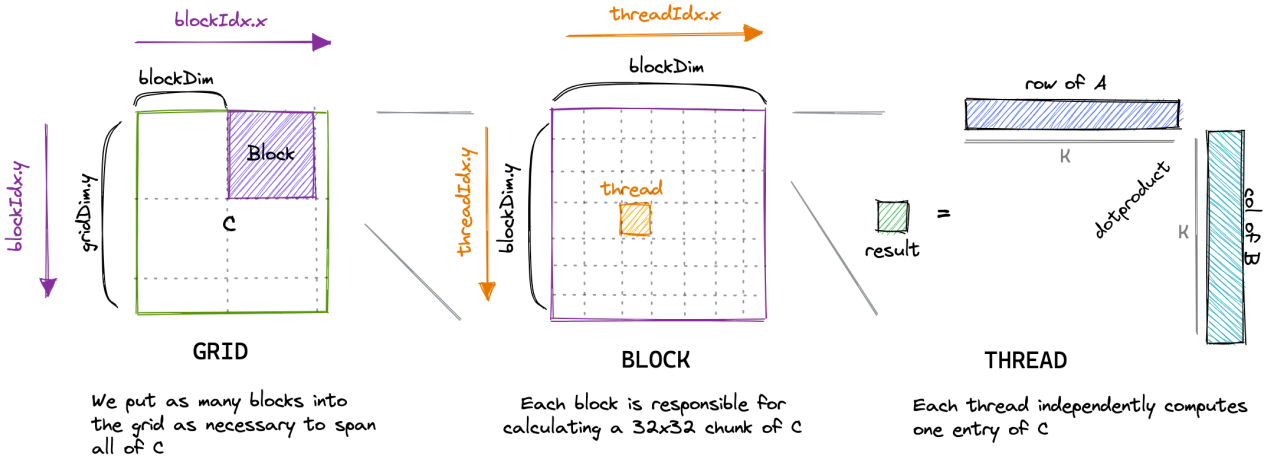
1 Kernel 1: Naive Implementation

三级层次结构: `grid` -- `block` -- `thread` （每个 `block` 至多 1024 `threads`）

位于同一 `block` 中的 `thread` 可以访问相同的共享内存区域(SMEM)。



`Block` 大小: `blockDim.x * blockDim.y` ;

> 注意上方 `blockIdx.x` 和 `threadIdx.x` 都是水平方向，相应的 `*.y` 是垂直方向，但是在下面 Kernel 计算时又相当于 `x` 映射到垂直方向，`y` 映射到水平方向。（类似于做了转置）

第一个 Kernel，每个 `thread` 仅计算一个结果。

```
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32));
dim3 blockDim(32, 32);

__global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
                            const float *B, float beta, float *C) {
  // compute position in C that this thread is responsible for
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  // `if` condition is necessary for when M or N aren't multiples of 32.
  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    // C = α*(A@B)+β*C
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
```

```
18     }
19 }
```

1.1 Lower Bounding the Fastest Possible Runtime

下面粗略进行理论计算：

> FMA (fused multiply-add) 计为两条 FLOPs

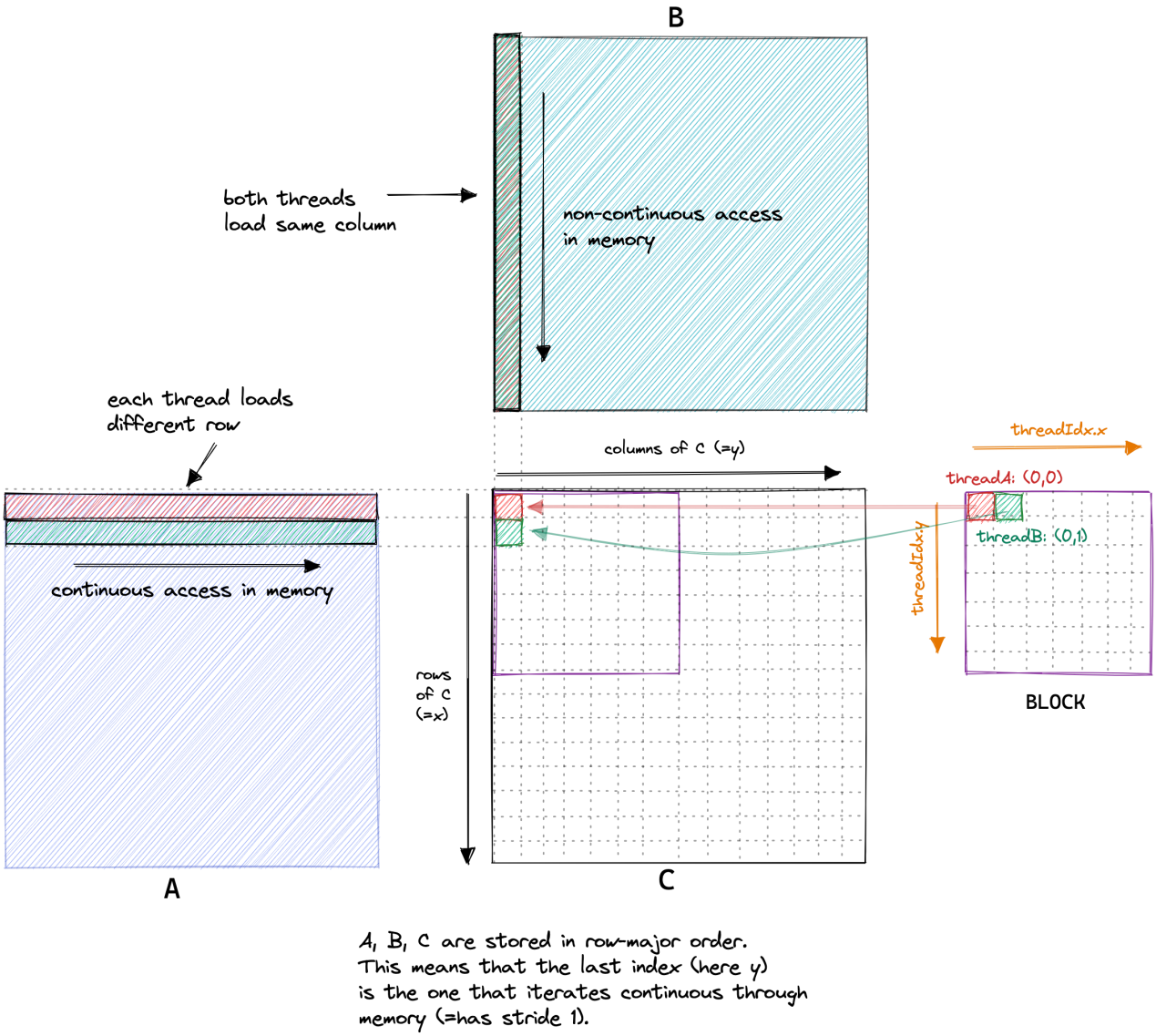
1. Total FLOPS：$2 * 4092^3 + 4092^2 = 137$ GFLOPS （`tmp` FMA 以及 `C` 的计算，4092^2 前面系数不重要）
2. Total data to read（至少）：$3 * 4092^3 * 4B = 201MB$ （A、B、C 三个矩阵）
3. Total data to store：$4092^2 * 4B = 67MB$ （每个 `thread` 一个 `tmp`）

calculation: $\frac{137G}{30T/s} = 4.5ms$; memory: $\frac{268MB}{768GB/s} = 0.34ms \Rightarrow$ 最后优化的内核应该是计算受限的（只要数据传输量不超过理论最小值的 10 倍）！

1.2 Memory Access Pattern of the Naive Kernel

假设最坏情况，cache 全部不命中，每个 `thread` 从 `global memory` 中读取 $2 * 4092 + 1$（一行 A，一列 B，一个 C）

共有 4092^2 个 `threads`，则有 $4B * [2 * 4092 + 1] * 4092^2 = 548GB$ 数据传输！



> 这张图印证了前文中 `thread` 和 `result` 的映射关系

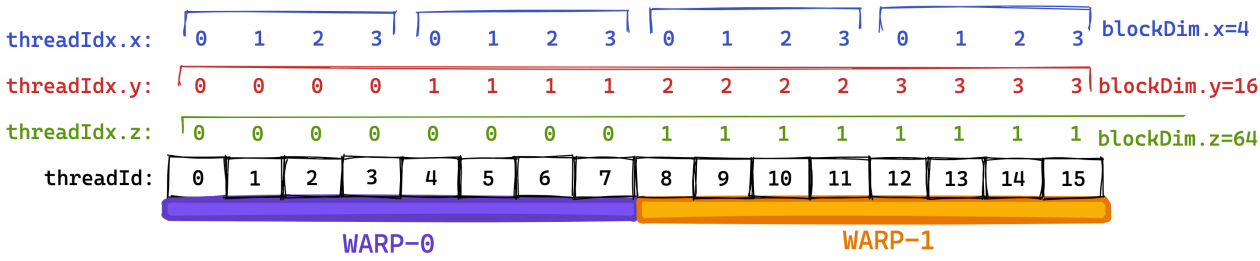
下一步：优化内存访问模式，全局内存合并访问。

2 Kernel 2: Global Memory Coalescing

32 `threads`（具有连续的 `threadId`）组成一个 `warp`。

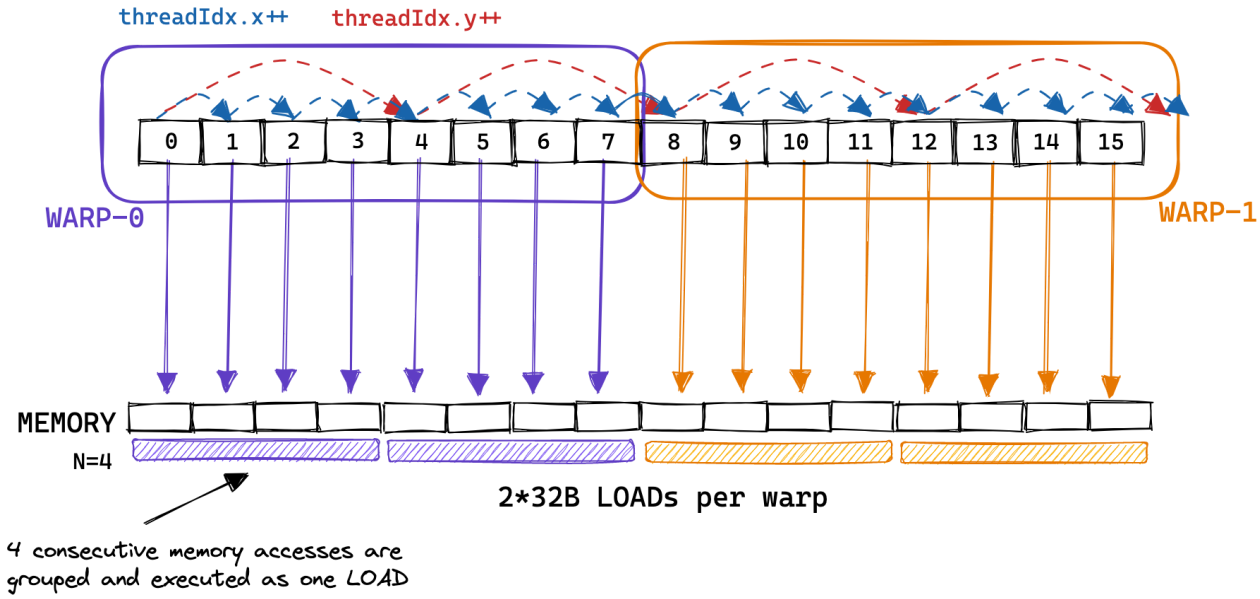
`threadId = threadIdx.x + blockDim.x * (threadIdx.y + blockDim.y * threadIdx.z)`

> 假如看作三维的话，`blockDim.y` 相当于一个面，`blockDim.x` 相当于一条线（`blockDim.z` 应该是整个立方体的体积）




```
threadId = threadIdx.x + blockDim.x*threadIdx.y + blockDim.x* blockDim.y *threadIdx.z
```

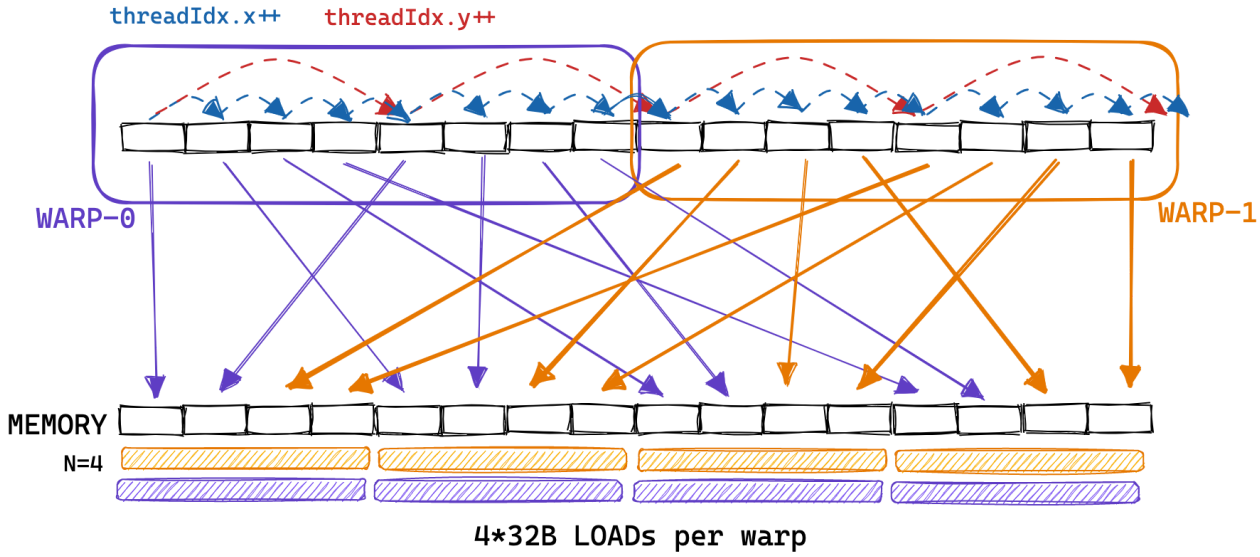
同一 `warp` 的部分 `threads` 对连续内存的访问可以合并为一个 `Load` ！



