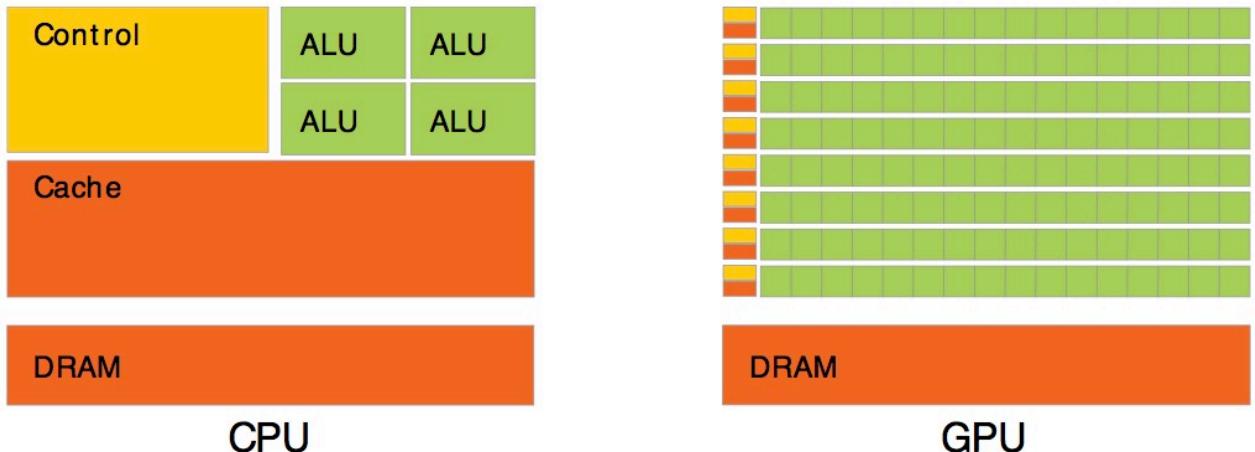


Parallel Computing

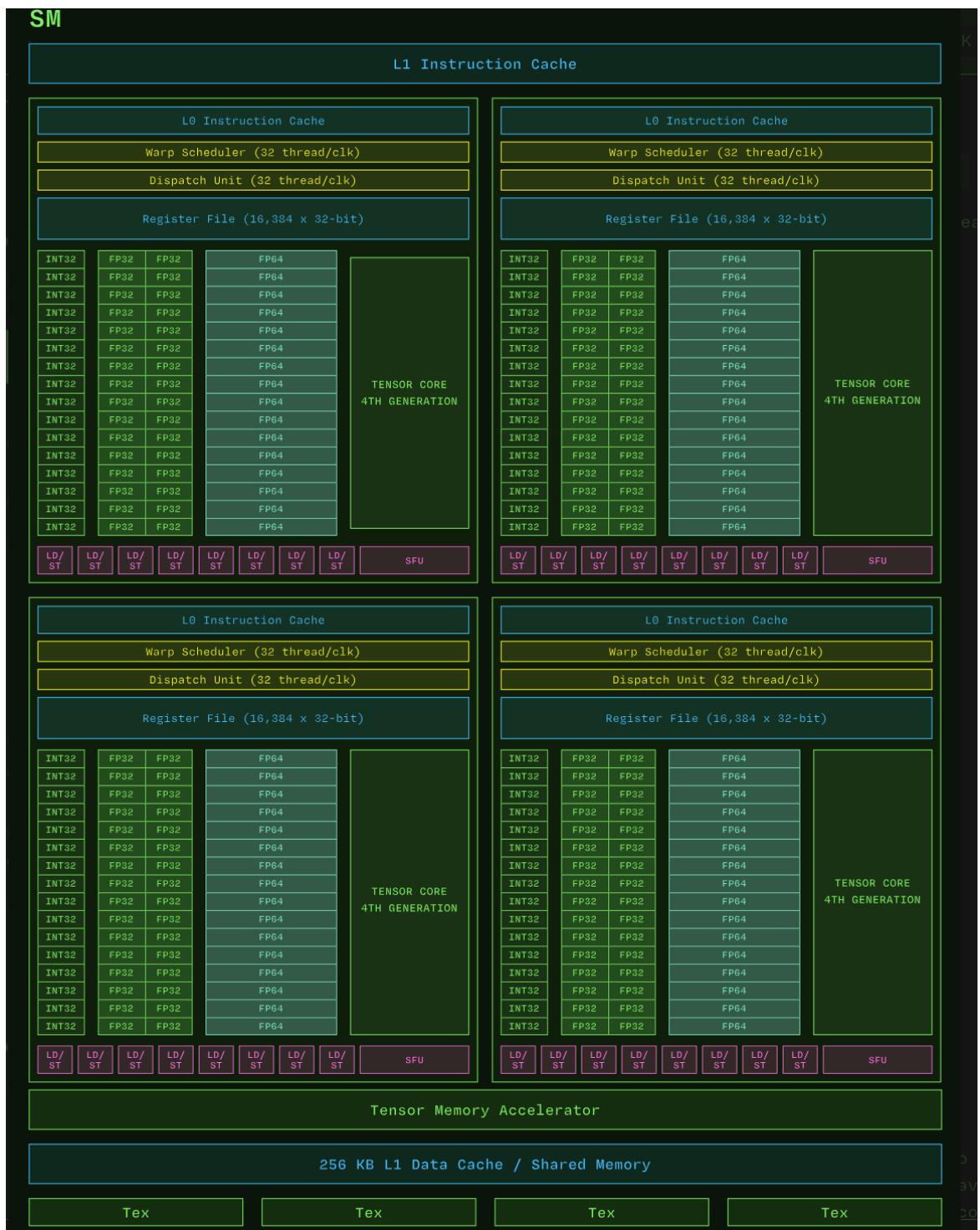
GPU结构

- GPU结构--NVIDIA



- 抽象结构: Grid, CTA(Block), WarpGroup, Warp, Thread
 - Grid: 软件编程单元--代表了你想要解决的整个问题
 - 作用: 定义一个完整的并行任务--比如处理一个1024×1024的图像
 - 特性: Grid之间的Blocks独立执行, 没有执行顺序的保证
 - CTA(Block): 程序员在编写CUDA Kernel时直接定义和操作的主要结构
 - 作用: 将一个大问题分解成可以并行处理的块, 一个block对应一个SM
 - 特性: 同一个Block之间的thread可以使用共享内存 (shared memory), 不同block之间的thread只能通过全局内存 (global memory) 进行通信
 - WarpGroup: 由 (4个) 相邻的Warps组成, 是为了增强线程间通信而引入的概念。
 - 作用: 支持TMA, 和异步拷贝等高级内存操作的基础单元
 - 特性: 允许不同warp之间进行数据交换和同步 (不是共享内存, 比共享内存延迟更低)
 - Warp: 最小的硬件调度和执行单元。Warp通常由很多个 (可能是32个) thread组成, 作用: Warp组织这些thread在同一时钟周期执行相同的指令 (SIMT), 也就是GPU在调度的时候, 是以Warp为单位进行调度的
 - 特性: SIMT--单指令多线程
 - Thread: (线程) :最小的执行单元
 - 作用: 每个thread处理数据的一部分, 比如处理一个巨大向量和, 每个thread负责一对元素的加法
 - 特性: 拥有自己私有的寄存器(Registers)和本地内存(Local Memory)。
- 硬件结构: SM, Cuda core, Tensor core, Warp Scheduler
 - SM (Streaming Multiprocessor) :相当于CPU的核心, 既执行并行计算, 也可用于将计算的状态存储在寄存器中
 - SM作用: 可以并发执行数百个线程

SM



SM (Streaming Multiprocessor)

- **SM** : 流式多处理器 , 核心组件包括CUDA核心、共享内存、寄存器等。SM包含许多为线程执行数学运算的Core , 是 NVIDIA 的核心。。主要包括 :

1. **CUDA Core** : 向量运行单元 (FP32-FPU、FP64-DPU、INT32-ALU) ;
2. **Tensor Core** : 张量运算单元 (FP16、BF16、INT8、INT4) ;
3. **Special Function Units** : 特殊函数单元 SFU (超越函数和数学函数 , e.g. 反平方根、正余弦等) ;
4. **Warp Scheduler** : 线程束调度器 (XX Thread / clock) ;
5. **Dispatch Unit** : 指令分发单元 (XX Thread / clock) ;
6. **Multi level Cache** : 多级缓存 (L0/L1 Instruction Cache、L1 Data Cache & Shared Memory) ;
7. **Register File** : 寄存器堆 ;
8. **Load/Store** : 访问存储单元LD/ST (负责数据处理) ;

- 运算单元:

- CUDA Cores--相当于高度流水线化的ALU
包含**FP32** (单精度) 和**INT32** (整数) 运算

- Tensor Cores
专为执行矩阵乘加运算
- 特殊功能单元-SFU
超越函数和数学函数, 比如正余弦, 平方根, 倒数等

- 调度-分发单元:

- Warp Scheduler: 决定下一个时钟周期哪些Warp可以执行
- Dispatch Unit: 将Warp Scheduler选中的Warp的指令实际发送 (分发) 到具体的执行单元。

- 内存单元: Register File, L0Cache, L1Cache/Shared Memory, Tex
- 加载/存储单元:

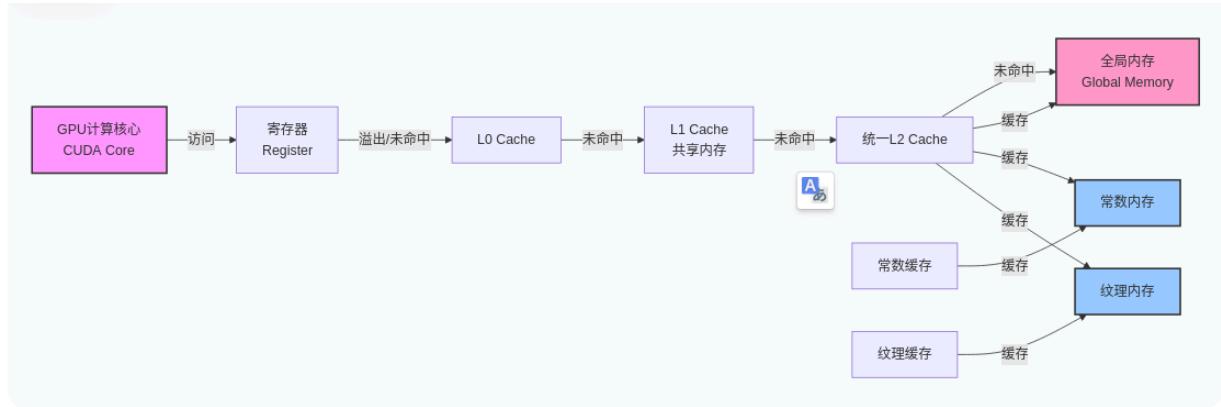
- LD/ST: load/store, 专门用来加载和存储的单元, 决定数据是否能及时喂到计算单元

- 内存层级: Global Mem, L2 Cache, Shared Mem, L1 Cache, Reg file

速度: Register>L0 Cache >L1 Cache = Share Memory >L2 Cache > Global Memory=Local Memory

Register > L0 Cache > Texture Cache > Constant Cache > L1 Cache = Shared Memory > L2 Cache > Constant Memory / Texture Memory > Global Memory = Local Memory

位置:



1. Global Memory (DRAM): 最慢, GPU的板载显存 (GPU片外), 全局享用, 用来GPU与CPU通信

- Texture/Surface Memory 纹理内存/表面内存: 快, 物理在全局内存上, 但有专门的只读缓存 (Texture Cache) 和硬件单元, 全局可读, 用来图形渲染设计, 也可以用于通用计算, 也可以简化图像处理代码
- Constant Memory: 快, 物理在全局内存上, 全局可读, 主机可写, 用于存储大量读取的常量参数

2. L2 Cache: 较快, GPU片内, SM片外, 全局享用, 用来作为Global Memory的cache

3. Share Memory: 非常快 (与L1Cache同级), 在SM内, 与L1 Cache一块物理内存, 同一个Block内的所有Threads共享,

4. L1 Cache: 非常快 (与SharedMemory同级), 在SM内, 与共享内存一块物理内存, 同一个Block内的所有Threads共享

L1Cache和ShareMemory的区别: L1是隐式的, ShareMemory是程序员可以显式调用的

5. L0 Cache: 速度快 (介于L1和Register之间), 在SM内, 用来进一步降低数据访问延迟

6. Register File: 速度最快, 在SM核心上, Thread私有, 每个thread对应多个寄存器, 用来存放变量等数据

如果太多, 会溢出到Local Memory (本地内存)

Local Memory: 非常慢 (与全局内存相同), 物理位置在全局内存中, Thread私有, 用来存放寄存器溢出的大型结构体, 大型数组等

- GPU和GPGPU区别

- 区别: GPU只专注于图形处理, GPGPU更加关注一些通用的计算
- GPGPU架构:

NVIDIA GPU架构发展

架构名称	Fermi	Kepler	Maxwell	Pascal	Volta	Turing	Ampere	Hopper
中文名字	费米	开普勒	麦克斯韦	帕斯卡	伏特	图灵	安培	赫柏
发布时间	2010	2012	2014	2016	2017	2018	2020	2022
核心参数	16个SM，每个SM包含32个CUDA Cores，一共512 CUDA Cores	15个SMX，每个SMX包括192个FP32+64个FP64 CUDA Cores	16个SM，每个SM包括4个处理块，每个处理块包括32个CUDA Cores +8个LD/ST Unit + 8 SFU	GP100有60个SM，每个SM包括32个FP64 +64 INT32+64 FP32+8个Tensor Cores	80个SM，每个SM包括64个FP32+64个INT32+32个DP Cores	102核心92个SM，SM重新设计，每个SM包含64个INT32+32+64个FP32+8个Tensor Cores	108个SM，每个SM包含64个FP32+64个INT32+32+64个FP32+8个Tensor Cores	132个SM，每个SM包含128个FP32+64个INT32+64个FP64+4个Tensor Cores
特点&优势	首个完整GPU计算架构，支持与共享存储结合的Cache层次GPU架构，支持ECC GPU架构	游戏性能大幅提升，首次支持GPU Direct技术	每组SM单元从192个减少到每组128个，每个SM单元拥有更多逻辑控制电路	NVLink第一代，双向互联带宽160 GB/s，P100拥有56个SM HBM	NVLink2.0，Tensor Cores第一代，支持AI运算	Tensor Core2.0，RT Core第一代	Tensor Core3.0，RT Core2.0，NVLink3.0，结构稀疏性矩阵MIG1.0	Tensor Core4.0，NVLink4.0，结构稀疏性矩阵MIG2.0
纳米制程	40/28nm 30亿晶体管	28nm 71亿晶体管	28nm 80亿晶体管	16nm 153亿晶体管	12nm 211亿晶体管	12nm 186亿晶体管	7nm 283亿晶体管	4nm 800亿晶体管
代表型号	Quadro 7000	K80 K40M	M5000 M4000 GTX 9XX系列	P100 P6000 TTX1080	V100 TiTan V	T4，2080Ti RTX 5000	A100 A30系列	H100

