grid`。

- threadgroup 内存：在函数体中声明 `threadgroup float scratch[32];`，同步用 `threadgroup_barrier(mem_flags::mem_threadgroup)`。
- 两级归约：`simd_sum` → 每个 simdgroup 的 lane 0 写入 `scratch[sg]` → barrier → 每个 simdgroup 都把 nsg 个部分和读回来，再做一次 `simd_sum` → barrier（之后 scratch 才能复用）。
- **地址空间的坑**：MLX 可能把很小的输入数组放在 `constant` 地址空间而不是 `device`。写 `const device T* xr = x + row * n;` 时，遇到小输入会编译失败（`cannot initialize ... 'const device float *' with ... 'const constant float *'`），用 `auto xr = x + row * n;` 就没有问题。本套题的参考解最初就踩了这个坑：只有 shape 为 (1,1) 和 (2,3) 的测试失败了。

图解与类比

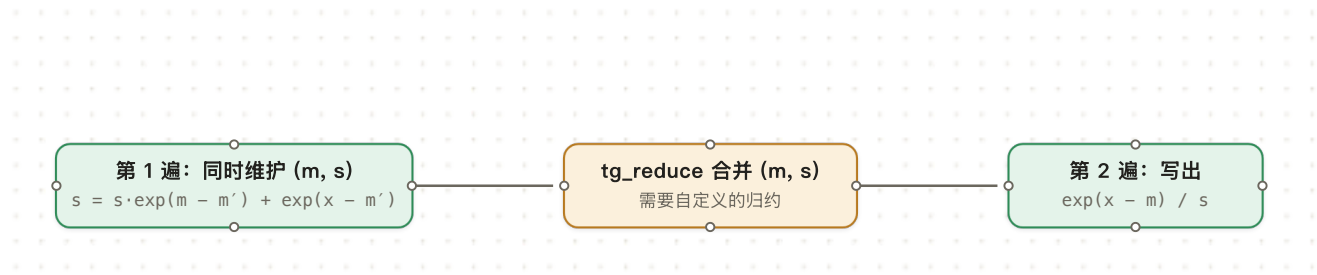
threadgroup 级归约



类比 · 快递送到前台还是仓库

MLX 可能把很小的输入放进 `constant` 地址空间（前台），大的放进 `device`（仓库）。写死 `const device T*` 等于坚持“包裹一定在仓库”，前台的包裹就取不到（编译错误）；写 `auto` 就是“送到哪我去哪取”。

改进：online softmax 把三遍读变成两遍



跨线程合并 (m, s) 对时，要先取 m 的最大值，再把每个线程的 s 按 $\exp(m_i - m)$ 折算后相加，和 F2 的 merge_state 是同一个公式。

类比 · simdgroup 是一桌人

一个 simdgroup 的 32 个线程像一桌吃饭的人，`simd_sum` 是这一桌直接凑钱，不用离开座位；threadgroup 内存是整个包间的公共托盘，几桌要先把各自的钱放到托盘上（写 scratch），等大家都放好（barrier），再由每桌派人来数托盘。

题目要求

在 `mine/x2_metal_softmax.py` 中补全：

1. `_HEADER`：`template <bool IS_MAX> inline float tg_reduce(float v, threadgroup float* scratch, uint lane, uint sg, uint nsg)`，返回整个 threadgroup 的归约结果（每个线程都要拿到）。
2. `_SOFTMAX`：一个 threadgroup 处理一行，三遍：求 max、求 exp 的和、写出结果。
3. `_RMSNORM`：平方和 → rstd → `x · rstd · w`。

wrapper 已写好：一行一个 threadgroup，线程数为 32–1024，按行长决定。

题干

mine/x2_metal_softmax.py 46 行

```
"""X2 · row softmax + RMSNorm with simdgroup reductions — YOUR IMPLEMENTATION.

See the drill page for the spec; the reference lives in solutions/."""
import mlx.core as mx

# Block-wide reduction helper: every thread returns the reduced value. `scratch` holds one
# slot per simdgroup; the trailing barrier lets the caller reuse `scratch` right away.
_HEADER = """
    // TODO: template <bool IS_MAX> inline float tg_reduce(float v, threadgroup float* scratch, uint lane, uint sg, uint nsg)
    // TODO: simd_max/simd_sum -> lane 0 writes scratch[sg] -> barrier -> reduce the nsg partials -> barrier
"""

_SOFTMAX = """
    // TODO: row = threadgroup_position_in_grid.x; threads stride over the row; use `auto xr = x + row * n` (address space!)
    // TODO: pass 1 max, pass 2 sum exp(x - max), pass 3 write
"""
```

```
_RMSNORM = """
    // TODO: sum of squares -> tg_reduce -> rstd = rsqrt(ss / n + eps) -> out = x
* rstd * w
"""

_softmax = mx.fast.metal_kernel(name="row_softmax", input_names=["x", "ncols"], o
utput_names=["out"],
                                source=_SOFTMAX, header=_HEADER)
_rmsnorm = mx.fast.metal_kernel(name="row_rmsnorm", input_names=["x", "w", "ncol
s", "eps"], output_names=["out"],
                                source=_RMSNORM, header=_HEADER)


def _threads(ncols: int) -> int:
    # one simdgroup per 32..1024 columns, at most 1024 threads per threadgroup
    return min(1024, max(32, 1 << (max(ncols, 1) - 1).bit_length())) if ncols < 1
024 else 1024


def softmax(x: mx.array) -> mx.array:
    assert x.ndim == 2
    rows, ncols = x.shape
    t = _threads(ncols)
    return _softmax(inputs=[x, ncols], template=[("T", x.dtype)], grid=(rows * t,
1, 1), threadgroup=(t, 1, 1),
                    output_shapes=[x.shape], output_dtypes=[x.dtype])[0]


def rms_norm(x: mx.array, w: mx.array, eps: float = 1e-6) -> mx.array:
    assert x.ndim == 2 and w.shape == (x.shape[1],)
    rows, ncols = x.shape
    t = _threads(ncols)
    return _rmsnorm(inputs=[x, w, ncols, eps], template=[("T", x.dtype)], grid=(r
ows * t, 1, 1),
                    threadgroup=(t, 1, 1), output_shapes=[x.shape], output_dtypes
=[x.dtype])[0]
```

验收

```
./check.sh x2
.venv-mlx/bin/python bench_mlx.py x2
```

tests/test_x2_metal_softmax.py 35 行

```
import pytest
```

```
mx = pytest.importorskip("mlx.core", reason="X track needs MLX on an Apple-silico
n Mac")
from drill import load  # noqa: E402


@pytest.mark.parametrize("shape", [(1, 1), (4, 7), (33, 128), (17, 1000), (8, 409
7), (2, 32000)])
def test_softmax(shape):
    m = load("x2_metal_softmax")
    x = mx.random.normal(shape) * 5
    assert mx.allclose(m.softmax(x), mx.softmax(x, axis=-1), atol=1e-6, rtol=1e-
5).item()


def test_softmax_large_values():
    m = load("x2_metal_softmax")
    x = mx.array([[1000.0, 1001.0, 1002.0], [-1e4, 0.0, 1e4]])
    out = m.softmax(x)
    assert mx.all(mx.isfinite(out)).item()
    assert mx.allclose(out, mx.softmax(x, axis=-1)).item()


def test_softmax_fp16():
    m = load("x2_metal_softmax")
    x = mx.random.normal((10, 257)).astype(mx.float16)
    out = m.softmax(x)
    assert out.dtype == mx.float16
    assert mx.allclose(out.astype(mx.float32), mx.softmax(x.astype(mx.float32), a
xis=-1), atol=1e-3).item()


@pytest.mark.parametrize("shape", [(1, 16), (7, 333), (32, 4096)])
def test_rms_norm(shape):
    m = load("x2_metal_softmax")
    x, w = mx.random.normal(shape), mx.random.normal((shape[1],))
    ref = mx.fast.rms_norm(x, w, 1e-6)
    assert mx.allclose(m.rms_norm(x, w, 1e-6), ref, atol=1e-4, rtol=1e-4).item()
```