实际上，GPU 支持 32B、64B 和 128B 内存访问。因此，如果每个线程从全局内存加载一个 32 位浮点数，则 `warp` 调度程序(可能是 MIO(memory input/output))可以将这个 32 * 4B = 128B 加载合并到一个事务中。

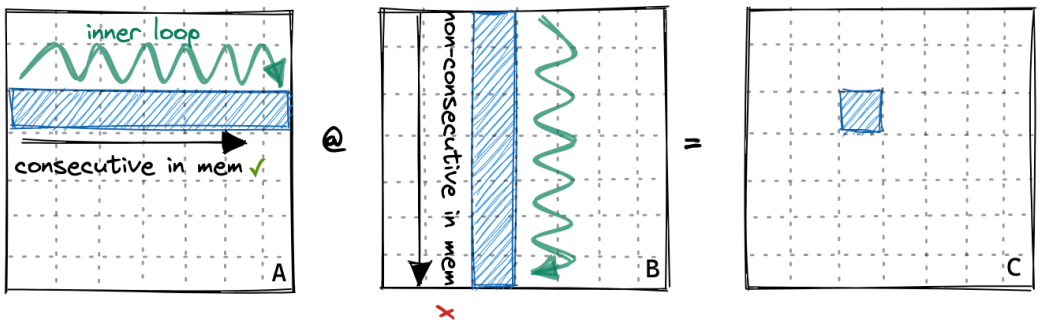
> 大致理解是，原来零散的 32 个 `Load`（或者是尽可能多的 32B `Load`） 现在可以用一个 128B `Load` 代替。

第一个 Kernel 中，同一 `warp` 的线程(具有连续的 `threadIdx.x` 的线程)非连续地从内存中加载 A 的行。

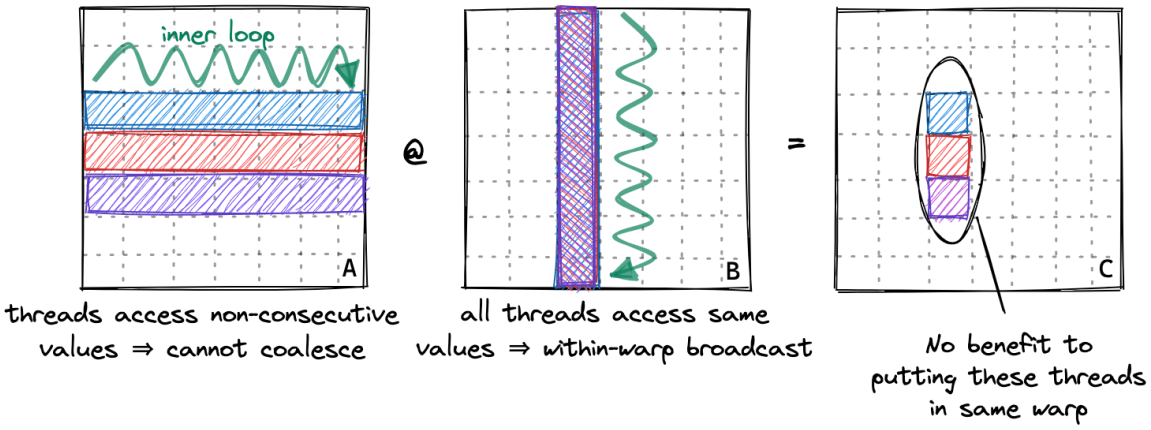


因此，重新分配线程到矩阵 C 元素的映射。

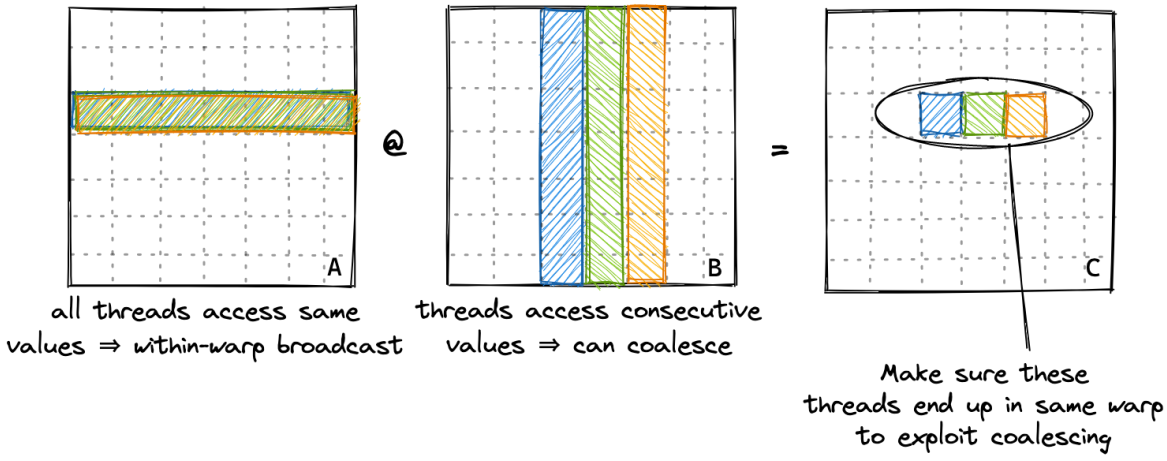
Matrix memory layout:



Naive kernel:



Coalescing kernel:



> 第一张图：矩阵的横行在内存中是连续的；
>
> 第二张图：原来的 Kernel 在每个 `thread` 访问 A 时，分别访问**每行**的第一个元素（而这在内存上是不连续的），当然，对于 B，访问的都是同一列（虽然在内存上也不连续），可以广播；

> 第三张图：改变映射关系之后，连续的 `threadIdx.x` 映射后在水平方向上是连续的，这样访问 A 时都是同一行（在内存中也连续），可以广播，访问 B 时，每个 `thread` 虽然都是访问一列，但是所有 `thread` 同时访问的可看作一行（在内存中连续），成功利用了合并访问。

具体实现：

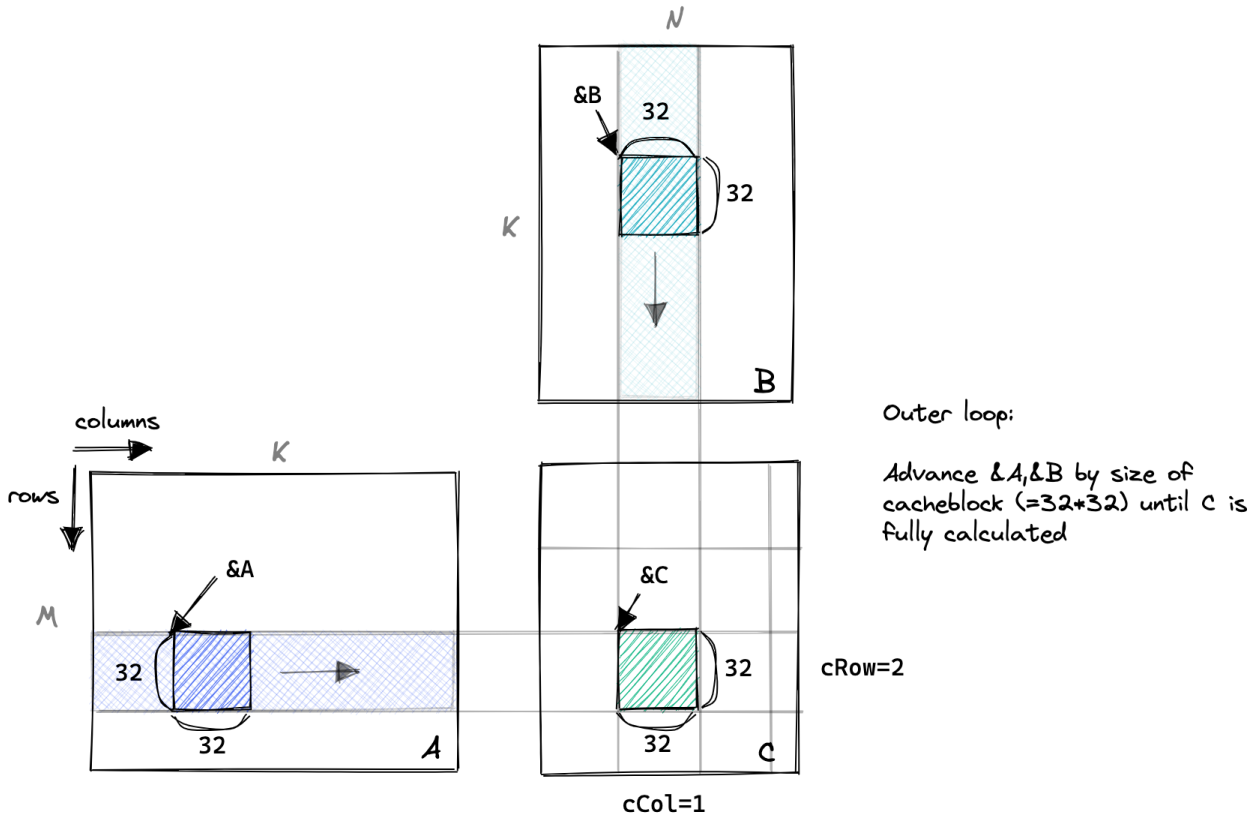
```
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32));
dim3 blockDim(32 * 32);

const int x = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOCKSIZE);
const int y = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOCKSIZE);

if (x < M && y < N) {
  float tmp = 0.0;
  for (int i = 0; i < K; ++i) {
    tmp += A[x * K + i] * B[i * N + y];
  }
  C[x * N + y] = alpha * tmp + beta * C[x * N + y];
}
```

> 注：这里调用的时候，`blockDim` 是一维的。

3 Kernel 3: Shared Memory Cache-Blocking

接下来使用共享内存来进行加速。每个 SM 有一个共享内存，线程可以通过共享内存与同一 `block` 中的其他线程进行通信。

思路是分块，将每一块从全局内存加载到共享内存中。



> 这里 `rows` 和 `cols` 在方向上很好理解，但是在下面代码中 `blockIdx.x` 作为 `cRow` 又似乎与之前的图有所矛盾！根据作者的脚注 19，他希望 `threadId` x 这一维在空间中是连续的（而在矩阵 C 中，y 才是连续的）。
>
> 但是事实上，如果不考虑这么多 corner cases，测试的时候 `M = N = K`，则可以不管这些。
>
> 又查阅了其他资料，`BlockIdx` 的方向与作者先前的描述应该是一致的，这里再回过头去看似乎 Kernel 2 中的 `x` 和 `y` 的计算也有些奇怪，经过一番理解，认为（Kernel 2 和 Kernel 3）对于 `Block` 的映射还是要进行转置，但是 `Block` 内的 `thread` 不用，总之理解起来比较别扭。

实现如下：

```
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32));
dim3 blockDim(32 * 32); // blockDim 仍为一维

// the output block that we want to compute in this threadblock
const uint cRow = blockIdx.x;   // cRow 表明计算的块在哪一行
const uint cCol = blockIdx.y;   // cCol 表明计算的块在哪一列

// allocate buffer for current block in fast shared mem
// shared mem is shared between all threads in a block
```

```
__shared__ float As[BLOCKSIZE * BLOCKSIZE];     // 共享内存就一个块的大小
__shared__ float Bs[BLOCKSIZE * BLOCKSIZE];

// the inner row & col that we're accessing in this thread
const uint threadCol = threadIdx.x % BLOCKSIZE; // 在 Block 里对应的位置
const uint threadRow = threadIdx.x / BLOCKSIZE;

// advance pointers to the starting positions
A += cRow * BLOCKSIZE * K;                    // row=cRow, col=0    A 在第 1 列上
B += cCol * BLOCKSIZE;                        // row=0, col=cCol    B 在第 1 行上
C += cRow * BLOCKSIZE * N + cCol * BLOCKSIZE; // row=cRow, col=cCol  C 在行列交叉的位置上

float tmp = 0.0;
// the outer loop advances A along the columns and B along
// the rows until we have fully calculated the result in C.
for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCKSIZE) {    // 按块进行加载
  // Have each thread load one of the elements in A & B from
  // global memory into shared memory.
  // Make the threadCol (=threadIdx.x) the consecutive index
  // to allow global memory access coalescing
  As[threadRow * BLOCKSIZE + threadCol] = A[threadRow * K + threadCol];
  Bs[threadRow * BLOCKSIZE + threadCol] = B[threadRow * N + threadCol];
  // 每个 thread 实际上只需要加载对应块的**一个**元素

  // block threads in this block until cache is fully populated
  __syncthreads();      // 同步，保证 SMEM 加载完毕

  // advance pointers onto next chunk
  A += BLOCKSIZE;       // 相当于 A 往右走
  B += BLOCKSIZE * N;   // 相当于 B 往下走

  // execute the dotproduct on the currently cached block
  for (int dotIdx = 0; dotIdx < BLOCKSIZE; ++dotIdx) {
    tmp += As[threadRow * BLOCKSIZE + dotIdx] *
           Bs[dotIdx * BLOCKSIZE + threadCol];
    // 访存模式和 Kernel 2 一样
  }
  // need to sync again at the end, to avoid faster threads
  // fetching the next block into the cache before slower threads are done
  __syncthreads();  // 同步，计算完了才加载下一块。
```

```
50  }
51  C[threadRow * N + threadCol] =
52      alpha * tmp + beta * C[threadRow * N + threadCol];
53  // 只需要算一个结果
```

在 `BLOCKSIZE = 32` 时，用了 `2 * 32 * 32 * 4B = 8KB` 共享内存。虽然可以增加块大小，但这样会使得一个 SM 加载更少的块，按 CUDA 的说法，增加每个块的共享内存利用率会降低占有率。占有率定义为每个 SM 的活跃 `warp` 数与每个 SM 的活跃 `warp` 最大可能数之比。

高占用率很有用的，因为它允许隐藏**高延迟**的操作（通过有更大的可发射指令池）。在 SM 上加载更多活跃块有三个主要限制: 寄存器、`warp` 和 SMEM 容量。

3.1 Occupancy Calculation for Kernel 3

任务按块粒度加载到 SM 上。

- **Shared memory**: 8192B/Block(8KB) + 1024B/Block for CUDA runtime usage = 9216B/Block. (102400B per SM) / (9216B per Block) = 11.11 ⇒ 11 Blocks upper limit.
- **Threads**: 1024 Threads per Block, max 1536 threads per SM ⇒ Upper limit 1 block.
- **Registers**: 37 regs per thread * 32 threads per warp = 1184 regs per warp. 寄存器分配按照 256 一组，上界应该是 1280（1024 threads / 32) = 32 warps per block, hence 1280 regs per warp * 32 warps per block = 40960 regs per block. Max 65536 regs per SM ⇒ upper limit 1 block.