翻了将近三倍

■ Ampere：

[Nvidia Ampere架构深度解析--知乎](#)

A100

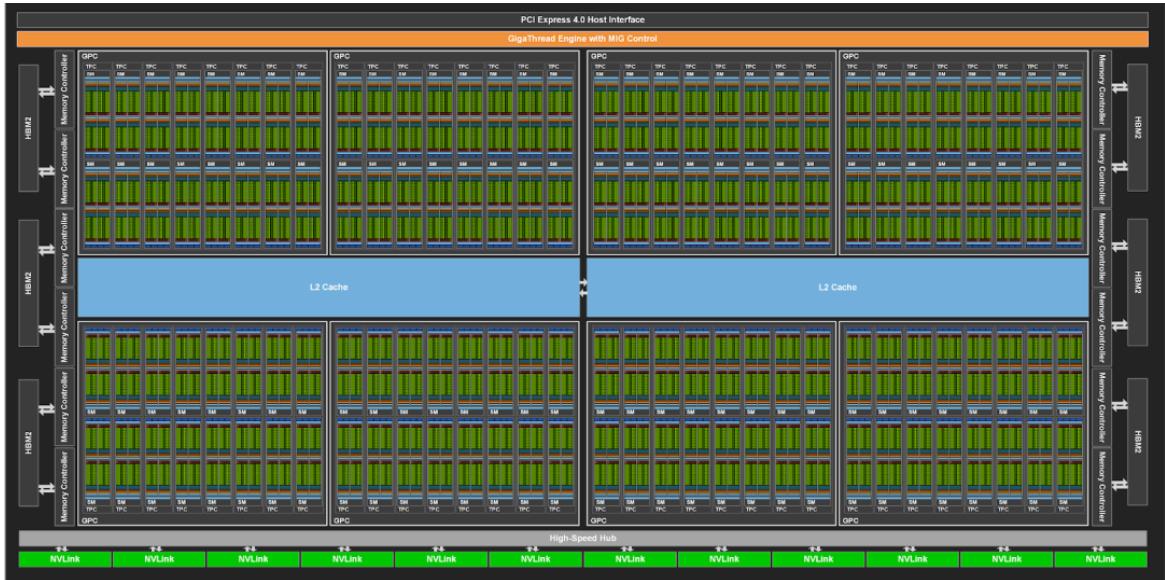
亮点3：多实例GPU：允许将一块A100GPU分成七个独立GPU（每个实例拥有独立的内存缓存和计算单元）

Ampere 安培架构

1. NVIDIA Ampere架构：超过540亿个晶体管，使其成为世界上最大的7纳米处理器；
2. Tensor Core3.0：新增TF32包括针对AI的扩展，可使FP32精度的AI性能提高20倍；
3. Multi-Instance GPU：多实例GPU，将单个A100GPU划分为多达七个独立GPU，为不同任务提供不同算力；
4. NVIDIA NVLink2.0：GPU间高速连接速度加倍，可在服务器中提供有效的性能扩展；
5. 结构稀疏性：利用了AI数学固有的稀疏特性来使性能提高一倍。

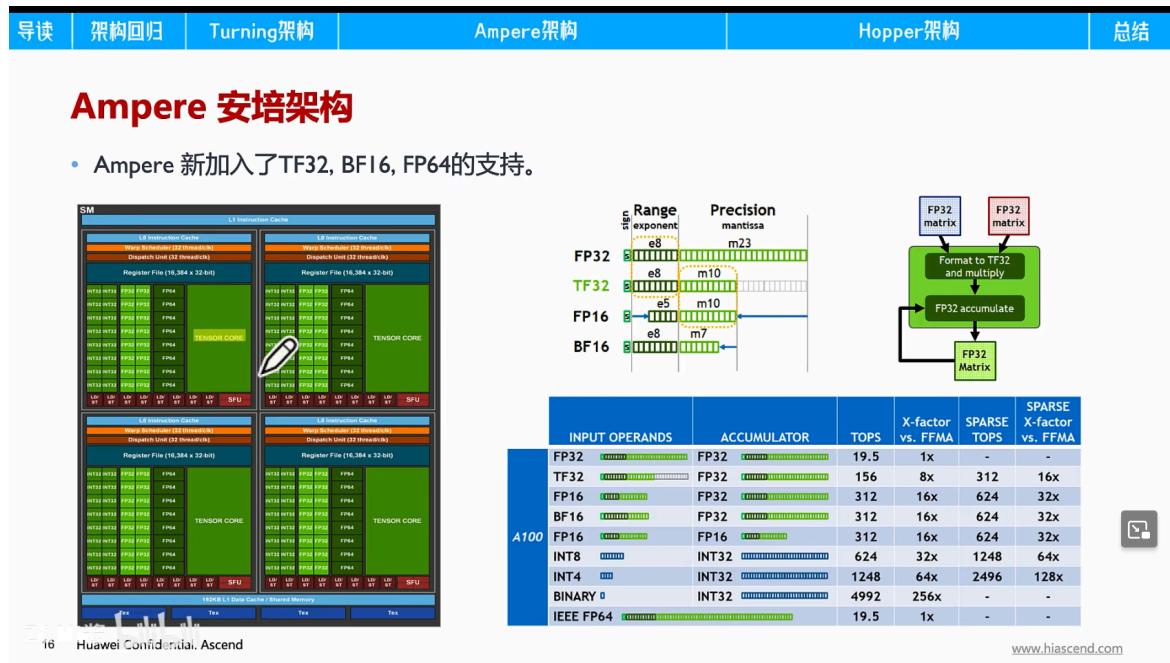


整体架构：常规元器件升级



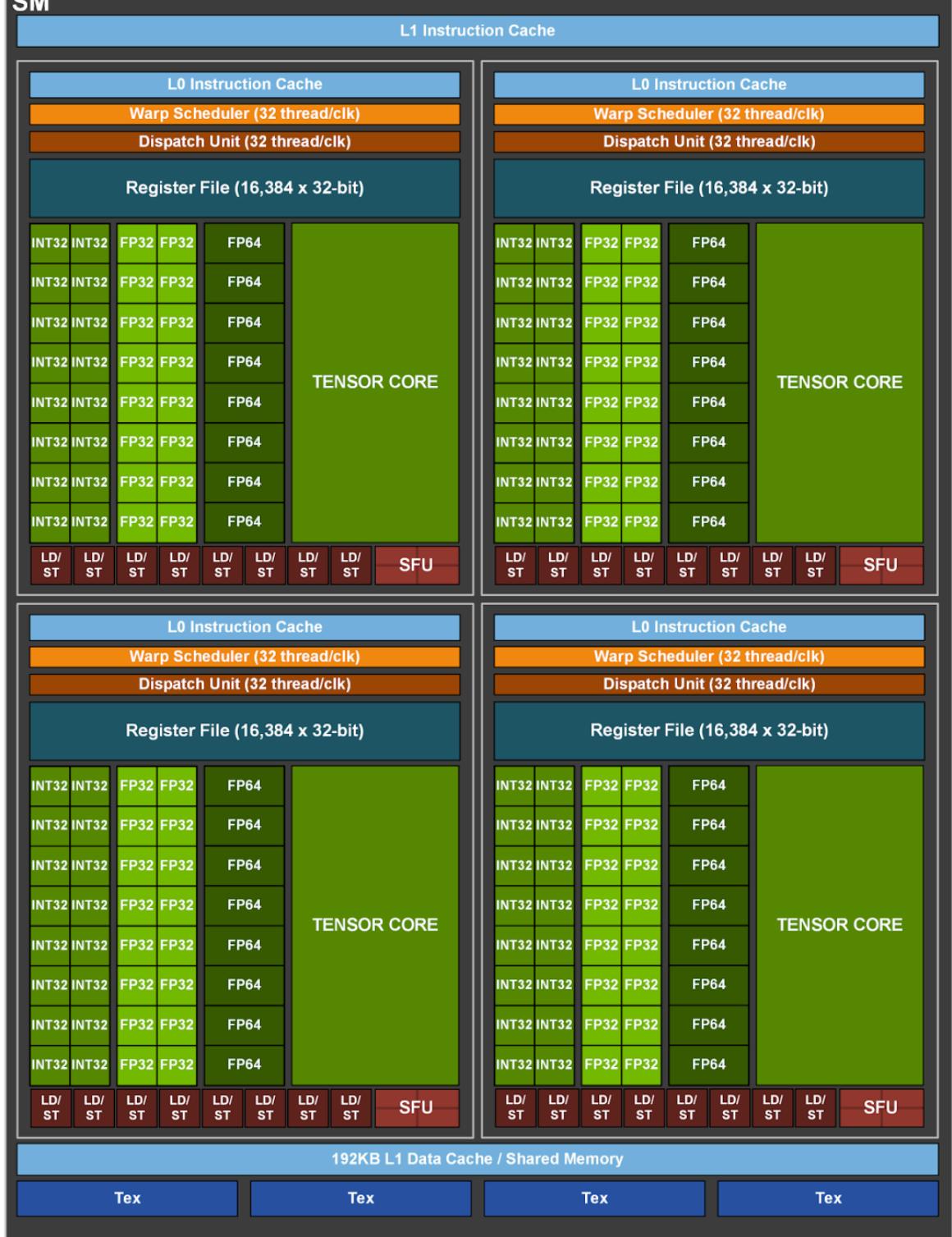
SM架构：

亮点1：加入了TF32, BF16, FP64的支持



刚才GPU部分的那张图就是A100的SM架构图

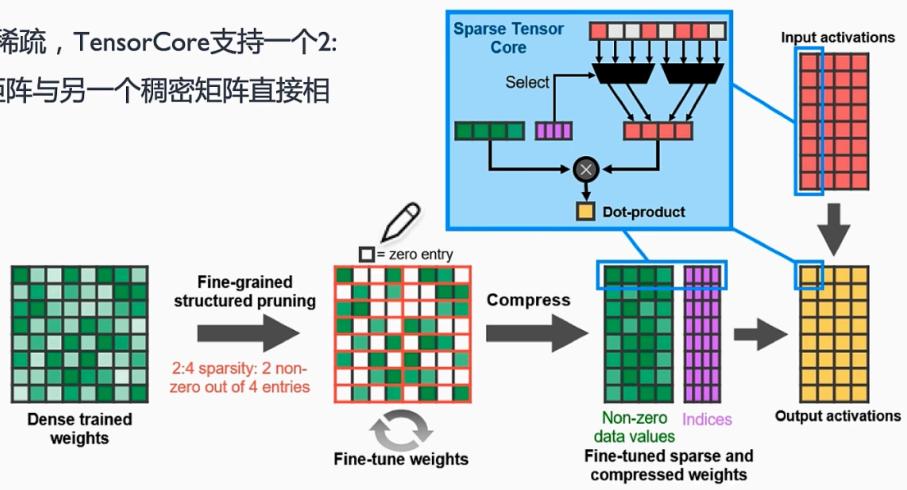
SM



亮点2：结构化稀疏

Ampere 安培架构

- 细粒度的结构化稀疏，TensorCore支持一个2:4的结构化稀疏矩阵与另一个稠密矩阵直接相乘。



■ Hopper:

H100

整体架构：

感觉就是常规升级，比如核心的数量增加了，硬件又升级版本了

Hopper 赫柏架构

- GPC 8组，66组TPC、132组SM，总计有16896个CUDA核心、528个Tensor核心、50MB二级缓存。显存为新一代HBM3，容量80GB，位宽5120-bit，带宽高达3TB/s



SM架构：

相比A100常规增加了一些计算单元外

另外增加了一个**Tensor Memory Accelerator**（亮点1）（作用与cache类似，增强了数据传输（专用于Tensor Core的数据））

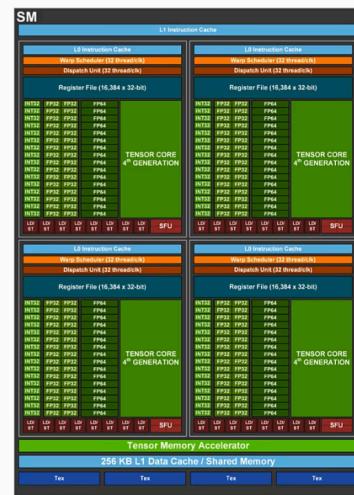
Hopper 赫柏架构

- **SM 结构：**

1. 4 个 Warp Scheduler , 4 个 Dispatch Unit (与 A100 一致)
2. 128 个 FP32 Core (4 * 32) (相比 A100 翻倍)
3. 64 个 INT32 Core (4 * 16) (与 A100 一致)
4. 64 个 FP64 Core (4 * 16) (相比 A100 翻倍)
5. 4 个 TensorCore (4 * 1)
6. 32 个 LD/ST Unit (4 * 8) (与 A100 一致)
7. 16 个 SFU (4 * 4) (与 A100 一致)
8. 相比 A100 增加了一个 Tensor Memory Accelerator

- **每个 Process Block :**

1. 1 个 Warp Scheduler , 1 个 Dispatch Unit (与 A100 一致)
2. 32 个 FP32 Core (相比 A100 翻倍)
3. 16 个 INT32 Core (与 A100 一致)
4. 16 个 FP64 Core (相比 A100 翻倍)
5. 1 个 TensorCore
6. 8 个 LD/ST Unit (与 A100 一致)
7. 4 个 SFU (与 A100 一致)



25 Huawei Confidential, Ascend

www.hiascend.com

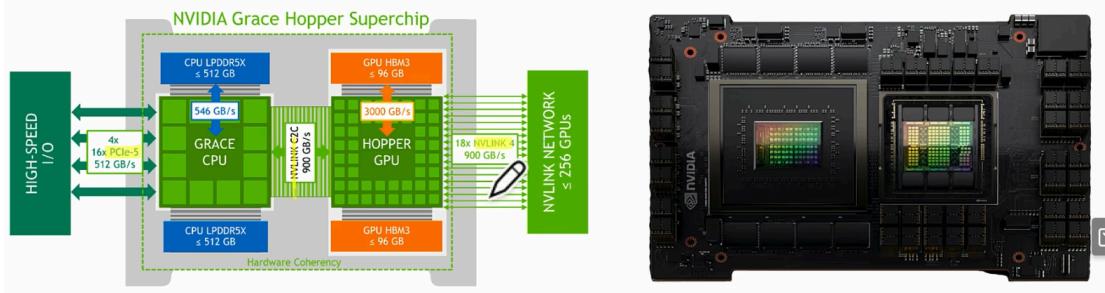
特点：

1. Grace Hopper Superchip (亮点2) :

CPU与GPU, GPU与GPU由pcie5.0升级为NVLink, 传输效率更高, 跨机还是pcie5.0

Hopper 赫柏架构

- NVIDIA Grace Hopper Superchip 架构将NVIDIA Hopper GPU的突破性性能与NVIDIA Grace CPU的多功能性结合在一起，在单个超级芯片中与高带宽和内存一致的 NVIDIA NVLink Chip-2-Chip (C2C)互连相连，并且支持新的NVIDIA NVLink 切换系统。

