Weiming HPC Training Camp
×
LCPU AI Infra Seminars

北京大学未名超算队
北京大学学生 Linux 俱乐部

FP32 GEMM
Quick Walkthrough

Zhou Yuxuan

2026. 07. 29

Weiming Supercomputing Team
Linux Club of Peking University

矩阵乘法 GENERAL MATRIX MUTIPLICATION

- 计算：$C = A \times B$

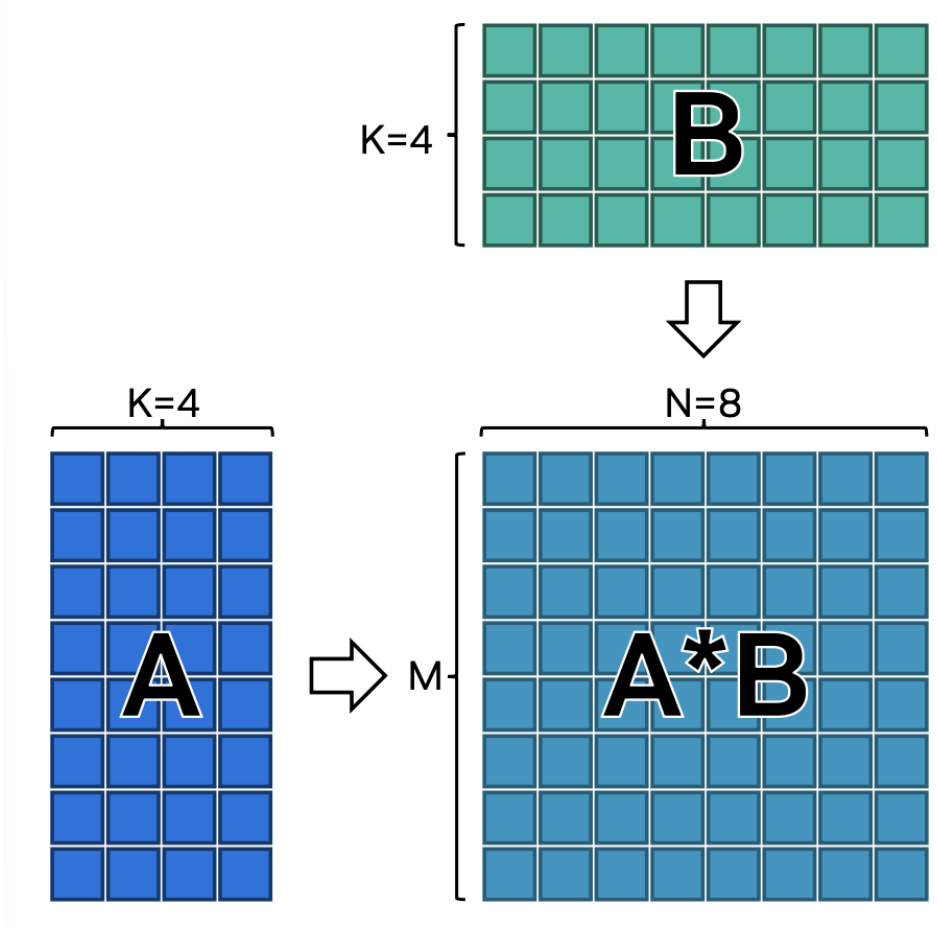
  $A \in \mathbb{R}^{M \times K}, B \in \mathbb{R}^{K \times N}, C \in \mathbb{R}^{M \times N}$

Recall

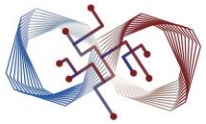
并行计算：拆分计算和数据

怎么拆?

拆到哪?



矩阵乘法 GENERAL MATRIX MUTIPLICATION

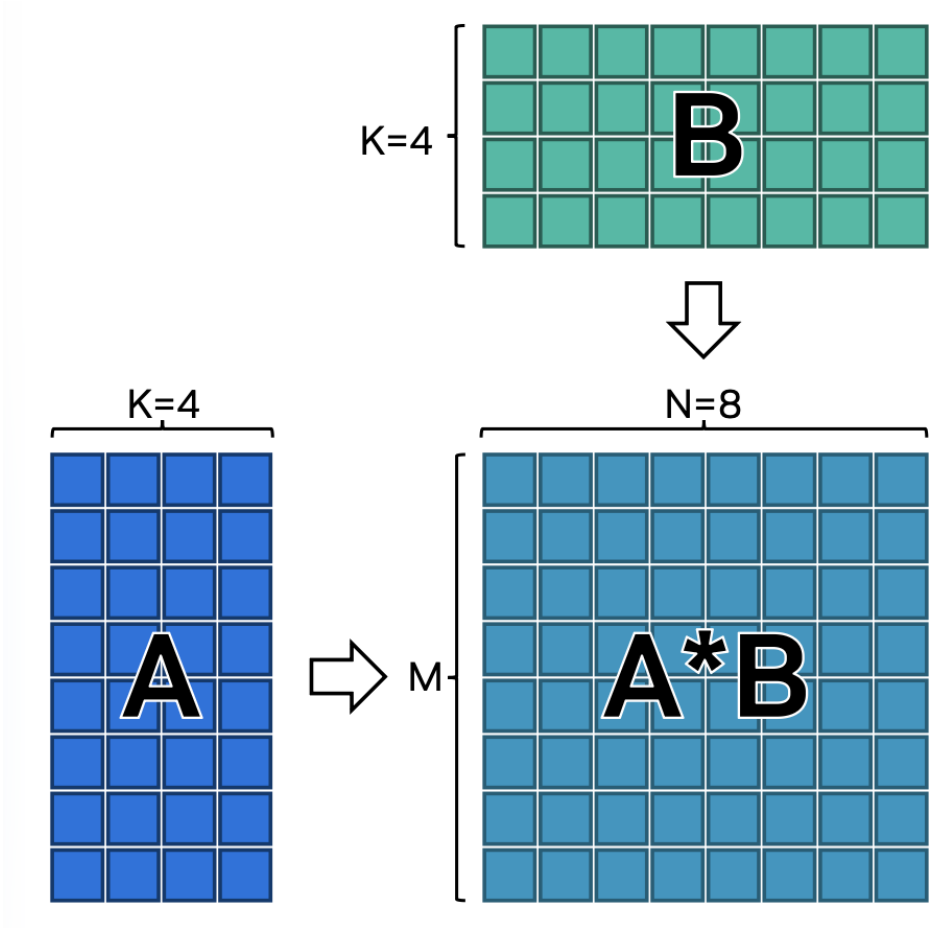


- 计算：$C = A \times B$

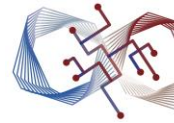
  $A \in \mathbb{R}^{M \times K}, B \in \mathbb{R}^{K \times N}, C \in \mathbb{R}^{M \times N}$