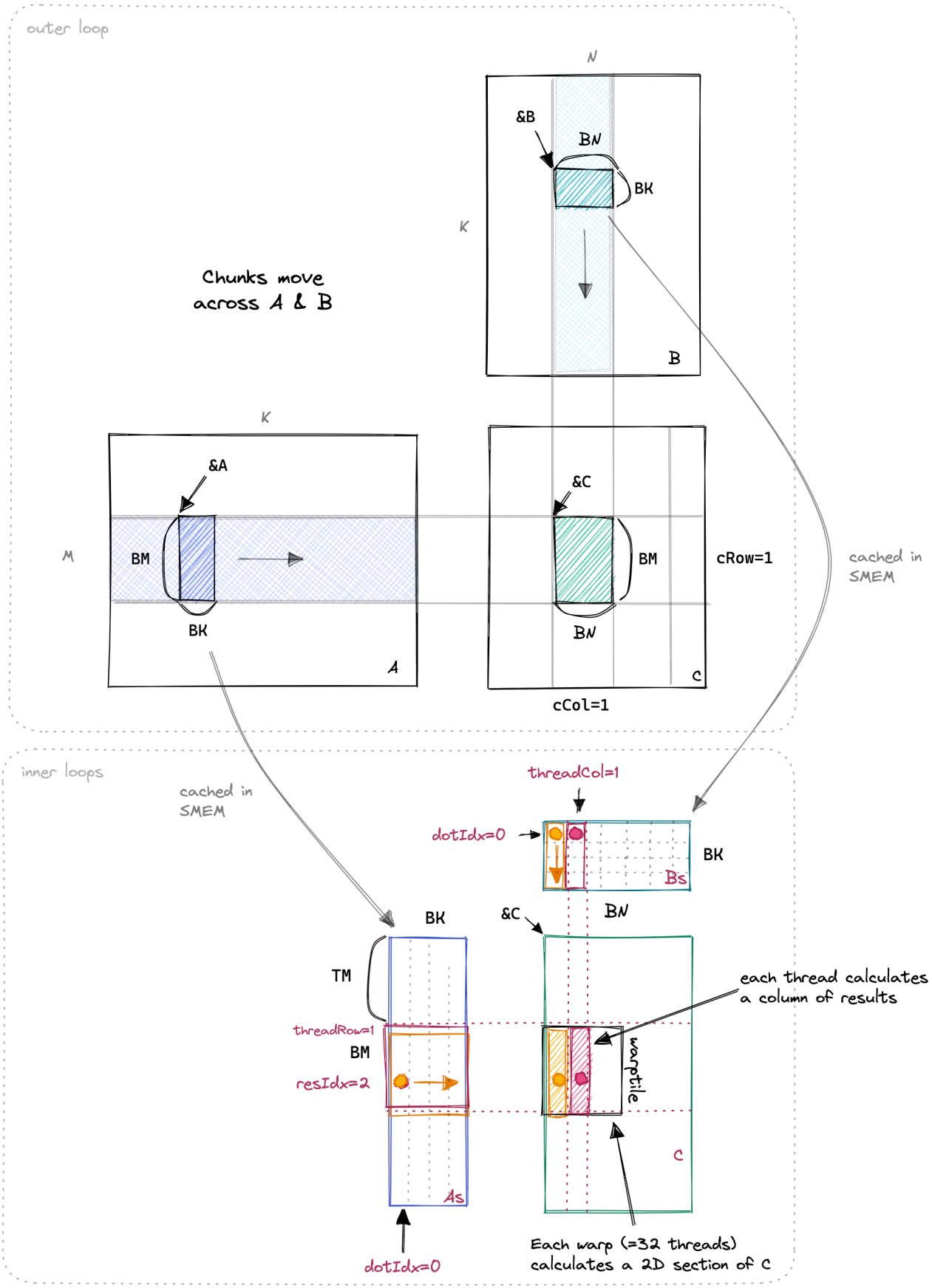
每个 SM 最多加载 1 个 block。通过 profiler 发现大部分指令时 `Load`（`Load` 比 `FMA` 指令具有更高的延迟）进一步分析 `Warp` 状态：

> Stall MIO Throttle: Warp 在等待 MIO (内存输入/输出)指令队列时 stall。在极端利用 MIO (包括特殊的数学指令、动态分支以及**共享内存指令**)的情况下，可能产生这种 stall。

所以 stall 的主要原因是等待 SMEM 访存。那么要如何减少发射 SMEM 指令呢？一种方法是让**每个线程计算多个输出元素**，这使得在**寄存器**中执行更多的工作，并减少对 SMEM 的依赖。

4 Kernel 4: 1D Blocktiling for Calculating Multiple Results per Thread

这个 Kernel 核心是让一个线程计算多个结果（具体为计算一列 TM = 8 个）。现在使用 $BM * BK + BN * BK = 64 * 8 + 64 * 8 = 1024$ 个浮点数的 SMEM 缓存大小，每个块使用的共享内存总大小为 4KB。




```
const uint BM = 64;
const uint BN = 64;
const uint BK = 8;
const uint TM = 8;
dim3 gridDim(CEIL_DIV(N, BN), CEIL_DIV(M, BM));
```

```
dim3 blockDim((BM * BN) / TM);

// If we flip x and y here we get ~30% less performance for large matrices.
// The current, 30% faster configuration ensures that blocks with sequential
// blockIDs access columns of B sequentially, while sharing the same row of A.
// The slower configuration would share columns of A, but access into B would
// be non-sequential. So the faster configuration has better spatial locality
// and hence a greater L2 hit rate.
const uint cRow = blockIdx.y;
const uint cCol = blockIdx.x;
// 这里的 cRow 和 cCol 和前面对上了，并且按照作者的说法具有更好的空间局部性，这一部分的分析其实和 Kernel 2 是一致的。

// each warp will calculate 32*TM elements, with 32 being the columnar dim.
const int threadCol = threadIdx.x % BN;
const int threadRow = threadIdx.x / BN;
// 对于矩阵 C（块大小是 BM * BN），将竖着的 TM 个看作一块，宽方向上的 BN 并没有合并
// 而实际上一个 block 内 thread 只有 BM * BN / TM 个

// allocate space for the current blocktile in SMEM
__shared__ float As[BM * BK];
__shared__ float Bs[BK * BN];

// Move blocktile to beginning of A's row and B's column
A += cRow * BM * K;
B += cCol * BN;
C += cRow * BM * N + cCol * BN;

// todo: adjust this to each thread to load multiple entries and
// better exploit the cache sizes
assert(BM * BK == blockDim.x);  // blockDim.x = BM * BN / TM = BM * BK
assert(BN * BK == blockDim.x);  // blockDim.x = BM * BN / TM = BN * BK
// 这相当于 A 和 B 中的块大小和 thread 的个数是一一对应的
const uint innerColA = threadIdx.x % BK; // warp-level GMEM coalescing
const uint innerRowA = threadIdx.x / BK;
const uint innerColB = threadIdx.x % BN; // warp-level GMEM coalescing
const uint innerRowB = threadIdx.x / BN;
// 对于矩阵 A（块大小是 BM * BK）
// 对于矩阵 B（块大小是 BK * BN）
```

```
44
45  // allocate thread-local cache for results in registerfile
46  float threadResults[TM] = {0.0};
47
48  // outer loop over block tiles
49  for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
50      // populate the SMEM caches
51      As[innerRowA * BK + innerColA] = A[innerRowA * K + innerColA];
52      Bs[innerRowB * BN + innerColB] = B[innerRowB * N + innerColB];
53      __syncthreads(); // 同步，每次只需要加载 A、B 块中的一个元素
54
55      // advance blocktile
56      A += BK;
57      B += BK * N;
58
59      // calculate per-thread results
60      for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
61          // we make the dotproduct loop the outside loop, which
    facilitates
62          // reuse of the Bs entry, which we can cache in a tmp var.
63          float tmpB = Bs[dotIdx * BN + threadCol];
64          for (uint resIdx = 0; resIdx < TM; ++resIdx) {
65              threadResults[resIdx] +=
66                  As[(threadRow * TM + resIdx) * BK + dotIdx] * tmpB;
67          }
68          // 计算 C 的每一列时，需要读取 A 的一列，但是 B 只需一个元素，因此先
    记作 tmpB
69      }
70      __syncthreads();    // 同步
71  }
72
73  // write out the results
74  for (uint resIdx = 0; resIdx < TM; ++resIdx) {
75      C[(threadRow * TM + resIdx) * N + threadCol] =
76          alpha * threadResults[resIdx] +
77          beta * C[(threadRow * TM + resIdx) * N + threadCol];
78  }
79  // 每个 thread 计算 TM 个
```

首先计算 Kernel 3 计算每个结果的访存：

- GMEM: K / 32 * 2 loads
- SMEM: K / 32 * BLOCKSIZE (=32) * 2 loads
- Memory accesses per result: K/16 GMEM, K * 2 SMEM

对于 Kernel 4 每个 Kernel 计算 8 个结果

- GMEM: K / 8 * 2 loads
- SMEM: K / 8 * BK(=8) * (1 + TM(=8))
- Memory accesses per result: K/32 GMEM, K * 9 / 8 SMEM

可以发现具有更少的 SMEM 访存。

4.1 Sidenote on Compiler Optimizations

先前代码的访存次数：BK * (TM + 1) = 8 * (8 + 1) = 72

```
  for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
    // we make the dotproduct loop the outside loop, which facilitates
    // reuse of the Bs entry, which we can cache in a tmp var.
    float Btmp = Bs[dotIdx * BN + threadCol];
    for (uint resIdx = 0; resIdx < TM; ++resIdx) {
      threadResults[resIdx] +=
          As[(threadRow * TM + resIdx) * BK + dotIdx] * Btmp;
    }
  }
```

没有 `tmpB` 的访存次数：TM * BK * 2 = 8 * 8 * 2 = 128

```
for (uint resIdx = 0; resIdx < TM; ++resIdx) {
  for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
    threadResults[resIdx] +=
      As[(threadRow * TM + resIdx) * BK + dotIdx] * Bs[dotIdx * BN +
threadCol];
  }
}
```

但是通过汇编代码可以发现：

```
// first inner-most loop
// r9 offset = 256 = 8 * 32, r8 offset = 4
; init %f212 = threadResults[resIdx]
ld.shared.f32   %f45, [%r9];                // %f45 = M[%r9] <->
Bs[dotIdx * BN + threadCol]
ld.shared.f32   %f46, [%r8];               // %f46 = M[%r8] <->
As[(threadRow * TM + resIdx) * BK + dotIdx]
fma.rn.f32      %f47, %f46, %f45, %f212;    // %f47 = %f46 * %f45 +
%f212
ld.shared.f32   %f48, [%r9+256];
ld.shared.f32   %f49, [%r8+4];
fma.rn.f32      %f50, %f49, %f48, %f47;
ld.shared.f32   %f51, [%r9+512];
ld.shared.f32   %f52, [%r8+8];
fma.rn.f32      %f53, %f52, %f51, %f50;
ld.shared.f32   %f54, [%r9+768];
```

```
ld.shared.f32   %f55, [%r8+12];
fma.rn.f32      %f56, %f55, %f54, %f53;
ld.shared.f32   %f57, [%r9+1024];
ld.shared.f32   %f58, [%r8+16];
fma.rn.f32      %f59, %f58, %f57, %f56;
ld.shared.f32   %f60, [%r9+1280];
ld.shared.f32   %f61, [%r8+20];
fma.rn.f32      %f62, %f61, %f60, %f59;
ld.shared.f32   %f63, [%r9+1536];
ld.shared.f32   %f64, [%r8+24];
fma.rn.f32      %f65, %f64, %f63, %f62;
ld.shared.f32   %f66, [%r9+1792];
ld.shared.f32   %f67, [%r8+28];
fma.rn.f32      %f212, %f67, %f66, %f65;
// second inner-most loop
ld.shared.f32   %f68, [%r8+32];
fma.rn.f32      %f69, %f68, %f45, %f211;    // note load %f45 before
ld.shared.f32   %f70, [%r8+36];
fma.rn.f32      %f71, %f70, %f48, %f69;
ld.shared.f32   %f72, [%r8+40];
fma.rn.f32      %f73, %f72, %f51, %f71;
ld.shared.f32   %f74, [%r8+44];
fma.rn.f32      %f75, %f74, %f54, %f73;
ld.shared.f32   %f76, [%r8+48];
fma.rn.f32      %f77, %f76, %f57, %f75;
ld.shared.f32   %f78, [%r8+52];
fma.rn.f32      %f79, %f78, %f60, %f77;
ld.shared.f32   %f80, [%r8+56];
fma.rn.f32      %f81, %f80, %f63, %f79;
ld.shared.f32   %f82, [%r8+60];
fma.rn.f32      %f211, %f82, %f66, %f81;
// ... continues like this for inner-loops 3-8 ...
```