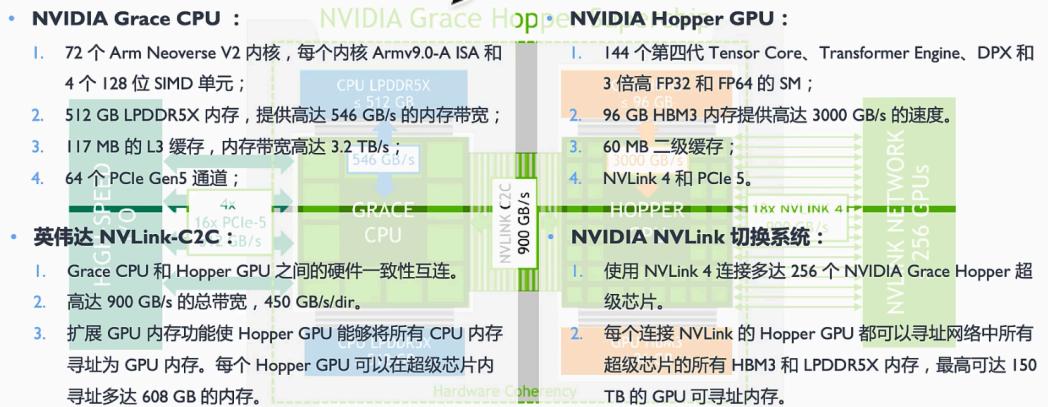


2. 其他两点

两颗NB的芯片和NVLink (亮点3)

Hopper 赫柏架构

第一个真正的异构加速平台，适用于高性能计算(HPC) 和AI工作负载



task_cuda

Task0：初始化：注意子模块的clone方法，进入父模块目录后，然后递归式克隆即可

```
lyjy@Lenovo:~$ git clone https://github.com/JoTang/JotangRecruitment.git
yjy@Lenovo:~$ cd JotangRecruitment/
lyjy@Lenovo:~/JotangRecruitment$ git submodule update --init --recursive
```

Task1: task_cuda

[CUDA入门教程--官方技术博客](#)

1. CUDA基本编程模型

- CUDA线程层次结构：Grid(在这个题中grid就是指的是两个128向量相加的这个问题)->Block->Thread
- 基本kernel语法： `__global__`

在函数加这个，告诉CUDA C++编译器，这个函数在GPU上面运行，指定的Thread都会运行这个函数

- 线程索引计算：

这里.x的x指代的是索引的方向，可以有x, y, z三个方向，下面的例子是一维的

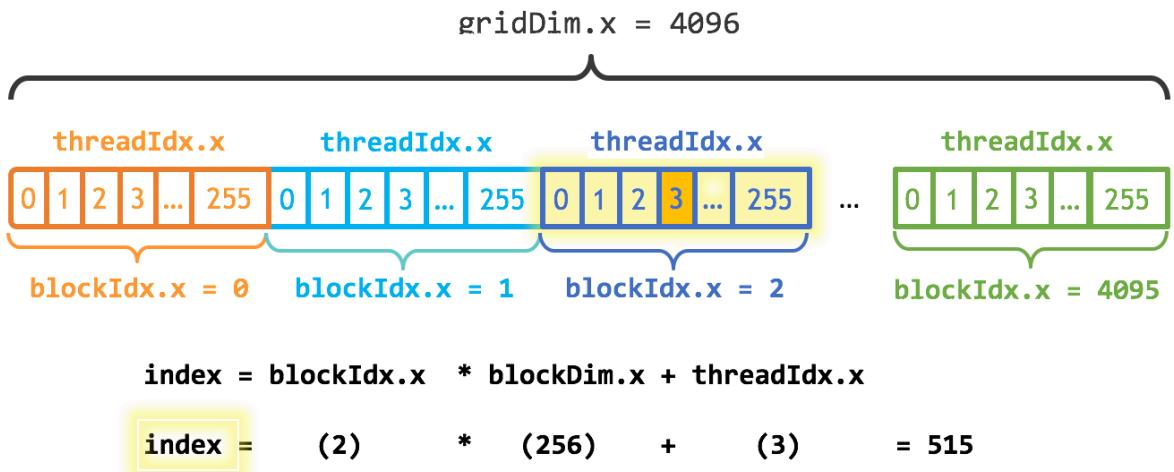
threadIdx.x:thread的索引

blockIdx.x: block的索引

gridDim.x: 一个grid里面的block的数量

blockDim.x: 一个block里面的thread的数量

注意后面两个谁是谁的数量



- Grid-Stride LOOP网格跨循环

引入stride的目的：为了让每个线程不止处理它所在的那一个元素，当线程总数（**stride**）小于 **num_elements** 时，可以让每个线程处理多个元素

```

int index = blockIdx.x * blockDim.x + threadIdx.x;
int stride = blockDim.x * gridDim.x;
for(int i = index; i < num_elements; i += stride)

```

- Kernel 启动配置: `<<<blocks, threads>>>`

threads为32的倍数

2. Pytorch C++扩展开发--详情可以去看代码

pybind/nanobind工作流：

1. 先编写cpp/cu文件，这里拿cu文件举例，先编写内核kernel函数，然后launch_kernel，然后包装成tensor函数，这个函数是以后暴露给py的函数，然后写pybind宏给出对应关系
2. setup.py--构建脚本
3. test.py--测试脚本

PyBind (在 PyTorch CUDA 扩展中的) 工作流

1. 写 CUDA/C++ 实现

- 在 `.cu/.cpp` 里写内核与 Host 包装。
- 使用 `torch::Tensor` 作为接口类型 (方便与 Python/Torch 交互)。
- 用 `data_ptr<T>()` 取得设备或主机指针 (取决于张量所在设备)。

2. 写绑定 (pybind11)

- `#include <torch/extension.h>` (间接带来 pybind11 + Tensor 类型转换)。
- `PYBIND11_MODULE(modname, m)` 注册模块与函数 (或类)。
- 也可用 `TORCH_LIBRARY` 注册成原生 op (更高级的集成)，但你当前用的是 pybind 的最简方式。

3. 写构建脚本

- `setup.py` 用 `CUDAExtension/BuildExtension`；或只用 `pyproject.toml + setuptools`。
- 指定 `sources`、编译参数 (调试/发布)、(可选) 自定义 `include/library` 路径等。

4. 构建 & 安装

- `pip install -v .` (PEP 517, 干净) 或 `python setup.py build_ext -- inplace` (本地生成 `.so`)。
- `BuildExtension` 自动配置: PyTorch/pybind11 头文件、NVCC、ABI 选项等。
- 产物是 `CUDA_Test.*.so`，位于包目录或 `site-packages`。

5. Python 端调用

- `import CUDA_Test` → 加载 `.so`。
- 创建 `torch.Tensor` (CPU/GPU) → 传入绑定函数。
- pybind11 + PyTorch 的类型转换器把 Python 侧的 `torch.Tensor` 转成 C++ 的 `torch::Tensor` (零拷贝封装)。
- C++ 里取 `data_ptr` 发射内核 (在当前 CUDA stream 上更规范: `at::cuda::getCurrentCUDAStream()`)。
- 内核写回到输出张量的同一块显存。
- 返回 `torch::Tensor` 给 Python (共享同一底层存储)。

6. 同步与错误处理 (工程实践)

- 尽量避免 `cudaDeviceSynchronize()` 全局同步，用 **当前流 + C10_CUDA_KERNEL_LAUNCH_CHECK()** 做错误检查。
- 对输入做 `TORCH_CHECK` (设备、`dtype`、`shape`、`contiguous`)。
- 发布构建用 `-O3`、去掉 `-G`，合适 `block/thread` 配置 (如 256/512 线程/块) 并考虑 `numBlocks`、`blockSize`、`numCUDAs` 的占

Test成功实拍:

The screenshot shows a terminal window with the following content:

```
cuda.cu 2, M  X  test_cuda.py 1
MLSys > task_cuda > cuda.cu > kernel(const float * __restrict__ const float * __restrict__ float * __restrict__ const int
1  #include <cmath>
2
3
4
5
6
7  global void kernel(const float * __restrict__ A_ptr,
8          const float * __restrict__ B_ptr,
9          float * __restrict__ result_ptr,
10         const int num_elements) {
11     // TODO: Implement the Kernel Logic
12     // Sum up all the elements in the input tensor
13     int index = blockIdx.x * blockDim.x + threadIdx.x;
14     int stride = blockDim.x * gridDim.x;
15     for(int i = index; i < num_elements; i += stride) result_ptr[i] = A_ptr[i] + B_ptr[i];
16 }
17
18 static void launch_kernel(const void *A_ptr, const void *B_ptr, void *output_ptr,
19                         const int num_elements) {
20     // TODO: Implement the LaunchKernel Logic
21     int blockSize = 32;
22     int numBlocks = (num_elements + blockSize - 1)/blockSize;
23     kernel<<<numBlocks,blockSize>>>((const float*)A_ptr,(const float*)B_ptr, (float*)output_ptr);
24     std::cout << "\nWarning: Need to implement the this!!\n\n" << std::endl;
25     cudaDeviceSynchronize();
26 }
27
28 torch::Tensor test_kernel(const torch::Tensor &A, const torch::Tensor &B) {
29     torch::Tensor result_tensor = torch::empty_like(A);
30     const int element_count = A.numel();
31
32     launch_kernel(A.data_ptr<float>(), B.data_ptr<float>(), result_tensor.data_ptr<float>(),
33                   element_count);
34 }
```

终端输出:

```
bash-task_cuda [ ] ×
, 0.7951478362883435, 1.6543710231781086, 0.7034199833869934, 1.0862088203430176, 0.366586714982986
45, 0.35412260890000702, 1.1871788501739562, 1.7464871406555176, 1.3459932804107666, 1.5428597927093
506, 0.8053569793701172, 1.119266390800476, 0.9181014895439148, 0.8967684507369995, 0.9228112101554
871, 0.9854409694671631, 1.1505610942840576, 1.1595016717910767, 0.9166581630796787, 1.161376714706
421, 0.8205728530883789, 0.9738439321517944, 0.494543194770813, 0.5947635769844055, 1.2193336486816
406, 1.4934245347976685, 0.9563506841659546, 1.0340826511383057, 0.7213424444198608, 0.962548255920
4102, 0.5651145577430725, 1.3388419151306152, 0.8373092412948608, 0.9926177859306335, 0.969807505600
7605, 0.5645904541015625, 0.8617590069778813, 0.4793970286846161, 1.2291861772537231, 0.57806694507
59888, 0.8525081276893616, 1.1572681665420532, 1.6173107624053955, 1.0562862157821655, 1.0181710720
062256, 1.0893210172653198, 1.507350206375122, 1.3939523696899414, 1.6031924486160278, 0.7491721510
887146, 1.250277042388916, 1.5245563983917236, 1.2227314710617065, 1.304854033775024, 1.9209115505
218506, 0.9372850656509399, 1.6427509784698486, 0.9215759038925171, 1.2939114578617676, 1.103055000
3051758, 0.7144927978515625, 0.5532891154289246, 1.2059760093688965, 1.528724193572998, 1.245609045
0286865, 0.29813575744628906, 1.1167173385620117, 0.8960847854614258, 0.7508409023284912, 0.9485052
824828386, 0.9385963082313538, 1.7531893253326416, 1.1061846017837524, 0.9268558025360107, 1.704822
1691131592, 1.2758774757385254, 0.90376216173172, 0.19317156076431274, 0.4793774485588074, 0.590528
2627182007, 1.9178650379180908, 1.8082164525985718, 1.4124231338500977, 1.4203763008117676]
Test passed! Result is correct
```

Test completed!

```
bash-task_cuda [ ] ×
(task_cuda) (nanobindPractice) lyjy@Lenovo:~/JotangRecruitment/MLSys/task_cuda$
```

并行基础

- 什么是指令并行, 数据并行, 任务并行:
 - 指令并行: 指在单个处理器核心内部, 通过硬件技术 (如流水线、多发射、乱序执行, 超标量等) 让多条机器指令同时处于执行阶段的特性。
 - 数据并行: 指同样的操作 (指令) 同时应用于不同的数据元素上。这是最直观、最常见的一种并行模式。
 - 任务并行: 指不同的处理单元同时执行不同的任务 (函数或操作)。这些任务可能是同一个程序的不同部分, 也可能是完全不同的程序。
- 什么是SIMD, SMT, SPMD:
 - SIMD: 单指令多数据。一种硬件指令集架构, 允许一条指令同时操作多个数据。

例子: 向量化指令: 向量寄存器, 一条SIMD指令可以对这个寄存器中的多个数据一次性同时相加, 而不是执行四次单独的加法指令

单个核心上实现数据并行

- SIMT: 单指令多线程。GPU采用的执行模型。

一组内所有线程在同一周期内执行相同的指令，但操作的是不同的数据，并且拥有自己独立的指令地址计数器和寄存器状态

实现数据并行，但以线程为单位

- SPMD: 单程序多数据。一种并行编程模型。所有处理单元执行同一段程序，但操作的数据不同

例子：CUDA多线程程序

CUDA编程

- CUDA编程[CUDA编程指南工具书](#)

[CUDA工具书](#)

- CUDA编程范式：

1. 主机(Host)和设备(Device)

- Host:CPU端，运行一般逻辑代码
- Device: GPU端，运行大规模并行计算

2. 异构编程：

- CPU 负责 调度、内存管理、串行逻辑。
- GPU 负责 大规模并行计算核心。

3. Kernel函数

4. 线程层次结构

1. 初始化与数据准备 (CPU):

- 在 CPU 上分配内存 (`malloc`) 。
- 初始化需要计算的数据。

2. 设备内存分配 (CPU → GPU):

- 使用 `cudaMalloc()` 在 GPU 上分配内存。此时 CPU 获得的是指向 GPU 内存的指针，不能直接解引用。

3. 数据传输 (CPU → GPU):

- 使用 `cudaMemcpy(..., cudaMemcpyHostToDevice)` 将输入数据从 CPU 内存拷贝到第 2 步分配的 GPU 内存中。这是一个相对耗时的操作。

4. 启动内核 (CPU → GPU):

- CPU 通过 `kernel<<<grid, block>>>(parameters)` 的语法启动 GPU 上的内核函数。这是一个**异步**操作，CPU 发出启动命令后几乎立即继续执行后续代码，而 GPU 开始并行计算。

5. 并行执行内核 (GPU):

