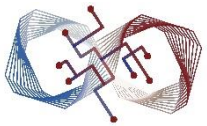
Weiming HPC Training Camp
×
LCPU AI Infra Seminars

北京大学未名超算队
北京大学学生 Linux 俱乐部

Session 03.
Tensor Core：从 mma.sync 到 tcgen05

孙远航
2026. 08. 02
Weiming Supercomputing Team
Linux Club of Peking University

Overview



核心问题：Tensor Core 是什么，以及如何为它准备数据

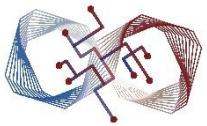
目录：

1. Tensor Core 是什么：计算模型与计算强度
2. sm80：fragment、ldmatrix、mma.sync
3. sm90：wgmma、异步与 proxy、descriptor、swizzle
4. sm100：TMEM、tcgen05、mbarrier、2-CTA
5. 低精度：FP8、block scaling、硬件 scaling

P1 · Tensor Core 是什么

P1 · Tensor Core 是什么

fp32 GEMM 的效率



naive fp32 GEMM 的计算强度只有 0.25 FLOP/byte，受带宽约束只能跑到 fp32 峰值的 2.6%

```
for (k = 0; k < K; ++k)
    acc += A[i][k] * B[k][j];
```

内层每一步：读 2 个 fp32 = **8 B**，做 1 次 FMA = **2 FLOP**

naive GEMM 的计算强度	A100 fp32 平衡点	A100 fp16 TC 平衡点
0.25	**9.75**	**156**
FLOP / byte	19.5 TFLOPS / 2.0 TB/s	312 TFLOPS / 2.0 TB/s

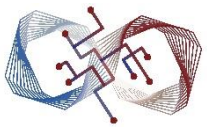
实际能跑多快

0.25 × 2.0 TB/s = **0.5 TFLOPS**， 只有 fp32 峰值的 **2.6%**

瓶颈在哪

瓶颈不是计算速度，是**数据供给**。每个元素只用一次就丢弃，没有任何复用

矩阵乘为什么适合硬件加速

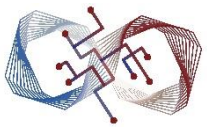


深度学习中最重要的计算形态

性质好：$O(n^3)$ 计算 vs $O(n^2)$ 数据 → 天然适合提高计算密度

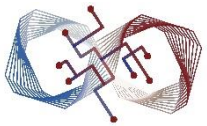
各种加速设备很早就在做矩阵乘加速，Tensor Core 是其中比较成功的一个

想一想：硬件为什么不直接收整个 tensor



为什么硬件不直接传入四个完整的 tensor，算完返回结果?

Tensor Core 的定义

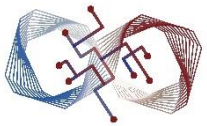


定义：一个用硬件算 `D = A×B + C` 的单元，A/B/C 形状固定，而且是一小块

若做成整个 tensor 进出，中间无法做任何处理，CUDA core 与访存都会闲置

把这个能力**拆解到已有的硬件单元上**是刻意的设计

mma 指令的字段 (1/2)



把限定符竖起来看

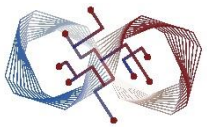
mma	Matrix Multiply-Accumulate
.sync	执行前隐含一次 warp 同步
.aligned	warp 内 32 线程必须同时执行，否则未定义行为
.m16n8k16	A 是 m×k = 16×16，B 是 n×k = 8×16
.row.col	A 行主序、B 列主序 —— 本质是让 K 维连续
.f32.f16.f16.f32	D / A / B / C：操作数 fp16，累加 fp32

mma = Matrix Multiply-Accumulate

aligned = warp 内 32 线程必须同时执行，否则未定义行为。这条指令由**全 warp 协作完成**

sync = 指令执行前隐含一次 warp sync

mma 指令的字段 (2/2)



把限定符竖起来看

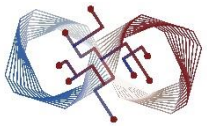
```
mma             Matrix Multiply-Accumulate
.sync           执行前隐含一次 warp 同步
.aligned        warp 内 32 线程必须同时执行，否则未定义行为
.m16n8k16       A 是 m×k = 16×16，B 是 n×k = 8×16
.row.col        A 行主序、B 列主序 —— 本质是让 K 维连续
.f32.f16.f16.f32   D / A / B / C：操作数 fp16，累加 fp32
```

`row.col`：A 行主序、B 列主序，目的是让 **K 维连续**

`m16n8k16`：A 是 m×k = 16×16，B 是 n×k = 8×16

`f32.f16.f16.f32`：D / A / B / C。操作数是 fp16，**累加精度是 fp32**

PTX 文档里的 mma 指令表

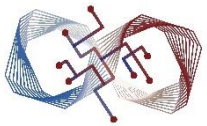


PTX ISA 文档的 Warp Level Matrix Instructions 一节

https://docs.nvidia.com/cuda/parallel-thread-execution/#warp-level-matrix-instructions-mma

没有 fp32 的 Tensor Core，最多只有 tf32，原因是节省电路

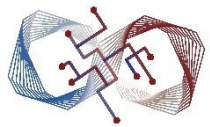
算一算：一条 mma 的计算强度



这一条 m16n8k16 指令的计算强度是多少?

(fp16 输入，fp32 累加)

计算强度：mma 与 ffma



mma.m16n8k16

fp16 输入，fp32 累加

MACs	16 × 8 × 16 = 2048
FLOPs	4096
read A	16 × 16 × 2 B = 512 B
read B	16 × 8 × 2 B = 256 B
write D	16 × 8 × 4 B = 512 B
total traffic	**1280 B**

3.2
FLOP / byte

ffma (CUDA core)

fp32，3 读 1 写

MACs	1
FLOPs	2
read a, b, c	3 × 4 B = 12 B
write d	1 × 4 B = 4 B
total traffic	**16 B**

0.125
FLOP / byte

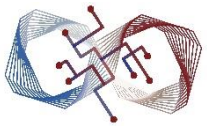
3.2 ÷ 0.125 = **26**

MAC 数 = 16×8×16 = 2048 → **4096 FLOP**

读 A 16×16×2 + B 16×8×2 = 768 B，写 D 16×8×4 = 512 B → 共 **1280 B**

4096 / 1280 = **3.2 FLOP/byte**

增大 MNK 的代价

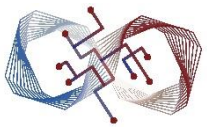


提高 MNK 能进一步提高计算强度

但**数据输入压力**随之增大，且 tile 需要存放空间

这是后续几代硬件改动的直接动因

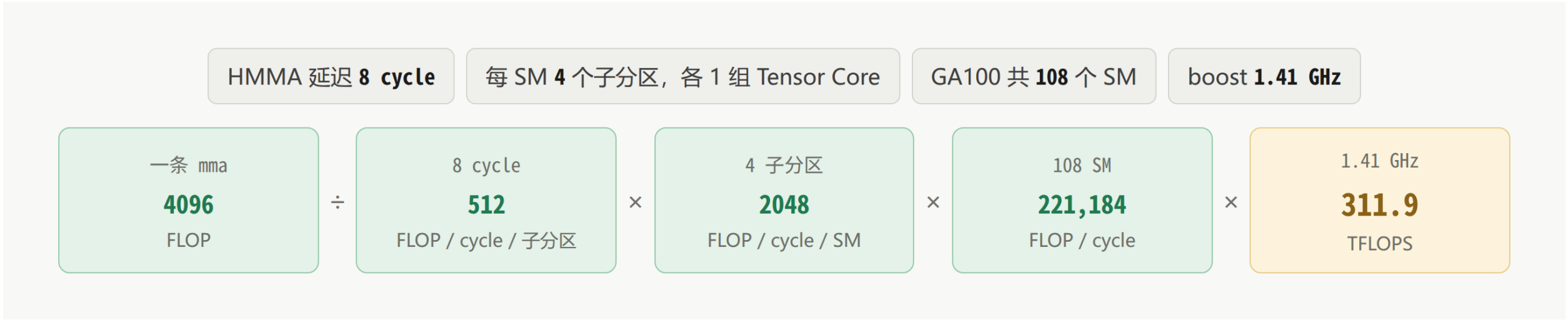
算一算：A100 的 FP16 峰值



A100 的 FP16 Tensor Core 峰值是多少 TFLOPS?

已知：HMMA 延迟 8 cycles；GA100 每 SM 4 个子分区、各含 1 组三代 TC；共 108 SM；boost 1.41 GHz

A100 FP16 峰值的推导



4096 FLOP / 8 cycle = 512 FLOP/cycle/子分区

× 4 子分区 = 2048 FLOP/cycle/SM

× 108 SM = 221,184 FLOP/cycle

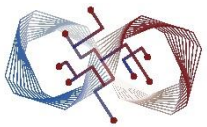
× 1.41 GHz = **311.9 TFLOPS**

机器平衡点与单条 mma 的差距

A100：312 TFLOPS / 2.0 TB/s ≈ **156 FLOP/byte**

单条 mma 的计算强度是 3.2，相差约 50 倍

三代 Tensor Core 的瓶颈与对策



计算单元：**D = A×B + C 的脉动阵列**，三代基本相同

sm80 Ampere · A100

FP16 dense **312** TFLOPS

问题

A / B / C / D **全在寄存器**，
每个线程自己算地址取数

方案

`fragment` `ldmatrix` `swizzle`

sm90 Hopper · H100

FP16 dense **990** TFLOPS

问题

寄存器带宽不够，
且 TC 计算时 CUDA core 空闲

方案

`descriptor` `异步 wgmma` `TMA` `mbarrier`

sm100 Blackwell · B200

FP16 dense **2250** TFLOPS

问题

累加器**占用大量寄存器**，
发射指令**不需要线程参与**

方案

`TMEM` `单线程发射` `2-CTA`

NVIDIA 每代算力至少翻倍，但数据搬运跟不上

H100 / B200 的**计算单元变化不大**，都是做 D = A×B + C 的脉动阵列

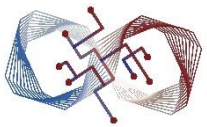
为了喂饱 tensor core，产生了一整串概念：

fragment → ldmatrix → swizzle → descriptor → TMA → tmem

P2 序 · Fragment

P2 · Fragment 与三代 Tensor Core 指令

mma 的每线程操作数

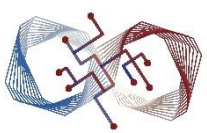


PTX 是以线程为单位执行的：这是每个线程眼里的一条 mma

```
.reg .f16 %Ra<4>, %Rb<2>;
.reg .f32 %Rc<4>, %Rd<4>;

mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32
  {%Rd0, %Rd1, %Rd2, %Rd3},      // D
  {%Ra0, %Ra1, %Ra2, %Ra3},      // A
  {%Rb0, %Rb1},                  // B
  {%Rc0, %Rc1, %Rc2, %Rc3};      // C
```

每个线程：4 个 D、4 个 A、2 个 B、4 个 C

mma 的每线程操作数

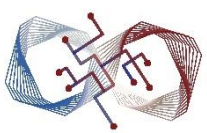


PTX 是以线程为单位执行的：这是每个线程眼里的一条 mma

```
.reg .f16 %Ra<4>, %Rb<2>;
.reg .f32 %Rc<4>, %Rd<4>;

mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32
  {%Rd0, %Rd1, %Rd2, %Rd3},      // D
  {%Ra0, %Ra1, %Ra2, %Ra3},      // A
  {%Rb0, %Rb1},                  // B
  {%Rc0, %Rc1, %Rc2, %Rc3};      // C
```

每个线程：4 个 D、4 个 A、2 个 B、4 个 C

mma 的每线程操作数

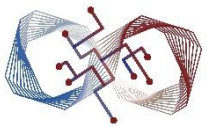


PTX 是以线程为单位执行的：这是每个线程眼里的一条 mma

```
.reg .f16 %Ra<4>, %Rb<2>;
.reg .f32 %Rc<4>, %Rd<4>;

mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32
  {%Rd0, %Rd1, %Rd2, %Rd3},      // D
  {%Ra0, %Ra1, %Ra2, %Ra3},      // A
  {%Rb0, %Rb1},                  // B
  {%Rc0, %Rc1, %Rc2, %Rc3};      // C
```

每个线程：4 个 D、4 个 A、2 个 B、4 个 C

fragment 的寄存器分配



元素总数 D/A/B/C = **128 / 256 / 128 / 128**，由一个 warp 的 32 个线程分摊

一个寄存器 32 bit，**fp16 两个打包在一个寄存器**，fp64 占两个

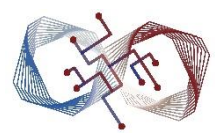
Tensor Core 编程的核心问题是**如何为它准备数据**

需要掌握两件事：看懂 fragment 图，用好脚手架

2.2 sm80：mma.sync

A / B / C / D 全部位于寄存器

A 矩阵的 fragment 布局



R\C	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	T0:{a0,a1}		T1:{a0,a1}		T2:{a0,a1}		T3:{a0,a1}		T0:{a4,a5}		T1:{a4,a5}		T2:{a4,a5}		T3:{a4,a5}	
1	T4:{a0,a1}		T5:{a0,a1}		T6:{a0,a1}		T7:{a0,a1}		T4:{a4,a5}		T5:{a4,a5}		T6:{a4,a5}		T7:{a4,a5}	
2																
..																
7	T28:{a0,a1}		T29:{a0,a1}		T30:{a0,a1}		T31:{a0,a1}		T28:{a4,a5}		T29:{a4,a5}		T30:{a4,a5}		T31:{a4,a5}	
8	T0:{a2,a3}		T1:{a2,a3}		T2:{a2,a3}		T3:{a2,a3}		T0:{a6,a7}		T1:{a6,a7}		T2:{a6,a7}		T3:{a6,a7}	
9	T4:{a2,a3}		T5:{a2,a3}		T6:{a2,a3}		T7:{a2,a3}		T4:{a6,a7}		T5:{a6,a7}		T6:{a6,a7}		T7:{a6,a7}	
10																
..																
15	T28:{a2,a3}		T29:{a2,a3}		T30:{a2,a3}		T31:{a2,a3}		T28:{a6,a7}		T29:{a6,a7}		T30:{a6,a7}		T31:{a6,a7}	

%laneid:{fragments}

laneid = 线程在 warp 内的编号

行是 row、列是 col，格子里是该位置的元素属于哪个线程、存在哪个寄存器

fragment 图的读法

分成四个象限，每个标签横跨两列

`T0:{a0,a1}` 占列 0–1 → T0 的 a0 在 (0,0)、a1 在 (0,1)。每个线程 4 个寄存器的分工：

row (M) \ col (K) →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3
1	4	4	5	5	6	6	7	7	4	4	5	5	6	6	7	7
2	8	8	9	9	10	10	11	11	8	8	9	9	10	10	11	11
3	12	12	13	13	14	14	15	15	12	12	13	13	14	14	15	15
4	16	16	17	17	18	18	19	19	16	16	17	17	18	18	19	19
5	20	20	21	21	22	22	23	23	20	20	21	21	22	22	23	23
6	24	24	25	25	26	26	27	27	24	24	25	25	26	26	27	27
7	28	28	29	29	30	30	31	31	28	28	29	29	30	30	31	31
8	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3
9	4	4	5	5	6	6	7	7	4	4	5	5	6	6	7	7
10	8	8	9	9	10	10	11	11	8	8	9	9	10	10	11	11
11	12	12	13	13	14	14	15	15	12	12	13	13	14	14	15	15
12	16	16	17	17	18	18	19	19	16	16	17	17	18	18	19	19
13	20	20	21	21	22	22	23	23	20	20	21	21	22	22	23	23
14	24	24	25	25	26	26	27	27	24	24	25	25	26	26	27	27
15	28	28	29	29	30	30	31	31	28	28	29	29	30	30	31	31

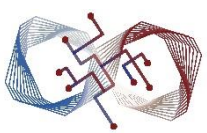
格子里是持有该元素的 lane 编号

a[0] = {a0,a1}　a[1] = {a2,a3}　a[2] = {a4,a5}　a[3] = {a6,a7}

lane = gid × 4 + tig　·　gid = lane >> 2　·　tig = lane & 3

row = gid + 8 × (r & 1)　·　col = 2 × tig + 8 × (r >> 1)

A 的 row / col 公式

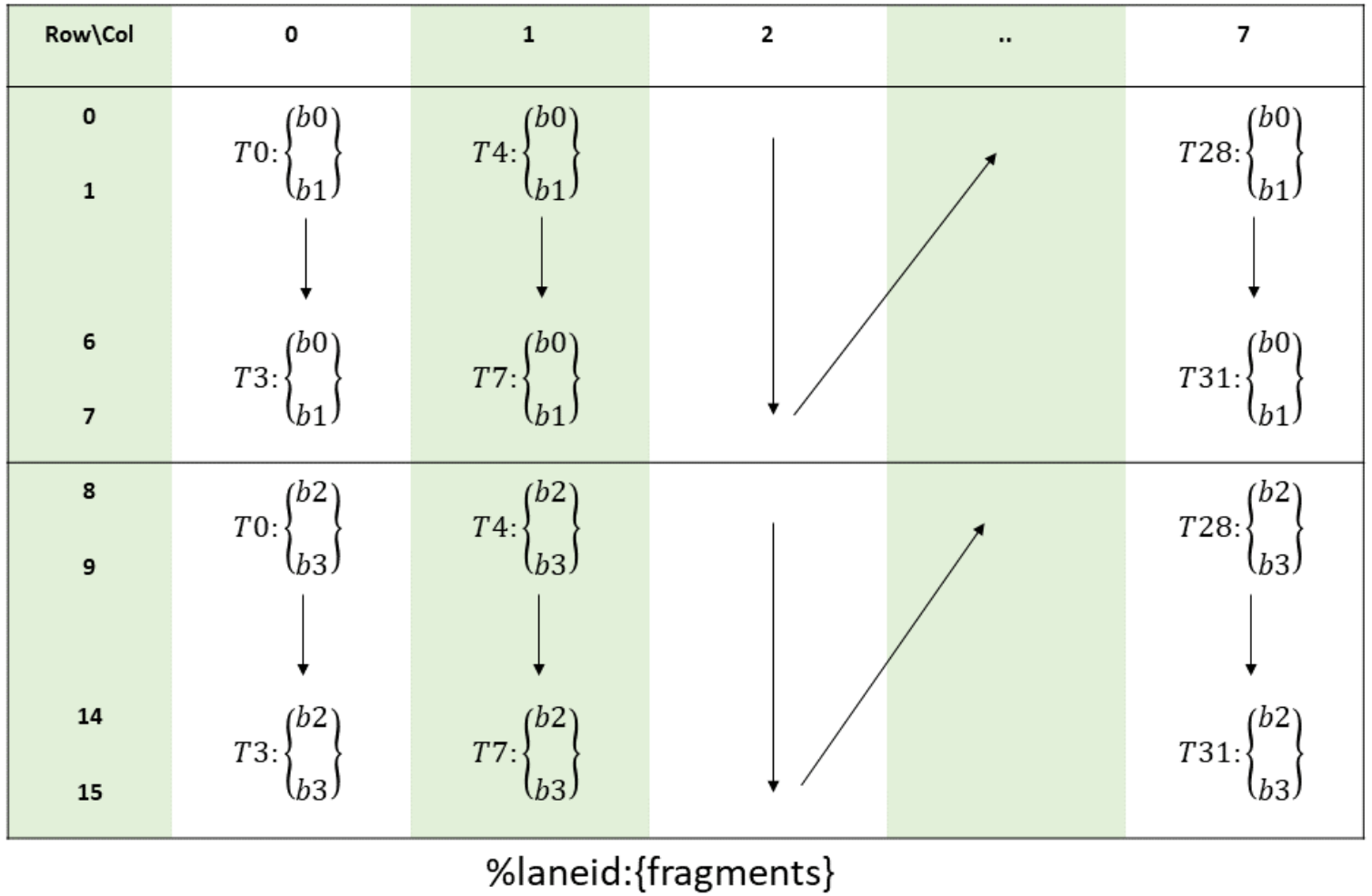


从 fragment 图读出来的两个式子

```
gid = lane >> 2          tig = lane & 3

row = gid   + 8 * (r & 1)        // r&1 : 上 / 下半区
col = 2*tig + 8 * (r >> 1)       // r>>1: 左 / 右半区
```

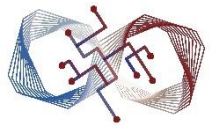
想一想：B 矩阵的 fragment 布局

根据 B 的这张图，推测一下 B 是如何划分的?



