大家好，这周本来是想写vllm的blockmanager的，结果在整理笔记时，看见之前入门cuda时画的一些手稿，一时手痒将它们整理成这篇文章。**除了图解外，所有代码都配上了非常详细的注释**，希望对于cuda，能和大家一起从入门到不放弃。

【全文目录如下】

一、前置阅读

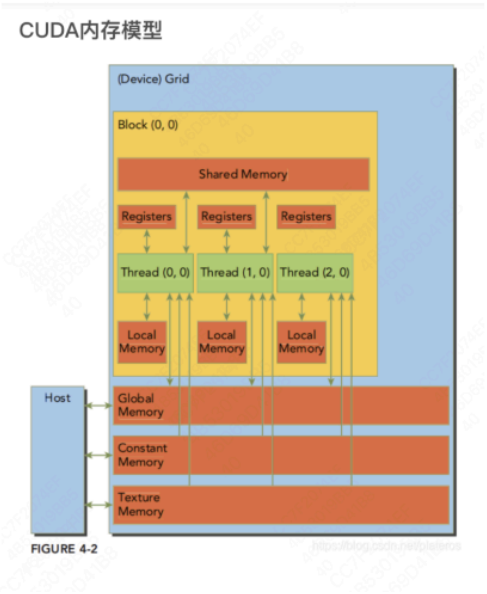
如果你对cuda和gpu架构比较陌生的话，推荐先阅读这篇文章：https://zhuanlan.zhihu.com/p/34587739，特别关注文章中对grid，block，warp，thread的描述。

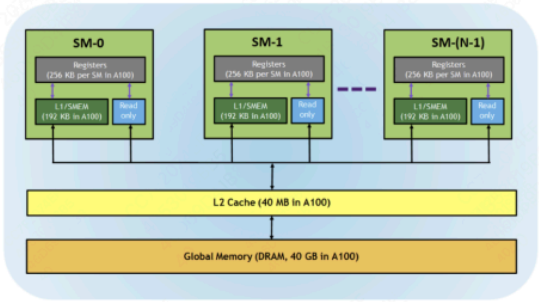
1.1 cuda与gpu

GPU存储可分为**物理内存（硬件真实存在的）**和**逻辑内存（由cuda做抽象的）**。

为什么要这么分呢？因为各个GPU的物理内存架构可能有所不同，如果你写代码时还要考虑每个GPU的独特性，那可太痛苦了。所以cuda在这里帮了大忙：它对内存架构做了一层抽象，你只要按照它抽象后的框架写代码就可以。实际计算时，再由cuda在背后帮你在物理内存上读/写数据。

下图左侧为逻辑内存，右侧为物理内存。





我们先快速过一下右图中的物理内存结构：

Registers：寄存器

L1 cache/Shared memory(SMEM)：L1缓存/共享内存。每个SM独有，不同SM间不能相互访问。其中shared memory可以由用户写代码进行数据的读写控制，L1则不行。

Read-only cache：只读缓存

L2 cache：L2缓存，所有SM都可以访问

Global memory：全局内存。

忽略read-only cache，以上物理内存满足：

内存大小：global memory > L2 cache > L1/SMEM > Register

带宽大小：Register > L1/SMEM > L2 cache > global memory

接下来我们来看左图中的逻辑内存结构，并将其与右图的物理内存结构对应：

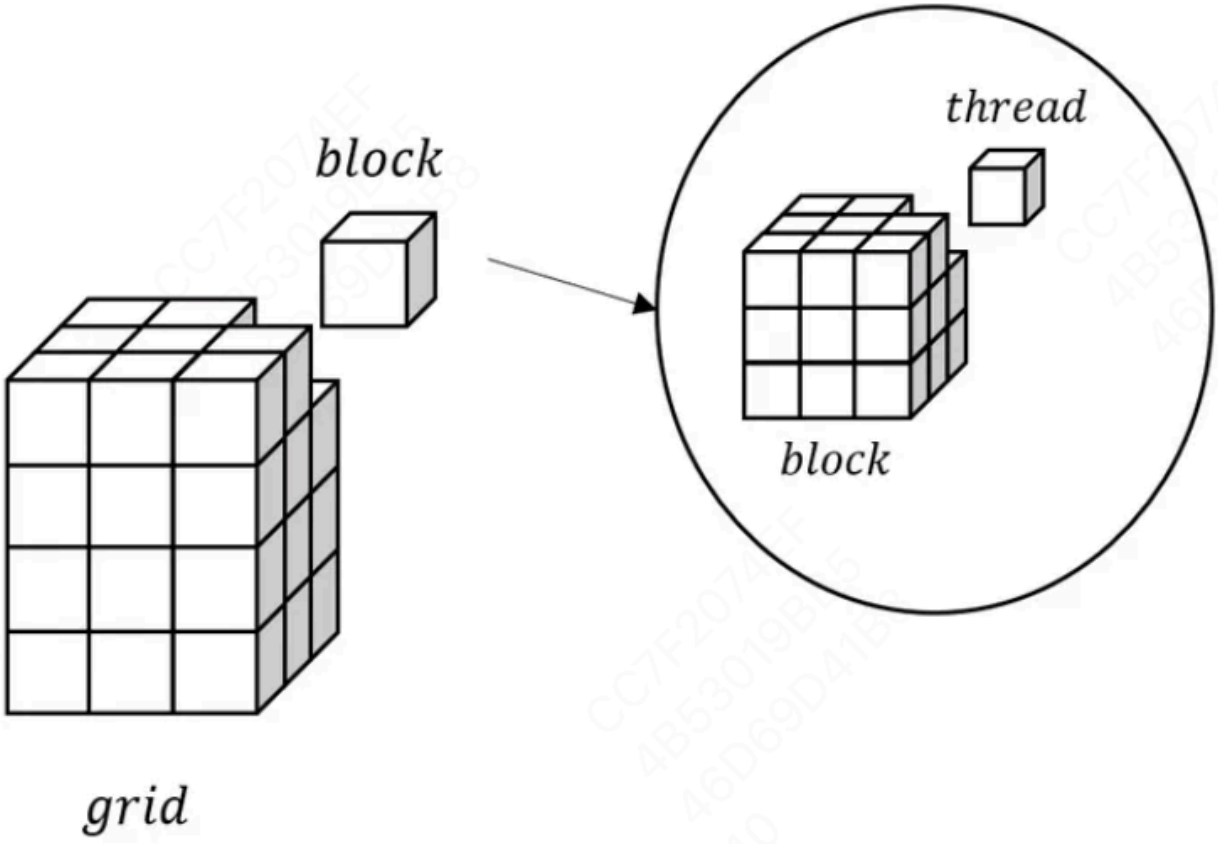
逻辑内存类型	可见范围	生命周期	访问权限	存放数据类型	对应的物理内存

Register（寄存器）	单个thread	所在thread的生命周期	可读可写	在kernel长度中定义的不加任何限定的变量，或者确定长度的数组。
				例如以下代码中的变量row就是一个寄存器变量：
				`__global__ void kernel_func(T* A, T* B, T* C,`
				`  int row = blockIdx.y * 32 + threadIdx.y;`
				`  ...`
				`}`
					On-chip，register

Local memory（局部内存）	单个thread	所在thread的生命周期	可读可写	• 当SM内寄存器使用达到上限，或者在编译时无法确定长度的数组。 • 当寄存器溢出时，编译器可能会自动将一些局部变量放入 Local Memory 中，这是一种编译器级别的优化行为。	off-chip，global memory
Shared memory（共享内存，SMEM）	单个block	所在block的生命周期	可读可写	一个block中所有thread都需要用到的数据	On-chip, l1 cache/SMEM
Global memory（全局内存）	所有thread和host端	由host分配和释放	可读可写	我们常说的“显存”中的大部分就属于全局内存，它为kernel提供计算所需的数据，同时也在host和device，以及device和device间做数据传输	Off-chip, global memory
Constant memory（常量内存）	所有thread和host端	由host分配和释放	只读	定义在kernel函数的外面	• off-chip，本身是存在global memory上 • 但是它需要用on-chip的read-only meomry cache进行读取
Texture memory（纹理内存）	所有thread和host端	由host分配和释放	只读	类似constant memory	• off-chip，本身是存在global memory上 • 但是它需要用on-chip的read-only meomry cache进行读取

1.2 grid, block与thread

一张图总结三者关系：



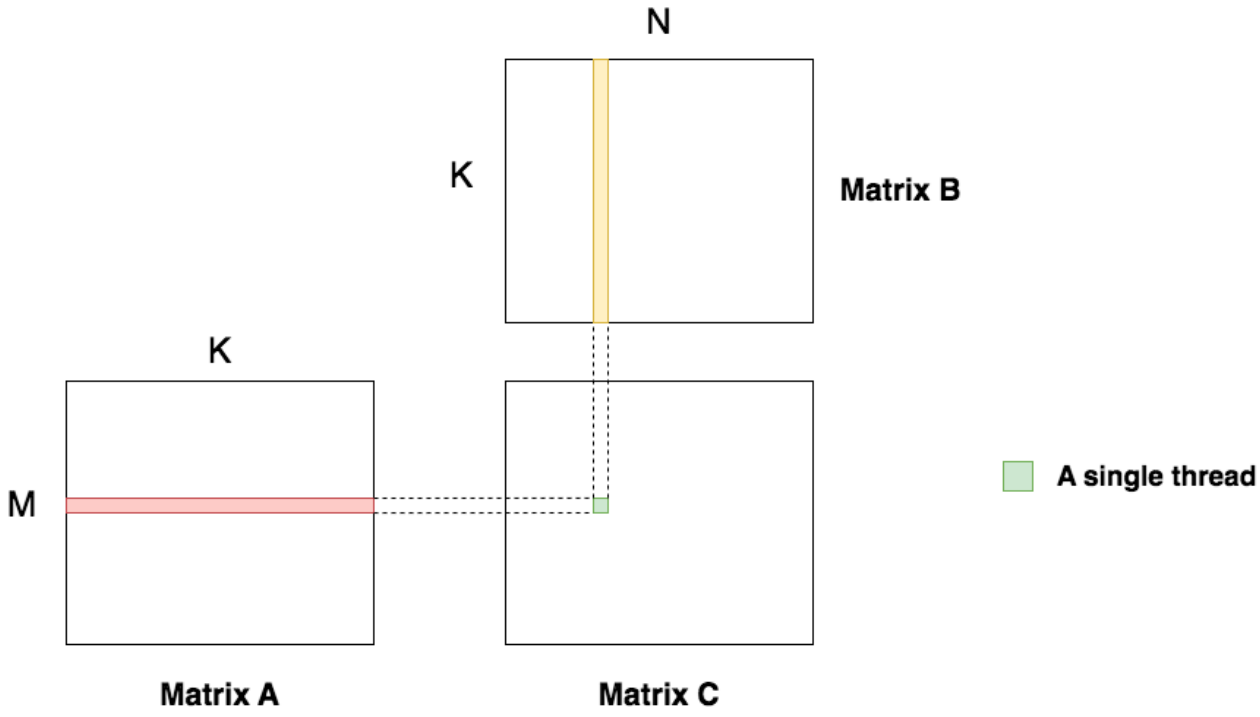
有了这些前置知识，现在我们可以来看cuda矩阵优化的过程了。

假设现在要做的矩阵乘法如下：

$$A = (M, K) = (512, 512)$$
$$B = (K, N) = (512, 512)$$

二、Naive GEMM

Naive GEMM



每个thread负责读取A矩阵的一行和B矩阵的一列，去计算C矩阵的一个元素。则一共需要M*N个thread。

矩阵A和矩阵B都存储在global memory，每个thread直接从global memory上进行读数，完成计算：

为了计算出C中的某个元素，每个thread每次都需要从global memory上读取A矩阵的一行（K个元素），B矩阵的一列（K个元素），则每个thread从global memory上的读取次数为$2K$。

C中共有M*N个thread，则为了计算出C，对global memory的总读取次数为：$2MNK$。

这里及之后的分析中，我们不考虑把结果矩阵C写回global memory需要的次数，只考虑“读”。

Naive GEMM的代码见下（完整代码见 https://github.com/ifromeast/cuda_learning/blob/main/03_gemm/sgemm_naive.cu）：

blockDim：`(32, 32)`，因为一个block内最多1024个thread

gridDim：`(16, 16)`

```
// 将二维数组的行列索引转成一维数组的行列索引，这样可以更高效访问数据
// row, col：二维数组实际的行列索引，ld表示该数组实际的列数
// 例：二维数组实际的行列索引为(1, 3)，即第二行第四个元素，二维数据的总列数 = 5
// 返回的一位数组形式的索引为：1*5 + 3 = 8
#define OFFSET(row, col, ld) ((row) * (ld) + (col))

// 定义naive gemm的kernel函数
__global__ void naiveSgemm(
    float * __restrict__ a, float * __restrict__ b, float * __restrict__ c,
    const int M, const int N, const int K) {

    // 当前thread在C矩阵中的row
    int m = blockIdx.y * blockDim.y + threadIdx.y;
    // 当前thread在C矩阵中的col
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    if (m < M && n < N) {
        float psum = 0.0;
        // 告知编译器自动展开循环体，这样可以减少循环控制的开销（循环次数小的时候可以这么做）
        #pragma unroll
        // 取出A[row]和B[col]，然后逐个元素相乘累加，得到最终结果
        for (int k = 0; k < K; k++) {
            // a[OFFSET(m, k, K)]：获取A[m][k]
            // b[OFFSET(k, n, N)]：获取B[k][n]
            psum += a[OFFSET(m, k, K)] * b[OFFSET(k, n, N)];
        }
        c[OFFSET(m, n, N)] = psum;
    }
}

const int BM = 32, BN = 32;
const int M = 512, N = 512, K = 512;
dim3 blockDim(BN, BM);
dim3 gridDim((N + BN - 1) / BN, (M + BM - 1) / BM);
```

可想而知，由于这种办法要重复从global memory上读取数据，所以读取数据上消耗了大量时间，它肯定没有办法充足利用起GPU的算力。

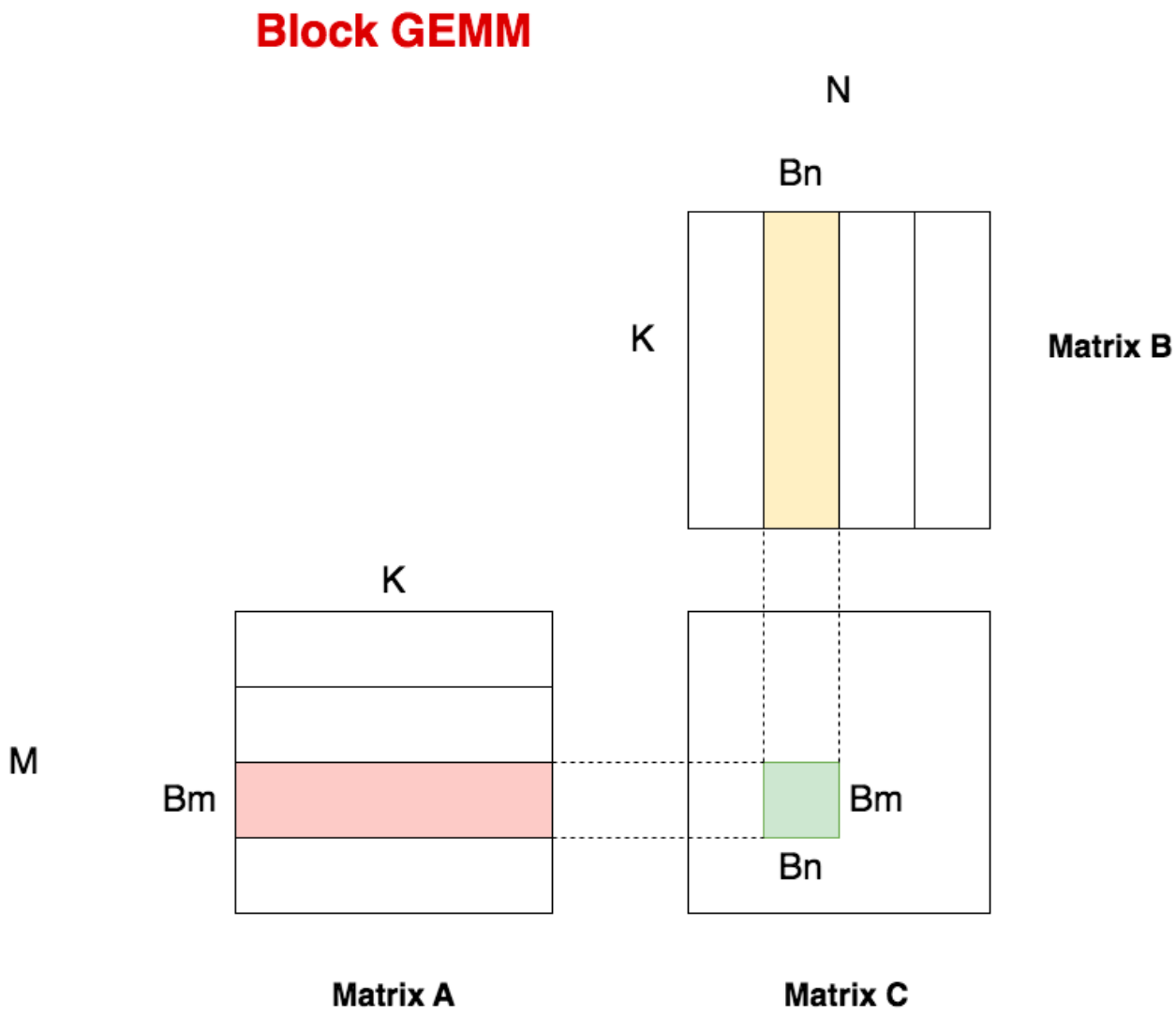
注：在naive gemm的实现中，我们暂不考虑warp级别的调度及合并访存问题，这一点我们放在后文讲解。

