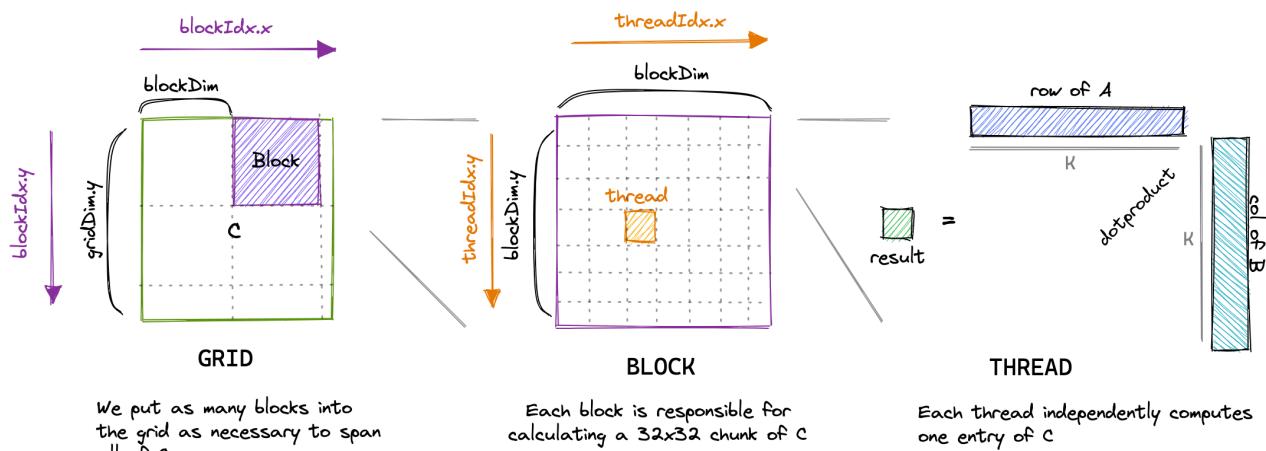


1 Kernel 1: Naive Implementation

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。

```
1 dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32));
2 dim3 blockDim(32, 32);
3
4 __global__ void sgemm_naive(int M, int N, int K, float alpha, const
5                             float *A,
6                             const float *B, float beta, float *C) {
7     // compute position in C that this thread is responsible for
8     const uint x = blockIdx.x * blockDim.x + threadIdx.x;
9     const uint y = blockIdx.y * blockDim.y + threadIdx.y;
10
11    // `if` condition is necessary for when M or N aren't multiples of
12    // 32.
13    if (x < M && y < N) {
14        float tmp = 0.0;
15        for (int i = 0; i < K; ++i) {
16            tmp += A[x * K + i] * B[i * N + y];
17        }
18        // C = α*(A@B)+β*C
19        C[x * N + y] = alpha * tmp + beta * C[x * N + y];
20    }
21}
```

```
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

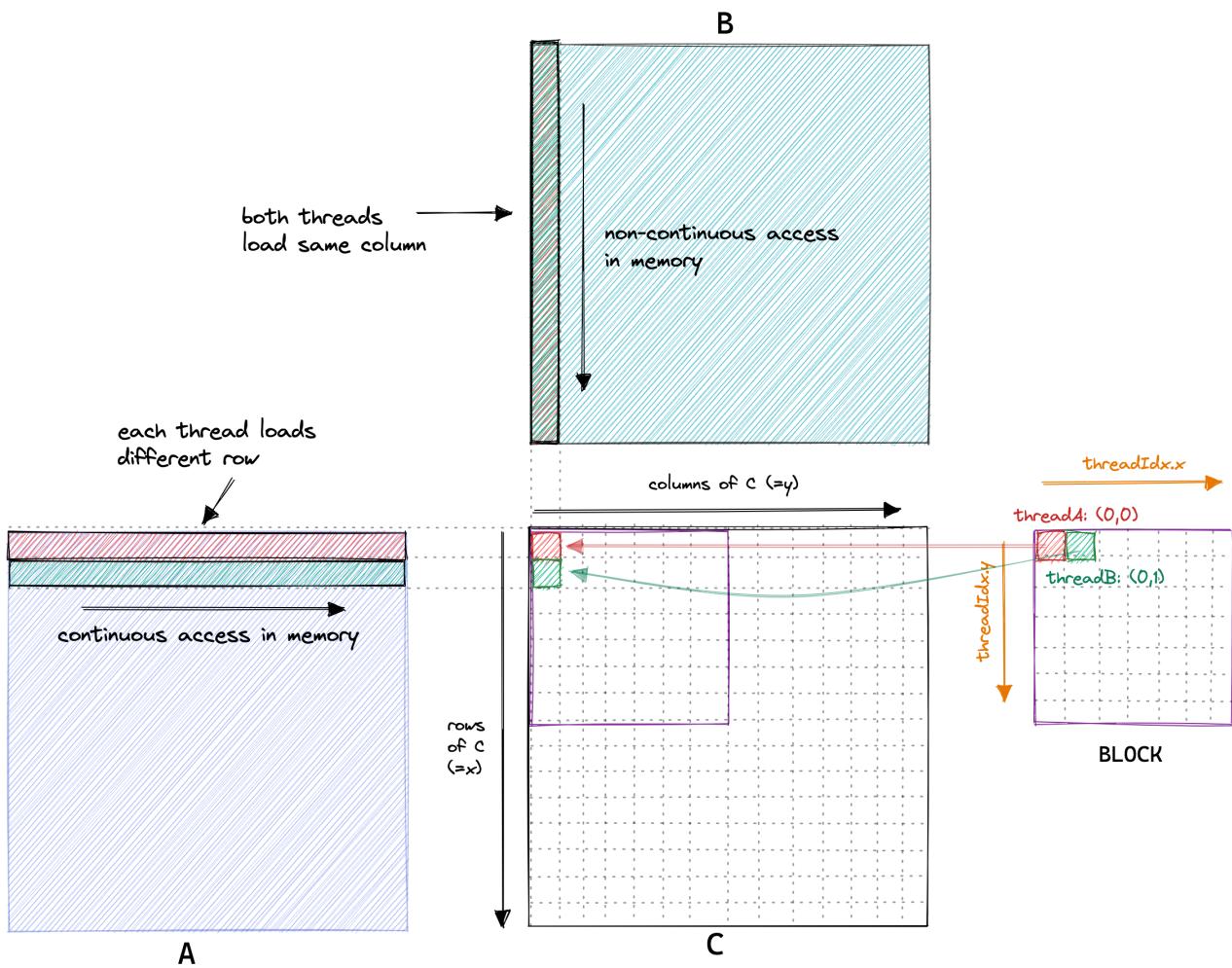
1. Total FLOPS: $2 * 4092^3 + 4092^2 = 137 \text{ GFLOPS}$ (`tmp` FMA 以及 `C` 的计算, 4092^2 前面系数不重要)
2. Total data to read (至少) : $3 * 4092^3 * 4B = 201MB$ (A、B、C 三个矩阵)
3. Total data to store: $4092^2 * 4B = 67MB$ (每个 `thread` 一个 `tmp`)

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 !



A, B, C are stored in row-major order.
This means that the last index (here y)
is the one that iterates continuous through
memory (=has stride 1).

这张图印证了前文中 `thread` 和 `result` 的映射关系

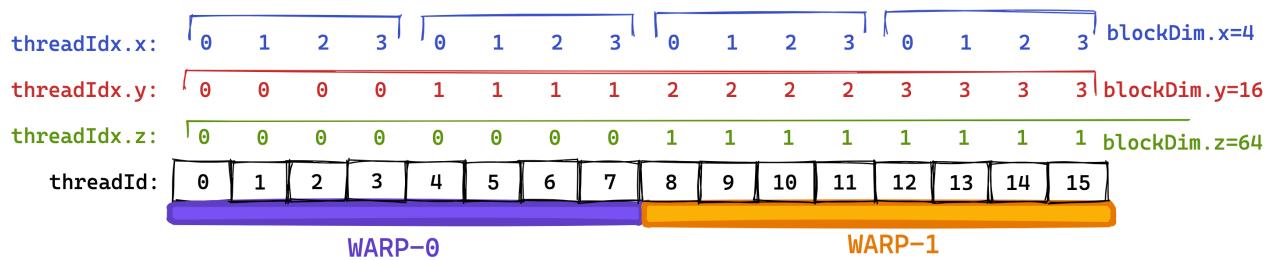
下一步：优化内存访问模式，全局内存合并访问。

2 Kernel 2: Global Memory Coalescing

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

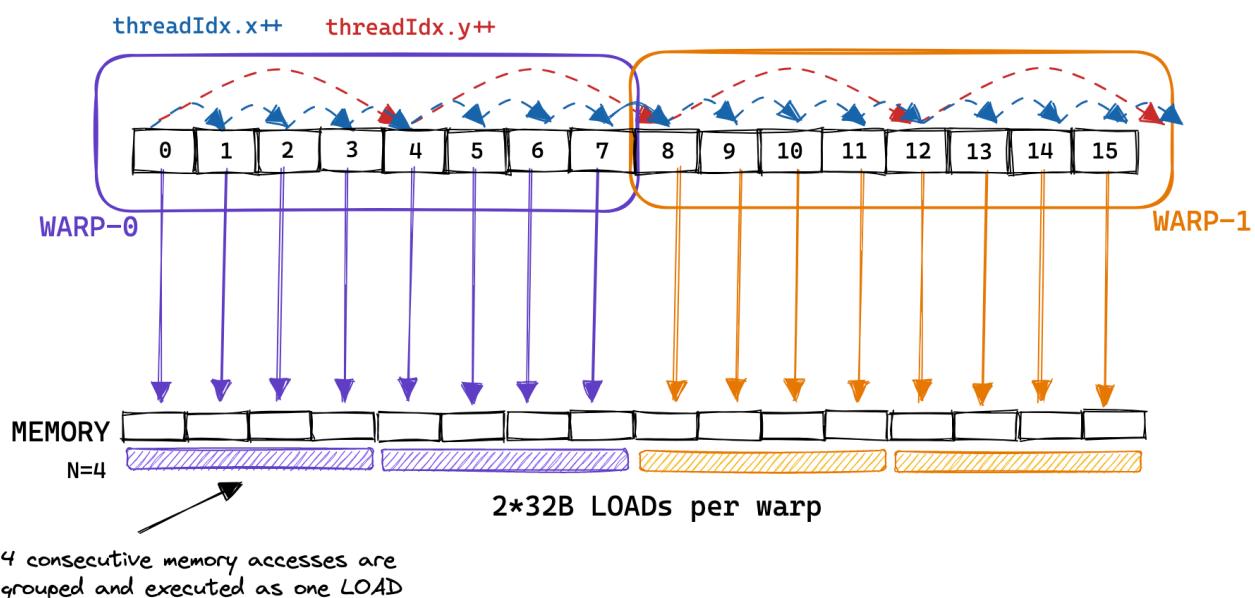
```
threadId = threadIdx.x + blockDim.x * (threadIdx.y + blockDim.y *  
threadIdx.z)
```

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
```

