机器学习系统 2026春

第三章 GPU架构与CUDA编程

张燕咏 讲席教授 yanyongz@ustc.edu.cn
张午阳 特任教授 wuyangz@ustc.edu.cn

中国科学技术大学
University of Science and Technology of China

为什么需要GPU计算？

- **摩尔定律放缓**
- 单核频率自2005年停滞在~4GHz（功耗墙）
- 晶体管仍在增加，但单线程性能增长<5%/年
- **并行计算的崛起**
- 解决方案：用更多简单核心代替更快的单核
- GPU：数千个核心，专为并行吞吐量设计
- **AI/ML驱动的需求**
- 大模型训练需要ExaFLOPS级算力
- GPT-4训练估计消耗~2×10^{25} FLOPs

单核性能 vs 并行性能趋势

单核性能：2005后趋于平缓

GPU并行性能：指数增长

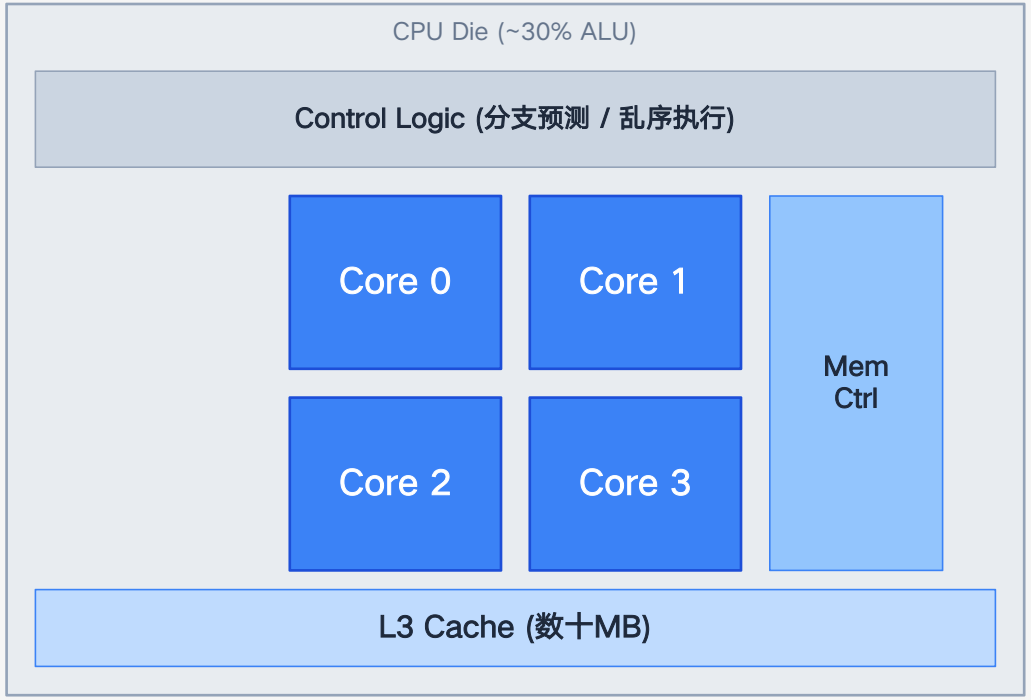
AI算力需求：每18个月翻4倍

1000 ×

H100 GPU vs 同代CPU的吞吐量提升
（深度学习推理，FP8精度）

CPU vs GPU：设计哲学

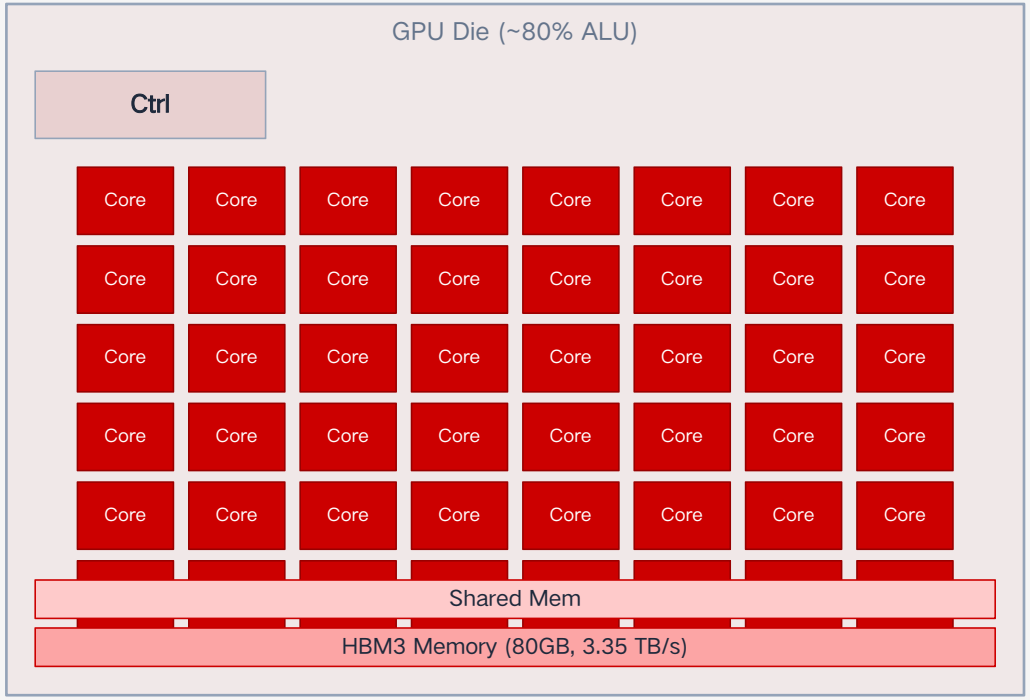
CPU — 延迟优化



少量高性能核心 · 大缓存 · 复杂控制 · ~5 GHz

适用：串行逻辑、操作系统、数据库

GPU — 吞吐量优化



数千简单核心 · 小缓存 · 简单控制 · ~2 GHz

适用：并行计算、深度学习、图形渲染

GPU计算格局 (2024-2025)

	NVIDIA H100	NVIDIA B200	AMD MI300X	Intel Gaudi 3
架构	Hopper (2022)	Blackwell (2024)	CDNA3 (2023)	Habana (2024)
晶体管	800亿	2080亿	1530亿	—
显存	80GB HBM3	192GB HBM3e	192GB HBM3	128GB HBM2e
带宽	3.35 TB/s	8 TB/s	5.3 TB/s	3.7 TB/s
FP16算力	990 TFLOPS	4.5 PFLOPS	1.3 PFLOPS	1.8 PFLOPS
FP8算力	1980 TFLOPS	9 PFLOPS	2.6 PFLOPS	—
互连	NVLink 4.0	NVLink 5.0	Infinity Fabric	RoCE/OFI
生态系统	CUDA (成熟)	CUDA (成熟)	ROCm (成长中)	oneAPI

NVIDIA凭借CUDA生态系统在AI/ML领域占据主导地位，但竞争格局正在变化

关注重点: 关注三个核心指标:
①显存容量(决定能装多大模型) ②内存带宽(推理和训练都依赖高带宽) ③Tensor Core算力(决定训练和推理吞吐量). CUDA生态系统是NVIDIA最大护城河

本章内容概览

01

GPU硬件基础

了解你的计算平台

02

CUDA编程模型

软件如何映射到硬件

03

第一个GPU核函数

写代码、跑性能、理解瓶颈

04

CUDA编程优化

从1%到90%峰值性能

05

现代GPU架构演进

每一代解决一个瓶颈

06

前沿研究

将原理应用于实际问题

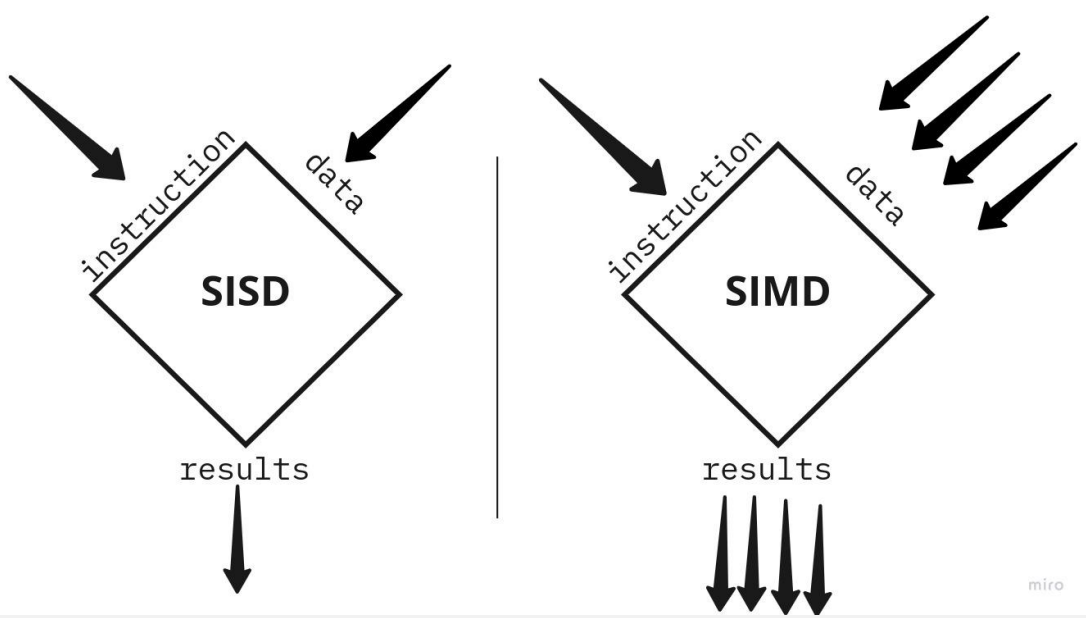
PART 01

GPU硬件基础

SIMD原理 · SM结构 · 内存层次 · 性能天花板

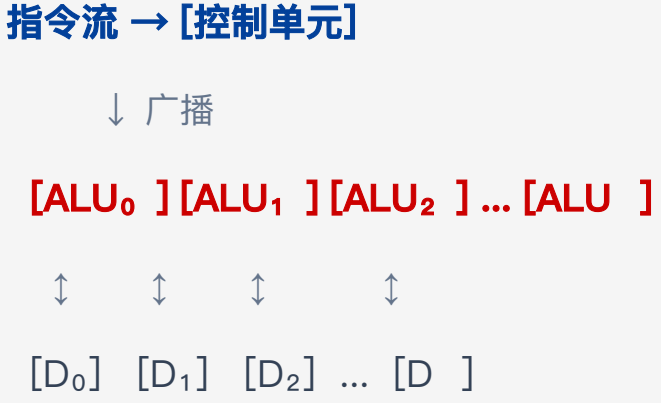
SIMD：单指令多数据

- SIMD (Single Instruction, Multiple Data)
- 一条指令同时对多个数据元素执行相同操作
- 与SISD（传统串行）相比：同一时钟周期处理N个数据
- SIMD处理器结构
- 控制单元广播同一条指令到多个ALU
- 每个ALU独立操作自己的数据
- 通过增加ALU数量线性扩展吞吐量
- Flynn分类法
- SISD → SIMD → MIMD → GPU (SIMT)
- GPU采用SIMT模型：线程可以有不同的执行路径（但会有代价）



SIMD执行模型详解

SIMD处理器示意图



- **关键特征**
- 所有ALU执行相同指令
- 每个ALU操作不同数据
- 通过增加ALU数量扩展并行度
- **性能公式**
- 吞吐量 = ALU数量 × 时钟频率
- GPU示例：H100有16896个CUDA核心
- **对比CPU**
- CPU: 高频率(~5GHz) × 少量核心(~64)
- GPU: 较低频率(~2GHz) × 大量核心(~16K)

关键要点: GPU本质是SIMD思想的极致——数千ALU执行同一指令。CPU用复杂控制换低延迟，GPU用简单控制换高吞吐

NVIDIA Hopper H100 架构总览

H100 SXM5 关键参数

制程: TSMC 4N　　晶体管: 800亿

SM数量: 132 (144设计, 132启用)

CUDA核心: 16,896 (FP32)

Tensor Core: 528 (4th gen)

显存: 80GB HBM3　带宽: 3.35 TB/s

L2 Cache: 50 MB

NVLink: 4.0 (900 GB/s)

TDP: 700W

FP8: 3,958 TFLOPS

4th Gen Tensor Core

FP8精度，6倍于A100芯片级性能

Transformer Engine

动态FP8/FP16精度切换，自动缩放

TMA

硬件加速的异步张量内存传输

Thread Block Cluster

跨SM协作的新编程层次

DPX指令

动态规划加速（基因组学等）40倍

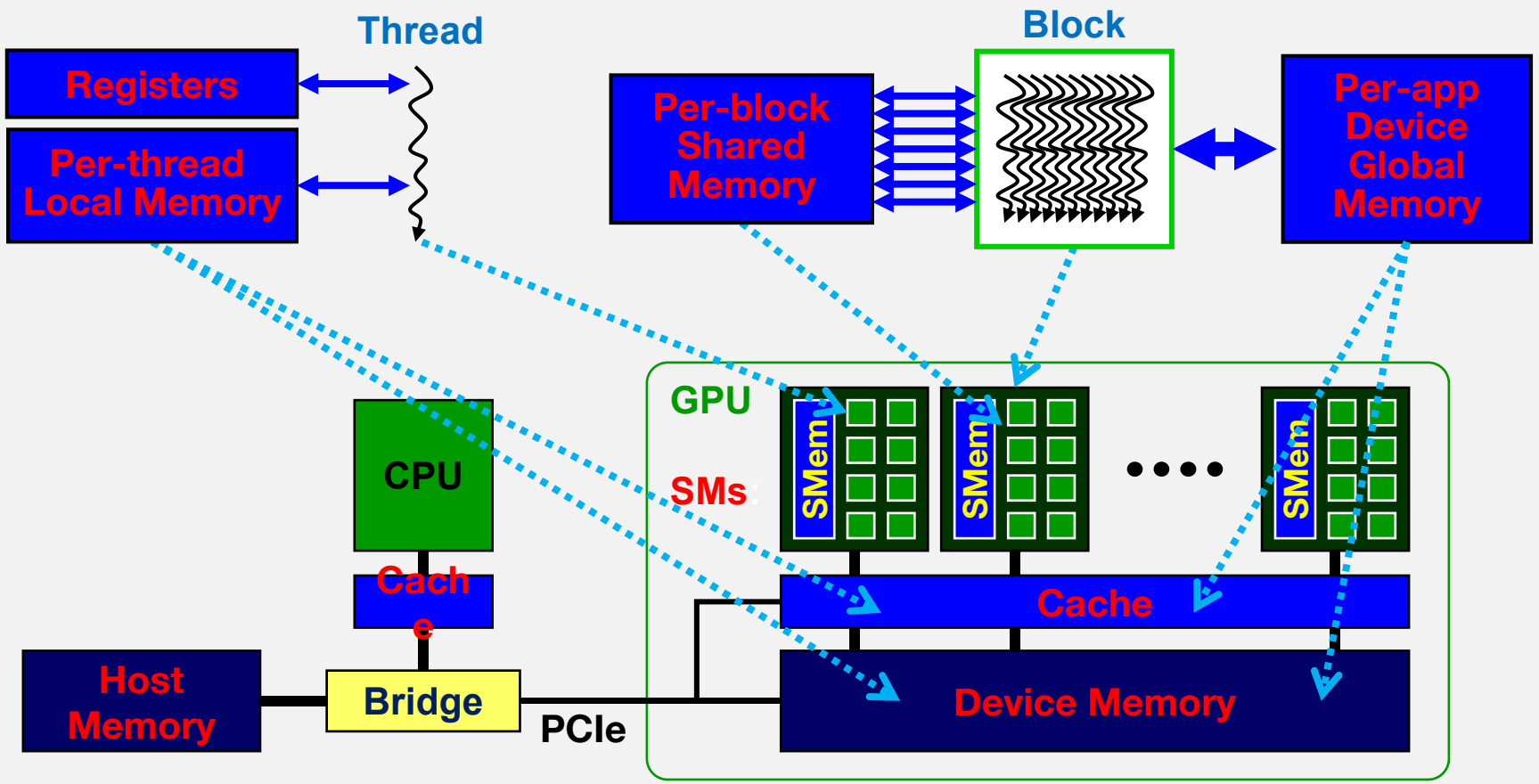
Streaming Multiprocessor (SM) 结构

- **计算单元**
- 128个FP32 CUDA核心
- 64个FP64核心（FP32的一半速率）
- 4个第四代Tensor Core
- 32个特殊函数单元(SFU: sin, cos, exp)
- **调度单元**
- 4个Warp调度器
- 每周期每调度器发射1条指令
- 最多64个并发Warp (2048线程)
- **存储单元**
- 256KB寄存器文件 (65536个32位寄存器)
- 256KB L1缓存+共享内存（可配置）
- 共享内存最大228KB
- 32个Load/Store单元