三、GEMM优化：矩阵分块，从global memory到SMEM

我们知道on-chip内存的带宽要比off-chip内存的带宽大得多。所以如果我把矩阵A和B都搬运到on-chip的SMEM上，然后采用和naive GEMM一样的计算方法，那么尽管还是会在SMEM上发生重复读数据的情况（也即总的读写次数和naive一样，只不过现在不是从global memory读取，是从SMEM上读取），可是因为带宽变大了，总体来说数据读取时间肯定减少了。

但是问题是，SMEM的存储要比global memory小很多，当矩阵比较大时，根本没办法把完整的矩阵搬运到SMEM上。那该怎么办呢？

很简单，**如果搬运不了完整的矩阵，那我对矩阵切切块，搬运它的一部分，不就行了吗？**



如图:

把A矩阵横着切分成四块，每块大小为$(B_m, K) = (128, 512)$

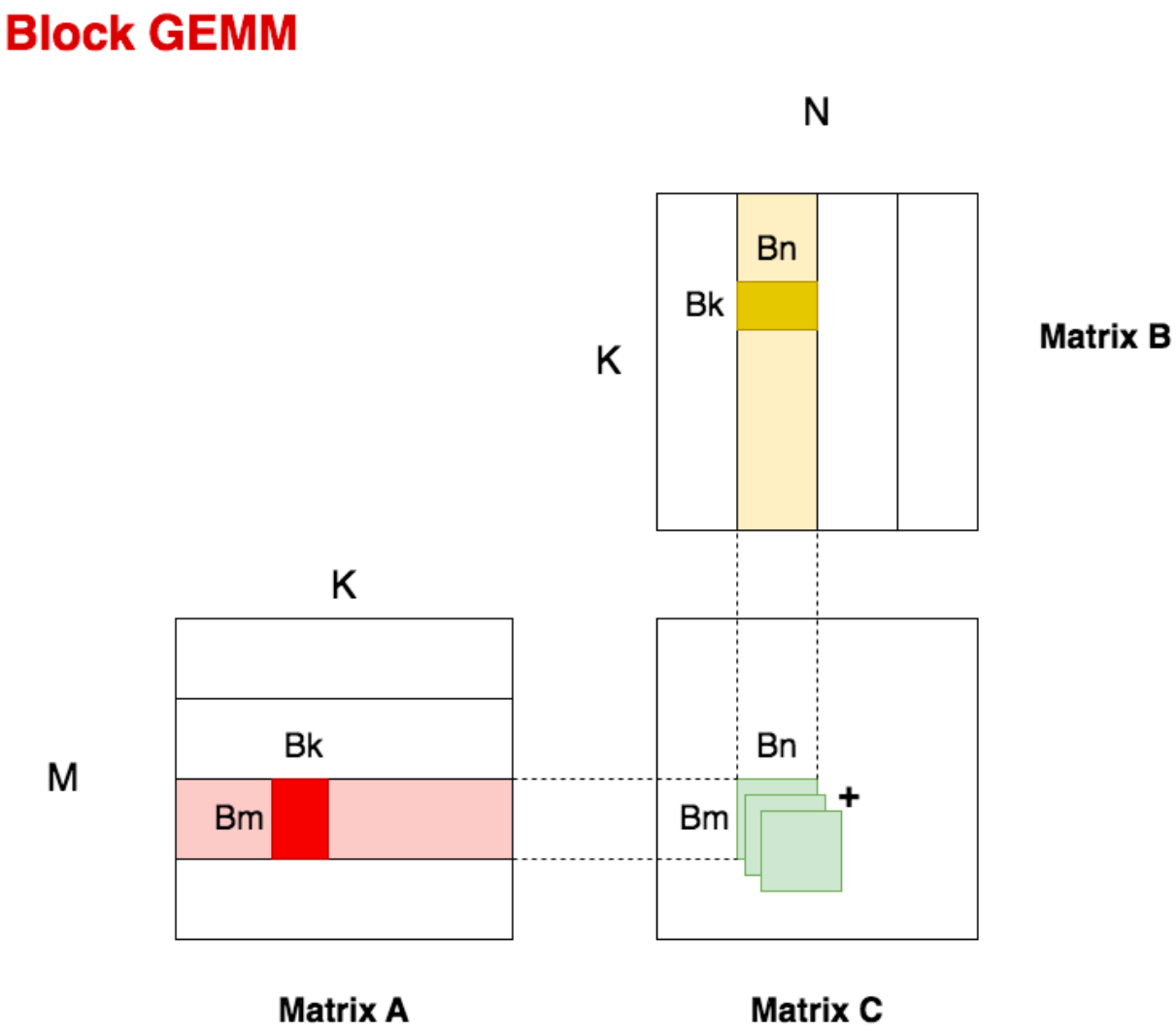
把B矩阵纵着切分成四块，每块大小为$(K, B_n) = (512, 128)$。

A和B对应的切块（如图中的红色和黄色块）组成一个cuda编程里的block，这里我们共有4*4 = 16个block，每个block负责计算C矩阵中大小为(B_m, B_n)的部分（图中绿色块）。易知每个block间的计算是独立的。

好！那么现在我只需要把A的分块（红色）与B的分块（黄色）从global memory搬运到SMEM上，然后再从SMEM做一系列读取操作去计算C。**如此循环，直到所有的C分块都计算出来为止。**这不就能帮我省一笔读取数据的时间么？

这个策略虽然可行，但现在我们再上点难度：如果SMEM还是装不下(B_m, K)，$(K，B_n)$大小的切块，那要怎么办？

那就再继续切呗：



上图中A矩阵的高亮红块，B矩阵中的高亮黄块，就是我们再切割的结果：

A矩阵中的(B_m, B_k)，一般我们取$B_k = 8$，因此最终A切块的大小为$(128, 8)$

B矩阵中的(B_k, B_n)，最终B切块的大小为$(8, 128)$

按照现在的划分，我们再来理一下**一个block**内做的事情：

每次取A矩阵的一个分块(B_m, B_k)，取B矩阵的一个分块(B_k, B_n)，将两者相乘得到分块矩阵C

对A矩阵，向右找到下一个分块；对B矩阵，向下找到下一个分块，然后再相乘得到分块矩阵C，累加到上一个分块矩阵C上。

如此循环，当我们遍历完所有的A分块和B分块后，就可以得到最终的分块矩阵C了。也就是我们图中的高亮绿块(B_m, B_n)。

现在我们来计算下切块方式下对global memory的读取次数：

对于图中一块尺寸为(B_m, B_n)矩阵分块C，每次都要从global memory读取大小为(B_m, B_k)矩阵分块A和大小为(B_k, B_n)矩阵分块B，对global memory的读取次数为$B_m B_k + B_k B_n$。每个block内这样的操作一共要经历$\frac{K}{B_k}$次。

最终每个block在global memory的读取次数为：

block的数量为$\frac{M}{B_M} * \frac{N}{B_n}$

综上两点，**切块方式下对global memory最终的读取次数为**：

所以现在我们有：

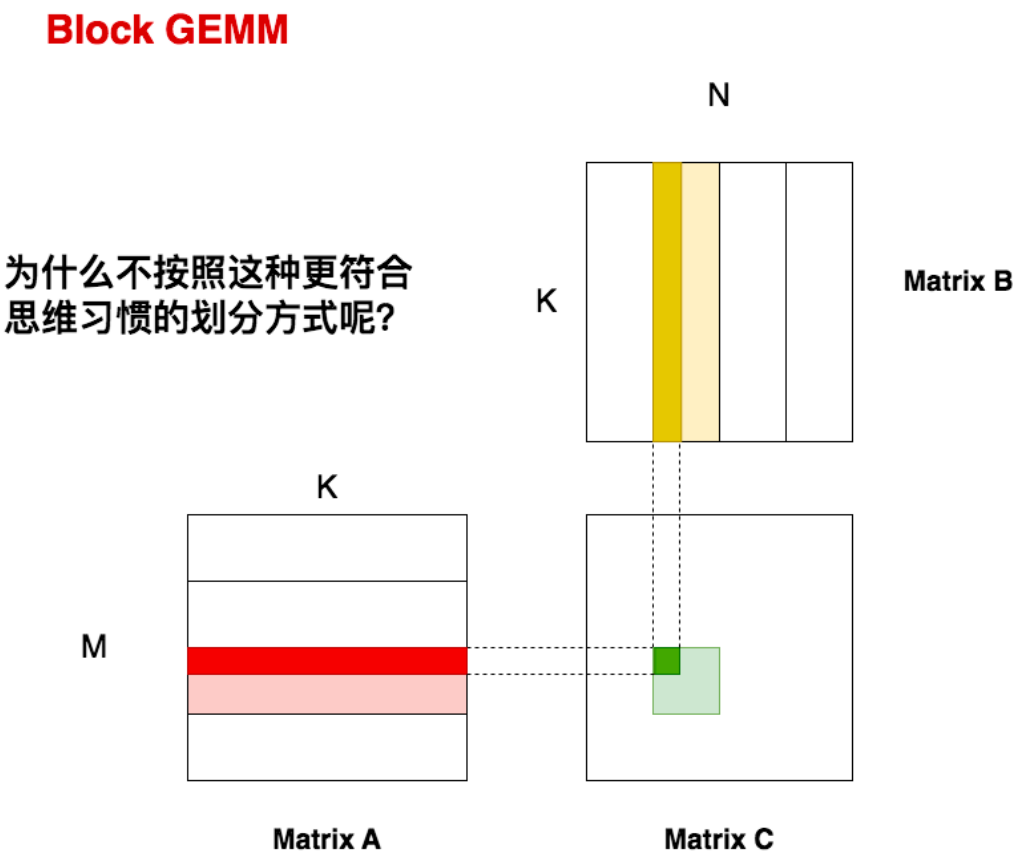
不分块情况（naive gemm）下对global memory的读取次数：$2MNK$

分块情况下对global memory的读取次数：$MNK(\frac{1}{B_m} + \frac{1}{B_n})$

由此可知B_m, B_n越大时，分块情况下对global memory的读写次数越少，使得gpu相对花更多的时间在计算而不是在读数上，更有效利用gpu。但是受到SMEM大小的限制，B_m, B_n也不宜过大，不然一次加载不了那么多数据。

为什么沿着K维度切分（Split-by-k）

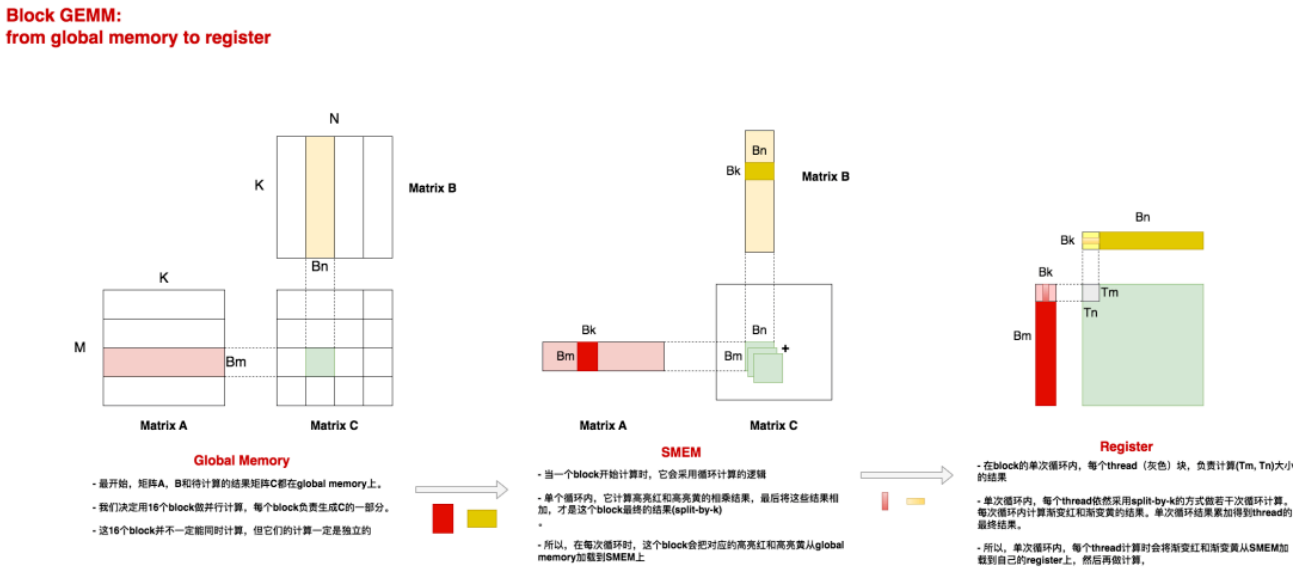
好，现在我们把目光集中到一块block内，你可能想问：为什么我们不按照一种更熟悉的方式，即横着切A，竖着切B，然后再去计算矩阵C呢：



这是因为，如果按照这种方法切块的话，会重复读取数据。例如对于图中的一块A（高亮），它和B中的若干块对应，也就意味着A的这个分块会被重复加载若干次（和naive GEMM是一个道理）。但是如果我们竖着切A，横着切B（此时A和B都是沿着K方向切割的），**这样所有的A分块和B分块都只会被加载1次。可以能帮助我们节省加载数据的时间。**

这个split-by-k的优化很重要，在接下来进一步的矩阵优化中，我们可以发现基本都采用的是这种切割方式。

四、GEMM再优化：从SMEM到register



比对这上面这张图，我们总结下，**到目前为止，我们为了更好利用SMEM，减少从global memory读数据，做了以下事情。**

Global memory

在global memory上，存放着用于计算的矩阵A，B；和结果矩阵C（初始化状态，还没被算出来）

我们不想从低带宽的global memory上一个个读数据，我们想多利用高带宽的SMEM。**因此我们设计了16个可以独立计算的block（绿色），每个block处理一块A（浅红色）与一块B（浅黄色）**。理想情况下，每个block计算时，它会将浅红和浅黄加载到SMEM上，然后做计算。