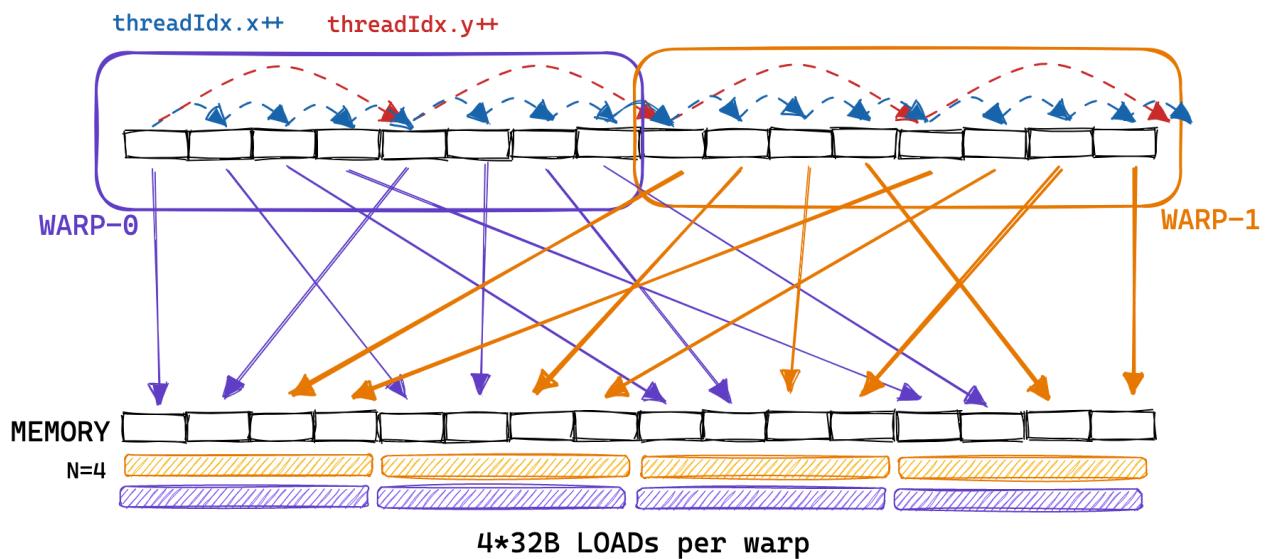
同一 warp 的部分 threads 对连续内存的访问可以合并为一个 Load !



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。

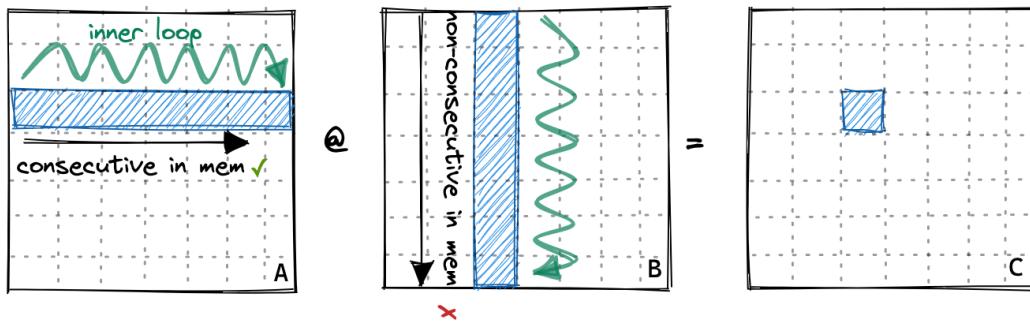
大致理解是，原来零散的 32 个 Load（或者是尽可能多的 32B Load）现在可以用一个 128B Load 替代。

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。

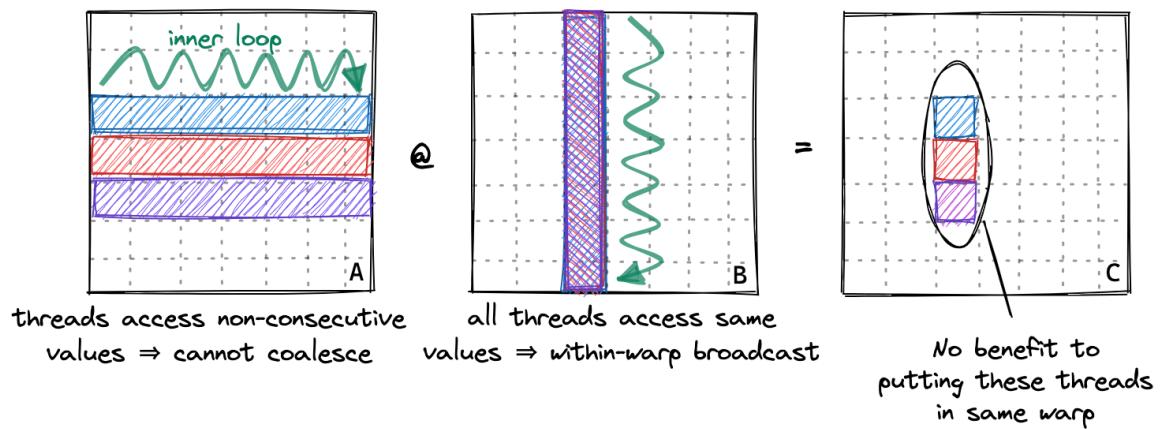


因此，重新分配线程到矩阵 C 元素的映射。

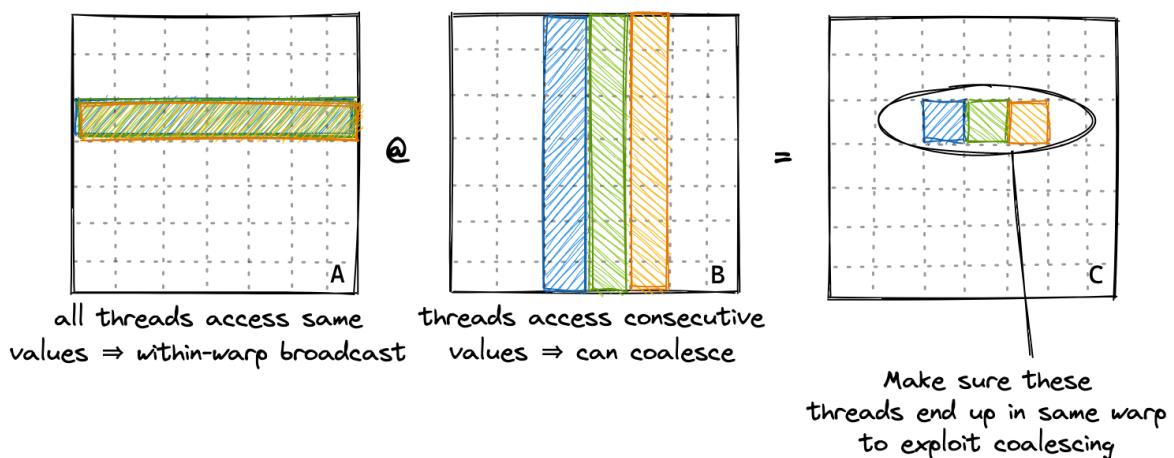
Matrix memory layout:



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：

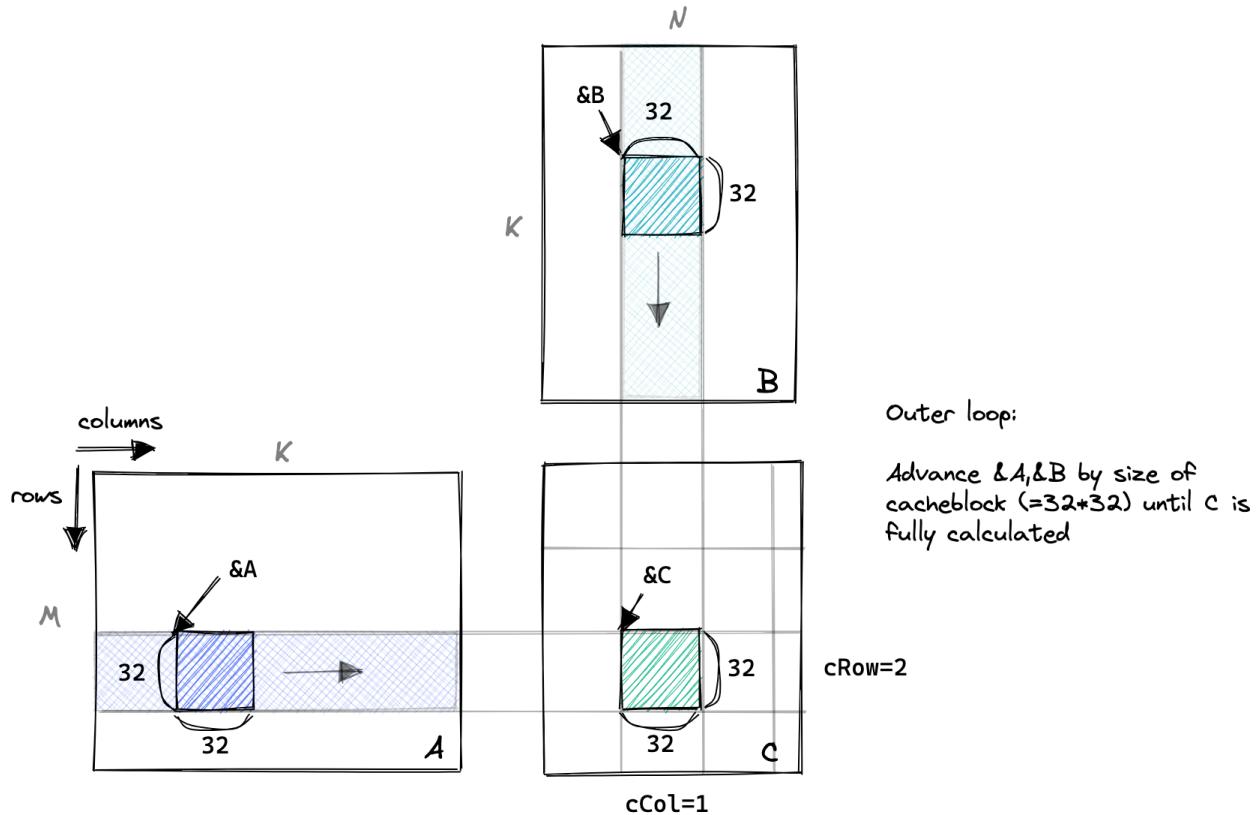
```
1 dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32));
2 dim3 blockDim(32 * 32);
3
4 const int x = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOCKSIZE);
5 const int y = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOCKSIZE);
6
7 if (x < M && y < N) {
8     float tmp = 0.0;
9     for (int i = 0; i < K; ++i) {
10         tmp += A[x * K + i] * B[i * N + y];
11     }
12     C[x * N + y] = alpha * tmp + beta * C[x * N + y];
13 }
```

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 `threadIdx.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：

```

1 dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32));
2 dim3 blockDim(32 * 32); // blockDim 仍为一维
3
4 // the output block that we want to compute in this threadblock
5 const uint cRow = blockIdx.x;    // cRow 表明计算的块在哪一行
6 const uint cCol = blockIdx.y;    // cCol 表明计算的块在哪一列
7
8 // allocate buffer for current block in fast shared mem
9 // shared mem is shared between all threads in a block