提示

L1 tg_reduce 的结构 方向（轻度剧透） 收起

```
v = IS_MAX ? simd_max(v) : simd_sum(v);
if (lane == 0) scratch[sg] = v;
threadgroup_barrier(mem_flags::mem_threadgroup);
v = lane < nsg ? scratch[lane] : (IS_MAX ? -INFINITY : 0.0f);
v = IS_MAX ? simd_max(v) : simd_sum(v);
threadgroup_barrier(mem_flags::mem_threadgroup);
return v;
```

L2 常见坑 拆解与常见坑（中度剧透） 收起

- 第二轮只让 simdgroup 0 去归约：其他 simdgroup 拿不到结果，接下来的 `x - max` 就是错的。让每个 simdgroup 都读 scratch 再归约一次，最简单。
- 少了最后那个 barrier：softmax 的第二次 `tg_reduce` 会在别人还在读 scratch 时覆盖它。
- 循环变量和 `ncols`（`constant int&`）的有符号 / 无符号比较：先 `uint n = uint(ncols);`。

L3 softmax 函数体 几乎是答案（重度剧透） 收起

```
uint row = threadgroup_position_in_grid.x, tid = thread_position_in_threadgroup
uint nthr = threads_per_threadgroup.x, lane = thread_index_in_simdgroup;
uint sg = simdgroup_index_in_threadgroup, nsg = (nthr + 31) / 32, n = uint(ncol
threadgroup float scratch[32];
auto xr = x + row * n;
float m = -INFINITY;
for (uint c = tid; c < n; c += nthr) m = max(m, float(xr[c]));
m = tg_reduce<true>(m, scratch, lane, sg, nsg);
float s = 0.0f;
for (uint c = tid; c < n; c += nthr) s += metal::exp(float(xr[c]) - m);
s = tg_reduce<false>(s, scratch, lane, sg, nsg);
for (uint c = tid; c < n; c += nthr) out[row * n + c] = T(metal::exp(float(xr[c
```

参考解

解 **solutions/x2_metal_softmax.py** 先自己写完、跑过测试再看 收起 ▾

solutions/x2_metal_softmax.py 95 行

```
"""X2 · Row softmax and RMSNorm in Metal: simdgroup reductions + threadgroup
memory — reference solution.

Apple GPU vocabulary <-> CUDA:
    threadgroup            <-> block           threadgroup memory     <-> sha
red memory
    simdgroup (32 threads) <-> warp            simd_sum / simd_max    <-> war
p shuffle reductions
    threadgroup_barrier(mem_flags::mem_threadgroup)                   <-> __s
yncthreads()
One threadgroup per row: each thread strides over the row, simd_* reduces ins
ide a simdgroup,
then lane 0 of each simdgroup posts its partial to threadgroup memory for the
final reduction.
"""
import mlx.core as mx

# Block-wide reduction helper: every thread returns the reduced value. `scrat
ch` holds one
# slot per simdgroup; the trailing barrier lets the caller reuse `scratch` ri
ght away.
_HEADER = """
template <bool IS_MAX>
inline float tg_reduce(float v, threadgroup float* scratch, uint lane, uint s
g, uint nsg) {
    v = IS_MAX ? simd_max(v) : simd_sum(v);
    if (lane == 0) scratch[sg] = v;
    threadgroup_barrier(mem_flags::mem_threadgroup);
    float init = IS_MAX ? -INFINITY : 0.0f;
    v = lane < nsg ? scratch[lane] : init;
    v = IS_MAX ? simd_max(v) : simd_sum(v);       // every simdgroup reduces
the partials itself
    threadgroup_barrier(mem_flags::mem_threadgroup);
    return v;
}
"""

_SOFTMAX = """
    uint row = threadgroup_position_in_grid.x;
    uint tid = thread_position_in_threadgroup.x;
    uint nthr = threads_per_threadgroup.x;
    uint lane = thread_index_in_simdgroup;
```

```
    uint sg = simdgroup_index_in_threadgroup;
    uint nsg = (nthr + 31) / 32;
    threadgroup float scratch[32];
    uint n = uint(ncols);
    // `auto`, not `const device T*`: MLX may pass small inputs in the `const
ant` address space.
    auto xr = x + row * n;

    float m = -INFINITY;                                    // pass 1: row max
    for (uint c = tid; c < n; c += nthr) m = max(m, float(xr[c]));
    m = tg_reduce<true>(m, scratch, lane, sg, nsg);

    float s = 0.0f;                                         // pass 2: sum of
exp(x - max)
    for (uint c = tid; c < n; c += nthr) s += metal::exp(float(xr[c]) - m);
    s = tg_reduce<false>(s, scratch, lane, sg, nsg);

    float inv = 1.0f / s;                                   // pass 3: normali
se and write
    for (uint c = tid; c < n; c += nthr) out[row * n + c] = T(metal::exp(floa
t(xr[c]) - m) * inv);
"""

_RMSNORM = """
    uint row = threadgroup_position_in_grid.x;
    uint tid = thread_position_in_threadgroup.x;
    uint nthr = threads_per_threadgroup.x;
    uint lane = thread_index_in_simdgroup;
    uint sg = simdgroup_index_in_threadgroup;
    uint nsg = (nthr + 31) / 32;
    threadgroup float scratch[32];
    uint n = uint(ncols);
    // `auto`, not `const device T*`: MLX may pass small inputs in the `const
ant` address space.
    auto xr = x + row * n;

    float ss = 0.0f;
    for (uint c = tid; c < n; c += nthr) { float v = float(xr[c]); ss += v *
v; }
    ss = tg_reduce<false>(ss, scratch, lane, sg, nsg);
    float rstd = metal::precise::rsqrt(ss / float(n) + eps);
    for (uint c = tid; c < n; c += nthr) out[row * n + c] = T(float(xr[c]) *
rstd * float(w[c]));
"""

_softmax = mx.fast.metal_kernel(name="row_softmax", input_names=["x", "ncol
s"], output_names=["out"],
                                source=_SOFTMAX, header=_HEADER)
_rmsnorm = mx.fast.metal_kernel(name="row_rmsnorm", input_names=["x", "w", "n
cols", "eps"], output_names=["out"],
```

```
                                    source=_RMSNORM, header=_HEADER)


def _threads(ncols: int) -> int:
    # one simdgroup per 32..1024 columns, at most 1024 threads per threadgrou
p
    return min(1024, max(32, 1 << (max(ncols, 1) - 1).bit_length())) if ncols
< 1024 else 1024


def softmax(x: mx.array) -> mx.array:
    assert x.ndim == 2
    rows, ncols = x.shape
    t = _threads(ncols)
    return _softmax(inputs=[x, ncols], template=[("T", x.dtype)], grid=(rows
* t, 1, 1), threadgroup=(t, 1, 1),
                    output_shapes=[x.shape], output_dtypes=[x.dtype])[0]


def rms_norm(x: mx.array, w: mx.array, eps: float = 1e-6) -> mx.array:
    assert x.ndim == 2 and w.shape == (x.shape[1],)
    rows, ncols = x.shape
    t = _threads(ncols)
    return _rmsnorm(inputs=[x, w, ncols, eps], template=[("T", x.dtype)], gri
d=(rows * t, 1, 1),
                    threadgroup=(t, 1, 1), output_shapes=[x.shape], output_dt
ypes=[x.dtype])[0]
```