%laneid:{fragments}

B 的 row / col 公式



`n = lane >> 2`（gid，0..7） `k = 2*(lane&3) + 8*r`（r 选 K 的上/下半区）

B 只有 2 个寄存器，因为 B 是 8×16 = 128 个元素

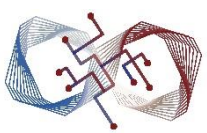
k (K) \ n (N) →	0	1	2	3	4	5	6	7
0	0	4	8	12	16	20	24	28
1	0	4	8	12	16	20	24	28
2	1	5	9	13	17	21	25	29
3	1	5	9	13	17	21	25	29
4	2	6	10	14	18	22	26	30
5	2	6	10	14	18	22	26	30
6	3	7	11	15	19	23	27	31
7	3	7	11	15	19	23	27	31
8	0	4	8	12	16	20	24	28
9	0	4	8	12	16	20	24	28
10	1	5	9	13	17	21	25	29
11	1	5	9	13	17	21	25	29
12	2	6	10	14	18	22	26	30
13	2	6	10	14	18	22	26	30
14	3	7	11	15	19	23	27	31
15	3	7	11	15	19	23	27	31

n（N）→
格子里是持有该元素的 lane 编号

- **b[0] = {b0, b1}**
- **b[1] = {b2, b3}**

`n = gid = lane >> 2`
`k = 2 × tig + 8 × r`
B 是 8 × 16 = 128 个元素，32 lane 各 4 个，所以只要 2 个寄存器

手搓 ① 数据摆放约定

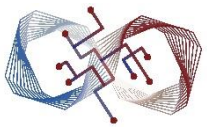


数据摆放约定：两个都让 K 维连续

```
__shared__ __align__(16) half sA[16 * 16];  // A: M×K = 16×16, row-major -> sA[m*16 + k]
__shared__ __align__(16) half sB[ 8 * 16];  // B: K×N = 16× 8, col-major -> sB[n*16 + k]
```

两个都让 **K 维连续**

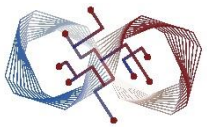
手搓 ② load_A_manual



手搓 A 的 fragment

```
__device__ void load_A_manual(const half* sA, uint32_t a[4]) {
    const int lane = threadIdx.x & 31;
    const int gid  = lane >> 2;        // 0..7
    const int tig  = lane & 3;         // 0..3

    #pragma unroll
    for (int r = 0; r < 4; ++r) {
        int row = gid   + 8 * (r & 1);      // 上 / 下半区
        int col = 2*tig + 8 * (r >> 1);     // 左 / 右半区
        a[r] = *reinterpret_cast<const uint32_t*>(&sA[row * 16 + col]);
    }
}
```

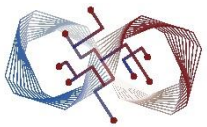
手搓 ② load_A_manual



手搓 A 的 fragment

```
__device__ void load_A_manual(const half* sA, uint32_t a[4]) {
    const int lane = threadIdx.x & 31;
    const int gid  = lane >> 2;         // 0..7
    const int tig  = lane & 3;          // 0..3

    #pragma unroll
    for (int r = 0; r < 4; ++r) {
        int row = gid   + 8 * (r & 1);      // 上 / 下半区
        int col = 2*tig + 8 * (r >> 1);     // 左 / 右半区
        a[r] = *reinterpret_cast<const uint32_t*>(&sA[row * 16 + col]);
    }
}
```

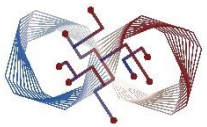
手搓 ② load_A_manual



手搓 A 的 fragment

```
__device__ void load_A_manual(const half* sA, uint32_t a[4]) {
    const int lane = threadIdx.x & 31;
    const int gid  = lane >> 2;        // 0..7
    const int tig  = lane & 3;         // 0..3

    #pragma unroll
    for (int r = 0; r < 4; ++r) {
        int row = gid   + 8 * (r & 1);      // 上 / 下半区
        int col = 2*tig + 8 * (r >> 1);     // 左 / 右半区
        a[r] = *reinterpret_cast<const uint32_t*>(&sA[row * 16 + col]);
    }
}
```

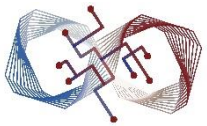
手搓 ③ load_B_manual



B 的 fragment：只要 2 个寄存器

```
__device__ void load_B_manual(const half* sB, uint32_t b[2]) {
    const int lane = threadIdx.x & 31;
    const int gid  = lane >> 2;        // 0..7 -> n
    const int tig  = lane & 3;         // 0..3

    #pragma unroll
    for (int r = 0; r < 2; ++r) {
        int n = gid;                   // N 方向
        int k = 2 * tig + 8 * r;       // r: K 的上 / 下半区
        b[r] = *reinterpret_cast<const uint32_t*>(&sB[n * 16 + k]);
    }
}
```

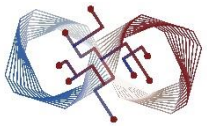
手搓 ④ 发射 mma



发射 mma，再把 D 写回

```
float c[4] = {0.f, 0.f, 0.f, 0.f};
float d[4];

asm volatile(
    "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32 "
    "{%0,  %1,  %2,  %3},  "     // D
    "{%4,  %5,  %6,  %7},  "     // A
    "{%8,  %9},            "     // B
    "{%10, %11, %12, %13};"      // C
    : "=f"(d[0]), "=f"(d[1]), "=f"(d[2]), "=f"(d[3])
    : "r"(a[0]), "r"(a[1]), "r"(a[2]), "r"(a[3]),
      "r"(b[0]), "r"(b[1]),
      "f"(c[0]), "f"(c[1]), "f"(c[2]), "f"(c[3])
);

#pragma unroll
for (int i = 0; i < 4; ++i) {
    int row = gid + 8 * (i >= 2);       // 注意：和 A 的索引不一样
    int col = 2 * tig + (i & 1);
    gD[row * 8 + col] = d[i];           // D: 16×8 row-major
}
```

手搓 ⑤ 写回 D

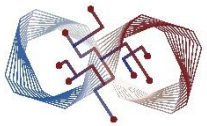


发射 mma，再把 D 写回

```
float c[4] = {0.f, 0.f, 0.f, 0.f};
float d[4];

asm volatile(
    "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32 "
    "{%0,  %1,  %2,  %3},  "     // D
    "{%4,  %5,  %6,  %7},  "     // A
    "{%8,  %9},            "     // B
    "{%10, %11, %12, %13};"      // C
    : "=f"(d[0]), "=f"(d[1]), "=f"(d[2]), "=f"(d[3])
    : "r"(a[0]), "r"(a[1]), "r"(a[2]), "r"(a[3]),
      "r"(b[0]), "r"(b[1]),
      "f"(c[0]), "f"(c[1]), "f"(c[2]), "f"(c[3])
);

#pragma unroll
for (int i = 0; i < 4; ++i) {
    int row = gid + 8 * (i >= 2);       // 注意：和 A 的索引不一样
    int col = 2 * tig + (i & 1);
    gD[row * 8 + col] = d[i];           // D: 16×8 row-major
}
```

D 的索引与 A **不同**：`row = gid + 8*(i>=2), col = 2*tig + (i&1)`

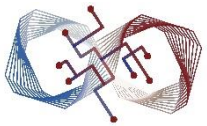
ldmatrix 的动机



手工计算每个线程的地址繁琐且容易出错

`ldmatrix` 把这一步交给硬件完成

ldmatrix 的调用方式

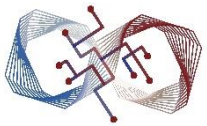


ldmatrix：每个 lane 提供一个 8×8 小矩阵某一行的首地址

```
__device__ __forceinline__ uint32_t smem_u32(const void* p) {
    return static_cast<uint32_t>(__cvta_generic_to_shared(p));
}

__device__ void load_A_ldmatrix(const half* sA, uint32_t a[4]) {
    const int lane = threadIdx.x & 31;
    //  lane  0-7  -> 左上象限 (行 0-7,  列 0)
    //  lane  8-15 -> 左下象限 (行 8-15, 列 0)
    //  lane 16-23 -> 右上象限 (行 0-7,  列 8)
    //  lane 24-31 -> 右下象限 (行 8-15, 列 8)
    int row = lane & 15;
    int col = (lane >> 4) * 8;

    uint32_t addr = smem_u32(&sA[row * 16 + col]);
    asm volatile(
        "ldmatrix.sync.aligned.m8n8.x4.shared.b16 {%0, %1, %2, %3}, [%4];"
        : "=r"(a[0]), "=r"(a[1]), "=r"(a[2]), "=r"(a[3])
        : "r"(addr)
    );
}
```

lane 0-7 → 左上象限，8-15 → 左下，16-23 → 右上，24-31 → 右下

ldmatrix 的调用方式

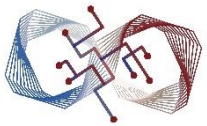


ldmatrix：每个 lane 提供一个 8×8 小矩阵某一行的首地址

```cpp
__device__ __forceinline__ uint32_t smem_u32(const void* p) {
    return static_cast<uint32_t>(__cvta_generic_to_shared(p));
}

__device__ void load_A_ldmatrix(const half* sA, uint32_t a[4]) {
    const int lane = threadIdx.x & 31;
    //  lane  0-7  -> 左上象限 (行 0-7,  列 0)
    //  lane  8-15 -> 左下象限 (行 8-15, 列 0)
    //  lane 16-23 -> 右上象限 (行 0-7,  列 8)
    //  lane 24-31 -> 右下象限 (行 8-15, 列 8)
    int row = lane & 15;
    int col = (lane >> 4) * 8;

    uint32_t addr = smem_u32(&sA[row * 16 + col]);
    asm volatile(
        "ldmatrix.sync.aligned.m8n8.x4.shared.b16 {%0, %1, %2, %3}, [%4];"
        : "=r"(a[0]), "=r"(a[1]), "=r"(a[2]), "=r"(a[3])
        : "r"(addr)
    );
}
```

lane 0-7 → 左上象限，8-15 → 左下，16-23 → 右上，24-31 → 右下

ldmatrix 的调用方式

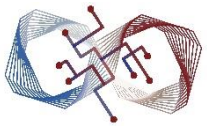


ldmatrix：每个 lane 提供一个 8×8 小矩阵某一行的首地址

```
__device__ __forceinline__ uint32_t smem_u32(const void* p) {
    return static_cast<uint32_t>(__cvta_generic_to_shared(p));
}

__device__ void load_A_ldmatrix(const half* sA, uint32_t a[4]) {
    const int lane = threadIdx.x & 31;
    //  lane  0-7  -> 左上象限 (行 0-7,  列 0)
    //  lane  8-15 -> 左下象限 (行 8-15, 列 0)
    //  lane 16-23 -> 右上象限 (行 0-7,  列 8)
    //  lane 24-31 -> 右下象限 (行 8-15, 列 8)
    int row = lane & 15;
    int col = (lane >> 4) * 8;

    uint32_t addr = smem_u32(&sA[row * 16 + col]);
    asm volatile(
        "ldmatrix.sync.aligned.m8n8.x4.shared.b16 {%0, %1, %2, %3}, [%4];"
        : "=r"(a[0]), "=r"(a[1]), "=r"(a[2]), "=r"(a[3])
        : "r"(addr)
    );
}
```

lane 0-7 → 左上象限，8-15 → 左下，16-23 → 右上，24-31 → 右下

ldmatrix 的三个注意点



01

地址语义是反直觉的

每个 lane 给的是**一个 8×8 小矩阵某一行的首地址**，不是这个 lane 自己期望拿到的元素地址

```
lane 0–7 → 左上象限
lane 8–15 → 左下象限
lane 16–23 → 右上象限
lane 24–31 → 右下象限
```

02

填写顺序决定拿到的顺序

四个小矩阵按什么次序填进去，就按什么次序回到 {%0,%1,%2,%3}。填对了就直接是正确的 fragment，不用再挪

```
ldmatrix.sync.aligned
  .m8n8.x4.shared.b16
  {a0,a1,a2,a3}, [addr];
```

03

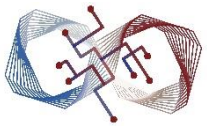
不会帮你消除 bank conflict

它只负责把数据摆成 fragment 的形状。地址落在哪些 bank 上，仍然由你的 smem 布局决定

```
行跨度 128 B 的整数倍时
→ 每行都从同一个 bank 起
→ 要靠 swizzle 打散
```

1. **地址语义反直觉**：给的是小矩阵首元素的地址，不是该线程期望拿到的元素地址
2. **填写小矩阵的顺序决定拿到元素的顺序**，按正确顺序填即可得到正确的 fragment
3. **不消除 bank conflict**

想一想：哪些 lane 会落在同一组 bank



行跨度不同时，32 个 lane 的地址会落在几组 bank 上?

行跨度对 ldmatrix 的影响

ldmatrix.x4：32 lane × 16 B = **512 B** · shared memory **128 B / cycle** · 理论下限 **4 cycle**

row stride = 32 B

16×16 fp16 tile

bank step = 32 B / 4 = **8 banks**

lane 0–15 (col 0) → group 0, 8, 16, 24

lane 16–31 (col 8) → group 4, 12, 20, 28

bank 占用

0-3	4-7	8-11	12-15	16-19	20-23	24-27	28-31

8 组 bank，各 4 个请求 → **4 cycle**

等于下限，无损失

row stride = 128 B

64×64 fp16 分块

bank step = 128 B / 4 = 32 ≡ **0**

lane 0–15 (col 0) → group 0 **全部相同**

lane 16–31 (col 8) → group 4 **全部相同**

bank 占用

0-3	4-7	8-11	12-15	16-19	20-23	24-27	28-31

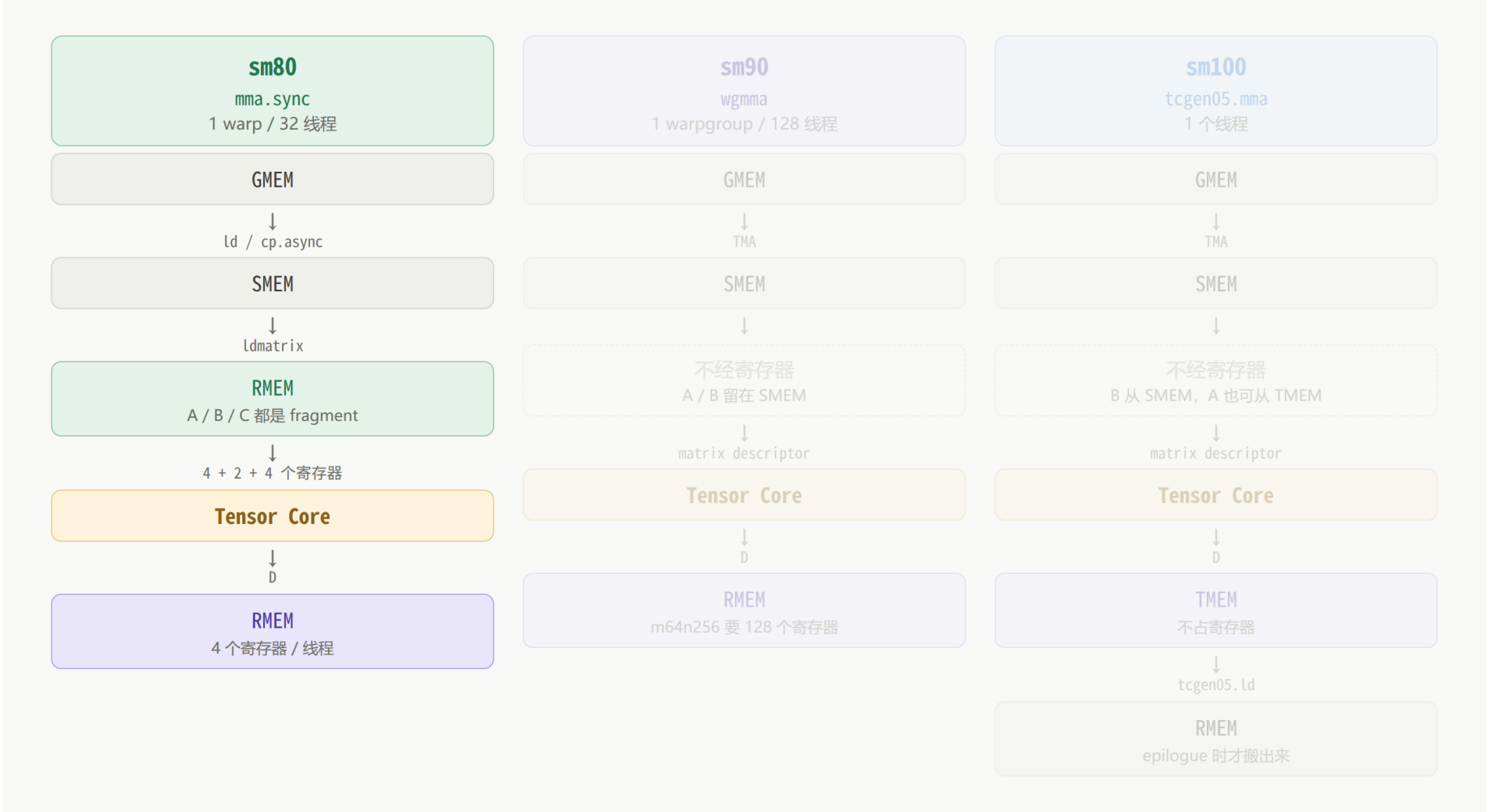
2 组 bank，各 16 个请求 → **16 cycle**

4 倍损失，24 个 bank 闲置

一次 `ldmatrix.x4` 取 32 lane × 16 B = 512 B，smem 每 cycle 吐 128 B → **理论下限 4 cycle**

`row stride = 32 B`（16×16 tile）：落在 8 组 bank，各 4 个请求，4 cycle，**等于下限，无损失**

sm80 小结



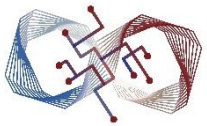
发起者：warp（32 线程各发一条） A / B / C / D **全在寄存器**

同步：隐式 warp sync 脚手架：fragment 图 + ldmatrix + swizzle

2.3 sm90：wgmma

操作数移入 shared memory，指令改为异步

mma 的寄存器带宽开销



	mma.m16n8k16 每条指令	摊到 8 cycle
读	A **4** + B **2** + C **4** = 10 reg × 32 lane × 4 B = **1280 B**	**160** B / cycle
写	D **4** reg × 32 lane × 4 B = **512 B**	**64** B / cycle

Hopper 想让 Tensor Core 性能翻倍

寄存器读写带宽也要跟着翻倍：**320 / 128 B per cycle**

但寄存器是最贵的资源

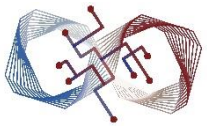
代际之间做不到翻倍。出路只有两条：**提高每次读的复用率**，或者**让操作数不走寄存器**

摊到 8 cycle：读 **160 B/cycle**，写 **64 B/cycle**

Tensor Core 性能翻倍要求寄存器读写带宽同样翻倍

寄存器是最昂贵的资源，代际之间无法翻倍

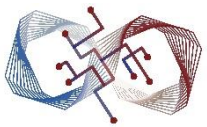
sm90 的三个改动方向



1. 提高每次读取的复用率，即增大 tile
2. 让操作数**不经过寄存器**，A / B 放到 shared memory
3. Tensor Core 计算时 CUDA core 空闲，让两者并行工作