```

```

10 __shared__ float As[BLOCKSIZE * BLOCKSIZE];      // 共享内存就一个块的大小
11 __shared__ float Bs[BLOCKSIZE * BLOCKSIZE];
12
13 // the inner row & col that we're accessing in this thread
14 const uint threadCol = threadIdx.x % BLOCKSIZE; // 在 Block 里对应的位置
15 const uint threadRow = threadIdx.x / BLOCKSIZE;
16
17 // advance pointers to the starting positions
18 A += cRow * BLOCKSIZE * K;                      // row=cRow, col=0    A
在第 1 列上
19 B += cCol * BLOCKSIZE;                          // row=0, col=cCol    B
在第 1 行上
20 C += cRow * BLOCKSIZE * N + cCol * BLOCKSIZE; // row=cRow, col=cCol  C
在行列交叉的位置上
21
22 float tmp = 0.0;
23 // the outer loop advances A along the columns and B along
24 // the rows until we have fully calculated the result in C.
25 for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCKSIZE) { // 按块进行加载
26     // Have each thread load one of the elements in A & B from
27     // global memory into shared memory.
28     // Make the threadCol (=threadIdx.x) the consecutive index
29     // to allow global memory access coalescing
30     As[threadRow * BLOCKSIZE + threadCol] = A[threadRow * K +
threadCol];
31     Bs[threadRow * BLOCKSIZE + threadCol] = B[threadRow * N +
threadCol];
32     // 每个 thread 实际上只需要加载对应块的**一个**元素
33
34     // block threads in this block until cache is fully populated
35     __syncthreads(); // 同步, 保证 SMEM 加载完毕
36
37     // advance pointers onto next chunk
38     A += BLOCKSIZE; // 相当于 A 往右走
39     B += BLOCKSIZE * N; // 相当于 B 往下走
40
41     // execute the dotproduct on the currently cached block
42     for (int dotIdx = 0; dotIdx < BLOCKSIZE; ++dotIdx) {
43         tmp += As[threadRow * BLOCKSIZE + dotIdx] *
44             Bs[dotIdx * BLOCKSIZE + threadCol];
45         // 访存模式和 Kernel 2 一样
46     }
47     // need to sync again at the end, to avoid faster threads
48     // fetching the next block into the cache before slower threads are
done
49     __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 = $9216B/Block$. $(102400B \text{ per SM}) / (9216B \text{ per Block}) = 11.11 \Rightarrow 11 \text{ Blocks upper limit.}$
- **Threads:** 1024 Threads per Block, max 1536 threads per SM \Rightarrow Upper limit 1 block.
- **Registers:** $37 \text{ regs per thread} * 32 \text{ threads per warp} = 1184 \text{ regs per warp}$. 寄存器分配按照 256 一组, 上界应该是 1280 ($1024 \text{ threads} / 32$) = 32 warps per block, hence $1280 \text{ regs per warp} * 32 \text{ warps per block} = 40960 \text{ regs per block}$. Max $65536 \text{ regs per SM} \Rightarrow$ upper limit 1 block.

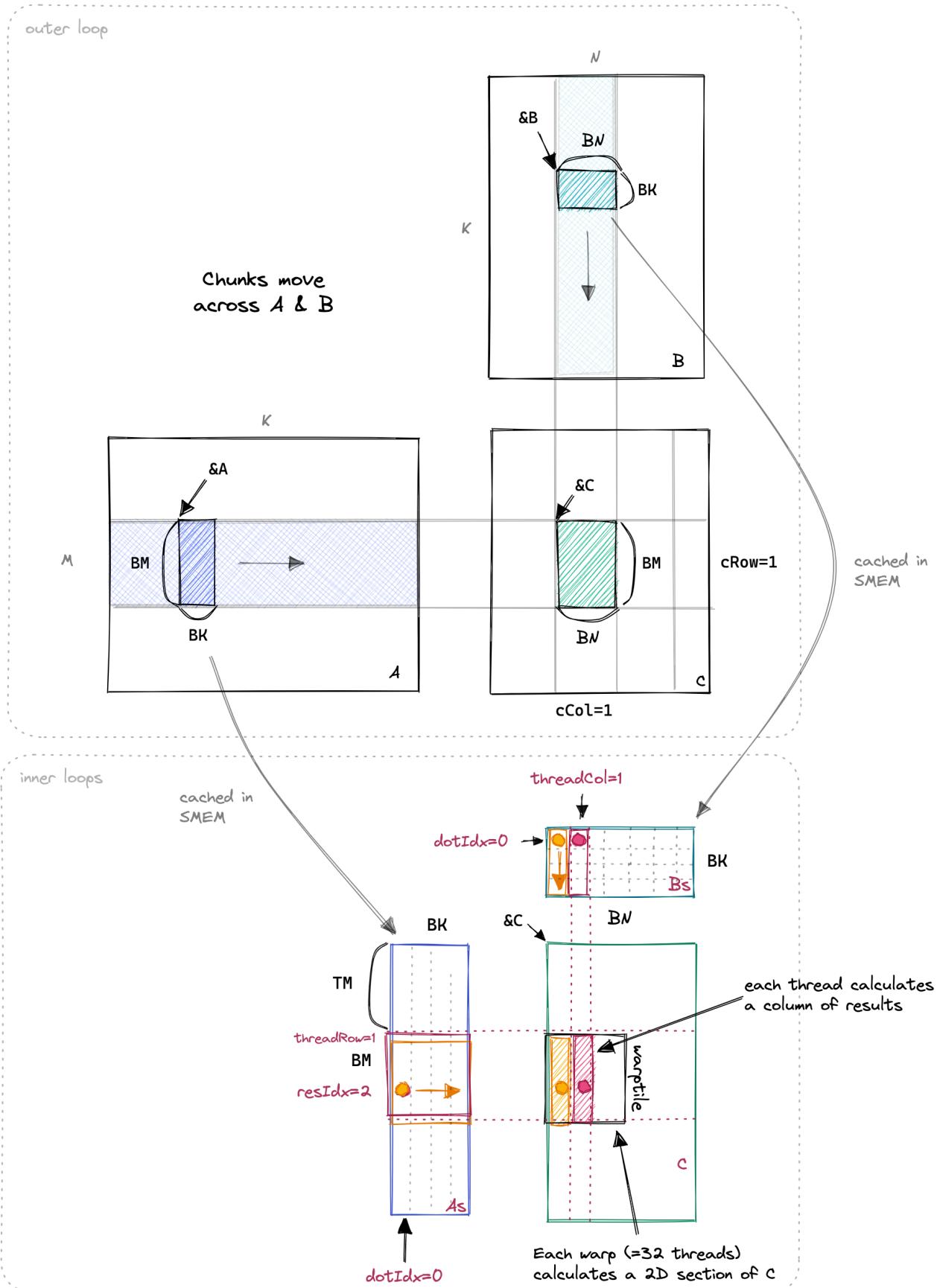
每个 SM 最多加载 1 个 block。通过 profiler 发现大部分指令时 `Load` (`Load` 比 `FMA` 指令具有更高的延迟) 进一步分析 `Warp` 状态:

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。



```

1 const uint BM = 64;
2 const uint BN = 64;
3 const uint BK = 8;
4 const uint TM = 8;
5 dim3 gridDim(CEIL_DIV(N, BN), CEIL_DIV(M, BM));

```

```

6 dim3 blockDim((BM * BN) / TM);
7
8 // If we flip x and y here we get ~30% less performance for large
matrices.
9 // The current, 30% faster configuration ensures that blocks with
sequential
10 // blockIDs access columns of B sequentially, while sharing the same
row of A.
11 // The slower configuration would share columns of A, but access into
B would
12 // be non-sequential. So the faster configuration has better spatial
locality
13 // and hence a greater L2 hit rate.
14 const uint cRow = blockIdx.y;
15 const uint cCol = blockIdx.x;
16 // 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。
17
18 // each warp will calculate 32*TM elements, with 32 being the columnar
dim.
19 const int threadCol = threadIdx.x % BN;
20 const int threadRow = threadIdx.x / BN;
21 // 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
22 // 而实际上一个 block 内 thread 只有 BM * BN / TM 个
23
24 // allocate space for the current blocktile in SMEM
25 __shared__ float As[BM * BK];
26 __shared__ float Bs[BK * BN];
27
28 // Move blocktile to beginning of A's row and B's column
29 A += cRow * BM * K;
30 B += cCol * BN;
31 C += cRow * BM * N + cCol * BN;
32
33 // todo: adjust this to each thread to load multiple entries and
34 // better exploit the cache sizes
35 assert(BM * BK == blockDim.x); // blockDim.x = BM * BN / TM = BM * BK
36 assert(BN * BK == blockDim.x); // blockDim.x = BM * BN / TM = BN * BK
37 // 这相当于 A 和 B 中的块大小和 thread 的个数是一一对应的
38 const uint innerColA = threadIdx.x % BK; // warp-level GMEM coalescing
39 const uint innerRowA = threadIdx.x / BK;
40 const uint innerColB = threadIdx.x % BN; // warp-level GMEM coalescing
41 const uint innerRowB = threadIdx.x / BN;
42 // 对于矩阵 A (块大小是 BM * BK)
43 // 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) {
    // populate the SMEM caches
    As[innerRowA * BK + innerColA] = A[innerRowA * K + innerColA];
    Bs[innerRowB * BN + innerColB] = B[innerRowB * N + innerColB];
    __syncthreads(); // 同步, 每次只需要加载 A、B 块中的一个元素
50
51     // advance blocktile
52     A += BK;
53     B += BK * N;
54
55     // calculate per-thread results
56     for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
57         // we make the dotproduct loop the outside loop, which
58         facilitates
59         // reuse of the Bs entry, which we can cache in a tmp var.
60         float tmpB = Bs[dotIdx * BN + threadCol];
61         for (uint resIdx = 0; resIdx < TM; ++resIdx) {
62             threadResults[resIdx] +=
63                 As[(threadRow * TM + resIdx) * BK + dotIdx] * tmpB;
64         }
65         // 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
66         // 记作 tmpB
67     }
68     __syncthreads(); // 同步
69 }
70
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: $K / 32 * \text{BLOCKSIZE} (=32) * 2$ 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: $BK * (TM + 1) = 8 * (8 + 1) = 72$

```

1  for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
2      // we make the dotproduct loop the outside loop, which facilitates
3      // reuse of the Bs entry, which we can cache in a tmp var.
4      float Btmp = Bs[dotIdx * BN + threadCol];
5      for (uint resIdx = 0; resIdx < TM; ++resIdx) {
6          threadResults[resIdx] +=
7              As[(threadRow * TM + resIdx) * BK + dotIdx] * Btmp;
8      }
9  }
```

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

```

1  for (uint resIdx = 0; resIdx < TM; ++resIdx) {
2      for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
3          threadResults[resIdx] +=
4              As[(threadRow * TM + resIdx) * BK + dotIdx] * Bs[dotIdx * BN +
5 threadCol];
6      }
7  }
```