- GPU 接收到指令后，创建指定的网格和块。
- 成千上万个线程被调度到 GPU 的多个流式多处理器 (SMs) 上同时执行。每个线程执行内核函数中的代码，但处理不同的数据（通过唯一的线程索引 `i` 区分）。

6. 结果回传 (GPU → CPU):

- 内核计算完成后，使用 `cudaMemcpy(..., cudaMemcpyDeviceToHost)` 将计算结果从 GPU 内存拷贝回 CPU 内存。这个操作会**同步** Host 和 Device，即 CPU 会等待这个拷贝操作完成。

7. 清理资源 (CPU):

- 使用 `cudaFree()` 释放 GPU 内存。
- 使用 `free()` 释放 CPU 内存。
- 限定符：
 - `__global__` :

1. __global__

- **含义：**声明一个 **GPU 内核函数 (kernel function)** 。
- **执行位置：**在 **device (GPU)** 上执行。
- **调用位置：**只能由 **host (CPU)** 调用。
- **返回值：**必须是 `void`。
- **调用方式：**使用 CUDA 特有的 `<<<gridDim, blockDim>>>` 语法来启动。

例子：

cpp

复制代码

```
__global__  
void add(int *a, int *b, int *c, int N) {  
    int idx = threadIdx.x + blockDim.x * blockIdx.x;  
    if (idx < N) {  
        c[idx] = a[idx] + b[idx];  
    }  
}  
  
int main() {  
    // 假设内存已分配并拷贝  
    add<<<(N+255)/256, 256>>>(d_a, d_b, d_c, N); // 在 GPU 上启动核函数  
}
```

■ __device__

2. __device__

- **含义：**声明一个 **设备函数 (device function)** 。
- **执行位置：**在 **device (GPU)** 上执行。
- **调用位置：**只能由 **device (GPU)** 内的其它函数 (包括 `__global__` 或 `__device__`) 调用。
- **返回值：**可以是任意类型。
- **调用方式：**和普通 C++ 函数调用一样 (不能在 **host** 端直接调用) 。

例子：

cpp

复制代码

```
__device__  
int square(int x) {  
    return x * x;  
}  
  
__global__  
void squareArray(int *d_out, int *d_in, int N) {  
    int idx = threadIdx.x + blockDim.x * blockIdx.x;  
    if (idx < N) {  
        d_out[idx] = square(d_in[idx]); // 调用 __device__ 函数  
    }  
}
```

■ __host__

3. `__host__`

- **含义：**显式声明一个 **主机函数 (host function)**。
- **执行位置：**在 **host (CPU)** 上执行。
- **调用位置：**只能由 host 调用。
- **默认情况：**如果没有写任何修饰符，函数默认就是 `__host__`。

例子：

```
cpp
```

```
__host__
void printHello() {
    printf("Hello from CPU\n");
}
```

 复制代码

■ `__host__ __device__`

4. 组合修饰符

- 可以同时用 `__host__ __device__` 修饰一个函数，表示：
 - 该函数既能在 host 端调用 (CPU 执行) ，
 - 也能在 device 端调用 (GPU 执行) 。

例子：

```
cpp
```

```
__host__ __device__
int add(int a, int b) {
    return a + b;
}

__global__
void kernel(int *out) {
    out[threadIdx.x] = add(threadIdx.x, 10); // GPU 上调用
}

int main() {
    printf("%d\n", add(3, 4)); // CPU 上调用
}
```

 复制代码

 注意：有时候 host 和 device 编译器环境不同，比如 `printf` 不能在 device 函数里用，所以要写跨平台函数时需要小心。

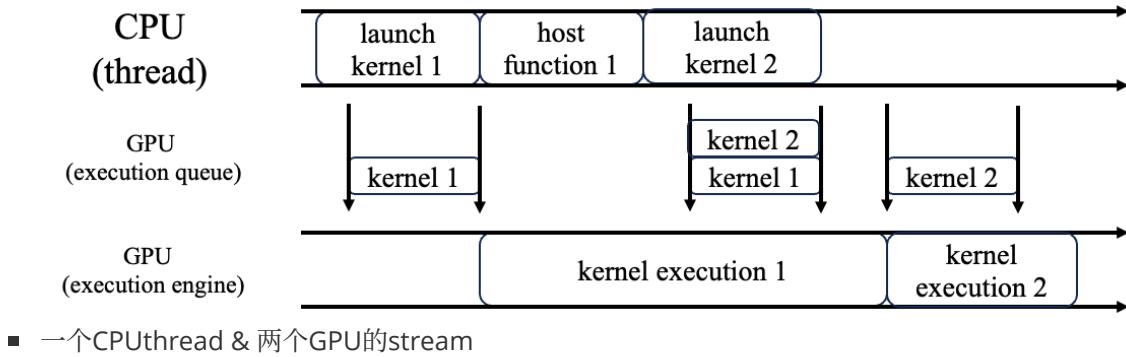
○ CUDA异步特性

CUDA的异步特性：让CPU和GPU尽可能的同时忙碌起来

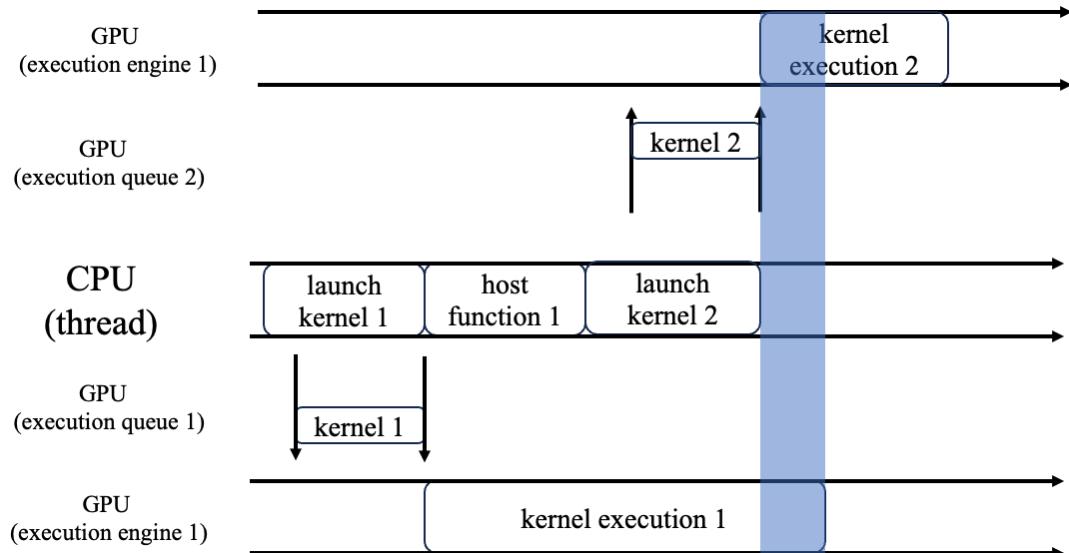
■ CUDA Stream

[CUDAsream--里面有例子用的时候再去看](#)

- 一个CPUthread & 一个GPU的stream



- 一个CPU thread & 两个GPU的stream



蓝条指定了两个kernel的并发执行阶段

但是能不能并发执行（比如2执行需要用到1的资源）需要用户来决定，所以增加了stream的概念

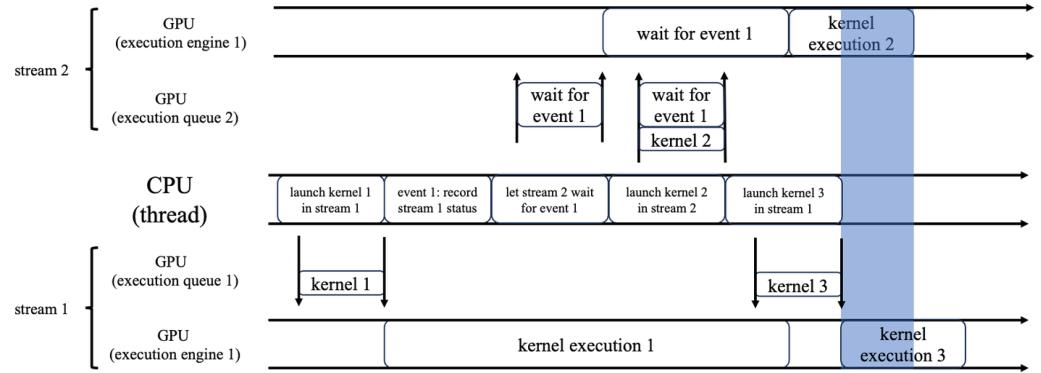
为了给用户提供这种控制权，于是我们就有了stream的概念。一个stream就对应于一个执行队列（加一个执行单元），用户可以自行决定是否把两个kernel分开放在两个队列里。

stream是cuda为上层应用提供的抽象，应用可以创建任意多个stream，下发任意多个kernel。但如果正在执行的kernel数目超过了硬件的execution engine的数量，那么即使当前stream里没有kernel正在执行，下发的kernel也必须在队列里继续等待，直到有硬件资源可以执行。（注意：此时kernel执行与CPU下发kernel之间依然是异步的）

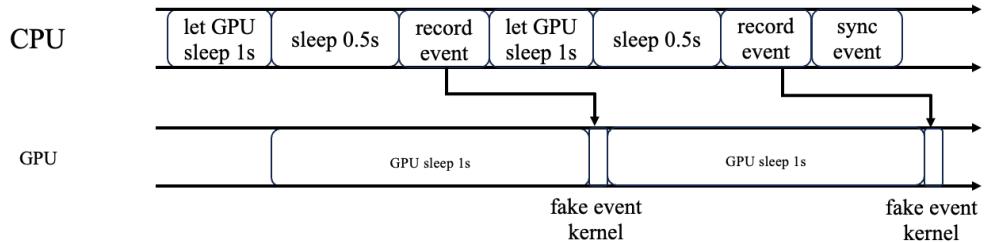
由此，我们可以总结得到：一个GPU kernel可以执行的必要条件是它在stream的队列首位且存在执行kernel的空闲硬件资源。

CUDA event:

- 它的作用1：让2必须要等待1进行完才能继续执行



- 作用2：计时

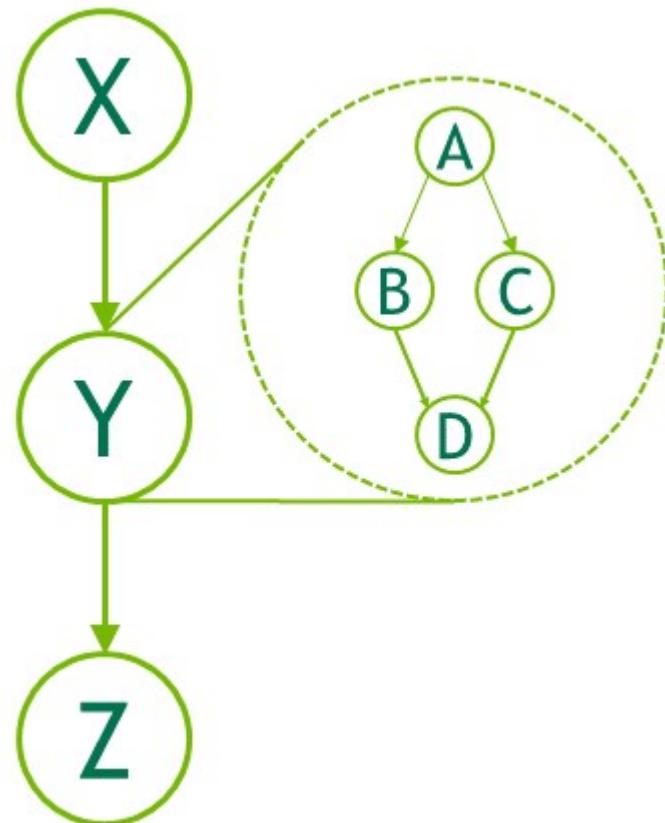


- CUDA Graph

同样知道是什么东西大概流程是怎样的就行了，剩下的回头再去看

CUDA graph

- 流程：将一系列预先定义好的 CUDA 操作（内核启动、内存拷贝等）组合成一个整体的“图”（Graph），然后作为一个单个的、庞大的单元来一次性提交和执行
- 节点：



- 与stream相比：

GPU主导，GPU在拿到完整的图后，可以全局优化整个工作流的执行，例如更早的发现并行性

极低的CPU启动开销：CPU只需要下发整个图一次，而不是图中的每一个操作，这消除了绝大部分的启动开销

但是CUDA stream是动态的，灵活性更高，也更简单

- GEMM

- 选中特定的进行调试：ctrl+shift+p -> 调试：选择并开始调试，然后选择你需要用的cuda-gdb

记得每次修改完代码都需要手动cmake重新构建

- CPU版本基本实现：

计算过程

矩阵乘法公式：

$$C[i][j] = \sum_{k=0}^{K-1} A[i][k] \times B[k][j]$$

这个公式意思是：C 的每个元素 $C[i][j]$ ，是 A 的第 i 行和 B 的第 j 列对应元素的乘积求和。

$$C[M, N] = A[M, K] * B[K, N]$$

$M \times N$ 的 C 矩阵，外循环，遍历 C 中每一个的元素然后计算，内循环循环 k 次（自己去模拟一下就明白了）

```
void gemm_example(float* A, float* B, float* C, int M, int N, int K) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            float sum = 0.0f;
            for (int k = 0; k < K; k++) {
                sum += A[i * K + k] * B[k * N + j];
            }
            C[i * N + j] = sum;
        }
    }
}
```