  目标：FLOP/s

  FMA



矩阵乘法 GENERAL MATRIX MUTIPLICATION



- 计算：$C = A \times B$

  $A \in \mathbb{R}^{M\times K}, B \in \mathbb{R}^{K\times N}, C \in \mathbb{R}^{M\times N}$

Recall

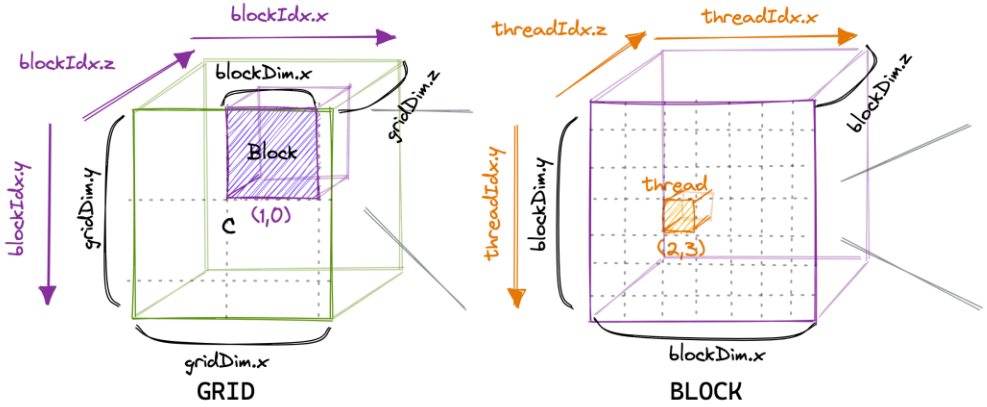
并行计算：拆分计算和数据

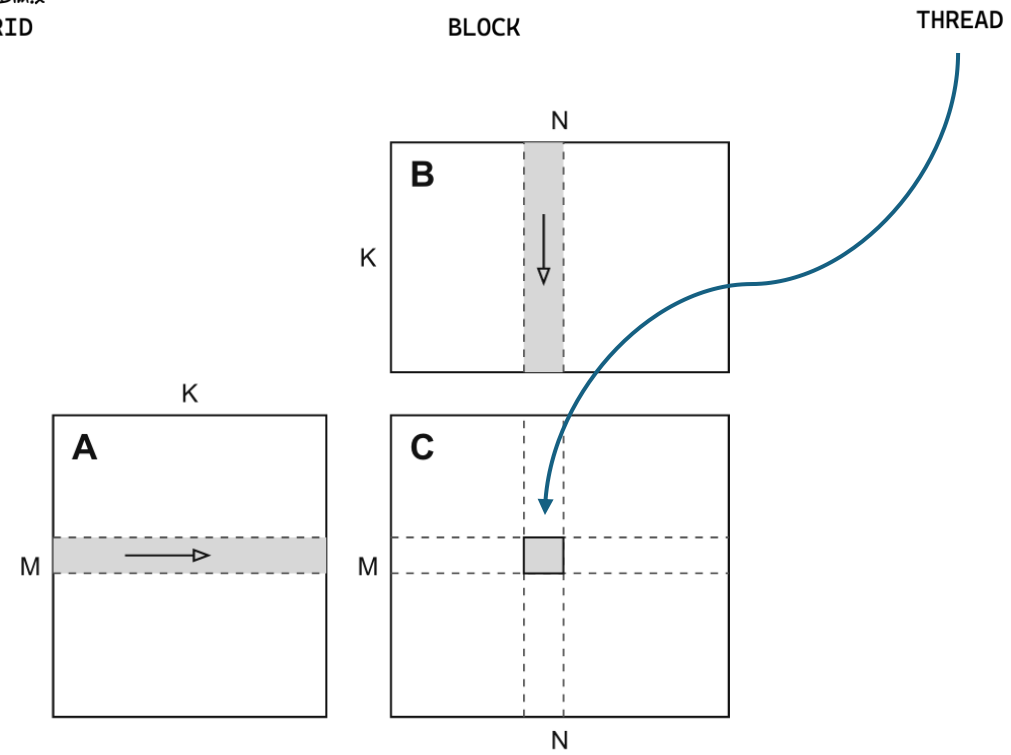
$C[i,j] = \sum_k A[i,k] * B[k,j]$

怎么拆？数据并行！

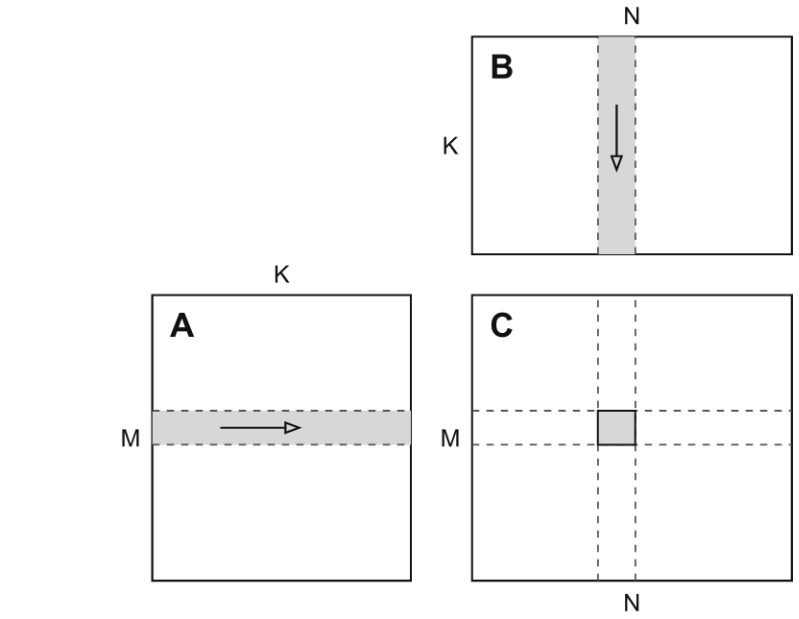
拆到哪？Thread！

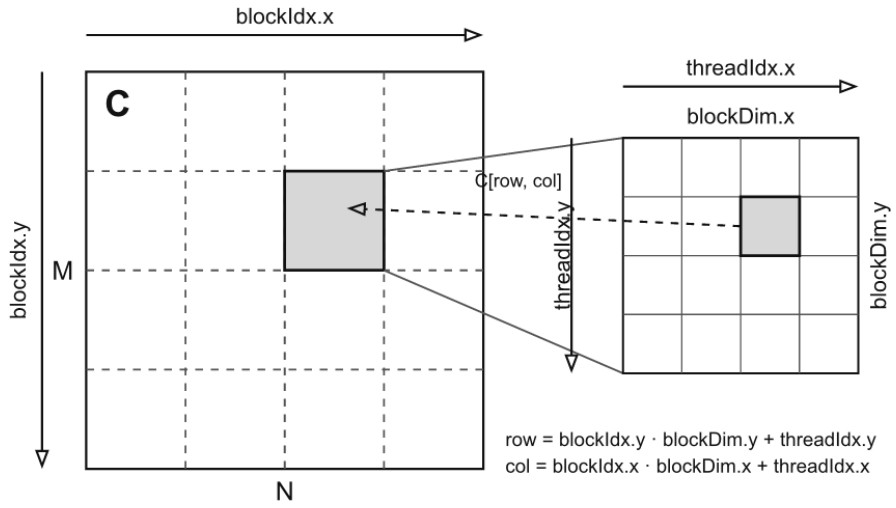
SIMT 以及 CUDA编程模型





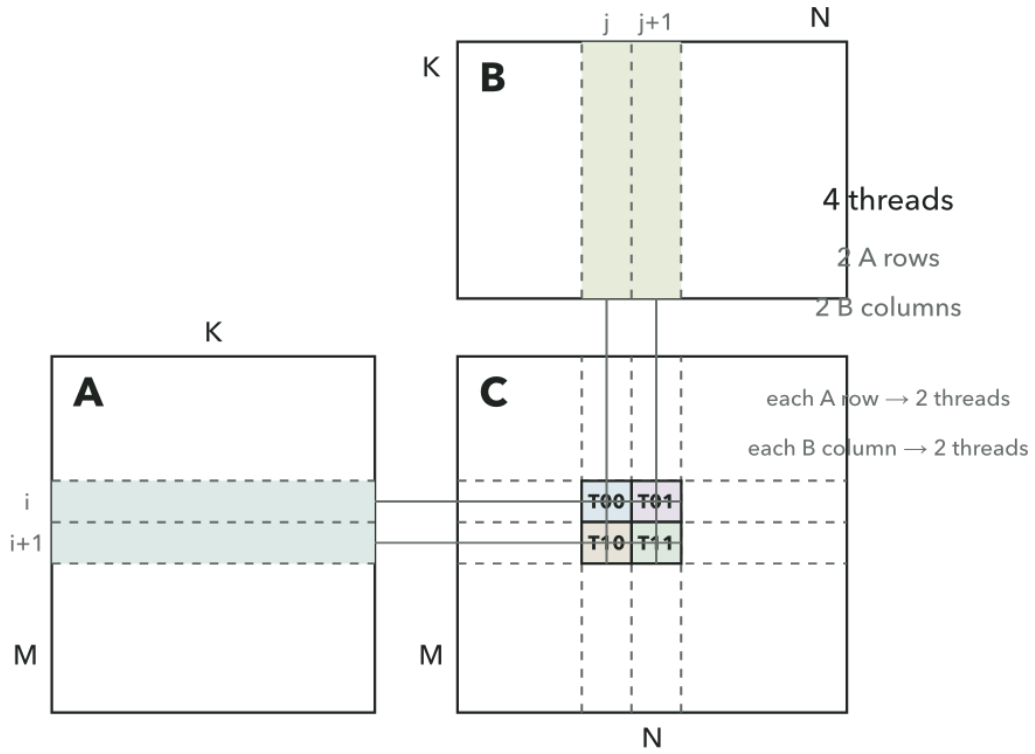
Naïve GEMM 工作拆分 – 每个thread一个元素






```cpp
__global__ void gemm_naive(
  const float* __restrict__ A,
  const float* __restrict__ B,
  float* __restrict__ 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) return;
  float acc = 0.0f;
  for (int k = 0; k < K; ++k) {
    acc += A[row * K + k] * B[k * N + col];
  }
  C[row * N + col] = acc;
}

dim3 block(16, 16);
dim3 grid(ceil_div(N, 16), ceil_div(M, 16));

gemm_naive<<<grid, block>>>(A, B, C, M, N, K);
```