但是通过汇编代码可以发现:

```

1 // first inner-most loop
2 // r9 offset = 256 = 8 * 32, r8 offset = 4
3 ; init %f212 = threadResults[resIdx]
4 ld.shared.f32    %f45, [%r9];           // %f45 = M[%r9] <->
5 Bs[dotIdx * BN + threadCol]
6 ld.shared.f32    %f46, [%r8];           // %f46 = M[%r8] <->
7 As[(threadRow * TM + resIdx) * BK + dotIdx]
8 fma.rn.f32      %f47, %f46, %f45, %f212; // %f47 = %f46 * %f45 +
9 %f212
10 ld.shared.f32   %f48, [%r9+256];
11 ld.shared.f32   %f49, [%r8+4];
12 fma.rn.f32     %f50, %f49, %f48, %f47;
13 ld.shared.f32   %f51, [%r9+512];
14 ld.shared.f32   %f52, [%r8+8];
15 fma.rn.f32     %f53, %f52, %f51, %f50;
16 ld.shared.f32   %f54, [%r9+768];
```

```

14 ld.shared.f32    %f55, [%r8+12];
15 fma.rn.f32      %f56, %f55, %f54, %f53;
16 ld.shared.f32    %f57, [%r9+1024];
17 ld.shared.f32    %f58, [%r8+16];
18 fma.rn.f32      %f59, %f58, %f57, %f56;
19 ld.shared.f32    %f60, [%r9+1280];
20 ld.shared.f32    %f61, [%r8+20];
21 fma.rn.f32      %f62, %f61, %f60, %f59;
22 ld.shared.f32    %f63, [%r9+1536];
23 ld.shared.f32    %f64, [%r8+24];
24 fma.rn.f32      %f65, %f64, %f63, %f62;
25 ld.shared.f32    %f66, [%r9+1792];
26 ld.shared.f32    %f67, [%r8+28];
27 fma.rn.f32      %f212, %f67, %f66, %f65;
28 // second inner-most loop
29 ld.shared.f32    %f68, [%r8+32];
30 fma.rn.f32      %f69, %f68, %f45, %f211; // note load %f45 before
31 ld.shared.f32    %f70, [%r8+36];
32 fma.rn.f32      %f71, %f70, %f48, %f69;
33 ld.shared.f32    %f72, [%r8+40];
34 fma.rn.f32      %f73, %f72, %f51, %f71;
35 ld.shared.f32    %f74, [%r8+44];
36 fma.rn.f32      %f75, %f74, %f54, %f73;
37 ld.shared.f32    %f76, [%r8+48];
38 fma.rn.f32      %f77, %f76, %f57, %f75;
39 ld.shared.f32    %f78, [%r8+52];
40 fma.rn.f32      %f79, %f78, %f60, %f77;
41 ld.shared.f32    %f80, [%r8+56];
42 fma.rn.f32      %f81, %f80, %f63, %f79;
43 ld.shared.f32    %f82, [%r8+60];
44 fma.rn.f32      %f211, %f82, %f66, %f81;
45 // ... 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）

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

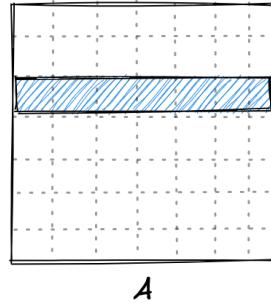
4.2 Areas of Improvement: Arithmetic Intensity

让每个线程计算更多结果 \Rightarrow 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）

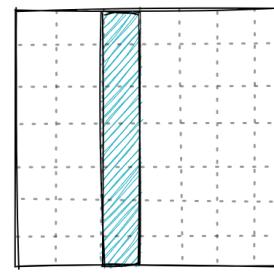
Calculating 1 result per thread requires:

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

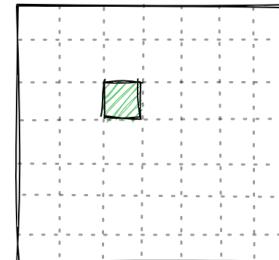
⇒ 15 loads & 1 store per result



A



B

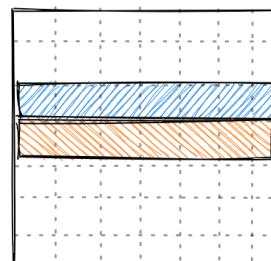


C

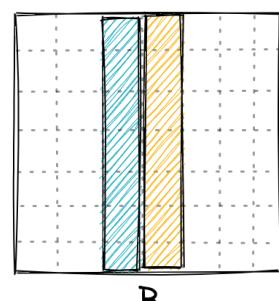
Calculating 4 results per thread requires:

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

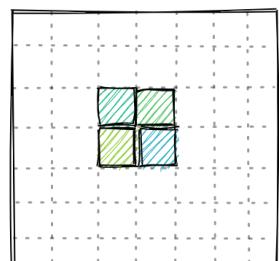
⇒ 8 loads & 1 store per result



A



B



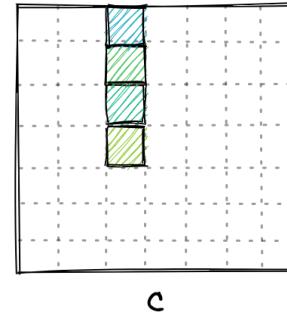
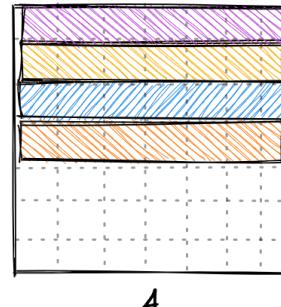
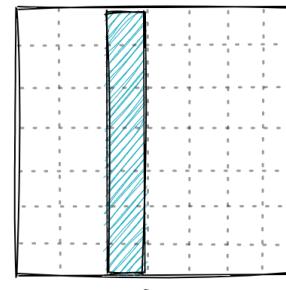
C

同时，按块计算结果比按列计算要好。

Calculating 4 results per thread requires:

- 28 loads from A
- 7 loads from B
- 4 loads & 4 stores to C

$\Rightarrow 9.75$ loads & 1 store per result



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

```
1 const uint BK = 8;
2 const uint TM = 8;
3 const uint TN = 8;
4 if (M >= 128 and N >= 128) {
5     const uint BM = 128;
6     const uint BN = 128;
7     dim3 gridDim(CEIL_DIV(N, BN), CEIL_DIV(M, BM));
8     dim3 blockDim((BM * BN) / (TM * TN)); // 一维, (BM * BN) / (TM *
TN) 个 thread
9 } else {
10    // this is a hacky solution to the underlying problem
11    // of not having proper bounds checking in the kernel
12    const uint BM = 64;      // 这一部分暂时没看明白
13    const uint BN = 64;
14    dim3 gridDim(CEIL_DIV(N, BN), CEIL_DIV(M, BM));
15    dim3 blockDim((BM * BN) / (TM * TN));
```

```
16 }
```

具体看 Kernel

```
1 // 同 Kernel 4
2 const uint cRow = blockIdx.y;
3 const uint cCol = blockIdx.x;
4
5 // 每个块计算 BM * BN 个结果
6 const uint totalResultsBlocktile = BM * BN;
7 // A thread is responsible for calculating TM*TN elements in the
8 // blocktile
9 const uint numThreadsBlocktile = totalResultsBlocktile / (TM * TN); // // thread 数 = 256
10
11 // ResultsPerBlock / ResultsPerThread == ThreadsPerBlock
12 assert(numThreadsBlocktile == blockDim.x);
13 // blockDim.x 也即 thread 数
14
15 // BN/TN are the number of threads to span a column
16 const int threadCol = threadIdx.x % (BN / TN); // (宽方向上 TN 个合并,
因此 BN / TN
17 const int threadRow = threadIdx.x / (BN / TN);
18
19 // allocate space for the current blocktile in smem
20 __shared__ float As[BM * BK];
21 __shared__ float Bs[BK * BN];
22 // 这里一个 thread 要加载多个元素 (后面算出来是加载 4 个元素)
23
24 // Move blocktile to beginning of A's row and B's column
25 A += cRow * BM * K;
26 B += cCol * BN;
27 C += cRow * BM * N + cCol * BN;
28
29 // calculating the indices that this thread will load into SMEM
30 const uint innerRowA = threadIdx.x / BK;
31 const uint innerColA = threadIdx.x % BK;
32 // calculates the number of rows of As that are being loaded in a
single step
33 // by a single block
34 const uint strideA = numThreadsBlocktile / BK;
35 // 可视作每一次加载 strideA(32) 行 (所有 thread 一块)
36 const uint innerRowB = threadIdx.x / BN;
37 const uint innerColB = threadIdx.x % BN;
```

```

38 // for both As and Bs we want each load to span the full column-width,
39 // for
40 // better GMEM coalescing (as opposed to spanning full row-width and
41 // iterating
42 // across columns)
43 const uint strideB = numThreadsBlocktile / BN;
44 // 同理, 每一次加载 strideB(=2) 行 (加载行是为了更好的空间局部性)
45
46 // allocate thread-local cache for results in registerfile
47 float threadResults[TM * TN] = {0.0};
48 // register caches for As and Bs
49 float regM[TM] = {0.0};
50 float regN[TN] = {0.0};
51
52 // outer-most loop over block tiles
53 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
}
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79