索引公式：

索引公式

如果你要访问矩阵的第 i 行、第 k 列元素 $A[i][k]$ ，在一维数组中对应的索引是：

$$\text{index} = i \times K + k$$

解释：

- $i * K$ → 跳过前 i 行，每行有 K 个元素
- $+ k$ → 在第 i 行中选择第 k 个元素

所以：

cpp

复制代码

```
A[i * K + k] // 就是矩阵 A 的第 i 行第 k 列
```

■ GPU实现版本kernel

```
__global__ void gemm_baseline_kernel(float *A, float *B, float *C, int M, int N, int K) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < K; k++) {
            sum += A[row * K + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

■ 优化版本：

[gemm优化1--知乎](#)

[gemm优化2--知乎](#)

[gemm优化3--知乎](#)

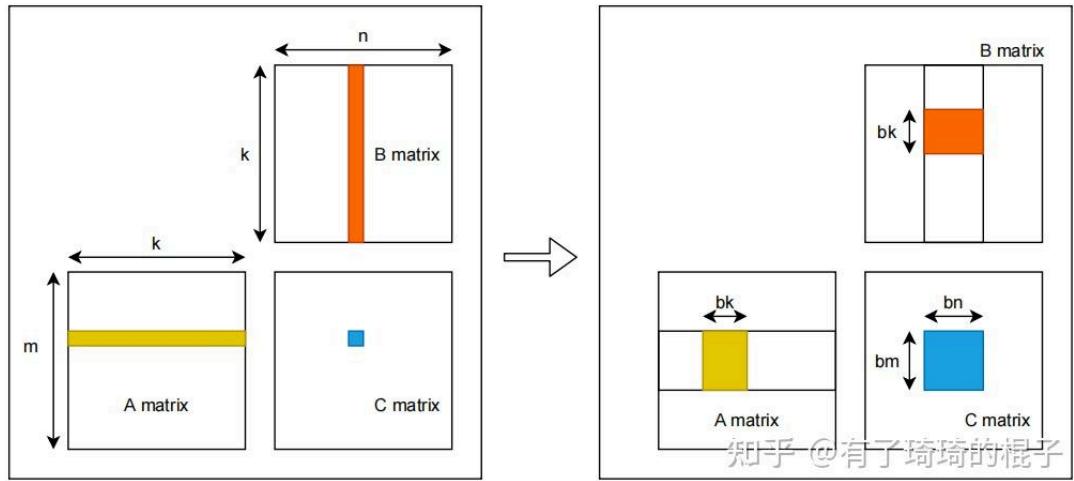
目前这里用到了，使用共享内存，线程块tiling，预取数据，向量化访问的操作

1. 从global memory 到 shared memory(使用shared memory)

核心理念：

1. 减少访存次数

2. 更好的利用了数据局部性，从shared memory拿出减少了访存的时延



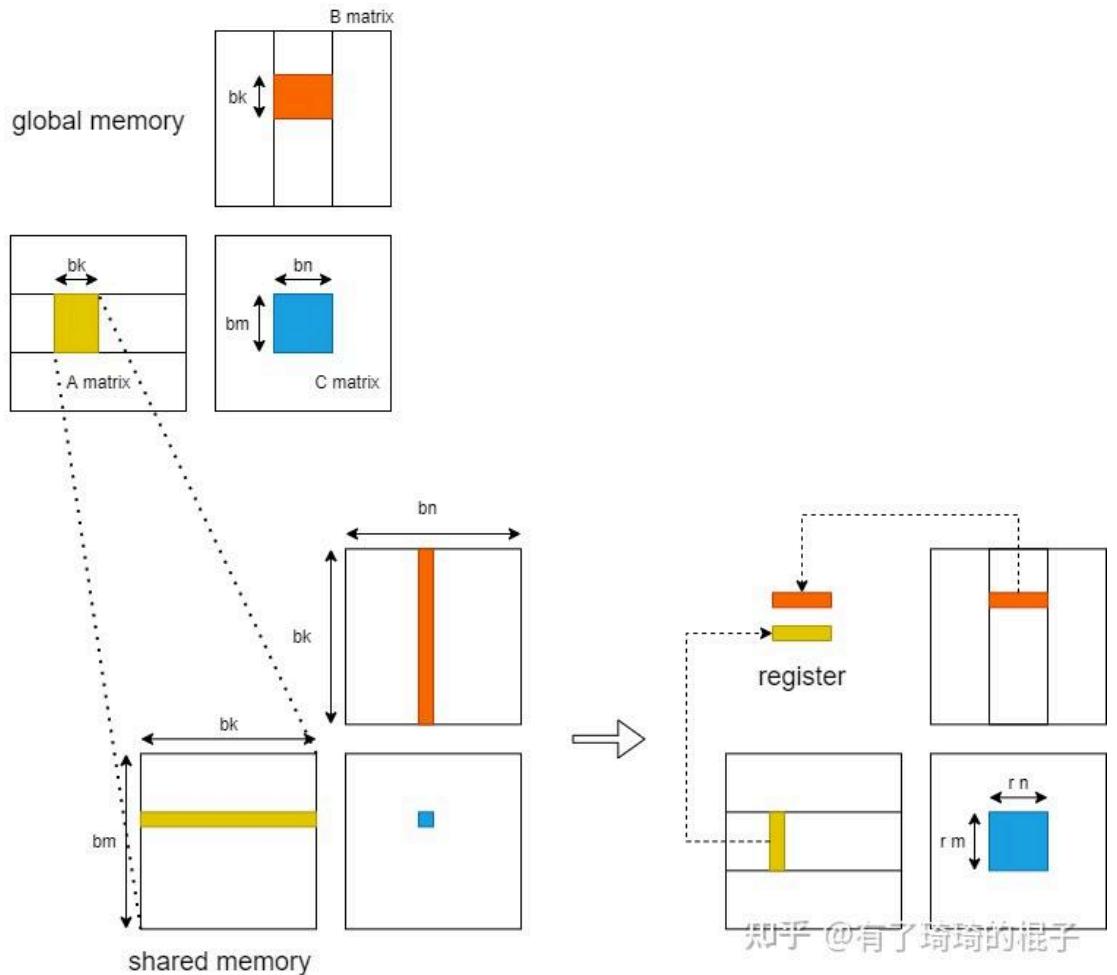
从global memory进行的 $m \times n$ 次写操作 (C矩阵每一个位置都需要写), $2mnk$ 次读操作 (从代码开始想, 内层循环一共k次每次访问A, B中各一个元素, 这就是 $2 \times k$ 个, 外层循环 $m \times n$, 这就是 $m \times n \times k \times 2$ 个)

分块逻辑: 首先将A、B、C三个矩阵划分为多个维度为 $bm \times bk$, $bk \times bn$, $bm \times bn$ 的小矩阵块。三个矩阵形成 $M \times K$, $K \times N$, $M \times N$ 的小矩阵网格。其中 $M=m/bm$, $N=n/bn$, $K=k/bk$ 。随后在GPU中开启 $M \times N$ 个block, 每个block负责C中一个维度为 $bm \times bn$ 的小矩阵块的计算。计算中一共有K次迭代, 每一次迭代都需要读取A中一个维度为 $bm \times bk$ 的小矩阵块和B中一个维度为 $bk \times bn$ 的小矩阵块, 并将其放置在shared memory中。因而, 完成C中所有元素的计算一共需要从global memory中读取 $M \times N \times K$ ($bm \times bk + bk \times bn$), 化简后, 为 $mnk(1/bn + 1/bm)$

2. 从shared memory到register

这里建立在上述存储到shared memory之后, 在K轮的某轮循环中, 将shared memory的 $bm \times bk$, $bk \times bn$ 存入register的过程上的优化

核心理念: 仍然是减少访存的次数和时延



同样，分块前 $2 \times bm \times bn \times bk$ 次读操作

分块后（和上面的思路一样）

而后考虑对shared memory进行分块，对 $bm \times bn$ 的小矩阵进行再一次划分，将其划分为多个维度为 $rm \times rn$ 的子矩阵。则一个block需要负责 $X \times Y$ 个子矩阵。其中， $X = \frac{bm}{rm}$ ， $Y = \frac{bn}{rn}$ 。随后，在一个block中开启 $X \times Y$ 个线程，**每个线程负责一个维度为 $rm \times rn$ 的子矩阵的计算**。在计算中，一个block一共需要从shared memory读取 $X \times Y \times (rm + rn) \times bk$ ，即 $bm \times bn \times bk \times (\frac{1}{rm} + \frac{1}{rn})$ 个单精度浮点数。相比于未分块的算法，对于shared memory中的访存量减少为原来的 $1/2 \times (\frac{1}{rm} + \frac{1}{rn})$ 。并且，由于将数据放入register中，可以直接对数据进行运算，减少了从shared memory中取数的时延。

3. register分块

考虑分到寄存器之后（一个thread内）

每个线程有 rm 个A矩阵寄存器值, rn 个B矩阵寄存器值, $rm \times rn$ 个C矩阵寄存器值

需要计算 $rm \times rn$ 个数值，需要 $rm \times rn$ 个FFMA指令

涉及到的问题：bank conflict

产生原因可以理解为当一条指令的源寄存器有2个以上来自同一bank，就会产生冲突，指令就会重发射，浪费一个cycle

解决方法：通过对C的精巧排列，但是这种排列无法完全避免

[扩展知识--知乎](#)

[为什么这种精巧排列会奏效--还没看](#)

这个时候会涉及到寄存器的bank conflict。在NV的GPU中，每个SM不仅会产生shared memroy之间的bank冲突，也会产生寄存器之间的bank冲突。这一点对于计算密集型的算子十分重要。像shared memory一样，寄存器的Register File也会被分为几个bank，如果一条指令的源寄存器有2个以上来自同一bank，就会产生冲突。指令会重发射，浪费一个cycle。PS：这个地方是从旷视的博客中看的。然后对于maxwell架构的GPU而言，bank数为4，寄存器 $id \% 4$ 即所属bank。

我们假设对这个thread来说， $rm = 4, rn = 4$ 。并且计算C的寄存器以一种非常naive的情况分配，如下图左侧所示。则需要产生16条FFMA指令，列举如下：

```
FFMA R0, R16, R20, R0
FFMA R1, R16, R21, R1
....
```



可以从中看出，这会产生大量的register bank冲突，所以需要对参与计算的寄存器重新进行分配和排布，如上图右侧所示。在有些地方，这种方式也可以叫做register分块。

4. prefetch

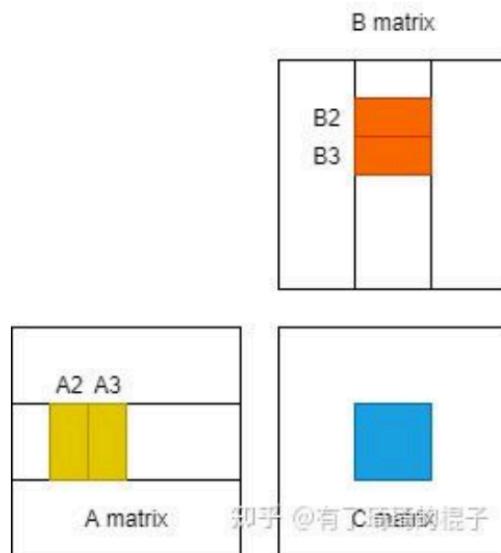
目的：减少访存的延迟

核心思想：通过在前一次循环的时候预分配下一轮需要用的计算资源，减少下一轮不必要的等待，从而实现让计算资源处于不断的计算之中的一种状态

数据的prefetch

最后，我们来讲讲如何通过对数据进行prefetch来减少访存的latency。我们再来看GEMM的过程，并且仔细地看看这个访存的latency到底是怎么导致的。对于一个**block**而言，需要计算一个 $bm * bn$ 的矩阵块，这个时候需要进行K次迭代，每次迭代都需要先将来自A和B的两个小块送到shared memory中再进行计算。而从global中访存实际上是非常慢的，所以导致了latency。虽然GPU中可以通过block的切换来掩盖这种latency，但是由于分配的shared memory比较多，活跃的block并不多，这种延时很难被掩盖。对于一个**thread**，需要计算一个 $rm * rn$ 的小矩阵，但是必须先将数据从shared memory传到寄存器上，才能开始进行计算。所以导致了每进行一次迭代，计算单元就需要停下来等待，计算单元不能被喂饱。

为此，需要进行数据的Prefetch来尽可能地掩盖这种latency。思想也比较简单，需要多开一个buffer，进行读写分离。示意图如下。当block进行第2轮迭代时，需要对A2和B2进行计算，在计算单元进行计算的同时，我们将A3和B3提前放置到shared memory。而后，在进行第3轮迭代时，就可以直接对shared memory中的A3和B3进行计算，而不需要等待从global memory搬运到shared memory的时间。寄存器上的Prefetch也是同理。



采用prefetch的计算流程：

未采用时，大迭代+小迭代

采用后，开启shared memory和register的数量有所不同，还需要提前将数据放置到shared memory和register中

有两种实现方法，一种是开启双倍的shared memory和register，另一种是将原来的分为一半

开启两倍的shared memory和register，一块用来当前循环读取，一块用来下一轮循环写，所以需要两倍的内存空间

为了实现数据预取，需要开启两倍的**shared memory**和**寄存器**。当然也可以将原来**shared memory**切分成两块，也就是将 $bm \times bk$ 和 $bk \times bn$ 的矩阵一分为二。以A中的小矩阵而言，变成了两个 $bm \times bk/2$ 。然后大迭代次数由原来的256变成了512。很多地方把这个技术叫做**双缓冲**，我感觉跟预取是同一个事情。无非是针对参数**bk**的大小换不同说法。所以在这里统一叫做数据预取。废话说得有点多。总之，我们还是开启两倍的**shared memory**和**寄存器**数据。在一个block中，原来在**shared memory**中需要存储的数据是 $bm \times bk + bk \times bn$ 。现在变成了 $bm \times bk \times 2 + bk \times bn \times 2$ 。在一个thread中，为了存储A和B的数据，原来需要使用 $rm + rn$ 个寄存器，现在需要使用 $2 \times (rm + rn)$ 个寄存器。为了后续方便介绍，我们用**read SM**和**write SM**代表用来读写的两块共享内存，并用**read REG**和**write REG**来表示用来读写的两块寄存器。

把共享内存和寄存器的事情说明白之后，我们来看看具体的计算逻辑。在执行256次大迭代之前，我们需要提前将第0次大迭代的数据存到**write SM**中，并且将第0次小迭代的数据存到**write REG**中。在完成这一个预取过程之后，我们再来仔细地看看第0个大迭代。需要注意的是，上一轮大迭代的**write SM**就是这一轮迭代的**read SM**。上一轮小迭代的**write REG**就是这一轮迭代的**read REG**。所以在进行第0个大迭代时，上面**write SM**就变成了**read SM**。然后我们首先需要将下一轮大迭代的数据存到**write SM**中。由于从**global memory**中取数的时钟周期非常多。所以在等待数据取回的同时，对**read SM**中的数据进行计算。也就是我们在等待的同时，需要开启8次小迭代来进行计算。而小迭代中也存在着读写分离，在对**read REG**进行计算之前，需要先执行**write REG**的操作，通过这种方式来掩盖访存的**latency**。所以整体的计算逻辑如下：

- 代码实现：

```
template <
    const int BLOCK_SIZE_M, //bm
    const int BLOCK_SIZE_K, //bk
    const int BLOCK_SIZE_N, //bn
    const int THREAD_SIZE_Y, //rm
    const int THREAD_SIZE_X, //rn
    const bool ENABLE_DOUBLE_BUFFER //是否开启prefetch (开启双缓冲)
>
__global__ void gemm_kernel(float *A, float *B, float *C, int M, int N, int K) {
    //TODO: 实现你自己的Kernel
    /*参数说明部分*/
    // Block index
    //纵: x代表第几列
    int bx = blockIdx.x;
    //横: y代表第几行
    int by = blockIdx.y;
    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    //每个Block中有多少个线程Thread
    //这个是纵
    const int THREAD_X_PER_BLOCK = BLOCK_SIZE_N / THREAD_SIZE_X;
    //这个是横
    const int THREAD_Y_PER_BLOCK = BLOCK_SIZE_M / THREAD_SIZE_Y;
    const int THREAD_NUM_PER_BLOCK = THREAD_X_PER_BLOCK *
        THREAD_Y_PER_BLOCK;
    //在一个block中thread的ID
```