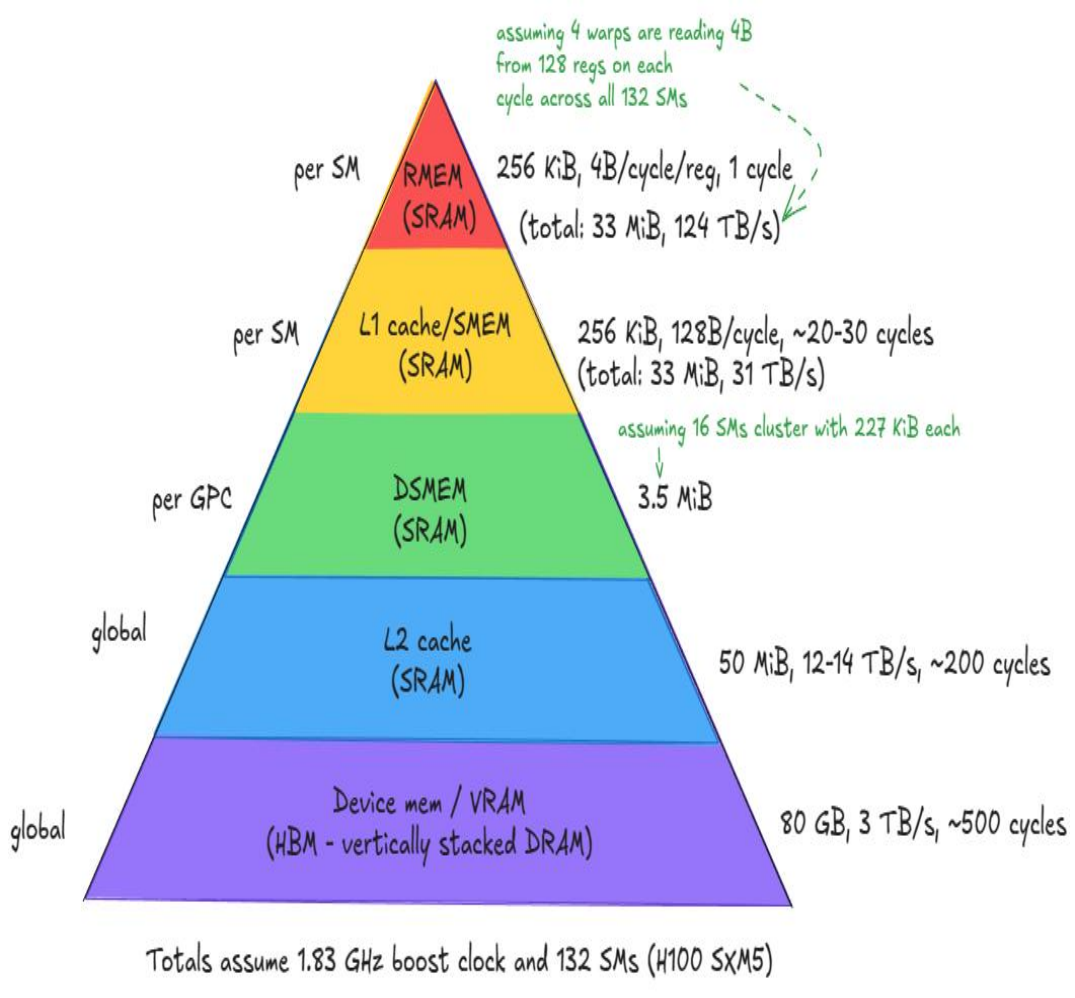
设计逻辑: SM是GPU核心计算单元。4个Warp Scheduler同时执行多个Warp来隐藏延迟; Tensor Core专为矩阵运算设计, 是AI算力核心

CUDA内存模型概述

float myVar; __shared__ float shVar; __device__ float glVar;



CUDA内存模型概述



- **寄存器 (Registers)**
- 每线程私有，最快，容量有限（255个/线程）
- **共享内存 (Shared Memory)**
- Block内共享，程序员管理的缓存
- H100: 每SM最大228KB
- **L2 Cache**
- 全GPU共享，自动管理
- H100: 50MB（可配置持久化区域）
- **全局内存 (Global/HBM)**
- 所有线程可访问，最大但最慢
- H100: 80GB HBM3, 3.35TB/s带宽
- **主机内存 (Host Memory)**
- CPU端，通过PCIe或NVLink-C2C访问

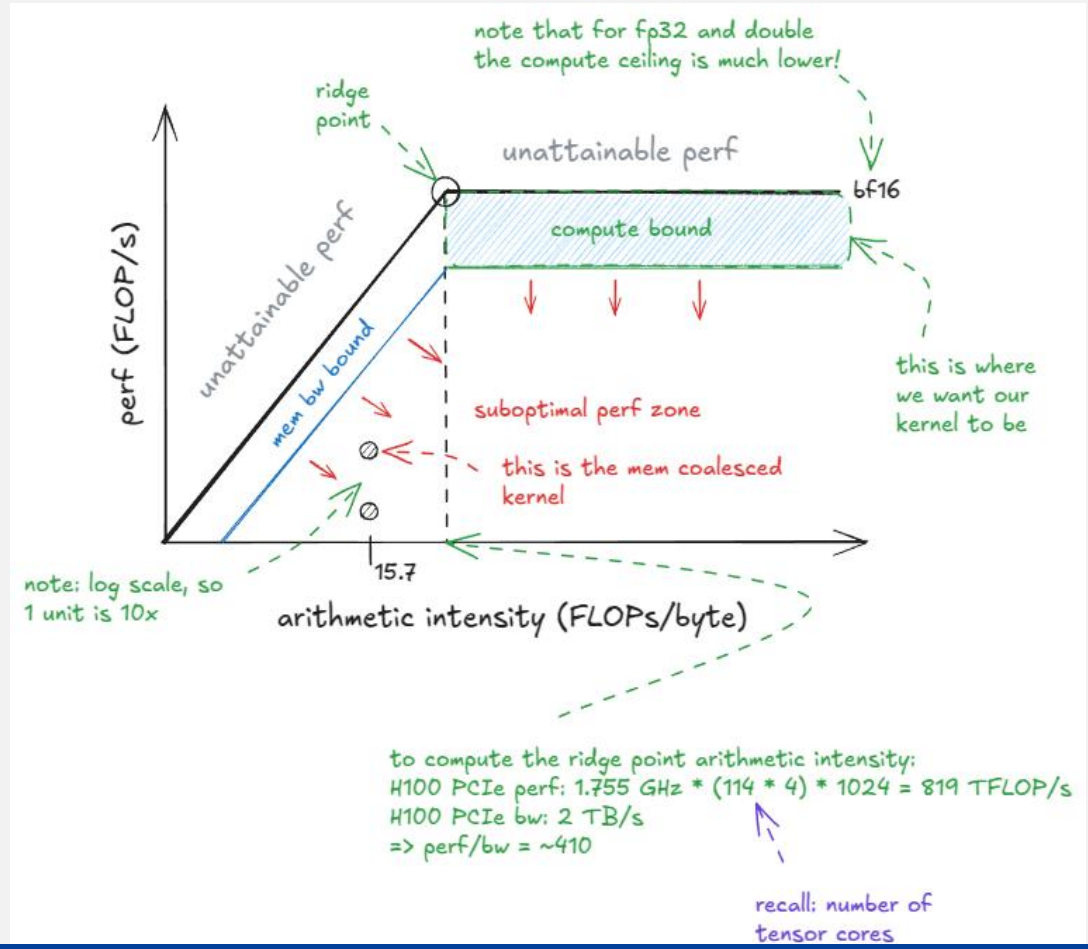
性能理论峰值

- **为什么要计算理论峰值？**
- 优化前必须知道天花板在哪里
- 实际性能 / 理论峰值 = 优化效率
- 效率 < 50% → 还有巨大优化空间
- **H100 SXM5 理论峰值**
- 频率: 1.83 GHz (boost)
- Tensor Core: 528个 (132 SM × 4)
- 每TC每周期: 256 FP16 FLOPs
- 理论峰值 = 1.83 × 528 × 256 = 247 TFLOPS
- 通过采用2：4稀疏模式以及其他优化, 峰值 上升至4倍，约 990 TFLOPS
- **优化效率**
- cuBLAS GEMM实测: ~900 TFLOPS (91%效率)
- 朴素kernel: ~20 TFLOPS (2%效率!)

关键要点: 优化前必须知道天花板：朴素kernel仅2%效率, 意味着还有50倍提升空间!

Roofline模型

- **核心概念**
- Arithmetic Intensity (AI) = FLOPs / Bytes
- 计算强度：每加载一个字节能做多少计算
- **两个瓶颈**
- 内存瓶颈: 性能 = AI × 内存带宽
- 计算瓶颈: 性能 = 峰值FLOPS
- 交叉点: Ridge Point（脊点）
- **H100 Ridge Point**
- FP32: 67 TFLOPS / 3.35 TB/s = 20 FLOP/B
- FP16 TC: 990 / 3.35 = 295 FLOP/B
- FP8 TC: 1979 / 3.35 = 590 FLOP/B
- FP8比FP16的屋顶变高了！
- **含义**
- AI < Ridge Point → 优化内存访问
- AI > Ridge Point → 优化计算效率



关键要点: AI < Ridge Point → 优化内存访问(共享内存, 合并访问); AI > Ridge Point → 用Tensor Core提升计算吞吐量

SM占用率 (Occupancy) 分析

- **占用率定义**
- 活跃Warp数 / SM最大Warp数
- H100: 最大64 Warp/SM
- **三大限制因素**
- **1. 寄存器数量**
- 每SM 65536个寄存器
- 每线程用64个 → 65536/64=1024线程=32 Warp → 50% 占用率
- **2. 共享内存**
- 每SM最大228KB
- 每Block用32KB → 228/32=7个Block
- **3. Block大小**
- 每SM最多32个Block
- blockSize=64 → 需要32个Block达到2048线程

占用率计算示例 (H100)

blockSize=256, regs/thread=40, smem=32KB

Reg: 65536/(40×256)=6.4→6 Block

Smem: 228/32=7.1→7 Block

Block: 32

Bottleneck=Reg: 6×256=1536 threads

Occupancy = 1536/2048 = 75%

优化建议

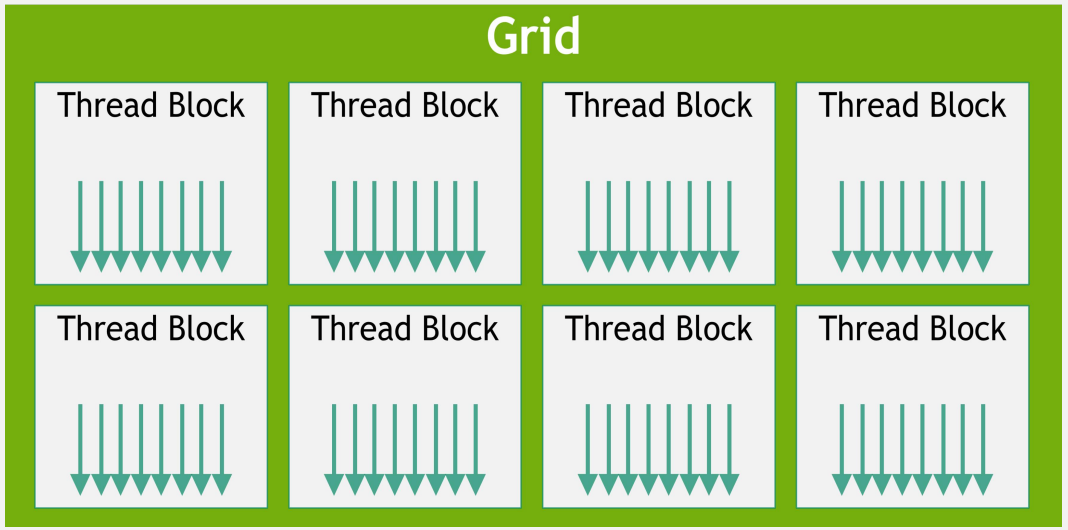
- --maxrregcount限制寄存器
- 调整blockSize和共享内存用量
- CUDA Occupancy Calculator辅助
- 50%+占用率通常足够

PART 02

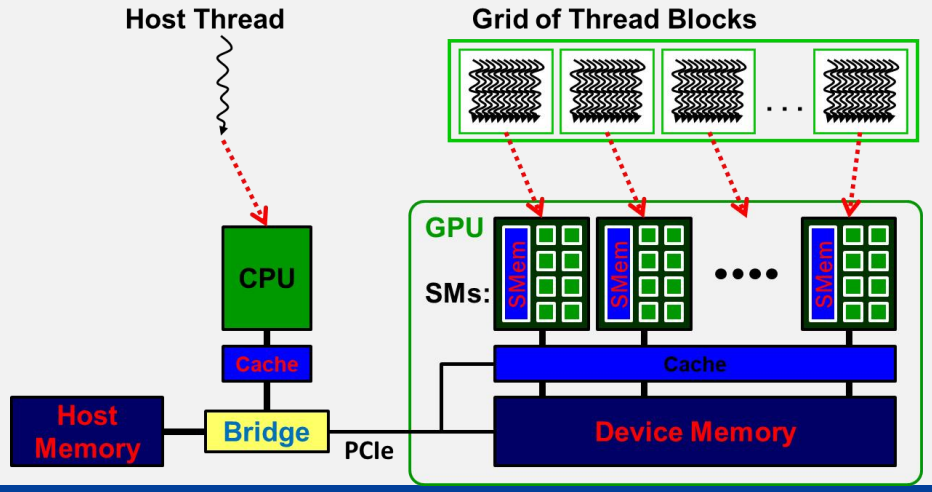
CUDA编程模型

软件如何映射到硬件 · 线程 · Warp · 核函数 · 同步机制

CUDA线程层次结构




```
kernel_func<<<nblk,nthread>>>(param, … );
```



- **Grid（网格）**
- 一个kernel启动产生一个Grid
- 由多个Block组成（1D/2D/3D）
- **Block（线程块）**
- Block内线程可以协作（共享内存、同步）
- 最多1024个线程/Block
- 映射到一个SM上执行
- **Thread（线程）**
- 最小执行单元
- 通过threadIdx + blockIdx计算全局ID
- **三维索引支持**
- gridDim, blockDim: 维度大小
- blockIdx, threadIdx: 当前索引

设计逻辑: 为什么三层? Grid→可扩展到任意GPU; Block→SM内协作(共享内存+同步); Thread→最小执行单元。只有Grid则线程间无法高效通信

线程块到SM的映射

动态调度示意

Grid: [B0][B1][B2][B3][B4][B5]...

↓ 动态分配

SM_0: [B0] → [B4] → ...

SM_1: [B1] → [B5] → ...

SM_2: [B2] → [B6] → ...

SM_3: [B3] → [B7] → ...