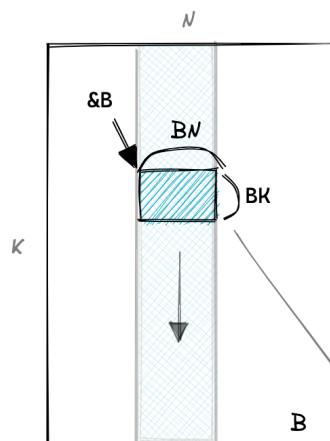
```

```
80         }
81     }
82 }
83 __syncthreads(); // 同步，等待计算完毕
84 }
85
86 // write out the results
87 for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
88     for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
89         C[(threadRow * TM + resIdxM) * N + threadCol * TN + resIdxN] =
90             alpha * threadResults[resIdxM * TN + resIdxN] +
91             beta * C[(threadRow * TM + resIdxM) * N + threadCol * TN +
92             resIdxN];
93     }
94 } // 计算 TM * TN 个结果
```

计算部分对照作者画的图看非常清晰

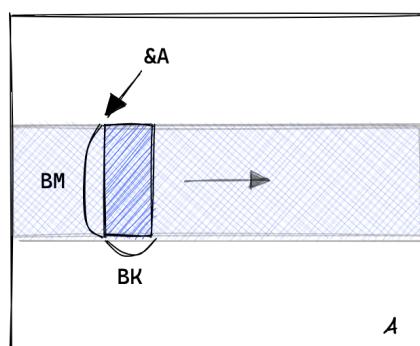
outer loop

Chunks move across A & B

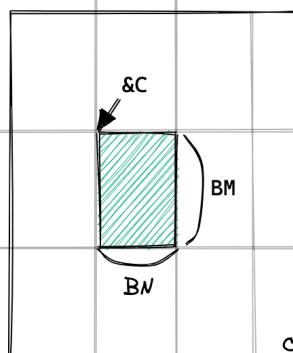


M

K



A



C

cRow=1

cached in SMEM

inner loops

cached in SMEM

resIdxN=1

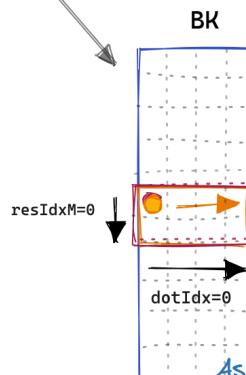
dotIdx=0

BK

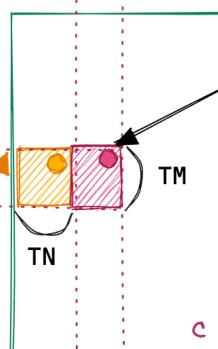


BN

each thread calculates
a grid of results, here 2x2



As



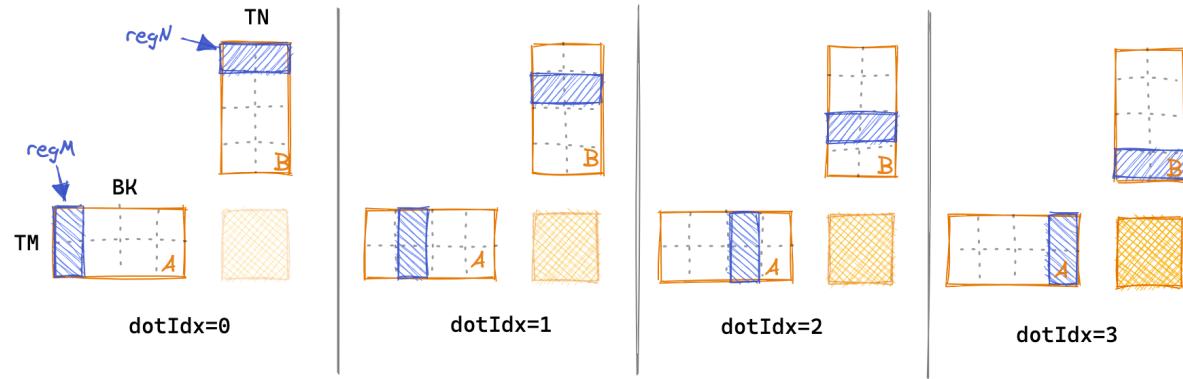
TM

BM

(threadRow=2, threadCol=0)

将 SMEM 加载 Registers 中也有清晰的图示：

Unrolled dotIdx loop:



at each timestep, load the 4 relevant As&Bs entries into regM and regN **registers**, and accumulate outer product into **threadResults**.

Benefit: We only issue 16 SMEM loads in total

性能计算：

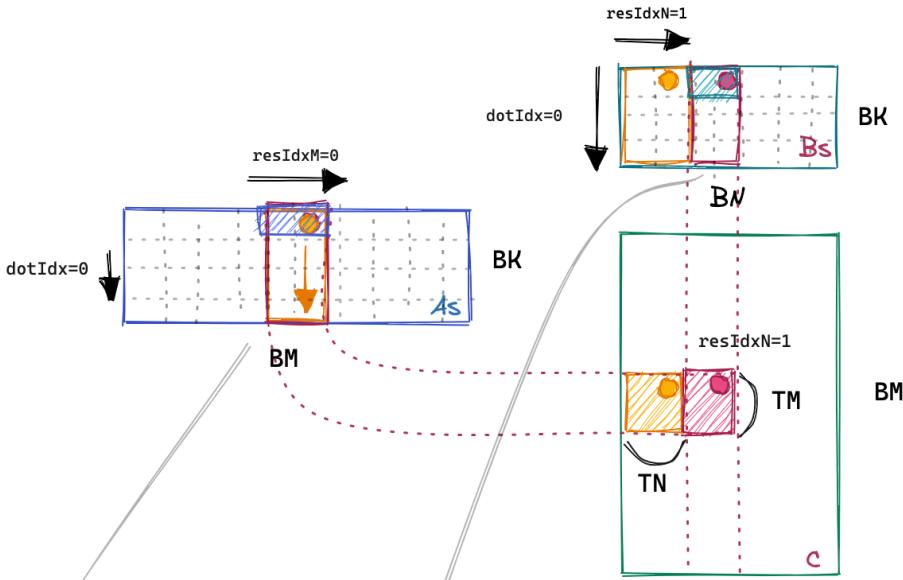
- GMEM: $K/8 * 2$ (矩阵 A 和 矩阵 B) * 1024/256 (sizeSMEM/numThreads) loads
- SMEM: $K/8 * 8$ (dotIdx) * 2 (A+B) * 8(TM/TN) loads
- Memory accesses per result: K/64 GMEM, K/4 SMEM

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。

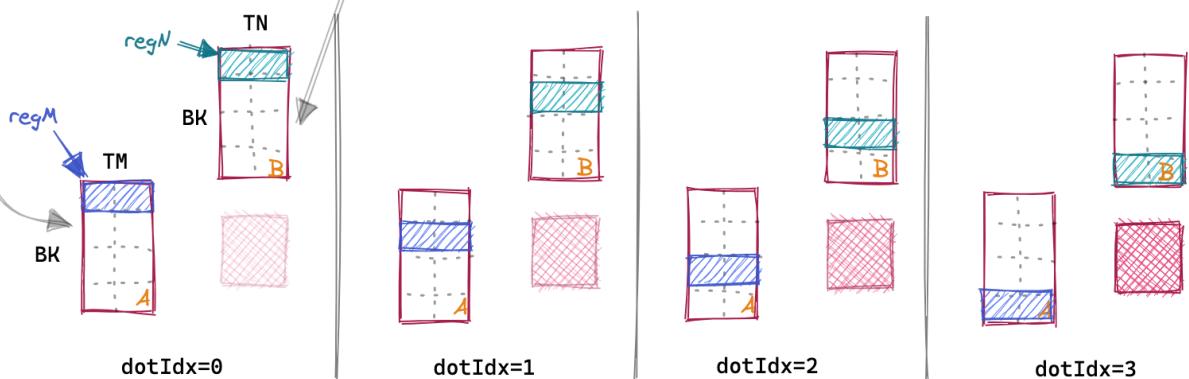
6 Kernel 6: Vectorize SMEM and GMEM Accesses

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）

inner loops: dotIdx, resIdxM, resIdxN



Unrolled dotIdx loop:



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`

```
1 | reinterpret_cast<float4 *>(&Bs[innerRowB * BN + innerColB * 4])[0] =
2 | 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

```

1 Bs[innerRowB * BN + innerColB * 4 + 0] = B[innerRowB * N + innerColB *
2   4 + 0];
3 Bs[innerRowB * BN + innerColB * 4 + 1] = B[innerRowB * N + innerColB *
4   4 + 1];
5 Bs[innerRowB * BN + innerColB * 4 + 2] = B[innerRowB * N + innerColB *
6   4 + 2];
7 Bs[innerRowB * BN + innerColB * 4 + 3] = B[innerRowB * N + innerColB *
8   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。

```

1 const uint cRow = blockIdx.y;
2 const uint cCol = blockIdx.x;
3
4 // BN/TN are the number of threads to span a column
5 const int threadCol = threadIdx.x % (BN / TN);
6 const int threadRow = threadIdx.x / (BN / TN);
7
8 // allocate space for the current blocktile in smem
9 __shared__ float As[BM * BK];
10 __shared__ float Bs[BK * BN];
11
12 // Move blocktile to beginning of A's row and B's column
13 A += cRow * BM * K;
14 B += cCol * BN;
15 C += cRow * BM * N + cCol * BN;
16
17 // calculating the indices that this thread will load into SMEM
18 // we'll load 128bit / 32bit = 4 elements per thread at each step
19 const uint innerRowA = threadIdx.x / (BK / 4);
20 const uint innerColA = threadIdx.x % (BK / 4);
21 const uint innerRowB = threadIdx.x / (BN / 4);
22 const uint innerColB = threadIdx.x % (BN / 4);
23 // 多了 /4, 为了一次加载 4 个 32 bit
24
25 // allocate thread-local cache for results in registerfile
26 float threadResults[TM * TN] = {0.0};
27 float regM[TM] = {0.0};
28 float regN[TN] = {0.0};
29

```

```

30 // outer-most loop over block tiles
31 for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
32     // populate the SMEM caches
33     // transpose A while loading it
34     float4 tmp =
35         reinterpret_cast<float4 *>(&A[innerRowA * K + innerColA * 4])
36     [0];
37     As[(innerColA * 4 + 0) * BM + innerRowA] = tmp.x;
38     As[(innerColA * 4 + 1) * BM + innerRowA] = tmp.y;
39     As[(innerColA * 4 + 2) * BM + innerRowA] = tmp.z;
40     As[(innerColA * 4 + 3) * BM + innerRowA] = tmp.w;
41     // 加载一列 4 个
42     // 加载一行 4 个
43     reinterpret_cast<float4 *>(&Bs[innerRowB * BN + innerColB * 4])[0]
44     =
45         reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])
46     [0];
47     __syncthreads();
48
49     // advance blocktile
50     A += BK;      // move BK columns to right
51     B += BK * N; // move BK rows down
52
53     // calculate per-thread results
54     for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
55         // block into registers
56         for (uint i = 0; i < TM; ++i) {
57             regM[i] = As[dotIdx * BM + threadRow * TM + i];
58         }
59         for (uint i = 0; i < TN; ++i) {
60             regN[i] = Bs[dotIdx * BN + threadCol * TN + i];
61         }
62         for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
63             for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
64                 threadResults[resIdxM * TN + resIdxN] +=
65                     regM[resIdxM] * regN[resIdxN];
66             }
67         }
68     }
69     __syncthreads();
70
71     // write out the results
72     for (uint resIdxM = 0; resIdxM < TM; resIdxM += 1) {
73         for (uint resIdxN = 0; resIdxN < TN; resIdxN += 4) {

```

```

73     // load C vector into registers
74     float4 tmp = reinterpret_cast<float4 *>(
75         &C[(threadRow * TM + resIdxM) * N + threadCol * TN +
76             resIdxN])[0];
77     // perform GEMM update in reg 在 reg 中计算
78     tmp.x = alpha * threadResults[resIdxM * TN + resIdxN] + beta *
79     tmp.x;
80     tmp.y = alpha * threadResults[resIdxM * TN + resIdxN + 1] +
81     beta * tmp.y;
82     tmp.z = alpha * threadResults[resIdxM * TN + resIdxN + 2] +
83     beta * tmp.z;
84     tmp.w = alpha * threadResults[resIdxM * TN + resIdxN + 3] +
85     beta * tmp.w;
86     // write back 写回
87     reinterpret_cast<float4 *>(
88         &C[(threadRow * TM + resIdxM) * N + threadCol * TN +
89             resIdxN])[0] =
90         tmp;
91     }
92 }
```

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-bit，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

```
1 const uint innerRowB = threadIdx.x / (BN / 4); // innerRowB 范围为 [0, 8)
2 const uint innerColB = threadIdx.x % (BN / 4); // innerColB 范围为 [0, 32)
3
4 // Kernel 6
5 reinterpret_cast<float4 *>(&Bs[innerRowB * BN + innerColB * 4])[0] =
6     reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])
7 [0];
8 // innerRowB * BN + innerColB * 4 + i
9
10 // Kernel 7
11 // "linearize" Bs while storing it
12 tmp = reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])
13 [0];
14 Bs[((innerColB % 2) * 4 + innerRowB * 8 + 0) * 16 + innerColB / 2] =
15     tmp.x;
16 Bs[((innerColB % 2) * 4 + innerRowB * 8 + 1) * 16 + innerColB / 2] =
17     tmp.y;
18 Bs[((innerColB % 2) * 4 + innerRowB * 8 + 2) * 16 + innerColB / 2] =
19     tmp.z;
20 Bs[((innerColB % 2) * 4 + innerRowB * 8 + 3) * 16 + innerColB / 2] =
21     tmp.w;
22 // 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

```

1 const int extraCols = 5;
2 __shared__ float Bs[BK * (BN + extraCols)];
3
4 tmp = reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])[0];
5 Bs[innerRowB * (BN + extraCols) + innerColB * 4 + 0] = tmp.x;
6 Bs[innerRowB * (BN + extraCols) + innerColB * 4 + 1] = tmp.y;
7 Bs[innerRowB * (BN + extraCols) + innerColB * 4 + 2] = tmp.z;
8 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。