测完之后

Apple M2 实测（fp16，2048 行）：

N	`mx.softmax`	你的实现
1024	**310 us**	640 us
4096	**716 us**	1152 us
32000	4432 us	**4032 us**

- 行较短时 MLX 更快：它的 softmax 在行能放进寄存器时只读一次输入（“looped”和“single row”两种实现），而你的三遍写法要读三次。行很长时（32000，词表大小）两者都受带宽限制，你的反而略快。
- 改进一：**online softmax**，像 T5 那样一遍同时维护 (m, s)，读取次数从 3 次降到 2 次。

- 改进二：每个线程一次读 4 个连续元素（`vec<half, 4>` 或手动展开），减少指令数。
- 读源码：MLX 的 `mlx/backend/metal/kernels/softmax.h` 和 `rms_norm.metal`，对照它们是怎么选择每行线程数和单遍 / 多遍实现的。

▼ 面试追问：simd_sum 和 CUDA 的 warp shuffle 归约有什么区别?

语义上，`simd_sum` 相当于一整轮 butterfly 归约（`__shfl_xor_sync` 5 轮）：simdgroup 内所有 lane 都得到结果。CUDA 里你通常手写这 5 轮，或者使用 `cg::reduce` / CUB 的 WarpReduce；Metal 直接提供了这条内建函数，还有 `simd_prefix_inclusive_sum`、`simd_shuffle_xor` 等更底层的操作。跨 simdgroup 的部分在两边完全一样：写进共享内存（threadgroup 内存），同步，再归约一次。

进阶 · Track X · MLX / Metal · 第 25 题

X3 Metal GEMM：threadgroup tiling → simdgroup_matrix

在 Apple GPU 上走一遍 C4 的路线：先用 threadgroup 内存做 tiling，再换成 simdgroup_matrix 8×8 矩阵指令，性能翻倍，并且看清和 MLX 自带的 steel GEMM 还差在哪里。

75 min | 难度 难 | 你写 `mine/x3_metal_matmul.py` | 验收 `./check.sh x3` | 参考解 `solutions/x3_metal_matmul.py`

1. 背景
2. 源码与资料
3. 要用到的知识
4. 图解与类比
5. 题目要求
6. 题干
7. 验收
8. 提示
9. 参考解
10. 测完之后

背景

Apple GPU 没有 NVIDIA 那样的大块 Tensor Core MMA，它的矩阵加速单位是 **simdgroup_matrix**：一个 simdgroup（32 个线程）协作持有一个 8×8 的矩阵，`simdgroup_multiply_accumulate` 一次完成 8×8×8 的乘加。MLX 的 GEMM（“steel” kernel，`mlx/backend/metal/kernels/steel/`）、attention 和量化 matmul 都建立在它上面。M5 起，GPU 里还加入了 neural accelerator，MLX 源码中对应 `*_nax` 系列 kernel。

源码与资料

`mlx › MLX steel mma.h` · `mlx › steel loader.h` · `mlx › steel gemm.h` · `mlx › MLX quantized.h (qmv / qmm)` · `mlx › MLX sdpa_vector.h` · Metal Shading Language Specification（simdgroup_matrix）

要用到的知识

- **threadgroup tiling**（与 C4 v1 相同）：16×16 的 threadgroup 计算 C 的一个 16×16 块，A、B 的 tile 放进 threadgroup 内存，每个值复用 16 次。
- **simdgroup_matrix**：

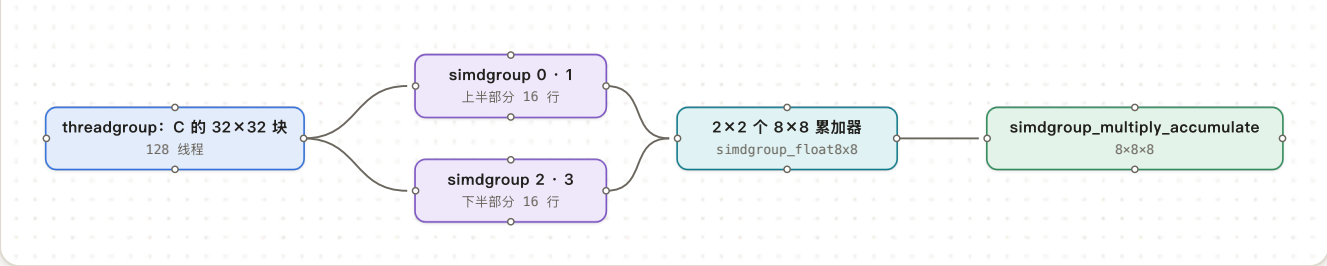
```
#include <metal_simdgroup_matrix>                    // 通过 metal_kernel 的 header
simdgroup_float8x8 a, b, c = make_filled_simdgroup_matrix<float, 8, 8>(0.0f);
simdgroup_load(a, ptr, row_stride);                  // 从 threadgroup / device 内存
simdgroup_multiply_accumulate(c, a, b, c);           // c = a @ b + c
simdgroup_store(c, ptr, row_stride);
```

ptr 指向这个 8×8 块的第一个元素，row_stride 是这块数据在内存中的行跨度（以元素计）。

- 本题的分工：一个 threadgroup 有 128 个线程（4 个 simdgroup，排成 2×2），计算 C 的 32×32 块；每个 simdgroup 负责一个 16×16 的四分之一，持有 2×2 个 8×8 累加器；BK = 16，每个 K 块做两步 8 宽的 MMA。
- 二维 grid：MLX 的 `grid` 是线程数，所以 `grid = (128 · N 方向的 tile 数, M 方向的 tile 数, 1)`，`threadgroup = (128, 1, 1)`；kernel 里用 `threadgroup_position_in_grid.xy` 定位 tile。
- 边界：tile 加载时越界部分填 0；累加器先 `simdgroup_store` 到 threadgroup 内存，再由 128 个线程按 mask 写回 C。

图解与类比

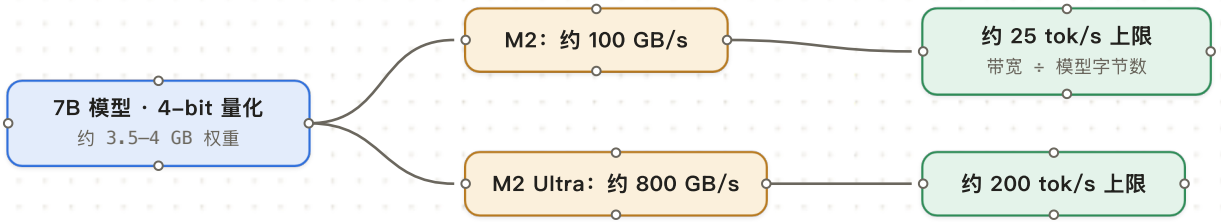
simdgroup_matrix GEMM 的分工



类比 · 32 个人合抬一块拼图板

`simdgroup_matrix` 是一个 simdgroup（32 个线程）合力持有一块 8×8 的拼图板，一条指令把两块板相乘再加到第三块上。一个 threadgroup 的 4 组人各拼 4 块（2×2），合起来就是 C 的一个 32×32 区域。MLX 的 steel GEMM 让每组人同时拼更多块，复用更多，所以更快。