但是，浅红和浅黄，可能对于SMEM来说还是太大了。所以，我们选择再次切割，每个block做计算时，加载高亮红和高亮黄去SMEM上。

SMEM

单个block在做计算时，会有若干次循环

在每次循环内，block会从global memory上加载一块高亮红和高亮黄到SMEM上（每个thread加载这块高亮红和高亮黄的一部分），然后计算得到单次循环结果。所有循环结果累加，即得到这块block的最终结果（**split-by-k**）

以上两部分是对上文内容的总结，现在我们来看从SMEM -> Register的步骤

Register

单个block做单次循环时，实际负责计算的是它当中的threads，如上图，每个threads负责计算这个block内(Tm, Tn)大小的矩阵。

这个矩阵由上图右侧的浅红色块和浅黄色块加载而来，而这两个色块在SMEM上，也就是thread会从SMEM上逐一取数。

在on-chip的memory上，register是比SMEM带宽更高，存储更小的数据。

所以比起一次次从SMEM上读数，不如类比于global memory -> SMEM的思路，把数据切块后，加载到register中，再做计算。

所以，**单个block的单次循环下，单个thread也存在若干次循环。每次循环内，该thread从SMEM上读取渐变红和渐变黄色块到register，然后再做计算**，thread所有循环的结果相加，即得到该thread的最终结果（**split-by-k**）。

我们马上进入代码实践讲解，在此之前我们先比对上图，把矩阵的各个维度再明确下：

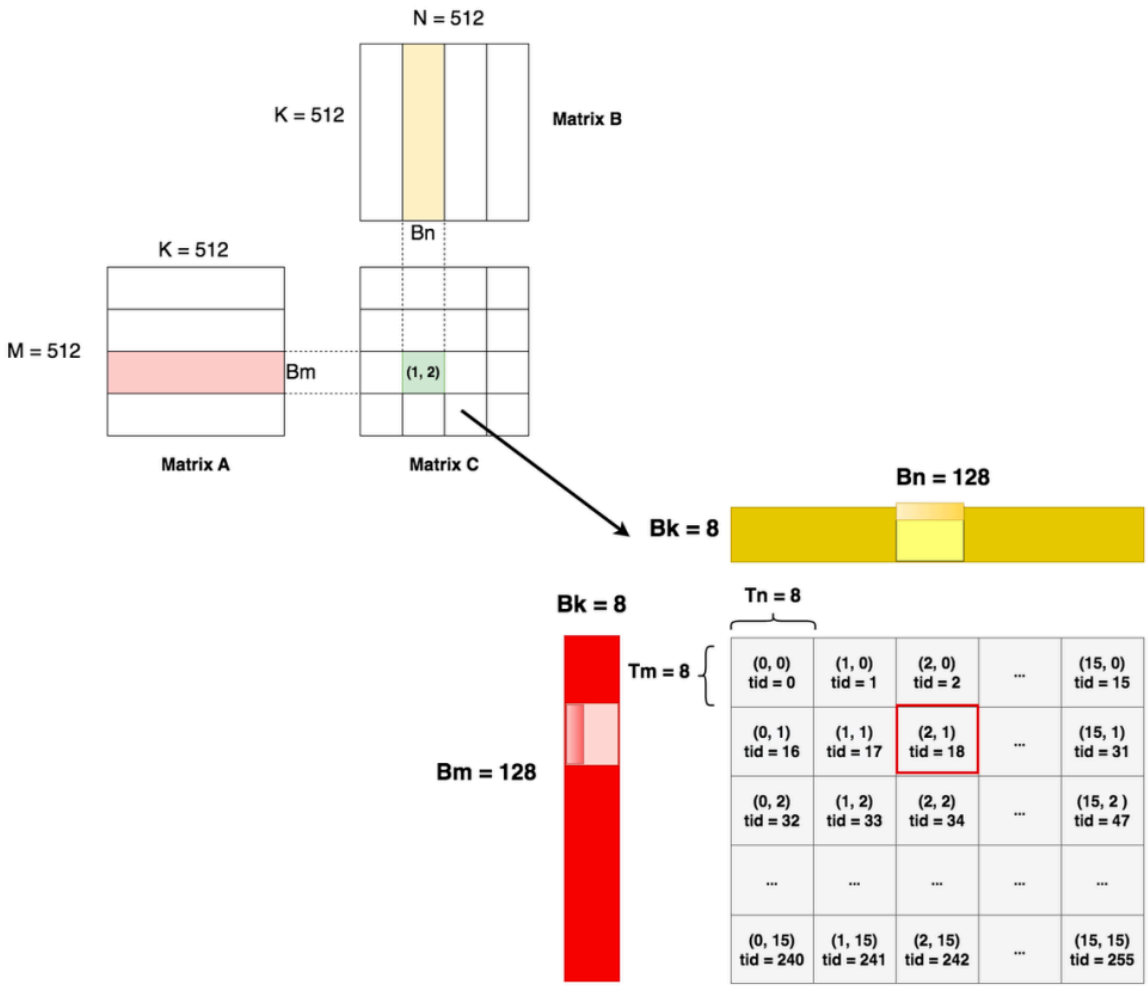
M = N = K = 512

Bm = Bn = 128

Bk = 8

Tm = Tn = 8

在单个block的单次循环内，计算某对高亮红和高亮黄时，block内线程的排布如下：



相关代码如下（附详细注解），在看代码时，大家可以任意带入某个block下的某个thread来看看它是怎么做计算，以及怎么把计算结果写会global memory上C矩阵的对应位置的。

```
__global__ void sgemm_V1(
    float * __restrict__ a, float * __restrict__ b, float * __restrict__ c,
    const int M, const int N, const int K) {

    /*
    在我们的例子里，
    dim3 blockDim(BN/TN, BM/TM) = (16, 16)，即一个block中有256个thread
    dim3 gridDim((N + BN - 1) / BN, (M + BM - 1) / BM) = (4, 4)，即一共16个block
    */
    const int BM = 128;
    const int BN = 128;
    const int BK = 8;
    const int TM = 8;
    const int TN = 8;

    const int bx = blockIdx.x;
    const int by = blockIdx.y;
    const int tx = threadIdx.x; // thread在对应block内的行id
    const int ty = threadIdx.y; // thread在对应block内的列id
    const int tid = ty * blockDim.x + tx; // thread在对应block中的全局id（从左到右，从上到下，从0开始逐一标）

    /*
    在SMEM上对A和B，分别开辟大小为(BM, BK), (BK, BN)的空间
    对应到图例中，s_a为高亮红，s_b为高亮黄
    */
    __shared__ float s_a[BM][BK];
    __shared__ float s_b[BK][BN];

    /*
    初始化当前thread所维护的C矩阵（确定长度的数组，应该是定义在寄存器上的）
    */
    float r_c[TM][TN] = {0.0};

    /*
    例:
    对于tid = 0的thread，以下四个值分别为((0, 0), (0, 0)),
    意味着它负责把s_a(0,0)开始的连续4个数，s_b(0,0)开始的连续4个数，从global memory加载到SMEM

    对于tid = 1的thread，以下四个值分别为((0, 4), (0, 4)),
```

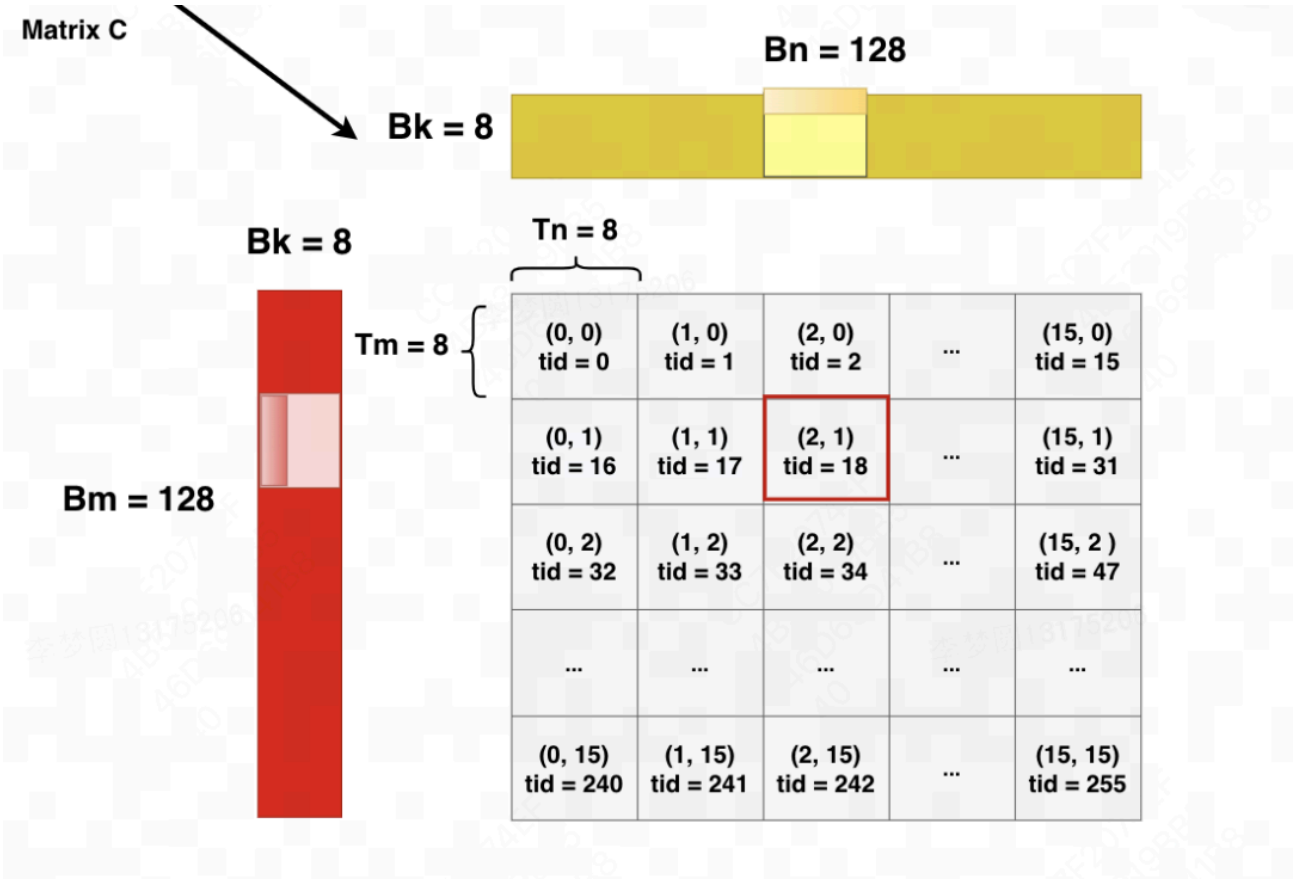
```
意味着它负责把s_a(0,4)开始的连续4个数，s_b(0,4)开始的连续4个数，从global memory加载到SMEM

对于tid = 2的thread，以下四个值分别为((1, 0), (0, 8))
此时s_a第一行的8个数已经被前面的thread取完了，所以现在从s_a第二行开始取，s_b第一行没取完，继续进行

对于tid = 18的thread，以下四个值分别为((9, 0), (0, 72))，含义同上
*/

// 当前thread负责把A中的相关数据从global memory加载到SMEM，
// 这里在计算该thread负责加载的第一个数在s_a中的row
int load_a_smem_m = tid >> 1;  // tid/2, row of s_a
// 当前thread负责加载的第一个数在s_a中的col
int load_a_smem_k = (tid & 1) << 2;  // (tid % 2 == 0) ? 0 : 4, col of s_a

// 当前thread负责把B中的相关数据从global memory加载到SMEM，
// 这里在计算该thread负责加载的第一个数在s_b中的row
int load_b_smem_k = tid >> 5;   // tid/32, row of s_b
// 当前thread负责加载的第一个数在s_b中的col
int load_b_smem_n = (tid & 31) << 2;  // (tid % 32) * 4, col of s_b

/*
例:
对于tid = 0的thread，以下两个值为(256, 128)，
表示该thread从s_a上取的第一个数，其位置在A（位于global memory）上的row 256
该thread从s_b上取的第一个数，其位置在B（位于global memory）上的col 128

对于tid = 18的thread，以下两个值为(265, 200)，道理同上
*/
int load_a_gmem_m = by * BM + load_a_smem_m;  // global row of a
int load_b_gmem_n = bx * BN + load_b_smem_n;  // global col of b

/*
对每个block，它都要经历K/Bk = 128/8 = 16次循环，每次循环计算一块s_a * s_b的结果
这也意味着，对每个block内的每个thread，它的外循环也是16次
*/
for (int bk = 0; bk < (K + BK - 1) / BK; bk++) {
    /*
    1. 在block的单次循环中，需要把对应的s_a（高亮红）和s_b(高亮黄)从global memory
```

```
    加载到SMEM上，因此每个thread负责加载一部分s_a，s_b的数据，最后的__syncthreads()
    是保证thread们在正式计算前，都干完了自己加载的活，即完整的s_a，s_b已被加载到SMEM上
    */
    // 在这次循环中，当前thread从s_a上取的第一个数，其位置在A（位于global memory）上的col，与load_a_gmem_m
    int load_a_gmem_k = bk * BK + load_a_smem_k;   // global col of a
    // 在这次循环中，当前thread从s_a上取的第一个数，在A中的地址，即A[load_a_gmem_m][load_a_gmem_k]
    int load_a_gmem_addr = OFFSET(load_a_gmem_m, load_a_gmem_k, K);
    // 从这个地址开始，取出连续的4个数，将其从A所在的global memory上，加载到s_a上
    // 注：采用FLOAT4的好处是便于连续访存。如果存储的4个数在地址上不连续，你就发4条指令。float4的数据类型就只
    FLOAT4(s_a[load_a_smem_m][load_a_smem_k]) = FLOAT4(a[load_a_gmem_addr]);
    // 在这次循环中，当前thread从s_b上取的第一个数，其位置在B（位于global memory）上的row，与load_b_gmem_n
    int load_b_gmem_k = bk * BK + load_b_smem_k;   // global row of b
    // 在这次循环中，当前thread从s_b上取的第一个数，在B中的地址，即B[load_b_gmem_k][load_b_gmem_n]
    int load_b_gmem_addr = OFFSET(load_b_gmem_k, load_b_gmem_n, N);
    // 同理将相关的数据从global memory加载到SMEM上
    FLOAT4(s_b[load_b_smem_k][load_b_smem_n]) = FLOAT4(b[load_b_gmem_addr]);
    // 在所有thread间做一次同步，保证在下面的计算开始时，s_a，s_b相关的数据已经全部从global memory搬运到SME
    __syncthreads();

    #pragma unroll
    /*
    2. 在block的单次循环中，每个thread采用split-by-k的方式，
    逐步累加计算当前thread所维护的(TM, TN)块的结果
    */
    // 遍历每一个(渐变红，渐变黄)对，可参见图例
    for (int k = 0; k < BK; k++) {
        #pragma unroll
        for (int m = 0; m < TM; m++) {
            #pragma unroll
            for (int n = 0; n < TN; n++) {
                int comp_a_smem_m = ty * TM + m;
                int comp_b_smem_n = tx * TN + n;
                // 每次从SMEM上，各加载渐变红和渐变黄上的1个元素，到register，然后再计算
                r_c[m][n] += s_a[comp_a_smem_m][k] * s_b[k][comp_b_smem_n];
            }
        }
    }
    // 做一次同步，保证所有的thread都计算完当前所维护的（TM，TN）块
    __syncthreads();
}

#pragma unroll
```

```
    /*
    3.
    此时，所有的block已做完循环，
    我们把当前thread计算出的结果（存放在r_c中，尺寸为(Tm, Tn)）写回
    global memory上的C矩阵对应位置中
    */
    // 遍历当前thread结果矩阵的每一行
    for (int i = 0; i < TM; i++) {
        // 计算该thread结果矩阵的这一行，在C矩阵上对应的全局row
        int store_c_gmem_m = by * BM + ty * TM + i;
        #pragma unroll
        // 以4个数为1组，遍历该thread结果矩阵的每一列
        for (int j = 0; j < TN; j += 4) {
            // 计算这4个数中的第一个数在C矩阵上对应的全局col
            int store_c_gmem_n = bx * BN + tx * TN + j;
            // 将这4个数以FLOAT4写回global memory
            int store_c_gmem_addr = OFFSET(store_c_gmem_m, store_c_gmem_n, N);
            FLOAT4(c[store_c_gmem_addr]) = FLOAT4(r_c[i][j]);
        }
    }
}
```