Tensor Core 不依赖任何线程的操作，因此可以改为异步

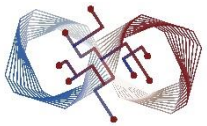
wgmma 指令



wgmma：一条指令驱动整个 SM 的 4 组 Tensor Core

```
wgmma.mma_async.sync.aligned.m64n64k16.f32.f16.f16
    {d0, d1, ..., d31},   // D: 64×64 f32, 64*64/128 = 32 个寄存器 / 线程
    a_desc,               // A: 64×16, 64-bit descriptor, 指向 smem
    b_desc,               // B: 16×64
    scale_d,              // 1 -> D = A×B + D ; 0 -> D = A×B
    1, 1,                 // imm-scale-a / imm-scale-b: 对 A/B 整体取正负
    0, 0;                 // imm-trans-a / imm-trans-b: 读的时候是否转置
```

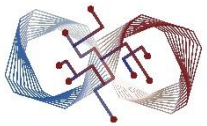
wgmma 指令



wgmma：一条指令驱动整个 SM 的 4 组 Tensor Core

```
wgmma.mma_async.sync.aligned.m64n64k16.f32.f16.f16
    {d0, d1, ..., d31},   // D: 64×64 f32, 64*64/128 = 32 个寄存器 / 线程
    a_desc,               // A: 64×16, 64-bit descriptor, 指向 smem
    b_desc,               // B: 16×64
    scale_d,              // 1 -> D = A×B + D ; 0 -> D = A×B
    1, 1,                 // imm-scale-a / imm-scale-b: 对 A/B 整体取正负
    0, 0;                 // imm-trans-a / imm-trans-b: 读的时候是否转置
```

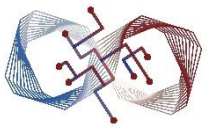
wgmma 指令



wgmma：一条指令驱动整个 SM 的 4 组 Tensor Core

```
wgmma.mma_async.sync.aligned.m64n64k16.f32.f16.f16
    {d0, d1, ..., d31},   // D: 64×64 f32, 64*64/128 = 32 个寄存器 / 线程
    a_desc,               // A: 64×16, 64-bit descriptor, 指向 smem
    b_desc,               // B: 16×64
    scale_d,              // 1 -> D = A×B + D ; 0 -> D = A×B
    1, 1,                 // imm-scale-a / imm-scale-b: 对 A/B 整体取正负
    0, 0;                 // imm-trans-a / imm-trans-b: 读的时候是否转置
```

wgmma 的字段 (1/2)



wgmma：一条指令驱动整个 SM 的 4 组 Tensor Core

```
wgmma.mma_async.sync.aligned.m64n64k16.f32.f16.f16
    {d0, d1, ..., d31},   // D: 64×64 f32, 64*64/128 = 32 个寄存器 / 线程
    a_desc,               // A: 64×16, 64-bit descriptor, 指向 smem
    b_desc,               // B: 16×64
    scale_d,              // 1 -> D = A×B + D ; 0 -> D = A×B
    1, 1,                 // imm-scale-a / imm-scale-b: 对 A/B 整体取正负
    0, 0;                 // imm-trans-a / imm-trans-b: 读的时候是否转置
```

wgmma：4 个 warp 组成一个 warpgroup，一条指令调动整个 SM 的 4 组 Tensor Core，tile 为 64×N

aligned：128 个线程必须全部执行，发散是未定义行为

mma_async：指令发射完**立刻返回**

warpgroup 的构成

一条指令驱动整个 SM

wgmma 把 4 组 Tensor Core 一起调动，tile 是 64 × N

M 为什么是 64

每个 warp 仍然分到 16 行，4 个 warp 合起来 M = 64

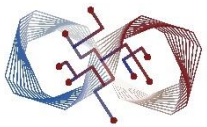
aligned 的含义

128 个线程必须**全部**执行这条指令，发散是未定义行为

1 SM = 4 个子分区 = 4 组 tensor core

4 个 warp 组成 1 个 warpgroup，一条指令调动整个 SM

wgmma 的字段 (2/2)



wgmma：一条指令驱动整个 SM 的 4 组 Tensor Core

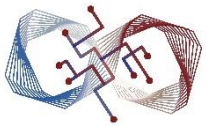
```
wgmma.mma_async.sync.aligned.m64n64k16.f32.f16.f16
    {d0, d1, ..., d31},   // D: 64×64 f32, 64*64/128 = 32 个寄存器 / 线程
    a_desc,               // A: 64×16, 64-bit descriptor, 指向 smem
    b_desc,               // B: 16×64
    scale_d,              // 1 -> D = A×B + D ; 0 -> D = A×B
    1, 1,                 // imm-scale-a / imm-scale-b: 对 A/B 整体取正负
    0, 0;                 // imm-trans-a / imm-trans-b: 读的时候是否转置
```

m64n64k16：4 warp 每个还是 16 行 → M=64；**K 由数据类型决定，长度恒为 32B**（fp16/bf16→16，tf32→8，fp8/int8→32，b1→256）；N 从 8 起、步长 8、最大 256

· 不再有 .row.col：主序信息移入 descriptor 与末尾两个 trans 字段。A 可 K-major 或 M-major，B 可 K-major 或 N-major

· a_desc / b_desc：A/B 不再是寄存器 fragment，而是 64-bit matrix descriptor 指向 smem（也有 RS mode，A 可从寄存器读）

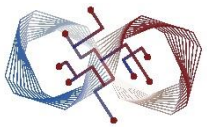
wgmma 的三个限制



必须 `-arch=sm_90a`，**缺少末尾的 a 会找不到 wgmma**。不向前兼容，**Blackwell 不支持 wgmma**

D 仍留在寄存器：m64n256 每线程要 256/2 = **128 个 f32 寄存器**，而单线程上限是 255

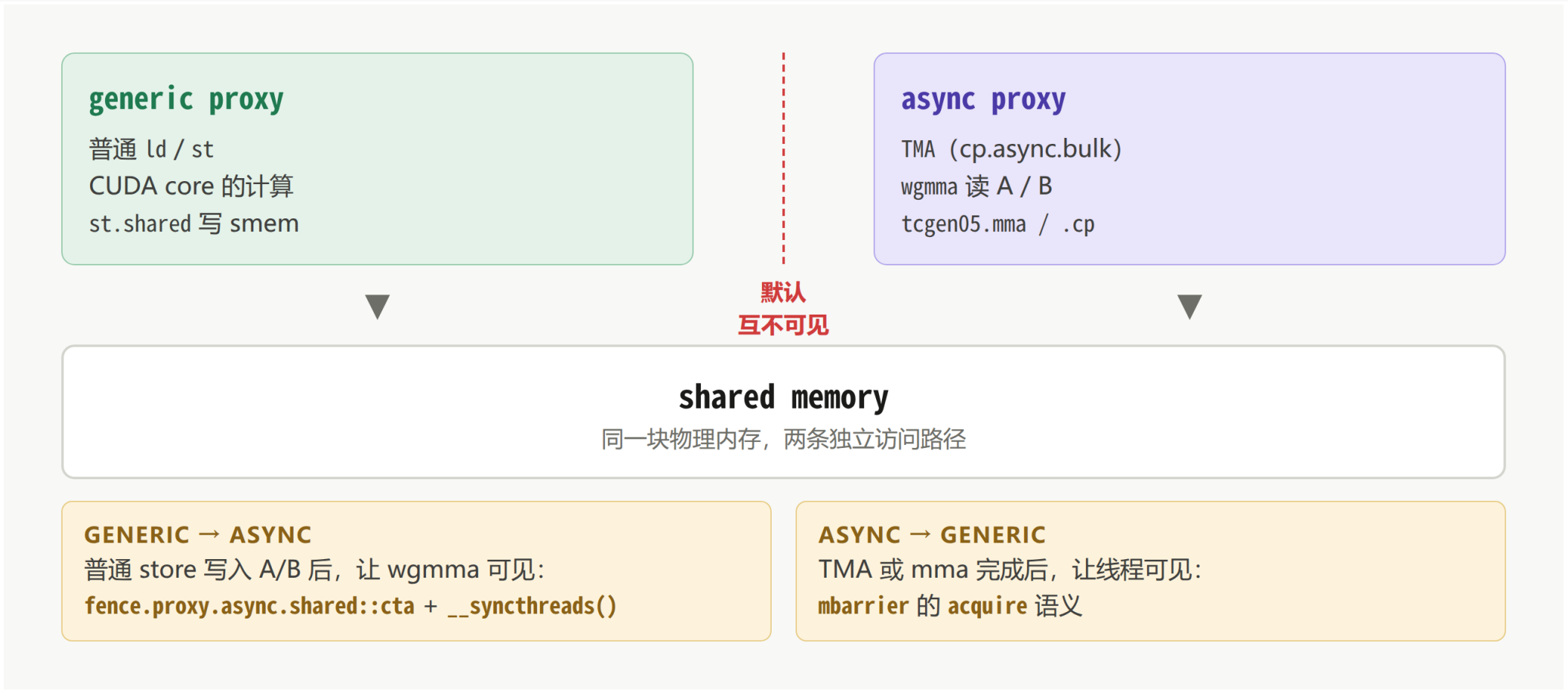
想一想：异步带来的两个问题



① 什么时候可以安全地读累加器 D?

② 什么时候可以安全地覆写 A/B 所在的 shared memory?

generic proxy 与 async proxy

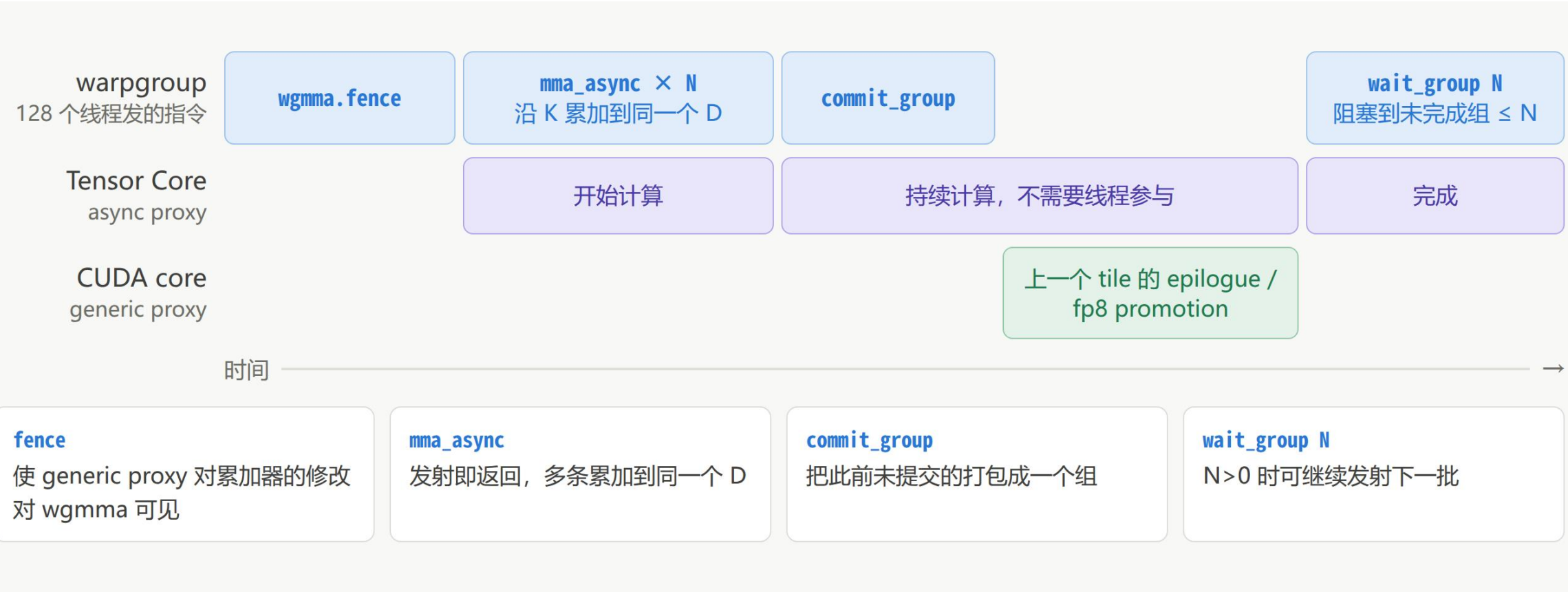


generic proxy：普通 ld/st、CUDA core 计算

async proxy：TMA、wgmma

两条通道独立访问同一块内存。不显式同步时，async proxy **不保证看到** generic proxy 刚写入的数据，反之亦然

wgmma 的生命周期



和 cp.async 的 commit / wait 是同一个模式

wgmma 的五条使用规则

	违反即未定义行为	对应的做法
1	从发射到对应的 wait 完成，**不能碰累加器 D** —— 读和写都不行	没有补救，只能不碰
2	wait 完成前，**不能改 wgmma 正在读的那块 shared memory**。沿 K 多级缓冲时最容易踩	`mbarrier` 确保写进空闲 buffer
3	A / B 若由普通 `st.shared` 写入，wgmma **默认看不见**这些写	`fence.proxy.async` `+ __syncthreads()`
4	A / B 若由 TMA 写入，同属 async proxy，**天然可见**	只需 `mbarrier` 不用额外 `fence`
5	CUDA core 改过 D 之后，下一轮 wgmma 之前**必须重新 fence**	`wgmma.fence`