> 编译器会做公共子表达式消除（**common subexpression elimination**），没有 `tmpB` 也不会有性能损失。

PTX -> SASS，向量化（这 part 没看懂）

```
LDS     R26, [R35.X4+0x800] // a 32b(4B) load from As
LDS.128 R8,  [R2]           // a 128b(16B) load from Bs
LDS.128 R12, [R2+0x20]
LDS     R24, [R35.X4+0x900]
LDS.128 R20, [R2+0x60]
LDS     R36, [R35.X4+0xb00]
LDS.128 R16, [R2+0x40]
LDS.128 R4,  [R2+0x80]
LDS     R38, [R35.X4+0xd00]
```

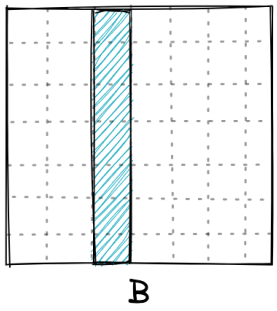
4.2 Areas of Improvement: Arithmetic Intensity

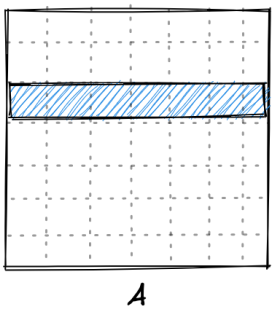
让每个线程计算更多结果 ⇒ 增加了算术强度，定义为在 GMEM 和 SMEM 之间传输（`load` 和 `store`）的每字节执行的 FLOPs 数。（相当于充分利用传输的每一字节进行更多的计算）

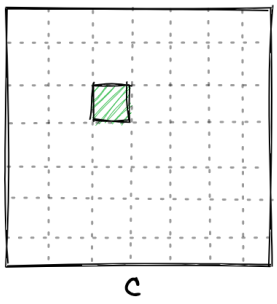
Calculating 1 result per thread requires:

- 7 loads from A
- 7 loads from B
- 1 load & 1 store to C

⇒ 15 loads & 1 store per result



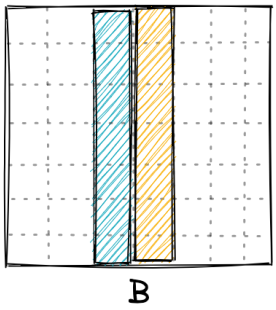


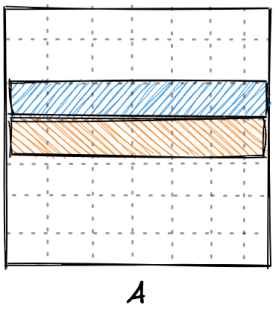


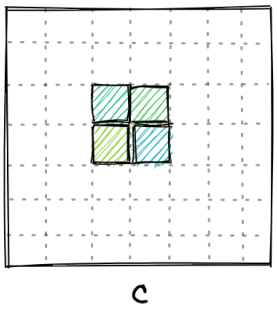
Calculating 4 results per thread requires:

- 14 loads from A
- 14 loads from B
- 4 loads & 4 stores to C

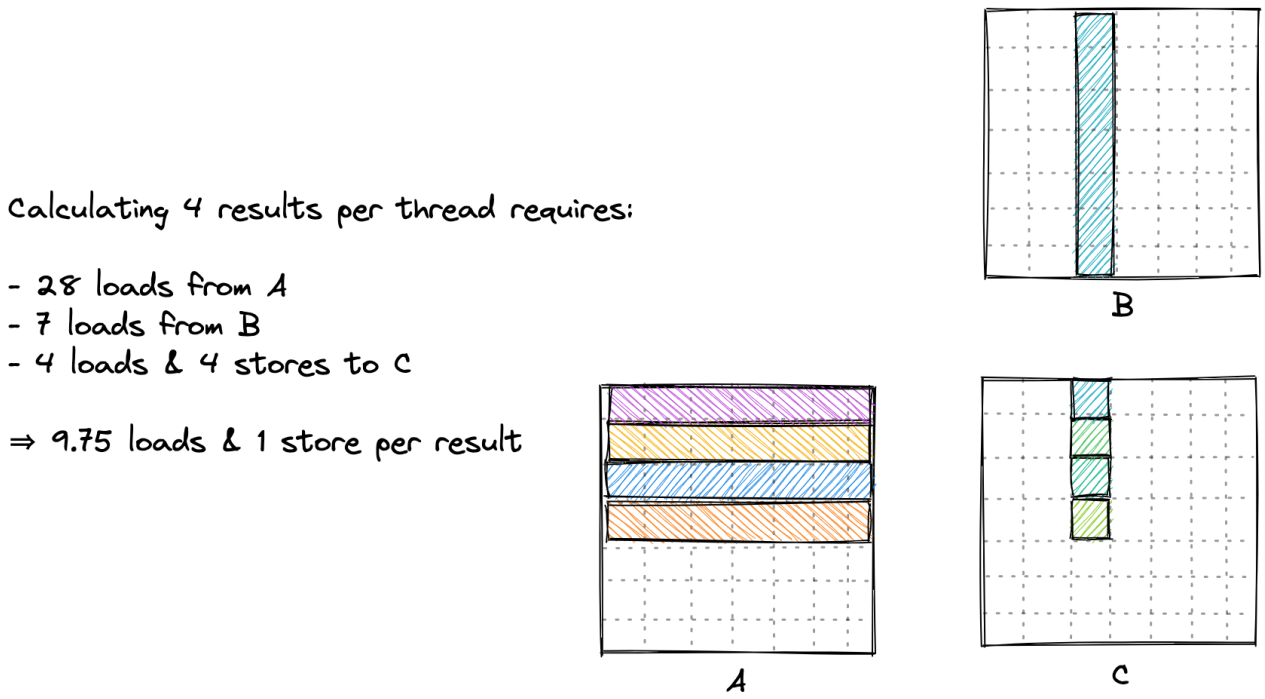
⇒ 8 loads & 1 store per result







同时，**按块**计算结果比按列计算要好。



> 每个 `thread` 只计算一个相当于行列都没被复用；计算一列，相当于 B 的那一列被复用了；而计算一块，相当于 A、B 的行列都被**复用**。

总之，所有的内核执行相同数量的 FLOP，但是可以通过每个线程计算更多的结果来减少 GMEM 访存。（目前仍然是访存受限）

5 Kernel 5: Increasing Arithmetic Intensity via 2D Blocktiling

Kernel 5 核心是每个 `thread` 计算 8 * 8 个结果。

首先看如何调用

```
const uint BK = 8;
const uint TM = 8;
const uint TN = 8;
if (M >= 128 and N >= 128) {
    const uint BM = 128;
    const uint BN = 128;
    dim3 gridDim(CEIL_DIV(N, BN), CEIL_DIV(M, BM));
    dim3 blockDim((BM * BN) / (TM * TN)); // 一维，(BM * BN) / (TM * TN) 个 thread
} else {
    // this is a hacky solution to the underlying problem
    // of not having proper bounds checking in the kernel
    const uint BM = 64;     // 这一 part 暂时没看明白
    const uint BN = 64;
    dim3 gridDim(CEIL_DIV(N, BN), CEIL_DIV(M, BM));
    dim3 blockDim((BM * BN) / (TM * TN));
```

```
}
```

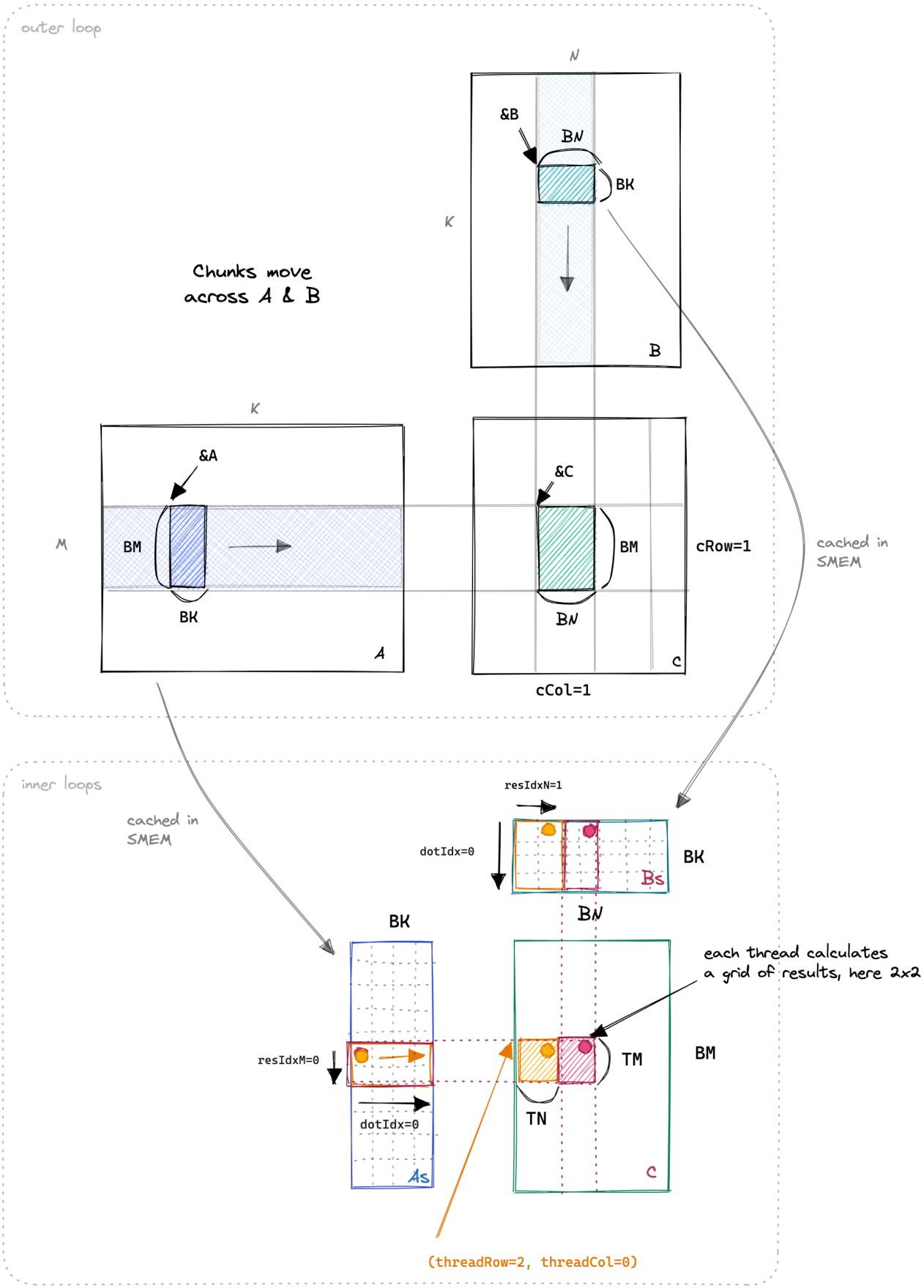
具体看 Kernel

```
// 同 Kernel 4
const uint cRow = blockIdx.y;
const uint cCol = blockIdx.x;

// 每个块计算 BM * BN 个结果
const uint totalResultsBlocktile = BM * BN;
// A thread is responsible for calculating TM*TN elements in the blocktile
const uint numThreadsBlocktile = totalResultsBlocktile / (TM * TN); // thread 数 = 256

// ResultsPerBlock / ResultsPerThread == ThreadsPerBlock
assert(numThreadsBlocktile == blockDim.x);
// blockDim.x 也即 thread 数

// BN/TN are the number of threads to span a column
const int threadCol = threadIdx.x % (BN / TN); // （宽方向上 TN 个合并，因此 BN / TN
const int threadRow = threadIdx.x / (BN / TN);

// allocate space for the current blocktile in smem
__shared__ float As[BM * BK];
__shared__ float Bs[BK * BN];
// 这里一个 thread 要加载多个元素（后面算出来是加载 4 个元素）

// Move blocktile to beginning of A's row and B's column
A += cRow * BM * K;
B += cCol * BN;
C += cRow * BM * N + cCol * BN;

// calculating the indices that this thread will load into SMEM
const uint innerRowA = threadIdx.x / BK;
const uint innerColA = threadIdx.x % BK;
// calculates the number of rows of As that are being loaded in a single step
// by a single block
const uint strideA = numThreadsBlocktile / BK;
// 可视作每一次加载 strideA(32) 行（所有 thread 一块）

const uint innerRowB = threadIdx.x / BN;
const uint innerColB = threadIdx.x % BN;
```