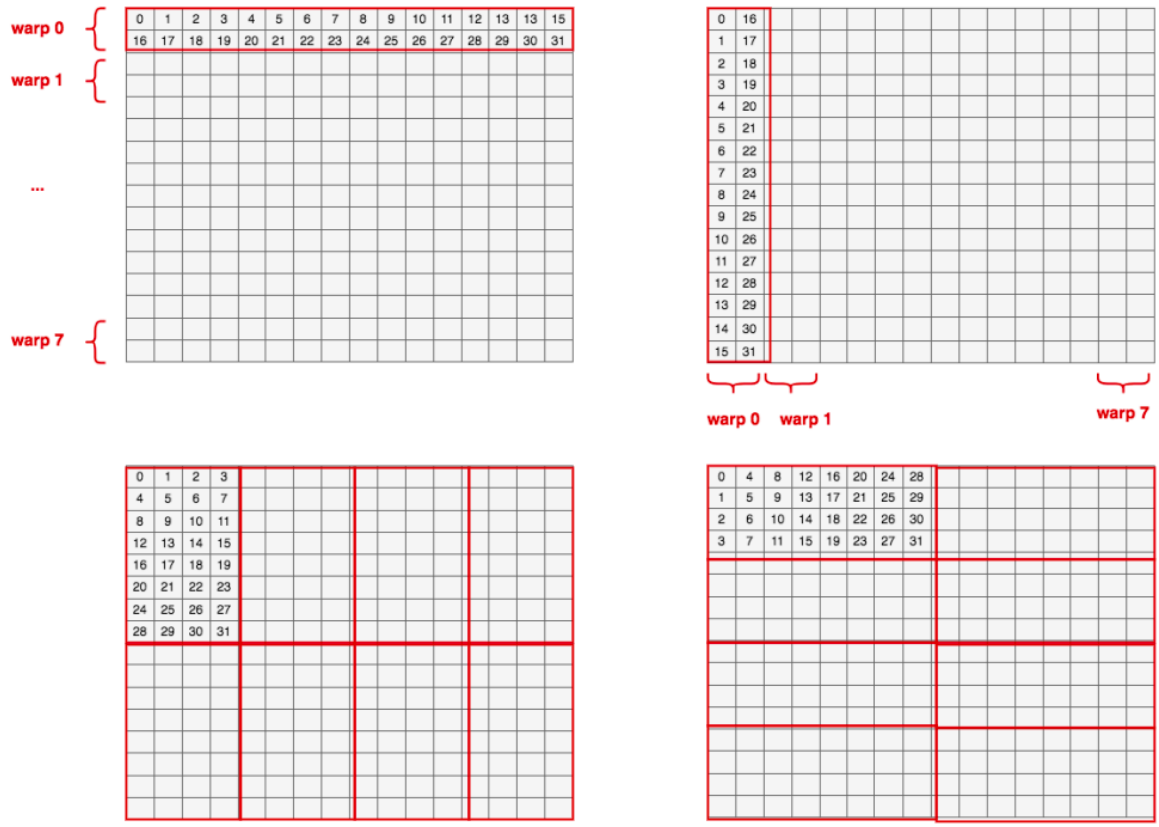
当大家对FLOAT4的用法了解后，就会发现这里还有优化的地方：当某个thread从SMEM上加载数据到register时，它是一个数一个数加载的，这样就需要反射发送多次指令。如果数据是连续存储的，我们完全可以用FLOAT4，一次加载连续的4个数（一共16bytes）的数据去register。别着急，我们接下来就会做这个优化。在此之前，我们先来看一个更为重要的问题。

五、SMEM的bank conflict问题



当你看见这张图的时候，**你可能有疑惑：一个thread的tid一定是像上面那样，从左到右，从上到下排布的吗？答案是否定的**，例如你也可以让第一列的tid是0～15，第二列的tid是16～31，以此类推。只要你能写得出代码，线程的排布可以依你的需要决定。

下图给出了4种不同的线程排布方式（但实际情况中肯定不止这4种），其中左上角的图就对应着我们上面例子中的排布：



以tid = 18的线程为例，当线程排布改变时，这个线程在整个block内负责计算的(Tm, Tn)尺寸的矩阵也会不一样。例如在左上图中，它负责计算block中第二行第三列的(Tm, Tn)矩阵；在右下图中，它负责计算第三行第四列的（Tm, Tn）矩阵。**与之对应的，这个线程读去register上的渐变红和渐变黄块也会不一样。**

到这里我们稍微总结下：

假设现在一个block的尺寸是16 * 16

你可以将左上图（对应着本节最开始的那张图）的构造理解成是这个block内线程的一种形式排布。即线程的二维id，例如(0, 0), (1,0),....(15, 15)等在逻辑上是按照左上图那样排布的。而根据二维id计算出来的一维id（也即tid）也是按左上图那样分布的

一个block内线程实际计算时遵循的排布可认为是一种实际排布，你可以写代码控制它。正如上图所绘，同一个tid在不同排布策略下，负责计算和读取的数据也会有变。

在cuda内部按照tid（其实更准确说应该是线程的二维id）对线程划分warp。即tid = 0 ~ 31为warp0，32 ~ 63为warp1，以此类推。由上图可知，线程排布不同时，warp的形状也会有所不同。**由于warp内固定是32个线程，所以它的形状可能是*216, 162, 48, 84, 132, 321***

”形式排布“和”实际排布“在cuda官方文档中没有理论支持，只是我为了方便理解自己命名的。

看到这里你可能又有一点更深的体会了：**原来不同的线程排布除了影响单个thread的读数和计算，还影响到了warp的组成（也即warp的形状）。**

那么当warp的组成/形状不同时，对我们的计算又有什么影响呢？由前文可知，在SM内，warp是最基本的调度单元。同一warp内的不同线程在计算时，都需要去读取自己所需的数据。在排布合理的情况下，一个warp内的所有线程可以用阻塞最小的方式把自己要的数据从SMEM上取回来，也即尽量减少读数时间。

以上这段描述对你来说可能还有些抽象，**此时你可能迫切想了解两件事：**

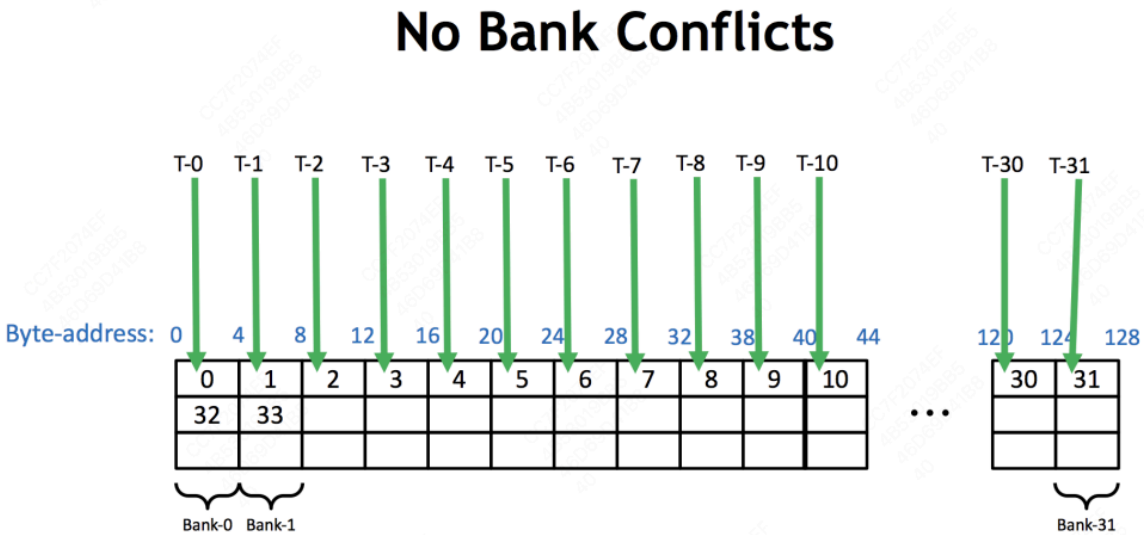
一个warp内的线程们从SMEM上读数时，可能会发生什么样的“阻塞”问题？

线程的排布（也即warp的形状）又是如何避免这种“阻塞”的？

我们依次来对这两个问题做解答。

5.1 不同取数指令下的bank conflict情况

首先我们来回答：当一个warp从SMEM上读取数据时，会发生什么样的“阻塞”。 在Nvidia gpu的SMEM上，数据是被划分为bank存储的，如下图：



SMEM上有32个bank，每个bank存放一个4byte的数。举例来说:

设一个矩阵的形状为(64, 128)，那么当我们把它加载到SMEM上时，对它的第一行，我们先用前32个元素填满上图的第一层banks；再取32个元素填满上图的第二层banks，以此类推直到把这个矩阵的第一行都加载到SMEM上。其余行也是同理。

设一个矩阵的形状为(128, 8)，那么当我们把它加载到SMEM上时，它的前四行就填满上图的第一层banks。以此类推

而这32个bank，正好和一个warp中32个线程的数量对应上，那这意味着什么呢?

（1）LDS.32

假设一个warp现在被调度了，它的32个thread此刻要去SMEM上读数。warp发送了一个LDS.32的指令（意思是让所有的thread都取1个数，其大小为4byte，换算成bit就是32）。此时，在cuda的运算中有如下规定：

一个warp发送1次取数指令（不一定是LDS.32），它最多只能从SMEM上读取128bytes（32个数）的数据。这是每个warp发送1次取数指令的能取到的数据量上限。

如果这个warp发送的是LDS.32指令，意味着它让每个thread都从SMEM上取1个数。

对于这个warp中全部的threads：

如果每个thread要取的数，来自不同的bank，我们就认为没有bank conflict。在没有bank confict的情况下，warp发送1次指令，所有的threads即可取回自己想要的数据。

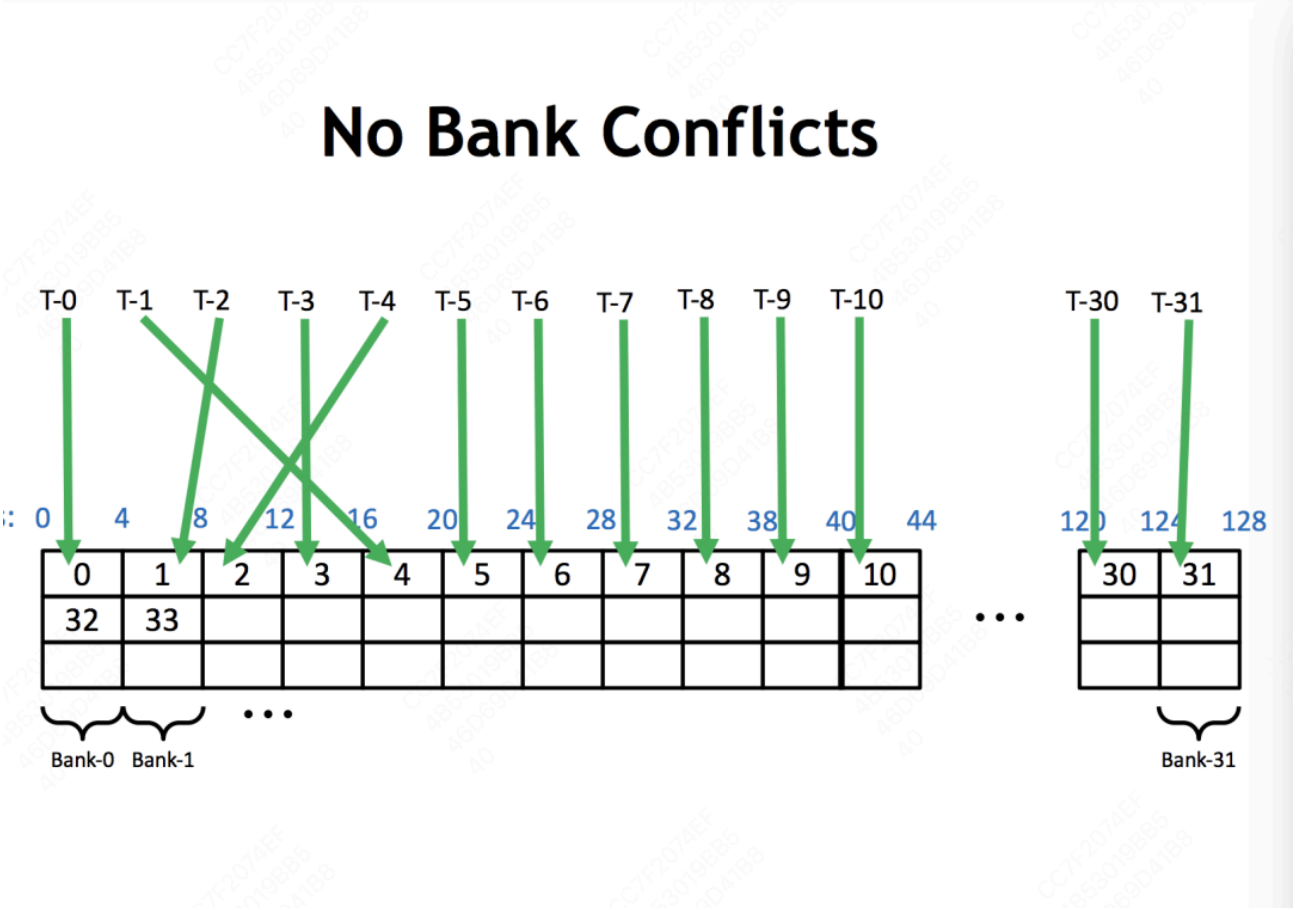
如果某些threads要取的数，来自同一个bank，但他们要取的是这个bank上的同一个数（同一个bank的相同地址），此时我们也认为没有bank conflict，也是1次指令拿回全部的数据

如果某些threads要取的数，来自同一个bank，但它们要取的是这个bank上的不同数（同一个bank的不同地址），此时我们称发生了bank conflict。假设此时对于某个bank，同个warp内不同的若干个threads想要访问它下面n个不同的地址，我们就称这个bank此时发生了**n-way bank conflict**（n头bank conflict）。**那么本来该warp发送1次指令就能取回全部数，现在就需要串行发送n次指令，增加了读取数据的时间**