实际写代码时看到的大多是 **`ptxas warning`**：它会替你把 wgmma 串行化，程序还是对的，但性能塌掉

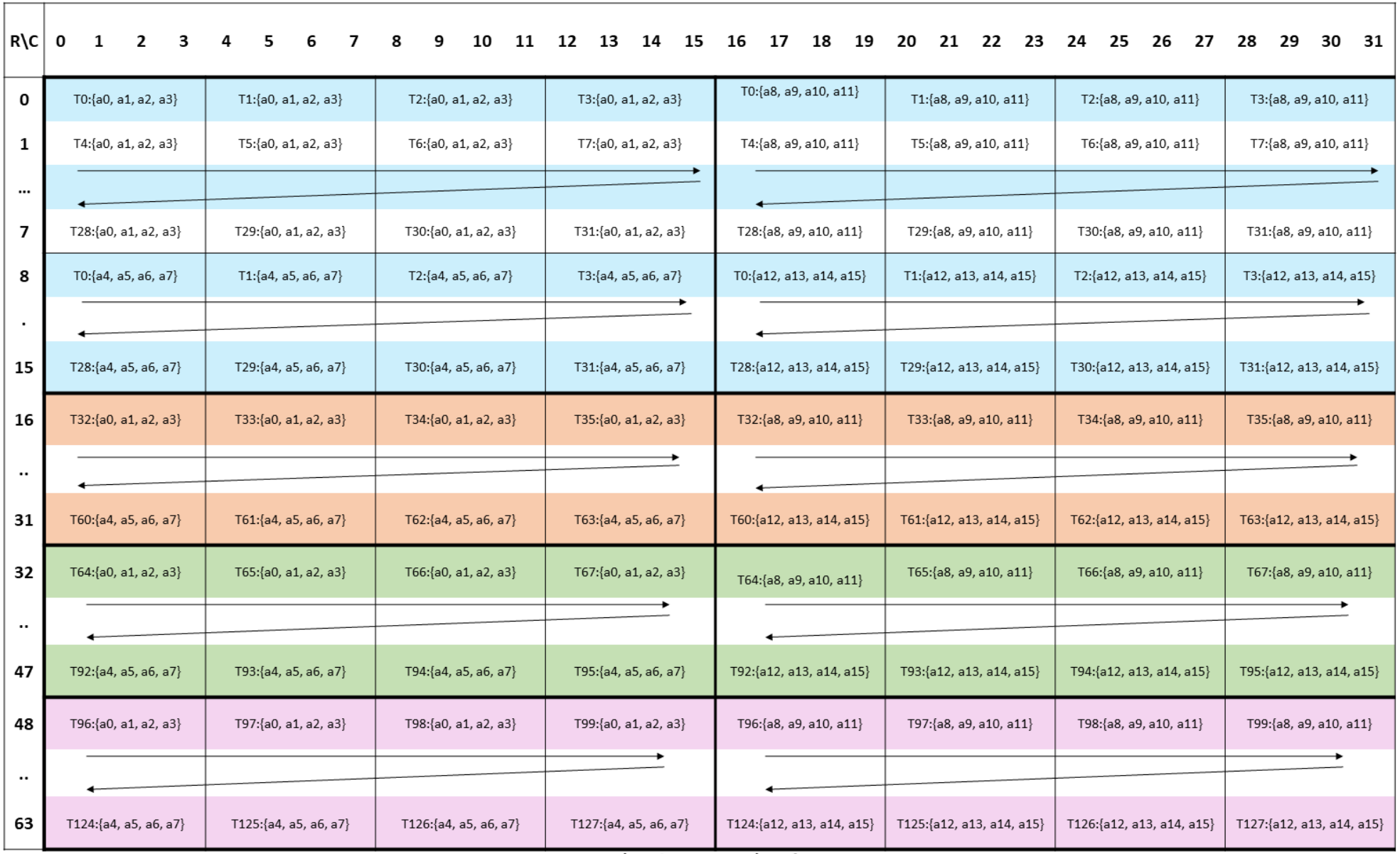
commit 与 wait 之间的窗口

commit_group 之后、wait_group 之前，本 warpgroup 的 CUDA core 是空闲的

可用于上一个 tile 的 **epilogue** 或 **fp8 promotion**

进一步让不同 warpgroup 分工：producer 专职发 TMA，consumer 专职发 wgmma

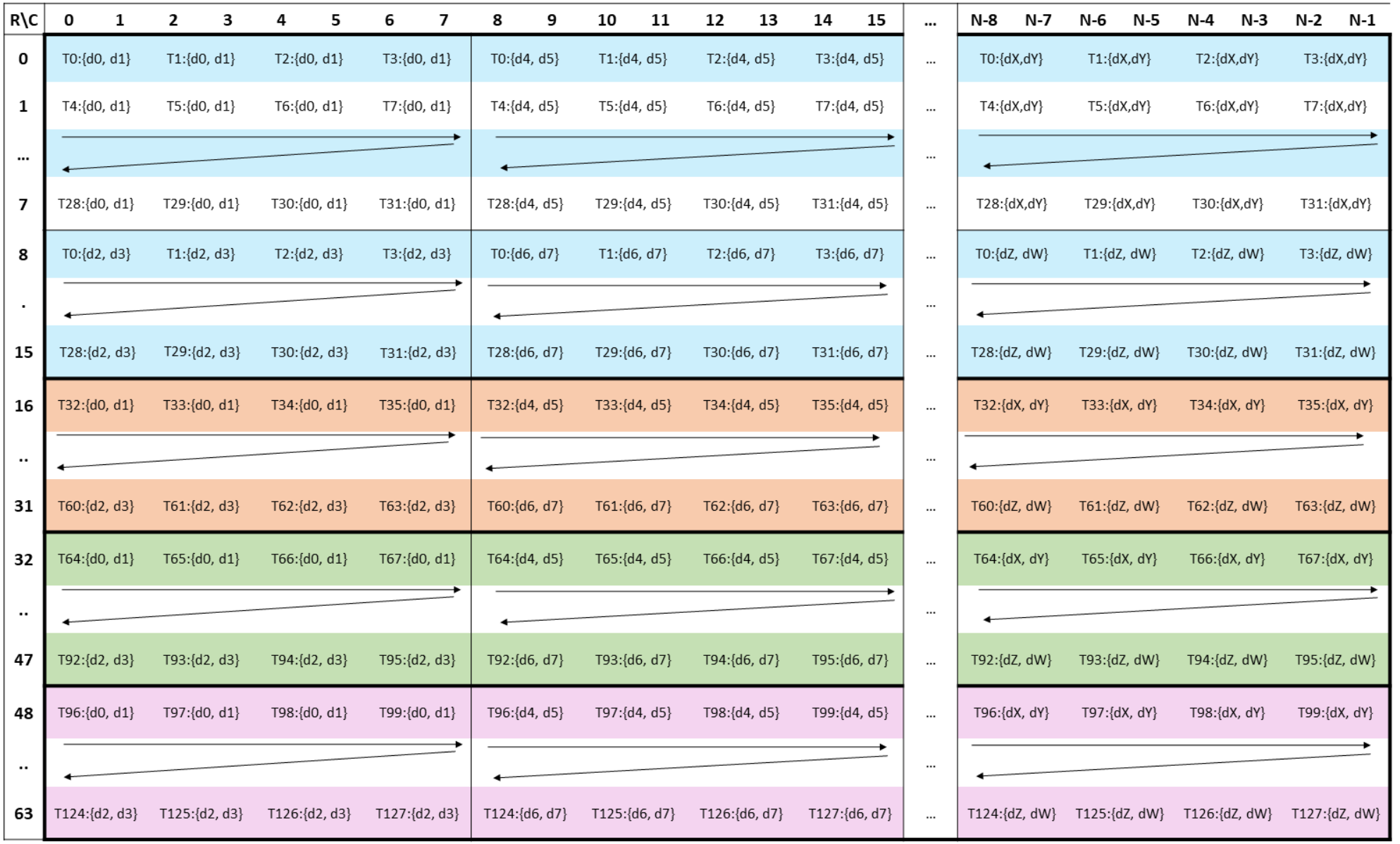
sm90 的 fragment：A（fp8，from register）



R\C	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	T0:{a0, a1, a2, a3}				T1:{a0, a1, a2, a3}				T2:{a0, a1, a2, a3}				T3:{a0, a1, a2, a3}				T0:{a8, a9, a10, a11}				T1:{a8, a9, a10, a11}				T2:{a8, a9, a10, a11}				T3:{a8, a9, a10, a11}			
1	T4:{a0, a1, a2, a3}				T5:{a0, a1, a2, a3}				T6:{a0, a1, a2, a3}				T7:{a0, a1, a2, a3}				T4:{a8, a9, a10, a11}				T5:{a8, a9, a10, a11}				T6:{a8, a9, a10, a11}				T7:{a8, a9, a10, a11}			
...																																
7	T28:{a0, a1, a2, a3}				T29:{a0, a1, a2, a3}				T30:{a0, a1, a2, a3}				T31:{a0, a1, a2, a3}				T28:{a8, a9, a10, a11}				T29:{a8, a9, a10, a11}				T30:{a8, a9, a10, a11}				T31:{a8, a9, a10, a11}			
8	T0:{a4, a5, a6, a7}				T1:{a4, a5, a6, a7}				T2:{a4, a5, a6, a7}				T3:{a4, a5, a6, a7}				T0:{a12, a13, a14, a15}				T1:{a12, a13, a14, a15}				T2:{a12, a13, a14, a15}				T3:{a12, a13, a14, a15}			
.																																
15	T28:{a4, a5, a6, a7}				T29:{a4, a5, a6, a7}				T30:{a4, a5, a6, a7}				T31:{a4, a5, a6, a7}				T28:{a12, a13, a14, a15}				T29:{a12, a13, a14, a15}				T30:{a12, a13, a14, a15}				T31:{a12, a13, a14, a15}			
16	T32:{a0, a1, a2, a3}				T33:{a0, a1, a2, a3}				T34:{a0, a1, a2, a3}				T35:{a0, a1, a2, a3}				T32:{a8, a9, a10, a11}				T33:{a8, a9, a10, a11}				T34:{a8, a9, a10, a11}				T35:{a8, a9, a10, a11}			
..																																
31	T60:{a4, a5, a6, a7}				T61:{a4, a5, a6, a7}				T62:{a4, a5, a6, a7}				T63:{a4, a5, a6, a7}				T60:{a12, a13, a14, a15}				T61:{a12, a13, a14, a15}				T62:{a12, a13, a14, a15}				T63:{a12, a13, a14, a15}			
32	T64:{a0, a1, a2, a3}				T65:{a0, a1, a2, a3}				T66:{a0, a1, a2, a3}				T67:{a0, a1, a2, a3}				T64:{a8, a9, a10, a11}				T65:{a8, a9, a10, a11}				T66:{a8, a9, a10, a11}				T67:{a8, a9, a10, a11}			
..																																
47	T92:{a4, a5, a6, a7}				T93:{a4, a5, a6, a7}				T94:{a4, a5, a6, a7}				T95:{a4, a5, a6, a7}				T92:{a12, a13, a14, a15}				T93:{a12, a13, a14, a15}				T94:{a12, a13, a14, a15}				T95:{a12, a13, a14, a15}			
48	T96:{a0, a1, a2, a3}				T97:{a0, a1, a2, a3}				T98:{a0, a1, a2, a3}				T99:{a0, a1, a2, a3}				T96:{a8, a9, a10, a11}				T97:{a8, a9, a10, a11}				T98:{a8, a9, a10, a11}				T99:{a8, a9, a10, a11}			
..																																
63	T124:{a4, a5, a6, a7}				T125:{a4, a5, a6, a7}				T126:{a4, a5, a6, a7}				T127:{a4, a5, a6, a7}				T124:{a12, a13, a14, a15}				T125:{a12, a13, a14, a15}				T126:{a12, a13, a14, a15}				T127:{a12, a13, a14, a15}			

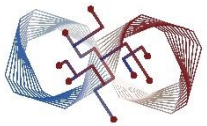
(tid % 128) : fragments

sm90 的 fragment： D



(tid % 128) : fragments

descriptor 的作用



descriptor 告诉硬件：数据已按**硬件能识别的固定格式**（canonical layout）摆放

布局的基本单位：**core matrix = 8 行 × 16 字节**

整个大矩阵 = 这些小块的二维平铺

core matrix

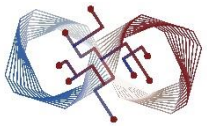
对任意数据类型，令 `T = 128 / sizeof_bits(元素)`，把元素打包成 16 字节单元

fp16 → 8 个一组；fp8 → 16 个一组

后续所说的格子都是这种 **16B 单元，与数据类型无关**

core matrix = 8 行 × 1 个 16B 单元

LBO 与 SBO

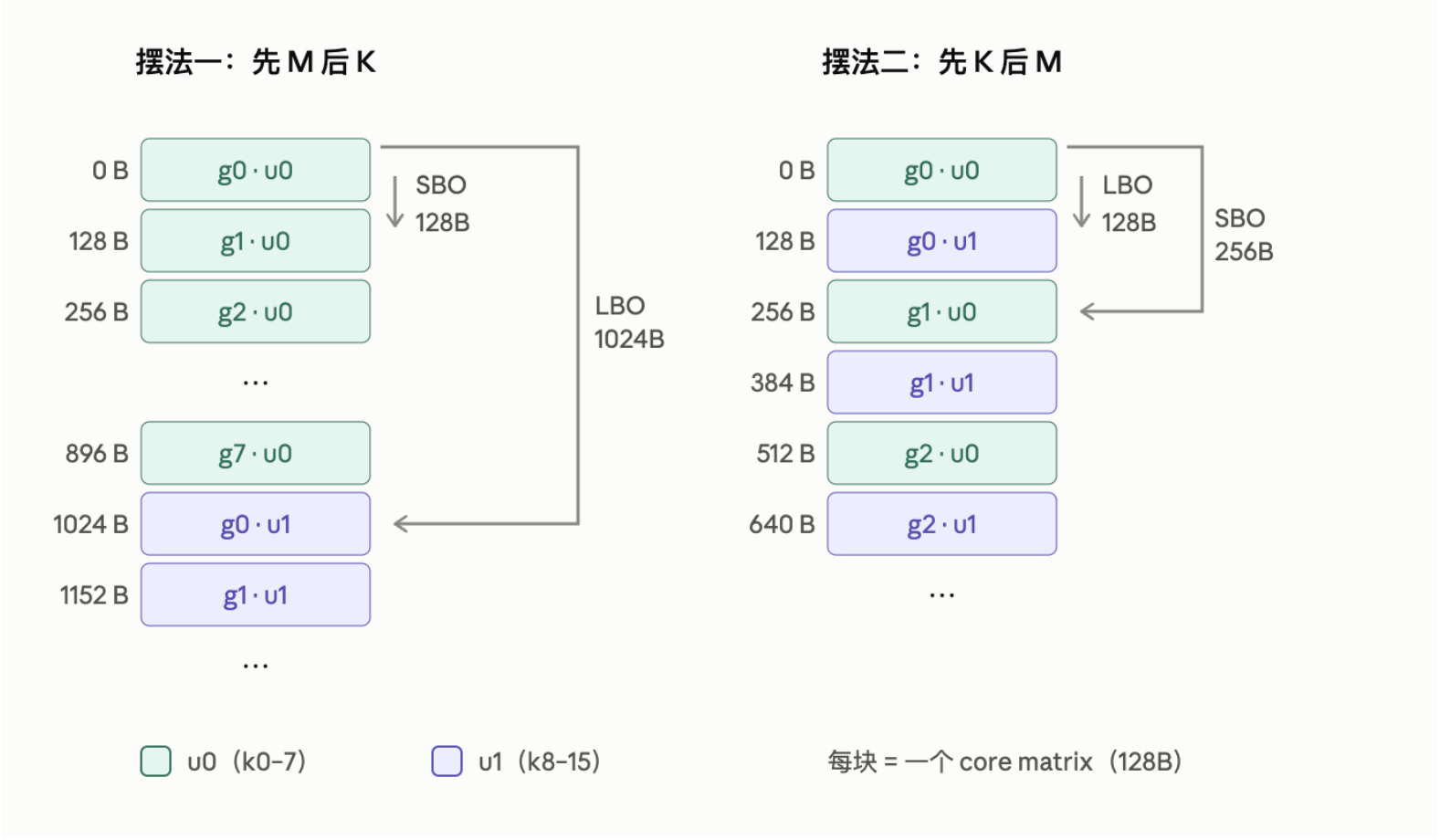


小块是二维的，平铺时**需要两个方向的步长**

LBO (leading dimension byte offset) + **SBO** (stride dimension byte offset)

加上起始地址、swizzle 模式与 base offset，刚好装进 64 位

两种 canonical layout

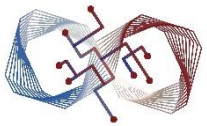


例：64×16 fp16 的 A tile，T=8，K 方向 2 个单元、M 方向 8 个行组 → 8×2 = 16 个 core matrix

摆法一（先 M 后 K）：SBO = 128B → 编码 8；LBO = 1024B → 编码 64

摆法二（先 K 后 M）：SBO = 256B → 编码 16；LBO = 128B → 编码 8

descriptor 定义



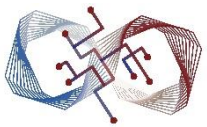
64 位的 matrix descriptor

```
union GmmaDescriptor {
    uint64_t desc_;
    struct {
        uint16_t start_address_       : 14, : 2;  // bit  0-13: smem 起始地址 >> 4
        uint16_t leading_byte_offset_ : 14, : 2;  // bit 16-29: LBO >> 4
        uint16_t stride_byte_offset_  : 14, : 2;  // bit 32-45: SBO >> 4
        uint8_t  : 1, base_offset_    : 3,  : 4;  // bit 49-51: swizzle 用的基偏移
        uint8_t  : 6, layout_type_    : 2;        // bit 62-63: 0=无 1=128B 2=64B 3=32B
    } bitfield;
};
```

所有字节值填进字段前都要 **右移 4 位**，因为地址与偏移必须 16B 对齐，低 4 位恒为 0

`base_offset` 处理起始地址没落在 swizzle 边界上的情况；实践中直接对齐 shared memory 让它恒为 0

descriptor 定义



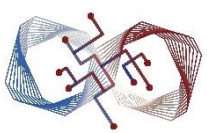
64 位的 matrix descriptor

```
union GmmaDescriptor {
    uint64_t desc_;
    struct {
        uint16_t start_address_       : 14, : 2;  // bit  0-13: smem 起始地址 >> 4
        uint16_t leading_byte_offset_ : 14, : 2;  // bit 16-29: LBO >> 4
        uint16_t stride_byte_offset_  : 14, : 2;  // bit 32-45: SBO >> 4
        uint8_t  : 1, base_offset_    : 3,  : 4;  // bit 49-51: swizzle 用的基偏移
        uint8_t  : 6, layout_type_    : 2;        // bit 62-63: 0=无 1=128B 2=64B 3=32B
    } bitfield;
};
```

所有字节值填进字段前都要 **右移 4 位**，因为地址与偏移必须 16B 对齐，低 4 位恒为 0

`base_offset` 处理起始地址没落在 swizzle 边界上的情况；实践中直接对齐 shared memory 让它恒为 0

descriptor 定义



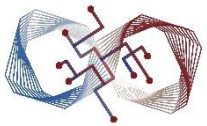
64 位的 matrix descriptor

```
union GmmaDescriptor {
    uint64_t desc_;
    struct {
        uint16_t start_address_       : 14, : 2;  // bit  0-13: smem 起始地址 >> 4
        uint16_t leading_byte_offset_ : 14, : 2;  // bit 16-29: LBO >> 4
        uint16_t stride_byte_offset_  : 14, : 2;  // bit 32-45: SBO >> 4
        uint8_t  : 1, base_offset_    : 3,  : 4;  // bit 49-51: swizzle 用的基偏移
        uint8_t  : 6, layout_type_    : 2;        // bit 62-63: 0=无 1=128B 2=64B 3=32B
    } bitfield;
};
```

所有字节值填进字段前都要 **右移 4 位**，因为地址与偏移必须 16B 对齐，低 4 位恒为 0

`base_offset` 处理起始地址没落在 swizzle 边界上的情况；实践中直接对齐 shared memory 让它恒为 0

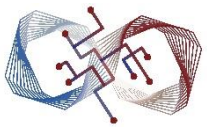
Recall：bank 与 bank group



shared memory 有 32 个 bank，每个 4 B，一次访问 32 个 bank 性能最佳

core matrix 以 16B 为单位且 16B 对齐，这 4 个 bank 合称一个 **bank group**

128 B 行跨度下的冲突



ldmatrix.x4：32 lane × 16 B = **512 B** · shared memory **128 B / cycle** · 理论下限 **4 cycle**

row stride = 32 B

16×16 fp16 tile

bank step = 32 B / 4 = **8 banks**

lane 0–15 (col 0) → group 0, 8, 16, 24

lane 16–31 (col 8) → group 4, 12, 20, 28

bank 占用

0-3	4-7	8-11	12-15	16-19	20-23	24-27	28-31

8 组 bank，各 4 个请求 → **4 cycle**

等于下限，无损失

row stride = 128 B

64×64 fp16 分块

bank step = 128 B / 4 = 32 ≡ **0**

lane 0–15 (col 0) → group 0 **全部相同**

lane 16–31 (col 8) → group 4 **全部相同**

bank 占用

0-3	4-7	8-11	12-15	16-19	20-23	24-27	28-31

2 组 bank，各 16 个请求 → **16 cycle**

4 倍损失，24 个 bank 闲置

128B 对齐的 tiling（fp16 就是 64×64 分块，非常常见）下，每一行都从同一个 bank 起

一个 core matrix 的 8 行落在同一组 bank 上，形成 **8 路冲突**

swizzle 的地址位映射

地址共 10 位，bit 0 最低：

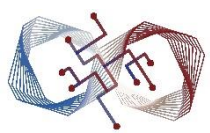
bit 0–1：bank 内偏移 bit 2–3：bank group 内的 offset

· bit 4–6：8 个 bank group 的 id ← 同一个 core matrix 里完全相撞

· bit 7–9：core matrix 的行

· 把 bit 4–6 与 bit 7–9 异或，等于给每一行换一个 bank group，且是一一映射

swizzle 前后的槽位分布



swizzle 前

物理槽位 s = u

	u0	u1	u2	u3	u4	u5	u6	u7
r=0	0	1	2	3	4	5	6	7
r=1	0	1	2	3	4	5	6	7
r=2	0	1	2	3	4	5	6	7
r=3	0	1	2	3	4	5	6	7
r=4	0	1	2	3	4	5	6	7
r=5	0	1	2	3	4	5	6	7
r=6	0	1	2	3	4	5	6	7
r=7	0	1	2	3	4	5	6	7

同一列的 8 行全是 **0**

swizzle 后

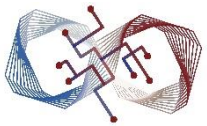
物理槽位 s = u ⊕ r

	u0	u1	u2	u3	u4	u5	u6	u7
r=0	0	1	2	3	4	5	6	7
r=1	1	0	3	2	5	4	7	6
r=2	2	3	0	1	6	7	4	5
r=3	3	2	1	0	7	6	5	4
r=4	4	5	6	7	0	1	2	3
r=5	5	4	7	6	1	0	3	2
r=6	6	7	4	5	2	3	0	1
r=7	7	6	5	4	3	2	1	0

同一列的 8 行取遍 **0–7**

swizzle 后**每一列都是 0–7 的一个排列**，因此无冲突

手搓 ① 脚手架



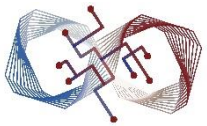
手搓 sm90：脚手架（编译要带 -arch=sm_90a，注意那个 a）

```
__device__ __forceinline__ void warpgroup_fence() {
    asm volatile("wgmma.fence.sync.aligned;\n" ::: "memory");
}
__device__ __forceinline__ void warpgroup_commit() {
    asm volatile("wgmma.commit_group.sync.aligned;\n" ::: "memory");
}
template <int N>
__device__ __forceinline__ void warpgroup_wait() {
    asm volatile("wgmma.wait_group.sync.aligned %0;\n" :: "n"(N) : "memory");
}
```

```
__device__ __forceinline__ uint64_t make_desc(const void* smem_ptr,
                                              uint32_t lbo, uint32_t sbo,
                                              uint32_t layout_type) {
    GmmaDescriptor d{};
    d.bitfield.start_address_       = __cvta_generic_to_shared(smem_ptr) >> 4;
    d.bitfield.leading_byte_offset_ = lbo >> 4;
    d.bitfield.stride_byte_offset_  = sbo >> 4;
    d.bitfield.base_offset_         = 0;
    d.bitfield.layout_type_         = layout_type;   // 0 = no swizzle
    return d.desc_;
}
```

本例不做 swizzle，便于观察 shared memory 中的实际排布；生产环境的 smem layout 由 TMA 处理

手搓 ① 脚手架



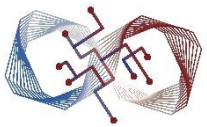
手搓 sm90：脚手架（编译要带 -arch=sm_90a，注意那个 a）

```
__device__ __forceinline__ void warpgroup_fence() {
    asm volatile("wgmma.fence.sync.aligned;\n" ::: "memory");
}
__device__ __forceinline__ void warpgroup_commit() {
    asm volatile("wgmma.commit_group.sync.aligned;\n" ::: "memory");
}
template <int N>
__device__ __forceinline__ void warpgroup_wait() {
    asm volatile("wgmma.wait_group.sync.aligned %0;\n" :: "n"(N) : "memory");
}
```

```
__device__ __forceinline__ uint64_t make_desc(const void* smem_ptr,
                                              uint32_t lbo, uint32_t sbo,
                                              uint32_t layout_type) {
    GmmaDescriptor d{};
    d.bitfield.start_address_       = __cvta_generic_to_shared(smem_ptr) >> 4;
    d.bitfield.leading_byte_offset_ = lbo >> 4;
    d.bitfield.stride_byte_offset_  = sbo >> 4;
    d.bitfield.base_offset_         = 0;
    d.bitfield.layout_type_         = layout_type;   // 0 = no swizzle
    return d.desc_;
}
```

本例不做 swizzle，便于观察 shared memory 中的实际排布；生产环境的 smem layout 由 TMA 处理

手搓 ② smem 摆放



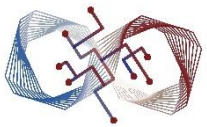
手搓 sm90：smem 摆放，按 K 方向优先连续

```
// A: 逻辑上 64×16 K-major (A[m][k])
//    core matrix 坐标 (bm, bk) = (m/8, k/8)，共 8×2 块
__device__ int a_off(int m, int k) {
    int brick = (m / 8) * 2 + (k / 8);
    return brick * 64 + (m % 8) * 8 + (k % 8);
}
// -> K 方向相邻砖差 128B (LBO=128)，M 方向相邻砖差 256B (SBO=256)

// B: 逻辑上 16×8 K-major (B[k][n])，共 1×2 块
__device__ int b_off(int k, int n) {
    int brick = k / 8;
    return brick * 64 + (n % 8) * 8 + (k % 8);
}
// -> LBO=128; N 方向只有一块砖，SBO 填 256 占位

__shared__ __align__(128) half sA[64 * 16];
__shared__ __align__(128) half sB[16 *  8];
```