```

const int tid = ty * THREAD_X_PER_BLOCK + tx;

//global memory->shared memory经过register--这里搬运的时候利用了向量化的知识, 指一次搬4个float
//BLOCK_SIZE_M * BLOCK_SIZE_K / THREAD_NUM_PER_BLOCK--一个thread有多少个元素
//再除以4为一个线程需要搬几次到寄存器上
const int ldg_num_a = BLOCK_SIZE_M * BLOCK_SIZE_K /
(THREAD_NUM_PER_BLOCK * 4);
const int ldg_num_b = BLOCK_SIZE_K * BLOCK_SIZE_N /
(THREAD_NUM_PER_BLOCK * 4);
//4×ldg_num_a其实是一个线程有多少元素, 即这里开的其实是一个线程的寄存器
float ldg_a_reg[4*ldg_num_a];
float ldg_b_reg[4*ldg_num_b];

//搬运A一行需要多少线程, 一行bk个元素, 一个线程每次搬4个float
const int A_TILE_THREAD_PER_ROW = BLOCK_SIZE_K / 4;
const int B_TILE_THREAD_PER_ROW = BLOCK_SIZE_N / 4;
//当前线程需要搬运的横向坐标, 搬运第几行的数据
const int A_TILE_ROW_START = tid / A_TILE_THREAD_PER_ROW;
const int B_TILE_ROW_START = tid / B_TILE_THREAD_PER_ROW;
//当前线程需要搬运的纵向坐标, 起始点
const int A_TILE_COL = tid % A_TILE_THREAD_PER_ROW * 4;
const int B_TILE_COL = tid % B_TILE_THREAD_PER_ROW * 4;
//步长: 多次搬运需要跨越多少行
const int A_TILE_ROW_STRIDE = THREAD_NUM_PER_BLOCK /
A_TILE_THREAD_PER_ROW;
const int B_TILE_ROW_STRIDE = THREAD_NUM_PER_BLOCK /
B_TILE_THREAD_PER_ROW;
//shared memory, 这里A转置存储了(这里利用了行主序的知识, 指行的内存地址是连续的)
//因为后面需要固定k去取元素, 这样转置后固定k后取元素就变成连续的了, 这样就变快了
__shared__ float As[2][BLOCK_SIZE_K][BLOCK_SIZE_M];
__shared__ float Bs[2][BLOCK_SIZE_K][BLOCK_SIZE_N];

//register
// registers for C
float accum[THREAD_SIZE_Y][THREAD_SIZE_X] = {0};
// registers for A and B
float frag_a[2][THREAD_SIZE_Y];
float frag_b[2][THREAD_SIZE_X];

//prefetch: 迭代之前预取数据
//第一部分: 将A global memory->shared memory
//这里是循环展开的使用
#pragma unroll
//这个循环代表着block中的每个线程需要搬几次才能把数据搬到shared memory
for ( int i = 0 ; i < BLOCK_SIZE_M ; i += A_TILE_ROW_STRIDE) {
    //指向存储的起始位置
    int ldg_index = i / A_TILE_ROW_STRIDE * 4;

    FETCH_FLOAT4(ldg_a_reg[ldg_index]) = FETCH_FLOAT4(A[OFFSET(
        BLOCK_SIZE_M * by + A_TILE_ROW_START + i, // row
        A_TILE_COL, // col

```

```

        K )]);
    As[0][A_TILE_COL][A_TILE_ROW_START + i]=ldg_a_reg[ldg_index];
    As[0][A_TILE_COL+1][A_TILE_ROW_START + i]=ldg_a_reg[ldg_index+1];
    As[0][A_TILE_COL+2][A_TILE_ROW_START + i]=ldg_a_reg[ldg_index+2];
    As[0][A_TILE_COL+3][A_TILE_ROW_START + i]=ldg_a_reg[ldg_index+3];
}
// load B from global memory to shared memory
#pragma unroll
for ( int i = 0 ; i < BLOCK_SIZE_K; i += B_TILE_ROW_STRIDE) {
    FETCH_FLOAT4(Bs[0][B_TILE_ROW_START + i][B_TILE_COL]) =
    FETCH_FLOAT4(B[OFFSET(
        B_TILE_ROW_START + i, // row
        B_TILE_COL + BLOCK_SIZE_N * bx, // col
        N )]);
}
//确保线程块都执行完了才会执行下面的代码
__syncthreads();

// load A from shared memory to register
//注意这里的As已经是转置之后的了，所以计算逻辑和Bs相同
//取bn/bm个数，只取第一次的，即As[0][0]
#pragma unroll
for (int thread_y = 0; thread_y < THREAD_SIZE_Y; thread_y += 4) {
    FETCH_FLOAT4(frag_a[0][thread_y]) = FETCH_FLOAT4(As[0][0]
[THREAD_SIZE_Y * ty + thread_y]);
}
// load B from shared memory to register
#pragma unroll
for (int thread_x = 0; thread_x < THREAD_SIZE_X; thread_x += 4) {
    FETCH_FLOAT4(frag_b[0][thread_x]) = FETCH_FLOAT4(Bs[0][0]
[THREAD_SIZE_X * tx + thread_x]);
}

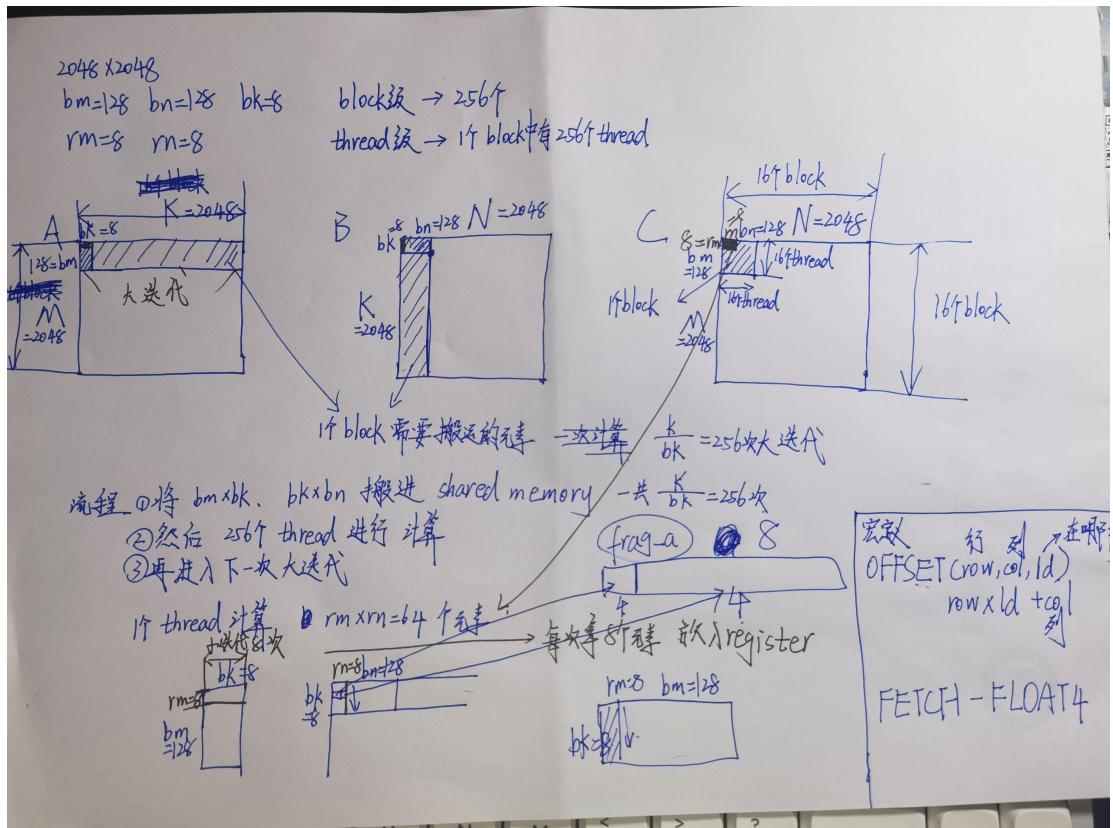
//大迭代
int write_stage_idx = 1;
//每次读取bk行（转置了），直到读到K结束
int tile_idx = 0;
do{
    tile_idx += BLOCK_SIZE_K;
    //保证每次写的和读的是不同块
    int load_stage_idx = write_stage_idx ^ 1;
    //具体的大迭代，如果还有下一次大迭代，则将下一个迭代的数据块搬运到寄存器上
    if(tile_idx< K){
        #pragma unroll
        for ( int i = 0 ; i < BLOCK_SIZE_M ; i += A_TILE_ROW_STRIDE) {
            int ldg_index = i / A_TILE_ROW_STRIDE * 4;
            FETCH_FLOAT4(ldg_a_reg[ldg_index]) = FETCH_FLOAT4(A[OFFSET(
                BLOCK_SIZE_M * by + A_TILE_ROW_START + i, // row
                A_TILE_COL + tile_idx, // col
                K )]);
        }
        #pragma unroll
        for ( int i = 0 ; i < BLOCK_SIZE_K; i += B_TILE_ROW_STRIDE) {
    
```

```

        int ldg_index = i / A_TILE_ROW_STRIDE * 4;
        FETCH_FLOAT4(ldg_b_reg[ldg_index]) = FETCH_FLOAT4(B[OFFSET(
            tile_idx + B_TILE_ROW_START + i, // row
            B_TILE_COL + BLOCK_SIZE_N * bx, // col
            N)]);
    }
}
//小迭代
#pragma unroll
for(int j=0; j<BLOCK_SIZE_K-1; ++j){
    // load next tile from shared mem to register
    // load A from shared memory to register
    #pragma unroll
    for (int thread_y = 0; thread_y < THREAD_SIZE_Y; thread_y += 4) {
        FETCH_FLOAT4(frag_a[(j+1)%2][thread_y]) =
    FETCH_FLOAT4(As[load_stage_idx][j+1][THREAD_SIZE_Y * ty + thread_y]);
    }
    // load B from shared memory to register
    #pragma unroll
    for (int thread_x = 0; thread_x < THREAD_SIZE_X; thread_x += 4) {
        FETCH_FLOAT4(frag_b[(j+1)%2][thread_x]) =
    FETCH_FLOAT4(Bs[load_stage_idx][j+1][THREAD_SIZE_X * tx + thread_x]);
    }
    // compute C THREAD_SIZE_X x THREAD_SIZE_Y, 计算本轮
    #pragma unroll
    for (int thread_y = 0; thread_y < THREAD_SIZE_Y; ++thread_y) {
        #pragma unroll
        for (int thread_x = 0; thread_x < THREAD_SIZE_X; ++thread_x) {
            accum[thread_y][thread_x] += frag_a[j%2][thread_y] *
frag_b[j%2][thread_x];
        }
    }
}
//将刚才存储在临时寄存器的数据搬到shared memory (为什么刚才不搬, 因为中间用到了)
if(tile_idx < K){
    #pragma unroll
    for ( int i = 0 ; i < BLOCK_SIZE_M ; i += A_TILE_ROW_STRIDE) {
        int ldg_index = i / A_TILE_ROW_STRIDE * 4;
        As[write_stage_idx][A_TILE_COL][A_TILE_ROW_START +
i]=ldg_a_reg[ldg_index];
        As[write_stage_idx][A_TILE_COL+1][A_TILE_ROW_START +
i]=ldg_a_reg[ldg_index+1];
        As[write_stage_idx][A_TILE_COL+2][A_TILE_ROW_START +
i]=ldg_a_reg[ldg_index+2];
        As[write_stage_idx][A_TILE_COL+3][A_TILE_ROW_START +
i]=ldg_a_reg[ldg_index+3];
    }
    // load B from global memory to shared memory
    #pragma unroll
    for ( int i = 0 ; i < BLOCK_SIZE_K; i += B_TILE_ROW_STRIDE) {
        int ldg_index = i / A_TILE_ROW_STRIDE * 4;
        FETCH_FLOAT4(Bs[write_stage_idx][B_TILE_ROW_START + i][B_TILE_COL]) =
    FETCH_FLOAT4(ldg_b_reg[ldg_index]);
}

```

画的草图



- 所用的优化手段: 转置 (A转置存储到shared memory中), 向量化 (使用float4取元素), 循环展开 (#pragma unroll), prefetch, 数据分块 (global memory->shared memory -> register)

- 性能报告 (不知道什么原因, 优化不高)

我觉得可能是bm, bn, bk, rm, rn这些值给的不好的问题

```
(triton-puzzle) lyjy@Lenovo:~/JotangRecruitment/MLSys/Gemm$ sudo
/usr/local/cuda-12.9/bin/ncu --target-processes all -f --export profile
./build/gemm_example
==PROF== Connected to process 52261
(/home/lyjy/JotangRecruitment/MLSys/Gemm/build/gemm_example)
Found 1 CUDA devices
Device name: NVIDIA GeForce RTX 4060 Laptop GPU
Compute capability: 8.9

--- 测试矩阵大小: 256x256 x 256x256 ---
CPU参考时间: 7 ms
==PROF== Profiling "gemm_baseline_kernel" - 0: 0%....50%....100% - 8
passes
Baseline GEMM [256x256] x [256x256] = [256x256]
执行时间: 345.411 ms
性能: 0.0971435 GFLOPS
==PROF== Profiling "gemm_kernel" - 1: 0%....50%....100% - 8 passes
Opt GEMM [256x256] x [256x256] = [256x256]
执行时间: 111.605 ms
性能: 0.300653 GFLOPS
结果验证CPU和GPU: 通过
结果验证GPU和GPUOPT: 通过
加速比 = 3.09493
```

```
--- 测试矩阵大小: 512x512 x 512x512 ---
CPU参考时间: 65 ms
==PROF== Profiling "gemm_baseline_kernel" - 2: 0%....50%....100% - 8
passes
Baseline GEMM [512x512] x [512x512] = [512x512]
执行时间: 110.758 ms
性能: 2.42362 GFLOPS
==PROF== Profiling "gemm_kernel" - 3: 0%....50%....100% - 8 passes
Opt GEMM [512x512] x [512x512] = [512x512]
执行时间: 147.708 ms
性能: 1.81734 GFLOPS
结果验证CPU和GPU: 通过
结果验证GPU和GPUOPT: 通过
加速比 = 0.749846

--- 测试矩阵大小: 1024x1024 x 1024x1024 ---
CPU参考时间: 2215 ms
==PROF== Profiling "gemm_baseline_kernel" - 4: 0%....50%....100% - 8
passes
Baseline GEMM [1024x1024] x [1024x1024] = [1024x1024]
执行时间: 139.605 ms
性能: 15.3826 GFLOPS
==PROF== Profiling "gemm_kernel" - 5: 0%....50%....100% - 8 passes
Opt GEMM [1024x1024] x [1024x1024] = [1024x1024]
执行时间: 117.257 ms
性能: 18.3143 GFLOPS
结果验证CPU和GPU: 通过
结果验证GPU和GPUOPT: 通过
加速比 = 1.19058