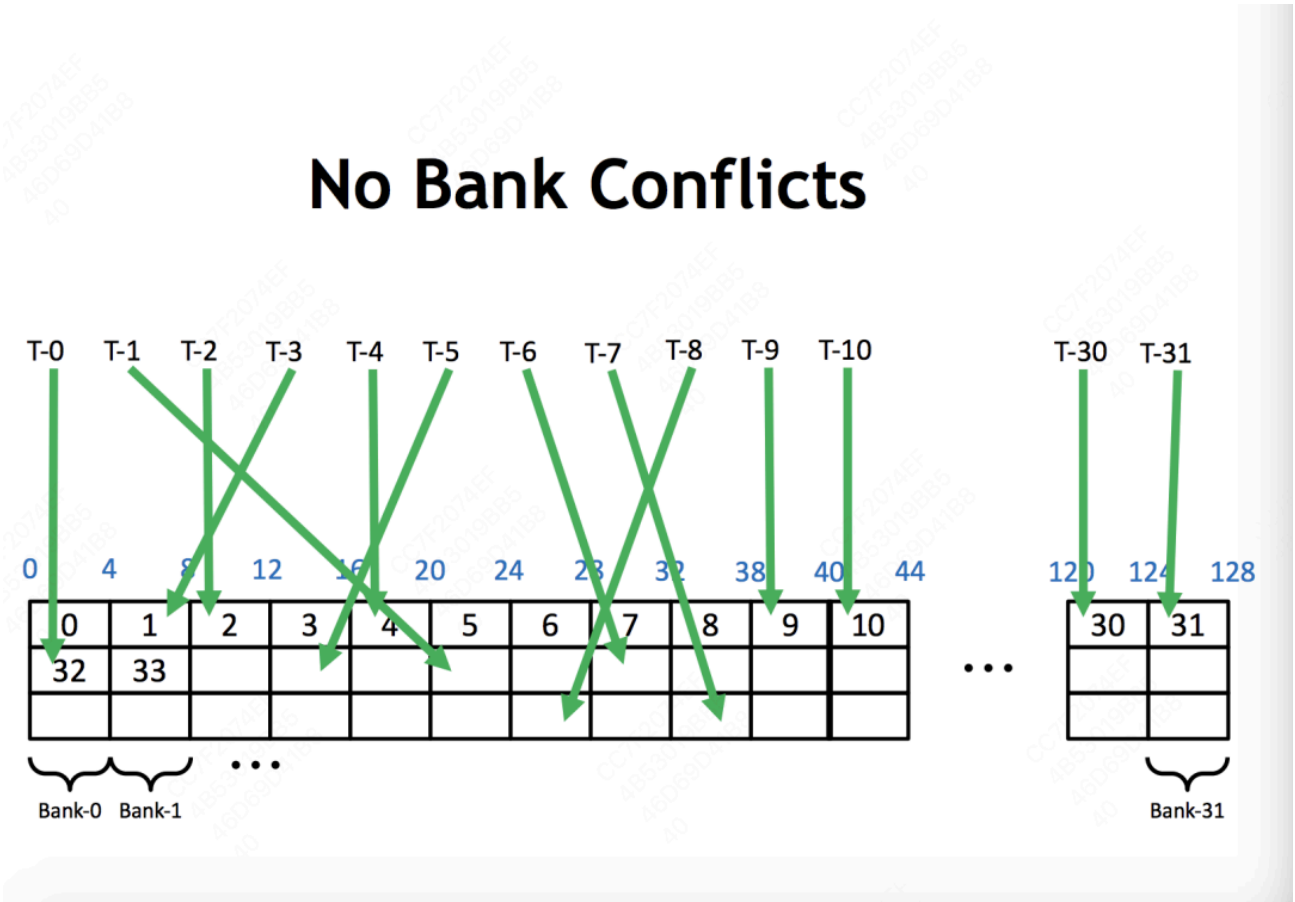
我们更具像化地看几个LDS.32指令下“有bank conflict”和“没有bank conflict”的例子。

例1：即上面那张图，明显没有bank conflict

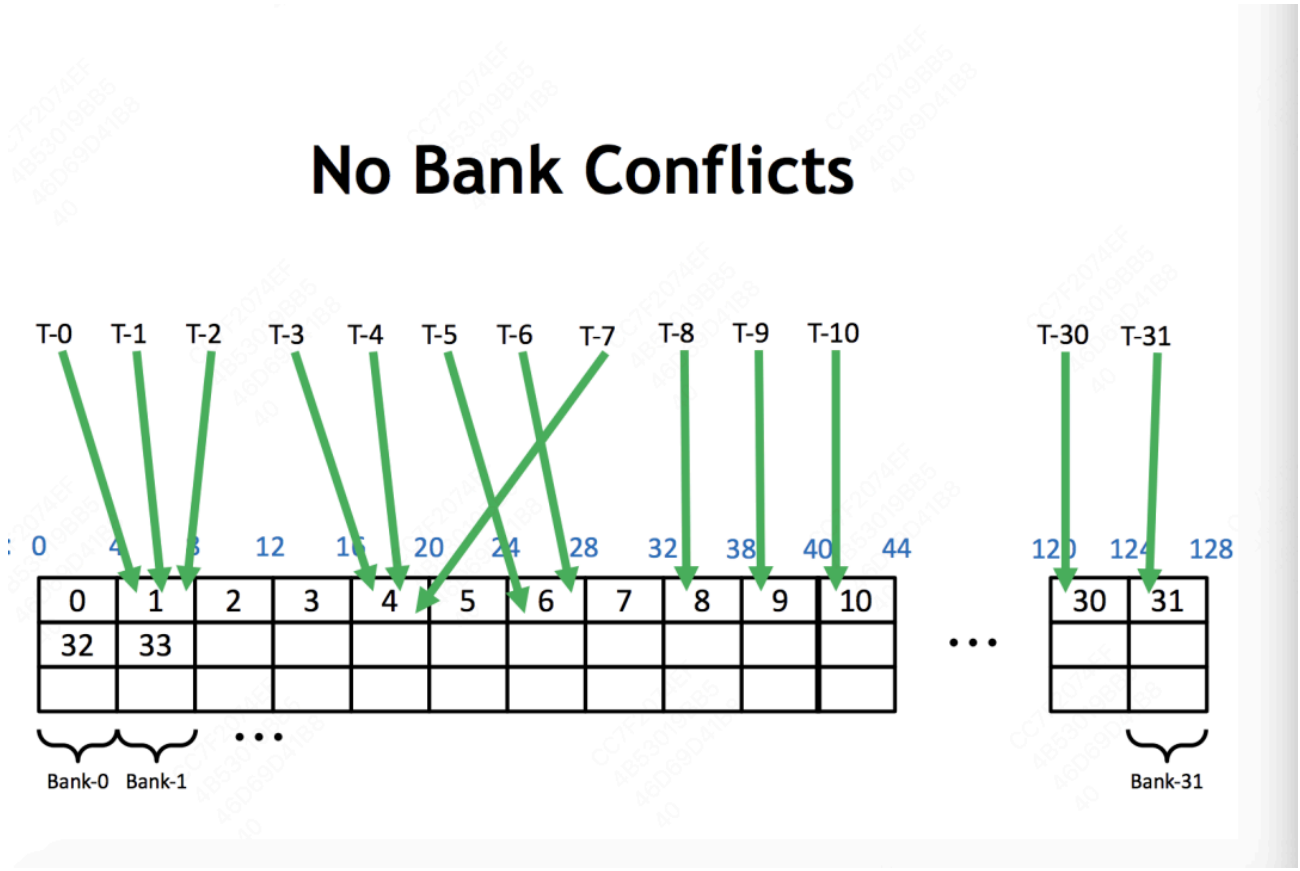
例2: warp内的每个thread访问的也是不同的bank，依不存在bank conflict



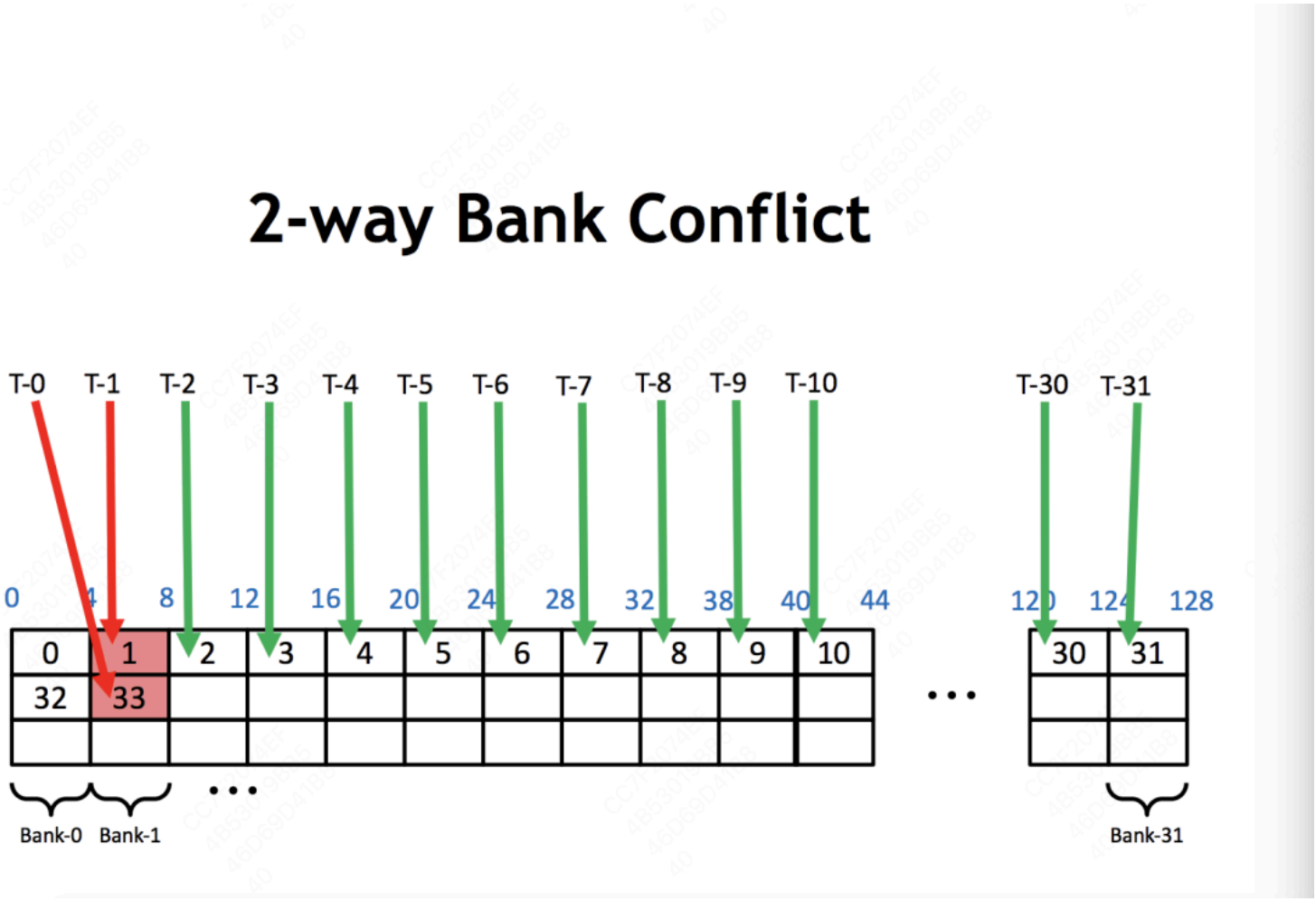
例3: warp内每个thread访问了不同的layer，但是这些thread依然访问的是不同的bank，所以没有bank conflict



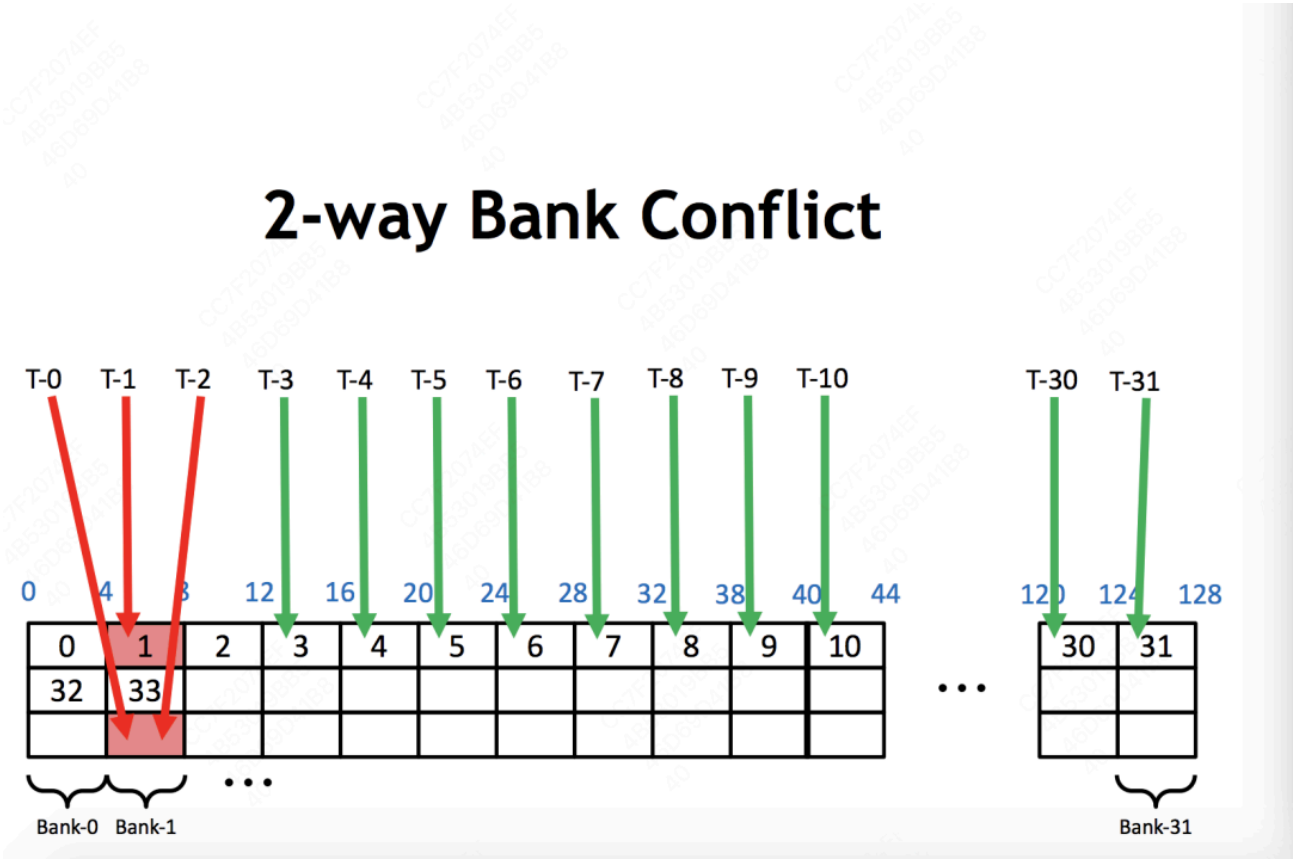
例4：在一个warp内，虽然存在不同的thread访问同一个bank的情况，例如thread0~2都访问了第一个bank。但由于它们访问的是同一个bank中的相同地址，所以此时会**触发广播机制**，即thread0 ~ 2中只有1个thread在取数，取完后它广播给别的threa），也不存在bank conflict。