按 K 方向优先连续排列，K 方向相邻砖相差 128 B（LBO），M 方向相邻砖相差 256 B（SBO）

B 只有一块砖，SBO 填 256 占位

手搓 ② smem 摆放



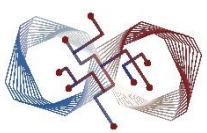
手搓 sm90：smem 摆放，按 K 方向优先连续

```
// A: 逻辑上 64×16 K-major (A[m][k])
//    core matrix 坐标 (bm, bk) = (m/8, k/8)，共 8×2 块
__device__ int a_off(int m, int k) {
    int brick = (m / 8) * 2 + (k / 8);
    return brick * 64 + (m % 8) * 8 + (k % 8);
}
// -> K 方向相邻砖差 128B (LBO=128)，M 方向相邻砖差 256B (SBO=256)

// B: 逻辑上 16×8 K-major (B[k][n])，共 1×2 块
__device__ int b_off(int k, int n) {
    int brick = k / 8;
    return brick * 64 + (n % 8) * 8 + (k % 8);
}
// -> LBO=128; N 方向只有一块砖，SBO 填 256 占位

__shared__ __align__(128) half sA[64 * 16];
__shared__ __align__(128) half sB[16 *  8];
```

按 K 方向优先连续排列，K 方向相邻砖相差 128 B（LBO），M 方向相邻砖相差 256 B（SBO）

B 只有一块砖，SBO 填 256 占位

手搓 ③ 发射

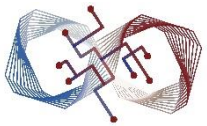


手搓 sm90：发射三拍

```
float d[4] = {0.f, 0.f, 0.f, 0.f};

uint64_t desc_a = make_desc(sA, 128, 256, /*layout=*/0);
uint64_t desc_b = make_desc(sB, 128, 256, /*layout=*/0);

__syncthreads();                                  // 数据是整个 block 写的
asm volatile("fence.proxy.async.shared::cta;\n");   // 普通 store -> async proxy 可见

warpgroup_fence();
asm volatile(
    "{\n"
    ".reg .pred p;\n"
    "setp.ne.b32 p, %6, 0;\n"
    "wgmma.mma_async.sync.aligned.m64n8k16.f32.f16.f16 "
    "{%0, %1, %2, %3}, %4, %5, p, 1, 1, 0, 0;\n"
    "}\n"
    : "+f"(d[0]), "+f"(d[1]), "+f"(d[2]), "+f"(d[3])
    : "l"(desc_a), "l"(desc_b), "r"(0));   // scale_d=0: 首轮，清零累加
warpgroup_commit();
warpgroup_wait<0>();
```

1. `__syncthreads()` + `fence.proxy.async.shared::cta` ← 对应规则 3
2. `warpgroup_fence()` ← 对应规则 5
3. asm 本体 + `commit` + `wait<0>`

手搓 ③ 发射

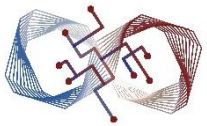


手搓 sm90：发射三拍

```
float d[4] = {0.f, 0.f, 0.f, 0.f};

uint64_t desc_a = make_desc(sA, 128, 256, /*layout=*/0);
uint64_t desc_b = make_desc(sB, 128, 256, /*layout=*/0);

__syncthreads();                                  // 数据是整个 block 写的
asm volatile("fence.proxy.async.shared::cta;\n");   // 普通 store -> async proxy 可见

warpgroup_fence();
asm volatile(
    "{\n"
    ".reg .pred p;\n"
    "setp.ne.b32 p, %6, 0;\n"
    "wgmma.mma_async.sync.aligned.m64n8k16.f32.f16.f16 "
    "{%0, %1, %2, %3}, %4, %5, p, 1, 1, 0, 0;\n"
    "}\n"
    : "+f"(d[0]), "+f"(d[1]), "+f"(d[2]), "+f"(d[3])
    : "l"(desc_a), "l"(desc_b), "r"(0));   // scale_d=0: 首轮，清零累加
warpgroup_commit();
warpgroup_wait<0>();
```

1. `__syncthreads()` + `fence.proxy.async.shared::cta` ← 对应规则 3
2. `warpgroup_fence()` ← 对应规则 5
3. asm 本体 + `commit` + `wait<0>`

手搓 ③ 发射

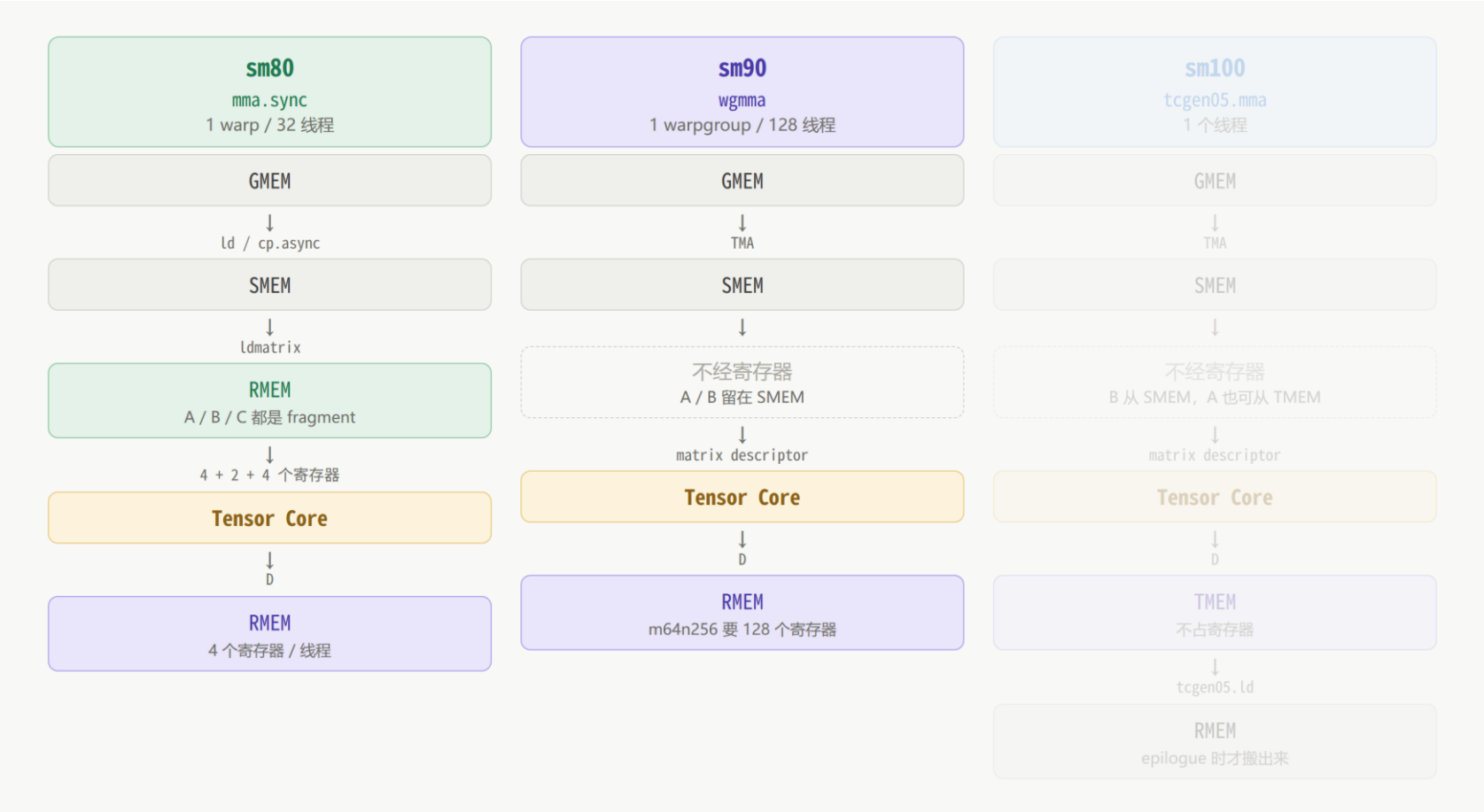


手搓 sm90：发射三拍

```
float d[4] = {0.f, 0.f, 0.f, 0.f};

uint64_t desc_a = make_desc(sA, 128, 256, /*layout=*/0);
uint64_t desc_b = make_desc(sB, 128, 256, /*layout=*/0);

__syncthreads();                                  // 数据是整个 block 写的
asm volatile("fence.proxy.async.shared::cta;\n");   // 普通 store -> async proxy 可见

warpgroup_fence();
asm volatile(
    "{\n"
    ".reg .pred p;\n"
    "setp.ne.b32 p, %6, 0;\n"
    "wgmma.mma_async.sync.aligned.m64n8k16.f32.f16.f16 "
    "{%0, %1, %2, %3}, %4, %5, p, 1, 1, 0, 0;\n"
    "}\n"
    : "+f"(d[0]), "+f"(d[1]), "+f"(d[2]), "+f"(d[3])
    : "l"(desc_a), "l"(desc_b), "r"(0));   // scale_d=0: 首轮，清零累加
warpgroup_commit();
warpgroup_wait<0>();
```

1. `__syncthreads()` + `fence.proxy.async.shared::cta` ← 对应规则 3
2. `warpgroup_fence()` ← 对应规则 5
3. asm 本体 + `commit` + `wait<0>`

sm90 小结

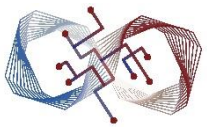


三代的 GMEM → SMEM 这一段都在，变的是操作数从哪里读、累加器放在哪里

sm80：ldmatrix 把 A/B/C 装进寄存器，D 也回寄存器

sm90：A/B 留在 SMEM，通过 64-bit descriptor 直接送入 Tensor Core，D 仍在寄存器

三代对比（sm80 / sm90）



	mma.sync sm80 · Ampere	wgmma sm90 · Hopper	tcgen05.mma sm100 · Blackwell
发射者	1 warp / 32 线程	1 warpgroup / 128 线程	—
A 来自	RMEM fragment	SMEM descriptor	—
B 来自	RMEM fragment	SMEM descriptor	—
C / D 在	RMEM	RMEM	—
完成同步	隐式 warp sync	commit_group / wait_group	—
最大 tile	m16n8k16	m64n256k16	—
脚手架	fragment 图 + ldmatrix + swizzle	matrix descriptor + TMA	—

发射者从 1 warp 变为 1 warpgroup

操作数从全部在寄存器，变为 A / B 留在 shared memory 并通过 descriptor 访问

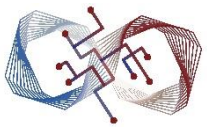
完成同步从隐式 warp sync 变为 commit_group / wait_group

单条指令的最大 tile 从 m16n8k16 增长到 m64n256k16

2.4 sm100：tcgen05

累加器移入 TMEM，改为单线程发射

wgmma 遗留的两个问题



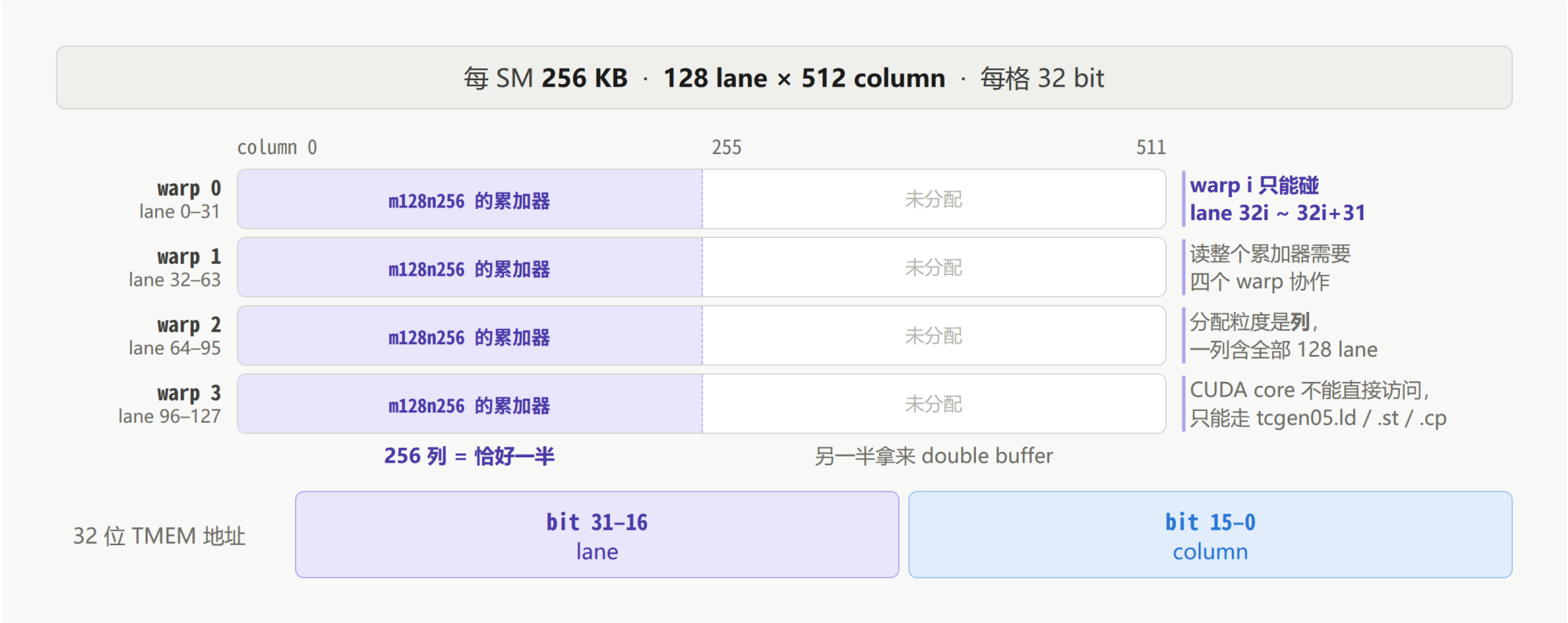
wgmma **解决的**：scaling、CUDA core 与 Tensor Core 的 overlap、fp8 计算支持

遗留的两个问题：

1. 累加器仍在寄存器里，占用大量寄存器，且对这些寄存器的操作会引发 wgmma 串行化
2. 发射一条 wgmma 仍需 warpgroup 的 128 个线程共同执行

但是，**Tensor Core 本身不需要线程参与计算?**

TMEM 的组织方式



每个 SM 一块 **256 KB** 的 tensor memory。 32 位地址索引：**高 16 位 lane，低 16 位 column**

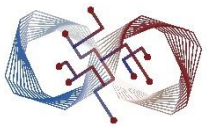
按 **128 lane × 512 column** 组织，每格 32 bit，与整块寄存器堆同一量级。

· 不能被 CUDA core 直接访问，必须使用 tcgen05.ld / st / cp

· warp i 只能访问 lane 32i ~ 32i+31，读取整个累加器需要 warpgroup 四个 warp 协作

· 分配粒度是列：分配一列 = 分配它全部 128 个 lane

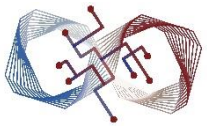
m128n256 与 TMEM 容量



m128n256 的累加器 = 128 lane × 256 col = **恰好半个 TMEM**

适合 double buffer，Tensor Core 不必等累加器搬走即可继续计算

TMEM 的分配



TMEM 的分配与释放（编译要带 -arch=sm_100a）

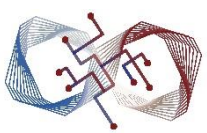
```
__shared__ uint32_t s_taddr[1];   // 注意：分配结果落在 smem 里，不是寄存器

// alloc 是 warp 级指令，需要一个完整的 warp 执行
if (warp_id == 0) {
    uint32_t dst = (uint32_t)__cvta_generic_to_shared(s_taddr);
    asm volatile("tcgen05.alloc.cta_group::1.sync.aligned.shared::cta.b32 [%0], %1;"
                 :: "r"(dst), "r"(NUM_COLS));            // 列数必须是 2 的幂且 >= 32
    // 声明本 CTA 不再 alloc，把配额让给同 SM 的其他 CTA
    asm volatile("tcgen05.relinquish_alloc_permit.cta_group::1.sync.aligned;");
}
__syncthreads();
uint32_t taddr = s_taddr[0];

// ... 用完必须显式释放 ...
__syncthreads();                 // 等所有 warp 的 epilogue 结束
if (warp_id == 0)
    asm volatile("tcgen05.dealloc.cta_group::1.sync.aligned.b32 %0, %1;"
                 :: "r"(taddr), "r"(NUM_COLS));
```

三个细节：

· 列数必须是 **2 的幂且 ≥ 32**

· 分配结果（TMEM 基地址）**不是写进寄存器，而是写到一个 smem 地址**

TMEM 的分配



TMEM 的分配与释放（编译要带 -arch=sm_100a）

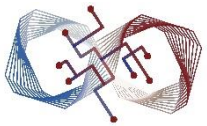
```
__shared__ uint32_t s_taddr[1];   // 注意：分配结果落在 smem 里，不是寄存器

// alloc 是 warp 级指令，需要一个完整的 warp 执行
if (warp_id == 0) {
    uint32_t dst = (uint32_t)__cvta_generic_to_shared(s_taddr);
    asm volatile("tcgen05.alloc.cta_group::1.sync.aligned.shared::cta.b32 [%0], %1;"
                 :: "r"(dst), "r"(NUM_COLS));            // 列数必须是 2 的幂且 >= 32
    // 声明本 CTA 不再 alloc，把配额让给同 SM 的其他 CTA
    asm volatile("tcgen05.relinquish_alloc_permit.cta_group::1.sync.aligned;");
}
__syncthreads();
uint32_t taddr = s_taddr[0];

// ... 用完必须显式释放 ...
__syncthreads();                  // 等所有 warp 的 epilogue 结束
if (warp_id == 0)
    asm volatile("tcgen05.dealloc.cta_group::1.sync.aligned.b32 %0, %1;"
                 :: "r"(taddr), "r"(NUM_COLS));
```

三个细节：

列数必须是 **2 的幂且 ≥ 32**

alloc 是 warp 级指令，需要一个完整的 warp 执行

TMEM 的分配



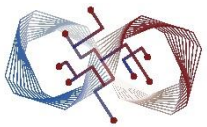
TMEM 的分配与释放（编译要带 -arch=sm_100a）

```
__shared__ uint32_t s_taddr[1];   // 注意：分配结果落在 smem 里，不是寄存器

// alloc 是 warp 级指令，需要一个完整的 warp 执行
if (warp_id == 0) {
    uint32_t dst = (uint32_t)__cvta_generic_to_shared(s_taddr);
    asm volatile("tcgen05.alloc.cta_group::1.sync.aligned.shared::cta.b32 [%0], %1;"
                 :: "r"(dst), "r"(NUM_COLS));            // 列数必须是 2 的幂且 >= 32
    // 声明本 CTA 不再 alloc，把配额让给同 SM 的其他 CTA
    asm volatile("tcgen05.relinquish_alloc_permit.cta_group::1.sync.aligned;");
}
__syncthreads();
uint32_t taddr = s_taddr[0];

// ... 用完必须显式释放 ...
__syncthreads();                 // 等所有 warp 的 epilogue 结束
if (warp_id == 0)
    asm volatile("tcgen05.dealloc.cta_group::1.sync.aligned.b32 %0, %1;"
                 :: "r"(taddr), "r"(NUM_COLS));
```

三个细节：

· **必须显式 dealloc**

tcgen05.mma 指令



tcgen05.mma：不再需要 .sync.aligned，单线程就能发

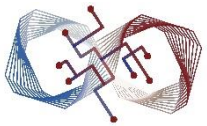
```
tcgen05.mma.cta_group::1.kind::mxf8f6f4  [taddr_d], [taddr_a], b_desc, idesc,
                                         [tmem_sfa], [tmem_sfb], p;
```

不再需要 **.sync.aligned**

cta_group::1/2：支持两个 CTA 共同计算一条 MMA，也可以只用一个

kind：粗粒度的类型类，**精确类型由 idesc 确定**

tcgen05 的 kind



kind	操作数类型	K (dense)
`f16`	fp16 / bf16	16
`tf32`	tf32	8
`i8`	s8 / u8	32
`f8f6f4`	e4m3 / e5m2 / e3m2 / e2m3 / e2m1	32
`mxf8f6f4`	同上，外加硬件 block scaling	32
`mxf4` / `mxf4nvf4`	fp4，外加硬件 block scaling	64

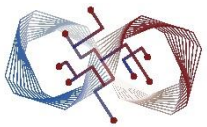
带 mx 前缀的，scale factor 由硬件读取并反量化

kind 是粗粒度的类型类，精确类型由 idesc 确定

带 mx 前缀的类别，scale factor 由硬件读取并反量化，不需要 CUDA core 参与

K 随元素位宽变化：f16 是 16，tf32 是 8，8 bit 类是 32，fp4 类是 64

tcgen05.mma 的其余操作数



`[taddr_d]`：累加器是一个 **TMEM 地址**

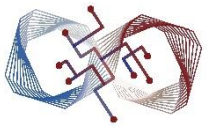
`a_desc / b_desc`：**B 一定来自 smem descriptor**；A 可以 smem descriptor 或 TMEM

`idesc`：32 位 instruction descriptor，运行时确定

`p` (enable_input_d)：谓词，决定累加还是重新做 gemm

另有 `.ws`（weight stationary）变体，支持 M=32/64 和 B 的驻留复用

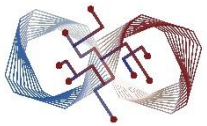
idesc 的位域



32 位的 instruction descriptor，运行时构造

```
// m128n(N)k16，fp16 输入，f32 累加，两边都 K-major 不转置
constexpr uint32_t make_idesc(uint32_t M, uint32_t N) {
    return (1u << 4)          // bit 4-5:   D 类型，1 = f32
         | (0u << 7)          // bit 7-9:   A 类型，kind::f16 下 0 = fp16，1 = bf16
         | (0u << 10)         // bit 10-12: B 类型，同上
                              // bit 13/14: A/B 取负；bit 15/16: A/B 转置
         | ((N >> 3) << 17)   // bit 17-22: N，以 8 为单位
         | ((M >> 4) << 24);  // bit 24-28: M，以 16 为单位
}
```

想一想：如何知道 mma 已完成



Hopper 有 `wgmma.wait_group`，tcgen05 呢?