Mac 上 decode 的速度上限



类比 · 每生成一个字都要把书读一遍

decode 时每生成一个 token，都要把整个模型的权重从内存里读一遍，就像每写一个字都要从头翻一遍参考书。翻书的速度（带宽）除以书的厚度（模型字节数）就是写字速度的上限：所以把书印得更薄（4-bit 量化）比换一支更快的笔（更快的 GPU 计算）更有效。

题目要求

在 `mine/x3_metal_matmul.py` 中补全 `_TILED` 和 `_SIMD` 两段 Metal 源码（wrapper 已写好）。fp32 和 fp16 输入，fp32 累加；M、N、K 任意（包括 1、非 32 的倍数），与 `a @ b` 的误差 ⩽ 1e-3。

题干

mine/x3_metal_matmul.py 39 行

```
"""X3 · Metal matmul: threadgroup tiling + simdgroup_matrix — YOUR IMPLEMENTATIO
N.

See the drill page for the spec; the reference lives in solutions/."""
import mlx.core as mx

_TILED = """
    // TODO: 16x16 threadgroup, one C element per thread; As/Bs [16][16] in threa
dgroup memory, zero-filled edges
    // TODO: K loop: load tiles, barrier, 16 FMAs, barrier; masked store
"""

_SIMD = """
    // TODO: 128 threads = 4 simdgroups (2x2) on a 32x32 C tile, BK = 16
    // TODO: stage A [32x16] and B [16x32] in threadgroup memory; each simdgroup:
2x2 simdgroup_float8x8 accumulators
```

```
    // TODO: simdgroup_load / simdgroup_multiply_accumulate; simdgroup_store into
Cs, barrier, masked copy to C
"""

_tiled = mx.fast.metal_kernel(name="matmul_tiled", input_names=["A", "B", "M",
"N", "K"], output_names=["C"], source=_TILED)
_simd = mx.fast.metal_kernel(name="matmul_simd", input_names=["A", "B", "M", "N",
"K"], output_names=["C"], source=_SIMD,
                             header="#include <metal_simdgroup_matrix>\n")


def _check(a, b):
    assert a.ndim == 2 and b.ndim == 2 and a.shape[1] == b.shape[0] and a.dtype =
= b.dtype
    return a.shape[0], b.shape[1], a.shape[1]


def matmul_tiled(a: mx.array, b: mx.array) -> mx.array:
    M, N, K = _check(a, b)
    grid = (((N + 15) // 16) * 16, ((M + 15) // 16) * 16, 1)       # threads, not
threadgroups
    return _tiled(inputs=[a, b, M, N, K], template=[("T", a.dtype)], grid=grid, t
hreadgroup=(16, 16, 1),
                  output_shapes=[(M, N)], output_dtypes=[a.dtype])[0]


def matmul_simd(a: mx.array, b: mx.array) -> mx.array:
    M, N, K = _check(a, b)
    # one 128-thread threadgroup per 32x32 tile: grid x = 128 * tiles_n, grid y =
tiles_m
    grid = (((N + 31) // 32) * 128, (M + 31) // 32, 1)
    return _simd(inputs=[a, b, M, N, K], template=[("T", a.dtype)], grid=grid, th
readgroup=(128, 1, 1),
                 output_shapes=[(M, N)], output_dtypes=[a.dtype])[0]
```

验收

```
./check.sh x3
.venv-mlx/bin/python bench_mlx.py x3
```

tests/test_x3_metal_matmul.py 26 行

```
import pytest

mx = pytest.importorskip("mlx.core", reason="X track needs MLX on an Apple-silico
n Mac")
from drill import load  # noqa: E402
```

```
SHAPES = [(1, 1, 1), (17, 33, 9), (32, 32, 32), (64, 64, 64), (100, 70, 130), (25
7, 129, 1025)]


@pytest.mark.parametrize("fn", ["matmul_tiled", "matmul_simd"])
@pytest.mark.parametrize("mnk", SHAPES)
def test_matmul(fn, mnk):
    M, N, K = mnk
    m = load("x3_metal_matmul")
    a, b = mx.random.normal((M, K)), mx.random.normal((K, N))
    out = getattr(m, fn)(a, b)
    assert out.shape == (M, N)
    assert mx.allclose(out, a @ b, atol=1e-3, rtol=1e-3).item()


def test_matmul_fp16():
    m = load("x3_metal_matmul")
    a, b = mx.random.normal((96, 64)).astype(mx.float16), mx.random.normal((64, 8
0)).astype(mx.float16)
    for fn in ("matmul_tiled", "matmul_simd"):
        out = getattr(m, fn)(a, b)
        assert out.dtype == mx.float16
        assert mx.allclose(out.astype(mx.float32), a.astype(mx.float32) @ b.astyp
e(mx.float32), atol=5e-2, rtol=1e-2).item()
```

提示

L1 **先完成 tiled 版本** 方向（轻度剧透） 收起

它和 C4 的 `matmul_smem_kernel` 几乎逐行对应：`threadIdx` → `thread_position_in_threadgroup.xy`，`blockIdx` → `threadgroup_position_in_grid.xy`，`__syncthreads()` → `threadgroup_barrier(mem_flags::mem_threadgroup)`。

L2 simdgroup 版本的地址 拆解与常见坑（中度剧透） 收起

threadgroup 内存用一维数组：`As[32 * 16]`（行跨度 BK = 16），`Bs[16 * 32]`（行跨度 BN = 32）。simdgroup `sg` 负责的四分之一：`sg_r = sg / 2`，`sg_c = sg % 2`。

```
simdgroup_load(a0, As + (sg_r * 16 + 0) * 16 + kk, 16);   // A 的两个 8 行块
simdgroup_load(a1, As + (sg_r * 16 + 8) * 16 + kk, 16);
simdgroup_load(b0, Bs + kk * 32 + sg_c * 16 + 0, 32);     // B 的两个 8 列块
simdgroup_load(b1, Bs + kk * 32 + sg_c * 16 + 8, 32);
```

L3 主循环 几乎是答案（重度剧透） 收起

```
for (uint k0 = 0; k0 < K; k0 += 16) {
    for (uint e = tid; e < 512; e += 128) { /* 加载 As[e] 和 Bs[e]，越界填 0 */ }
    threadgroup_barrier(mem_flags::mem_threadgroup);
    for (uint kk = 0; kk < 16; kk += 8) {
        /* 4 次 simdgroup_load */
        simdgroup_multiply_accumulate(c00, a0, b0, c00); simdgroup_multiply_acc
        simdgroup_multiply_accumulate(c10, a1, b0, c10); simdgroup_multiply_acc
    }
    threadgroup_barrier(mem_flags::mem_threadgroup);
}
/* 4 次 simdgroup_store 到 Cs，barrier，按 mask 写回 C */
```