Naïve GEMM 的问题 – 数据复用

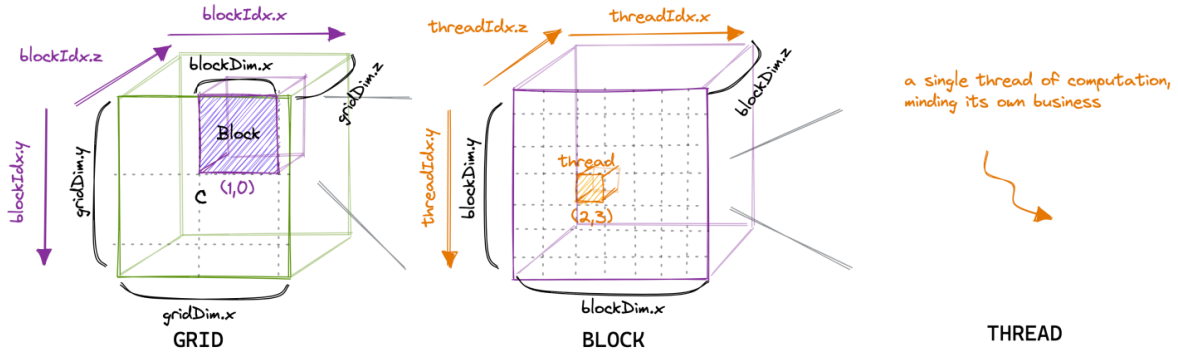


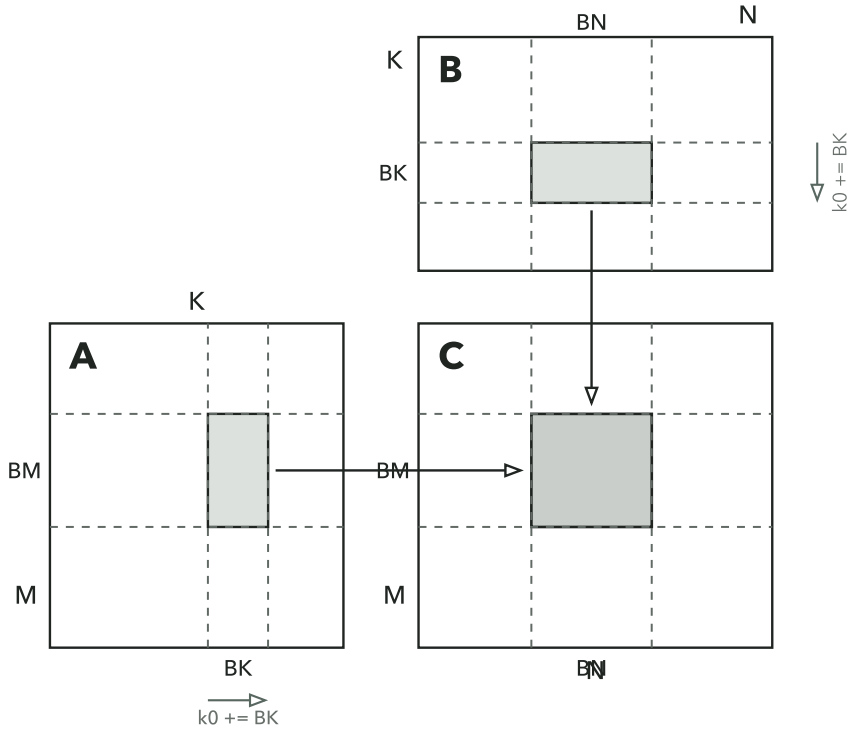
- 4 个相邻线程计算 4 个输出
- 却只依赖 2 行 A 和 2 列 B

- 每个线程访问了 $2K$ 个输入和1个输出
- 做了 K 次乘加（FMA）
- $AI = \frac{2K}{8K+4} \approx 0.25$ FLOPs/Bytes

- Recall Roofline：
  - 理想的大规模匀称 GEMM 是 compute bound 的
  - $AI = \frac{2MNK}{4(MN+MK+NK)} = \frac{\mathrm{n}}{6}, \mathrm{n} = \mathrm{M} = \mathrm{N} = \mathrm{K}$

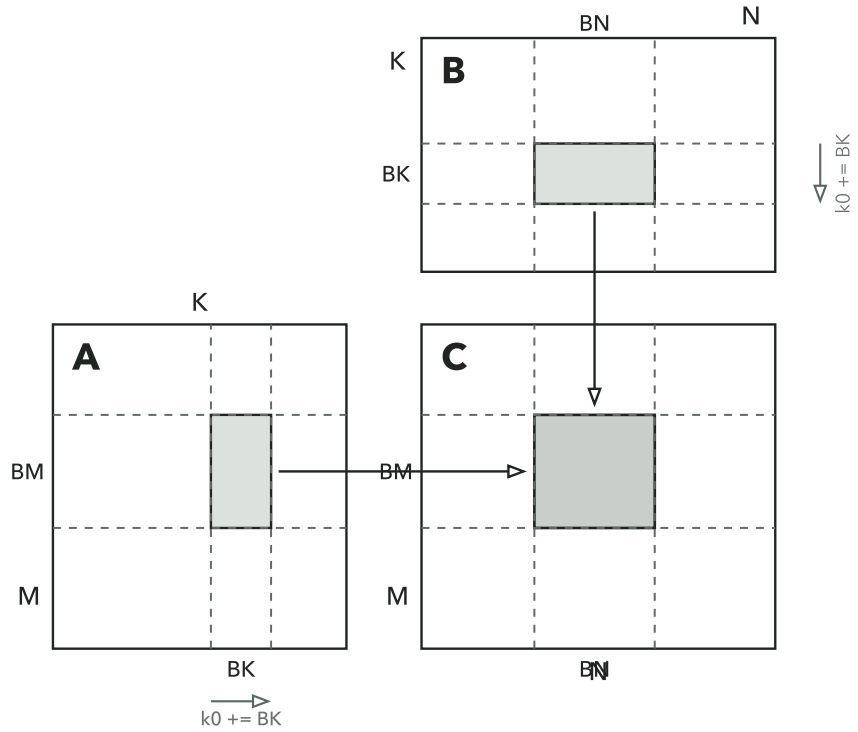
Shared Memory 用于数据复用





- Cache 可能捕获一部分复用，但代码没有显式组织和保证这种协作
- 利用2D CTA grid，让每个 CTA 负责一个$B_M \times B_N$的输出tile
  - 这次不同的是 threads 协作把需要的所有数据搬运到 shared memory