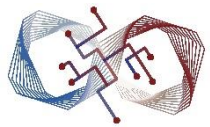
mbarrier 的构成

mbarrier 是 Hopper 引入的概念，本场此前没有用到

smem 里的一个 **64 位对象**，维护三件事：**phase**：0,1,2,… 单调递增，**只看奇偶性**，用于翻转

arrival count：还差多少次 arrive； tx-count：还差多少字节的异步传输，同时归零就翻转

mbarrier 的初始化与等待



mbarrier 的初始化与等待

```
__shared__ uint64_t mbar[1];
uint32_t mbar_u32 = (uint32_t)__cvta_generic_to_shared(mbar);

if (tid == 0) {
    asm volatile("mbarrier.init.shared::cta.b64 [%0], %1;" :: "r"(mbar_u32), "r"(1));
    asm volatile("fence.mbarrier_init.release.cluster;");
}
__syncthreads();
```

```
__device__ void mbarrier_wait(uint32_t mbar_u32, uint32_t parity) {
    asm volatile(
        "{\n"
        ".reg .pred P;\n"
        "WAIT:\n"
        "mbarrier.try_wait.parity.acquire.cta.shared::cta.b64 P, [%0], %1;\n"
        "@P bra.uni DONE;\n"
        "bra.uni WAIT;\n"
        "DONE:\n"
        "}\n" :: "r"(mbar_u32), "r"(parity));
}
```

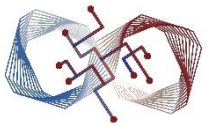
`try_wait`：尝试等待，硬件可以挂起该线程去执行其它工作

`parity`：只关注相位

`acquire`：此条之前的内存操作在返回后可见（相反是 relaxed）

`cta.shared`：只在本线程块内成立；`.cluster` 可与同 cluster 内多个 CTA 同步

tcgen05.commit



wgmma.wait_group

发射的 warpgroup 128 线程 → 发 wgmma → 同一个 warpgroup 等 128 线程

谁发的谁等。发射者和消费者**必须是同一批线程**，想让别人去做 epilogue 就得额外协调

tcgen05.commit + mbarrier

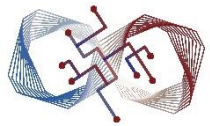
1 个线程 elect.sync → 发 mma 再发 commit → epilogue 的 4 个 warp 128 线程

commit 的语义是「我发过的 tcgen05 全完成时，在这个 mbarrier 上 arrive 一次」。 **任何线程都能等**，发射者和消费者彻底解耦

发过的 tcgen05 全部完成时，在这个 mbarrier 上 arrive 一次

```
tcgen05.commit.cta_group::1.mbarrier::arrive::one.shared::cluster.b64 [%0];
```

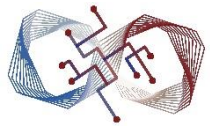
一个完整的 pipeline 骨架



一个完整的 pipeline 骨架

```
// (1) TMA 把 A/B 搬进 smem，完成汇报到 tma_mbar
// (2) 等 TMA
mbarrier_wait(tma_mbar, phase);

// (3) 一个线程沿 K 连发 mma
if (elected) {
    for (int k = 0; k < BLOCK_K / MMA_K; ++k)
        tcgen05_mma(...);
    // (4) 打包汇报到 mma_mbar
    tcgen05_commit(mma_mbar);
}

// (5) 需要读累加器的线程等 mma_mbar
mbarrier_wait(mma_mbar, phase);
phase ^= 1;
```

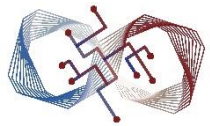
一个完整的 pipeline 骨架



一个完整的 pipeline 骨架

```
// (1) TMA 把 A/B 搬进 smem，完成汇报到 tma_mbar
// (2) 等 TMA
mbarrier_wait(tma_mbar, phase);

// (3) 一个线程沿 K 连发 mma
if (elected) {
    for (int k = 0; k < BLOCK_K / MMA_K; ++k)
        tcgen05_mma(...);
    // (4) 打包汇报到 mma_mbar
    tcgen05_commit(mma_mbar);
}

// (5) 需要读累加器的线程等 mma_mbar
mbarrier_wait(mma_mbar, phase);
phase ^= 1;
```

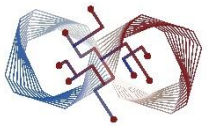
一个完整的 pipeline 骨架



一个完整的 pipeline 骨架

```
// (1) TMA 把 A/B 搬进 smem，完成汇报到 tma_mbar
// (2) 等 TMA
mbarrier_wait(tma_mbar, phase);

// (3) 一个线程沿 K 连发 mma
if (elected) {
    for (int k = 0; k < BLOCK_K / MMA_K; ++k)
        tcgen05_mma(...);
    // (4) 打包汇报到 mma_mbar
    tcgen05_commit(mma_mbar);
}

// (5) 需要读累加器的线程等 mma_mbar
mbarrier_wait(mma_mbar, phase);
phase ^= 1;
```

tcgen05.ld



tcgen05.ld：.sync.aligned 又回来了，搬数据需要每个线程各自参与

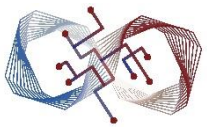
```
float t[8];
uint32_t addr = taddr + ((warp_id * 32) << 16) + col;   // 高 16 位 lane，低 16 位 column

asm volatile("tcgen05.ld.sync.aligned.32x32b.x8.b32 "
             "{%0,%1,%2,%3,%4,%5,%6,%7}, [%8];"
             : "=f"(t[0]), "=f"(t[1]), "=f"(t[2]), "=f"(t[3]),
               "=f"(t[4]), "=f"(t[5]), "=f"(t[6]), "=f"(t[7])
             : "r"(addr));

asm volatile("tcgen05.wait::ld.sync.aligned;");   // ld 本身也是异步的，读寄存器前必须等
```

`tcgen05.ld.sync.aligned.32x32b.x8.b32 {r0..r7}, [taddr];`

`.sync.aligned` 重新出现，与 mma 形成对照：

mma 由单线程发射，但把数据从 TMEM 搬到寄存器需要每个线程各自参与

tcgen05.ld



tcgen05.ld：.sync.aligned 又回来了，搬数据需要每个线程各自参与

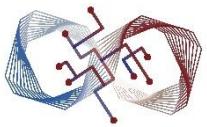
```
float t[8];
uint32_t addr = taddr + ((warp_id * 32) << 16) + col;   // 高 16 位 lane，低 16 位 column

asm volatile("tcgen05.ld.sync.aligned.32x32b.x8.b32 "
             "{%0,%1,%2,%3,%4,%5,%6,%7}, [%8];"
             : "=f"(t[0]), "=f"(t[1]), "=f"(t[2]), "=f"(t[3]),
               "=f"(t[4]), "=f"(t[5]), "=f"(t[6]), "=f"(t[7])
             : "r"(addr));

asm volatile("tcgen05.wait::ld.sync.aligned;");   // ld 本身也是异步的，读寄存器前必须等
```

sync.aligned 重新出现，与 mma 形成对照；同时 ld 本身也是异步的，读寄存器前必须 tcgen05.wait

mma 由单线程发射，但把数据从 TMEM 搬到寄存器需要每个线程各自参与

shape：.32x32b / .16x64b / .16x128b / .16x256b / .16x32bx2。num：.x1 到 .x128（2 的幂）

tcgen05 的同步规则



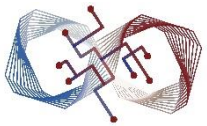
generic / async proxy 的可见性规则**原样继承**：smem 上用普通 `st.shared` 写入的数据要让 `tcgen05.mma` 看到，需要 `fence.proxy.async.shared::cta`

`tcgen05.fence::after_thread_sync`：跨线程场景（线程 A 发 mma、B 组做 epilogue）下，在 `__syncthreads()` / mbarrier 建立顺序**之后**、执行自己的 tcgen05 指令**之前**

`tcgen05.st` 方向相反，用法对称

`tcgen05.cp` 是 smem → TMEM 的异步拷贝，主要用途是把 **scale factor 矩阵**搬进 TMEM

手搓：七步流程



01

分配

`mbarrier.init`
`tcgen05.alloc`

1 个线程 init，1 个完整 warp alloc

02

写 A / B 进 smem

`st.shared`

全体线程；生产环境用 TMA

03

让 async proxy 看见

`fence.proxy.async`
`__syncthreads()`

同时广播 taddr

04

发射

`elect.sync`
`tcgen05.mma`
`tcgen05.commit`

只要 1 个线程

05

等完成

`mbarrier.try_wait`
`.parity`

全体等；acquire 保证累加器可见

06

读累加器

`tcgen05.fence`
`::after_thread_sync`
`tcgen05.ld + wait::ld`

每个 warp 只能读自己的 32 条 lane

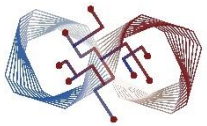
07

释放

`__syncthreads()`
`tcgen05.dealloc`

等所有读结束，1 个 warp 释放

手搓 ① 分配与写入



手搓 sm100：m128n16k16、fp16、no swizzle、cta_group::1

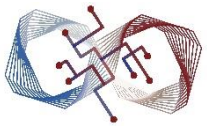
```
// (1) mbarrier 初始化（1 个线程）+ TMEM 分配（1 个完整 warp）
if (warp_id == 0) {
    if (lane == 0) {
        asm volatile("mbarrier.init.shared::cta.b64 [%0], %1;" :: "r"(mbar_u32), "r"(1));
        asm volatile("fence.mbarrier_init.release.cluster;");
    }
    uint32_t dst = (uint32_t)__cvta_generic_to_shared(s_taddr);
    asm volatile("tcgen05.alloc.cta_group::1.sync.aligned.shared::cta.b32 [%0], %1;"
                 :: "r"(dst), "r"(32));       // n=16 也必须分配最小值 32 列
    asm volatile("tcgen05.relinquish_alloc_permit.cta_group::1.sync.aligned;");
}

// (2) 全体线程用普通 store 写 A/B（生产环境用 TMA）
for (int i = tid; i < M * K; i += blockDim.x) sA[a_off(i / K, i % K)] = gA[i];
for (int i = tid; i < N * K; i += blockDim.x) sB[b_off(i / K, i % K)] = gB[i];

// (3) generic proxy 的写 -> async proxy 可见；同时广播 taddr
asm volatile("fence.proxy.async.shared::cta;");
__syncthreads();
const uint32_t taddr = s_taddr[0];
```

参数：m128n16k16、fp16、no swizzle、cta_group::1

mbarrier init 由 1 个线程执行，TMEM alloc 需要一个完整的 warp

A / B 用普通 st.shared 写入，生产环境改用 TMA

手搓 ① 分配与写入



手搓 sm100：m128n16k16、fp16、no swizzle、cta_group::1

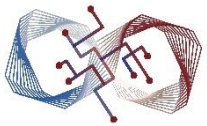
```
// (1) mbarrier 初始化（1 个线程）+ TMEM 分配（1 个完整 warp）
if (warp_id == 0) {
    if (lane == 0) {
        asm volatile("mbarrier.init.shared::cta.b64 [%0], %1;" :: "r"(mbar_u32), "r"(1));
        asm volatile("fence.mbarrier_init.release.cluster;");
    }
    uint32_t dst = (uint32_t)__cvta_generic_to_shared(s_taddr);
    asm volatile("tcgen05.alloc.cta_group::1.sync.aligned.shared::cta.b32 [%0], %1;"
                 :: "r"(dst), "r"(32));       // n=16 也必须分配最小值 32 列
    asm volatile("tcgen05.relinquish_alloc_permit.cta_group::1.sync.aligned;");
}

// (2) 全体线程用普通 store 写 A/B（生产环境用 TMA）
for (int i = tid; i < M * K; i += blockDim.x) sA[a_off(i / K, i % K)] = gA[i];
for (int i = tid; i < N * K; i += blockDim.x) sB[b_off(i / K, i % K)] = gB[i];

// (3) generic proxy 的写 -> async proxy 可见；同时广播 taddr
asm volatile("fence.proxy.async.shared::cta;");
__syncthreads();
const uint32_t taddr = s_taddr[0];
```

参数：m128n16k16、fp16、no swizzle、cta_group::1

mbarrier init 由 1 个线程执行，TMEM alloc 需要一个完整的 warp

A / B 用普通 st.shared 写入，生产环境改用 TMA

手搓 ① 分配与写入



手搓 sm100：m128n16k16、fp16、no swizzle、cta_group::1

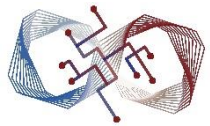
```
// (1) mbarrier 初始化（1 个线程）+ TMEM 分配（1 个完整 warp）
if (warp_id == 0) {
    if (lane == 0) {
        asm volatile("mbarrier.init.shared::cta.b64 [%0], %1;" :: "r"(mbar_u32), "r"(1));
        asm volatile("fence.mbarrier_init.release.cluster;");
    }
    uint32_t dst = (uint32_t)__cvta_generic_to_shared(s_taddr);
    asm volatile("tcgen05.alloc.cta_group::1.sync.aligned.shared::cta.b32 [%0], %1;"
                 :: "r"(dst), "r"(32));       // n=16 也必须分配最小值 32 列
    asm volatile("tcgen05.relinquish_alloc_permit.cta_group::1.sync.aligned;");
}

// (2) 全体线程用普通 store 写 A/B（生产环境用 TMA）
for (int i = tid; i < M * K; i += blockDim.x) sA[a_off(i / K, i % K)] = gA[i];
for (int i = tid; i < N * K; i += blockDim.x) sB[b_off(i / K, i % K)] = gB[i];

// (3) generic proxy 的写 -> async proxy 可见；同时广播 taddr
asm volatile("fence.proxy.async.shared::cta;");
__syncthreads();
const uint32_t taddr = s_taddr[0];
```

参数：m128n16k16、fp16、no swizzle、cta_group::1

mbarrier init 由 1 个线程执行，TMEM alloc 需要一个完整的 warp

A / B 用普通 st.shared 写入，生产环境改用 TMA

手搓 ② 发射与等待



手搓 sm100：发射与 epilogue

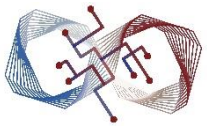
```
// (4) 一个线程发射 mma + commit
uint32_t elected;
asm volatile("{\n.reg .pred P;\nelect.sync _|P, 0xFFFFFFFF;\nselp.b32 %0, 1, 0, P;\n}"
             : "=r"(elected));
if (warp_id == 0 && elected) {
    asm volatile("tcgen05.fence::after_thread_sync;");   // 跨线程规则
    asm volatile(
        "{\n.reg .pred p;\nsetp.ne.b32 p, %4, 0;\n"
        "tcgen05.mma.cta_group::1.kind::f16 [%0], %1, %2, %3, p;\n}\n"
        :: "r"(taddr), "l"(a_desc), "l"(b_desc), "r"(idesc), "r"(0));
    asm volatile("tcgen05.commit.cta_group::1.mbarrier::arrive::one"
                 ".shared::cluster.b64 [%0];" :: "r"(mbar_u32) : "memory");
}

// (5) 全体等 mma 完成; acquire 语义保证累加器可见
mbarrier_wait(mbar_u32, 0);

// (6) epilogue: 每个 warp 只能读自己的 32 条 lane
asm volatile("tcgen05.fence::after_thread_sync;");
for (int n = 0; n < N; n += 8) { /* tcgen05.ld -> wait::ld -> 写回 gD */ }

// (7) 所有读结束后，一个 warp 释放 TMEM
__syncthreads();
if (warp_id == 0) asm volatile("tcgen05.dealloc.cta_group::1.sync.aligned.b32 %0, %1;"
                               :: "r"(taddr), "r"(32));
```

`elect.sync` 选出一个线程发射 mma 与 commit

跨线程场景下，线程同步之后、执行自己的 tcgen05 指令之前需要 `tcgen05.fence::after_thread_sync`

全体线程等 mbarrier，acquire 语义保证累加器可见

手搓 ② 发射与等待



手搓 sm100：发射与 epilogue

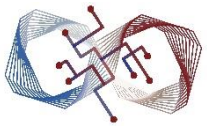
```
// (4) 一个线程发射 mma + commit
uint32_t elected;
asm volatile("{\n.reg .pred P;\nelect.sync _|P, 0xFFFFFFFF;\nselp.b32 %0, 1, 0, P;\n}"
             : "=r"(elected));
if (warp_id == 0 && elected) {
    asm volatile("tcgen05.fence::after_thread_sync;");   // 跨线程规则
    asm volatile(
        "{\n.reg .pred p;\nsetp.ne.b32 p, %4, 0;\n"
        "tcgen05.mma.cta_group::1.kind::f16 [%0], %1, %2, %3, p;\n}\n"
        :: "r"(taddr), "l"(a_desc), "l"(b_desc), "r"(idesc), "r"(0));
    asm volatile("tcgen05.commit.cta_group::1.mbarrier::arrive::one"
                 ".shared::cluster.b64 [%0];" :: "r"(mbar_u32) : "memory");
}

// (5) 全体等 mma 完成; acquire 语义保证累加器可见
mbarrier_wait(mbar_u32, 0);

// (6) epilogue: 每个 warp 只能读自己的 32 条 lane
asm volatile("tcgen05.fence::after_thread_sync;");
for (int n = 0; n < N; n += 8) { /* tcgen05.ld -> wait::ld -> 写回 gD */ }

// (7) 所有读结束后，一个 warp 释放 TMEM
__syncthreads();
if (warp_id == 0) asm volatile("tcgen05.dealloc.cta_group::1.sync.aligned.b32 %0, %1;"
                               :: "r"(taddr), "r"(32));
```