```

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

```

```

7   (K9_NUM_THREADS * 4) % K9_BN == 0,
8   "NUM_THREADS*4 must be multiple of K9_BN to avoid quantization
9   issues "
10  "during GMEM->SMEM tiling (loading only parts of the final row of
As "
11 // 保证从 GMEM 到 SMEM 时都是整行读取的
12 static_assert(
13     K9_BN % (16 * K9_TN) == 0,
14     "K9_BN must be a multiple of 16*K9_TN to avoid quantization
15 effects");
16 static_assert(
17     K9_BM % (16 * K9_TM) == 0,
18     "K9_BM must be a multiple of 16*K9_TM to avoid quantization
19 effects");
20 // 16 可能和后面的 warptiling 有关
21 static_assert((K9_BM * K9_BK) % (4 * K9_NUM_THREADS) == 0,
22                 "K9_BM*K9_BK must be a multiple of 4*256 to vectorize
23 loads");
24 static_assert((K9_BN * K9_BK) % (4 * K9_NUM_THREADS) == 0,
25                 "K9_BN*K9_BK must be a multiple of 4*256 to vectorize
26 loads");
27 // 保证向量化加载

```

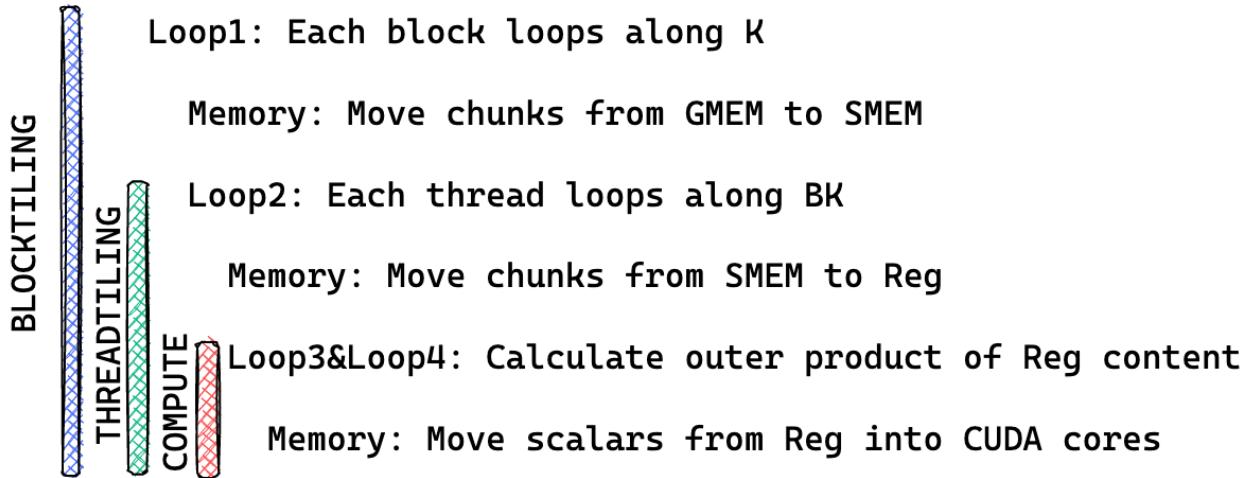
根据作者的结果表明：最优参数配置在不同 GPU 上表现的差异很大。同时为什么这一组参数是最优的也没法解释（每个高性能库都有自动调优工程）

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

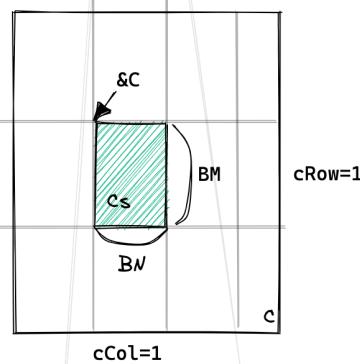
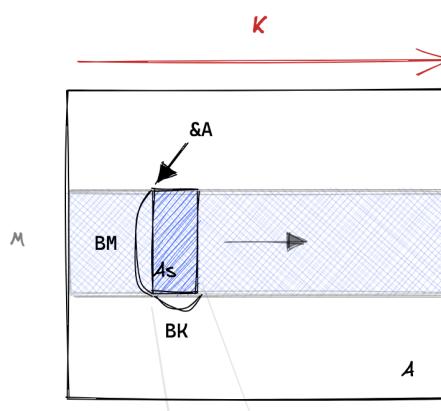
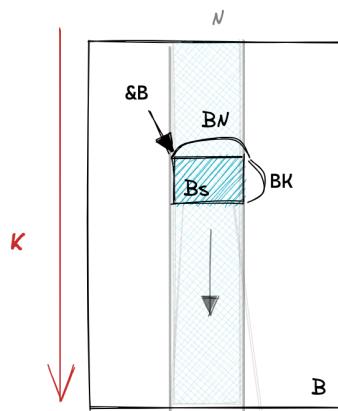


现在考虑 `block` 和 `thread` 中间的一层: `warp`, 可以计算给定线程的 `warpId` 为 `warpId = threadIdx.x % warpSize(=32)`

每个 `warp` 将计算 $(WSUBN * WNITER) \times (WSUBM * WMITER)$ 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

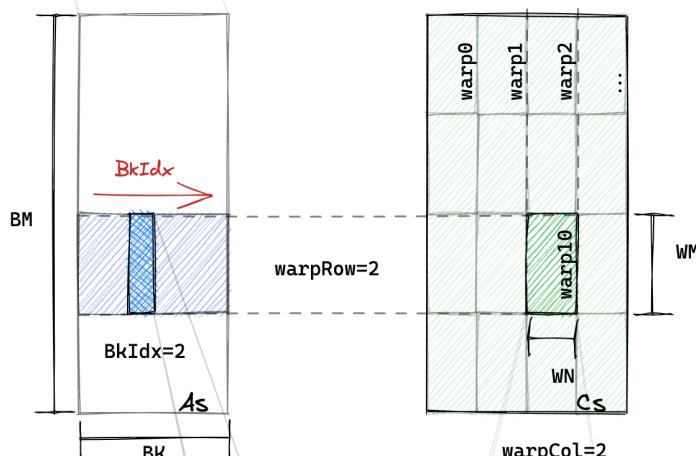
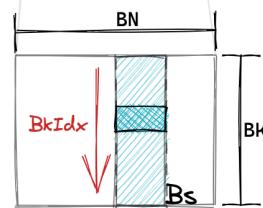


Whole GPU

cache from GMEM into SM-local SMEM

Loop over BK

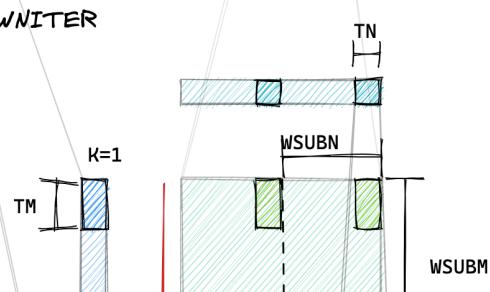
This loop determines which parts of As & Bs get loaded from SM-wide SMEM into Warpscheduler-local registerfile

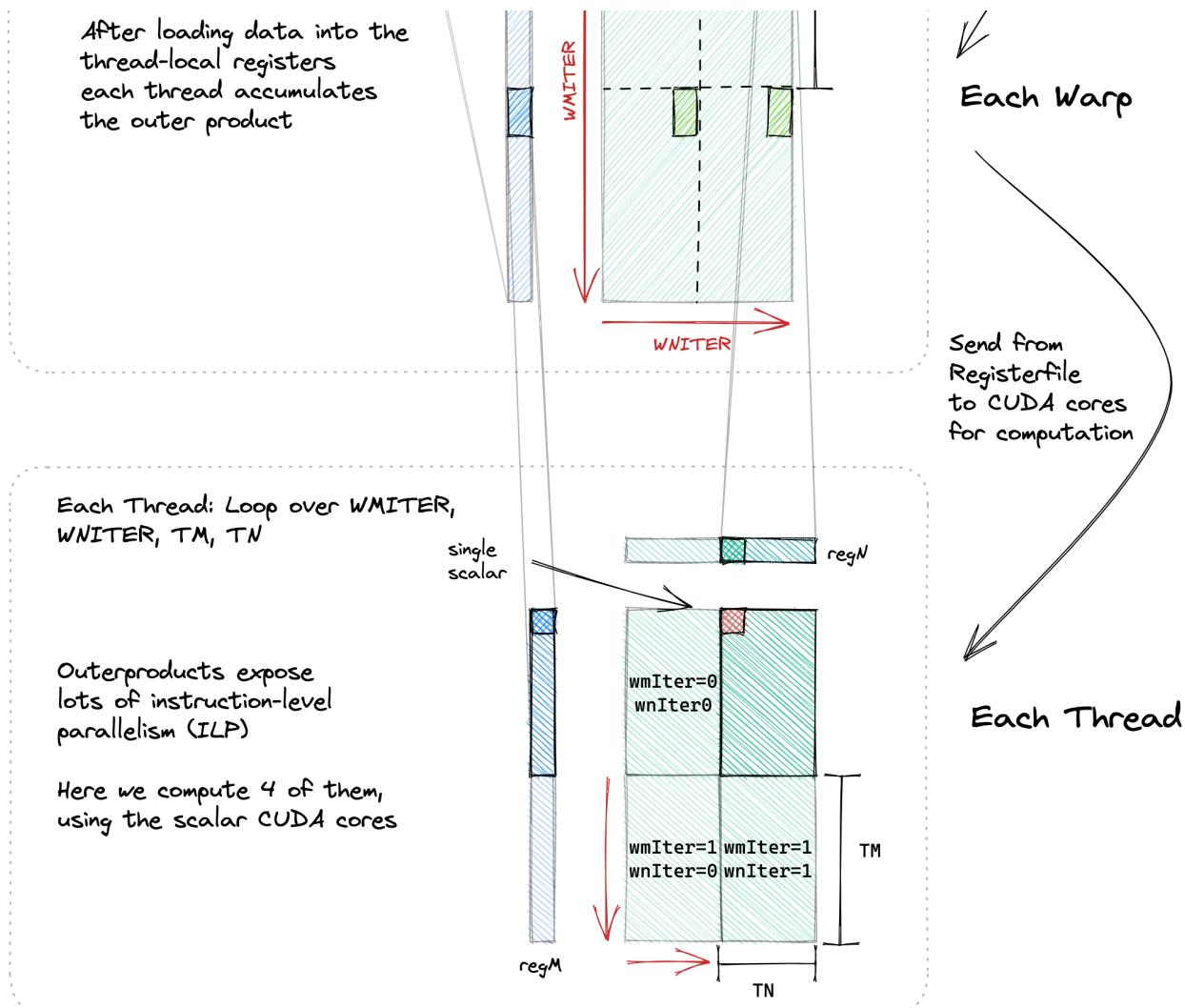


Each SM

cache from SMEM into Registerfile

Warptiling: Loop over WMITER, WNITER





启动参数

