CUDA（三）：通用矩阵乘法：从入门到熟练

通用矩阵乘法 (General Matrix Multiplication，GEMM) 是各种模型和计算中的核心部分，同时也是评估计算硬件性能 (FLOPS) 的标准技术。本文将通过对 GEMM 的实现和优化，来试图理解高性能计算和软硬件系统。

一、GEMM的基本特征

1.1 GEMM计算过程及复杂度

GEMM 的定义为：

$$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A} \boldsymbol{B} + \beta \boldsymbol{C}\\$$

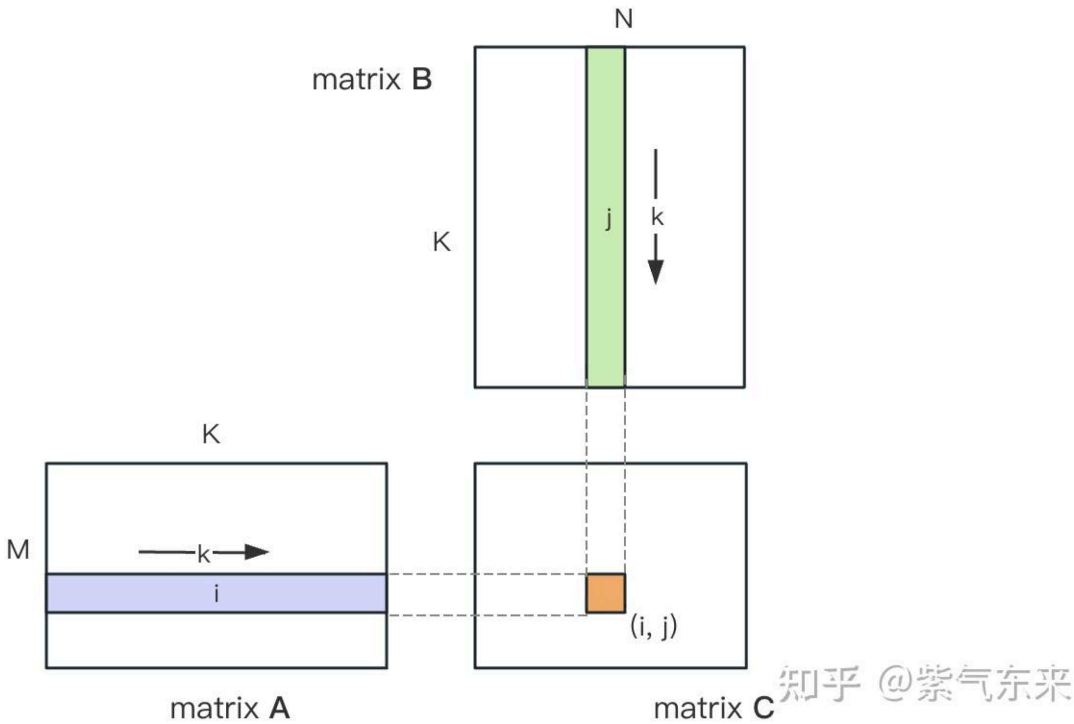
即将矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 进行矩阵相乘，并将结果缩放 α 倍，然后与缩放 β 倍的矩阵 $\boldsymbol{C}$ 相加，并将最终结果存入 $\boldsymbol{C}$ 中。

接下来分析计算复杂度，假设$\boldsymbol{A}$ 的形状是 $M \times K$ ，$\boldsymbol{B}$ 的形状是 $K \times N$ ，则 $\boldsymbol{C}$ 形状是 $M \times N$ 。其中主要的部分是 $\boldsymbol{A} \boldsymbol{B}$ 矩阵相乘，根据矩阵乘法的定义

$$\begin{equation} \boldsymbol{A} \boldsymbol{B}=\left(\begin{array}{ccc} a_{1,1} & \cdots & a_{1, K} \\ \vdots & \ddots & \vdots \\ a_{M, 1} & \cdots & a_{M, K} \end{array}\right)\left(\begin{array}{ccc} b_{1,1} & \cdots & b_{1, N} \\ \vdots & \ddots & \vdots \\ b_{K, 1} & \cdots & b_{K, N} \end{array}\right)=\left(\begin{array}{ccc} \sum_{k=1}^{K} a_{1, k} b_{k, 1} & \cdots & \sum_{k=1}^{K} a_{1, k} b_{k, N} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{N} a_{M, k} b_{k, 1} & \cdots & \sum_{k=1}^{K} a_{M, k} b_{k, N} \end{array}\right) \end{equation} \\$$