`elect.sync` 选出一个线程发射 mma 与 commit

跨线程场景下，线程同步之后、执行自己的 tcgen05 指令之前需要 `tcgen05.fence::after_thread_sync`

全体线程等 mbarrier，acquire 语义保证累加器可见

手搓 ② 发射与等待



手搓 sm100：发射与 epilogue

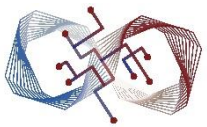
```
// (4) 一个线程发射 mma + commit
uint32_t elected;
asm volatile("{\n.reg .pred P;\nelect.sync _|P, 0xFFFFFFFF;\nselp.b32 %0, 1, 0, P;\n}"
             : "=r"(elected));
if (warp_id == 0 && elected) {
    asm volatile("tcgen05.fence::after_thread_sync;");   // 跨线程规则
    asm volatile(
        "{\n.reg .pred p;\nsetp.ne.b32 p, %4, 0;\n"
        "tcgen05.mma.cta_group::1.kind::f16 [%0], %1, %2, %3, p;\n}\n"
        :: "r"(taddr), "l"(a_desc), "l"(b_desc), "r"(idesc), "r"(0));
    asm volatile("tcgen05.commit.cta_group::1.mbarrier::arrive::one"
                 ".shared::cluster.b64 [%0];" :: "r"(mbar_u32) : "memory");
}

// (5) 全体等 mma 完成; acquire 语义保证累加器可见
mbarrier_wait(mbar_u32, 0);

// (6) epilogue: 每个 warp 只能读自己的 32 条 lane
asm volatile("tcgen05.fence::after_thread_sync;");
for (int n = 0; n < N; n += 8) { /* tcgen05.ld -> wait::ld -> 写回 gD */ }

// (7) 所有读结束后，一个 warp 释放 TMEM
__syncthreads();
if (warp_id == 0) asm volatile("tcgen05.dealloc.cta_group::1.sync.aligned.b32 %0, %1;"
                               :: "r"(taddr), "r"(32));
```

`elect.sync` 选出一个线程发射 mma 与 commit

跨线程场景下，线程同步之后、执行自己的 tcgen05 指令之前需要 `tcgen05.fence::after_thread_sync`

全体线程等 mbarrier，acquire 语义保证累加器可见

手搓 ③ epilogue 与释放



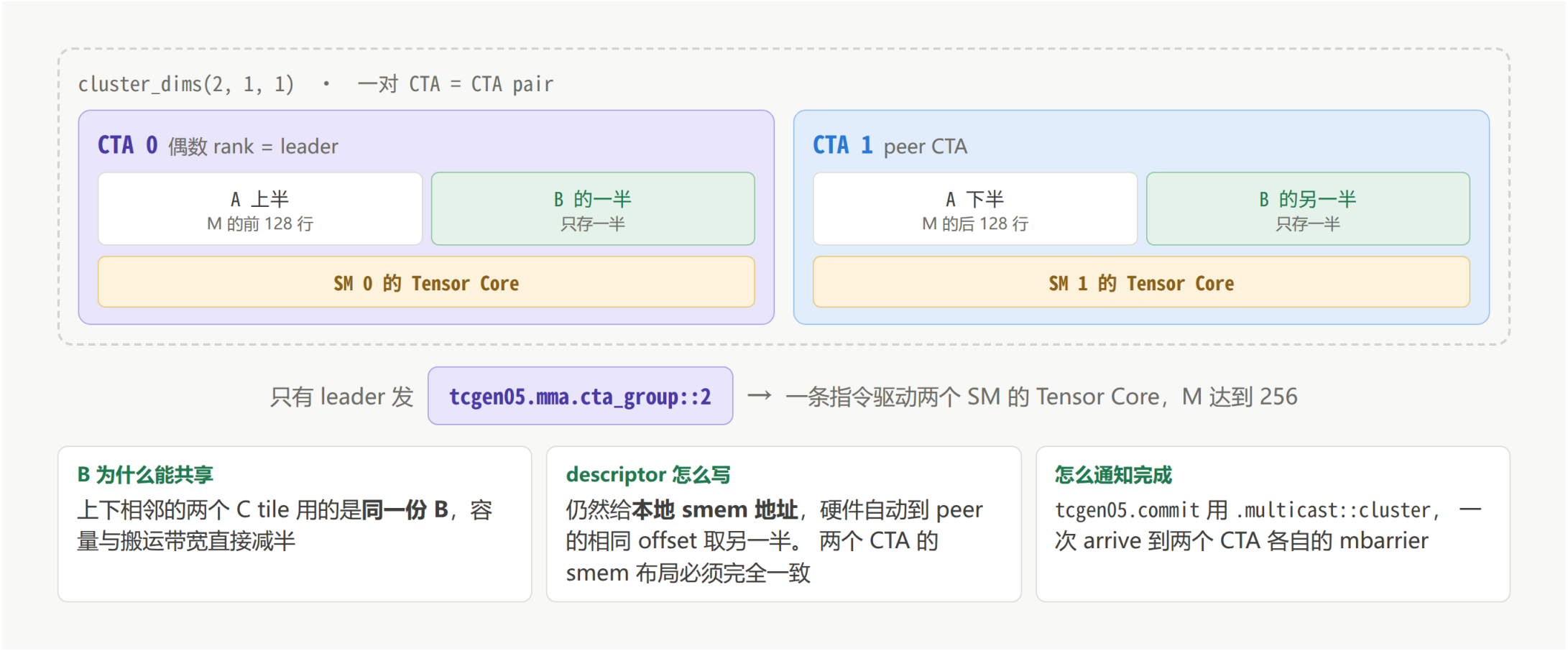
手搓 sm100：发射与 epilogue

```
// (4) 一个线程发射 mma + commit
uint32_t elected;
asm volatile("{\n.reg .pred P;\nelect.sync _|P, 0xFFFFFFFF;\nselp.b32 %0, 1, 0, P;\n}"
             : "=r"(elected));
if (warp_id == 0 && elected) {
    asm volatile("tcgen05.fence::after_thread_sync;");   // 跨线程规则
    asm volatile(
        "{\n.reg .pred p;\nsetp.ne.b32 p, %4, 0;\n"
        "tcgen05.mma.cta_group::1.kind::f16 [%0], %1, %2, %3, p;\n}\n"
        :: "r"(taddr), "l"(a_desc), "l"(b_desc), "r"(idesc), "r"(0));
    asm volatile("tcgen05.commit.cta_group::1.mbarrier::arrive::one"
                 ".shared::cluster.b64 [%0];" :: "r"(mbar_u32) : "memory");
}

// (5) 全体等 mma 完成; acquire 语义保证累加器可见
mbarrier_wait(mbar_u32, 0);

// (6) epilogue: 每个 warp 只能读自己的 32 条 lane
asm volatile("tcgen05.fence::after_thread_sync;");
for (int n = 0; n < N; n += 8) { /* tcgen05.ld -> wait::ld -> 写回 gD */ }

// (7) 所有读结束后，一个 warp 释放 TMEM
__syncthreads();
if (warp_id == 0) asm volatile("tcgen05.dealloc.cta_group::1.sync.aligned.b32 %0, %1;"
                               :: "r"(taddr), "r"(32));
```

每个 warp 只能读自己的 32 条 lane

所有读结束后由一个 warp 执行 `tcgen05.dealloc`

2-CTA MMA

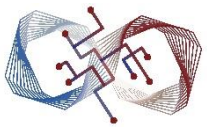


MMA 形状越大计算强度越高，但 M=128、N=256 时输入侧压力已经很大

一对相邻 CTA 若计算 C 中上下相邻的两个 tile，**使用的 B 是同一份**

共同计算时 B 只搬一份、每个 CTA 存半份，**smem 容量与搬运带宽减半**，M 可达 256

三代对比（完整）



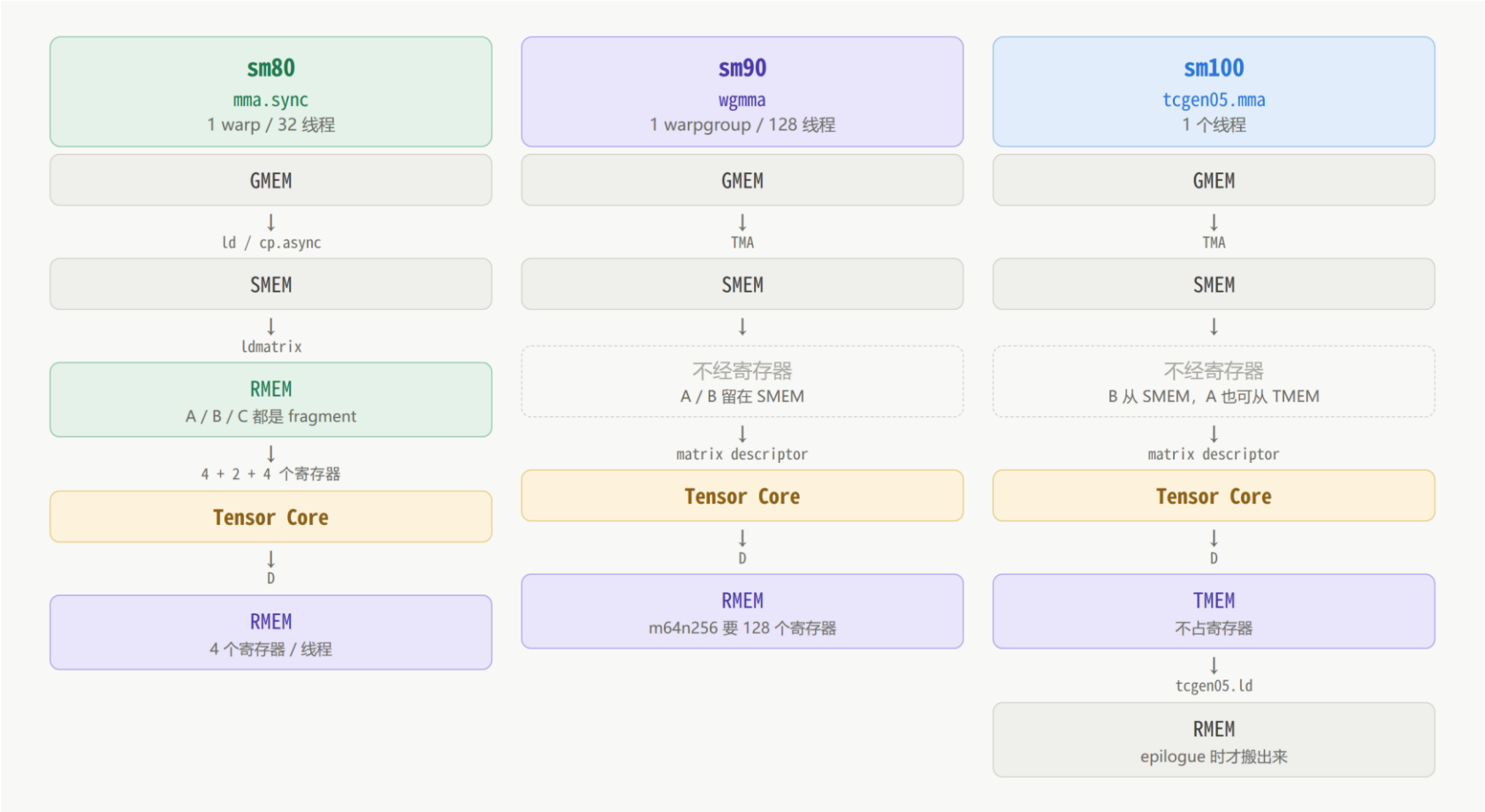
	mma.sync sm80 · Ampere	**wgmma** sm90 · Hopper	**tcgen05.mma** sm100 · Blackwell
发射者	1 warp / 32 线程	1 warpgroup / 128 线程	**1 个线程**
A 来自	RMEM fragment	SMEM descriptor	SMEM descriptor 或 TMEM
B 来自	RMEM fragment	SMEM descriptor	SMEM descriptor
C / D 在	RMEM	RMEM	**TMEM**
完成同步	隐式 warp sync	commit_group / wait_group	**mbarrier（与 TMA 统一）**
最大 tile	m16n8k16	m64n256k16	单 CTA m128n256k16
脚手架	fragment 图 + ldmatrix + swizzle	matrix descriptor + TMA	TMEM 分配 + CTA pair

发射者从 1 warp 收缩到 1 warpgroup，再收缩到 1 个线程

操作数从全寄存器，到 A/B 进 smem，再到累加器进 TMEM

完成同步从隐式 warp sync，到 commit / wait_group，再到与 TMA 统一的 mbarrier

三代数据通路



三代的 GMEM → SMEM 这一段都在，变的是操作数从哪里读、累加器放在哪里

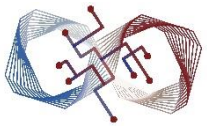
sm80：ldmatrix 把 A/B/C 装进寄存器，D 也回寄存器

sm90：A/B 留在 SMEM，通过 64-bit descriptor 直接送入 Tensor Core，D 仍在寄存器

P3・低精度与 block scaling

FP8 / FP4 与 scale factor

低精度的动机



每代算力翻倍，但仍不足以满足需求

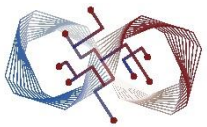
目标是减少显存与带宽占用、提高计算强度，因此探索更低的位数

Hopper 是**第一代支持 FP8 计算**的 GPU

FP8 有限的位数会影响 LLM 表现，直到 **DeepSeek 公开其方案**后，FP8 训练才被广泛采用

关键是 **scale factor**

FP8 的两种格式



E4M3

S	E 4 bit	M 3 bit

最大值	448
最小正规数	$2^{-6} = 0.015625$
最小次正规数	$2^{-9} = 0.001953$
相对精度	$2^{-4} \approx 6\%$

没有 inf；NaN 只占一个编码，因此比标准 IEEE 多出一档可用数值

E5M2

S	E 5 bit	M 2 bit

最大值	57344
最小正规数	$2^{-14} \approx 6.1e\text{-}5$
最小次正规数	$2^{-16} \approx 1.5e\text{-}5$
相对精度	$2^{-3} \approx 12\%$

动态范围大得多，但尾数只有 2 位，精度换范围

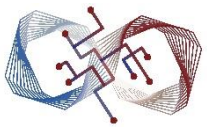
无论选哪个，**8 bit 的动态范围都盖不住训练中一个 tensor 的真实分布** —— 所以必须配 scale factor

E4M3：最大 448，没有 inf，NaN 只占一个编码

E5M2：动态范围最大到 57344，但尾数只有 2 位

8 bit 的动态范围**无法覆盖训练中 tensor 的真实分布**，因此必须配 scale factor

per-tensor scale

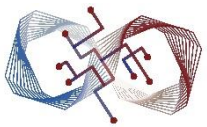


与 NVIDIA 的训练框架 Transformer Engine 一同提出

```
x_q = round(x / s), s = amax / 448
```

s 即 scale factor，反量化时乘回

想一想：per-tensor scale 的失效场景



大部分值在 −1 ~ +1，突然出现一个 3000。

用 s = amax / 448 量化之后，哪些值会出问题？

outlier 对 per-tensor scale 的影响

绝大多数值落在 **[-1, 1]** | 出现一个 outlier **3000** | s = amax / 448 = **6.696**

原值 x	x / s	最近的 E4M3	反量化	相对误差
3000	448.0	448.0	3000.0	**0.0%**
1.0	0.149333	0.156250	1.04632	**4.6%**
0.5	0.074667	0.078125	0.52316	**4.6%**
0.1	0.014933	0.015625	0.10463	**4.6%**
0.01	0.001493	0.001953	0.01308	**30.8%**
0.005	0.000747	0	0	**100%**
0.001	0.000149	0	0	**100%**

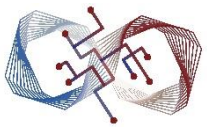
4.6% 是 E4M3 的固有下限

3 位尾数，相对步长 2^{-4}。这一档跟 outlier 无关，换多细的 scale 都省不掉

outlier 真正毁掉的是小值

s 被顶到 6.696，**x < 0.0065** 的元素落进次正规区以下，直接变成 0

block scaling 的数学约束



scale 随 (m, k) 任意变化

$$D[m,n] = \Sigma_k \; s_A[m,k] \cdot \hat{A}[m,k] \cdot s_B[k,n] \cdot \hat{B}[k,n]$$

scale 卡在求和号里面，**提不出来**。剩下的 $\hat{A} \cdot \hat{B}$ 不再是一个矩阵乘，Tensor Core 用不上

scale 沿 K 分段常数

$$D[m,n] = \Sigma_q \left(s_A[m,q] \cdot s_B[n/128,q] \right) \cdot \Sigma_{k \in q} \hat{A}[m,k] \cdot \hat{B}[k,n]$$

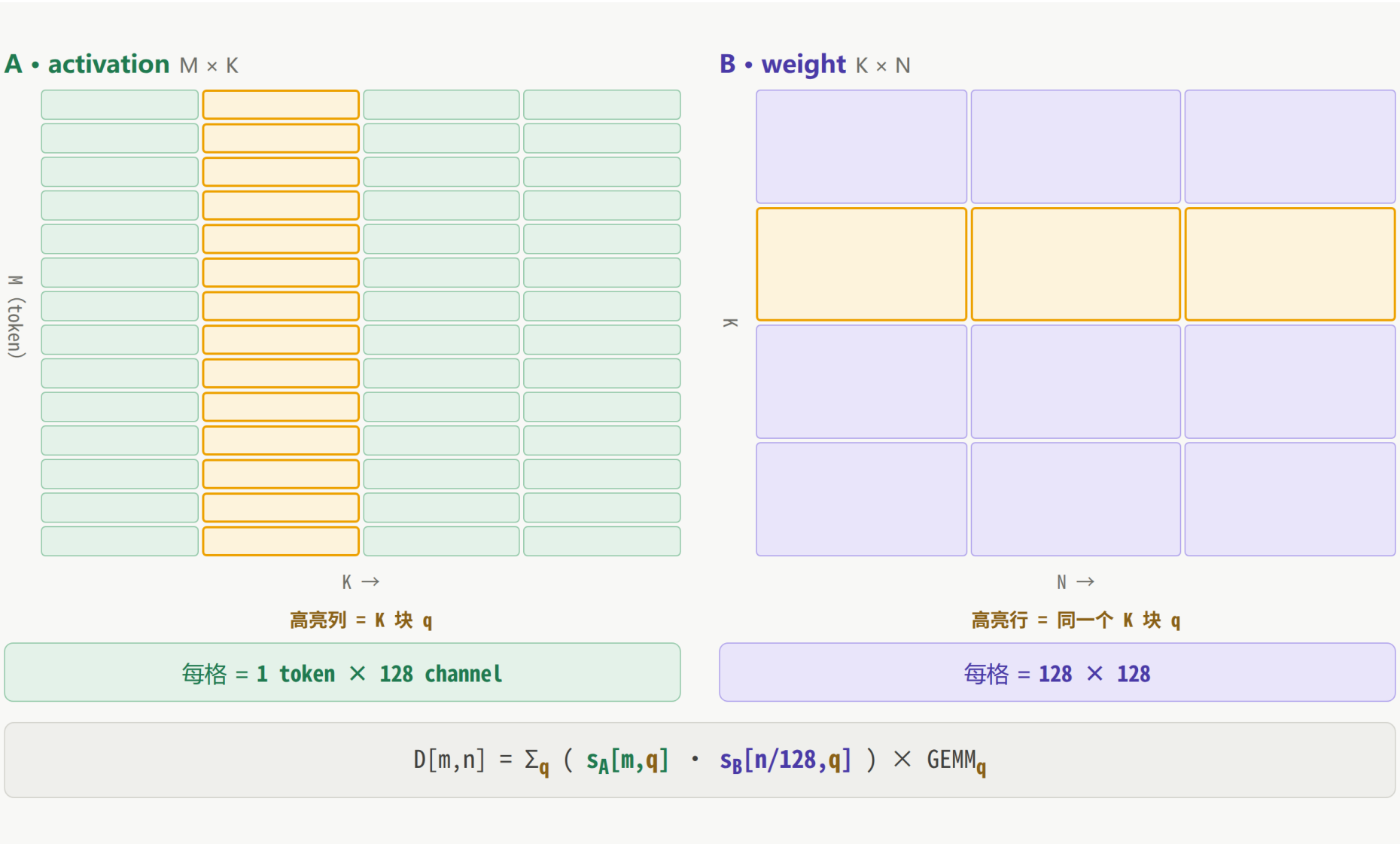
每一段内部 scale 是常数，可以提到求和号外面。 括号里的 $\Sigma_{k \in q}$ 就是一次**普通的 GEMM**，交给 Tensor Core； 外面那一层乘法留给 CUDA core