```

1 const uint K10_NUM_THREADS = 128;
2 const uint K10_BN = 128;
3 const uint K10_BM = 128;
4 const uint K10_BK = 16;
5 const uint K10_WN = 64;
6 const uint K10_WM = 64;
7 const uint K10_WNITER = 4;
8 const uint K10_TN = 4;
9 const uint K10_TM = 8;
10 dim3 blockDim(K10_NUM_THREADS);
11
12 constexpr uint NUM_WARPS = K10_NUM_THREADS / 32; // warp 数量
13
14 // warptile in threadblocktile
15 static_assert((K10_BN % K10_WN == 0) and (K10_BM % K10_WM == 0));
16 static_assert((K10_BN / K10_WN) * (K10_BM / K10_WM) == NUM_WARPS);
17 // 每个 warp 加载 WM * WN 个数据
18
19 // threads in warpsubtile

```

```

20 static_assert((K10_WM * K10_WN) % (WARPSIZE * K10_TM * K10_TN *
21 K10_WNITER) ==
22     0);
23 constexpr uint K10_WMITER =
24     (K10_WM * K10_WN) / (32 * K10_TM * K10_TN * K10_WNITER);
25 // 每个 thread 计算 NTIER * MTIER 个 TM * TN 的小块
26
27 // warpsubtile in warptile
28 static_assert((K10_WM % K10_WMITER == 0) and (K10_WN % K10_WNITER == 0));
29 // 保证可划分
30
31 static_assert((K10_NUM_THREADS * 4) % K10_BK == 0,
32               "NUM_THREADS*4 must be multiple of K9_BK to avoid
33               quantization "
34               "issues during GMEM->SMEM tiling (loading only parts of
35               the "
36               "final row of Bs during each iteration)");
37 static_assert((K10_NUM_THREADS * 4) % K10_BN == 0,
38               "NUM_THREADS*4 must be multiple of K9_BN to avoid
39               quantization "
40               "issues during GMEM->SMEM tiling (loading only parts of
41               the "
42               "final row of As during each iteration)");
43 // 保证整行读取
44
45 static_assert(K10_BN % (16 * K10_TN) == 0,
46               "BN must be a multiple of 16*TN to avoid quantization
47               effects");
48 static_assert(K10_BM % (16 * K10_TM) == 0,
49               "BM must be a multiple of 16*TM to avoid quantization
50               effects");
51 // 避免 bank conflict?
52
53 static_assert((K10_BM * K10_BK) % (4 * K10_NUM_THREADS) == 0,
54               "BM*BK must be a multiple of 4*256 to vectorize loads");
55 static_assert((K10_BN * K10_BK) % (4 * K10_NUM_THREADS) == 0,
56               "BN*BK must be a multiple of 4*256 to vectorize loads");
57 // 保证向量化加载
58 dim3 gridDim(CEIL_DIV(N, K10_BN), CEIL_DIV(M, K10_BM));

```

从 GMEM 加载数据

```

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

```

```

3   const float4 tmp = reinterpret_cast<const float4 *>(
4     &A[(innerRowA + offset) * K + innerColA * 4])[0];
5   As[(innerColA * 4 + 0) * BM + innerRowA + offset] = tmp.x;
6   As[(innerColA * 4 + 1) * BM + innerRowA + offset] = tmp.y;
7   As[(innerColA * 4 + 2) * BM + innerRowA + offset] = tmp.z;
8   As[(innerColA * 4 + 3) * BM + innerRowA + offset] = tmp.w;
9 } // 带转置, 读一列
10
11 for (uint offset = 0; offset + rowStrideB <= BK; offset += rowStrideB)
{
12   reinterpret_cast<float4 *>(
13     &Bs[(innerRowB + offset) * BN + innerColB * 4])[0] =
14   reinterpret_cast<const float4 *>(
15     &B[(innerRowB + offset) * N + innerColB * 4])[0];
16 } // 读一行

```

SMEM 加载到 Reg

```

1 // processFromSmem
2 for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
3   // populate registers for whole warptile
4   for (uint wSubRowIdx = 0; wSubRowIdx < WMITER; ++wSubRowIdx) {
5     for (uint i = 0; i < TM; ++i) {
6       regM[wSubRowIdx * TM + i] =
7         As[(dotIdx * BM) + warpRow * WM + wSubRowIdx * WSUBM +
8             threadRowInWarp * TM + i];
9     }
10   }
11   for (uint wSubColIdx = 0; wSubColIdx < WNITER; ++wSubColIdx) {
12     for (uint i = 0; i < TN; ++i) {
13       regN[wSubColIdx * TN + i] =
14         Bs[(dotIdx * BN) + warpCol * WN + wSubColIdx * WSUBN +
15             threadColInWarp * TN + i];
16     }
17   }
18
19 // execute warptile matmul
20 for (uint wSubRowIdx = 0; wSubRowIdx < WMITER; ++wSubRowIdx) {
21   for (uint wSubColIdx = 0; wSubColIdx < WNITER; ++wSubColIdx) {
22     // calculate per-thread results
23     for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
24       for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
25         threadResults[(wSubRowIdx * TM + resIdxM) *
(WNITER * TN) +
26                         (wSubColIdx * TN) + resIdxN] +=
27         regM[wSubRowIdx * TM + resIdxM] *

```

```

28                     regN[wSubColIdx * TN + resIdxN];
29                 }
30             }
31         }
32     }
33 }
```

最后总的 Kernel

```

1 const uint cRow = blockIdx.y;
2 const uint cCol = blockIdx.x;
3
4 // Placement of the warp in the threadblock tile
5 const uint warpIdx = threadIdx.x / WARPSIZE; // the warp this thread
6 const uint warpCol = warpIdx % (BN / WN);
7 const uint warpRow = warpIdx / (BN / WN);
8 // 属于哪一个 warp, 以及该 warp 的位置
9
10 // size of the warp subtile
11 constexpr uint WMITER = (WM * WN) / (WARPSIZE * TM * TN * WNITER);
12 constexpr uint WSUBM = WM / WMITER; // 64/2=32
13 constexpr uint WSUBN = WN / WNITER; // 32/2=16
14
15 // Placement of the thread in the warp subtile
16 const uint threadIdxInWarp = threadIdx.x % WARPSIZE; // [0,
17 31]
17 const uint threadColInWarp = threadIdxInWarp % (WSUBN / TN); // i%
18 (16/4)
18 const uint threadRowInWarp = threadIdxInWarp / (WSUBN / TN); // i/4
19 // thread 要计算的那一大块区域 (WMITER * TM * WNITER * TN) 的位置
20
21 // allocate space for the current blocktile in SMEM
22 __shared__ float As[BM * BK];
23 __shared__ float Bs[BK * BN];
24
25 // Move blocktile to beginning of A's row and B's column
26 A += cRow * BM * K;
27 B += cCol * BN;
28 // Move C_ptr to warp's output tile
29 C += (cRow * BM + warpRow * WM) * N + cCol * BN + warpCol * WN;
30
31 // calculating the indices that this thread will load into SMEM
32 // we'll load 128bit / 32bit = 4 elements per thread at each step
33 const uint innerRowA = threadIdx.x / (BK / 4);
34 const uint innerColA = threadIdx.x % (BK / 4);
```

```

35 constexpr uint rowStrideA = (NUM_THREADS * 4) / BK;
36 const uint innerRowB = threadIdx.x / (BN / 4);
37 const uint innerColB = threadIdx.x % (BN / 4);
38 constexpr uint rowStrideB = NUM_THREADS / (BN / 4);
39
40 // allocate thread-local cache for results in registerfile
41 float threadResults[WMITER * TM * WNITER * TN] = {0.0};
42 // we cache into registers on the warptile level
43 float regM[WMITER * TM] = {0.0};
44 float regN[WNITER * TN] = {0.0};
45
46 // outer-most loop over block tiles
47 for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
48     wt::loadFromGmem<BM, BN, BK, rowStrideA, rowStrideB>(
49         N, K, A, B, As, Bs, innerRowA, innerColA, innerRowB,
50         innerColB);
51     __syncthreads();
52     wt::processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER, WSUBM,
53     WSUBN, TM,
54     TN>(regM, regN, threadResults, As, Bs, warpRow, warpCol,
55           threadRowInWarp, threadColInWarp);
56     A += BK;      // move BK columns to right
57     B += BK * N; // move BK rows down
58     __syncthreads();
59 }
60
61 // write out the results
62 for (uint wSubRowIndex = 0; wSubRowIndex < WMITER; ++wSubRowIndex) {
63     for (uint wSubColIndex = 0; wSubColIndex < WNITER; ++wSubColIndex) {
64         // move C pointer to current warp subtile
65         float *C_interim = C + (wSubRowIndex * WSUBM) * N + wSubColIndex *
66         WSUBN;
67         for (uint resIdxM = 0; resIdxM < TM; resIdxM += 1) {
68             for (uint resIdxN = 0; resIdxN < TN; resIdxN += 4) {
69                 // load C vector into registers
70                 float4 tmp = reinterpret_cast<float4 *>(
71                     &C_interim[(threadRowInWarp * TM + resIdxM) * N +
72                         threadColInWarp * TN + resIdxN])[0];
73                 // perform GEMM update in reg
74                 const int i = (wSubRowIndex * TM + resIdxM) * (WNITER *
75                 TN) +
76                         wSubColIndex * TN + resIdxN;
77                 tmp.x = alpha * threadResults[i + 0] + beta * tmp.x;
78                 tmp.y = alpha * threadResults[i + 1] + beta * tmp.y;
79                 tmp.z = alpha * threadResults[i + 2] + beta * tmp.z;
80                 tmp.w = alpha * threadResults[i + 3] + beta * tmp.w;

```

```

77         // write back
78         reinterpret_cast<float4 *>(
79             &C_interim[(threadRowInWarp * TM + resIdxM) * N +
80                           threadColInWarp * TN + resIdxN])[0] =
81     tmp;
82 }
83 }
84 } // 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

```

1 const uint K11_NUM_THREADS = 256;
2 const uint K11_BN = 256;
3 const uint K11_BM = 128;
4 const uint K11_BK = 16;
5 const uint K11_WN = 32;
6 const uint K11_WM = 128;
7 const uint K11_WNITER = 1;
8 const uint K11_TN = 8;
9 const uint K11_TM = 8;
10 dim3 blockDim(K11_NUM_THREADS);
11
12 constexpr uint NUM_WARPS = K11_NUM_THREADS / 32;
13
14 // warptile in threadblocktile
15 static_assert((K11_BN % K11_WN == 0) and (K11_BM % K11_WM == 0));
16 static_assert((K11_BN / K11_WN) * (K11_BM / K11_WM) == NUM_WARPS);
17
18 // threads in warpsubtile
19 static_assert((K11_WM * K11_WN) % (WARPSIZE * K11_TM * K11_TN *
20 K11_WNITER) ==
21               0);
22 constexpr uint K11_WMITER =
23     (K11_WM * K11_WN) / (32 * K11_TM * K11_TN * K11_WNITER);
24 // warpsubtile in warptile
25 static_assert((K11_WM % K11_WMITER == 0) and (K11_WN % K11_WNITER ==
26 0));
27
28 // 注意下面的 NUM_THREADS / 2