参考解

解 **solutions/x3_metal_matmul.py** 先自己写完、跑过测试再看 收起

solutions/x3_metal_matmul.py 105 行

```
"""X3 · Matmul in Metal: threadgroup-memory tiling, then simdgroup_matrix (8x
8 hardware MMA) — reference solution.

C[M, N] = A[M, K] @ B[K, N], row-major, fp32 accumulation.

matmul_tiled : C4's v1 on Apple GPUs. 16x16 threadgroup computes a 16x16 tile
of C; A and B tiles go
               through threadgroup memory, every value is reused 16 times.
matmul_simd  : 128 threads = 4 simdgroups compute a 32x32 tile; each simdgrou
p owns a 16x16 quarter
               as 2x2 simdgroup_float8x8 accumulators and uses simdgroup_mult
iply_accumulate,
               the Apple-GPU matrix instruction (what MLX's own "steel" GEMM
kernels are built on).
"""
import mlx.core as mx

_TILED = """
    const uint TS = 16;
    uint2 t = thread_position_in_threadgroup.xy;
    uint2 g = threadgroup_position_in_grid.xy;
    uint row = g.y * TS + t.y;
    uint col = g.x * TS + t.x;
    threadgroup float As[16][16];
    threadgroup float Bs[16][16];
    float acc = 0.0f;
    for (uint k0 = 0; k0 < K; k0 += TS) {
        As[t.y][t.x] = (row < M && k0 + t.x < K) ? float(A[row * K + k0 + t.
x]) : 0.0f;   // zero-fill edges
        Bs[t.y][t.x] = (k0 + t.y < K && col < N) ? float(B[(k0 + t.y) * N + c
ol]) : 0.0f;
        threadgroup_barrier(mem_flags::mem_threadgroup);
        for (uint k = 0; k < TS; ++k) acc += As[t.y][k] * Bs[k][t.x];
        threadgroup_barrier(mem_flags::mem_threadgroup);
    }
    if (row < M && col < N) C[row * N + col] = T(acc);
"""

_SIMD = """
    // Tile: BM = BN = 32, BK = 16. 128 threads = 4 simdgroups arranged 2 x
2.
    const uint BM = 32, BN = 32, BK = 16;
```

```
    uint tid = thread_index_in_threadgroup;
    uint sg = simdgroup_index_in_threadgroup;
    uint sg_r = sg / 2, sg_c = sg % 2;                // this simdgroup's 16
x16 quarter
    uint2 g = threadgroup_position_in_grid.xy;
    uint row0 = g.y * BM, col0 = g.x * BN;

    threadgroup float As[32 * 16];                    // [BM][BK]
    threadgroup float Bs[16 * 32];                    // [BK][BN]
    threadgroup float Cs[32 * 32];                    // staging for the mas
ked store

    simdgroup_float8x8 c00 = make_filled_simdgroup_matrix<float, 8, 8>(0.0f);
    simdgroup_float8x8 c01 = make_filled_simdgroup_matrix<float, 8, 8>(0.0f);
    simdgroup_float8x8 c10 = make_filled_simdgroup_matrix<float, 8, 8>(0.0f);
    simdgroup_float8x8 c11 = make_filled_simdgroup_matrix<float, 8, 8>(0.0f);
    simdgroup_float8x8 a0, a1, b0, b1;

    for (uint k0 = 0; k0 < K; k0 += BK) {
        for (uint e = tid; e < BM * BK; e += 128) {    // 512 values each, 4
per thread
            uint r = e / BK, k = e % BK;
            As[e] = (row0 + r < M && k0 + k < K) ? float(A[(row0 + r) * K + k
0 + k]) : 0.0f;
            uint kb = e / BN, c = e % BN;
            Bs[e] = (k0 + kb < K && col0 + c < N) ? float(B[(k0 + kb) * N + c
ol0 + c]) : 0.0f;
        }
        threadgroup_barrier(mem_flags::mem_threadgroup);
        for (uint kk = 0; kk < BK; kk += 8) {
            // simdgroup_load(matrix, pointer to the 8x8 block's first elemen
t, row stride in elements)
            simdgroup_load(a0, As + (sg_r * 16 + 0) * BK + kk, BK);
            simdgroup_load(a1, As + (sg_r * 16 + 8) * BK + kk, BK);
            simdgroup_load(b0, Bs + kk * BN + sg_c * 16 + 0, BN);
            simdgroup_load(b1, Bs + kk * BN + sg_c * 16 + 8, BN);
            simdgroup_multiply_accumulate(c00, a0, b0, c00);   // c += a @ b,
8x8x8
            simdgroup_multiply_accumulate(c01, a0, b1, c01);
            simdgroup_multiply_accumulate(c10, a1, b0, c10);
            simdgroup_multiply_accumulate(c11, a1, b1, c11);
        }
        threadgroup_barrier(mem_flags::mem_threadgroup);
    }
    simdgroup_store(c00, Cs + (sg_r * 16 + 0) * BN + sg_c * 16 + 0, BN);
    simdgroup_store(c01, Cs + (sg_r * 16 + 0) * BN + sg_c * 16 + 8, BN);
    simdgroup_store(c10, Cs + (sg_r * 16 + 8) * BN + sg_c * 16 + 0, BN);
    simdgroup_store(c11, Cs + (sg_r * 16 + 8) * BN + sg_c * 16 + 8, BN);
    threadgroup_barrier(mem_flags::mem_threadgroup);
    for (uint e = tid; e < BM * BN; e += 128) {        // masked write-back f
```

```
or partial edge tiles
        uint r = e / BN, c = e % BN;
        if (row0 + r < M && col0 + c < N) C[(row0 + r) * N + col0 + c] = T(Cs
[e]);
    }
"""

_tiled = mx.fast.metal_kernel(name="matmul_tiled", input_names=["A", "B",
"M", "N", "K"], output_names=["C"], source=_TILED)
_simd = mx.fast.metal_kernel(name="matmul_simd", input_names=["A", "B", "M",
"N", "K"], output_names=["C"], source=_SIMD,
                             header="#include <metal_simdgroup_matrix>\n")


def _check(a, b):
    assert a.ndim == 2 and b.ndim == 2 and a.shape[1] == b.shape[0] and a.dty
pe == b.dtype
    return a.shape[0], b.shape[1], a.shape[1]


def matmul_tiled(a: mx.array, b: mx.array) -> mx.array:
    M, N, K = _check(a, b)
    grid = (((N + 15) // 16) * 16, ((M + 15) // 16) * 16, 1)       # threads,
not threadgroups
    return _tiled(inputs=[a, b, M, N, K], template=[("T", a.dtype)], grid=gri
d, threadgroup=(16, 16, 1),
                  output_shapes=[(M, N)], output_dtypes=[a.dtype])[0]


def matmul_simd(a: mx.array, b: mx.array) -> mx.array:
    M, N, K = _check(a, b)
    # one 128-thread threadgroup per 32x32 tile: grid x = 128 * tiles_n, grid
y = tiles_m
    grid = (((N + 31) // 32) * 128, (M + 31) // 32, 1)
    return _simd(inputs=[a, b, M, N, K], template=[("T", a.dtype)], grid=gri
d, threadgroup=(128, 1, 1),
                 output_shapes=[(M, N)], output_dtypes=[a.dtype])[0]
```