--- 测试矩阵大小: 2048x2048 x 2048x2048 ---
CPU参考时间: 18039 ms
==PROF== Profiling "gemm_baseline_kernel" - 6: 0%....50%....100% - 8
passes
Baseline GEMM [2048x2048] x [2048x2048] = [2048x2048]
执行时间: 370.425 ms
性能: 46.3788 GFLOPS
==PROF== Profiling "gemm_kernel" - 7: 0%....50%....100% - 8 passes
Opt GEMM [2048x2048] x [2048x2048] = [2048x2048]
执行时间: 143.227 ms
性能: 119.949 GFLOPS
结果验证CPU和GPU: 通过
结果验证GPU和GPUOPT: 通过
加速比 = 2.58629
```

- 从汇编代码再次优化

- 从汇编代码分析程序性能
 - NV GPU提供了ptx和sass两个层面的汇编码
[使用cuobjdump和nvdism查看汇编码](#)
 - 看汇编代码的时候到底在看什么

- 访存密集型kernel
查看关于内存的优化在汇编层面有没有实现（即有没有正确实现优化）
- 计算密集型kernel
查看计算指令的占比高不高，有没有正确实现并行策略
- 对现有sgemm的代码分析及观察
 - scott的maxas(在CUDA层面，不涉及汇编)
 1. global->shared memory采用texture内存，将线程划分，一半线程只读A，一半线程只读B
 2. **shared memory->register**，将 8×8 的读取变成4个 4×4 的读取从而避免bank冲突（重点）
 3. store C的时候，为了合并访存，采用奇怪方式去store
- 汇编级代码调整--减少FFMA指令所产生的register bank冲突
 - 寄存器重映射--不同架构采用的寄存器映射方案不同
Ada Lovelace4060架构，应该去找该型号对应的寄存器映射方案
 - 调整FFMA顺序--执行顺序不同
作用就是调整FFMA执行顺序以再次减少register bank的冲突
- 代码实现

相比上面的优化

1. 解决shared memory bank冲突

如何将shared memory中 64×2 个元素平均分配给32个thread



一个wrap中有32个thread

tid/32确定是哪一个线程里面的，%32是在wrap中哪个位置

```
const int warp_id = tid / 32;
```

```
const int lane_id = tid % 32;
```

这个部分的内容主要是介绍一下怎么解决GEMM（二）所存在的shared memory冲突。其实scott的文章已经说了这一点，但是吧，实在是太费解了。首先，再来看看一下这个思路。我们一个block有256个线程，8个warp，8个warp要去取shared memory中的半行元素，也就是 $128/2=64$ 个元素。warp0和warp4取得是同样的16个元素。而warp里面，线程0、2、4、6、8、10、12、14是取得同样的4个元素。由于取得是同样的元素，同一个bank触发多播的机制，没有冲突。取多少元素说清楚了，就得说一下shared memory的索引了。scott给出的256线程版本索引是：

我来说一下我的计算方法，以B矩阵对应的shared memory为例，首先，计算warp_id，也就是当前线程属于哪个warp，由 $\text{tid}/32$ 即可得。随后计算lane_id，即当前线程属于这个warp上得哪个线程，由 $\text{tid}\%32$ 即可得。随后就是通过warp_id和lane_id来算出，对应128个元素得哪一个元素。先算 $(\text{warp_id}\%4)\times 16$ ，假设是warp2，就是上图左侧的第2个（从0算）warp。前面有2个warp，跳过了 $2\times 16=32$ 个元素。然后再看看当前lane_id。0-15在左半边，16-31在右半边。所以lane_id/16，先看是左半边还是右半边。右半边的话，先跳过8个元素。最后再看lane_id的奇偶数，如果奇数的话，就再跳一个四个元素。代码实现如下，这个就是正常人可以看懂的方式了。对A矩阵的映射关系同理。

- 解读：

在共享内存下的布局

- `As[double_buffer][k][m]` - 对于固定的k, m从0到127变化
- `Bs[double_buffer][k][n]` - 对于固定的k, n从0到127变化

2. 把storeC的代码从 8×8 改成了4个 4×4

为什么使用 `a_tile_index` 和 `b_tile_index` 计算全局内存位置：因为共享内存和全局内存的存储模式是对称的

```
const int c_block_row = a_tile_index;
const int c_block_col = b_tile_index;

//store C00 block
for(int i=0; i<4; i++){
    FETCH_FLOAT4(C[OFFSET(
        BLOCK_SIZE_M * by + c_block_row + i,
        BLOCK_SIZE_N * bx + c_block_col,
        N)]) = FETCH_FLOAT4(accum[i][0]);
}

//store C01 block
for(int i=0; i<4; i++){
    FETCH_FLOAT4(C[OFFSET(
        BLOCK_SIZE_M * by + c_block_row + i,
        BLOCK_SIZE_N * bx + c_block_col + 64,
        N)]) = FETCH_FLOAT4(accum[i][4]);
}

//store C10 block
for(int i=0; i<4; i++){
    FETCH_FLOAT4(C[OFFSET(
        BLOCK_SIZE_M * by + c_block_row + 64 + i,
        BLOCK_SIZE_N * bx + c_block_col,
        N)]) = FETCH_FLOAT4(accum[i+4][0]);
}
```

```

//store C11 block
for(int i=0; i<4; i++){
    FETCH_FLOAT4(C[OFFSET(
        BLOCK_SIZE_M * by + c_block_row + 64 + i,
        BLOCK_SIZE_N * bx + c_block_col + 64,
        N)]) = FETCH_FLOAT4(accum[i+4][4]);
}

```

```

template <
    const int BLOCK_SIZE_M, // height of block of C that each thread
    block calculate
    const int BLOCK_SIZE_K, // width of block of A that each thread
    block load into shared memory
    const int BLOCK_SIZE_N, // width of block of C that each thread
    block calculate
    const int THREAD_SIZE_Y, // height of block of C that each thread
    calculate
    const int THREAD_SIZE_X, // width of block of C that each thread
    calculate
    const bool ENABLE_DOUBLE_BUFFER // whether enable double buffering
or not
>
__global__ void gemm_kernel(
    float * __restrict__ A,
    float * __restrict__ B,
    float * __restrict__ C,
    const int M,
    const int N,
    const int K) {
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // the threads number in Block of X,Y
    const int THREAD_X_PER_BLOCK = BLOCK_SIZE_N / THREAD_SIZE_X;
    const int THREAD_Y_PER_BLOCK = BLOCK_SIZE_M / THREAD_SIZE_Y;
    const int THREAD_NUM_PER_BLOCK = THREAD_X_PER_BLOCK *
    THREAD_Y_PER_BLOCK;

    // thread id in cur Block
    const int tid = ty * THREAD_X_PER_BLOCK + tx;

    // shared memory
    __shared__ float As[2][BLOCK_SIZE_K][BLOCK_SIZE_M];
    __shared__ float Bs[2][BLOCK_SIZE_K][BLOCK_SIZE_N];
    // registers for C
    float accum[THREAD_SIZE_Y][THREAD_SIZE_X];
}

```

```

#pragma unroll
for(int i=0; i<THREAD_SIZE_Y; i++){
    #pragma unroll
    for(int j=0; j<THREAD_SIZE_X; j++){
        accum[i][j]=0.0;
    }
}
// registers for A and B
float frag_a[2][THREAD_SIZE_Y];
float frag_b[2][THREAD_SIZE_X];
// registers load global memory
const int ldg_num_a = BLOCK_SIZE_M * BLOCK_SIZE_K /
(THREAD_NUM_PER_BLOCK * 4);
const int ldg_num_b = BLOCK_SIZE_K * BLOCK_SIZE_N /
(THREAD_NUM_PER_BLOCK * 4);
float ldg_a_reg[4*ldg_num_a];
float ldg_b_reg[4*ldg_num_b];

// threads number in one row
const int A_TILE_THREAD_PER_ROW = BLOCK_SIZE_K / 4;
const int B_TILE_THREAD_PER_ROW = BLOCK_SIZE_N / 4;

// row number and col number that needs to be loaded by this thread
const int A_TILE_ROW_START = tid / A_TILE_THREAD_PER_ROW;
const int B_TILE_ROW_START = tid / B_TILE_THREAD_PER_ROW;

const int A_TILE_COL = tid % A_TILE_THREAD_PER_ROW * 4;
const int B_TILE_COL = tid % B_TILE_THREAD_PER_ROW * 4;

// row stride that thread uses to load multiple rows of a tile
const int A_TILE_ROW_STRIDE = THREAD_NUM_PER_BLOCK /
A_TILE_THREAD_PER_ROW;
const int B_TILE_ROW_STRIDE = THREAD_NUM_PER_BLOCK /
B_TILE_THREAD_PER_ROW;

A = &A[(BLOCK_SIZE_M * by)* K];
B = &B[BLOCK_SIZE_N * bx];

//load index of the tile
const int warp_id = tid / 32;
const int lane_id = tid % 32;
const int a_tile_index = warp_id/2*16 + lane_id/8*4; //warp_id * 8
+ (lane_id / 16)*4; // (warp_id/4)*32 + ((lane_id%16)/2)*4;
const int b_tile_index = warp_id%2*32 + lane_id%8*4; //((lane_id %
16) * 4; // (warp_id%4)*16 + (lane_id/16)*8 + (lane_id%2)*4;

//transfer first tile from global mem to shared mem
// load A from global memory to shared memory
#pragma unroll
for ( int i = 0 ; i < BLOCK_SIZE_M ; i += A_TILE_ROW_STRIDE) {
    int ldg_index = i / A_TILE_ROW_STRIDE * 4;
    FETCH_FLOAT4(ldg_a_reg[ldg_index]) = FETCH_FLOAT4(A[OFFSET(
        A_TILE_ROW_START + i, // row

```

```

        A_TILE_COL, // col
        K ]));
As[0][A_TILE_COL][A_TILE_ROW_START + i]=ldg_a_reg[ldg_index];
As[0][A_TILE_COL+1][A_TILE_ROW_START +
i]=ldg_a_reg[ldg_index+1];
As[0][A_TILE_COL+2][A_TILE_ROW_START +
i]=ldg_a_reg[ldg_index+2];
As[0][A_TILE_COL+3][A_TILE_ROW_START +
i]=ldg_a_reg[ldg_index+3];
}
// load B from global memory to shared memory
#pragma unroll
for ( int i = 0 ; i < BLOCK_SIZE_K; i += B_TILE_ROW_STRIDE) {
    FETCH_FLOAT4(Bs[0][B_TILE_ROW_START + i][B_TILE_COL]) =
    FETCH_FLOAT4(B[OFFSET(
        B_TILE_ROW_START + i, // row
        B_TILE_COL, // col
        N )]);
}
__syncthreads();

// load A from shared memory to register
FETCH_FLOAT4(frag_a[0][0]) = FETCH_FLOAT4(As[0][0][a_tile_index]);
FETCH_FLOAT4(frag_a[0][4]) = FETCH_FLOAT4(As[0][0][a_tile_index +
64]);
// load B from shared memory to register
FETCH_FLOAT4(frag_b[0][0]) = FETCH_FLOAT4(Bs[0][0][b_tile_index]);
FETCH_FLOAT4(frag_b[0][4]) = FETCH_FLOAT4(Bs[0][0][b_tile_index +
64]);
int write_stage_idx = 1;
int tile_idx = 0;
do{
    // next tile index
    tile_idx += BLOCK_SIZE_K;
    // load next tile from global mem
    if(tile_idx< K){
        #pragma unroll
        for ( int i = 0 ; i < BLOCK_SIZE_M ; i +=
A_TILE_ROW_STRIDE) {
            int ldg_index = i / A_TILE_ROW_STRIDE * 4;
            FETCH_FLOAT4(ldg_a_reg[ldg_index]) =
            FETCH_FLOAT4(A[OFFSET(
                A_TILE_ROW_START + i, // row
                A_TILE_COL + tile_idx, // col
                K )]);
        }
        #pragma unroll
        for ( int i = 0 ; i < BLOCK_SIZE_K; i += B_TILE_ROW_STRIDE)
        {
            int ldg_index = i / B_TILE_ROW_STRIDE * 4;

```

```

        FETCH_FLOAT4(ldg_b_reg[ldg_index]) =
FETCH_FLOAT4(B[OFFSET(
            tile_idx + B_TILE_ROW_START + i, // row
            B_TILE_COL, // col
            N )]);
    }
}

int load_stage_idx = write_stage_idx ^ 1;

#pragma unroll
for(int j=0; j<BLOCK_SIZE_K - 1; ++j){
    // load next tile from shared mem to register
    // load A from shared memory to register
    FETCH_FLOAT4(frag_a[(j+1)%2][0]) =
FETCH_FLOAT4(As[load_stage_idx][(j+1)][a_tile_index]);
    FETCH_FLOAT4(frag_a[(j+1)%2][4]) =
FETCH_FLOAT4(As[load_stage_idx][(j+1)][a_tile_index + 64]);
    // load B from shared memory to register
    FETCH_FLOAT4(frag_b[(j+1)%2][0]) =
FETCH_FLOAT4(Bs[load_stage_idx][(j+1)][b_tile_index]);
    FETCH_FLOAT4(frag_b[(j+1)%2][4]) =
FETCH_FLOAT4(Bs[load_stage_idx][(j+1)][b_tile_index + 64]);
    // compute C THREAD_SIZE_X x THREAD_SIZE_Y
    #pragma unroll
    for (int thread_y = 0; thread_y < THREAD_SIZE_Y;
++thread_y) {
        #pragma unroll
        for (int thread_x = 0; thread_x < THREAD_SIZE_X;
++thread_x) {
            accum[thread_y][thread_x] += frag_a[j%2][thread_y]
* frag_b[j%2][thread_x];
        }
    }
}

if(tile_idx < K){
    // load A from global memory to shared memory
    #pragma unroll
    for ( int i = 0 ; i < BLOCK_SIZE_M ; i +=
A_TILE_ROW_STRIDE) {
        int ldg_index = i / A_TILE_ROW_STRIDE * 4;
        As[write_stage_idx][A_TILE_COL][A_TILE_ROW_START +
i]=ldg_a_reg[ldg_index];
        As[write_stage_idx][A_TILE_COL+1][A_TILE_ROW_START +
i]=ldg_a_reg[ldg_index+1];
        As[write_stage_idx][A_TILE_COL+2][A_TILE_ROW_START +
i]=ldg_a_reg[ldg_index+2];
        As[write_stage_idx][A_TILE_COL+3][A_TILE_ROW_START +
i]=ldg_a_reg[ldg_index+3];
    }
    // load B from global memory to shared memory
    #pragma unroll
}

```