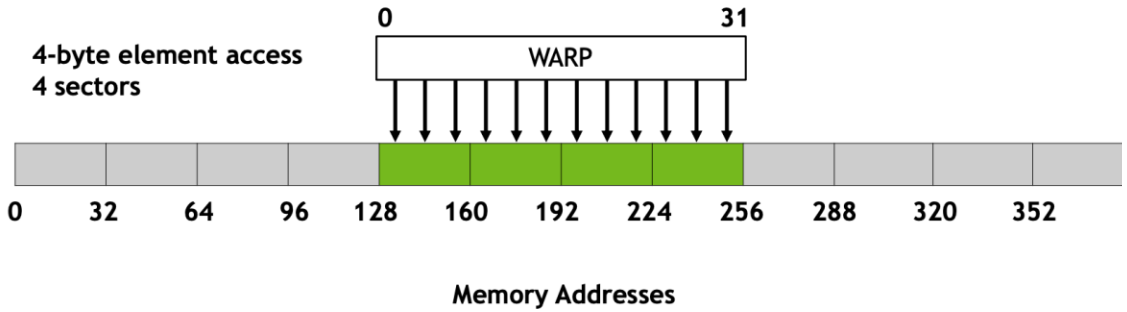
Shared Memory 用于数据复用



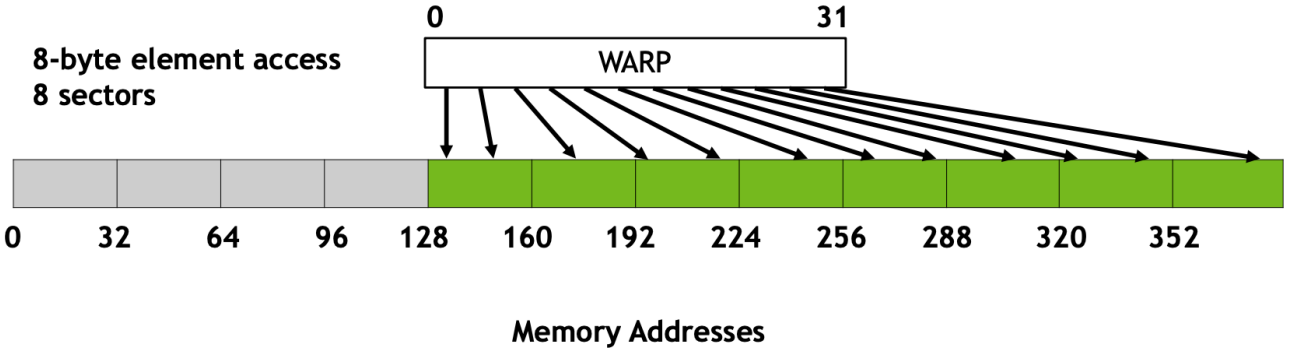
- $AI = \frac{2B_M B_N B_K}{4(B_M B_N + B_N B_K + B_M B_K)} = \frac{B_M B_N}{2(B_M + B_N)}$
- $= 8\ FLOPs/Byte$ (T=32)

```
constexpr int T = 32; // BM = BN = BK = T
__shared__ float As[T][T];
__shared__ float Bs[T][T];

int tx = threadIdx.x, ty = threadIdx.y;
int row = blockIdx.y * T + ty;
int col = blockIdx.x * T + tx;
float acc = 0.0f;

for (int k0 = 0; k0 < K; k0 += T) {
  As[ty][tx] = load_A(row, k0 + tx);
  Bs[ty][tx] = load_B(k0 + ty, col);

  __syncthreads(); // tile 写完后才能读
  for (int k = 0; k < T; ++k)
    acc = fmaf(As[ty][k], Bs[k][tx], acc);
  __syncthreads(); // 全部读完后才能覆盖
  }
}
```

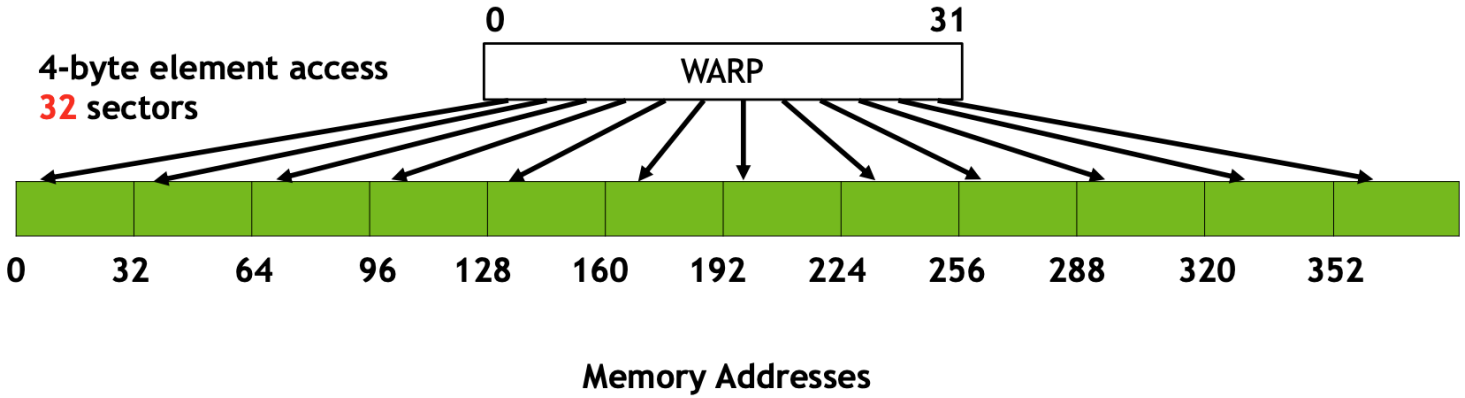
Global to Shared Coalescing



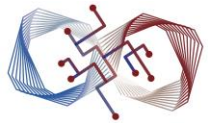
COALESCED!



COALESCED!