测完之后

Apple M2 实测（fp32，单位 TFLOP/s）：

n	MLX steel GEMM	你的 tiled	你的 simdgroup_matrix
512	0.80	0.29	0.57
1024	2.19	0.37	0.79

n	MLX steel GEMM	你的 tiled	你的 simdgroup_matrix
2048	**2.62**	0.39	0.79

- simdgroup_matrix 比纯 threadgroup tiling 快 2 倍左右，但还只有 MLX 的 30%。差距在哪：MLX 的 steel GEMM 每个 simdgroup 持有更多累加器（更大的寄存器 tile，复用度更高），用双缓冲把下一块的加载和当前块的计算重叠，并且在 threadgroup 内存的布局上做了避免 bank conflict 的处理。这和 C4 → T4 的经验完全一致。
- 实验：把每个 simdgroup 的累加器从 2×2 增加到 4×2（每个 threadgroup 计算 64×32），观察寄存器压力和速度的变化。
- 读源码：`mlx/backend/metal/kernels/steel/gemm/` 中的 `mma.h` 和 `loader.h`，对照你自己的实现。
- 量化 matmul（`mx.quantized_matmul`，4-bit 分组量化）是 Mac 上跑 LLM 的主力：decode 时它是 memory-bound 的 `qmv`，prefill 时是 `qmm`。读一读 `quantized.h`，看反量化是怎么和乘加融合在一起的。

▼ 面试追问：Mac 上跑 LLM，prefill 和 decode 的瓶颈分别是什么？

prefill 是 compute-bound 的，受 GPU 的矩阵吞吐限制（M2 的 fp16 大约只有几 TFLOP/s，比数据中心 GPU 低两个数量级），所以长 prompt 的首 token 延迟较高。decode 是 memory-bound 的，每个 token 都要读一遍全部权重，速度约等于“内存带宽 ÷ 模型字节数”：4-bit 量化的 7B 模型约 3.5–4 GB，在 100 GB/s 的 M2 上理论上限约为每秒 25 个 token，在 800 GB/s 的 Ultra 上约为每秒 200 个。所以 Mac 上最有效的优化是量化（减少字节数）和投机解码（读一次权重，验证多个 token）。

面试

面试题库

按主题整理的高频面试题。每题先自己口头回答一遍（限时 2 分钟），再展开参考答案；括号里是本套题中对应的练习。

1. 硬件与性能模型
2. CUDA 与 Triton
3. Attention
4. FlashInfer 与 serving
5. MoE 与低精度
6. 不同模型、不同硬件

硬件与性能模型

▼ 1. 怎么判断一个 kernel 是 compute-bound 还是 memory-bound? （Part 0）

算算术强度 AI = FLOPs / 必须读写的 HBM 字节数，和硬件拐点（峰值算力 ÷ 峰值带宽，H100 FP16 约 295 FLOP/B，RTX 5090 约 117 FLOP/B）比较。低于拐点是 memory-bound，用有效带宽（GB/s）衡量；高于拐点是 compute-bound，用 TFLOP/s 衡量。再用 ncu 验证：看 Memory Throughput 和 SM Throughput 哪个接近 100%，以及 warp 在等什么（Long Scoreboard 表示在等访存）。

▼ 2. 什么是合并访存? 什么是 bank conflict? 各自怎么解决? （C3）

一个 warp 的 32 次访存会被合并成尽量少的 32 字节 sector：相邻线程访问相邻地址时效率最高，跨步访问会让事务数成倍增加。解决办法是调整线程到数据的映射，或者通过 shared memory 中转（转置）。shared memory 有 32 个 bank，同一 warp 访问同一 bank 的不同地址时会串行化；解决办法是加 padding（`[32][33]`）或对地址做 swizzle（异或），访问同一地址则是广播。

▼ 3. occupancy 越高越好吗?

不一定。occupancy 高可以隐藏延迟，但 GEMM、FlashAttention 这类 kernel 往往故意用很多寄存器和 shared memory 来提高数据复用（每个线程算更大的 tile），occupancy 只有 1–2 个 block / SM，靠指令级并行和软件流水线（cp.async / TMA 预取）来隐藏延迟。判断标准是实际吞吐，不是 occupancy 数字。

▼ 4. H100、B200、RTX 5090 在写 kernel 时最大的区别? （全景地图）

H100（sm_90a）引入 wgmma（warpgroup 级异步 MMA）、TMA、线程块集群，最优 kernel 普遍使用 warp specialization（生产者 warp 用 TMA 搬数据，消费者 warp 做 wgmma）。B200（sm_100）换成 tcgen05 MMA，累加器放在专门的 TMEM 里，支持 2-CTA MMA 和硬件 block-scaled FP4 / FP8。RTX 5090（sm_120）虽然也叫 Blackwell，但没有 tcgen05 / TMEM / wgmma，仍是 `mma.sync`（扩展了 FP4 / FP6 / FP8），所以为 sm_100 写的 FA4、DeepGEMM 不能直接运行。

CUDA 与 Triton

▼ 5. 讲一下 reduction 的优化过程。（C2）

从 shared memory 树形归约开始（每轮 `__syncthreads`）；用 warp shuffle 取代 warp 内的那几轮（不需要同步、不经过 shared memory）；每个线程先用 grid-stride loop 在寄存器里累加多个元素，减少 block 数；跨 block 用一次 atomicAdd，或者用两趟 kernel 保证确定性。Triton 里一个 `tl.sum` 就会被编译成线程内累加、shuffle、shared memory 这三层。

▼ 6. GEMM 从 naive 到 cuBLAS，每一步优化了什么？（C4、T4）

naive 每次乘加都从 HBM 读两个数。shared memory tiling 让每个数在 block 内复用 tile 宽度次；register tiling 让每个线程计算一个小块，shared memory 的读取再减少 TM/TN 倍；然后是向量化加载、双缓冲 / 多级流水线（cp.async、TMA）、Tensor Core（mma / wgmma / tcgen05）、swizzle 消除 bank conflict、grouped ordering 提高 L2 命中，以及按形状 autotune。在 RTX 5090 上，本套题的 Triton GEMM（T4）从 1024^3 开始就和 cuBLAS 持平。

▼ 7. Triton 和 CUDA 怎么选？Triton 的局限是什么？

Triton 让你在 block 层面编程：shared memory、合并访存、流水线、Tensor Core 指令都由编译器决定，几十行就能写出接近 cuBLAS 的 GEMM，或者带宽跑满的融合算子，非常适合新算子和融合。局限在于对硬件的控制力：warp specialization、复杂的 TMA / 集群用法、精确的寄存器分配、非常规的数据布局，在 Hopper / Blackwell 上很难让编译器自动做到最好。这时就用 CUDA / CUTLASS / CuTe DSL，或者 Triton 的底层方言 Gluon（显式控制 layout 和数据搬运）。

▼ 8. Triton 的 num_warps 和 num_stages 分别控制什么？（T4）

num_warps 是每个 program（CTA）的 warp 数，决定一个块级张量被分给多少线程，以及每个线程持有多少寄存器。num_stages 是软件流水线深度：K 循环里的 load 会被提前几轮发出，经 shared memory 缓冲，计算和访存因此重叠。stage 越多预取越深，但每个 stage 都要占一份 shared memory，可能降低 occupancy，甚至超出容量。最优值依赖形状和硬件，所以用 autotune 来选。

Attention

▼ 9. FlashAttention 为什么快？它的核心技巧？（T5）

它没有减少 FLOPs，减少的是 HBM 读写：通过 tiling 和 online softmax（维护行最大值 m 和分母 l，新块到来时用 exp(m_old − m_new) 重新缩放旧的累加值），S×S 的分数矩阵从不写回 HBM，IO 从 $O(S^2)$ 降到 $O(S \cdot D)$。FA2 改进了并行划分（沿序列维度并行、减少非矩阵运算）；FA3 在 Hopper 上用 wgmma、TMA、warp specialization 和 FP8；FA4 在 Blackwell 上用 tcgen05 / TMEM，并用 CuTe DSL 实现。