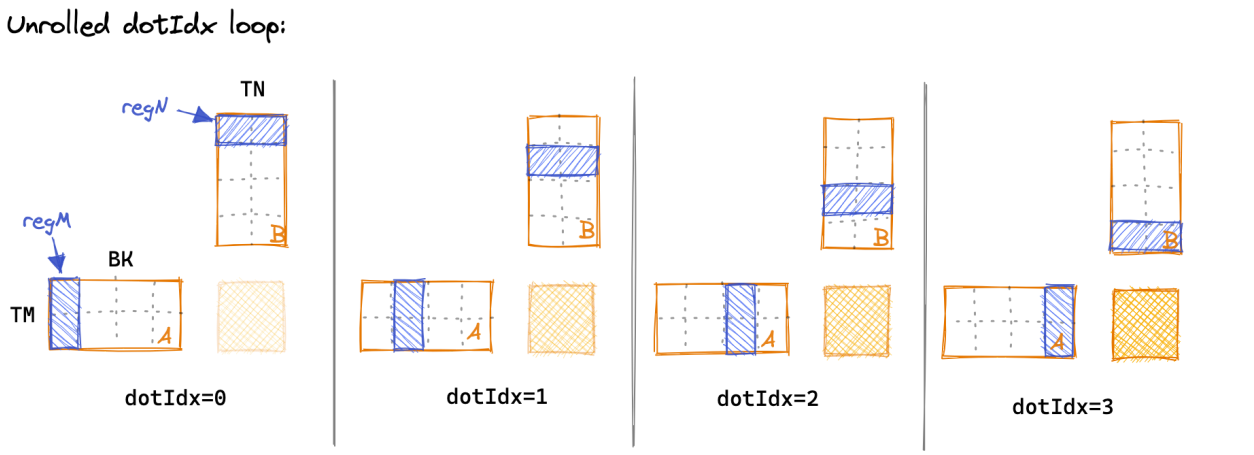
```
// for both As and Bs we want each load to span the full column-width, for
// better GMEM coalescing (as opposed to spanning full row-width and iterating
// across columns)
const uint strideB = numThreadsBlocktile / BN;
// 同理，每一次加载 strideB(=2) 行（加载行是为了更好的空间局部性）

// allocate thread-local cache for results in registerfile
float threadResults[TM * TN] = {0.0};
// register caches for As and Bs
float regM[TM] = {0.0};
float regN[TN] = {0.0};

// outer-most loop over block tiles
for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
    // populate the SMEM caches
    for (uint loadOffset = 0; loadOffset < BM; loadOffset += strideA) { // 循环 4 次
        As[(innerRowA + loadOffset) * BK + innerColA] =
            A[(innerRowA + loadOffset) * K + innerColA];
    }
    for (uint loadOffset = 0; loadOffset < BK; loadOffset += strideB) { // 循环 4 次
        Bs[(innerRowB + loadOffset) * BN + innerColB] =
            B[(innerRowB + loadOffset) * N + innerColB];
    }   // 类似于每个 thread 是跳着行加载
    __syncthreads(); // 同步，等待局部内存加载完毕

    // advance blocktile
    A += BK;     // move BK columns to right
    B += BK * N; // move BK rows down

    // calculate per-thread results
    for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
        // block into registers
        for (uint i = 0; i < TM; ++i) {
            regM[i] = As[(threadRow * TM + i) * BK + dotIdx];
        } // 缓存 As 的一列 TM 个
        for (uint i = 0; i < TN; ++i) {
            regN[i] = Bs[dotIdx * BN + threadCol * TN + i];
        } // 缓存 Bs 的一行 TN 个
        for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
            for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
                threadResults[resIdxM * TN + resIdxN] +=
                    regM[resIdxM] * regN[resIdxN];
```

```
            }
        }
    }
    __syncthreads(); // 同步，等待计算完毕
}

// write out the results
for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
    for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
        C[(threadRow * TM + resIdxM) * N + threadCol * TN + resIdxN] =
            alpha * threadResults[resIdxM * TN + resIdxN] +
            beta * C[(threadRow * TM + resIdxM) * N + threadCol * TN +
resIdxN];
    }
}
// 计算 TM * TN 个结果
```

计算部分对照作者画的图看非常清晰



将 SMEM 加载 Registers 中也有清晰的图示：



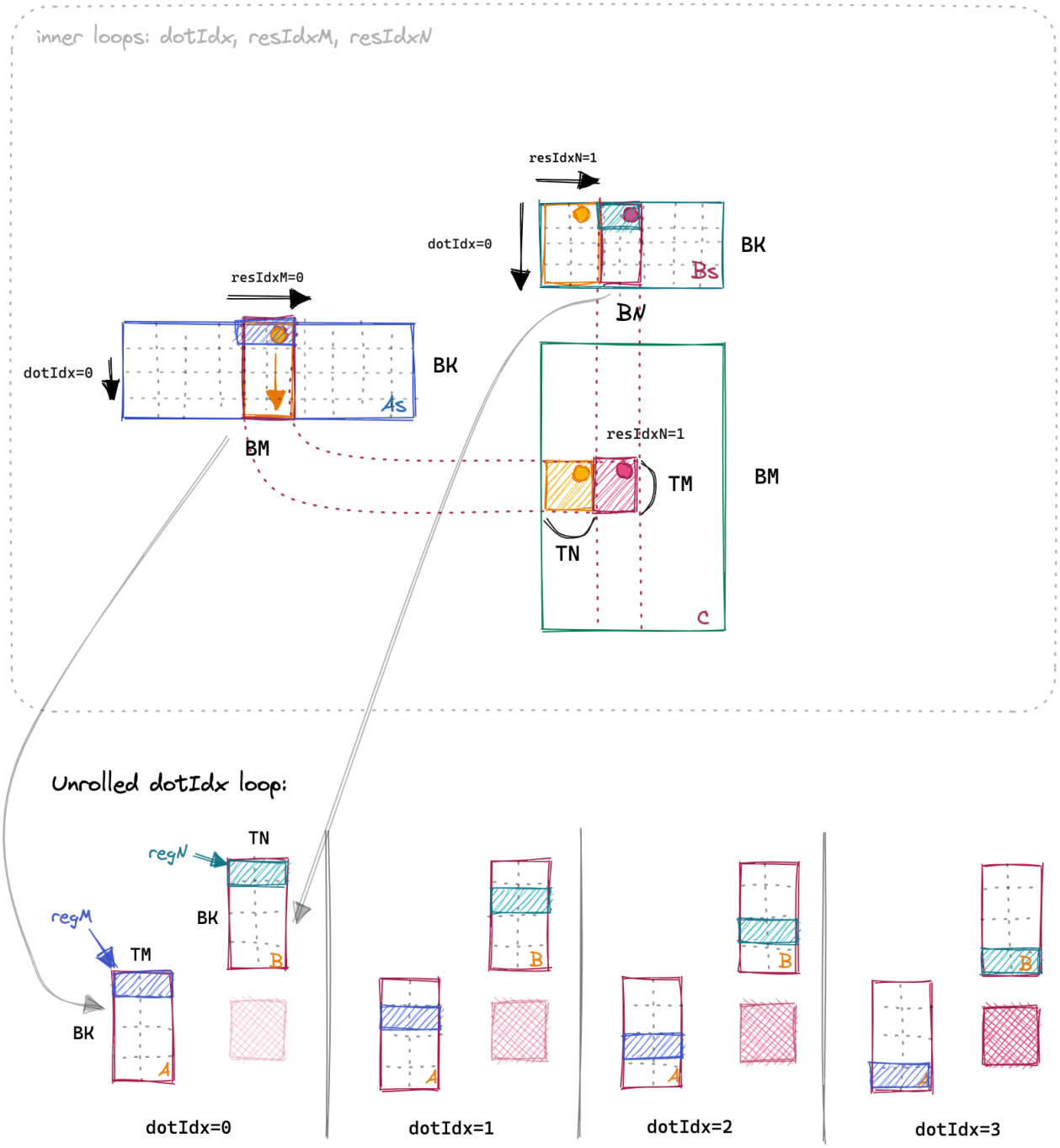
性能计算：

- GMEM: K/8 * 2 (矩阵 A 和 矩阵 B) * 1024/256 (sizeSMEM/numThreads) loads
- SMEM: K/8 * 8 (dotIdx) * 2 (A+B) * 8(TM/TN) loads
- Memory accesses per result: K/64 GMEM, K/4 SMEM

尽管性能有所提升，但是现在的 warp stall 依然明显，下一步：转置 `As` 以进行向量化 `Load` 以及 GMEM 访问编译对齐。

6 Kernel 6: Vectorize SMEM and GMEM Accesses

转置的目的是加载更长的行以具有更好的空间局部性（从图来看，原来的 As 比较瘦长，因此相当于将其横过来）



Only change to previous kernel: Now populating regM from As can also be done using a vectorized SMEM load, just like it already had been for regN

这样可以将原来的 32b `LDS` 变为 128b 的 `LDS.128`。

接下来还是用 `float4` 来从 GMEM 读取，可以将 `LDG.E` 和 `STG.E` 替换为 `LDG.E.128` 和 `STG.E.128`

```
reinterpret_cast<float4 *>(&Bs[innerRowB * BN + innerColB * 4])[0] =
    reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])[0];
```

作者提及，上面这段代码回避下面这段循环展开更快

```
Bs[innerRowB * BN + innerColB * 4 + 0] = B[innerRowB * N + innerColB * 4 + 0];
Bs[innerRowB * BN + innerColB * 4 + 1] = B[innerRowB * N + innerColB * 4 + 1];
Bs[innerRowB * BN + innerColB * 4 + 2] = B[innerRowB * N + innerColB * 4 + 2];
Bs[innerRowB * BN + innerColB * 4 + 3] = B[innerRowB * N + innerColB * 4 + 3];
```

难道编译器不能为第二个版本生成 128b `Load`？原因可能是编译器没有办法验证传递给 Kernel 的 `float* B` 指针是对齐的（这是 `LDG.E.128` 的前提），因此使用 `reinterpret_cast` 来保证对齐。

> 对比 SMEM `Load`，编译器会自动生成向量化的 `Load` 因为共享内存不是用户管理的。

```
const uint cRow = blockIdx.y;
const uint cCol = blockIdx.x;

// BN/TN are the number of threads to span a column
const int threadCol = threadIdx.x % (BN / TN);
const int threadRow = threadIdx.x / (BN / TN);

// allocate space for the current blocktile in smem
__shared__ float As[BM * BK];
__shared__ float Bs[BK * BN];

// Move blocktile to beginning of A's row and B's column
A += cRow * BM * K;
B += cCol * BN;
C += cRow * BM * N + cCol * BN;

// calculating the indices that this thread will load into SMEM
// we'll load 128bit / 32bit = 4 elements per thread at each step
const uint innerRowA = threadIdx.x / (BK / 4);
const uint innerColA = threadIdx.x % (BK / 4);
const uint innerRowB = threadIdx.x / (BN / 4);
const uint innerColB = threadIdx.x % (BN / 4);
// 多了 /4，为了一次加载 4 个 32 bit

// allocate thread-local cache for results in registerfile
float threadResults[TM * TN] = {0.0};
float regM[TM] = {0.0};
float regN[TN] = {0.0};

```

```
// outer-most loop over block tiles
for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
    // populate the SMEM caches
    // transpose A while loading it
    float4 tmp =
        reinterpret_cast<float4 *>(&A[innerRowA * K + innerColA * 4])
[0];
    As[(innerColA * 4 + 0) * BM + innerRowA] = tmp.x;
    As[(innerColA * 4 + 1) * BM + innerRowA] = tmp.y;
    As[(innerColA * 4 + 2) * BM + innerRowA] = tmp.z;
    As[(innerColA * 4 + 3) * BM + innerRowA] = tmp.w;
    // 加载一列 4 个

    // 加载一行 4 个
    reinterpret_cast<float4 *>(&Bs[innerRowB * BN + innerColB * 4])[0]
=
        reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])
[0];
    __syncthreads();

    // advance blocktile
    A += BK;     // move BK columns to right
    B += BK * N; // move BK rows down

    // calculate per-thread results
    for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
        // block into registers
        for (uint i = 0; i < TM; ++i) {
            regM[i] = As[dotIdx * BM + threadRow * TM + i];
        }
        for (uint i = 0; i < TN; ++i) {
            regN[i] = Bs[dotIdx * BN + threadCol * TN + i];
        }
        for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
            for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
                threadResults[resIdxM * TN + resIdxN] +=
                    regM[resIdxM] * regN[resIdxN];
            }
        }
    }
    __syncthreads();
}

// write out the results
for (uint resIdxM = 0; resIdxM < TM; resIdxM += 1) {
    for (uint resIdxN = 0; resIdxN < TN; resIdxN += 4) {
```

```
        // load C vector into registers
        float4 tmp = reinterpret_cast<float4 *>(
            &C[(threadRow * TM + resIdxM) * N + threadCol * TN +
resIdxN])[0];
        // perform GEMM update in reg 在 reg 中计算
        tmp.x = alpha * threadResults[resIdxM * TN + resIdxN] + beta *
tmp.x;
        tmp.y = alpha * threadResults[resIdxM * TN + resIdxN + 1] +
beta * tmp.y;
        tmp.z = alpha * threadResults[resIdxM * TN + resIdxN + 2] +
beta * tmp.z;
        tmp.w = alpha * threadResults[resIdxM * TN + resIdxN + 3] +
beta * tmp.w;
        // write back 写回
        reinterpret_cast<float4 *>(
            &C[(threadRow * TM + resIdxM) * N + threadCol * TN +
resIdxN])[0] =
            tmp;
    }
}
```