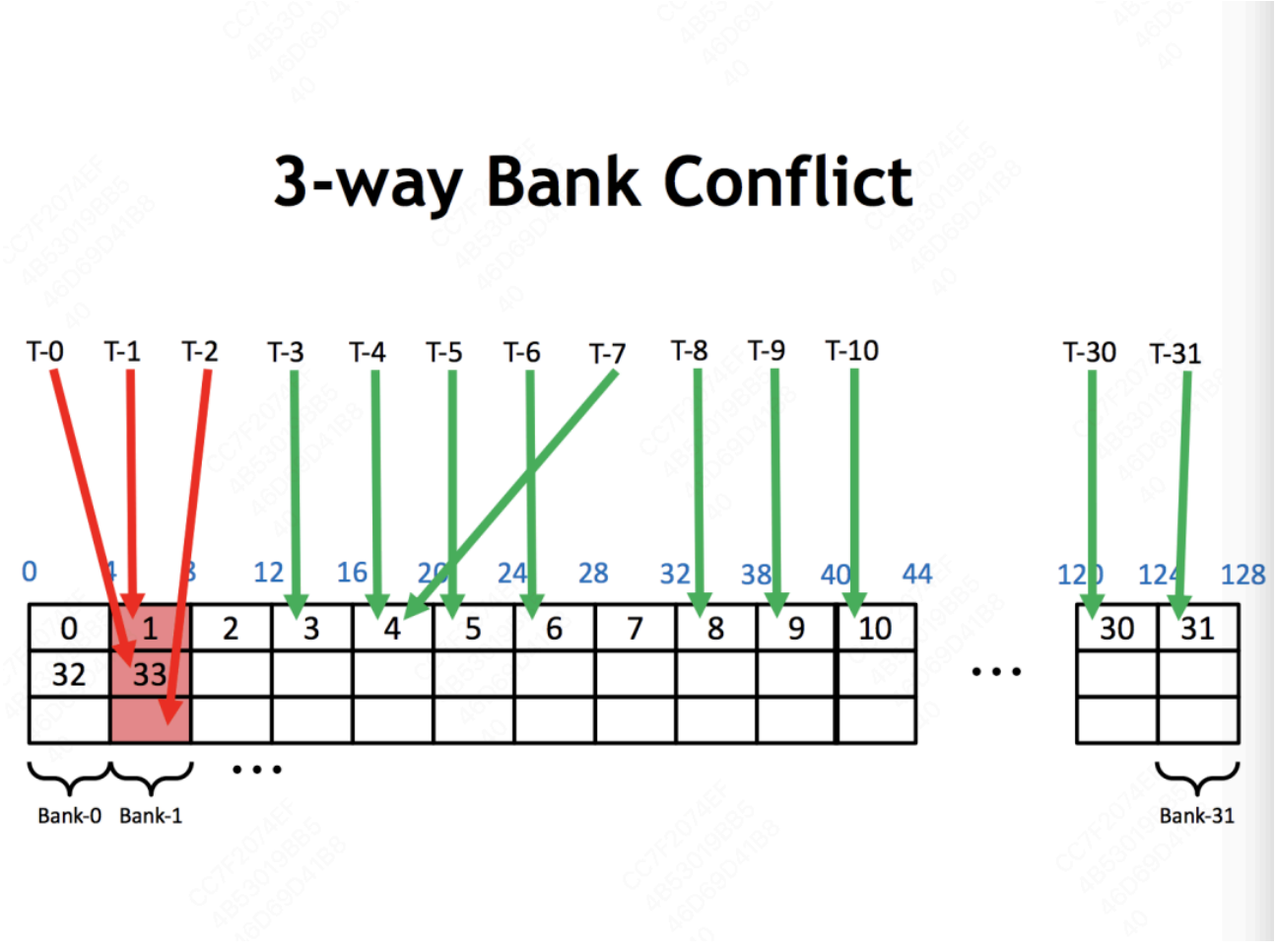
例5: 同个warp内不同thread访问同一个bank的不同地址，此时存在2-way bank conflict，warp需要串行发送两次LDS.32指令才能拿回全部的数据。



例6：同样也是**2-way bank conflict**



例7: **3-way bank conflict**



需要注意的是，bank conflict是针对一个warp内的threads定义的。不同的warp间不存在bank conflict这个概念。

（2）为什么要对SMEM做bank划分

通过第（1）部分的讲解，相信你已经了解了bank和bank conflict的概念，但我猜你一定和当时的我有一样的困惑：

为什么要对SMEM做bank划分？

又为什么要定义出bank conflict这个东西？

又为什么要在bank conflict发生时对warp做惩罚，让它只能串行发送指令？

又为什么bank conflict要定义在一个warp的范围内？

这些问题困扰了我很久，搜索了很久也没找到满意的回答。随着对cuda和gpu认识的加深，**现在我有了一些自己的理解（没有理论资料的支持，只是为了自己能好理解），所以也写在这里作为参考。**

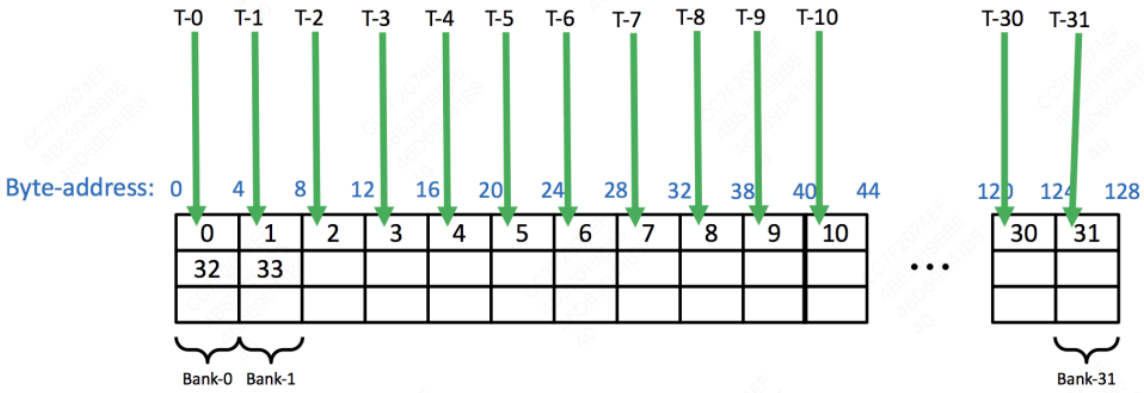
首先，对于SMEM来说，它的某种资源是有限的（例如带宽、或者每次能处理的访存请求数量等），我统一将其称为“资源”

接着，我做了一个（没有理论支持）的预设，即先有了SMEM bank这种硬件（或者说逻辑硬件）结构，然后才有了软件上warp的设计，并令每个warp中线程数量=bank数量=32。

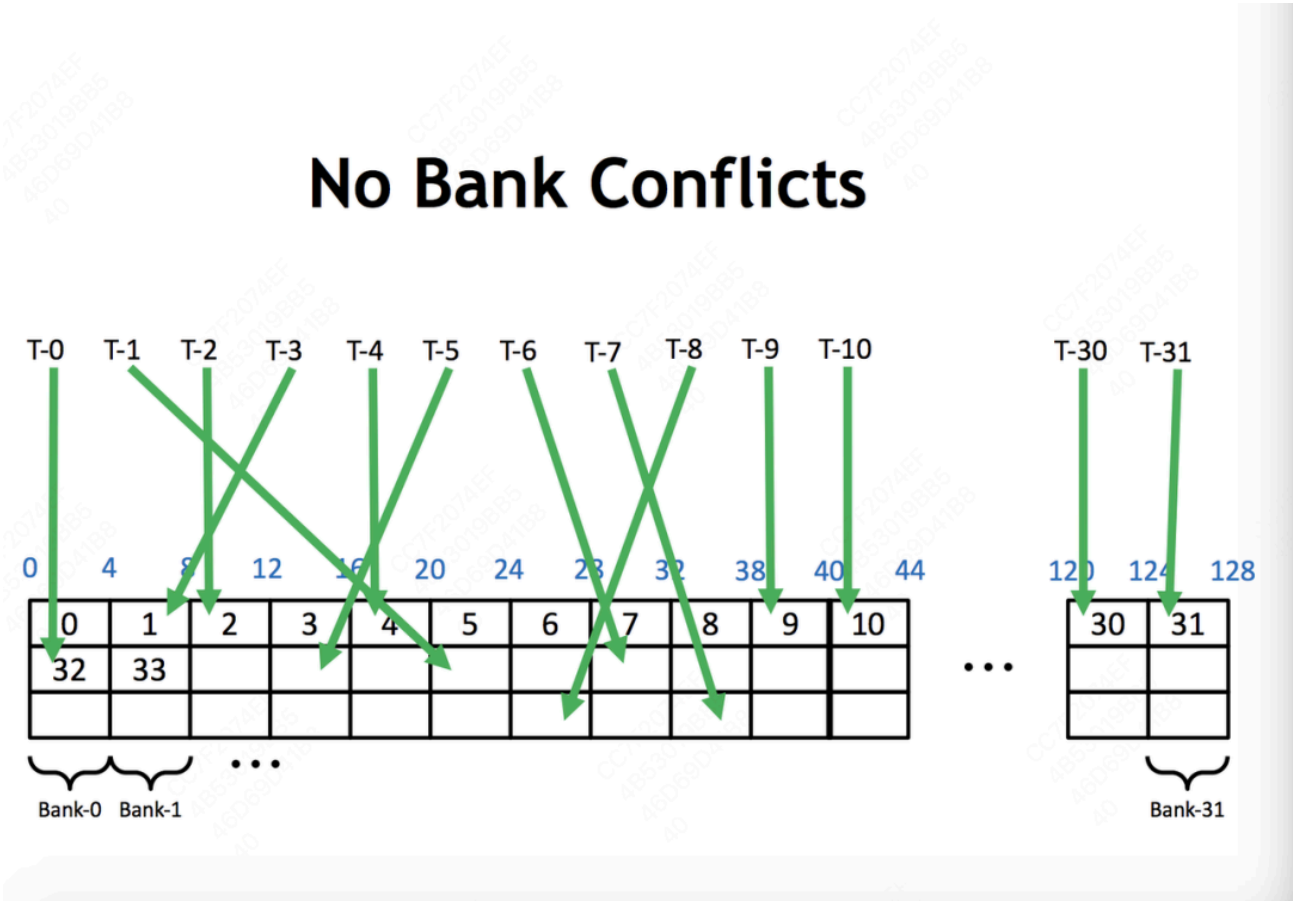
这个预设的含义是，在设计SMEM时，我把资源分配给每个bank。**你可以想象此时每个bank上长出了一条固定宽度的路，它的路宽就是这个bank拥有的资源配额。每当这个bank一个地址上的数据被访问，就占据1单位路宽（即消耗1单位资源配额）。当一个bank的路宽被打满时，它在这个周期内就不允许有新的数据访问了，只能等到下一个周期再处理。**

先有了硬件的假设，我们再来看cuda软件上的实现。假设现在没有warp这个东西，**某一时刻有若干threads都想从SMEM上取数，这时可能会发生它们都集中去某几个banks上取数的情况。**这些banks的路宽都被打满了，threads都在它们上面排队，**而此时其它路却很空。**这样就导致整体并行性低下，取数效率变低。

所以，我们需要一种更均衡的方法管理这些并行的threads，观察SMEM上bank的设计，我们从中映射出了warp的结构，即理想情况下是这样的：



也可能是这样的：

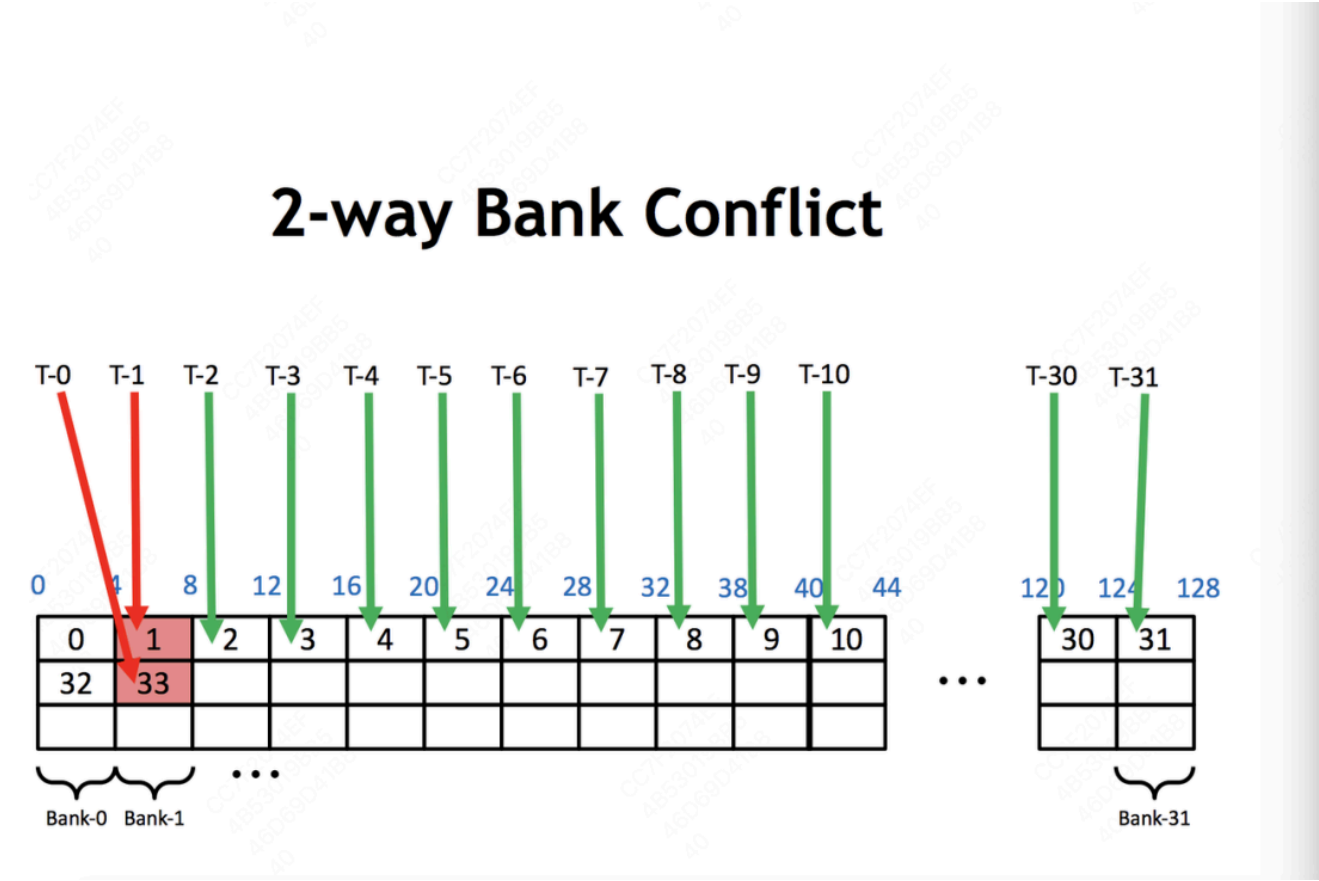


但它们表示的含义都是一样的：

- **为了让你均衡利用banks的路宽，我希望一个warp内的所有threads在banks间均匀分配数据访问请求。**

一个warp最多只允许占用某个bank的1单位路宽。为什么要这样呢？我们来看一个例子。

如图，在这个例子中，某个warp占据了bank1上2个单位路宽。如果此时这个bank的路宽刚好打满，而别的warp也想访问它时，就被阻塞了，别的warp就需要等待取数。**所以这里出现了一个warp阻塞另一个warp的情况。**当这个warp在某个bank上占用的路宽越多，则同一时刻内能并行执行的warp数就越少，**而我们从全局上考虑显然是希望：warp间尽量能够并行，不要相互阻碍。**



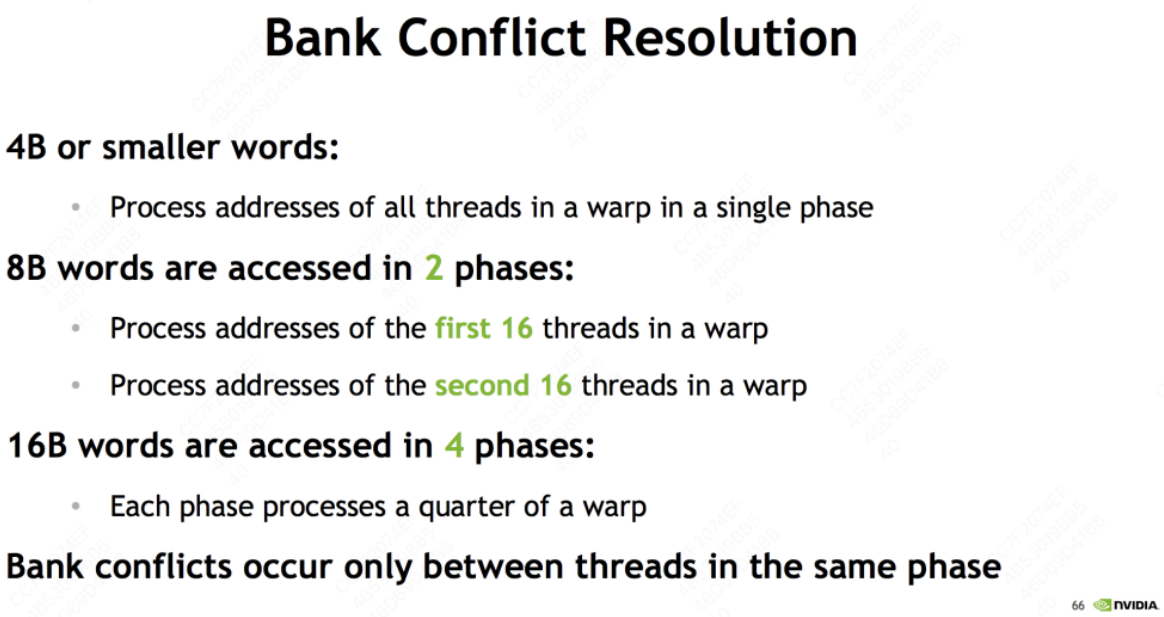
因此我们定下了规则（bank conflict），对阻塞别人的warp进行“惩罚”以保证系统的并行性：如果这个warp下不同thread访问了同一个bank的不同地址时，就需要串行执行这个warp的读取指令。bank conflict源起于硬件层面的资源限制，同时对开发者而言则更像一种惩罚机制，提醒他们在开发时要考虑总体并行能力。