▼ 10. decode attention 和 prefill attention 的瓶颈有什么不同？（L2、T5）

prefill 时 Q 有很多行，QK^T 是真正的矩阵乘，compute-bound，比拼 Tensor Core 利用率。decode 时每个序列只有一个 query，每个 KV token 只做约 4D 次运算却要读 2D 个数，是 memory-bound 的，时间约等于 KV cache 字节数 / 带宽。所以 decode 的优化方向是少读、快读：paged 布局、GQA 打包（每个 KV head 只读一次）、KV 量化（FP8 / FP4）、split-KV（让足够多的 SM 一起读）。

▼ 11. 什么是 split-KV / Flash-Decoding？什么时候用？（L3、F1）

batch 小、上下文长时，按 (序列, KV head) 划分的 program 数远少于 SM 数。split-KV 沿上下文维度再切成几段并行计算，每段输出 (o, lse)，最后用 LSE 加权精确合并。RTX 5090 上 B = 1、ctx = 32K 时，分 64 段快了 19 倍。batch 大时分段反而更慢（额外的中间结果写回和合并 kernel 成为纯开销），所以分几段要按形状决定：这正是 FlashInfer `plan()` 的工作。

▼ 12. 只给每段的输出 o_s，不给 lse，能合并吗？FlashInfer 的 LSE 有什么特别之处？（F2）

不能：合并需要每段 softmax 分母的相对大小，只有 o_s 时这个信息已经丢失了。合并公式是 w_s = exp(lse_s − max)，o = Σ w_s o_s / Σ w_s。FlashInfer 的 LSE 以 2 为底（$\log_2 \Sigma \exp$），kernel 内部全部使用 exp2 / log2，和自然对数版本混用会算出错误的权重。

▼ 13. GQA、MQA、MLA 对 KV cache 和 decode kernel 有什么影响？（L2、全景地图）

GQA / MQA 让多个 query head 共享一组 KV，KV cache 缩小 group 倍；decode kernel 让一个 program 处理同一 KV head 下的全部 query head，KV 只读一次，并可以把这些 query 组成矩阵、用上 Tensor Core。MLA（DeepSeek）把 KV 压缩成低秩 latent，cache 更小；decode 时把上投影吸收进 Q，直接在 latent 上做 attention（head_dim 512 + 64 的 RoPE 部分），算术强度更高，需要 FlashMLA、FlashInfer MLA 这样的专用 kernel。

▼ 14. PagedAttention 解决了什么问题？代价是什么？（L1、F1）

它把 KV cache 按固定大小的 page 分配，消除了按最大长度预留带来的碎片和浪费，让 KV 可以动态增长、在序列之间共享前缀（copy-on-write），并发量因此提高数倍。代价是 attention kernel 每读一个 page 都要先查一次页表（间接寻址），物理地址不连续。vLLM 用二维 block table，FlashInfer 用 CSR（indptr、indices、last_page_len），后者不需要按最长序列补齐。

FlashInfer 与 serving

▼ 15. FlashInfer 的 plan / run 模式是什么？为什么这样设计？（F1）

plan() 在 CPU 上根据这一批序列的长度做调度：是否分段、分几段、每个 CTA 处理哪些 page、中间结果放在 workspace 的哪里，然后把元数据拷到 GPU。run() 只启动 kernel。一个 batch 的形状确定后 plan 只做一次，所有层复用；run() 形状固定、没有 CPU 同步，可以被 CUDA graph 捕获。

▼ 16. 为什么采样 kernel 不排序？（F4）

top-k / top-p 只需要一个阈值。FlashInfer 反复对“大于 pivot 的元素”做 (质量, 个数) 的归约来移动 pivot，每轮一次全词表扫描，几轮就收敛，还能按请求使用不同的参数，并直接在同一个 kernel 里完成采样。RTX 5090 上 64 个请求、15 万词表时，`torch.sort` 本身就要 1 ms，FlashInfer 的 top-k renorm 只要 88 us。

▼ 17. decode 阶段为什么要融合 RMSNorm、残差加、SwiGLU 这些小算子？（F3）

decode 时每个 kernel 的工作量都很小，几微秒的固定开销（launch、调度、一次读写）就占了大头。一个 decoder layer 有十来个这样的小算子，不融合时它们的总开销可以和 GEMM 相当。融合（fused_add_rmsnorm、silu_and_mul、norm + 量化、RoPE + 写 KV cache）、CUDA graph 和 PDL 都是在减少这部分开销。

▼ 18. 你会怎么给 FlashInfer 贡献一个新算子？（F5）

kernel 写在 include/（只用裸指针，不 include Torch），launcher 写在 csrc/，加 TVM-FFI 绑定，在 flashinfer/jit/ 中写 JIT 生成器，再写 Python API（@functools.cache 缓存模块）、测试、AOT 注册、导出和 trace 模板。性能 PR 要给出改动前后的绝对数字，并写明 GPU 和问题规模；保留默认的 PR 模板；CI 用 `@flashinfer-bot run tests/<路径>` 只跑相关测试。消费级 GPU 的 kernel 走 experimental 路线。

MoE 与低精度

▼ 19. fused MoE kernel 是怎么工作的？（M1、M2）

先路由（top-k 专家及其权重），再把 (token, slot) 对按专家排序，并把每个专家的那一组补齐到 BLOCK_M 的整数倍（moe_align_block_size）。然后一次 grouped GEMM 的 launch：每个 M 块属于一个专家，A 的行通过 sorted_token_ids 间接读取，B 用这个专家的权重；路由权重乘进 epilogue；最后把 k 个专家的结果求和。prefill 时是 compute-bound 的，decode 时是读权重的 memory-bound。

▼ 20. DeepSeek-V3 的 FP8 block scaling 是怎么回事？为什么不用 per-tensor？（M3）

FP8 e4m3 的动态范围很小，per-tensor 的 scale 会让离群值把其他数压没。DeepSeek-V3 对激活每 1×128 取一个 scale，对权重每 128×128 取一个；每个 128 宽的 K 块做一次 FP8 Tensor Core 矩阵乘，再乘上这一块的 sa·sb，fp32 累加。粒度和 K 块一致，所以几乎没有额外开销。本套题在 RTX 5090 上的 Triton 实现达到 bf16 cuBLAS 的 1.7 倍。Blackwell 的 MXFP8 / NVFP4 把这种块缩放做进了硬件指令。

▼ 21. DeepGEMM 的 contiguous 和 masked 两种 grouped 布局分别给什么场景用?

contiguous：各专家的 token 连续排列（按专家补齐到对齐边界），适合 prefill 和训练，token 数在 CPU 上已知。masked：为每个专家预留固定的最大行数，用一个张量记录实际行数，kernel 内部屏蔽多出的部分，形状固定，适合 decode 阶段配合 CUDA graph（每步每个专家的 token 数在 GPU 上才知道）。

不同模型、不同硬件

▼ 22. 扩散模型（视频 DiT）的推理瓶颈和 LLM 有什么不同？（全景地图）

没有 KV cache 和 decode，每个去噪步都是对全部 token 的一次 compute-bound 前向；视频的序列长度可达数万到数十万，attention 往往占一半以上的时间。有效手段是最好的 FlashAttention 代际、量化 attention（SageAttention：INT8 / FP8，SageAttention3 在 RTX 5090 上用 FP4）、利用时空局部性的稀疏 attention、融合 adaLN，以及序列并行（Ulysses / ring）。端到端收益要算全：步数蒸馏和特征缓存的影响通常更大。