- **调度机制**
- 硬件调度器将Block动态分配到SM
- Block执行完成后，新Block被调度到空闲SM
- Block之间无执行顺序保证
- **资源约束**
- 每个SM可同时容纳多个Block
- 受限于：共享内存、寄存器、线程数
- H100: 每SM最多32个Block或2048线程
- **可扩展性**
- 相同代码在不同GPU上自动扩展
- 更多SM → 更多Block并行 → 更高吞吐

关键要点: Block数量应远多于SM数量(H100有132个SM), 这样GPU才能灵活调度、隐藏延迟。Block大小通常选128或256

同步机制：为什么GPU需要特殊的同步？

为什么: GPU数千线程并行执行，没有全局同步机制。同步只能在Block内(__syncthreads)或Warp内(__syncwarp)进行

- **Block内同步**
- __syncthreads(): 等待Block内所有线程
- 共享内存读写前必须同步
- 错误使用 → 数据竞争(Race Condition)
- **Warp内同步**
- __syncwarp(): 更轻量，仅Warp内
- Volta+支持Independent Thread Scheduling
- **跨Block通信**
- GPU没有全局Barrier!
- 唯一方式: atomicAdd等原子操作 (非常低效)
- H100支持cluster-wide barrier

```
// Block-level sync
__shared__ float smem[256];
smem[tid] = input[gid];
__syncthreads();  // ← MUST sync!
float val = smem[tid ^ 1];

// Warp-level sync (lighter)
__syncwarp();

// Cross-block: only atomics
atomicAdd(&global_sum, local_sum);
```

关键要点: 3层同步: __syncthreads(Block) → __syncwarp(Warp) → atomicAdd(跨Block)。没有全局Barrier是GPU编程最大的约束之一

CUDA核函数语法

为什么：了解了线程层次后，让我们写第一个GPU程序。CUDA通过C++扩展语法<<<grid, block>>>将计算派发到GPU

```
// Host code
int main() {
    float *h_A, *d_A;
    int N = 1000000;

    // Allocate memory
    h_A = (float*)malloc(N * sizeof(float));
    cudaMalloc(&d_A, N * sizeof(float));

    // Copy to device
    cudaMemcpy(d_A, h_A, N*sizeof(float),
               cudaMemcpyHostToDevice);

    // Launch kernel
    int blockSize = 256;
    int gridSize = (N + 255) / 256;
    myKernel<<<gridSize, blockSize>>>(d_A, N);

    // Copy back to host
    cudaMemcpy(h_A, d_A, N*sizeof(float),
               cudaMemcpyDeviceToHost);
    cudaFree(d_A);
}
```

- **函数修饰符**
- __global__: GPU上执行，CPU调用
- __device__: GPU上执行，GPU调用
- __host__: CPU上执行，CPU调用
- **启动语法**
- kernel<<<gridDim, blockDim>>>(args)
- gridDim: Block数量 (1D/2D/3D)
- blockDim: 每Block线程数 (≤1024)
- **执行流程**
- 1. CPU分配GPU内存 (cudaMalloc)
- 2. 数据传输到GPU (cudaMemcpy)
- 3. 启动kernel (<<<>>>)
- 4. 结果传回CPU (cudaMemcpy)
- 5. 释放GPU内存 (cudaFree)

数据传输：cudaMemcpy


```
// Basic data transfer
float *h_data, *d_data;
int size = N * sizeof(float);

// Allocate device memory
cudaMalloc((void**)&d_data, size);

// Host -> Device
cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);

// Execute kernel...
myKernel<<<grid, block>>>(d_data, N);

// Device -> Host
cudaMemcpy(h_data, d_data, size, cudaMemcpyDeviceToHost);

// Async transfer (overlap with compute)
cudaStream_t stream;
cudaStreamCreate(&stream);
cudaMemcpyAsync(d_data, h_data, size,
                cudaMemcpyHostToDevice, stream);
myKernel<<<grid, block, 0, stream>>>(d_data, N);
cudaMemcpyAsync(h_data, d_data, size,
```

关键要点: GPU编程5步模式: cudaMalloc→cudaMemcpy(H→D)→Kernel→cudaMemcpy(D→H)→cudaFree. 使用Async版本+Stream可隐藏传输延迟

线程数量与边界检查


```
// Problem: N=1000, blockSize=256
// gridSize = (1000+255)/256 = 4
// total threads = 4*256 = 1024 > 1000

__global__ void kernel(float *A, int N) {
    int i = blockDim.x * blockIdx.x
          + threadIdx.x;

    // Bounds check required!
    if (i < N) {
        A[i] = A[i] * 2.0f;
    }
    // Out-of-range threads do nothing
}

// Launch: round up
int blockSize = 256;
int gridSize = (N + blockSize - 1)
             / blockSize;
kernel<<<gridSize, blockSize>>>(d_A, N);
```

- **为什么需要边界检查？**
- Grid大小必须是Block大小的整数倍
- 当N不能整除blockSize时，会有多余线程
- 多余线程访问非法内存会导致崩溃
- **Grid大小计算公式**
- gridSize = ⌈N / blockSize⌉
- = (N + blockSize - 1) / blockSize
- **常见blockSize选择**
- 128, 256, 512（Warp=32的倍数）
- 太小 → 低占用率
- 太大 → 寄存器压力增大
- 256是最常用的默认值

关键要点: Grid大小必须覆盖所有数据元素，边界检查(if语句)防止越界访问。blockSize通常选128/256/512(Warp=32的倍数)

线程索引计算


```
// 1D indexing
__global__ void vecAdd(
    float *A, float *B,
    float *C, int N) {
  int i = blockDim.x * blockIdx.x
          + threadIdx.x;
  if (i < N)
    C[i] = A[i] + B[i];
}

// 2D indexing
__global__ void matOp(
    float *M, int W, int H) {
  int col = blockDim.x * blockIdx.x
            + threadIdx.x;
  int row = blockDim.y * blockIdx.y
            + threadIdx.y;
  if (col < W && row < H) {
    int idx = row * W + col;
    M[idx] = M[idx] * 2.0f;
  }
}
```

核心公式

1D: i = blockDim.x × blockIdx.x + threadIdx.x

2D: col = blockDim.x × blockIdx.x + threadIdx.x

row = blockDim.y × blockIdx.y + threadIdx.y

idx = row × width + col

- **启动配置示例**
- dim3 threads(16, 16); // 256/block
- dim3 blocks((W+15)/16, (H+15)/16);
- **注意事项**
- 始终检查边界 (if i < N)
- blockDim应为32的倍数(Warp大小)
- 2D索引常用于图像/矩阵操作

关键要点: 全局线程ID = blockDim × blockIdx + threadIdx。1D用于向量, 2D用于矩阵/图像。始终检查边界(if i < N)

设备函数与函数修饰符


```
// __global__: CPU calls, GPU runs
__global__ void processData(
    float *data, int N) {
  int i = blockDim.x * blockIdx.x
          + threadIdx.x;
  if (i < N)
    data[i] = transform(data[i]);
}

// __device__: GPU calls, GPU runs
__device__ float transform(float x) {
  return __expf(x)
         / (1.0f + __expf(x));
}

// __host__ __device__: both
__host__ __device__
float clamp(float x, float lo,
            float hi) {
  return fminf(fmaxf(x, lo), hi);
}
```

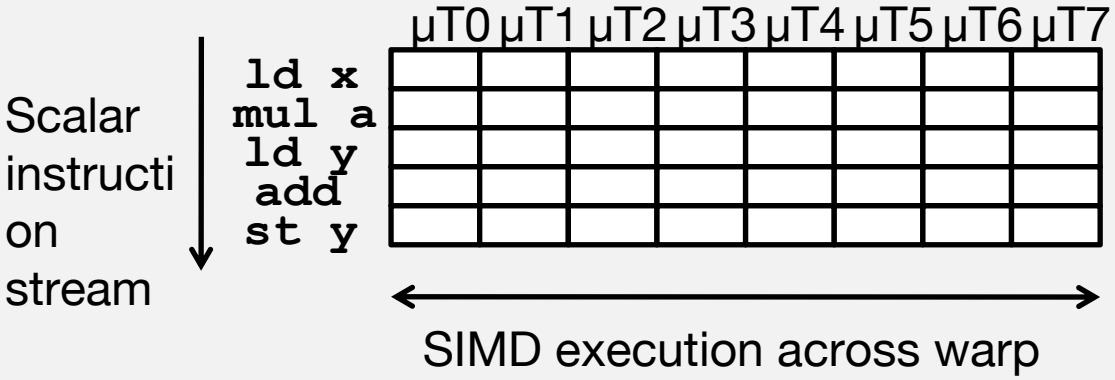
修饰符	执行位置	调用方	返回值
__global__	GPU	CPU	必须void
__device__	GPU	GPU	任意
__host__	CPU	CPU	任意
__host__ __device__	CPU+GPU	各自	任意

- **要点提示**
- __global__是kernel入口，必须返回void
- __device__函数通常被自动inline
- __host__ __device__编译两个版本
- 可用#ifdef __CUDA_ARCH__区分

关键要点: __global__是kernel入口(必须void), __device__是GPU辅助函数(常被inline), __host__ __device__编译两个版本供CPU/GPU共用

Warp

- GPUs 使用 SIMT模型, 每个CUDA线程的标量指令流汇聚在一起在硬件上以SIMD方式执行 (Nvidia groups 32 CUDA threads into a warp)



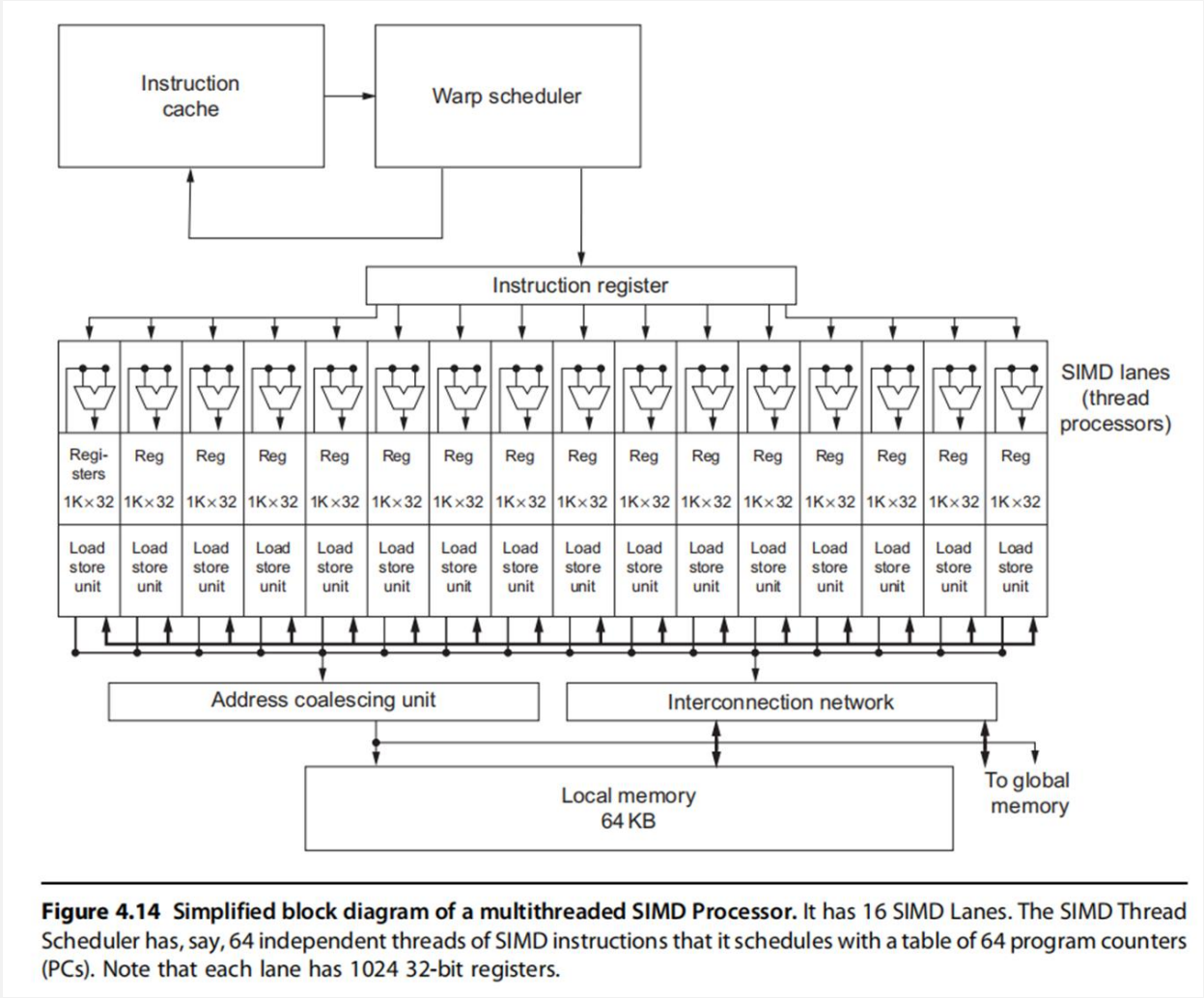


Figure 4.14 Simplified block diagram of a multithreaded SIMD Processor. It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

Warp：GPU执行的基本单位

- **Warp = 32个连续线程**
- Block内线程按32个一组分为Warp
- 例：256线程Block = 8个Warp
- Warp是实际硬件调度和执行的最小单位
- **SIMT执行**
- 同一Warp内的线程执行相同指令
- 每个时钟周期，Warp调度器选择一个就绪Warp
- 发射一条指令到Warp的32个线程
- **延迟隐藏**
- 当Warp等待数据（内存访问等待）
- 调度器切换到另一个就绪的Warp
- 零开销上下文切换（所有Warp状态常驻）
- 这就是为什么GPU需要大量线程

Warp调度器工作原理

Cycle 1: Warp0 → 执行算术指令

Cycle 2: Warp0 → 发起内存读取

Cycle 3: Warp0等待... → 切换Warp1

Cycle 4: Warp1 → 执行算术指令

Cycle 5: Warp1 → 发起内存读取

Cycle 6: Warp1等待... → 切换Warp2

...

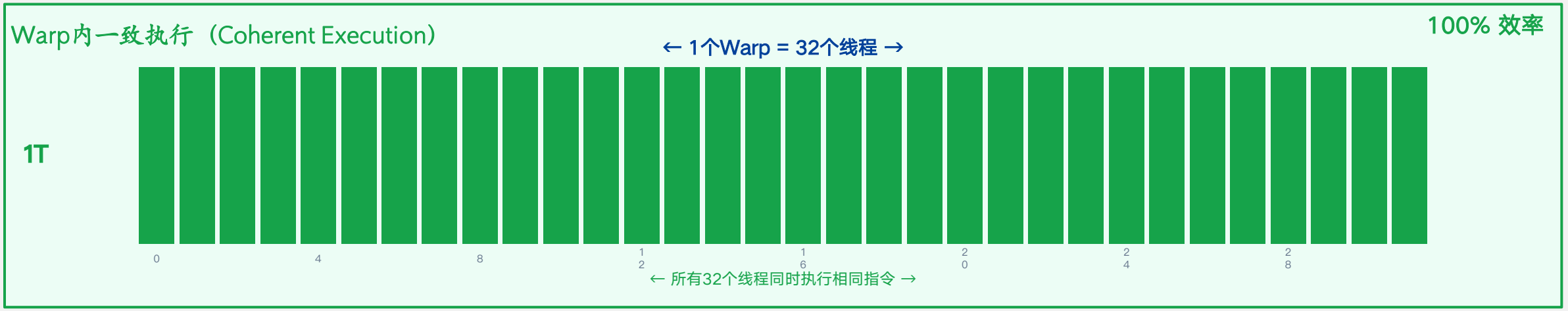
Cycle N: Warp0数据就绪 → 继续执行

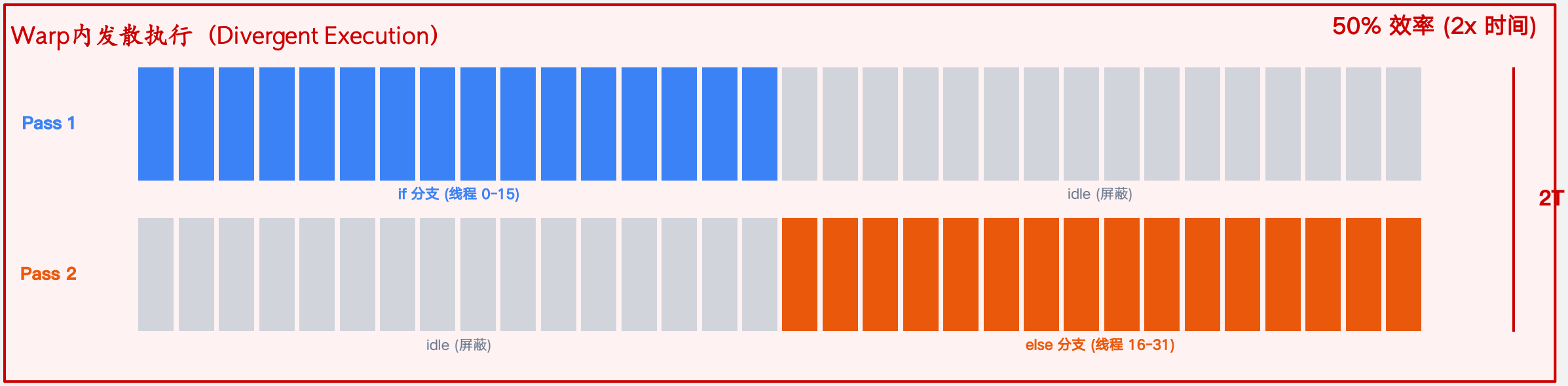
→ **ALU始终忙碌，内存延迟被隐藏**

延迟隐藏机制详解

- **延迟隐藏原理**
- Warp等待数据 → 调度器切换到另一Warp
- 切换开销为零（上下文常驻寄存器文件）
- 足够多的Warp → ALU永远不空闲
- **需要多少Warp?**
- 取决于：内存延迟 / 计算延迟
- 计算密集型：较少Warp即可
- 内存密集型：需要更多Warp
- **Little's Law**
- 并发量 = 吞吐率 × 延迟
- 要隐藏400 cycle延迟：
- 需要约400/32 ≈ 12个Warp（假设每指令1 cycle）
- **实际考虑**
- 不是越多越好：Warp多→每Warp寄存器少
- 最优warp数目需要profiling确定

一致执行 vs 发散执行





Warp是GPU真正的执行单位。不理解Warp → 不理解分支发散 → 不理解为什么if-else让GPU变慢50%.