```

```

27 static_assert((K11_NUM_THREADS / 2 * 4) % K11_BK == 0,
28                 "NUM_THREADS*4 must be multiple of BK to avoid
quantization "
29                 "issues during GMEM->SMEM tiling (loading only parts of
the "
30                 "final row of Bs during each iteration)");
31 static_assert((K11_NUM_THREADS / 2 * 4) % K11_BN == 0,
32                 "NUM_THREADS*4 must be multiple of BN to avoid
quantization "
33                 "issues during GMEM->SMEM tiling (loading only parts of
the "
34                 "final row of As during each iteration)");
35 static_assert(K11_BN % (16 * K11_TN) == 0,
36                 "BN must be a multiple of 16*TN to avoid quantization
effects");
37 static_assert(K11_BM % (16 * K11_TM) == 0,
38                 "BM must be a multiple of 16*TM to avoid quantization
effects");
39 static_assert((K11_BM * K11_BK) % (4 * K11_NUM_THREADS / 2) == 0,
40                 "BM*BK must be a multiple of 4*256 to vectorize loads");
41 static_assert((K11_BN * K11_BK) % (4 * K11_NUM_THREADS / 2) == 0,
42                 "BN*BK must be a multiple of 4*256 to vectorize loads");
43
44 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

```

1 const uint cRow = blockIdx.y;
2 const uint cCol = blockIdx.x;
3
4 // Placement of the warp in the threadblock tile
5 const uint warpIdx = threadIdx.x / WARPSIZE; // the warp this thread
is in
6 const uint warpCol = warpIdx % (BN / WN);
7 const uint warpRow = warpIdx / (BN / WN);
8
9 // size of the warp subtile
10 constexpr uint WMITER = (WM * WN) / (WARPSIZE * TM * TN * WNITER);
11 constexpr uint WSUBM = WM / WMITER; // 64/2=32
12 constexpr uint WSUBN = WN / WNITER; // 32/2=16
13
14 // Placement of the thread in the warp subtile
15 const uint threadIdxInWarp = threadIdx.x % WARPSIZE; // [0,
31]
16 const uint threadColInWarp = threadIdxInWarp % (WSUBN / TN); // i%
(16/4)

```

```

17 const uint threadRowInWarp = threadIdxInWarp / (WSUBN / TN); // i/4
18
19 // allocate space for the current blocktile in SMEM
20 __shared__ float As[2 * BM * BK]; // 从这里开始不同，开了两倍的共享内存
21 __shared__ float Bs[2 * BK * BN];
22
23 // setup double buffering split
24 bool doubleBufferIdx = threadIdx.x >= (NUM_THREADS / 2); // 是否是后一半
25
26 // Move blocktile to beginning of A's row and B's column
27 A += cRow * BM * K;
28 B += cCol * BN;
29 // Move C_ptr to warp's output tile
30 C += (cRow * BM + warpRow * WM) * N + cCol * BN + warpCol * WN;
31
32 // calculating the indices that this thread will load into SMEM
33 // for the loading, we're pretending like there's half as many
34 // threads
35 // as there actually are
36 const uint innerRowA = (threadIdx.x % (NUM_THREADS / 2)) / (BK / 4);
37 const uint innerColA = (threadIdx.x % (NUM_THREADS / 2)) % (BK / 4);
38 constexpr uint rowStrideA = ((NUM_THREADS / 2) * 4) / BK;
39 const uint innerRowB = (threadIdx.x % (NUM_THREADS / 2)) / (BN / 4);
40 const uint innerColB = (threadIdx.x % (NUM_THREADS / 2)) % (BN / 4);
41 constexpr uint rowStrideB = (NUM_THREADS / 2) / (BN / 4);
42 // 注意这里的 NUM_THREADS / 2
43 // 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
44 // 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
45
46 // allocate thread-local cache for results in registerfile
47 float threadResults[WMITER * TM * WNITER * TN] = {0.0};
48 // we cache into registers on the warptile level
49 float regM[WMITER * TM] = {0.0};
50 float regN[WNITER * TN] = {0.0};
51
52 if (doubleBufferIdx == 0) { // 加载当前这块
53     // load first (B0)
54     db::loadFromGmem<BM, BN, BK, rowStrideA, rowStrideB>(
55         N, K, A, B, As, Bs, innerRowA, innerColA, innerRowB,
56         innerColB);
57 }
58 __syncthreads();
59
60 // outer-most loop over block tiles
61 for (uint bkIdx = 0; bkIdx < K; bkIdx += 2 * BK) {

```

```

60     if (doubleBufferIdx == 0) {
61         // process current (B0) // 处理当前这块
62         db::processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER,
63         WSUBM, WSUBN, TM,
64             TN>(regM, regN, threadResults, As, Bs, warpRow,
65                 warpCol, threadRowInWarp, threadColInWarp);
66             __syncthreads();
67
68         // process current+1 (B1)
69         if (bkIdx + BK < K) { // 处理后面这块
70             db::processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER,
71             WSUBM, WSUBN,
72                 TM, TN>(regM, regN, threadResults, As + (BM * BK),
73                     Bs + (BK * BN), warpRow, warpCol,
74                     threadRowInWarp, threadColInWarp);
75         }
76         __syncthreads();
77
78         // load current + 2 (B0)
79         if (bkIdx + 2 * BK < K) { // 加载后面第二块
80             db::loadFromGmem<BM, BN, BK, rowStrideA, rowStrideB>(
81                 N, K, A + 2 * BK, B + 2 * BK * N, As, Bs, innerRowA,
82                 innerColA,
83                     innerRowB, innerColB);
84         }
85     } else {
86         // load current + 1 (B1)
87         if (bkIdx + BK < K) { // 加载后面这块
88             db::loadFromGmem<BM, BN, BK, rowStrideA, rowStrideB>(
89                 N, K, A + BK, B + BK * N, As + (BM * BK), Bs + (BK *
90                 BN), innerRowA,
91                     innerColA, innerRowB, innerColB);
92         }
93         __syncthreads();
94
95         // process current (B0) 处理当前这块
96         db::processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER,
97         WSUBM, WSUBN, TM,
98             TN>(regM, regN, threadResults, As, Bs, warpRow,
99                 warpCol, threadRowInWarp, threadColInWarp);
100            __syncthreads();
101
102         // process current+1 (B1)
103         if (bkIdx + BK < K) { // 处理后面这块
104             db::processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER,
105             WSUBM, WSUBN,

```

```

100         TM, TN>(regM, regN, threadResults, As + (BM * BK),
101                 Bs + (BK * BN), warpRow, warpCol,
102                 threadRowInWarp, threadColInWarp);
103     }
104 }
105
106 A += 2 * BK;      // move BK columns to right
107 B += 2 * BK * N; // move BK rows down
108 __syncthreads();
109 }
110
111 // write out the results
112 for (uint wSubRowIdx = 0; wSubRowIdx < WMITER; ++wSubRowIdx) {
113     for (uint wSubColIdx = 0; wSubColIdx < WNITER; ++wSubColIdx) {
114         // move C pointer to current warp subtile
115         float *C_interim = C + (wSubRowIdx * WSUBM) * N + wSubColIdx
116 * WSUBN;
117         for (uint resIdxM = 0; resIdxM < TM; resIdxM += 1) {
118             for (uint resIdxN = 0; resIdxN < TN; resIdxN += 4) {
119                 // load C vector into registers
120                 float4 tmp = reinterpret_cast<float4 *>(
121                     &C_interim[(threadRowInWarp * TM + resIdxM) * N +
122                                 threadColInWarp * TN + resIdxN])[0];
123                 // perform GEMM update in reg
124                 const int i = (wSubRowIdx * TM + resIdxM) * (WNITER *
125 TM) +
126                         wSubColIdx * TN + resIdxN;
127                 tmp.x = alpha * threadResults[i + 0] + beta * tmp.x;
128                 tmp.y = alpha * threadResults[i + 1] + beta * tmp.y;
129                 tmp.z = alpha * threadResults[i + 2] + beta * tmp.z;
130                 tmp.w = alpha * threadResults[i + 3] + beta * tmp.w;
131                 // write back
132                 reinterpret_cast<float4 *>(
133                     &C_interim[(threadRowInWarp * TM + resIdxM) * N +
134                                 threadColInWarp * TN + resIdxN])[0] =
135                     tmp;
136     }
}

```

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

```
1 auto block = cooperative_groups::this_thread_block();
2 __shared__ cuda::barrier<cuda::thread_scope::thread_scope_block>
frontBarrier;
3 __shared__ cuda::barrier<cuda::thread_scope::thread_scope_block>
backBarrier;
4 auto frontBarrierPtr = &frontBarrier;
5 auto backBarrierPtr = &backBarrier;
6 if (block.thread_rank() == 0) {
7     init(&frontBarrier, block.size());
8     init(&backBarrier, block.size());
9 }
10 __syncthreads();
11 // 两个 barrier, 大约是为了同步
12
13 // allocate space for the current blocktile in SMEM
14 __shared__ float As[2 * BM * BK];    // 同样开了两倍内存
15 __shared__ float Bs[2 * BK * BN];
16
```

```

17 int As_offset = 0;
18 int Bs_offset = 0;
19
20 // double-buffering: load first blocktile into SMEM
21 loadFromGmem<BM, BN, BK, rowStrideA, rowStrideB>(
22     N, K, A, B, As + As_offset * BM * BK, Bs + Bs_offset * BK * BN,
23     innerRowA,
24     innerColA, innerRowB, innerColB, (*frontBarrierPtr));
25
26 // outer-most loop over block tiles
27 for (uint bkIdx = 0; bkIdx < K - BK; bkIdx += BK) {
28     // double-buffering: load next blocktile into SMEM
29     loadFromGmem<BM, BN, BK, rowStrideA, rowStrideB>(
30         N, K, A + BK, B + BK * N, As + (1 - As_offset) * BM * BK,
31         Bs + (1 - Bs_offset) * BK * BN, innerRowA, innerColA,
32         innerRowB,
33         innerColB, (*backBarrierPtr));
34
35     // compute the current blocktile
36     (*frontBarrierPtr).arrive_and_wait();
37     processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER, WSUBM, WSUBN,
38     TM, TN>(
39         regM, regN, threadResults, As + As_offset * BM * BK,
40         Bs + Bs_offset * BK * BN, warpRow, warpCol, threadRowInWarp,
41         threadColInWarp);
42     A += BK;      // move BK columns to right
43     B += BK * N; // move BK rows down
44
45     As_offset = 1 - As_offset;
46     Bs_offset = 1 - Bs_offset;
47     // swap the front and back barriers
48     auto tmp = frontBarrierPtr;
49     frontBarrierPtr = backBarrierPtr;
50     backBarrierPtr = tmp;
51
52     __syncthreads();
53 }
54 // 相当于交替加载、处理，本质上一次还是加载一块、处理一块。
55
56 // compute the last blocktile
57 (*frontBarrierPtr).arrive_and_wait();
58 processFromSmem<BM, BN, BK, WM, WN, WMITER, WNITER, WSUBM, WSUBN, TM,
59 TN>(
60     regM, regN, threadResults, As + As_offset * BM * BK,
61     Bs + Bs_offset * BK * BN, warpRow, warpCol, threadRowInWarp,
62     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