其中第 i 行第 j 列元素 $\sum_{k=1}^{K} a_{i, k} b_{k, j}$ ，即每个元素的计算需要 K 次乘法和 K-1 次加法，即计算 $\boldsymbol{A} \boldsymbol{B}$ 共需要执行 (2K-1)MN 次浮点数运算。

另外 $\boldsymbol{A} \boldsymbol{B}$ 和 $\boldsymbol{C}$ 的放缩都需要 MN 次浮点运算，那么总的浮点运算次数则为 $(2K+1)MN$ ，由于 $K\gg 1$ ，因此通常浮点运算次数近似等于 $2KMN$ ，单位为FLOPS(Float Point Operations Per Second)，为便于表示通常使用 GFLOPS ($= 10^9$ FLOPS) 和 TFLOPS ($= 10^{12}$ FLOPS)。



矩阵乘法的计算示意

1.2 简单实现及过程分析

加下来尝试来实现 GEMM，为了便于计算，令 \alpha=1， \beta=0 ， 同时使用单精度(FP32)，即 SGEMM。

下面是按照原始定义实现的 CPU 上实现的代码，之后用以作为精度的对照

```
#define OFFSET(row, col, ld) ((row) * (ld) + (col))

void cpuSgemm(
    float *a, float *b, float *c, const int M, const int N, const int K) {

    for (int m = 0; m < M; m++) {
        for (int n = 0; n < N; n++) {
            float psum = 0.0;
            for (int k = 0; k < K; k++) {
                psum += a[OFFSET(m, k, K)] * b[OFFSET(k, n, N)];
            }
            c[OFFSET(m, n, N)] = psum;
        }
    }
}
```

下面使用CUDA实现最简单的矩阵乘法的Kernal，一共使用 M * N 个线程完成整个矩阵乘法。每个线程负责矩阵 \boldsymbol{C} 中一个元素的计算，需要完成K次乘累加。矩阵\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} 均存放与全局内存中（由修饰符 `__global__` 确定），完整代码见 sgemm_naive.cu 。