如果明确了这点，就好理解为什么在一个warp内，不同thread访问同一个bank的同一个地址时不会触发bank conflict惩罚了：此时它们读取相同的数，因此我们可以只让一个thread去取数，然后广播给别的thread。**这样这个warp仍只是占用了这个bank的1个单位路宽**，不会影响到别的warp。

（3）LDS.64与LDS.128

上面我们给出的是LDS.32指令下bank conflict的例子。**那如果一个warp发送的是LDS.64指令（一次取8bytes的数，即连续的2个数），或者LDS.128指令（一次取16bytes的数，即连续的4个数）时，bank conflict是怎么样的呢?**

我们直接来看nividia给出回复：



我们来理一下：

一个warp在向SMEM发送一次访存请求(memory transaction)时，它最多只能取128bytes（32个数）的数据。

一个warp在发起memory transaction时，它可以发送不同类型的指令。

warp内的每个thread都会按照这个指令去SMEM上取数，**假设1个数4bytes**，那么：

LDS.32下每个thread去SMEM上取1个数；

LDS.64下每个thread去SMEM上取连续的2个数

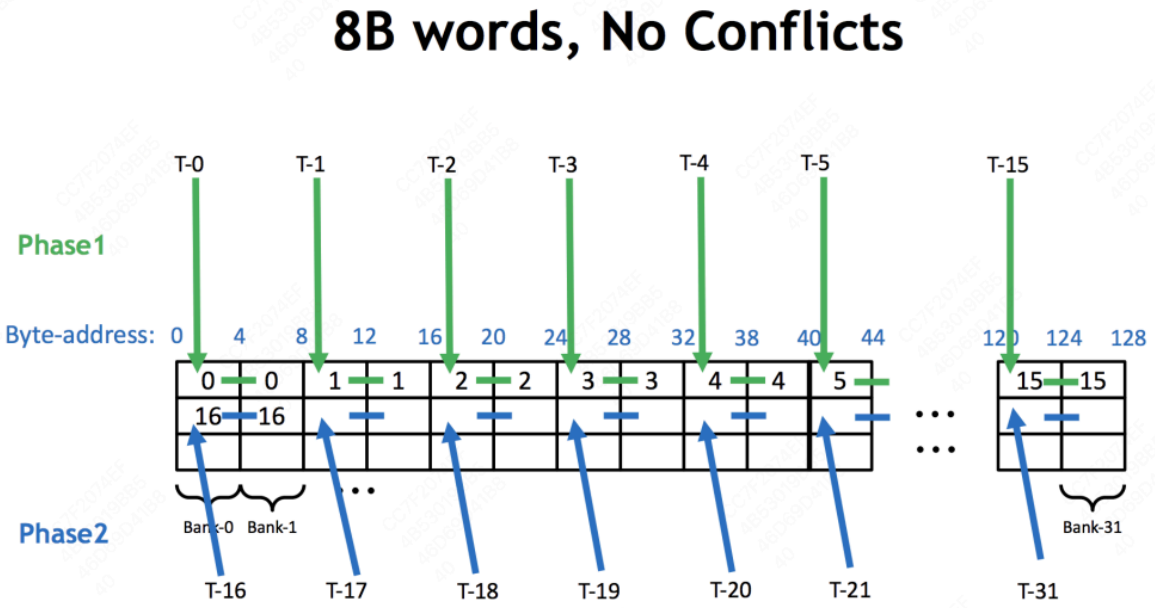
LDS.128下每个thread去SMEM上取连续的4个数。

当你采用LDS.64指令时，一个warp内共需取2*32 = 64个数，已经超过了warp单次memory transaction允许的取数上限。**所以该warp为了取回这64个数，会把取数过程拆成2个串行的phase（即2次串行的memory transaction）**：即0～15号线程先取回32个数，16～31号线程再取回剩下的32个数。**这时bank conflict是被定义在每个phase之内的（也就是1/2个warp之内）**

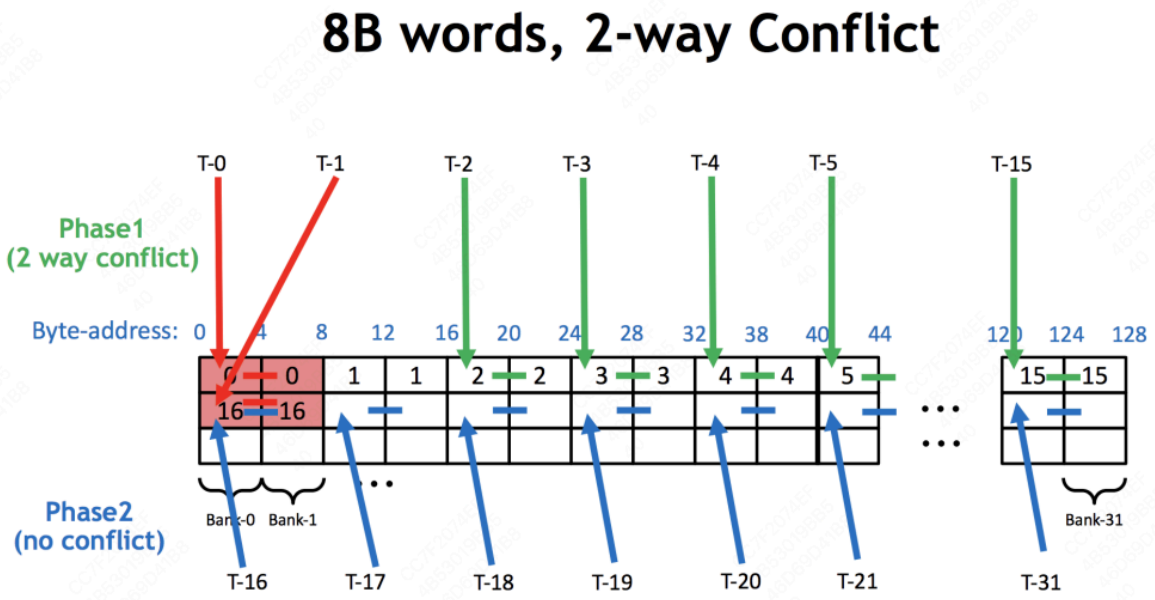
当你采用LDS.128指令时，一个warp共需取4*32 = 128个数，已经超过warp单次memory transaction允许的取数上限。所以**该warp会把取数过程拆成4个串行的phase（即4次串行的memory transcation）**：即0～7，8～15，16～23，24～31。**这时bank conflict被定义在每个phase（也就是1/4个warp之内）。**

来看两个例子，就能理解了。

case1: **使用LDS.64取数，该warp串行发起2次memory transaction，每次1/2个warp的线程在执行取数。所以我们只需关心在1/2个warp内是否发生bank conflict即可**



case2: **使用LDS.64取数，理想情况下应该如case1，每个1/2warp内都没有bank conflict，这样2次memory transaction就能取回数据。**但在下图这个case里，**在第一个1/2warp（线程0～15）**，t0和t1都访问了bank0和bank1上的不同地址，所以**发生bank conflict**，这样**第一个1/2warp就需要发起2次memory transaction**取回全部的数。而第二个1/2warp（线程16～31）则没有bank conflict，只需发起1次memory transaction。所以**共计发起3次memory transaction。**

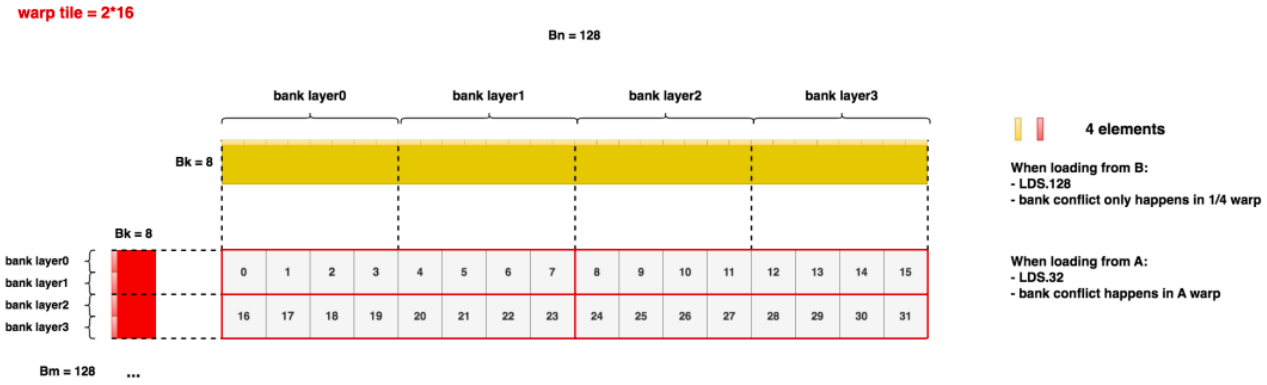


5.2 不同的warp tiling方式对bank conflict的影响

我们在前文说过，不同的线程排布方式（影响warp的形状），可能会引起SMEM上bank conflict的问题，现在我们就通过例子来仔细分析。

（1）2*16 warp

我们先看一个更符合我们直觉的例子，即warp的形状为2*16，线程排布如下：



对于矩阵B的这个(8, 128)分块：

当把该分块加载到SMEM上时，它是按bank组织的，每行放32个元素，放满之后另起一行，继续操作。

每次每个thread一共要从B分块上取回连续的8个数（Tn = 8）。由于这8个数在SMEM上的也是连续排布的，所以这个thread可以采用LDS.128指令，分两次取数，每次取4个连续的数回来（我们称这4个连续的数位float4）

指令是由warp统一发起给各个thread执行的，也就意味着warp要发起2次访存请求。

根据前文说的规则，每次请求发起时，该warp分成4个1/4warp执行（0 ~ 7， 8 ~ 15， 16 ~ 23， 24 ~ 31），**每个1/4warp发起1次memory transaction，一共发起4次memory transaction（此时不存在任何bank conflict，是整个warp从SMEM上读取B分块时最理想的情况）**

现在按上图的排布，来分析实际操作时，这个warp从SMEM上读取B分块会发生什么：

现在warp发起LDS.128指令，第1个1/4warp（0 ~ 7）先去执行，它的目标是让每个thread取回属于自己的第一组连续的4个数。

此时，**0&4，1&5，2&6，3&7这几个线程对，访问了同一个bank的不同地址。**以0&4来说，thread0访问bank0 ~ 3的layer0，thread1访问bank0 ~ 3的layer1。**很明显它们发生了bank conflict。所以对于这1/4个warp，理想情况**

下是1次memory transaction，但现在拆成了串行的2次memory transaction。其余的1/4个warp同理

总结起来，当采用上图排布方式时，由于存在bank conflict，memory transaction的次数变多，读取B更慢了。

对矩阵A的这个（128，8）分块：

注意一下A矩阵在SMEM上bank layer的排布方式，由于A也是按行存储的，所以它是前4行的所有元素占据bank layer0，其余以此类推。

一个thread同样要从A上取8个数（Tm = 8）。**但由于A分块在SMEM上排布的方式问题，要取的这8个数在SMEM上是分散存储的。**我们无法向量化取数，所以这里采用LDS.32指令，每次取一个数。

所以此时，我们在一个warp内分析bank conflict。

不难发现，对于0 ~ 15，16 ~ 31，它们要取的数都相同（访问同样的bank的同样地址），所以会触发广播机制，不存在bank conflict。

但是对于0&16，1&17等线程对来说，它们每次都访问了同一个bank的不同地址，此时存在bank conflict，因此每次取数都会拆分成2个memory transaction。

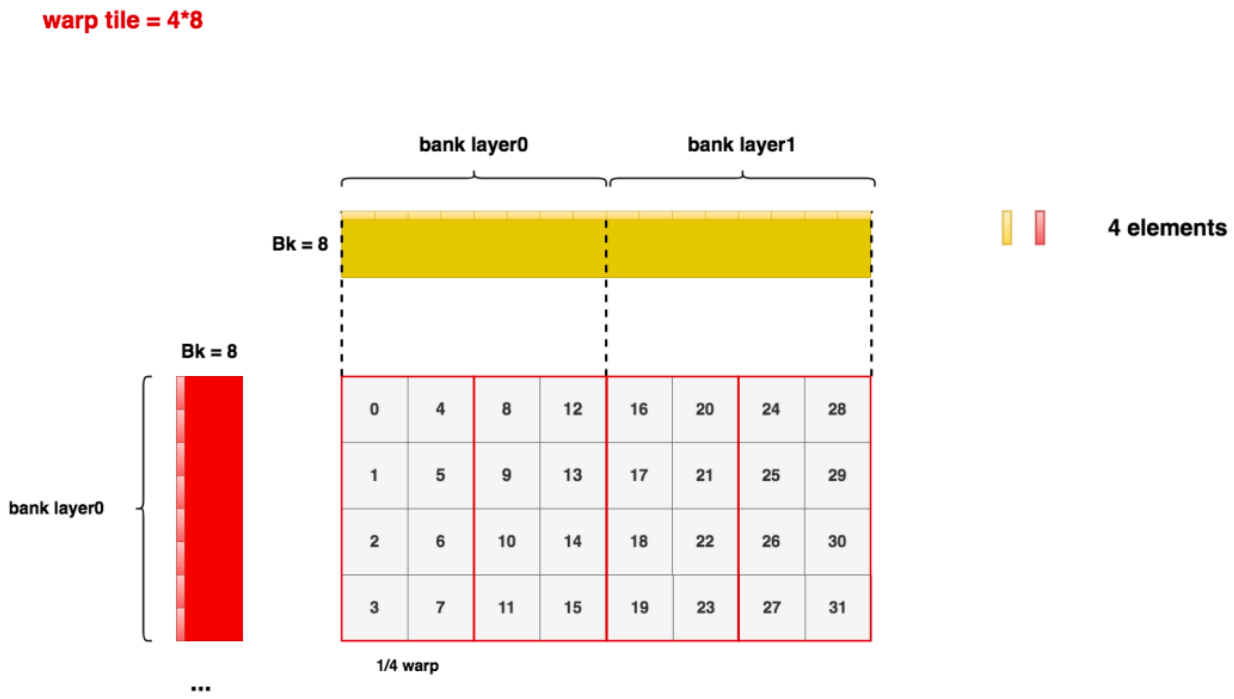
总结起来，当采用上图排布方式时，A方向上同样存在bank conflict，降低取数效率。

解决B方向上的bank conflict说来话长，**但敏感的你一定发现，这种排布下解决A方向上的bank conflict有一种简单的办法：把A转置后再存到SMEM上**，这样我们要取的数（图中细长的渐变红块）在SMEM上就是连续的了，我们可以采用LDS.128进行取数，这样不仅减少指令发射次数，而且1/4warp内也不存在bank conflict（触发了广播机制）。详细的图我就不画了，大家可以类比推理下。