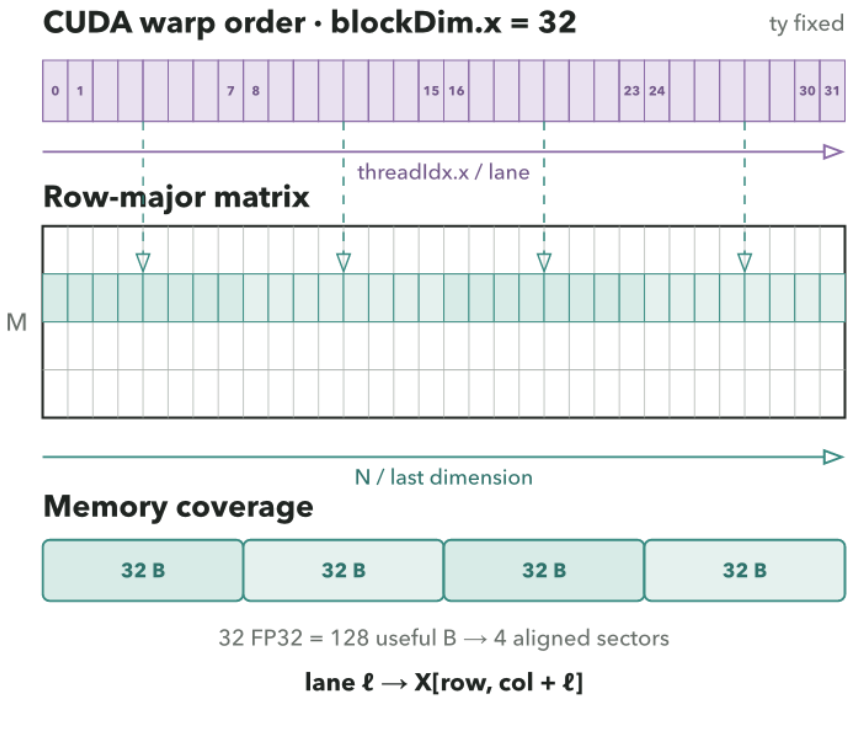
Global to Shared Coalescing – 对齐最连续的维度

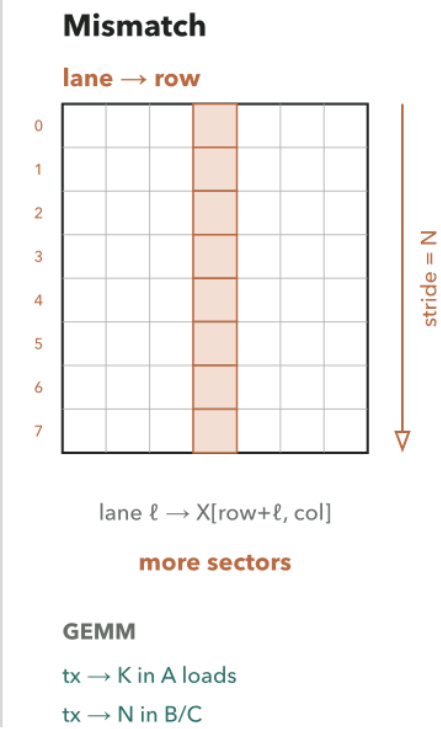


- 逻辑的高维 thread 和矩阵，对应的硬件执行和内存地址总是 1D 的

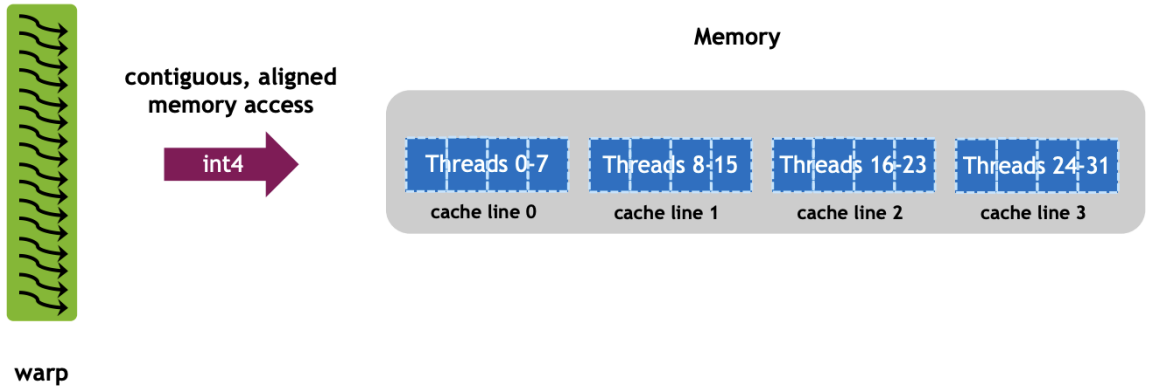
- 我们会有一些 **convention**
- C Style 数组 Row major，意味着
  - A[i][j] = A[i * K + j]
  - 观察连续变动的维度

- CUDA 的规定是
- linear_tid = tx + blockDim.x * ty;



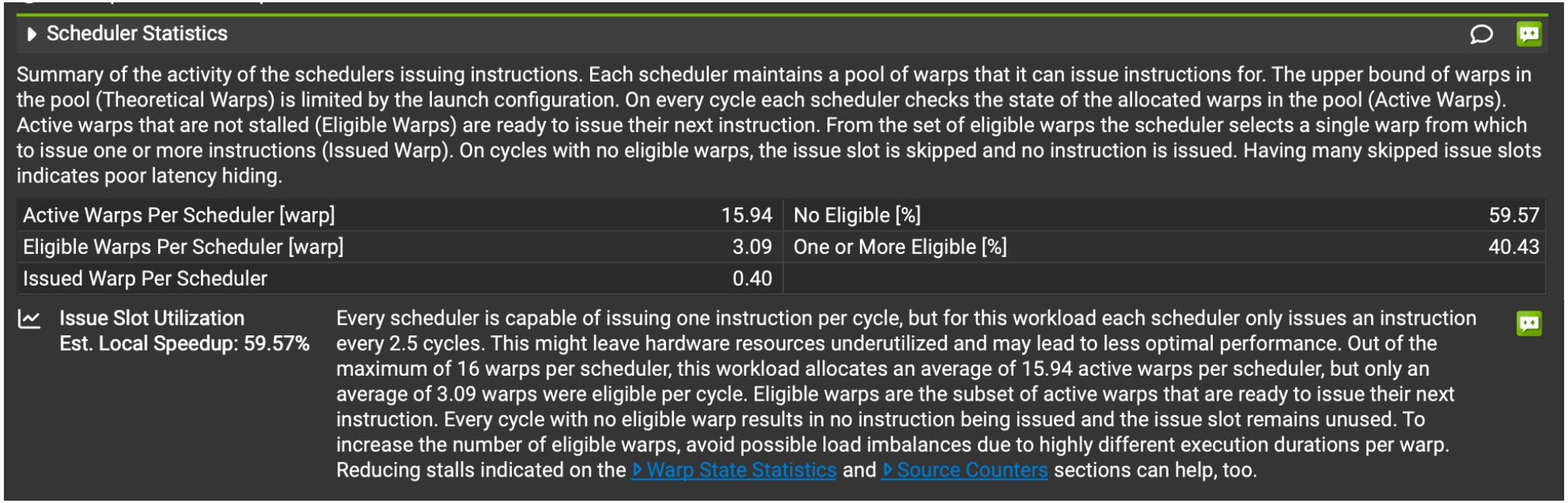


Global to Shared Vectorized




```
int row = tid / (BK / 4);
int col4 = (tid % (BK / 4)) * 4;

float4 value =
*reinterpret_cast<const float4*>(
&A[(block_m + row) * K + k0 + col4]
);

*reinterpret_cast<float4*>(
&As[row][col4]
) = value;
```