▼ 23. VLA 模型（比如 $\pi 0$）在机器人上部署，kernel 层面最该优化什么？

batch = 1、对延迟有硬性要求，瓶颈往往不是某一个大 kernel，而是上百个小 kernel 的 launch 开销和 CPU 调度（包括 flow matching 的多步去噪循环）。最有效的组合是：用 Triton 融合小算子、预分配静态 buffer、把整条前向（含去噪循环）捕获进一个 CUDA graph；再根据端侧硬件（Jetson Thor / Orin）用 TensorRT 和低比特量化。

▼ 24. 在 Mac 上用 MLX 跑 LLM，性能主要由什么决定？（X1–X3）

decode 是 memory-bound 的：每个 token 读一遍权重，tok/s ≈ 带宽 ÷ 模型字节数，所以 4-bit 分组量化（MLX 的 qmv / qmm）是最有效的手段。prefill 受 GPU 矩阵吞吐限制（simdgroup_matrix 8×8；M5 起还有 neural accelerator）。统一内存免去了拷贝，但带宽只有数据中心 GPU 的几分之一到几十分之一。

▼ 25. 同一个模型在 H100 和 AMD MI300X 上，kernel 栈有什么不同?

NVIDIA 上：FlashAttention-3 / FlashInfer、cuBLAS / CUTLASS / DeepGEMM、Triton。AMD 上：AITER（Triton + Composable Kernel + 手写汇编，提供 attention、fused MoE、MLA、GEMM），配合 vLLM / SGLang 的 ROCm 后端（如 `VLLM_ROCM_USE_AITER=1`）；Triton 的 ROCm 后端让同一份 Triton kernel 也能运行，但最优配置（tile 大小、wave 数）不同。MI300X 显存大（192 GB），单卡能放下更大的模型或更多 KV。

附录

附录：下一步与参考资料

做完这些题之后：继续深入的方向、源码阅读路线和参考资料。

继续深入的方向

方向	建议的下一步
Tensor Core（CUDA）	用 `nvcuda::wmma` 改写 C4，在 FP16 下对比 cuBLAS；再读 CUTLASS 的 `cute` 入门例子（MMA atom、TiledMMA）
CuTe DSL	在 RTX 5090 上跑通 CUTLASS `examples/python/CuTeDSL/` 中的 elementwise 和 GEMM 示例，然后读 FlashInfer 的 `flashinfer/cute_dsl/` 中的一个 kernel
Hopper / Blackwell 专用指令	有 H100 / B200 时：读 FlashAttention-3（`hopper/`）和 FlashAttention-4（`flash_attn/cute/`）、DeepGEMM 的主循环，理解 TMA + wgmma / tcgen05 + warp specialization 的分工
Gluon	Triton 官方的 Gluon tutorial：从 layout 和 shared memory 开始，最后一课是 warp-specialized GEMM。把 T4 改写成 Gluon，对比生成的 `ttgir`
FlashAttention backward	在 T5 中保存 lse，按 FlashAttention-2 论文的 Algorithm 2 实现 dQ / dK / dV
线性 attention	读 flash-linear-attention 中 GDN 的 chunked kernel，再看 FlashInfer 0.7 的 `gdn_prefill` / `gdn_fused_decode_step`
通信	DeepEP 的 dispatch / combine，NCCL 的 all-reduce 算法，以及 FlashInfer `comm/` 中把 all-reduce 和 RMSNorm 融合在一起的 kernel
MLX	读 `mlx/backend/metal/kernels/steel/gemm/` 和 `sdpa_vector.h`；给 MLX 写一个它还没有的融合算子（例如带量化的 RMSNorm），用 `mx.fast.metal_kernel` 先实现原型

源码阅读路线（按本套题的顺序）

所有链接都固定在某个 commit 上，行号不会因为上游改动而漂移；想看最新版本，在 GitHub 页面按 `y` 切换，或把 URL 中的 commit 换成 `main`。

题目	FlashInfer	其他
T2、T3、F3	`flashinfer › RMSNormKernel` · `flashinfer › activation.cuh` · `flashinfer › norm.py`	`vllm › layernorm_kernels.cu` · `triton › 02-fused-softmax.py` · `triton › 05-layer-norm.py`
T5	`flashinfer › prefill.cuh`	`triton › 06-fused-attention.py` `fa › FA3` · `fa › FA4`
L1、F1	`flashinfer › page.cuh` · `flashinfer › append_paged_kv_cache`	`vllm › reshape_and_cache_flash_k`
L2、L3、F1	`flashinfer › decode.cuh` · `flashinfer › scheduler.cuh` · `flashinfer › BatchDecodeWithPagedKVCacheWrapper`	`vllm › triton_decode_attention.p` `sglang › decode_attention.py`
F2	`flashinfer › cascade.cuh` · `flashinfer › merge_state`	
F4	`flashinfer › TopPRenormProb` · `flashinfer › topk.cuh` · `flashinfer › top_p_renorm_probs`	`vllm › topk_topp_sampler.py`
M1、M2	`flashinfer › fused_moe/`	`vllm › fused_moe_kernel` · `vllm › moe_align_sum_kernels.cu`
M3	`flashinfer › deep_gemm.py`	`deepgemm › sm90_fp8_gemm_1d2d.cu` `vllm › w8a8_triton_block_scaled_`
G1、G2	`flashinfer › cute_dsl/`	`triton › 02-layouts.py` · `cutlass › 02_layout_algebra.md` `cutlass › composition`
X1—X3	—	`mlx › softmax_looped` · `mlx › mm` `mlx › quantized.h` · `mlx › sdpa_vector.h`

参考资料

资料	用在哪
CUDA C++ Programming Guide：Programming Model、Memory Hierarchy、Performance Guidelines	Part 0、C 系列
Mark Harris, “Optimizing Parallel Reduction in CUDA”	C2
Simon Boehm, “How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance”	C4
Triton 官方 tutorials（Vector Addition、Fused Softmax、Layer Norm、Matrix Multiplication、Fused Attention）、Triton Puzzles	T 系列
FlashAttention 1 / 2 / 3 论文，FlashAttention-4（arXiv 2603.05451，MLSys 2026）	T5、全景地图
Kwon et al., PagedAttention（vLLM，SOSP 2023）；Flash-Decoding（2023 博客）	L 系列
FlashInfer 论文（MLSys 2025）、文档、仓库中的 `CONTRIBUTING.md` 与 `CLAUDE.md`	F 系列
DeepSeek-V3 技术报告（FP8 训练与推理）；DeepGEMM、DeepEP 仓库	M 系列
CuTe 文档（CUTLASS 仓库 `media/docs/cpp/cute/`）、CuTe DSL 文档	G2
MLX 文档 “Custom Metal Kernels”；Metal Shading Language Specification	X 系列
SageAttention 1–3 论文；SVDQuant；xDiT	全景地图：扩散
PMPP（Programming Massively Parallel Processors）	随用随查

每题完成后的记录模板

```
## F2 · merge_state
- 环境：RTX 5090 / CUDA 13.0 / torch 2.13 / triton 3.7 / flashinfer 0.7.2
- 正确性：与 flashinfer.merge_state 对拍，max abs err = …
- 性能：S=8192, H=32, D=128 fp16：yours … us（… GB/s），flashinfer … us
- 瓶颈：memory-bound；ncu Memory Throughput …%
- 实验：每个 program 处理 2 个 head：…
- 结论 / 下一步：…
```