GEMM 的内积 $D[m,n] = \Sigma_k A[m,k] \cdot B[k,n]$

若 A 的 scale 随 $[m,k]$ 任意变化，scale 提不出求和号，**不再是矩阵乘**

scale **必须沿 K 分段常数**：per-row，或沿 K 再切分的 per-block

DeepSeek-V3 的量化 recipe

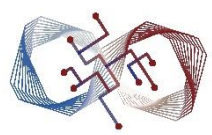


FP8 累加的精度损失

Hopper 的 fp8 TC 累加时，会把一组尾数的乘积**按最大指数右移对齐**，保留最高 **14 bit** 参与加法，超出直接截断

fp32 尾数应该有 24 bit → **丢 10 bit，K=4096 时约 2% 误差**

DeepGEMM 的 promotion 循环



手搓 sm100：发射与 epilogue

```
// (4) 一个线程发射 mma + commit
uint32_t elected;
asm volatile("{\n.reg .pred P;\nelect.sync _|P, 0xFFFFFFFF;\nselp.b32 %0, 1, 0, P;\n}"
             : "=r"(elected));
if (warp_id == 0 && elected) {
    asm volatile("tcgen05.fence::after_thread_sync;");   // 跨线程规则
    asm volatile(
        "{\n.reg .pred p;\nsetp.ne.b32 p, %4, 0;\n"
        "tcgen05.mma.cta_group::1.kind::f16 [%0], %1, %2, %3, p;\n}\n"
        :: "r"(taddr), "l"(a_desc), "l"(b_desc), "r"(idesc), "r"(0));
    asm volatile("tcgen05.commit.cta_group::1.mbarrier::arrive::one"
                 ".shared::cluster.b64 [%0];" :: "r"(mbar_u32) : "memory");
}

// (5) 全体等 mma 完成; acquire 语义保证累加器可见
mbarrier_wait(mbar_u32, 0);

// (6) epilogue: 每个 warp 只能读自己的 32 条 lane
asm volatile("tcgen05.fence::after_thread_sync;");
for (int n = 0; n < N; n += 8) { /* tcgen05.ld -> wait::ld -> 写回 gD */ }

// (7) 所有读结束后，一个 warp 释放 TMEM
__syncthreads();
if (warp_id == 0) asm volatile("tcgen05.dealloc.cta_group::1.sync.aligned.b32 %0, %1;"
                               :: "r"(taddr), "r"(32));
```

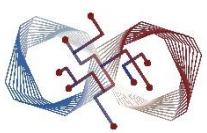
每隔一段做一次 **fp32 全精度累加**

两个累加器：`acc`（CUDA core 拥有，真 fp32）+ `part`（wgmma 的临时累加器）

每 `4 × k32 = 128` 做一次 promotion，**与量化块对齐**

这一版没有做 overlap，CUDA core 的占用见下一页

DeepGEMM 的 promotion 循环



手搓 sm100：发射与 epilogue

```
// (4) 一个线程发射 mma + commit
uint32_t elected;
asm volatile("{\n.reg .pred P;\nelect.sync _|P, 0xFFFFFFFF;\nselp.b32 %0, 1, 0, P;\n}"
             : "=r"(elected));
if (warp_id == 0 && elected) {
    asm volatile("tcgen05.fence::after_thread_sync;");   // 跨线程规则
    asm volatile(
        "{\n.reg .pred p;\nsetp.ne.b32 p, %4, 0;\n"
        "tcgen05.mma.cta_group::1.kind::f16 [%0], %1, %2, %3, p;\n}\n"
        :: "r"(taddr), "l"(a_desc), "l"(b_desc), "r"(idesc), "r"(0));
    asm volatile("tcgen05.commit.cta_group::1.mbarrier::arrive::one"
                 ".shared::cluster.b64 [%0];" :: "r"(mbar_u32) : "memory");
}

// (5) 全体等 mma 完成; acquire 语义保证累加器可见
mbarrier_wait(mbar_u32, 0);

// (6) epilogue: 每个 warp 只能读自己的 32 条 lane
asm volatile("tcgen05.fence::after_thread_sync;");
for (int n = 0; n < N; n += 8) { /* tcgen05.ld -> wait::ld -> 写回 gD */ }

// (7) 所有读结束后，一个 warp 释放 TMEM
__syncthreads();
if (warp_id == 0) asm volatile("tcgen05.dealloc.cta_group::1.sync.aligned.b32 %0, %1;"
                               :: "r"(taddr), "r"(32));
```

每隔一段做一次 **fp32 全精度累加**

两个累加器：`acc`（CUDA core 拥有，真 fp32）+ `part`（wgmma 的临时累加器）

每 `4 × k32 = 128` 做一次 promotion，**与量化块对齐**

这一版没有做 overlap，CUDA core 的占用见下一页

DeepGEMM 的 promotion 循环

手搓 sm100：发射与 epilogue

```
// (4) 一个线程发射 mma + commit
uint32_t elected;
asm volatile("{\n.reg .pred P;\nelect.sync _|P, 0xFFFFFFFF;\nselp.b32 %0, 1, 0, P;\n}"
             : "=r"(elected));
if (warp_id == 0 && elected) {
    asm volatile("tcgen05.fence::after_thread_sync;");   // 跨线程规则
    asm volatile(
        "{\n.reg .pred p;\nsetp.ne.b32 p, %4, 0;\n"
        "tcgen05.mma.cta_group::1.kind::f16 [%0], %1, %2, %3, p;\n}\n"
        :: "r"(taddr), "l"(a_desc), "l"(b_desc), "r"(idesc), "r"(0));
    asm volatile("tcgen05.commit.cta_group::1.mbarrier::arrive::one"
                 ".shared::cluster.b64 [%0];" :: "r"(mbar_u32) : "memory");
}

// (5) 全体等 mma 完成; acquire 语义保证累加器可见
mbarrier_wait(mbar_u32, 0);

// (6) epilogue: 每个 warp 只能读自己的 32 条 lane
asm volatile("tcgen05.fence::after_thread_sync;");
for (int n = 0; n < N; n += 8) { /* tcgen05.ld -> wait::ld -> 写回 gD */ }

// (7) 所有读结束后，一个 warp 释放 TMEM
__syncthreads();
if (warp_id == 0) asm volatile("tcgen05.dealloc.cta_group::1.sync.aligned.b32 %0, %1;"
                               :: "r"(taddr), "r"(32));
```

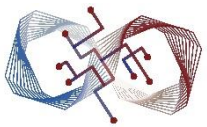
每隔一段做一次 **fp32 全精度累加**

两个累加器：`acc`（CUDA core 拥有，真 fp32）+ `part`（wgmma 的临时累加器）

每 `4 × k32 = 128` 做一次 promotion，**与量化块对齐**

这一版没有做 overlap，CUDA core 的占用见下一页

Blackwell 上 promotion 的开销



$$\text{promotion FLOP} / \text{mma FLOP} = 2 \cdot M \cdot N / (2 \cdot M \cdot N \cdot \text{BLOCK\_K}) = \mathbf{1 / BLOCK\_K}$$ （与 tile 大小无关）

H100

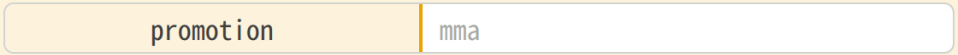
fp8 Tensor Core	1979 TFLOPS
fp32 CUDA core	67 TFLOPS
速率比	29.5×
BLOCK_K = 128 时 FLOP 占比	1 / 128
折算成时间	1/128 × 29.5

23% 的 mma 时间用在 promotion

B200

fp8 Tensor Core	4500 TFLOPS
fp32 CUDA core	80 TFLOPS
速率比	56.2×
BLOCK_K = 128 时 FLOP 占比	1 / 128
折算成时间	1/128 × 56.2

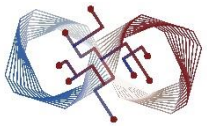
44% 的 mma 时间用在 promotion



TC 性能又翻倍（fp8 dense **1979 → 4500 TFLOPS**）

CUDA core 的 fp32 从 67 涨到 80 TFLOPS，**promotion 的时间占比从 23% 升到 44%**

硬件 block scaling



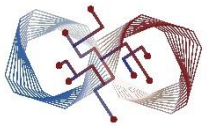
DeepGEMM 的 promotion 循环（对应 sm90_fp8_gemm_1d2d.cuh，此处为教学简化版）

```
float acc[ACC_N];      // 主累加器，CUDA core 拥有，真 fp32
float part[ACC_N];     // wgmma 的临时累加器

for (int q = 0; q < K / 128; ++q) {
    warpgroup_fence();                    // CUDA core 改过 part，对应第 5 条规则
    #pragma unroll
    for (int i = 0; i < 4; ++i)           // 4 × k32 = 128，对齐量化块
        wgmma_m64nNk32(part, desc_a(q,i), desc_b(q,i), /*scale_d=*/ i > 0);
    warpgroup_commit();
    warpgroup_wait<0>();

    float s = s_a_frag[q] * s_b[n_blk][q];        // 1×128 与 128×128 的 scale
    #pragma unroll
    for (int i = 0; i < ACC_N; ++i)
        acc[i] += s * part[i];            // FFMA，fp32 全精度累加
}
```

K 维每 32（或 16）个元素配一个 scale factor

在硬件层面反量化，不需要 CUDA core 参与

硬件 block scaling



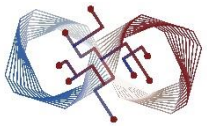
DeepGEMM 的 promotion 循环（对应 sm90_fp8_gemm_1d2d.cuh，此处为教学简化版）

```
float acc[ACC_N];      // 主累加器，CUDA core 拥有，真 fp32
float part[ACC_N];     // wgmma 的临时累加器

for (int q = 0; q < K / 128; ++q) {
    warpgroup_fence();                  // CUDA core 改过 part，对应第 5 条规则
    #pragma unroll
    for (int i = 0; i < 4; ++i)         // 4 × k32 = 128，对齐量化块
        wgmma_m64nNk32(part, desc_a(q,i), desc_b(q,i), /*scale_d=*/ i > 0);
    warpgroup_commit();
    warpgroup_wait<0>();

    float s = s_a_frag[q] * s_b[n_blk][q];       // 1×128 与 128×128 的 scale
    #pragma unroll
    for (int i = 0; i < ACC_N; ++i)
        acc[i] += s * part[i];          // FFMA，fp32 全精度累加
}
```

K 维每 32（或 16）个元素配一个 scale factor

在硬件层面反量化，不需要 CUDA core 参与

硬件 block scaling



DeepGEMM 的 promotion 循环（对应 sm90_fp8_gemm_1d2d.cuh，此处为教学简化版）

```
float acc[ACC_N];      // 主累加器，CUDA core 拥有，真 fp32
float part[ACC_N];     // wgmma 的临时累加器

for (int q = 0; q < K / 128; ++q) {
    warpgroup_fence();                    // CUDA core 改过 part，对应第 5 条规则
    #pragma unroll
    for (int i = 0; i < 4; ++i)           // 4 × k32 = 128，对齐量化块
        wgmma_m64nNk32(part, desc_a(q,i), desc_b(q,i), /*scale_d=*/ i > 0);
    warpgroup_commit();
    warpgroup_wait<0>();

    float s = s_a_frag[q] * s_b[n_blk][q];        // 1×128 与 128×128 的 scale
    #pragma unroll
    for (int i = 0; i < ACC_N; ++i)
        acc[i] += s * part[i];            // FFMA，fp32 全精度累加
}
```

K 维每 32（或 16）个元素配一个 scale factor

在硬件层面反量化，不需要 CUDA core 参与

fp6 与 fp4 的格式

格式	结构 S+E+M	非负格点 / 最大值
E2M1 （fp4）	1 + 2 + 1	只有 {0, 0.5, 1, 1.5, 2, 3, 4, 6} 共 8 个
E3M2 （fp6）	1 + 3 + 2	最大 28
E2M3 （fp6）	1 + 2 + 3	最大 7.5
UE8M0	0 + 8 + 0	2^{-127} ~ 2^{127}，外加 NaN。纯指数，只用来存 scale
UE4M3	0 + 4 + 3	最大 448。无符号，只用来存 scale

这两行不是数据格式，是 scale factor 的格式

E2M1 只有 8 个非负格点：0, 0.5, 1, 1.5, 2, 3, 4, 6

E3M2 最大 28，E2M3 最大 7.5，同为 6 bit，范围与精度互换

UE8M0 与 UE4M3 不是数据格式，是 scale factor 的格式；无符号，UE8M0 是纯指数

MXFP 与 NVFP 打包格式

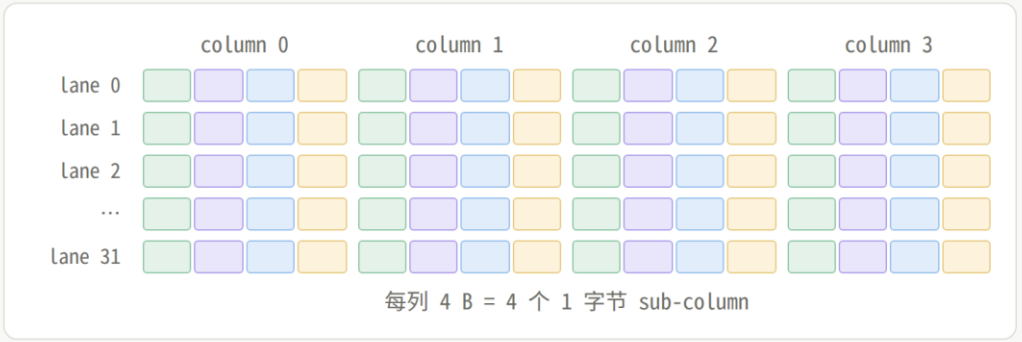
打包格式	元素类型	块长（沿 K）	scale 类型	出处
MXFP8	E4M3 / E5M2	32	UE8M0	OCP 标准
MXFP6	E3M2 / E2M3	32	UE8M0	OCP 标准
MXFP4	E2M1	32	UE8M0	OCP 标准
NVFP4	E2M1	16	UE4M3	NVIDIA 私有

MX = **Microscaling**，OCP（Open Compute Project）标准， 2023 年由 AMD、Arm、Intel、Meta、微软、NVIDIA、高通共同发布。 NVFP4 块更短、scale 精度更高，另有一层 per-tensor 的 fp32 二级 scale

MX = **Microscaling**，OCP（Open Compute Project）标准

2023 年 AMD、Arm、Intel、Meta、微软、NVIDIA、高通一起发布

scale factor 在 TMEM 里的 layout



一个 SF tile

32 lane × 4 column。每列 4 字节拆成 4 个 sub-column，UE8M0 恰好 1 字节 → **32 × 16 = 512 个 scale**

SFA_ID / SFB_ID

idesc 里的 2 bit，选本条 mma 用 4 个 sub-column 中的哪一个 → 一个 SF tile 能放**沿 K 连续 4 条 mma** 的 scale

要几列

M = 128 的 A 需要 4 列放 SFA；SFB 的列数随 N 从 1 到 8

必须复制到 4 个分区

PTX 要求 scale factor 在全部 4 个 32-lane 分区各存一份， `tcgen05.cp` 有专门的 `.warpx4` multicast 变体

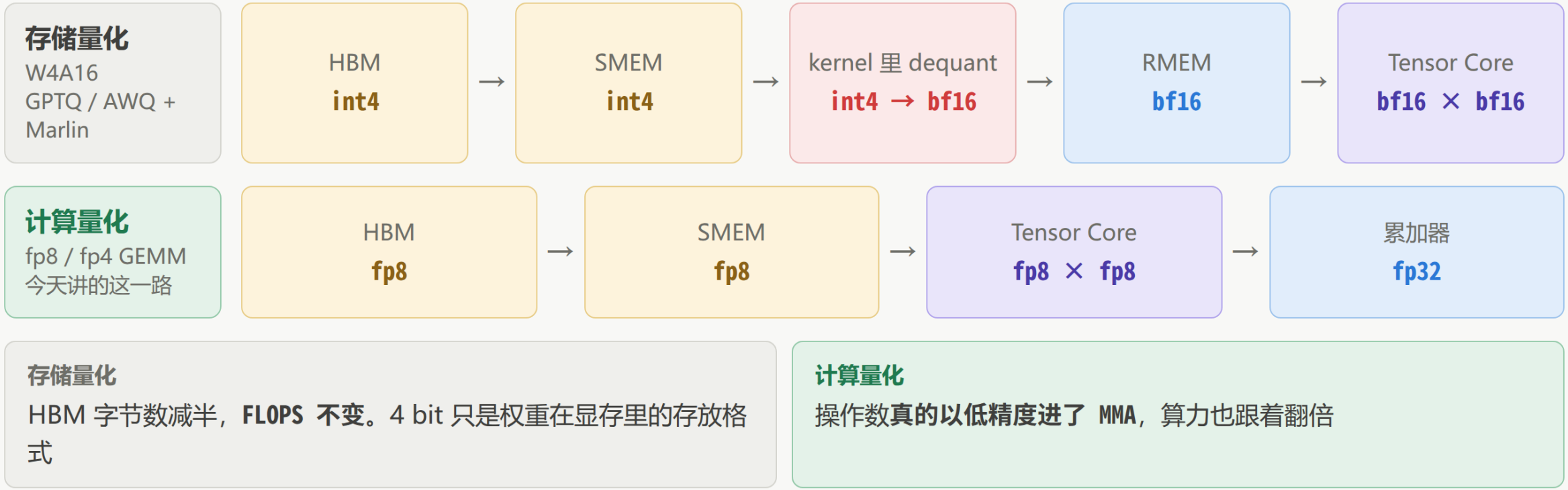
sub-col 0 1 2 3

SFA/SFB 相当于 MMA 的**第三、第四个操作数**，也从 TMEM 读取

`tcgen05.cp` 的主要用途就是把 SF 矩阵搬进 TMEM

`tcgen05.cp` 与 `tcgen05.mma` 在**同一条 pipeline** 上，同一发射者先 cp 后 mma，**硬件保证按序执行，不需要显式同步**

存储量化与计算量化

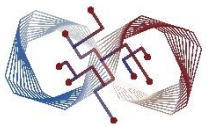


社区里最流行的一族是 **W4A16**：GPTQ / AWQ 这些算法，配 Marlin 这类 kernel

· weight 用 4 bit 存在 HBM 里

· 但**进 Tensor Core 之前，kernel 先把 weight dequant 回 bf16**，MMA 做的是 bf16 × bf16

量化的三个维度

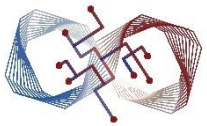


量化什么：weight？activation？KV cache？gradient？optimizer state？

什么粒度：per-tensor / per-channel(row) / per-block / per-group；scale 用什么格式存

什么时机：训练完再量化（PTQ）？训练时模拟量化（QAT）？还是像 DeepSeek-V3 那样**训练本身就跑在低精度上**？

延伸阅读



KV cache 量化

PTQ vs QAT vs 原生低精度训练

算法侧的 outlier 处理有两条路线：

· 细粒度 scale 直接承受 outlier，即今天讲的这一路

· 另一路在量化前抹平 outlier：**SmoothQuant** 把 activation 的量化难度按比例迁移给 weight；**QuaRot / SpinQuant** 用旋转矩阵把 outlier 摊匀到所有 channel

总结

三代 Tensor Core 面对的是同一个问题：算力翻倍后如何把数据送到它面前

	上一代留下的问题	这一代的答案
sm80 Ampere	数据全在寄存器，每个线程自己算地址取数	`fragment 图` · `ldmatrix` · `swizzle`
sm90 Hopper	寄存器带宽不够，且 Tensor Core 计算时 CUDA core 空闲	`smem descriptor` · `异步 wgmma` · `TMA`
sm100 Blackwell	累加器占用大量寄存器，而发射指令根本不需要线程	`TMEM` · `单线程发射` · `mbarrier` · `2-CTA`
低精度 FP8 / FP4	带宽已接近极限，转向减少每个元素的位数	`block scaling` · `promotion` · `硬件反量化`

下一场：**`tiling` · `pipeline` · `warp specialization`**，用于补上这 50 倍的差距