分支发散 (Branch Divergence)

- **问题场景**
- Warp中32个线程必须执行相同指令
- 遇到if-else时：部分线程走if，部分走else
- 硬件处理：串行化两条路径
- **执行机制**
- Step 1: 所有线程执行if分支（else的线程被屏蔽）
- Step 2: 所有线程执行else分支（if的线程被屏蔽）
- Step 3: 合并继续执行
- **性能影响**
- 最坏情况：性能降低50%（一半线程idle）
- 嵌套分支更严重
- 不同Warp之间无影响

分支发散执行过程

```
if (threadIdx.x < 16) {
    A(); // 线程0–15执行
} else {
    B(); // 线程16–31执行
}
```

时间线:

T0: [0–15:A()] [16–31:idle]

T1: [0–15:idle] [16–31:B()]

T2: [all threads resume]

关键要点: Warp内if-else导致两条路径串行执行, 最坏情况性能降低50%。分支发散是Warp内问题, 不同Warp之间互不影响

避免分支发散的策略


```
// BAD: divergent — odd/even threads take different paths
__global__ void bad(float *data, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i % 2 == 0)      // threads split within warp!
        data[i] *= 2.0f;
    else
        data[i] += 1.0f;
}

// GOOD: no divergence — branch at warp granularity
__global__ void good(float *data, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int warpId = i / 32;
    if (warpId % 2 == 0)  // entire warp takes same path
        data[i] *= 2.0f;
    else
        data[i] += 1.0f;
}

// Tip: replace branches with math
// BAD:  if (x > 0) y = x; else y = 0;
// GOOD: y = fmaxf(x, 0.0f);  // ReLU without branch
```

- **策略一：Warp粒度分支**
- BAD: i%2 → 同一Warp内奇偶线程走不同路径
- 每个Warp都发散，50%线程被屏蔽
- GOOD: warpId%2 → 整个Warp走同一路径
- 零发散，100%效率
- **策略二：数学运算替代条件**
- fmaxf(x, 0) 代替if(x>0)的ReLU
- 无分支指令，硬件直接执行
- **编译器优化**
- 简单分支可能被自动转为predication
- 复杂分支仍需手动优化

关键要点: 以Warp(32线程)为单位做分支，或用数学运算替代条件(如fmaxf代替ReLU的if)。编译器有时自动优化简单分支

PART 03

第一个GPU核函数

写代码 · 看性能 · 理解瓶颈

矩阵乘法：为什么重要？

- **深度学习中的矩阵乘法**
- 全连接层: Y = XW + b
- 注意力机制: QK^T, Attention×V
- 卷积: im2col后转为GEMM
- 占LLM推理计算的60-80%
- **GEMM: C = αAB + βC**
- A: M×K, B: K×N, C: M×N
- 计算量: 2MNK FLOPs
- 数据量: (MK + KN + MN) × sizeof
- **Arithmetic Intensity**
- AI = 2MNK / (2(MK+KN+MN)) ≈ M (当M=N=K)
- M=4096: AI≈4096 >> Ridge Point
- → 大矩阵是计算瓶颈，可充分利用GPU

GEMM优化层次

Level 0: 朴素实现

~1% 峰值性能

Level 1: 寄存器Tiling

~10% 峰值性能

Level 2: 共享内存Tiling

~30% 峰值性能

Level 3: Tensor Core + TMA

~90%+ 峰值性能

→ CUTLASS/cuBLAS已实现Level 3

矩阵乘法：朴素实现


```
// Each thread computes one element of C
__global__ void matmul_naive(
    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;
  }
}

// Launch
dim3 block(16, 16);  // 256 threads
dim3 grid((N+15)/16, (M+15)/16);
matmul_naive<<<grid, block>>>(
    d_A, d_B, d_C, M, N, K);
```

- **分析**
- 每线程: 读A的一行(K个) + B的一列(K个)
- 总全局内存读取: 2×M×N×K
- 计算: 2×M×N×K FLOPs
- → AI = 1 FLOP/Byte (极低!)
- **问题**
- 大量冗余读取：相邻线程重复读A的同一行
- B的列访问模式可能不连续
- 完全没有数据复用
- **性能**
- H100上约1-2% 峰值 (~20 TFLOPS FP32)
- 瓶颈: 全局内存带宽

关键要点: Arithmetic Intensity仅1 FLOP/Byte，完全内存瓶颈。相邻线程重复读A的同一行却毫无复用——这就是为什么只有2%效率

性能分析：为什么只有2%效率？

- **朴素kernel性能**
- H100实测: ~20 TFLOPS (FP32)
- 理论峰值: 990 TFLOPS (FP16 TC)
- 效率: ~2% → 巨大的优化空间!
- **瓶颈在哪里？**
- 每个线程读A的一行 + B的一列
- 相邻线程重复读相同数据
- Arithmetic Intensity = 1 FLOP/Byte (极低)
- → 完全是内存瓶颈!
- **关键问题**
- Q1: 什么是内存合并访问？
- Q2: 如何减少全局内存读取？
- Q3: 如何提高计算/访存比？

性能差距

朴素实现: ~20 TFLOPS (2%)

共享内存优化: ~200 TFLOPS (20%)

Tensor Core: ~900 TFLOPS (91%)

接下来我们逐步解答:

1. 内存合并访问 → 理解DRAM物理结构
2. 共享内存 → 数据复用减少读取
3. 算术强度 → Roofline模型指导优化
4. Tensor Core → 硬件加速矩阵运算

内存合并访问：一个操作符的13倍差距


```
// Version A: Coalesced (fast)
// row = blockIdx.y*blockDim.y + threadIdx.y
// col = blockIdx.x*blockDim.x + threadIdx.x
sum += A[row*K + k] * B[k*N + col];  // ★ 连续访问 → 1次事务
// Thread 0 reads B[k*N+0]
// Thread 1 reads B[k*N+1]  <- adjacent!
// → Coalesced: one memory transaction

// Version B: Uncoalesced (13x slower!)
sum += A[col*K + k] * B[k*N + row];  // ✗ 散乱访问 → 32次事务
// Thread 0 reads B[k*N+0]
// Thread 1 reads B[k*N+32] <- scattered!
// → Uncoalesced: 32 memory transactions
```

- **仅交换row和col → 13倍性能下降!**
- Version A: 3,171 GFLOP/s
- Version B: 243 GFLOP/s
- **为什么?**
- GPU内存以128字节为单位读取
- 32个线程同时访问 → 地址连续最优
- 连续 = 1次内存事务 (coalesced)
- 散乱 = 32次内存事务 (uncoalesced)
- **启示**
- 不理解硬件 → 无法解释性能差异
- 这就是为什么需要学习内存模型!

关键要点: 仅交换row与col（一个操作符的差异）就导致13倍性能差距。根因: 连续线程是否访问连续地址，决定了1次vs32次内存事务

设备内存详解 (H100参数)

内存类型	作用域	生命周期	容量 (H100)	延迟	带宽
寄存器	单线程	线程	255个/线程, 256KB/SM	~1 cycle	~20 TB/s
共享内存	Block内	Block	最大228KB/SM	~20 cycles	~19 TB/s
L1 Cache	SM内	自动	与共享内存共用256KB	~30 cycles	~12 TB/s
L2 Cache	全GPU	自动	50 MB	~200 cycles	~6 TB/s
全局内存	全GPU	应用	80 GB HBM3	~400 cycles	3.35 TB/s
常量内存	全GPU	应用	64 KB + 缓存	~4 cycles*	广播
纹理内存	全GPU	应用	通过L1缓存	~100 cycles*	空间局部性

如何理解这些数字?

寄存器(1 cycle) vs 全局内存(400 cycles) = 400倍延迟差距! 选错内存层次 → 性能灾难

共享内存(20TB/s) vs HBM(3.35TB/s) = 6倍带宽差距 → 这就是为什么要用共享内存做数据复用

示例：1D卷积的共享内存优化


```
// Naive: each element re-reads global mem
__global__ void conv1d_naive(
    float *in, float *out,
    float *mask, int N, int R) {
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  float sum = 0;
  for (int j = -R; j <= R; j++)
    if (i+j>=0 && i+j<N)
      sum += in[i+j] * mask[j+R];
  out[i] = sum;
}

// Shared memory: load once, reuse many
__global__ void conv1d_shared(...) {
  extern __shared__ float s[];
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  s[threadIdx.x+R] = in[i]; // center
  if (threadIdx.x < R) {    // halo
    s[threadIdx.x] = in[i-R];
    s[threadIdx.x+blockDim.x+R] = in[i+blockDim.x];
  }
  __syncthreads();
  float sum = 0;
  for (int j = 0; j <= 2*R; j++)
    sum += s[threadIdx.x+j] * mask[j];
```

- **问题分析**
- 朴素版：每元素读(2R+1)次全局内存
- 相邻线程有大量重复读取
- **共享内存优化**
- Block协作加载数据到共享内存
- 每个元素只从全局内存读1次
- 后续计算从共享内存读取（快20倍）
- __syncthreads()
- Block内线程屏障同步
- 确保共享内存加载完成后再读取
- 仅在Block内有效
- **性能提升**
- 全局内存访问减少~(2R+1)倍

关键要点: 共享内存优化核心模式: ①协作加载到SMEM ②__syncthreads()同步 ③从SMEM计算。7倍内存访问减少!

内存合并访问详解

合并访问 vs 非合并访问

合并 (Coalesced):

Thread 0 → addr[0]

Thread 1 → addr[1]

Thread 2 → addr[2] → 1次128B事务

非合并 (Strided):

Thread 0 → addr[0]

Thread 1 → addr[16]

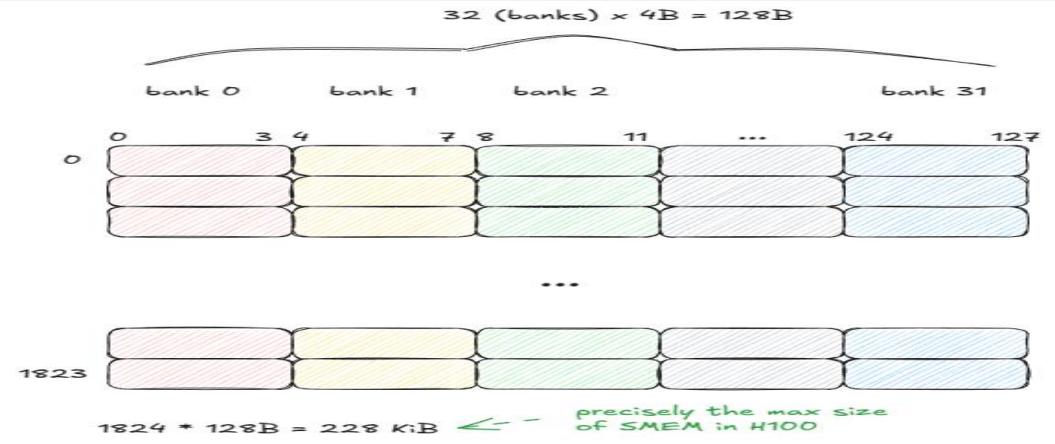
Thread 2 → addr[32] → 32次32B事务

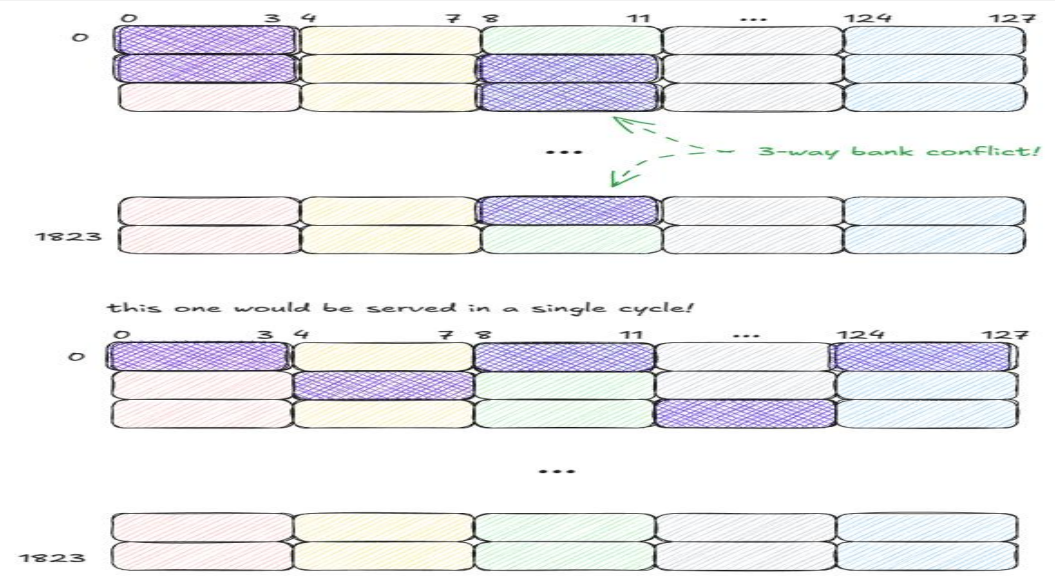
性能差距: 可达10-30倍

- **合并规则 (sm_60+)**
- Warp的32线程访问合并为32B对齐事务
- 连续地址 → 最少4次128B事务 (512B)
- 随机地址 → 最多32次32B事务 (1024B)
- **常见模式**
- 数组遍历: a[threadIdx.x] 合并
- 跨步访问: a[threadIdx.x * stride]
- 矩阵列访问: a[row * N + threadIdx.x]
- 矩阵行访问: a[threadIdx.x * N + col]
- **优化策略**
- 转置数据布局 (AoS → SoA)
- 使用共享内存做中间转置
- 利用L1/L2缓存缓解非合并影响

共享内存Bank冲突

为什么: 共享内存是GPU优化的关键工具，但如果访问模式不对，性能反而更差。Bank冲突可导致32倍性能下降!