目前 `profiler` 仍然显示了一些优化的方向：

- bank conflict（cuBLAS 避免了）
- 占用率较高
- 没有实现双缓冲（CUTLASS 文档认为这很有用）

7 Kernel 7：Avoid Bank Conflicts (Linearize)

作者在文章中并没有介绍 kernel 7 和 Kernel 8。根据学过的并行计算知识，回顾以下 Bank conflict：

- Shared Memory 被分为了 16 个 bank，单位是 32-bi，相邻数据在不同的 bank 中，对 16 余数相同的数据在用一 bank
- Half warp 中的 16 个线程访问 shared memory 时最好一一对应，如果多个 thread 同时访问属于同一 bank 的数据时将发生 bank conflict
- 16 个线程**读**同一数据时，会发生一次广播，只用一个 cycle，没有 bank conflict

常见 Bank Conflict 模式：Shared Memory 存放 2D 浮点数组

- 16 × 16-element shared memory
- 1 个线程处理矩阵的一行（循环处理一行 16 个元素）
- 同一 block 的线程同时访问一列
- 16-way bank conflicts

> Kernel 5-8 的调用方式都一样

Kernel 7 与 Kernel 6 的区别如下

```
const uint innerRowB = threadIdx.x / (BN / 4);  // innerRowB 范围为 [0, 8)
const uint innerColB = threadIdx.x % (BN / 4);  // innerColB 范围为 [0, 32)

// Kernel 6
reinterpret_cast<float4 *>(&Bs[innerRowB * BN + innerColB * 4])[0] =
        reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])[0];
// innerRowB * BN + innerColB * 4 + i

// Kernel 7
// "linearize" Bs while storing it
tmp = reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])[0];
Bs[((innerColB % 2) * 4 + innerRowB * 8 + 0) * 16 + innerColB / 2] = tmp.x;
Bs[((innerColB % 2) * 4 + innerRowB * 8 + 1) * 16 + innerColB / 2] = tmp.y;
Bs[((innerColB % 2) * 4 + innerRowB * 8 + 2) * 16 + innerColB / 2] = tmp.z;
Bs[((innerColB % 2) * 4 + innerRowB * 8 + 3) * 16 + innerColB / 2] = tmp.w;
// innerRowB * BN + (innerColB % 2) * 64 + 16 * i + innerColB / 2
```

回顾一下 `Bs` 是 BK * BN（8 * 128）大小，原来的访存模式就是每个 `thread` 读取一行四个，而现在的访存模式变为了读取的 4 个元素是相隔 16 个，每行 128 个元素需要 32 个 `thread` 读取，这 32 个 `thread` 会分为 2 组，前 16 个（`innerColB % 2 == 0`）读取前 64 个元素，起点为 `innerColB / 2`。这样保证同一时刻，不会有不同的 `thread` 访问相同的 bank。

> 从结果上看，其实 Kernel 7 和 Kernel 8 效果反而是不如 Kernel 6 的。就 Kernel 7 来说确实在一定程度上避免了 bank conflict（并非完全避免，毕竟一共有 256 个 `thread`，而 bank 只有 16 个），但是性能反而下降，推测是没有向量化访存导致的。

8 Kernel 8: Avoid Bank Conflicts (Offset)

Kernel 8 与 Kernel 7 的区别如下

```
const int extraCols = 5;
__shared__ float Bs[BK * (BN + extraCols)];

tmp = reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])[0];
Bs[innerRowB * (BN + extraCols) + innerColB * 4 + 0] = tmp.x;
Bs[innerRowB * (BN + extraCols) + innerColB * 4 + 1] = tmp.y;
Bs[innerRowB * (BN + extraCols) + innerColB * 4 + 2] = tmp.z;
Bs[innerRowB * (BN + extraCols) + innerColB * 4 + 3] = tmp.w;
```

可以发现 `Bs` 多了 5 列（其实相当于整体往右移了 5 列），增加的这个 offset 相当于形成了不均匀的访问，也确实有利于减缓 bank conflict，效果比 Kernel 7 略好，但是也不如 Kernel 6（这里应该是有向量化访存的，或许是偏移之后的不均匀导致的没有对齐、cache 命中或者其他问题引起性能下降）

> 试想，原来 `innerColB = 0、1、2、3` 分别访问 bank `0-3、4-7、8-11、12-15`，`innerColB = 4` 时又会访问 `0-3` 导致 bank conflict。
>
> 而现在，向右偏移 5 列后，分别访问 `5-8、9-12、13-0、1-5、...`，bank conflict 得到了改善

9 Kernel 9: Autotuning

现在可调整的参数如下：

- BM、 BN 和 BK，确定 SMEM 块的大小，相当于从 GMEM 缓存到 SMEM 的数据量。
- TM 和 TN，确定每个 thread 计算多少结果，相当于从 SMEM 缓存到寄存器中。

Kernel 9 相当于是这些参数组合的最优版本

- 首先，这些参数组合本身得是合理的（满足一些基本约束）
- 其次，保证 Kernel 能正确计算结果

> 例：如果想向量化加载，那么 BM * BK (As 的大小)需要被 `4 * NUM_THREADS` 整除，因为块中的每个线程在每次 GMEM 到 SMEM 传输时加载 4 的倍数的数据。

```
static_assert(
    (K9_NUM_THREADS * 4) % K9_BK == 0,
    "NUM_THREADS*4 must be multiple of K9_BK to avoid quantization issues "
    "during GMEM->SMEM tiling (loading only parts of the final row of Bs "
    "during each iteraion)");
static_assert(
```

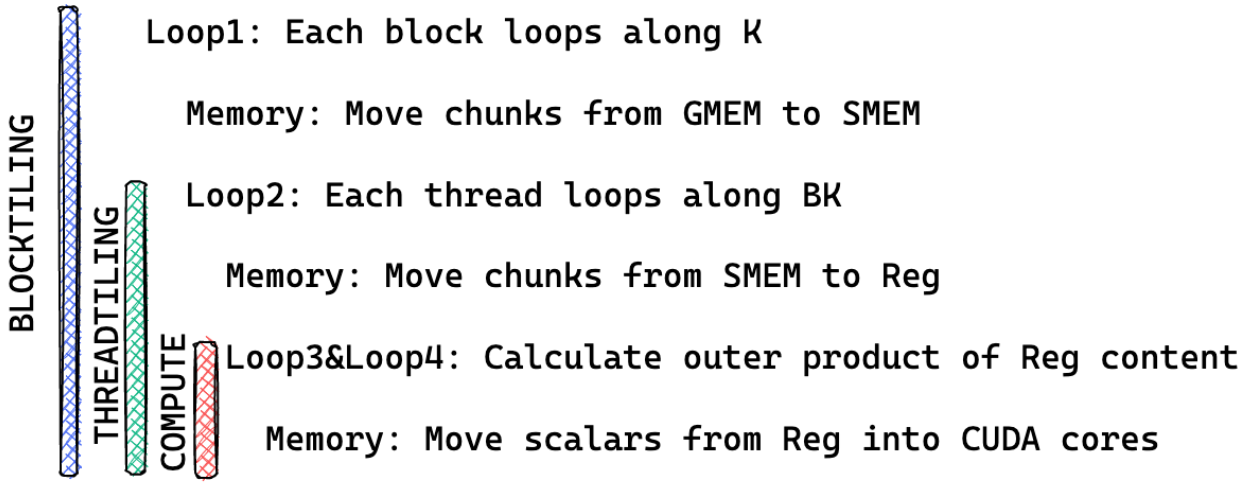
```
    (K9_NUM_THREADS * 4) % K9_BN == 0,
    "NUM_THREADS*4 must be multiple of K9_BN to avoid quantization issues "
    "during GMEM->SMEM tiling (loading only parts of the final row of As "
    "during each iteration)");
// 保证从 GMEM 到 SMEM 时都是整行读取的
static_assert(
    K9_BN % (16 * K9_TN) == 0,
    "K9_BN must be a multiple of 16*K9_TN to avoid quantization effects");
static_assert(
    K9_BM % (16 * K9_TM) == 0,
    "K9_BM must be a multiple of 16*K9_TM to avoid quantization effects");
// 16 可能和后面的 warptiling 有关
static_assert((K9_BM * K9_BK) % (4 * K9_NUM_THREADS) == 0,
              "K9_BM*K9_BK must be a multiple of 4*256 to vectorize loads");
static_assert((K9_BN * K9_BK) % (4 * K9_NUM_THREADS) == 0,
              "K9_BN*K9_BK must be a multiple of 4*256 to vectorize loads");
// 保证向量化加载
```

根据作者的结果表明：最优参数配置在不同 GPU 上表现的差异很大。同时为什么这一组参数是最优的也没法解释（每个高性能库都有自动调优工程）

> cuBLAS 可能在 cuBLAS 二进制文件中存储从{GPU 类型，矩阵大小，dtype，...}到最佳 GEMM 实现的预计算映射。
>
> A100的 fp32性能比 A6000差，Nvidia 对 A100的评级为19.5 TFLOP，对 A6000的评级为38.7 TFLOP。