B方向上的bank conflict其实也有很多解决方式，这里我们介绍其中两种思路。

(2) 4*8 warp

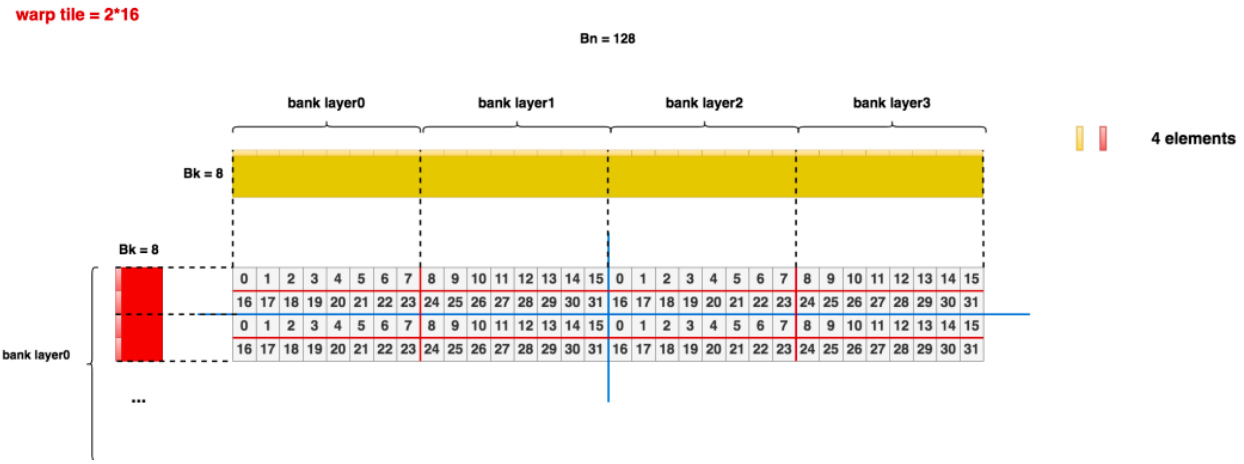
一种简单的办法就是去更改warp的排布（如下图所示），**一会我们给的代码示例就是按照这个排布来做的，大家可以对照着看。**



此时我们已经在SMEM上将A转置，在这种排布下，A和B方向上都不会有bank conflict。具体的分析就不写了，大家可以根据上文的讲解自行推理一下。

（3）将(8, 8)拆成4个(4, 4)

现在，我们再提供解决（1）B方向上bank conflict的另一种办法：



在这张图里，你看到了密密麻麻一堆线程，但是注意，**一个warp内依然只有32个线程，这是不变的。**

这张图的意思是，**原来每个线程算的是一个连续的(8, 8)区域，现在我们把它拆成4个(4, 4)区域**，上图画的就是拆完后每个线程负责计算的区域。

这样拆分后，**每个线程一共还是要读8个数，也还是要使用LDS.128指令读两次（注意这里A已经转置了）。但比起（1），现在在1/4warp内不存在bank conflict的情况了。**例如对于第1个1/4warp（0 ~ 7），它们刚好读满一个bank

layer，其余1/4warp也是同理。

拆分的核心思想是，尽量遵循bank设计的初衷，让不同的线程从一层bank layer上连续读数，而不要错开到不同的bank layer上

(4) 什么样的warp形状更合理

根据前文，一个warp可能长2 * 16，*也可能长4 * 8*，诸如此类，那么我们能办法评估下哪种形状更好吗？

假设这个warp负责计算的矩阵尺寸为(x, y)

那么易推知A上参与这个warp计算的矩阵尺寸为(x, Bk), B上的为(Bk, y)

则总计算次数为：2 * x * y * Bk

总数据读取次数为：x * Bk + Bk * y

则计算访存比 = 以上两者相除 = 2 / (1/x + 1/y)

因此不难知道，当x和y尽量接近时，计算访存比更高，所以一般我们选择4 * 8或者8 * 4这样的warp

(5) 代码实现

最后我们给出一版按5.2（2）排布的代码实现，代码来自：https://github.com/AyakaGEMM/Hands-on-GEMM/blob/main/src/cuda/warp_tile_gemm.cu

```cpp
#include <cstdlib>
#include <cuda_runtime.h>
#include <algorithm>
#include <vector>
#ifndef __CUDACC__
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
void __syncthreads(); // workaround __syncthreads warning
void __syncwarp();
#endif
#include <iostream>
constexpr size_t BLOCK_SIZE = 16; // we assume that every block has equal blockDim.x and blockDim.y
constexpr size_t BLOCK_M = 128;   // These const values decide how many thing a thread compute and the am
constexpr size_t BLOCK_N = 128;
constexpr size_t BLOCK_K = 8; // don't set 64 here, it will cause bank conflict and lower occupancy.
constexpr size_t BLOCK_M_COMPUTE = BLOCK_M / BLOCK_SIZE; // Tm = 8
constexpr size_t BLOCK_N_COMPUTE = BLOCK_N / BLOCK_SIZE; // Tn = 8

// s_a维护的矩阵元素数量
constexpr int shared_memory_A = BLOCK_M * BLOCK_K;
// s_b维护的矩阵元素数量
constexpr int shared_memory_B = BLOCK_N * BLOCK_K;
// s_a + sb维护的矩阵元素数量
constexpr int shared_memory_element = shared_memory_A + shared_memory_B;
// s_a + s_b在SMEM上占据的大小，=它们的矩阵元素总数量 * 单元素大小（4byte）
constexpr int shared_memory_size = shared_memory_element * sizeof(float); // shared memory to use.

// i = 列索引，j = 行索引，想取A[j][i]位置的元素
#define colM(a, i, j, lda) a[((j) * (lda)) + (i)]
// i = 行索引，j = 列索引，想取A[i][j]位置的元素
#define rowM(a, i, j, lda) a[(j) + (i) * (lda)]

__global__ void matrixMul(const float *A, const float *B, float *C,
                          int M, int N, int K, float alpha, float beta)
{
    // 该thread所属的block计算出的结果矩阵中的第一个元素，在C矩阵N方向上的偏移量
    // 如图例，对于(1,2)这个block，baseX = 1*16*8 = 128
    const size_t baseX = blockIdx.x * blockDim.x * BLOCK_M_COMPUTE;
    // 该thread所属的block计算出的结果矩阵中的第一个元素，在C矩阵M方向上的偏移量
```

```
// 如图例，对于(1,2)这个block，baseX = 2*16*8 = 256
const size_t baseY = blockIdx.y * blockDim.y * BLOCK_N_COMPUTE;

// (128*8*2)/(16*16)/2 = 4
const int moveNum = shared_memory_element / (BLOCK_SIZE * BLOCK_SIZE) / 2;
// 该thread的tid，如图例，(2,1)这个thread的tid = 18
const size_t baseIdx = threadIdx.y * blockDim.x + threadIdx.x;

// 每个block中维护的线程数量
constexpr size_t threadsNum = BLOCK_SIZE * BLOCK_SIZE;

// 初始化c矩阵，用于存放该thread所维护的(Tm, Tn)区域的计算结果
float c[BLOCK_M_COMPUTE * BLOCK_N_COMPUTE] = {};
// 存放计算结果
float resC[BLOCK_M_COMPUTE * BLOCK_N_COMPUTE] = {};

// 在SMEM上开辟空间存放高亮红块subA，高亮黄块subB(也就是前面说的s_a，s_b)
__shared__ float subA[BLOCK_M * BLOCK_K];
__shared__ float subB[BLOCK_N * BLOCK_K];

// 在寄存器中，为渐变红regA和渐变黄regB开辟了存放空间
float4 regB[BLOCK_M_COMPUTE / 4]; // hopefully, these should reside in register.
float4 regA[BLOCK_M_COMPUTE / 4];

// 该thread所属的block，要取的浅红色块的第一个元素，在矩阵A上的地址
const float *baseA = A + baseY * K;
// 该thread所属的block，要取的浅黄色块的第一个元素，在矩阵B上的地址
const float *baseB = B + baseX;
// N * 2^3
const auto ldb8 = N << 3;

/*
当前thread负责从global memory加载一部分高亮红、一部分高亮黄到SMEM，
因此所有thread一起加载了完整的高亮红(s_a，本代码中也称为subA)，高亮黄(s_b，即subB)到SMEM
加载方式和上例中代码描述的一致，这里不再重复说明

rowA：该thread负责加载的第一个数在s_a中的row
colA：该thread负责加载的第一个数在s_a中的col
rowB：该thread负责加载的第一个数在s_b中的row
colB：该thread负责加载的第一个数在s_b中的col
```

```
*/
int rowA = baseIdx >> 1, rowB = baseIdx >> 5, colA = (baseIdx & 1) << 2, colB = (baseIdx << 2) & 127;

/*
baseIdx即tid
warpId：当前thread所属的warp id。这里0～31为warp0，32～63为warp1，以此类推。例如tid=18的
        线程属于warp0
warpBaseId：即tid%32，即当前thread在所属warp中的相对位置，例如tid=18的线程在warp中的相对位置
            是18。tid = 33的线程在warp中的相对位置是1
*/
int warpId = baseIdx >> 5, warpBaseId = baseIdx & 31;

/*
当前thread计算的(Tm, Tn)区域的第一个元素在其所属的block所维护的那块C矩阵中的位置
例如当前thread的tid = 18，则rowC = 16, colC = 32
*/
int rowC = ((warpId >> 1 << 2) + (warpBaseId & 3)) << 3, colC = (((warpId & 1) << 3) + (warpBaseId >>

/*
该thread计算的(Tm, Tn)区域的第一个元素，对应在完整的C矩阵中的地址
*/
float *baseC = C + (baseY + rowC) * N + baseX + colC;

/*
对每个block，它都要经历K/Bk = 128/8 = 16次循环，每次循环计算一块s_a * s_b的结果
这也意味着，对每个block内的每个thread，它的外循环也是16次
*/
for (int i = 0; i < K; i += BLOCK_K)
{
    /*
    1. 在block的单次循环中，需要把对应的s_a（高亮红）和s_b(高亮黄)从global memory
    加载到SMEM上，因此每个thread负责加载一部分s_a，s_b的数据，最后的__syncthreads()
    是保证thread们在正式计算前，都干完了自己加载的活，即完整的s_a，s_b已被加载到SMEM上
    */

    // 加载当前thread所负责加载的s_a上的那4个数
    // 这里是从global memory加载，所以计算的是在A矩阵上的位置
    regA[0] = *reinterpret_cast<const float4 *>(baseA + rowA * K + colA);
    // 加载当前thread所负责加载的s_b上的那4个数
```

```
        regB[0] = *reinterpret_cast<const float4 *>(baseB + rowB * N + colB);

        // 对s_b正常装载4个数
        *reinterpret_cast<float4 *>(&subB[baseIdx * 4]) = regB[0];

        // 对s_a则做了转置，这是为了避免SMEM bank conflict
        subA[rowA + colA * BLOCK_M] = regA[0].x;
        subA[(rowA) + (colA + 1) * BLOCK_M] = regA[0].y;
        subA[(rowA) + (colA + 2) * BLOCK_M] = regA[0].z;
        subA[(rowA) + (colA + 3) * BLOCK_M] = regA[0].w;

        baseA += BLOCK_K;
        baseB += ldb8;
        // 在所有thread间做一次同步，保证在下面的计算开始时，s_a, s_b相关的数据已经全部从global memory搬运到SME
        __syncthreads();
#pragma unroll
        for (int ii = 0; ii < BLOCK_K; ii++)
        {
            // 取出当前thread所要取的第一个float4渐变黄块 (32)
            regB[0] = *reinterpret_cast<float4 *>(&subB[colC + BLOCK_N * ii]);
            // 取出当前thread所要取的第二个float4渐变黄块 (36)
            regB[1] = *reinterpret_cast<float4 *>(&subB[colC + 4 + BLOCK_N * ii]);
            // 取出当前thread所要取的第一个float4渐变红块 (16)
            regA[0] = *reinterpret_cast<float4 *>(&subA[rowC + ii * BLOCK_M]);
            // 取出当前thread所要取的第二个float4渐变黄块 (20)
            regA[1] = *reinterpret_cast<float4 *>(&subA[(rowC + 4) + ii * BLOCK_M]);

#pragma unroll
            // 该thread做循环计算及后续写回global memory操作，不提
            for (int cpi = 0; cpi < BLOCK_M_COMPUTE / 4; cpi++)
            {
#pragma unroll
                for (int cpj = 0; cpj < BLOCK_N_COMPUTE / 4; cpj++)
                {
                    c[cpi * 4 * BLOCK_M_COMPUTE + cpj * 4] += regA[cpi].x * regB[cpj].x;
                    c[cpi * 4 * BLOCK_M_COMPUTE + cpj * 4 + 1] += regA[cpi].x * regB[cpj].y;
                    c[cpi * 4 * BLOCK_M_COMPUTE + cpj * 4 + 2] += regA[cpi].x * regB[cpj].z;
                    c[cpi * 4 * BLOCK_M_COMPUTE + cpj * 4 + 3] += regA[cpi].x * regB[cpj].w;

                    c[(cpi * 4 + 1) * BLOCK_M_COMPUTE + cpj * 4] += regA[cpi].y * regB[cpj].x;
                    c[(cpi * 4 + 1) * BLOCK_M_COMPUTE + cpj * 4 + 1] += regA[cpi].y * regB[cpj].y;
                    c[(cpi * 4 + 1) * BLOCK_M_COMPUTE + cpj * 4 + 2] += regA[cpi].y * regB[cpj].z;
```

```
                    c[(cpi * 4 + 1) * BLOCK_M_COMPUTE + cpj * 4 + 3] += regA[cpi].y * regB[cpj].w;

                    c[(cpi * 4 + 2) * BLOCK_M_COMPUTE + cpj * 4] += regA[cpi].z * regB[cpj].x;
                    c[(cpi * 4 + 2) * BLOCK_M_COMPUTE + cpj * 4 + 1] += regA[cpi].z * regB[cpj].y;
                    c[(cpi * 4 + 2) * BLOCK_M_COMPUTE + cpj * 4 + 2] += regA[cpi].z * regB[cpj].z;
                    c[(cpi * 4 + 2) * BLOCK_M_COMPUTE + cpj * 4 + 3] += regA[cpi].z * regB[cpj].w;

                    c[(cpi * 4 + 3) * BLOCK_M_COMPUTE + cpj * 4] += regA[cpi].w * regB[cpj].x;
                    c[(cpi * 4 + 3) * BLOCK_M_COMPUTE + cpj * 4 + 1] += regA[cpi].w * regB[cpj].y;
                    c[(cpi * 4 + 3) * BLOCK_M_COMPUTE + cpj * 4 + 2] += regA[cpi].w * regB[cpj].z;
                    c[(cpi * 4 + 3) * BLOCK_M_COMPUTE + cpj * 4 + 3] += regA[cpi].w * regB[cpj].w;
                }
            }
        }
        __syncthreads();
    }

#pragma unroll
    for (int i = 0; i < BLOCK_M_COMPUTE; i++)
#pragma unroll
        for (int j = 0; j < BLOCK_N_COMPUTE; j += 4)
        {
            *reinterpret_cast<float4 *>(&regA[0]) = *reinterpret_cast<float4 *>(&baseC[i * N + j]);
            regA[0].x = regA[0].x * beta + alpha * c[i * BLOCK_M_COMPUTE + j];
            regA[0].y = regA[0].y * beta + alpha * c[i * BLOCK_M_COMPUTE + j + 1];
            regA[0].z = regA[0].z * beta + alpha * c[i * BLOCK_M_COMPUTE + j + 2];
            regA[0].w = regA[0].w * beta + alpha * c[i * BLOCK_M_COMPUTE + j + 3];
            *reinterpret_cast<float4 *>(&baseC[i * N + j]) = *reinterpret_cast<float4 *>(&regA[0]);
        }
}

void sgemm(int M, int N, int K, float *a, float *b, float *c, float alpha = 1, float beta = 0)
{
    dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 numBlocks((M + BLOCK_M - 1) / BLOCK_M, (N + BLOCK_N - 1) / BLOCK_N);
#ifdef __CUDACC__ // workaround for stupid vscode intellisense
    matrixMul<<<numBlocks, threadsPerBlock>>>(a, b, c, M, N, K, alpha, beta);
#endif
}
```

注：本文仅列出部分gemm优化办法，旨在帮助大家更加熟悉cuda编程的相关概念。