- **共享内存组织**
- 32个Bank，每个Bank 32位宽（4字节）
- 一个周期可提供 32×4 = 128字节
- **Bank冲突**
- 同一周期，不同线程访问同一Bank的不同地址
- N-way冲突 → 串行化为N次访问
- 最坏情况: 32-way → 32倍性能下降
- **避免方法**
- 让相邻线程访问相邻Bank
- 使用padding消除冲突
- Hopper: 使用swizzle模式
- **广播优化**
- 多线程读同一地址 → 自动广播（无冲突）

PART 04

CUDA编程优化

从2%到91%效率 · 共享内存Tiling · Tensor Core · 异步流水线

矩阵乘法：寄存器Tiling

- **思路**
- 每个线程计算V×V个C元素（而非1个）
- 加载A的V个元素和B的V个元素
- 计算V×V个输出
- **内存访问减少**
- 朴素: 2K次读取 → 1个输出
- Tiling: 2K次读取 → V^2个输出
- 全局内存访问减少V/2倍
- **V=4的例子**
- 每线程: 读A的4个元素 + B的4个元素
- 计算: 4×4 = 16个C元素
- 访问减少: 4/2 = 2倍
- **限制**
- V受寄存器数量限制
- V太大 → 寄存器溢出 → 性能骤降

寄存器Tiling示意 (V=4)

Thread计算C的4×4子矩阵:

A: 读取第row行的V个元素

$[a_0\ a_1\ a_2\ a_3]$ → 寄存器

B: 读取第col列的V个元素

$[b_0\ b_1\ b_2\ b_3]$ → 寄存器

C: 外积计算V×V输出

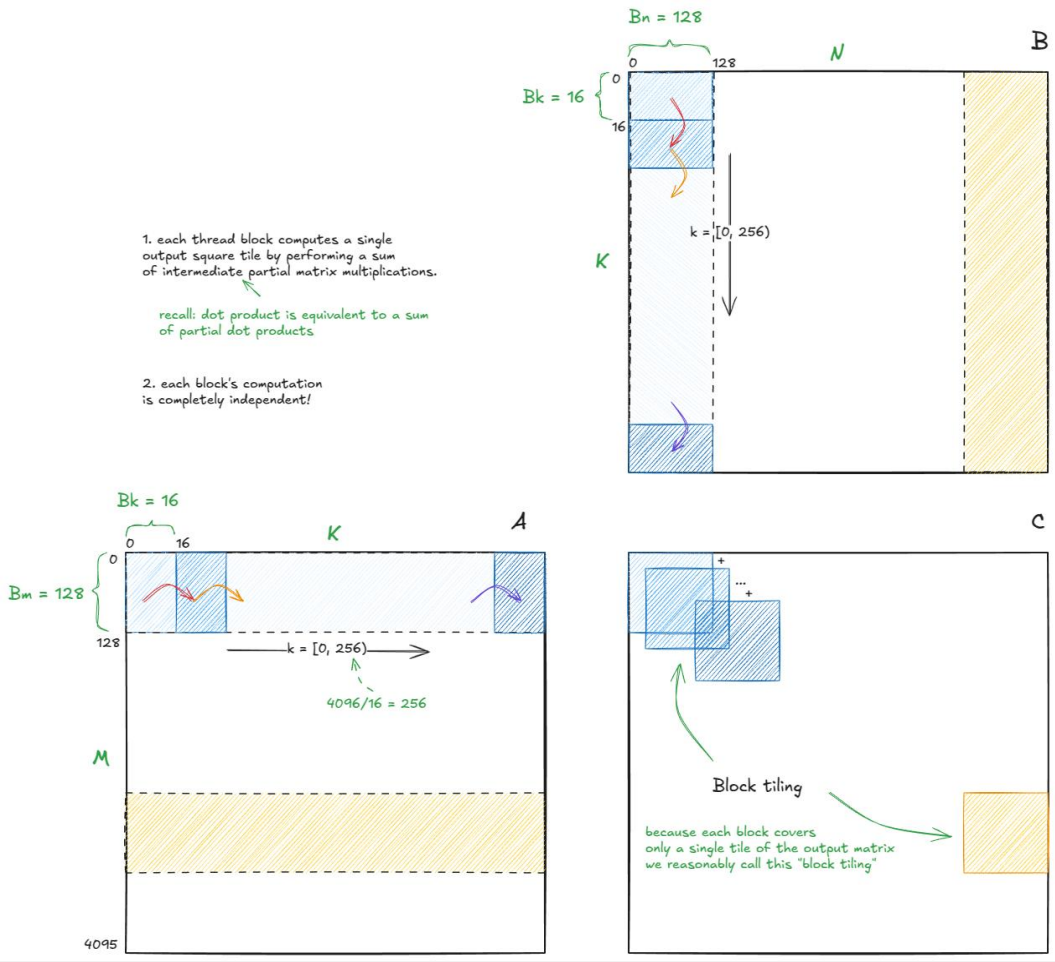
c[i][j] += a[i] * b[j]

4+4次读取 → 16次计算

全局内存: 2K/V 次读取per输出元素

矩阵乘法：共享内存Tiling

为什么：朴素kernel每个线程独立读全局内存，相邻线程重复读相同数据。共享内存让Block内线程协作加载，实现数据复用



- **优化原理**
- Block加载A和B的TILE×TILE块到共享内存
- Block内所有线程共享这些数据
- 每个全局内存元素被TILE个线程复用
- **性能分析**
- 全局内存读取: 2×M×N×K/TILE
- 比朴素减少TILE倍 (TILE=32 → 32倍)
- 共享内存延迟~20 cycles vs 全局~400
- **协作加载 (Cooperative Fetching)**
- 每线程加载1个元素到共享内存
- TILE×TILE个线程加载TILE×TILE个元素
- 全部加载完成后__syncthreads()
- **结合寄存器Tiling**
- 每线程从共享内存读取V行V列
- 计算V×V个输出 → 最佳性能

Tensor Core GEMM：硬件矩阵加速

为什么：CUDA Core做矩阵乘需要O(N³)独立FMA指令，Tensor Core用一条指令完成整个矩阵块运算

- **WMMA (Volta+)**
- 使用wmma::mma_sync执行16×16×16矩阵乘加
- 一条指令完成4096次FMA（vs CUDA Core的1次）
- 需要将数据按fragment格式组织
- **WGMMA (Hopper, sm_90)**
- Warp Group MMA: 4个Warp协作执行更大MMA
- 直接从共享内存消费数据（不需要先加载到寄存器）
- 减少寄存器压力，提高占用率
- **性能对比 (M=N=K=4096, FP16, H100)**
- CUDA Core: ~20 TFLOPS (2%峰值)
- 共享内存Tiling: ~200 TFLOPS (20%)
- Tensor Core (WMMA): ~700 TFLOPS (70%)
- → 硬件矩阵加速带来3.5倍提升

CUDA Core vs Tensor Core

CUDA Core (每周期):

1× FMA: a = b × c + d

4096个元素 → 4096条指令

Tensor Core (每周期):

D[16×16] = A[16×16] × B[16×16] + C[16×16]

4096个元素 → 1条指令

H100 FP16 峰值:

CUDA Core: 67 TFLOPS

Tensor Core: 990 TFLOPS (15倍!)

Hopper异步流水线：隐藏一切延迟

为什么: 即使用了Tensor Core (Level 3), 数据加载仍占30%时间。Hopper用硬件异步流水线让加载与计算完全重叠

- **TMA (Tensor Memory Accelerator)**
- 硬件自动完成地址计算和内存拷贝
- 释放Warp计算资源
- 支持2D/3D tensor切片
- **Producer-Consumer Pipeline**
- Producer Warp: 用TMA加载数据到SMEM
- Consumer Warp: 用WGMMA计算
- 多阶段流水线: 加载Stage N+1, 计算Stage N
- **效果**
- Level 3 → Level 4: 从~700到~900 TFLOPS
- 计算与内存访问完全重叠

异步流水线时序

Stage 0: [Load A_0,B_0] [Compute ----]

Stage 1: [Load A_1,B_1] [Compute A_0B_0]

Stage 2: [Load A_2,B_2] [Compute A_1B_1]

Stage 3: [----------] [Compute A_2B_2]

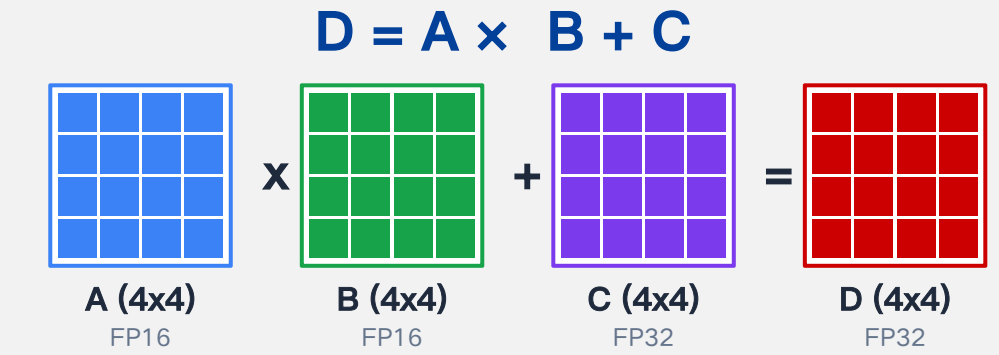
→ 加载延迟被计算完全隐藏!

传统: Load→Wait→Compute→Load→Wait→Compute

Pipeline: $Load_1$ // $Compute_0$ (永远不等待)

Tensor Core：矩阵加速引擎

- **基本操作**
- D = A × B + C (矩阵乘加)
- 每周期处理4×4×4 (Volta) 到 16×16×16 (Hopper)
- 一个Tensor Core一周期完成数百次乘加
- **CUDA核心 vs Tensor Core**
- CUDA Core: 每周期1次FMA
- Tensor Core: 每周期数百次FMA
- H100 FP16: CUDA 67 TFLOPS vs TC 990 TFLOPS
- 差距: ~15倍
- **支持精度**
- FP64, TF32, BF16, FP16, FP8, INT8
- 稀疏模式：2:4结构化稀疏，吞吐翻倍




```
// WMMA API (推荐)

wmma::fragment<...> a, b, c;

wmma::load_matrix_sync(a, ...);

wmma::mma_sync(c, a, b, c);

wmma::store_matrix_sync(...);


// 或使用CUTLASS/cuBLAS
```

CUTLASS：高性能矩阵运算库

- **什么是CUTLASS？**
- NVIDIA开源的C++ GEMM模板库
- 提供从朴素到极致优化的GEMM实现
- 支持Hopper TMA, WGMMA, Cluster
- cuBLAS的底层实现基础
- **CUTLASS 3.x架构**
- CuTe: 描述张量布局的DSL
- Kernel: TileScheduler + CollectiveMainloop + CollectiveEpilogue
- 自动选择最优tile大小和流水线深度
- **Epilogue Fusion**
- GEMM后直接做激活函数、Bias加法
- C = act(α×A×B + β×C + bias)
- 避免额外kernel launch和内存读写

```
// CUTLASS 3.x GEMM (simplified)
using Gemm = cutlass::gemm::device::GemmUniversalAdapter<
  cutlass::gemm::CollectiveBuilder<
    cutlass::arch::Sm90,     // Hopper
    cutlass::arch::OpClassTensorOp,
    cutlass::float_e4m3_t,   // FP8
    cutlass::layout::RowMajor,
    128,                     // Alignment
    cutlass::float_e4m3_t,
    cutlass::layout::ColumnMajor,
    128,
    float,                   // Accumulator
    Shape<_128, _256, _64>,  // Tile MNK
    Shape<_1, _2, _1>,       // Cluster
    cutlass::gemm::collective::StageCountAutoWithTma,
    cutlass::gemm::collective::KernelTmaWarpSpecialized
  >::CollectiveOp
>;

Gemm gemm_op;
gemm_op(args, workspace, stream);
```

Cooperative Groups


```
// Naive: shared memory + syncthreads
__shared__ float s[256];
s[tid] = val;
__syncthreads();
for (int i = 16; i > 0; i /= 2) {
  if (tid < i) s[tid] += s[tid+i];
  __syncthreads();
}
float sum = s[0];
```

```
// Cooperative Groups: shuffle, no smem
namespace cg = cooperative_groups;
auto warp = cg::tiled_partition<32>(
    cg::this_thread_block());
float sum = cg::reduce(warp, val,
    cg::plus<float>());
```

- **Warp-level Sum对比**
- Naive: 需要共享内存 + 多次__syncthreads
- CG: 一行cg::reduce，零共享内存
- **统一同步API**
- cg::this_thread_block() → Block同步
- cg::tiled_partition<N>() → 任意2^k子组
- cg::this_cluster() → Cluster同步 (Hopper)
- cg::coalesced_threads() → 活跃线程组
- **核心价值**
- 类型安全：编译期检查分组大小
- 可组合：任意嵌套分组
- 统一接口：sync/reduce/shfl全粒度一致

关键要点: Cooperative Groups统一了不同粒度的同步API。用tiled_partition+cg::reduce替代手写shared memory归约，更安全更简洁

Warp级原语


```
// __shfl_down_sync: 最重要的Warp原语
// 每个线程读取 lane+offset 的值
// 用途: Warp内归约 (求和/最大值)

float val = data[tid]; // 每线程一个值

// 5步完成32个元素求和!
val += __shfl_down_sync(0xFFFFFFFF, val, 16);
val += __shfl_down_sync(0xFFFFFFFF, val, 8);
val += __shfl_down_sync(0xFFFFFFFF, val, 4);
val += __shfl_down_sync(0xFFFFFFFF, val, 2);
val += __shfl_down_sync(0xFFFFFFFF, val, 1);
// lane 0 now holds sum of all 32 elements

// 为什么比共享内存快?
// → 直接寄存器到寄存器传输
// → 无需分配共享内存
// → 无需__syncthreads()
```

__shfl_down_sync 数据流

offset=16: [0+16] [1+17] ... [15+31]

offset=8: [0+8] [1+9] ... [7+15]

offset=4: [0+4] [1+5] [2+6] [3+7]

offset=2: [0+2] [1+3]

offset=1: [0+1] → lane 0 = 全部之和

- **Shuffle vs 共享内存**
- Shuffle: 寄存器直传，零额外延迟
- 共享内存: load→sync→store，~20 cycles
- Shuffle: 无Bank冲突风险
- **适用场景**
- Warp内归约（求和、最大值）

关键要点: 掌握__shfl_down_sync就能写出最优Warp归约——5步完成32元素求和，零共享内存，零同步开销

GEMM优化进度表：从朴素到SOTA

阶段	技术	性能	效率	关键改进
Level 0	朴素实现	~20 TFLOPS	~2%	每线程1元素,无数据复用
Level 1	寄存器Tiling	~60 TFLOPS	~6%	每线程V×V元素,提高AI
Level 2	共享内存Tiling	~200 TFLOPS	~20%	Block间数据复用
Level 3	Tensor Core	~700 TFLOPS	~70%	硬件矩阵加速
Level 4	异步流水线	~900 TFLOPS	~91%	TMA+WGMMA+Pipeline

每一步优化都针对一个特定瓶颈：数据复用 → 硬件加速 → 隐藏延迟

并行归约：树形方法

树形归约(8个元素求和)

Step 0: [1] [2] [3] [4] [5] [6] [7] [8]

\ / \ / \ / \ /

Step 1: [3] [7] [11] [15]

\ / \ /

Step 2: [10] [26]

\ /

Step 3: [36]

串行: O(N) 步

并行: $O(\log_2 N)$ 步 = 3步(N=8)

- **应用场景**
- 求和（向量点积、损失函数）
- 求最大/最小值（softmax, normalization）
- 计数（统计满足条件的元素）
- **CUDA实现挑战**
- CUDA没有全局同步原语
- 不能在一个kernel内跨Block同步
- 解决: 分两阶段kernel
- **两阶段方案**
- Kernel 1: 每Block归约→一个部分和
- Kernel 2: 归约所有部分和→最终结果
- 或: atomicAdd (简单但有竞争)

归约优化V1：交错寻址 (有分支发散)


```
// V1: interleaved addressing
for (int s = 1; s < blockDim.x; s *= 2) {
  if (tid % (2*s) == 0) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
// s=1: thread 0,2,4,6... active
//      thread 1,3,5,7... idle
// -> odd/even split within warp = DIVERGENT!

// V2: strided (no divergence)
for (int s = blockDim.x/2; s > 0; s >>= 1) {
  if (tid < s) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
// s=128: Warp 0–3 active, Warp 4–7 idle
// -> entire warp active or idle
// -> NO divergence!
```

- **V1问题: 分支发散**
- tid % (2*s) == 0 导致Warp内分叉
- 同一Warp中活跃/空闲线程交替
- 利用率仅50%
- **V2改进: 跨步寻址**
- 前半线程活跃，后半空闲
- 以Warp为单位：全活跃或全空闲
- 消除分支发散
- **性能对比**
- V1: Warp内50%线程空闲
- V2: 每步减半活跃Warp
- V2比V1快~1.5-2倍
- **还有问题**
- V2的内存访问模式不连续

问题: if(tid % (2*stride) == 0) 导致Warp内分支发散——奇数线程空闲，GPU利用率仅50%!