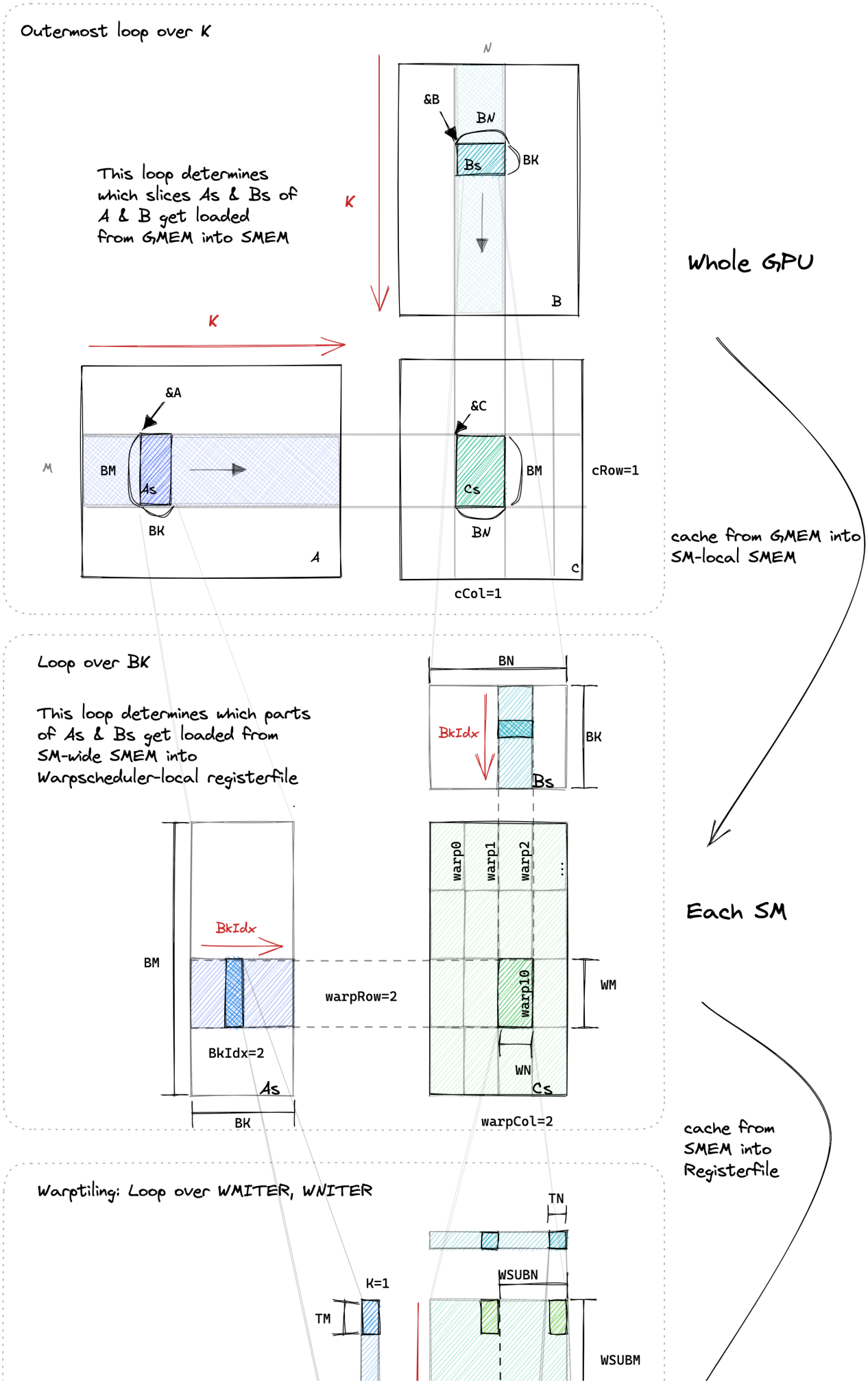
10 Kernel 10: Warptiling

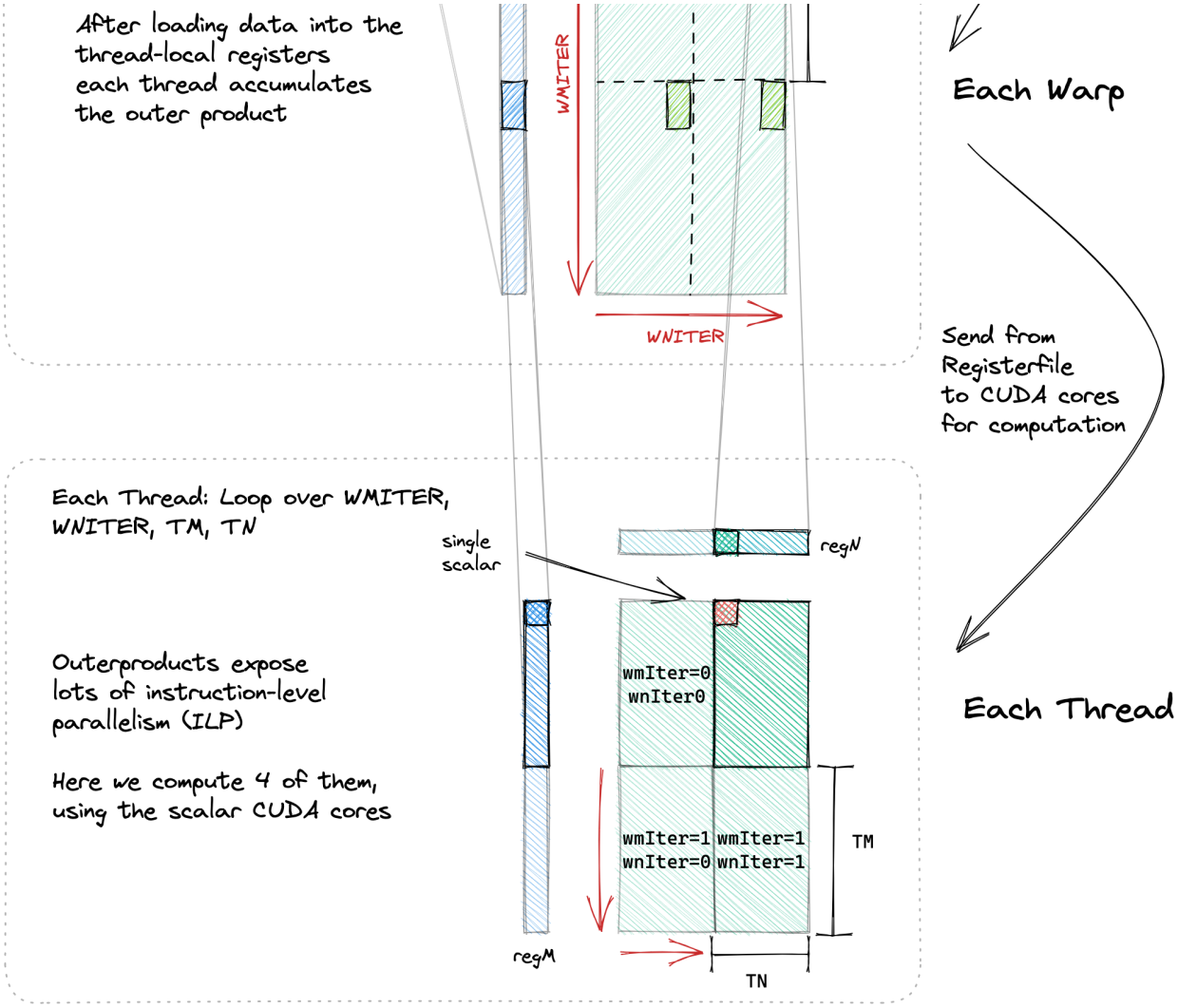
原先的层次结构是这样的



现在考虑 `blcok` 和 `thread` 中间的一层：`warp`，可以计算给定线程的 `warpId` 为 `warpId = threadIdx.x % warpSize(=32)`

每个 `warp` 将计算 `(WSUBN * WNITER) x (WSUBM * WMITER)` 的块，每个 `thread` 计算 `WNITER * WMITER` 个 `TM * TN` 大小的块。


Outermost loop over K
This loop determines which slices As & Bs of A & B get loaded from GMEM into SMEM
N
&B
BN
Bs
BK
K
B
K
&A
M
BM
As
BK
A
&C
Cs
BM
BN
C
cRow=1
cCol=1
Whole GPU
cache from GMEM into SM-local SMEM
Loop over BK
This loop determines which parts of As & Bs get loaded from SM-wide SMEM into Warpscheduler-local registerfile
BN
BkIdx
BK
Bs
BkIdx
BM
warpRow=2
BkIdx=2
As
BK
warp0
warp1
warp2
...
warp10
WM
WN
Cs
warpCol=2
Each SM
cache from SMEM into Registerfile
Warptiling: Loop over WMITER, WNITER
TN
K=1
WSUBN
TM
WSUBM



启动参数

```
const uint K10_NUM_THREADS = 128;
const uint K10_BN = 128;
const uint K10_BM = 128;
const uint K10_BK = 16;
const uint K10_WN = 64;
const uint K10_WM = 64;
const uint K10_WNITER = 4;
const uint K10_TN = 4;
const uint K10_TM = 8;
dim3 blockDim(K10_NUM_THREADS);

constexpr uint NUM_WARPS = K10_NUM_THREADS / 32; // warp 数量

// warptile in threadblocktile
static_assert((K10_BN % K10_WN == 0) and (K10_BM % K10_WM == 0));
static_assert((K10_BN / K10_WN) * (K10_BM / K10_WM) == NUM_WARPS);
// 每个 warp 加载 WM * WN 个数据

// threads in warpsubtile
```

```
static_assert((K10_WM * K10_WN) % (WARPSIZE * K10_TM * K10_TN *
K10_WNITER) ==
              0);
constexpr uint K10_WMITER =
    (K10_WM * K10_WN) / (32 * K10_TM * K10_TN * K10_WNITER);
// 每个 thread 计算 NTIER * MTIER 个 TM * TN 的小块

// warpsubtile in warptile
static_assert((K10_WM % K10_WMITER == 0) and (K10_WN % K10_WNITER ==
0));
// 保证可划分

static_assert((K10_NUM_THREADS * 4) % K10_BK == 0,
              "NUM_THREADS*4 must be multiple of K9_BK to avoid
quantization "
              "issues during GMEM->SMEM tiling (loading only parts of
the "
              "final row of Bs during each iteraion)");
static_assert((K10_NUM_THREADS * 4) % K10_BN == 0,
              "NUM_THREADS*4 must be multiple of K9_BN to avoid
quantization "
              "issues during GMEM->SMEM tiling (loading only parts of
the "
              "final row of As during each iteration)");
// 保证整行读取

static_assert(K10_BN % (16 * K10_TN) == 0,
              "BN must be a multiple of 16*TN to avoid quantization
effects");
static_assert(K10_BM % (16 * K10_TM) == 0,
              "BM must be a multiple of 16*TM to avoid quantization
effects");
// 避免 bank conflict?

static_assert((K10_BM * K10_BK) % (4 * K10_NUM_THREADS) == 0,
              "BM*BK must be a multiple of 4*256 to vectorize loads");
static_assert((K10_BN * K10_BK) % (4 * K10_NUM_THREADS) == 0,
              "BN*BK must be a multiple of 4*256 to vectorize loads");
// 保证向量化加载
dim3 gridDim(CEIL_DIV(N, K10_BN), CEIL_DIV(M, K10_BM));
```

从 GMEM 加载数据

```
// loadFromGmem
for (uint offset = 0; offset + rowStrideA <= BM; offset += rowStrideA)
{

```

```
    const float4 tmp = reinterpret_cast<const float4 *>(
        &A[(innerRowA + offset) * K + innerColA * 4])[0];
    As[(innerColA * 4 + 0) * BM + innerRowA + offset] = tmp.x;
    As[(innerColA * 4 + 1) * BM + innerRowA + offset] = tmp.y;
    As[(innerColA * 4 + 2) * BM + innerRowA + offset] = tmp.z;
    As[(innerColA * 4 + 3) * BM + innerRowA + offset] = tmp.w;
}   // 带转置，读一列

for (uint offset = 0; offset + rowStrideB <= BK; offset += rowStrideB)
{
    reinterpret_cast<float4 *>(
        &Bs[(innerRowB + offset) * BN + innerColB * 4])[0] =
        reinterpret_cast<const float4 *>(
        &B[(innerRowB + offset) * N + innerColB * 4])[0];
}   // 读一行
```

SMEM 加载到 Reg

```
// processFromSmem
for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
    // populate registers for whole warptile
    for (uint wSubRowIdx = 0; wSubRowIdx < WMITER; ++wSubRowIdx) {
        for (uint i = 0; i < TM; ++i) {
            regM[wSubRowIdx * TM + i] =
                As[(dotIdx * BM) + warpRow * WM + wSubRowIdx * WSUBM +
                   threadRowInWarp * TM + i];
        }
    }
    for (uint wSubColIdx = 0; wSubColIdx < WNITER; ++wSubColIdx) {
        for (uint i = 0; i < TN; ++i) {
            regN[wSubColIdx * TN + i] =
                Bs[(dotIdx * BN) + warpCol * WN + wSubColIdx * WSUBN +
                   threadColInWarp * TN + i];
        }
    }

    // execute warptile matmul
    for (uint wSubRowIdx = 0; wSubRowIdx < WMITER; ++wSubRowIdx) {
        for (uint wSubColIdx = 0; wSubColIdx < WNITER; ++wSubColIdx) {
            // calculate per-thread results
            for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
                for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
                    threadResults[(wSubRowIdx * TM + resIdxM) *
(WNITER * TN) +
                                  (wSubColIdx * TN) + resIdxN] +=
                        regM[wSubRowIdx * TM + resIdxM] *
```

```
                        regN[wSubColIdx * TN + resIdxN];
                }
            }
        }
    }
}
```

最后总的 Kernel

```
const uint cRow = blockIdx.y;
const uint cCol = blockIdx.x;

// Placement of the warp in the threadblock tile
const uint warpIdx = threadIdx.x / WARPSIZE; // the warp this thread is in
const uint warpCol = warpIdx % (BN / WN);
const uint warpRow = warpIdx / (BN / WN);
// 属于哪一个 warp，以及该 warp 的位置

// size of the warp subtile
constexpr uint WMITER = (WM * WN) / (WARPSIZE * TM * TN * WNITER);
constexpr uint WSUBM = WM / WMITER; // 64/2=32
constexpr uint WSUBN = WN / WNITER; // 32/2=16

// Placement of the thread in the warp subtile
const uint threadIdxInWarp = threadIdx.x % WARPSIZE;         // [0, 31]
const uint threadColInWarp = threadIdxInWarp % (WSUBN / TN); // i% (16/4)
const uint threadRowInWarp = threadIdxInWarp / (WSUBN / TN); // i/4
// thread 要计算的那一大块区域（WMTILER * TM * WNTILER * TN）的位置

// allocate space for the current blocktile in SMEM
__shared__ float As[BM * BK];
__shared__ float Bs[BK * BN];

// Move blocktile to beginning of A's row and B's column
A += cRow * BM * K;
B += cCol * BN;
// Move C_ptr to warp's output tile
C += (cRow * BM + warpRow * WM) * N + cCol * BN + warpCol * WN;

// calculating the indices that this thread will load into SMEM
// we'll load 128bit / 32bit = 4 elements per thread at each step
const uint innerRowA = threadIdx.x / (BK / 4);
const uint innerColA = threadIdx.x % (BK / 4);
```

```
constexpr uint rowStrideA = (NUM_THREADS * 4) / BK;
const uint innerRowB = threadIdx.x / (BN / 4);
const uint innerColB = threadIdx.x % (BN / 4);
constexpr uint rowStrideB = NUM_THREADS / (BN / 4);

// allocate thread-local cache for results in registerfile
float threadResults[WMITER * TM * WNITER * TN] = {0.0};
// we cache into registers on the warptile level
float regM[WMITER * TM] = {0.0};
float regN[WNITER * TN] = {0.0};

// outer-most loop over block tiles
for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
    wt::loadFromGmem<BM, BN, BK, rowStrideA, rowStrideB>(
        N, K, A, B, As, Bs, innerRowA, innerColA, innerRowB,
innerColB);
    __syncthreads();
    wt::processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER, WSUBM,
WSUBN, TM,
    TN>(regM, regN, threadResults, As, Bs, warpRow, warpCol,
        threadRowInWarp, threadColInWarp);
    A += BK;     // move BK columns to right
    B += BK * N; // move BK rows down
    __syncthreads();
}

// write out the results
for (uint wSubRowIdx = 0; wSubRowIdx < WMITER; ++wSubRowIdx) {
    for (uint wSubColIdx = 0; wSubColIdx < WNITER; ++wSubColIdx) {
        // move C pointer to current warp subtile
        float *C_interim = C + (wSubRowIdx * WSUBM) * N + wSubColIdx *
WSUBN;
        for (uint resIdxM = 0; resIdxM < TM; resIdxM += 1) {
            for (uint resIdxN = 0; resIdxN < TN; resIdxN += 4) {
                // load C vector into registers
                float4 tmp = reinterpret_cast<float4 *>(
                    &C_interim[(threadRowInWarp * TM + resIdxM) * N +
                               threadColInWarp * TN + resIdxN])[0];
                // perform GEMM update in reg
                const int i = (wSubRowIdx * TM + resIdxM) * (WNITER *
TN) +
                    wSubColIdx * TN + resIdxN;
                tmp.x = alpha * threadResults[i + 0] + beta * tmp.x;
                tmp.y = alpha * threadResults[i + 1] + beta * tmp.y;
                tmp.z = alpha * threadResults[i + 2] + beta * tmp.z;
                tmp.w = alpha * threadResults[i + 3] + beta * tmp.w;
```

```
                // write back
                reinterpret_cast<float4 *>(
                    &C_interim[(threadRowInWarp * TM + resIdxM) * N +
                               threadColInWarp * TN + resIdxN])[0] =
tmp;
            }
        }
    }
}   // 与 process 类似
```