瓶颈：Shared Tiling != 持续发射



▸ Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

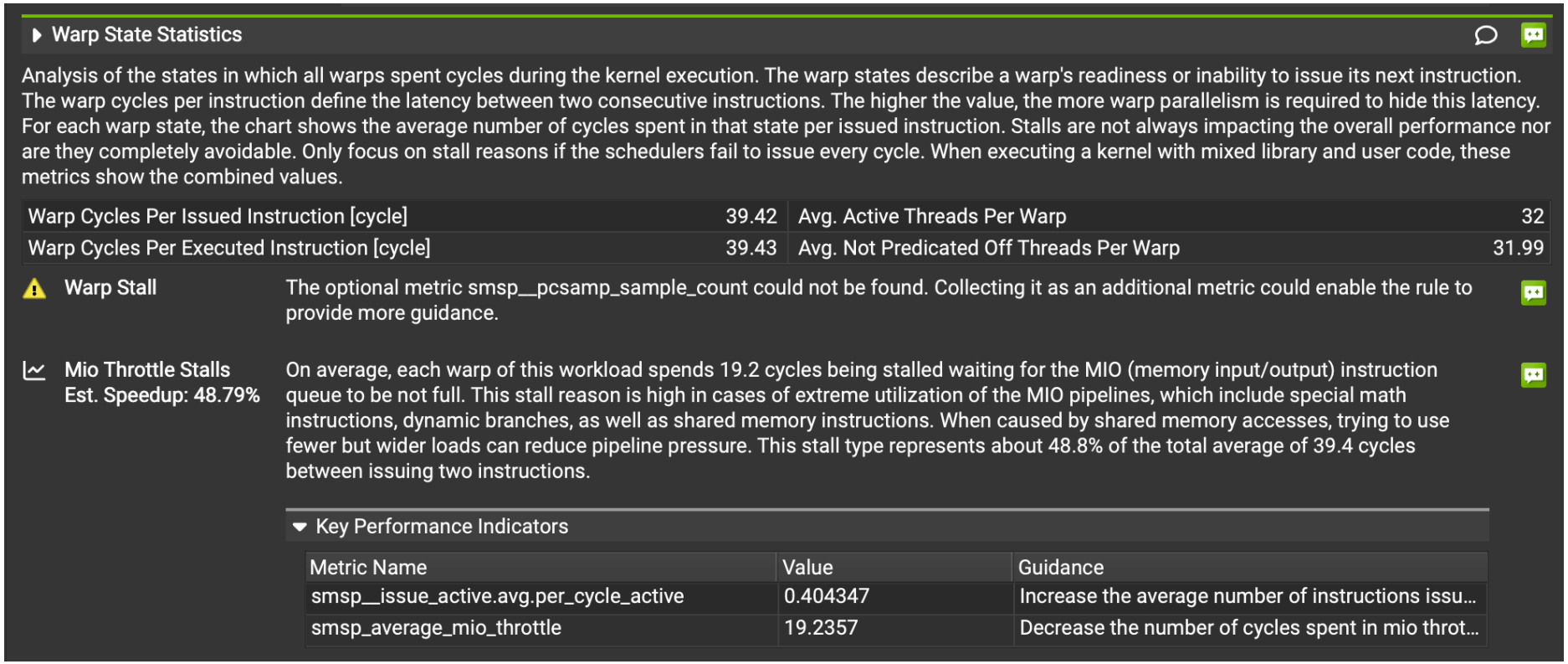
Active Warps Per Scheduler [warp]	15.94	No Eligible [%]	59.57
Eligible Warps Per Scheduler [warp]	3.09	One or More Eligible [%]	40.43
Issued Warp Per Scheduler	0.40		

Issue Slot Utilization
Est. Local Speedup: 59.57%

Every scheduler is capable of issuing one instruction per cycle, but for this workload each scheduler only issues an instruction every 2.5 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 16 warps per scheduler, this workload allocates an average of 15.94 active warps per scheduler, but only an average of 3.09 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, avoid possible load imbalances due to highly different execution durations per warp. Reducing stalls indicated on the ▸ Warp State Statistics and ▸ Source Counters sections can help, too.

- Scheduler 有足够多的 resident warps，但大多数周期没有 eligible warp

瓶颈：Shared Tiling != 持续发射



▸ Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	39.42	Avg. Active Threads Per Warp	32
Warp Cycles Per Executed Instruction [cycle]	39.43	Avg. Not Predicated Off Threads Per Warp	31.99

Warp Stall — The optional metric smsp__pcsamp_sample_count could not be found. Collecting it as an additional metric could enable the rule to provide more guidance.

Mio Throttle Stalls
Est. Speedup: 48.79%

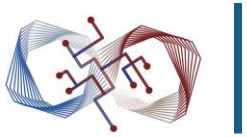
On average, each warp of this workload spends 19.2 cycles being stalled waiting for the MIO (memory input/output) instruction queue to be not full. This stall reason is high in cases of extreme utilization of the MIO pipelines, which include special math instructions, dynamic branches, as well as shared memory instructions. When caused by shared memory accesses, trying to use fewer but wider loads can reduce pipeline pressure. This stall type represents about 48.8% of the total average of 39.4 cycles between issuing two instructions.

▾ Key Performance Indicators

Metric Name	Value	Guidance
smsp__issue_active.avg.per_cycle_active	0.404347	Increase the average number of instructions issu...
smsp_average_mio_throttle	19.2357	Decrease the number of cycles spent in mio throt...