归约优化V2：顺序寻址 (合并内存访问)


```
// V2 issue: non-contiguous memory access
// when s=64: read sdata[0..63] + sdata[64..127]
// contiguous! but when s=1: only 2 elements

// V3: reduce during load (best coalescing)
int i = blockIdx.x * (blockDim.x*2)
        + threadIdx.x;
sdata[tid] = in[i] + in[i + blockDim.x];
__syncthreads();

for (int s = blockDim.x/2; s > 0; s >>= 1) {
  if (tid < s)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}

// Benefit: first load processes 2N elements
// only N/2 threads needed -> 2x efficiency
// while maintaining coalesced access
```

- **内存合并 (Coalescing)**
- 连续线程访问连续内存地址
- 硬件合并为最少的内存事务
- 32线程访问32个连续float → 1次128B事务
- **非合并的代价**
- 随机访问: 每线程1次事务 → 32次!
- 带宽利用率从100%降到~3%
- **V3: 加载时归约**
- 每线程加载2个元素并相加
- 线程数减半，处理元素不变
- 第一次全局内存读取的合并最重要
- **累计优化效果**
- V1 → V2: 消除分支发散 (~2x)
- V2 → V3: 加载时归约 (~2x)

关键要点: 加载时归约: 每线程加载2个元素并相加，线程数减半但处理量不变。第一次全局内存读取的合并效率最关键!

归约终极优化：Warp Shuffle


```
// Fully optimized reduction kernel
__global__ void reduce_optimized(
    float *in, float *out, int N) {
  __shared__ float sdata[32]; // 1 per warp
  int tid = threadIdx.x;
  int i = blockIdx.x*blockDim.x*2 + tid;

  // Grid-stride loop + reduce during load
  float sum = 0;
  for (; i < N; i += gridDim.x*blockDim.x*2)
    sum += in[i] + in[i + blockDim.x];

  // Warp-level reduction (no smem!)
  for (int off = 16; off > 0; off >>= 1)
    sum += __shfl_down_sync(0xFFFFFFFF,
                            sum, off);

  // Lane 0 of each warp -> smem
  if (tid % 32 == 0) sdata[tid/32] = sum;
  __syncthreads();

  // Final warp reduces partial sums
  if (tid < 32) {
    sum = (tid < blockDim.x/32)
          ? sdata[tid] : 0;
```

- **优化要点**
- 共享内存: TILE*sizeof(float) -> 仅32*sizeof(float)
- __syncthreads: log2(N)次 -> 仅1次
- **三阶段结构**
- 1. Grid-stride loop: 每线程加载2个元素累加
- 2. Warp内: __shfl_down_sync归约(零共享内存)
- 3. Warp间: 仅lane 0写共享内存, 最终Warp归约
- **为什么最优?**
- 合并内存访问(grid-stride)
- 寄存器直传(shuffle, 无bank冲突)
- 最少同步(仅1次__syncthreads)
- atomicAdd消除多阶段kernel

关键要点: __shfl_down_sync直接在寄存器间交换数据——无需共享内存，无需__syncthreads，最快的Warp内通信方式

归约优化总结

版本	技术	改进	关键原理
V1	交错寻址	基线	朴素树形归约
V2	跨步寻址	~2x	消除分支发散
V3	加载时归约	~2x	合并内存访问
V4	Warp Shuffle	~2x	消除共享内存同步
V5	Grid-stride loop	可扩展	处理任意大小
V6	atomicAdd	单kernel	消除多阶段

GPU优化核心原则总结

1. 消除Warp内分支发散 2. 合并全局内存访问 3. 利用共享内存做数据复用

4. 使用Warp级原语减少同步 5. 最大化占用率隐藏延迟 6. 使用Tensor Core加速矩阵运算

CUDA无全局同步：Kernel分解


```
// Stage 1: block -> partial sum
__global__ void reduce_stage1(
    float *in, float *partial, int N) {
  extern __shared__ float sdata[];
  int tid = threadIdx.x;
  int i = blockIdx.x*blockDim.x + tid;
  sdata[tid] = (i < N) ? in[i] : 0;
  __syncthreads();

  for (int s = blockDim.x/2; s > 0; s >>= 1) {
    if (tid < s) sdata[tid] += sdata[tid+s];
    __syncthreads();
  }
  if (tid == 0) partial[blockIdx.x] = sdata[0];
}

// Or use atomicAdd directly:
__global__ void reduce_atomic(
    float *in, float *result, int N) {
  __shared__ float sdata[256];
  int tid = threadIdx.x;
  int i = blockIdx.x*blockDim.x + tid;
  sdata[tid] = (i < N) ? in[i] : 0;
  __syncthreads();
  for (int s = blockDim.x/2; s > 0; s >>= 1) {
```

- **为什么需要Kernel分解？**
- CUDA没有全局同步原语
- 不能在一个kernel内跨Block同步
- Block之间执行顺序不确定
- **两阶段方案**
- Stage 1: 每Block归约 -> 一个部分和
- Stage 2: 归约所有部分和 -> 最终结果
- 隐式全局同步: kernel边界
- **atomicAdd方案**
- 单kernel: 每Block结果直接atomicAdd
- 简单但有竞争开销
- Block数少时性能可接受
- **根本约束**
- 这是GPU编程模型的设计选择

关键要点: CUDA无法在一个kernel内跨Block同步，因此大规模归约必须分两阶段或使用atomicAdd。这是GPU编程模型的根本约束

PART 05

现代GPU架构演进

每一代解决一个瓶颈 · Hopper · Blackwell · NVLink

GPU架构演进 (2012-2024)

为什么：每一代架构都在解决上一代的性能瓶颈，理解演进逻辑比记忆参数更重要

架构	年份	代表GPU	SM数	关键创新
Kepler	2012	GTX 680	8	动态并行, Hyper-Q
Maxwell	2014	GTX 980	16	能效优化, 共享内存改进
Pascal	2016	P100	56	HBM2, NVLink 1.0, 统一内存
Volta	2017	V100	80	Tensor Core (1st), 独立线程调度
Turing	2018	T4	40	RT Core, INT8推理
Ampere	2020	A100	108	TF32, 结构化稀疏, MIG
Hopper	2022	H100	132	FP8, TMA, Thread Block Cluster
Blackwell	2024	B200	—	FP4, 2nd Transformer Engine

每代架构都引入关键新特性，Tensor Core和混合精度是最重要的创新方向

H100 SM 深入分析

关键数字: 4个Tensor Core是AI算力核心, 228KB SMEM决定了FlashAttention的tile大小上限

组件	数量/容量	说明
FP32 CUDA Core	128	基础浮点运算
FP64 Core	64	双精度 (FP32的一半)
INT32 Core	64	整数运算
Tensor Core (4th gen)	4	矩阵MMA: FP8/FP16/BF16/TF32/FP64
SFU	32	超越函数 (sin/cos/exp/rsqrt)
LD/ST Unit	32	内存加载/存储
Warp Scheduler	4	每cycle每调度器发射1条指令
Register File	256 KB	65536个32位寄存器
L1 Cache + Shared Mem	256 KB	可配置: 共享内存最大228KB
Max Threads	2048	64 Warps
Max Blocks	32	

关键要点: 4个Tensor Core虽少但每个吞吐极高, 是AI计算的核心; 228KB共享内存决定了FlashAttention等算法的tile大小上限。vs A100: SMEM +39%, L2 +25%, 新增FP8支持

H100 内存子系统

H100 内存层次

寄存器: 256KB/SM × 132 SM

= 33 MB 总寄存器

L1+共享: 256KB/SM × 132 SM

= 33 MB 总L1/共享

L2 Cache: 50 MB (全GPU共享)

HBM3: 80 GB, 3.35 TB/s

总片上存储: ~116 MB

- **HBM3特性**
- 5个HBM3堆栈，5120-bit总线
- 3.35 TB/s带宽（比A100快1.6倍）
- ECC保护（数据中心级可靠性）
- **L2 Cache增强**
- 50MB（比A100多25%）
- 支持L2 Persistence: 可指定数据常驻L2
- cudaAccessPolicyWindow API控制
- **Inline Compression**
- 硬件透明压缩全局内存到SM的传输
- 有效带宽提升最多2倍
- 对程序员透明，无需代码修改
- **Compute-to-Memory比**
- FP16: 1979 TFLOPS / 3.35 TB/s
- = 590 FLOP/Byte → 计算密集型

Tensor Core 代际演进

趋势: 每代支持更低精度(FP16→FP8→FP4), 吞吐量翻倍, 推动大模型训练效率飞升

代	架构	年份	支持精度	关键特性
1st	Volta	2017	FP16	首次引入MMA
2nd	Turing	2018	FP16, INT8, INT4, INT1	推理优化
3rd	Ampere	2020	FP16, BF16, TF32, FP64, INT8	TF32自动加速, 稀疏
4th	Hopper	2022	FP8, FP16, BF16, TF32, FP64	FP8, Transformer Engine
5th	Blackwell	2024	FP4, FP6, FP8, FP16, BF16...	FP4, 2nd Transformer Engine

性能增长趋势 (FP16 Tensor TFLOPS)

V100: 125 → T4: 65 → A100: 312 → H100: 990 → B200: 4500

7年间FP16 Tensor性能增长约36倍

H100 Transformer Engine

- **为什么需要Transformer Engine?**
- FP8精度高但数值范围小，容易溢出
- 不同层、不同tensor需要不同精度
- 手动管理精度太复杂
- **工作原理**
- 每层自动选择FP8或FP16精度
- 根据tensor的统计特征动态决定
- 使用per-tensor scaling factor保持精度
- 软硬件协同：硬件支持+框架集成
- **FP8格式**
- E4M3: 4位指数+3位尾数（前向传播）
- E5M2: 5位指数+2位尾数（反向传播）
- 范围 vs 精度的权衡

精度动态切换流程

Input → Stats → Select Precision

↓

FP8 OK? → FP8 (2x throughput)

else → FP16 (safe precision)

↓

Output → next layer

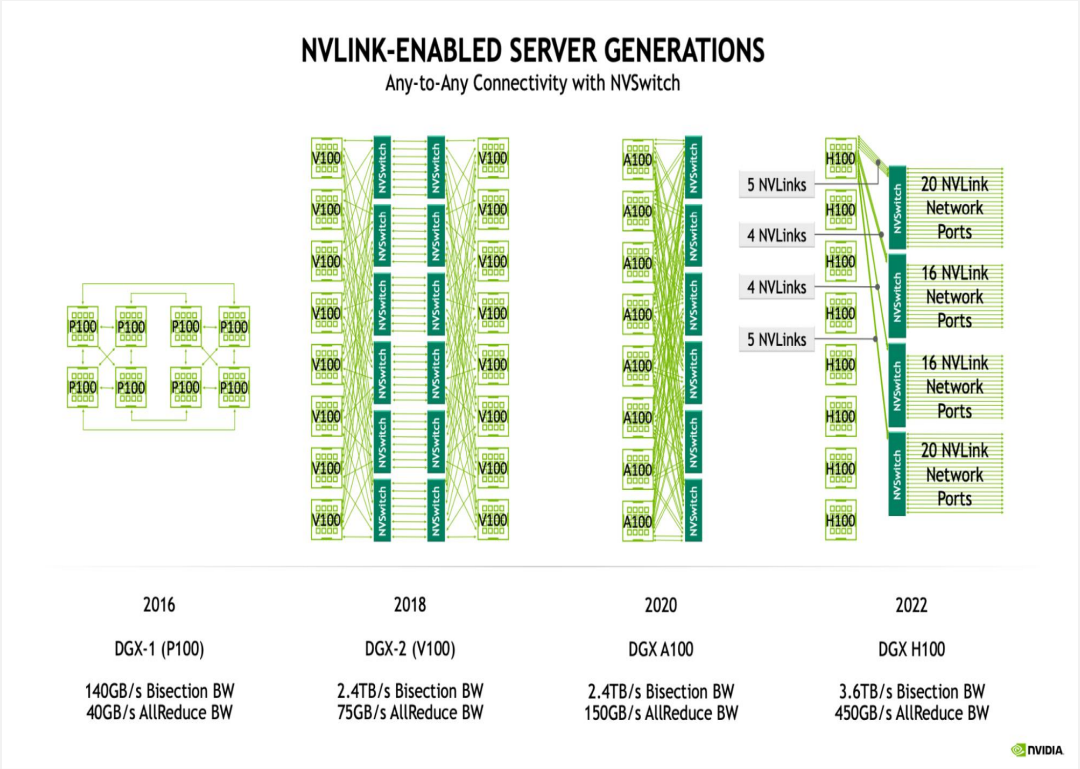
性能提升

- 训练: 比FP16快最高30% (无精度损失)
- 推理: FP8比FP16快2倍
- 框架支持: PyTorch, JAX, TensorRT
- 一行代码启用: with te.fp8_autocast()

NVLink 4.0 与 NVSwitch

为什么重要: 大模型训练必须多GPU协作, NVLink带宽(900GB/s)是PCIe的7倍, 直接决定张量并行的效率

- **NVLink 4.0 (Hopper)**
- 每GPU 18个NVLink链路
- 总带宽: 900 GB/s 双向
- 是PCIe 5.0 (128 GB/s) 的7倍
- 直接GPU-GPU内存访问
- **NVSwitch (3rd gen)**
- 全连接交换架构
- DGX H100: 8 GPU全互连
- 每GPU到其他任意GPU: 900 GB/s
- **NVLink Switch System**
- 跨节点NVLink扩展
- 最多256个H100全互连
- 总带宽: 57.6 TB/s
- 用于大模型训练集群



互连	带宽	典型用途
PCIe 5.0	128 GB/s	CPU-GPU
NVLink 4.0	900 GB/s	GPU-GPU (节点内)
NVLink-C2C	900 GB/s	CPU-GPU (Grace Hopper)
NVLink Switch	900 GB/s/GPU	GPU-GPU (跨节点)

NVIDIA Blackwell B200 架构

关键突破: 192GB显存可单卡装下LLaMA-70B全精度权重, H100(80GB)做不到

B200 关键参数

制程: TSMC 4NP

晶体管: 2080亿 (2.6倍H100)

显存: 192 GB HBM3e

带宽: 8 TB/s (2.4倍H100)

NVLink: 5.0 (1.8 TB/s, 2倍H100)

TDP: 1000W

FP4 Tensor: 20 PFLOPS

FP8 Tensor: 9 PFLOPS

FP16 Tensor: 4.5 PFLOPS

FP4 Tensor Core (5th gen)

4位浮点运算，20 PFLOPS，配合micro-scaling保持精度

2nd Gen Transformer Engine

支持MXFP4/MXFP6，与TensorRT-LLM深度集成

NVLink 5.0

1.8 TB/s/GPU，GB200 NVL72连接72个GPU

Decompression Engine

硬件数据库查询解压缩加速

Compute Capability sm_100

新指令集，WGMMA增强，更大Thread Block Cluster

实际影响: 192GB显存可单卡装下LLaMA-70B全精度权重, H100(80GB)做不到

Blackwell 关键创新详解

- **FP4精度与Micro-Scaling**
- 4位浮点: 1位符号 + 2位指数 + 1位尾数
- Micro-Scaling: 每组元素共享一个FP8 scale factor
- 配合FP32 global scale，有效保持模型精度
- 推理场景：LLaMA 3.1 405B从140GB(FP32)压缩到17.5GB(FP4)
- **GB200 NVL72 超级节点**
- 72个B200 GPU通过NVLink 5.0全互连
- 总GPU内存: 72 × 192GB = 13.8 TB
- 总NVLink带宽: 130 TB/s
- 可作为单一巨型加速器使用
- **LLM性能提升**
- 推理吞吐: 比H100快11-15倍/GPU
- 训练效率: 256个H100的工作量可用64个B200完成
- 能效比: 每token推理能耗降低25倍

Hopper vs Blackwell 对比

指标	H100 (Hopper)	B200 (Blackwell)	提升
制程	TSMC 4N	TSMC 4NP	—
晶体管	800亿	2080亿	2.6×
SM数量	132	—	—
显存	80 GB HBM3	192 GB HBM3e	2.4×
内存带宽	3.35 TB/s	8 TB/s	2.4×
FP16 Tensor	990 TFLOPS	4.5 PFLOPS	4.5×
FP8 Tensor	1,979 TFLOPS	9 PFLOPS	4.5×
FP4 Tensor	不支持	20 PFLOPS	新增
NVLink	900 GB/s	1.8 TB/s	2×
TDP	700W	1000W	1.4×
LLM推理	基准	11-15× /GPU	巨大