划分得更细，是为了更好地进行指令级并行（大概）

11 Kernel 11: Double Buffering

Kernel 11 预取下一个循环迭代所需的数据，即双缓冲。

首先看调用

```
const uint K11_NUM_THREADS = 256;
const uint K11_BN = 256;
const uint K11_BM = 128;
const uint K11_BK = 16;
const uint K11_WN = 32;
const uint K11_WM = 128;
const uint K11_WNITER = 1;
const uint K11_TN = 8;
const uint K11_TM = 8;
dim3 blockDim(K11_NUM_THREADS);

constexpr uint NUM_WARPS = K11_NUM_THREADS / 32;

// warptile in threadblocktile
static_assert((K11_BN % K11_WN == 0) and (K11_BM % K11_WM == 0));
static_assert((K11_BN / K11_WN) * (K11_BM / K11_WM) == NUM_WARPS);

// threads in warpsubtile
static_assert((K11_WM * K11_WN) % (WARPSIZE * K11_TM * K11_TN *
K11_WNITER) ==
              0);
constexpr uint K11_WMITER =
    (K11_WM * K11_WN) / (32 * K11_TM * K11_TN * K11_WNITER);
// warpsubtile in warptile
static_assert((K11_WM % K11_WMITER == 0) and (K11_WN % K11_WNITER ==
0));

// 注意下面的 NUM_THREADS / 2
```

```
static_assert((K11_NUM_THREADS / 2 * 4) % K11_BK == 0,
              "NUM_THREADS*4 must be multiple of BK to avoid quantization "
              "issues during GMEM->SMEM tiling (loading only parts of the "
              "final row of Bs during each iteraion)");
static_assert((K11_NUM_THREADS / 2 * 4) % K11_BN == 0,
              "NUM_THREADS*4 must be multiple of BN to avoid quantization "
              "issues during GMEM->SMEM tiling (loading only parts of the "
              "final row of As during each iteration)");
static_assert(K11_BN % (16 * K11_TN) == 0,
              "BN must be a multiple of 16*TN to avoid quantization effects");
static_assert(K11_BM % (16 * K11_TM) == 0,
              "BM must be a multiple of 16*TM to avoid quantization effects");
static_assert((K11_BM * K11_BK) % (4 * K11_NUM_THREADS / 2) == 0,
              "BM*BK must be a multiple of 4*256 to vectorize loads");
static_assert((K11_BN * K11_BK) % (4 * K11_NUM_THREADS / 2) == 0,
              "BN*BK must be a multiple of 4*256 to vectorize loads");

dim3 gridDim(CEIL_DIV(N, K11_BN), CEIL_DIV(M, K11_BM));
```

下面直接分析源码双缓冲怎么进行的

```
const uint cRow = blockIdx.y;
const uint cCol = blockIdx.x;

// Placement of the warp in the threadblock tile
const uint warpIdx = threadIdx.x / WARPSIZE; // the warp this thread is in
const uint warpCol = warpIdx % (BN / WN);
const uint warpRow = warpIdx / (BN / WN);

// size of the warp subtile
constexpr uint WMITER = (WM * WN) / (WARPSIZE * TM * TN * WNITER);
constexpr uint WSUBM = WM / WMITER; // 64/2=32
constexpr uint WSUBN = WN / WNITER; // 32/2=16

// Placement of the thread in the warp subtile
const uint threadIdxInWarp = threadIdx.x % WARPSIZE;         // [0, 31]
const uint threadColInWarp = threadIdxInWarp % (WSUBN / TN); // i% (16/4)
```

```
const uint threadRowInWarp = threadIdxInWarp / (WSUBN / TN); // i/4

// allocate space for the current blocktile in SMEM
__shared__ float As[2 * BM * BK];   // 从这里开始不同，开了两倍的共享内存
__shared__ float Bs[2 * BK * BN];

// setup double buffering split
bool doubleBufferIdx = threadIdx.x >= (NUM_THREADS / 2); // 是否是后一半

// Move blocktile to beginning of A's row and B's column
A += cRow * BM * K;
B += cCol * BN;
// Move C_ptr to warp's output tile
C += (cRow * BM + warpRow * WM) * N + cCol * BN + warpCol * WN;

// calculating the indices that this thread will load into SMEM
// for the loading, we're pretending like there's half as many threads
// as there actually are
const uint innerRowA = (threadIdx.x % (NUM_THREADS / 2)) / (BK / 4);
const uint innerColA = (threadIdx.x % (NUM_THREADS / 2)) % (BK / 4);
constexpr uint rowStrideA = ((NUM_THREADS / 2) * 4) / BK;
const uint innerRowB = (threadIdx.x % (NUM_THREADS / 2)) / (BN / 4);
const uint innerColB = (threadIdx.x % (NUM_THREADS / 2)) % (BN / 4);
constexpr uint rowStrideB = (NUM_THREADS / 2) / (BN / 4);
// 注意这里的 NUM_THREADS / 2
// 前一半的线程和后一半的线程对应的 innerRowA 等参数都是一样的
// 本质上，这里是每个线程加载单块的时候多加载了一倍的数据量

// allocate thread-local cache for results in registerfile
float threadResults[WMITER * TM * WNITER * TN] = {0.0};
// we cache into registers on the warptile level
float regM[WMITER * TM] = {0.0};
float regN[WNITER * TN] = {0.0};

if (doubleBufferIdx == 0) { // 加载当前这块
    // load first (B0)
    db::loadFromGmem<BM, BN, BK, rowStrideA, rowStrideB>(
        N, K, A, B, As, Bs, innerRowA, innerColA, innerRowB, innerColB);
}
__syncthreads();

// outer-most loop over block tiles
for (uint bkIdx = 0; bkIdx < K; bkIdx += 2 * BK) {
```

```
    if (doubleBufferIdx == 0) {
        // process current (B0) // 处理当前这块
        db::processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER,
WSUBM, WSUBN, TM,
        TN>(regM, regN, threadResults, As, Bs, warpRow,
            warpCol, threadRowInWarp, threadColInWarp);
        __syncthreads();

        // process current+1 (B1)
        if (bkIdx + BK < K) {   // 处理后面这块
            db::processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER,
WSUBM, WSUBN,
            TM, TN>(regM, regN, threadResults, As + (BM * BK),
                    Bs + (BK * BN), warpRow, warpCol,
                    threadRowInWarp, threadColInWarp);
        }
        __syncthreads();

        // load current + 2 (B0)
        if (bkIdx + 2 * BK < K) { // 加载后面第二块
            db::loadFromGmem<BM, BN, BK, rowStrideA, rowStrideB>(
                N, K, A + 2 * BK, B + 2 * BK * N, As, Bs, innerRowA,
innerColA,
                innerRowB, innerColB);
        }
    } else {
        // load current + 1 (B1)
        if (bkIdx + BK < K) { // 加载后面这块
            db::loadFromGmem<BM, BN, BK, rowStrideA, rowStrideB>(
                N, K, A + BK, B + BK * N, As + (BM * BK), Bs + (BK *
BN), innerRowA,
                innerColA, innerRowB, innerColB);
        }
        __syncthreads();

        // process current (B0) 处理当前这块
        db::processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER,
WSUBM, WSUBN, TM,
        TN>(regM, regN, threadResults, As, Bs, warpRow,
            warpCol, threadRowInWarp, threadColInWarp);
        __syncthreads();

        // process current+1 (B1)
        if (bkIdx + BK < K) {   // 处理后面这块
            db::processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER,
WSUBM, WSUBN,
```

```
            TM, TN>(regM, regN, threadResults, As + (BM * BK),
                    Bs + (BK * BN), warpRow, warpCol,
                    threadRowInWarp, threadColInWarp);
        }
    }

    A += 2 * BK;     // move BK columns to right
    B += 2 * BK * N; // move BK rows down
    __syncthreads();
}

// write out the results
for (uint wSubRowIdx = 0; wSubRowIdx < WMITER; ++wSubRowIdx) {
    for (uint wSubColIdx = 0; wSubColIdx < WNITER; ++wSubColIdx) {
        // move C pointer to current warp subtile
        float *C_interim = C + (wSubRowIdx * WSUBM) * N + wSubColIdx
* WSUBN;
        for (uint resIdxM = 0; resIdxM < TM; resIdxM += 1) {
            for (uint resIdxN = 0; resIdxN < TN; resIdxN += 4) {
                // load C vector into registers
                float4 tmp = reinterpret_cast<float4 *>(
                    &C_interim[(threadRowInWarp * TM + resIdxM) * N +
                               threadColInWarp * TN + resIdxN])[0];
                // perform GEMM update in reg
                const int i = (wSubRowIdx * TM + resIdxM) * (WNITER *
TN) +
                    wSubColIdx * TN + resIdxN;
                tmp.x = alpha * threadResults[i + 0] + beta * tmp.x;
                tmp.y = alpha * threadResults[i + 1] + beta * tmp.y;
                tmp.z = alpha * threadResults[i + 2] + beta * tmp.z;
                tmp.w = alpha * threadResults[i + 3] + beta * tmp.w;
                // write back
                reinterpret_cast<float4 *>(
                    &C_interim[(threadRowInWarp * TM + resIdxM) * N +
                               threadColInWarp * TN + resIdxN])[0] =
tmp;
            }
        }
    }
}
```

> 如何理解 `NUM_THREADS / 2`：本质上每个线程都需要完成自己的计算（在计算上并没有差别），关键的差别在 GMEM -> SMEM 加载数据上，由原先的加载一块、处理一块变为加载两块、处理两块。

> 首先将所有线程切半（类似于处理奇偶块），初始预加载了第一块；
>
> 其次，前半部分的线程处理当前这块（即已预加载的第一块），后半部分的线程此时加载后面那块（第二块），进行一次同步；
>
> 接着，前半部分的线程处理第二块（刚刚被加载），后半部分的线程处理第一块，进行同步；
>
> 然后，前半部分的线程预加载后面的块，后半部分的线程处理第二块。
>
> 本质上，每个线程处理的数据量其实没有变，但是相当于做了更加细粒度的划分，让前半部分和后半部分的线程同步进行处理和加载，理论上来说应该是有好处的，但是从实际结果来看，效果并没有更好，或许是同步等原因引起的。（但是事实上，虽然单步循环看上去引入了更多的同步，但是原先加载两块、处理两块需要四次同步，而现在其实只需要三次，同步次数其实更少，可能是同步的等待时间更长？）

12 Kernel12：Yet Another Double Buffering

直接看源码，这次没有 `NUM_THREADS / 2`。

首先是在从 GMEM 到 SMEM 时用了一些新的函数 `cuda::memcpy_async`、`cuda::aligned_size_t` 等，从函数名来看与异步数据传输和对齐相关，这一 part 暂未深入。

后面仅列举出与 Kernel11 的区别

```
auto block = cooperative_groups::this_thread_block();
__shared__ cuda::barrier<cuda::thread_scope::thread_scope_block> frontBarrier;
__shared__ cuda::barrier<cuda::thread_scope::thread_scope_block> backBarrier;
auto frontBarrierPtr = &frontBarrier;
auto backBarrierPtr = &backBarrier;
if (block.thread_rank() == 0) {
    init(&frontBarrier, block.size());
    init(&backBarrier, block.size());
}
__syncthreads();
// 两个 barrier，大约是为了同步

// allocate space for the current blocktile in SMEM
__shared__ float As[2 * BM * BK];   // 同样开了两倍内存
__shared__ float Bs[2 * BK * BN];

```

```
int As_offset = 0;
int Bs_offset = 0;

// double-buffering: load first blocktile into SMEM
loadFromGmem<BM, BN, BK, rowStrideA, rowStrideB>(
    N, K, A, B, As + As_offset * BM * BK, Bs + Bs_offset * BK * BN,
innerRowA,
    innerColA, innerRowB, innerColB, (*frontBarrierPtr));

// outer-most loop over block tiles
for (uint bkIdx = 0; bkIdx < K - BK; bkIdx += BK) {
    // double-buffering: load next blocktile into SMEM
    loadFromGmem<BM, BN, BK, rowStrideA, rowStrideB>(
        N, K, A + BK, B + BK * N, As + (1 - As_offset) * BM * BK,
        Bs + (1 - Bs_offset) * BK * BN, innerRowA, innerColA,
innerRowB,
        innerColB, (*backBarrierPtr));

    // compute the current blocktile
    (*frontBarrierPtr).arrive_and_wait();
    processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER, WSUBM, WSUBN,
TM, TN>(
        regM, regN, threadResults, As + As_offset * BM * BK,
        Bs + Bs_offset * BK * BN, warpRow, warpCol, threadRowInWarp,
        threadColInWarp);
    A += BK;     // move BK columns to right
    B += BK * N; // move BK rows down

    As_offset = 1 - As_offset;
    Bs_offset = 1 - Bs_offset;
    // swap the front and back barriers
    auto tmp = frontBarrierPtr;
    frontBarrierPtr = backBarrierPtr;
    backBarrierPtr = tmp;

    __syncthreads();
}
// 相当于交替加载、处理，本质上一次还是加载一块、处理一块。

// compute the last blocktile
(*frontBarrierPtr).arrive_and_wait();
processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER, WSUBM, WSUBN, TM,
TN>(
    regM, regN, threadResults, As + As_offset * BM * BK,
    Bs + Bs_offset * BK * BN, warpRow, warpCol, threadRowInWarp,
    threadColInWarp);
```

> 与上一个的区别就是，每个线程在单步循环中执行的就是一次加载（下一块）和处理（当前块），这种预取理解上比较自然。

13 总结

Kernel	优化思路
1: Naive	朴素实现
2: GMEM Coalescing	合并访存
3: SMEM Caching	共享内存
4: 1D Blocktiling	单 thread 计算一列
5: 2D Blocktiling	单 thread 计算一块
6: Vectorized Mem Access	向量化访存
7: Avoid Bank Conflicts (Linearize)	划分以避免 bank conflicts
8: Avoid Bank Conflicts (Offset)	偏移以避免 bank conflicts
9: Autotuning	自动调优参数
10: Warptiling	增加 warp 层
11: Double Buffering	双缓冲，一次处理两块
12: Double Buffering 2	交替访存与计算