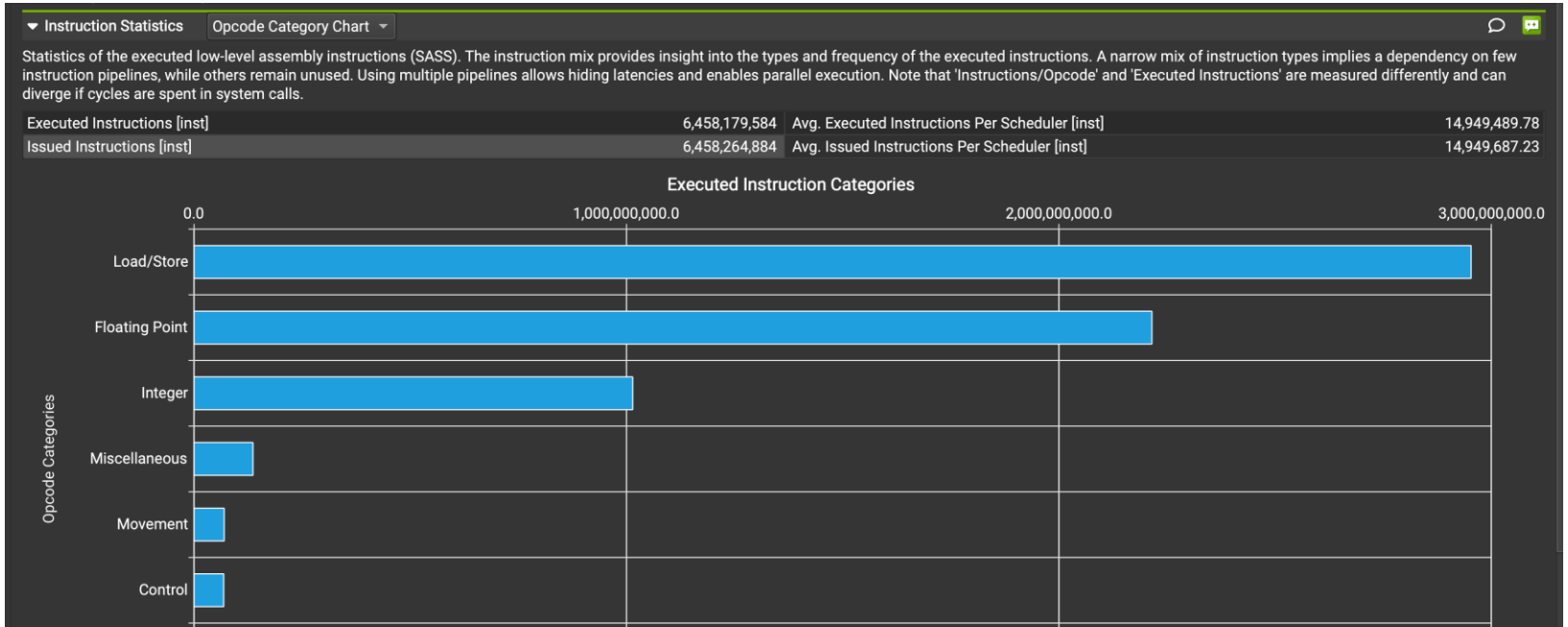
- Scheduler 有足够多的 resident warps，但大多数周期没有 eligible warp
- 主要原因是 MIO/shared-memory instruction pressure

瓶颈：Shared Tiling != 持续发射

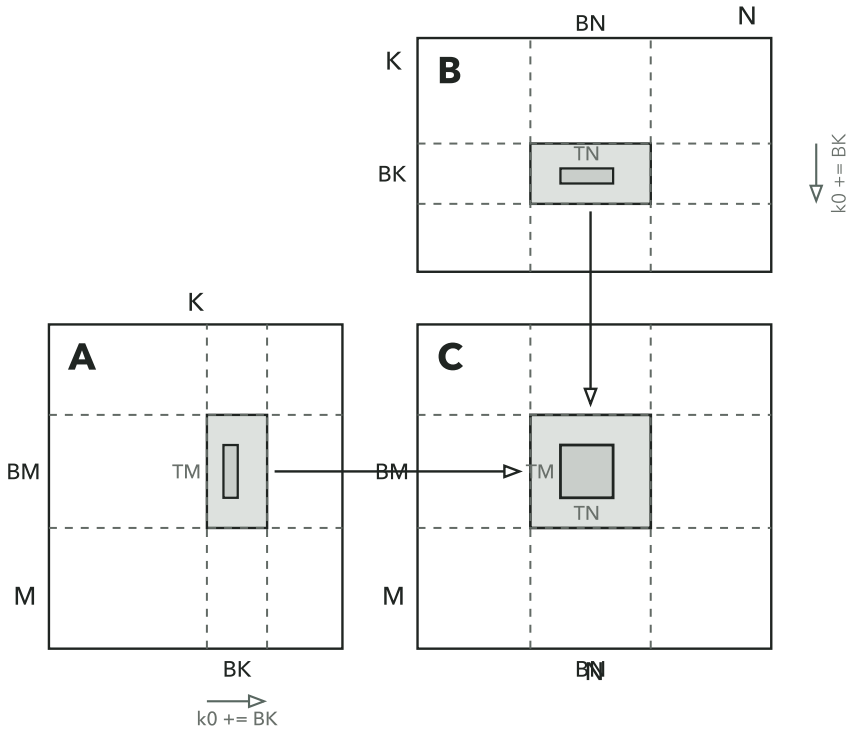


- Warp Scheduler
- ├─ FMA / math pipelines
- ├─ global/local-memory paths
- └─ MIO queue
- ├─ shared-memory instructions
- ├─ special math
- └─ dynamic branch-related operations

- Warp 想继续发 shared/MIO 指令，但相关 instruction queue 已经拥塞
- 需要 shared memory 更好的复用
- 绝大多数指令都是 load/store



Tile内的工作分配



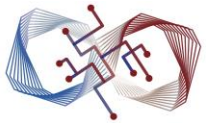
- 类似地，tile内每个 thread 处理更多输出元素可以减少shared-load instruction pressure
- 同时解放了threads!

```
constexpr int BM = 32, BN = 32;
constexpr int BK = 8;
constexpr int TM = 2, TN = 2;

// block = dim3(16, 16) → 256 threads
int tc = threadIdx.x; // group of TN columns
int tr = threadIdx.y; // group of TM rows

int row0 = blockIdx.y * BM + tr * TM;
int col0 = blockIdx.x * BN + tc * TN;

float acc[TM][TN] = {};
```

- CTA tile: 32 × 32
- Thread tile: 2 × 2
- Thread grid: 16 × 16 = 256 threads

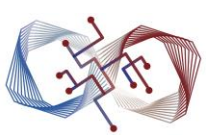
- per k:
- 2 A values + 2 B value → 4 FMAs

Register Tiling 解锁了更大的 CTA Tile



- $AI = \frac{2B_M B_N B_K}{4(B_M B_N + B_N B_K + B_M B_K)} = \frac{B_M B_N}{2(B_M + B_N)}$
- 提升 CTA tile 大小有利于 data reuse
- 一个 thread 负责更多的计算，

目前的局限以及展望



已达到：

- 80%+ cublas performance
- 87% fp32 pipe util

局限?

- 串行"流水"：load -> compute -> load -> … -> load -> …

  Load占据关键时间!

解法?

- 多缓冲：Stages 拉长了计算依赖距离
- 异步数据搬运
  - cp.async
  - Tensor Memory Acclerator 专用异步硬件