实际影响：这些数字意味着什么？

LLaMA-70B推理: H100 ~40 tokens/s → B200 ~440 tokens/s (11倍提升)

GPT-4级模型训练: H100集群需~90天 → B200集群预计仅需~15天。FP4精度是关键: 20 PFLOPS让推理成本大幅降低

Unified Memory：简化CPU-GPU编程

为什么：传统CUDA需要手动管理CPU/GPU内存拷贝。Unified Memory让CPU和GPU共享同一地址空间，Grace Hopper将其推向极致

- **Unified Memory优势**
- cudaMallocManaged: 一次分配,两端访问
- 页面级自动迁移(按需)
- 简化编程:无需手动cudaMemcpy
- **性能注意事项**
- 首次访问触发页面迁移(延迟)
- 频繁跨设备访问→性能下降
- cudaMemPrefetchAsync手动预取优化
- **Grace Hopper革命**
- NVLink-C2C: 900 GB/s CPU-GPU互连
- 硬件级缓存一致性
- 真正的统一内存,无页面迁移开销

```
// Traditional: manual transfers
float *h_data, *d_data;
cudaMalloc(&d_data, size);
cudaMemcpy(d_data, h_data, size,
           cudaMemcpyHostToDevice);
kernel<<<grid, block>>>(d_data);
cudaMemcpy(h_data, d_data, size,
           cudaMemcpyDeviceToHost);

// Unified Memory: automatic
float *data;
cudaMallocManaged(&data, size);
kernel<<<grid, block>>>(data);
cudaDeviceSynchronize();
// data accessible on both sides!

// Optimization hint
cudaMemPrefetchAsync(data, size,
    deviceId, stream);
```

Grace Hopper / Grace Blackwell 超级芯片

- **Grace Hopper Superchip**
- Grace CPU (72核 ARM Neoverse V2)
- + H100 GPU (132 SM)
- NVLink-C2C连接: 900 GB/s
- CPU内存: 512GB LPDDR5X (546 GB/s)
- GPU内存: 96GB HBM3 (3 TB/s)
- **革命性统一内存**
- CPU和GPU线程透明访问对方内存
- 硬件缓存一致性
- GPU可直接访问4倍于HBM的内存
- 消除了传统PCIe瓶颈

- **Grace Blackwell Superchip (GB200)**
- Grace CPU + 2× B200 GPU
- NVLink-C2C: 900 GB/s (每GPU)
- 总GPU内存: 384 GB HBM3e
- 总GPU带宽: 16 TB/s
- **NVLink-C2C vs PCIe**
- C2C带宽: 900 GB/s vs PCIe 5.0: 128 GB/s
- C2C延迟: 极低（片上互连级别）
- C2C功耗: 比PCIe低5倍
- **适用场景**
- 超大模型推理（需要大内存）
- 数据库加速（CPU-GPU协同）
- 科学计算（统一编程模型）

AMD MI300X 简介 (竞争格局)

- **MI300X 核心参数**
- CDNA3 架构, 8个计算芯粒 (chiplet)
- TSMC 5nm + 6nm
- 1530亿晶体管
- 显存: 192GB HBM3 (当时最大)
- 带宽: 5.3 TB/s
- Infinity Cache: 256 MB
- **性能特点**
- 大batch推理性能与H100相当
- 小batch延迟优势约40%
- 内存容量优势 (192GB vs H100 80GB)
- **挑战**
- ROCm生态不如CUDA成熟
- 部分框架支持尚不完善
- 市场份额仍较小

指标	MI300X	H100
显存	192 GB	80 GB
带宽	5.3 TB/s	3.35 TB/s
FP16	1.3 PFLOPS	990 TFLOPS
INT8	2.6 PFLOPS	1.98 PFLOPS
L2+Cache	256MB IF$	50MB L2
互连	Infinity Fabric	NVLink 4.0
编程	ROCm/HIP	CUDA

竞争加速了GPU创新，对整个行业有利

Thread Block Cluster 应用场景

- **GEMM优化中的Cluster**
- 传统: 每Block独立加载全局内存tile
- Cluster: 一个Block加载，其他Block通过分布式共享内存访问
- 减少全局内存带宽消耗（multicast loading）
- **TMA Multicast + Cluster**
- TMA支持将全局内存数据同时送到Cluster内多个Block
- 例: 2×2 Cluster加载4个tile只需2次全局内存读取（传统需要4次）
- 全局内存流量减少50%
- **FlashAttention-3中的应用**
- 利用Cluster实现高效的KV cache分布
- 不同Block处理不同attention head
- 通过分布式共享内存共享K/V数据
- **CUTLASS 3.x**
- NVIDIA的矩阵运算库已全面支持Cluster
- 自动利用分布式共享内存优化GEMM

Tensor Memory Accelerator (TMA)

解决的问题: 传统内存拷贝需要大量线程做地址计算, TMA用硬件自动完成, 释放计算资源

- **什么是TMA?**
- 专用硬件单元，加速张量数据传输
- 全局内存 ↔ 共享内存 (双向)
- 支持1D到5D张量寻址
- **核心优势**
- 单线程发起，释放整个Warp做计算
- 不占用SM计算资源（硬件offload）
- 支持多维张量直接传输（无需展平）
- 异步执行，可与计算重叠
- **Multicast模式**
- 一次全局内存读取 → 多个Block的共享内存
- 配合Thread Block Cluster使用
- 大幅减少内存带宽消耗

TMA vs 传统内存拷贝

```
// 传统：所有线程参与拷贝
// 32个线程各拷贝一部分
smem[tid] = gmem[base + tid];
__syncthreads();

// TMA：1个线程发起，其余做计算
if (threadIdx.x == 0) {
  cp.async.bulk.tensor.2d.shared
    ::cluster.global.mbarrier
    [smem], [gmem_desc], ...];
}
// 其他31个线程继续计算！
barrier.arrive_and_wait();
```

TMA 编程示例


```
// TMA descriptor (created on host)
CUtensorMap tensorMap;
cuTensorMapEncodeTiled(&tensorMap,
    CU_TENSOR_MAP_DATA_TYPE_FLOAT16,
    2,              // 2D tensor
    globalAddr,     // global memory addr
    globalDim,      // global dims {M, K}
    globalStrides,  // global strides
    boxDim,         // tile size
    elementStrides, // element strides
    CU_TENSOR_MAP_INTERLEAVE_NONE,
    CU_TENSOR_MAP_SWIZZLE_128B,
    CU_TENSOR_MAP_L2_PROMOTION_L2_128B,
    CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE);

// Using TMA in kernel
__global__ void gemm_tma(
  const __grid_constant__
    CUtensorMap tMap, ...) {
  extern __shared__ char smem[];
  auto bar = (cuda::barrier<...>*)smem;
  if (threadIdx.x == 0) {
    init(bar, blockDim.x);
    cp_async_bulk_tensor_2d_global_to_shared(
        smem + off, &tMap, x, y, *bar);
  }
  __syncthreads();
  bar->arrive_and_wait();
}
```

- **Host端: 创建TMA描述符**
- cuTensorMapEncodeTiled 描述张量布局
- 指定数据类型、维度、步长
- boxDim: 每次传输的tile大小
- Swizzle模式自动避免Bank冲突
- **Kernel端: 单线程发起传输**
- __grid_constant__: 描述符通过常量内存传入
- 仅线程0发起TMA操作
- 其余线程可同时执行计算
- Barrier同步等待数据就绪
- **vs 传统方式**
- 传统: 32线程各拷贝一部分数据
- TMA: 1线程 + 硬件自动搬运
- 释放31个线程做有用计算
- 自动处理边界条件(OOB fill)

Thread Block Cluster：解决跨SM数据共享瓶颈

解决的问题: 单个SM共享内存太小(228KB), Cluster让多个SM共享数据, 有效扩大SMEM容量

- **新的编程层次**
- Grid → Cluster → Block → Thread
- Cluster内Block保证在相邻SM上执行
- 支持跨Block的共享内存访问
- **分布式共享内存**
- Cluster内Block可读写其他Block的共享内存
- 延迟比全局内存低很多
- 适合需要跨Block协作的算法
- **Cluster大小**
- 可移植最大: 8个Block
- H100非可移植: 最多16个Block
- Blackwell: 最多16个Block (可移植)

```
// Compile-time cluster size
__cluster_dims__(2, 1, 1)
__global__ void kernel(...) {
  namespace cg = cooperative_groups;
  auto cluster = cg::this_cluster();

  // Access distributed shared memory
  extern __shared__ int smem[];

  // Access another block's shared memory
  int target_block = cyclic_next(cluster);
  int *remote_smem =
    cluster.map_shared_rank(smem, target_block);

  // Cluster-level sync
  cluster.sync();
}

// Runtime cluster config
cudaLaunchConfig_t config = {0};
cudaLaunchAttribute attrs[1];
attrs[0].id = cudaLaunchAttributeClusterDimension;
attrs[0].val.clusterDim = {2, 1, 1};
```

异步执行与流水线

Warp Specialization 流水线

Producer Warp (数据搬运):

[TMA Load S0][TMA Load S1][TMA Load S2]

Consumer Warp (计算):

[wait S0][WGMMA S0][wait S1][WGMMA S1]

→ 数据传输和计算完全重叠!

Stage 0 Stage 1 Stage 2 ...

[Load] [Load] [Load]

[Comp] [Comp] [Comp]

→ Ping-pong缓冲区交替使用

- **异步拷贝 (cp.async)**
- 全局内存→共享内存的异步传输
- 不需要中间寄存器（零拷贝语义）
- cp.async.bulk: TMA批量传输
- **Warp Specialization**
- 不同Warp承担不同角色
- Producer: 负责TMA数据加载
- Consumer: 负责Tensor Core计算
- 通过Barrier同步
- **多阶段流水线**
- 2-4个共享内存缓冲区
- Producer提前加载下一批数据
- Consumer处理当前批次数据
- 实现计算和内存完全重叠

CUDA Graphs

解决的问题: 小kernel的CPU启动开销(~5μs)占比过大, Graph将多个kernel打包一次提交

```
// Problem: small kernel launch overhead
// Each launch: ~5–10 us
// 100 small kernels: ~1 ms overhead!

// Solution: CUDA Graph capture & replay

// 1. Capture
cudaGraph_t graph;
cudaStreamBeginCapture(stream, ...);
  kernelA<<<g,b,0,stream>>>(...);
  kernelB<<<g,b,0,stream>>>(...);
  kernelC<<<g,b,0,stream>>>(...);
cudaStreamEndCapture(stream, &graph);

// 2. Instantiate (compile/optimize)
cudaGraphExec_t instance;
cudaGraphInstantiate(&instance, graph);

// 3. Replay (very low overhead)
for (int i = 0; i < 1000; i++) {
  cudaGraphLaunch(instance, stream);
}
// Replay overhead < 10 us (entire graph)
```

- **为什么需要CUDA Graphs?**
- kernel启动有CPU端开销（~5-10μs）
- 深度学习推理：大量小kernel
- CPU开销成为瓶颈
- **Graph的优势**
- 一次捕获，多次重放
- 减少CPU-GPU交互
- 编译时优化kernel依赖和调度
- 支持kernel间的依赖关系（DAG）
- **CUDA 12.x改进**
- 动态图更新（修改参数不需重建）
- 条件节点支持
- 与TMA和Cluster集成
- **PyTorch集成**
- torch.cuda.CUDAGraph()
- torch.compile自动使用

混合精度计算：FP32 → FP4

格式	位数	指数/尾数	范围	精度	H100 TFLOPS
FP32	32	8/23	$\pm 3.4\times10^{38}$	~7位有效数字	67
TF32	19	8/10	$\pm 3.4\times10^{38}$	~3位有效数字	990
BF16	16	8/7	$\pm 3.4\times10^{38}$	~2位有效数字	990
FP16	16	5/10	±65504	~3位有效数字	990
FP8 E4M3	8	4/3	±448	~1位有效数字	1979
FP8 E5M2	8	5/2	±57344	~0.5位有效数字	1979
FP4	4	2/1	±6	2个值	B200: 20000

趋势: 每降低一半精度，计算吞吐提升约2倍

关键: 通过Scaling Factor和Mixed Precision策略保持模型质量

实践: 训练用BF16/FP8混合，推理用FP8/FP4

实际影响: 从FP32到FP8, 计算吞吐量提升15倍(67→1979 TFLOPS), 是大模型训练的必备技术

FP8 训练与推理

- **FP8 训练**
- 前向传播: FP8 E4M3 (精度优先)
- 反向传播: FP8 E5M2 (范围优先)
- 权重主副本: FP32 (精度保持)
- Per-tensor Scaling Factor动态调整
- **Loss Scaling**
- 梯度值通常很小，FP8容易下溢
- 解决: 放大loss → 放大梯度 → FP8安全范围
- 更新前缩回原始尺度
- **框架支持**
- PyTorch: torch.float8_e4m3fn
- Transformer Engine: te.fp8_autocast()
- TensorRT-LLM: 自动FP8量化

```
# PyTorch Transformer Engine FP8
import transformer_engine.pytorch as te

# Training: one-line FP8 enable
model = te.Linear(768, 768)
with te.fp8_autocast():
    output = model(input)  # auto FP8

# Inference: TensorRT-LLM FP8
from tensorrt_llm import LLM
llm = LLM(model="llama-3-8b",
          quantization="fp8")
output = llm.generate("Hello")

# Manual FP8 (PyTorch 2.1+)
x_fp8 = x.to(torch.float8_e4m3fn)
scale = torch.tensor(1.0/absmax(x))
x_scaled = (x * scale).to(
    torch.float8_e4m3fn)
```

GPU性能分析工具

为什么: 没有profiling的优化是盲目的. Nsight Compute告诉你瓶颈在哪

- **Nsight Compute (kernel级)**
- 分析单个kernel的详细性能
- 内存吞吐量、计算吞吐量
- Warp执行效率、占用率
- Roofline分析
- **Nsight Systems (系统级)**
- 全系统性能时间线
- CPU-GPU交互分析
- kernel启动延迟
- 多GPU/多进程可视化
- **CUDA Occupancy Calculator**
- 计算理论占用率
- 识别资源瓶颈
- cudaOccupancyMaxActiveBlocksPerMultiprocessor()

```
# Nsight Compute (kernel-level)
ncu --set full ./my_app
ncu --kernel-name "gemm" ./my_app
ncu --metrics sm__throughput.avg_pct \
    ./my_app

# Nsight Systems (system-level)
nsys profile ./my_app
nsys profile --trace=cuda,nvtx ./my_app

# NVTX markers (in code)
#include <nvtx3/nvtx3.hpp>
nvtx3::scoped_range r("my_section");

# Python NVTX
import torch
with torch.cuda.nvtx.range("forward"):
    output = model(input)

# Occupancy API
int maxBlocks;
cudaOccupancyMaxActiveBlocksPerMultiprocessor(
    &maxBlocks, kernel, blockSize, sharedMem);
```

性能优化决策框架

- **Step 1: 确定瓶颈类型**
- 计算AI = 算法FLOPs / 数据传输Bytes
- 对比Ridge Point判断：内存瓶颈 or 计算瓶颈
- **Step 2: 内存瓶颈优化策略**
- 减少全局内存访问（共享内存缓存、数据复用）
- 改善内存访问模式（合并访问、避免Bank冲突）
- 使用TMA和异步拷贝（计算与传输重叠）
- 利用L2 Persistence提高缓存命中率
- **Step 3: 计算瓶颈优化策略**
- 使用Tensor Core（FP16/FP8/FP4矩阵运算）
- 降低精度（FP32 → TF32 → FP16 → FP8）
- 减少非矩阵计算（如FlashAttention减少softmax开销）
- 增加指令级并行度（循环展开、向量化）
- **Step 4: 验证**
- Nsight Compute profiling，确认瓶颈是否消除
- 注意：优化可能改变瓶颈类型

PART 06

前沿研究

将优化原理应用于实际问题 · FlashAttention · PagedAttention · Kernel Fusion

标准Attention的问题：$O(N^2)$内存瓶颈

解决的问题: 标准Attention需要$O(N^2)$内存存储中间矩阵, N=128K时需要64GB → 显存放不下!