```

        for ( int i = 0 ; i < BLOCK_SIZE_K; i += B_TILE_ROW_STRIDE)
    {
        int ldg_index = i / B_TILE_ROW_STRIDE * 4;
        FETCH_FLOAT4(Bs[write_stage_idx][B_TILE_ROW_START + i]
[B_TILE_COL]) = FETCH_FLOAT4(ldg_b_reg[ldg_index]);
    }
    // use double buffer, only need one sync
    __syncthreads();
    // switch
    write_stage_idx ^= 1;
}

// load first tile from shared mem to register of next iter
// load A from shared memory to register
FETCH_FLOAT4(frag_a[0][0]) = FETCH_FLOAT4(As[load_stage_idx^1]
[0][a_tile_index]);
FETCH_FLOAT4(frag_a[0][4]) = FETCH_FLOAT4(As[load_stage_idx^1]
[0][a_tile_index + 64]);
// load B from shared memory to register
FETCH_FLOAT4(frag_b[0][0]) = FETCH_FLOAT4(Bs[load_stage_idx^1]
[0][b_tile_index]);
FETCH_FLOAT4(frag_b[0][4]) = FETCH_FLOAT4(Bs[load_stage_idx^1]
[0][b_tile_index + 64]);
// compute C THREAD_SIZE_X x THREAD_SIZE_Y
#pragma unroll
for (int thread_y = 0; thread_y < THREAD_SIZE_Y; ++thread_y) {
    #pragma unroll
    for (int thread_x = 0; thread_x < THREAD_SIZE_X;
++thread_x) {
        accum[thread_y][thread_x] += frag_a[1][thread_y] *
frag_b[1][thread_x];
    }
}
}while(tile_idx< K);

const int c_block_row = a_tile_index;
const int c_block_col = b_tile_index;

//store C00 block
for(int i=0; i<4; i++){
    FETCH_FLOAT4(C[OFFSET(
        BLOCK_SIZE_M * by + c_block_row + i,
        BLOCK_SIZE_N * bx + c_block_col,
        N)]) = FETCH_FLOAT4(accum[i][0]);
}
//store C01 block
for(int i=0; i<4; i++){
    FETCH_FLOAT4(C[OFFSET(
        BLOCK_SIZE_M * by + c_block_row + i,
        BLOCK_SIZE_N * bx + c_block_col + 64,
        N)]) = FETCH_FLOAT4(accum[i][4]);
}
//store C10 block

```

```

    for(int i=0; i<4; i++){
        FETCH_FLOAT4(C[OFFSET(
            BLOCK_SIZE_M * by + c_block_row + 64 + i,
            BLOCK_SIZE_N * bx + c_block_col,
            N)]) = FETCH_FLOAT4(accum[i+4][0]);
    }
    //store C11 block
    for(int i=0; i<4; i++){
        FETCH_FLOAT4(C[OFFSET(
            BLOCK_SIZE_M * by + c_block_row + 64 + i,
            BLOCK_SIZE_N * bx + c_block_col + 64,
            N)]) = FETCH_FLOAT4(accum[i+4][4]);
    }
}

```

```

--- 测试矩阵大小: 256x256 x 256x256 ---
CPU参考时间: 6 ms
==PROF== Profiling "gemm_baseline_kernel" - 0: 0%....50%....100% - 8
passes
Baseline GEMM [256x256] x [256x256] = [256x256]
执行时间: 351.052 ms
性能: 0.0955825 GFLOPS
==PROF== Profiling "gemm_kernel" - 1: 0%....50%....100% - 8 passes
Opt GEMM [256x256] x [256x256] = [256x256]
执行时间: 109.557 ms
性能: 0.306273 GFLOPS
结果验证CPU和GPU: 通过
结果验证GPU和GPUOPT: 通过
加速比 = 3.20428

--- 测试矩阵大小: 512x512 x 512x512 ---
CPU参考时间: 59 ms
==PROF== Profiling "gemm_baseline_kernel" - 2: 0%....50%....100% - 8
passes
Baseline GEMM [512x512] x [512x512] = [512x512]
执行时间: 114.746 ms
性能: 2.3394 GFLOPS
==PROF== Profiling "gemm_kernel" - 3: 0%....50%....100% - 8 passes
Opt GEMM [512x512] x [512x512] = [512x512]
执行时间: 149.876 ms
性能: 1.79105 GFLOPS
结果验证CPU和GPU: 通过
结果验证GPU和GPUOPT: 通过
加速比 = 0.765605

--- 测试矩阵大小: 1024x1024 x 1024x1024 ---
CPU参考时间: 2132 ms
==PROF== Profiling "gemm_baseline_kernel" - 4: 0%....50%....100% - 8
passes
Baseline GEMM [1024x1024] x [1024x1024] = [1024x1024]
执行时间: 144.139 ms
性能: 14.8987 GFLOPS

```

```
==PROF== Profiling "gemm_kernel" - 5: 0%....50%....100% - 8 passes
Opt GEMM [1024x1024] x [1024x1024] = [1024x1024]
执行时间: 115.737 ms
性能: 18.5549 GFLOPS
结果验证CPU和GPU: 通过
结果验证GPU和GPUOPT: 通过
加速比 = 1.2454

--- 测试矩阵大小: 2048x2048 x 2048x2048 ---
CPU参考时间: 16985 ms
==PROF== Profiling "gemm_baseline_kernel" - 6: 0%....50%....100% - 8
passes
Baseline GEMM [2048x2048] x [2048x2048] = [2048x2048]
执行时间: 354.345 ms
性能: 48.4834 GFLOPS
==PROF== Profiling "gemm_kernel" - 7: 0%....50%....100% - 8 passes
Opt GEMM [2048x2048] x [2048x2048] = [2048x2048]
执行时间: 164.651 ms
性能: 104.341 GFLOPS
结果验证CPU和GPU: 通过
结果验证GPU和GPUOPT: 通过
加速比 = 2.15209
```

nvcc编译优化版本

```
Found 1 CUDA devices
Device name: NVIDIA GeForce RTX 4060 Laptop GPU
Compute capability: 8.9

--- 测试矩阵大小: 256x256 x 256x256 ---
CPU参考时间: 6 ms
Baseline GEMM [256x256] x [256x256] = [256x256]
执行时间: 0.09632 ms
性能: 348.364 GFLOPS
Opt GEMM [256x256] x [256x256] = [256x256]
执行时间: 0.037184 ms
性能: 902.389 GFLOPS
结果验证CPU和GPU: 通过
结果验证GPU和GPUOPT: 通过
加速比 = 2.59036

--- 测试矩阵大小: 512x512 x 512x512 ---
CPU参考时间: 55 ms
Baseline GEMM [512x512] x [512x512] = [512x512]
执行时间: 0.393088 ms
性能: 682.889 GFLOPS
Opt GEMM [512x512] x [512x512] = [512x512]
执行时间: 0.065152 ms
性能: 4120.14 GFLOPS
结果验证CPU和GPU: 通过
结果验证GPU和GPUOPT: 通过
加速比 = 6.0334
```

```
--- 测试矩阵大小: 1024x1024 x 1024x1024 ---
CPU参考时间: 2154 ms
Baseline GEMM [1024x1024] x [1024x1024] = [1024x1024]
执行时间: 3.01709 ms
性能: 711.774 GFLOPS
Opt GEMM [1024x1024] x [1024x1024] = [1024x1024]
执行时间: 0.318656 ms
性能: 6739.19 GFLOPS
结果验证CPU和GPU: 通过
结果验证GPU和GPUOPT: 通过
加速比 = 9.46817

--- 测试矩阵大小: 2048x2048 x 2048x2048 ---
CPU参考时间: 17618 ms
Baseline GEMM [2048x2048] x [2048x2048] = [2048x2048]
执行时间: 23.9839 ms
性能: 716.307 GFLOPS
Opt GEMM [2048x2048] x [2048x2048] = [2048x2048]
执行时间: 2.21741 ms
性能: 7747.73 GFLOPS
结果验证CPU和GPU: 通过
结果验证GPU和GPUOPT: 通过
加速比 = 10.8162

==== 测试完成 ===
```

- Cutlass(Cute):

- Cutlass: 提供一套用于CUDA, 模块化的, 高性能的线性代数模板库 (相当于C++中的STL)
- Cute: **CUDA Tensor-primitive Emporium**

Cute引入了一种直观的类似于数学符号的DSL (领域特定语言) 来简化代码

代码直接描述张量的形状, 步长, 分块等, 抽象层次高, 可读性强

```

cpp

// 1. 定义逻辑上的张量形状和布局
auto layout_a = make_layout(make_shape(M, K), make_stride(K, 1)); // Row-major
auto layout_b = make_layout(make_shape(N, K), make_stride(1, N)); // Column-major
auto layout_c = make_layout(make_shape(M, N), make_stride(N, 1)); // Row-major

// 2. 定义物理上的分块策略 (Tiling)
auto tile_shape = make_shape(Int<128>{}, Int<128>{}, Int<32>{}); // <M, N, K> tile
auto tiled_mma = make_tiled_mma(MmaAtom{}, tile_shape); // 创建一个分块的MMA操作

// 3. 对张量 A, B, C 进行分块划分
auto thr_layout = tiled_mma.get_thr_layout(); // 获取线程的布局
auto cta_tiler = make_shape(M, N);
auto cta_coord = make_coord(cta_idx_x, cta_idx_y);

// 4. 在循环中, 每个线程块加载自己的分块 (Tile)
auto gA = make_tensor(ptr_a, layout_a);
auto gB = make_tensor(ptr_b, layout_b);
auto gC = make_tensor(ptr_c, layout_c);

// 获取当前线程块负责的 Tile
auto tile_a = local_tile(gA, cta_tiler, cta_coord, tiled_mma.get_a_layout());
auto tile_b = local_tile(gB, cta_tiler, cta_coord, tiled_mma.get_b_layout());
auto tile_c = local_tile(gC, cta_tiler, cta_coord, tiled_mma.get_c_layout());

// 5. 线程块内部分配数据给各个线程并进行计算
auto rA = make_fragment_like(tile_a);
auto rB = make_fragment_like(tile_b);
auto rC = make_fragment_like(tile_c);

copy(tile_a, rA); // 将全局内存数据加载到寄存器
copy(tile_b, rB);
gemm(tiled_mma, rC, rA, rB); // 执行计算
copy(rC, tile_c); // 将结果写回全局内存

```

- Cute Layout:

统一了数据布局和线程布局: 比如在传统CUDA编写中, 你需要定义数据块大小 (比如将一个矩阵分成若干个数据块), 定义线程块 (一个block中线程如何组织), 和映射关系 (这些线程如何分工解决这些数据块)

而Cute Layout使用Shape+Stride的组合同时描述数据和线程

即达到了一种自动切分 (自动计算映射关系) 的结果

cpp

复制 下载

```
// 假设我们有:  
auto thr_layout = Layout<Shape<_4, _8>, Stride<_8, _1>>; // 4x8 的线程布局  
auto tile_layout = Layout<Shape<_128, _128>, Stride<_128, _1>>; // 128x128 的数据布局 (row-major)  
  
// 核心操作: 将数据块划分给线程  
// 这会产生一个新的布局, 描述每个线程负责的数据子块 (tile) 的形状和内存分布  
auto per_thr_tile_layout = local_partition(tile_layout, thr_layout, thread_id);
```

`local_partition` 这个函数是魔法发生的地方。它根据 `thr_layout` 的结构, 将大的 `tile_layout` 切分成许多个小块, 并返回当前线程 (`thread_id`) 所对应的那个数据子块的布局。

- Cutlass 4.0: Cutt从一个组件变成了整个库的核心

所有组件通过Cutt来实现, 并提升了性能和可组合性

- 降低CUDA难度

- Triton: 简单编写, 并且能媲美CUDA的性能的一种语言和编译器
 - 为什么需要triton: triton的语法简单, 并且让用户只需要关注算法逻辑即可, 无需关注优化层面是怎么实现的

解决编写CUDA内核的复杂性, 允许开发者使用类似Py的语法直接编写高效的GPU内核

但是在特定的极致调优下还是CUDA快一点

比如在矩阵乘法的编写中, 使用triton编写只需要关注, 每个kernel处理哪一部分的矩阵

(`offset`, `mask`), 在边界的处理上语法也比较简洁, 然后load进来之后, 就tl.dot就可以实现矩阵乘法了, 并且是优化的高效的版本

再一个就是广播机制的使用, 在12题中, 可以使用广播机制在新维度上并行解包, 得出的代码也是简单的高效的

- Triton-puzzles-lite--12道题上手Triton

已完成

- Bonus Time

- 探究Pytorch算子下降到CUDA代码/Triton的过程

- Pytorch下降到CUDA

```
Python -> pybind11 -> C++ ATen -> 算子分发dispatcher -> CUDA 包装器函数wrapper  
-> <<<...>>> 启动 CUDA 内核 -> GPU 执行
```

以 `torch.add(x, y)` 算子为例

1. Py接口调用: Py调用的其实是在import torch里面的一个由Pybind11暴露给py的C++函数
2. Pybind 11和ATen: pybind11这个库是用来暴露的, ATen库里面定义了所有算子的签名(函数名, 参数, 类型), 并定义了dispatch (即根据不同的设备类型, 分发到不同的add_kernel去实现)
3. dispatcher--调度: pytorch的dispatcher会根据输入的张量和 (设备类型和数据类型) 决定到底调用哪一个kernel

4. 执行具体的CUDA实现：决定调用哪一个kernel之后，会调用相应的**kernel**和对应的**wrapper**（用来决定启用多少个**blocks&threads**）
 5. CUDA编译（nvcc）与硬件执行
 6. 返回结果
- Pytorch下降到Triton

```
Python -> torch.compile 捕获计算图 -> Inductor 优化与融合 -> 动态生成 Triton 源  
码 -> Triton JIT 编译器 -> 生成 PTX -> 启动 Triton 内核 -> GPU 执行
```

前三个步骤大致一样，只是在dispatcher这里分隔开了

1. 当你调用带有`@torch.compile`标记的函数的时候，`torch.compile`捕获计算图，然后`Inductor`会接收这个计算图并进行优化，然后动态生成源代码，然后这个源代码会被传递给Triton的JIT编译器，Triton Compiler会将其编译成高效的PTX代码，Inductor会创建一个调用这个内核的包装器（类似于CUDA路径的包装器），计算启动参数，然后使用Triton的运行时接口启动这个编译好的内核，然后被加载到GPU上面执行
- ThunderKittens--用于编写高性能AI内核的框架
 - [ThunderKittens: Simple, Fast, and Adorable AI Kernels](#)（真正用的时候再去读）
 - 矛盾：triton易于编写，可读性高，但优化较少，CUDA难以编写，代码复杂，但是性能很高
 - 优势：既能保持高性能，又易于使用，学习，和维护
 - 下降路径：**ThunderKittens (C++ eDSL) -> 编译为 -> Triton 或 CUDA -> 编译为 -> PTX -> GPU 执行**