- **标准Attention计算流程**
- S = Q × K^T → N×N矩阵 (N=序列长度)
- P = softmax(S) → 需要完整N×N矩阵
- O = P × V → 又一次矩阵乘
- **内存问题**
- 中间矩阵S和P: $O(N^2)$内存
- N=4096: S需要 $4096^2 \times 2B = 32MB$ (可接受)
- N=32K: 需要 2GB (勉强)
- N=128K: 需要 32GB → 超过GPU显存!
- **IO问题**
- S和P必须写入HBM再读回
- HBM读写成为瓶颈 (bandwidth-bound)
- AI = d (head dim, 64-128) → 远低于Ridge Point

序列长度 vs 内存需求

N=1K: 2 MB √ 轻松

N=4K: 32 MB √ 可以

N=16K: 512 MB 占用大量显存

N=32K: 2 GB 显存紧张

N=64K: 8 GB ✗ 单头已占满

N=128K: 32 GB ✗ 超出显存!

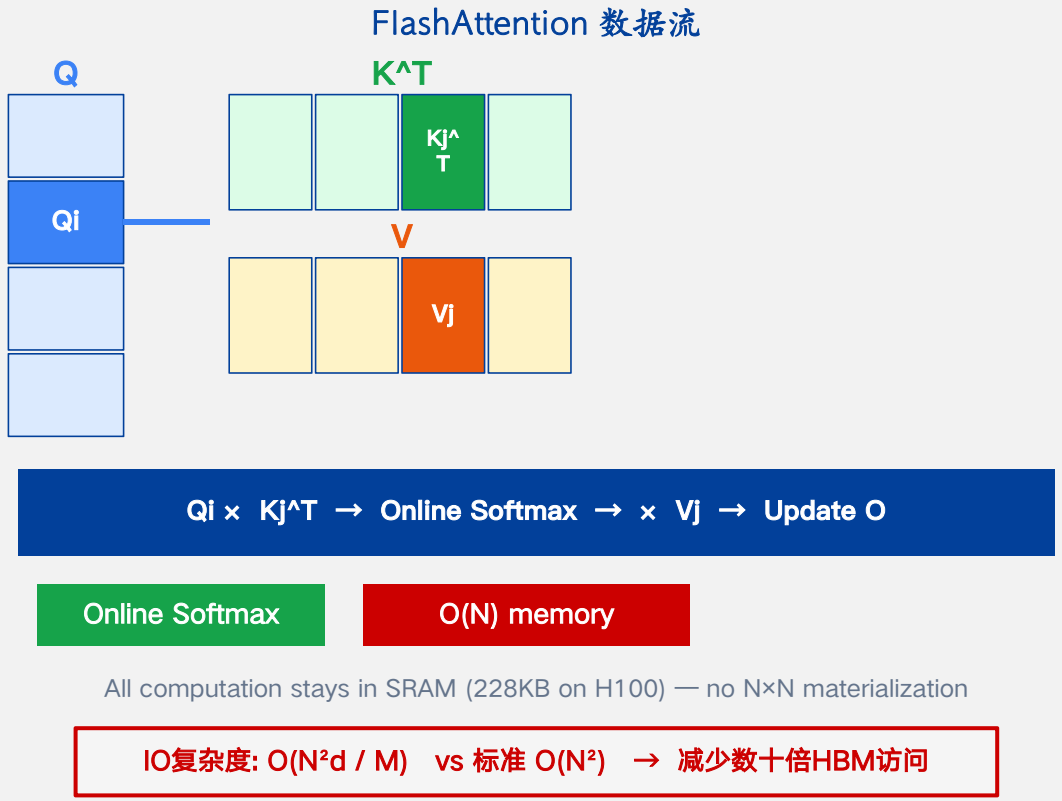
长序列是大模型的趋势

GPT-4: 128K, Claude: 200K, Gemini: 1M+

FlashAttention：分块 + Online Softmax

核心思想: 永远不materialize N×N矩阵, 分块计算 + online softmax, IO复杂度从O(N²)降到O(N²d/M)

- **分块计算**
- 将Q分成小块Qi, K/V分成小块Kj/Vj
- 每次只计算Qi×Kj^T的小块(在SRAM中)
- 无需存储完整N×N矩阵
- **Online Softmax**
- 传统softmax需要全局max和sum
- Online算法: 逐块更新max和sum
- 数学上等价, 但无需全局信息
- **IO复杂度分析**
- 标准: O(N²) HBM读写
- FlashAttention: O(N²d/M), M=SRAM大小
- H100 M=228KB → 减少数十倍HBM访问
- 计算量不变, 但IO大幅减少



FlashAttention-2 & 3

- **FlashAttention-2 (2023)**
- 减少非矩阵FLOPs（rescaling优化）
- 更好的Warp分区：减少通信
- Q循环在外层（减少HBM写入）
- 达到A100理论峰值的~73%
- 比FA1快2倍
- **FlashAttention-3 (2024, Hopper)**
- 利用Hopper硬件特性
- TMA异步加载Q/K/V tile
- Warp Specialization: producer + consumer
- WGMMA替代WMMA（更高效）
- FP8支持（block quantization降低误差2.6倍）
- 达到H100峰值的~75% (FA2仅35%)
- 比FA2在H100上快1.5-2倍

- **FlashAttention-4 (2025, Blackwell)**
- 首个突破1 PFLOP的Attention kernel
- 为sm_100 (Blackwell)专门优化
- 三次多项式近似exp2（避免SFU瓶颈）
- 同时计算两个指数值（双倍并行）
- 选择性rescaling: 仅在必要时rescale
- **FlashInfer (2025)**
- 可定制的CUDA模板 + JIT编译器
- 输入: Attention变体描述
- 输出: 优化的CUDA kernel
- 支持各种Attention模式
- **影响**
- 已成为所有LLM框架的标准组件
- PyTorch sdpa()默认使用FlashAttention

PagedAttention与vLLM

- **LLM推理的内存问题**
- KV Cache: 每token保存K和V向量
- 大小: 2 × layers × heads × head_dim × seq_len
- LLaMA-70B, seq=2048: ~5GB/请求
- 传统方法: 预分配最大长度 → 浪费60-80%
- **PagedAttention核心思想**
- 灵感: 操作系统虚拟内存分页
- KV Cache存储在非连续GPU内存块中
- 每Block固定大小（如16 tokens）
- Block Table记录虚拟→物理映射
- 动态分配/释放Block
- **关键好处**
- 内存浪费 < 4%（vs 60-80%）
- 支持更大batch size → 更高吞吐
- 支持Prefix Caching（共享前缀）

PagedAttention 内存管理

传统 (连续分配):

Req A: [████████████_____]

Req B: [█████████_____]

=使用 _=浪费(预分配)

PagedAttention (分页):

Block Table A: [B0→P5, B1→P2, B2→P7]

Block Table B: [B0→P1, B1→P4]

物理内存:

[P1:B][P2:A][___][P4:B][P5:A][___][P7:A]

→ 按需分配，几乎无浪费

→ Prefix共享: 多请求共用相同前缀Block

PagedAttention的影响

- vLLM (2023, UC Berkeley)
- 开源LLM推理引擎，实现PagedAttention
- 已成为最流行的LLM serving框架
- 支持: LLaMA, GPT, Mistral, Qwen等
- **性能提升**
- 吞吐量: 比HuggingFace高2-4倍
- 比TGI高2.2倍
- 内存效率: 支持更大batch
- Continuous Batching
- 传统: 等一批全完成再处理下批
- Continuous: 请求完成立即替换
- 配合PagedAttention动态管理KV Cache

Continuous Batching

传统 Static Batching:

t0: [A██████ B████__ C███___]

t1: [A██████ B████__ ____] ← C done, slot wasted

Continuous Batching:

t0: [A██████ B████__ C███___]

t1: [A██████ B████__ D██████] ← D fills slot!

其他LLM推理优化

- Speculative Decoding

  小模型预测 → 大模型并行验证 (2-3×加速)

- Prefix Caching

  共享系统提示的KV Cache

- Quantized KV Cache

  INT8/FP8压缩KV，减少内存占用

Kernel Fusion技术

- **为什么需要Kernel Fusion?**
- 每个kernel: 从HBM读 → 计算 → 写回HBM
- 连续kernel: 中间结果在HBM上往返
- element-wise操作（ReLU, Add）几乎纯内存操作
- 融合: 多个操作在一个kernel中完成
- **融合类型**
- Element-wise Fusion
- Y = ReLU(X*W + b) → 一个kernel
- Reduction Fusion
- LayerNorm: mean→var→normalize → 一个kernel
- Attention Fusion
- FlashAttention: QK^T→softmax→×V → 一个kernel

Unfused (3 kernels):

HBM–>[MatMul]–>HBM–>[Bias]–>HBM–>[GELU]–>HBM

6x HBM read/write

Fused (1 kernel):

HBM–>[MatMul+Bias+GELU]–>HBM

2x HBM read/write (3x less)

Auto Fusion Tools

- torch.compile: auto fusion
- TensorRT: NVIDIA inference optimizer
- Triton: Python DSL custom kernels
- XLA: Google compiler framework
- CUTLASS Epilogue: post-GEMM fusion

Triton：Python GPU编程

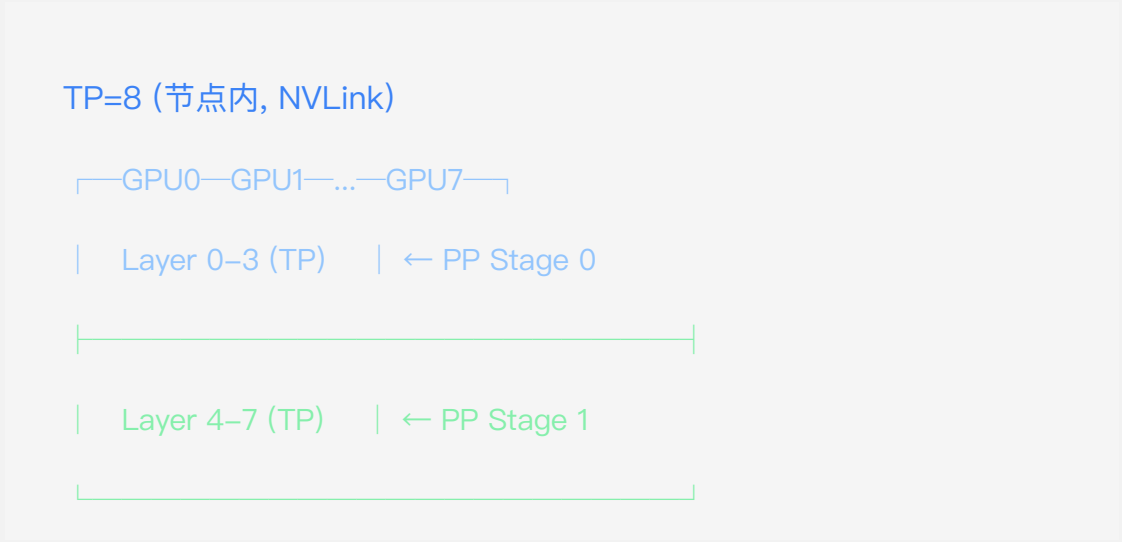

```
# Triton: Python DSL for GPU kernels
import triton
import triton.language as tl

@triton.jit
def fused_matmul_relu_kernel(
    A_ptr, B_ptr, C_ptr, M, N, K,
    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)

    acc = tl.zeros((BLOCK_M,BLOCK_N),
                   dtype=tl.float32)
    for k in range(0, K, BLOCK_K):
        offs_k = k + tl.arange(0, BLOCK_K)
        a = tl.load(A_ptr + offs_m[:,None]*K
                    + offs_k[None,:])
        b = tl.load(B_ptr + offs_k[:,None]*N
                    + offs_n[None,:])
        acc += tl.dot(a, b) # Tensor Core!

    acc = tl.maximum(acc, 0) # fused ReLU
    tl.store(C_ptr + offs_m[:,None]*N
             + offs_n[None,:], acc)
```

- **Triton vs CUDA**
- Python语法，无需C++/PTX
- 自动tiling和内存管理
- tl.dot自动使用Tensor Core
- **核心抽象**
- program_id: 类似blockIdx
- tl.arange: 向量化索引
- tl.load/store: 自动合并访问
- tl.dot: 矩阵乘自动Tensor Core
- **为什么重要**
- torch.compile后端生成Triton kernel
- 融合操作无额外HBM开销
- 性能接近手写CUDA (90%+)
- 开发效率提升10倍以上
- **生态**
- OpenAI开源，社区活跃
- 支持AMD GPU (ROCm)

多GPU并行策略（预览：后续章节详细讨论）

- **数据并行 (Data Parallelism)**
- 每GPU持有完整模型副本，处理不同数据batch
- 梯度通过AllReduce同步 (NVLink加速)
- 简单有效，但模型必须放得下单GPU
- **张量并行 (Tensor Parallelism)**
- 将单个layer的权重矩阵切分到多GPU
- 需要GPU间频繁通信 (AllReduce/AllGather)
- 适合节点内（NVLink高带宽），通常2-8路
- **流水线并行 (Pipeline Parallelism)**
- 将不同层分配到不同GPU
- GPU间只传递激活值（通信少）
- 存在流水线气泡（部分GPU空闲）

3D Parallelism 示意



实际部署: LLaMA-405B

- TP=8 (节点内NVLink)
- PP=16 (跨节点InfiniBand)
- DP=64 (数据并行)
- **总计: 8×16×64 = 8192 GPU**
- NVLink用于TP, IB用于PP和DP

高效LLM推理技术（预览）

- **Prefill vs Decode**
- Prefill: 处理输入prompt（计算密集）
- Decode: 逐token生成（内存密集）
- 两阶段特性完全不同
- **Prefill阶段**
- 大batch矩阵乘 → 计算瓶颈
- 可充分利用Tensor Core
- 延迟取决于prompt长度
- **Decode阶段**
- 每次只生成1个token
- 矩阵-向量乘 → 内存瓶颈
- KV Cache读取主导延迟
- batch越大GPU利用率越高

- **优化技术**
- **Continuous Batching**
- 请求完成立即替换，避免等待
- **Speculative Decoding**
- 小模型生成候选，大模型并行验证
- 加速2-3倍，保持输出质量
- **KV Cache优化**
- PagedAttention: 高效内存管理
- Quantized KV: FP8/INT8压缩
- Grouped Query Attention: 减少KV大小
- **Disaggregated Serving**
- Prefill和Decode用不同GPU集群
- 各自优化硬件配置
- 通过高速网络传递KV Cache

GPU编程的未来：超越CUDA

- **Triton (OpenAI, 2021-)**
- Python DSL，自动tile和调度
- torch.compile默认后端
- 大幅降低GPU编程门槛
- **torch.compile (PyTorch 2.0+)**
- 图捕获 + 自动优化 + Triton代码生成
- model = torch.compile(model) # 一行启用
- 自动kernel fusion, 内存优化
- **MLIR/XLA**
- 编译器基础设施，支持多种硬件后端
- JAX/TensorFlow使用XLA
- 统一IR实现跨平台优化
- **趋势与新兴方向**
- 从手写CUDA → 编译器自动优化，但理解底层仍然重要
- Chiplet架构（AMD MI300X 8芯粒, Blackwell多die）突破面积限制
- 光互连、存内计算(PIM)、AI驱动Kernel生成是未来方向

新兴趋势

- **Chiplet架构**
- AMD MI300X已采用8芯粒设计
- NVIDIA Blackwell也使用多die架构
- 突破单die面积限制，提高良率
- **光互连 (Optical Interconnect)**
- NVLink下一代可能采用光学连接
- 更高带宽、更低功耗、更远距离
- 解决大规模集群的通信瓶颈
- **存内计算 (Processing-in-Memory)**
- 在HBM内部集成计算逻辑
- 消除数据搬运瓶颈
- Samsung等厂商已有原型
- **AI驱动的Kernel生成**
- LLM生成优化CUDA代码（CUDA-LLM等）
- 强化学习优化kernel参数
- 自动化性能调优

第三章总结

01 GPU硬件基础

GPU vs CPU、SM结构、Warp执行、内存层次

02 CUDA编程模型

线程层次、核函数、内存管理、分支发散

03 第一个GPU核函数

向量加法、矩阵乘法、共享内存优化

04 优化之路

Roofline模型、合并访问、占用率、Profiling

05 现代GPU架构演进

Hopper H100、Blackwell B200、NVLink、TMA

06 前沿研究

FlashAttention、PagedAttention、Kernel Fusion、Triton