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

    int n = blockIdx.x * blockDim.x + threadIdx.x;
    int m = blockIdx.y * blockDim.y + threadIdx.y;
    if (m < M && n < N) {
        float psum = 0.0;
        #pragma unroll
        for (int k = 0; k < K; k++) {
            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);
```

编译完成，在Tesla V100-PCIE-32GB上执行的结果如下，根据V100的白皮书，FP32 的峰值算力为 15.7 TFLOPS，因此该方式算力利用率仅有11.5%。

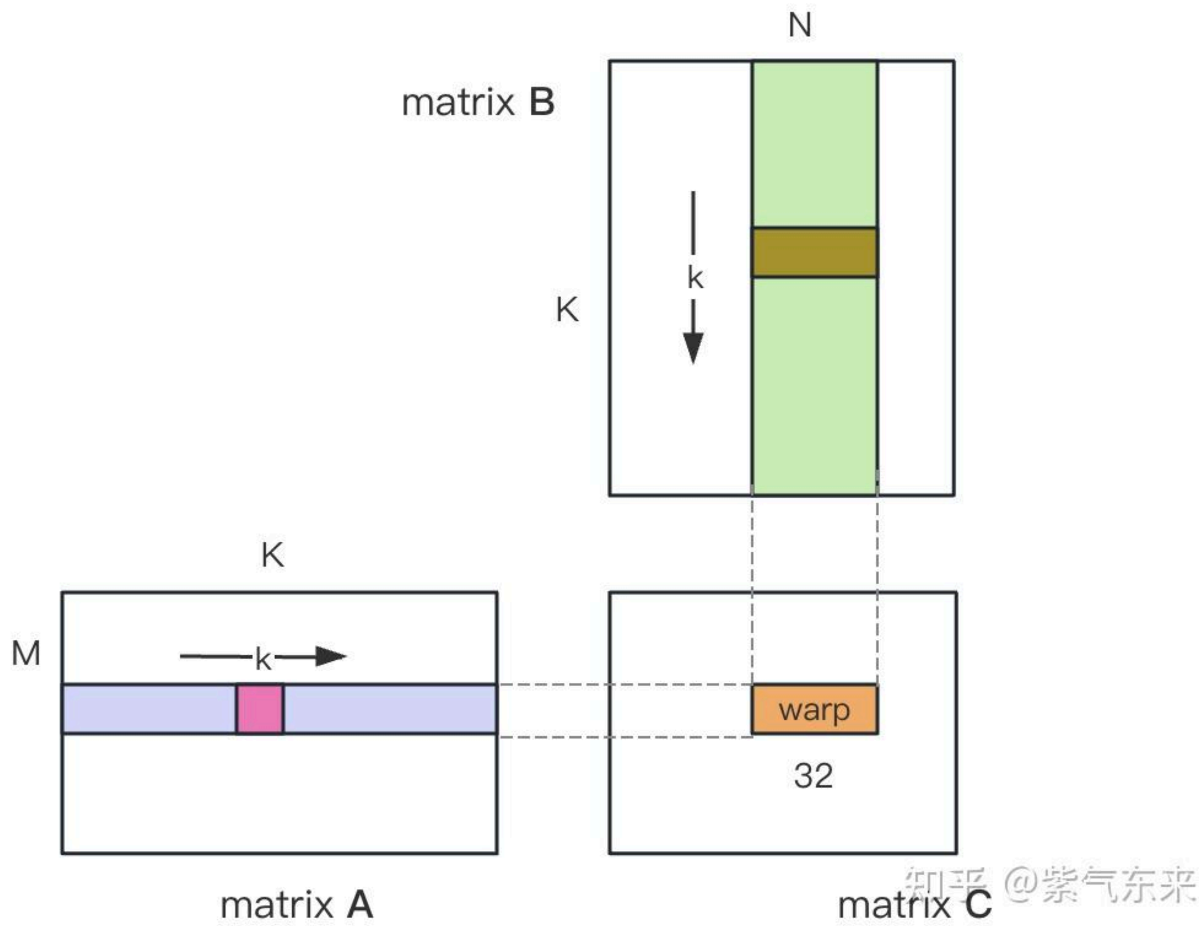
```
M N K =    128    128   1024, Time =   0.00010083   0.00010260   0.00010874 s, AVG Performance =   304.5951 Gflops
M N K =    192    192   1024, Time =   0.00010173   0.00010198   0.00010253 s, AVG Performance =   689.4680 Gflops
M N K =    256    256   1024, Time =   0.00010266   0.00010318   0.00010384 s, AVG Performance =  1211.4281 Gflops
M N K =    384    384   1024, Time =   0.00019475   0.00019535   0.00019594 s, AVG Performance =  1439.7206 Gflops
M N K =    512    512   1024, Time =   0.00037693   0.00037794   0.00037850 s, AVG Performance =  1322.9753 Gflops
M N K =    768    768   1024, Time =   0.00075238   0.00075558   0.00075776 s, AVG Performance =  1488.9271 Gflops
M N K =   1024   1024   1024, Time =   0.00121562   0.00121669   0.00121789 s, AVG Performance =  1643.8068 Gflops
M N K =   1536   1536   1024, Time =   0.00273072   0.00275611   0.00280208 s, AVG Performance =  1632.7386 Gflops
M N K =   2048   2048   1024, Time =   0.00487622   0.00488028   0.00488614 s, AVG Performance =  1639.2518 Gflops
M N K =   3072   3072   1024, Time =   0.01001603   0.01071136   0.01099990 s, AVG Performance =  1680.4589 Gflops
M N K =   4096   4096   1024, Time =   0.01771046   0.01792170   0.01803462 s, AVG Performance =  1785.5450 Gflops
M N K =   6144   6144   1024, Time =   0.03988969   0.03993405   0.04000595 s, AVG Performance =  1802.9724 Gflops
M N K =   8192   8192   1024, Time =   0.07119219   0.07139694   0.07160816 s, AVG Performance =  1792.7940 Gflops
M N K =  12288  12288   1024, Time =   0.15978026   0.15993242   0.16043369 s, AVG Performance =  1800.7606 Gflops
M N K =  16384  16384   1024, Time =   0.28559187   0.28567238   0.28573316 s, AVG Performance =  1792.2629 Gflops
```

下面以 M=512, K=512, N=512 为例，详细分析一下上述计算过程的workflow：

1. 在 Global Memory 中分别为矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ 分配存储空间.
2. 由于矩阵 $\boldsymbol{C}$ 中每个元素的计算均相互独立, 因此在并行度映射中让每个 thread 对应矩阵 $\boldsymbol{C}$ 中 1 个元素的计算.
3. 执行配置 (execution configuration)中 gridSize 和 blockSize 均有 x(列向)、y(行向)两个维度, 其中

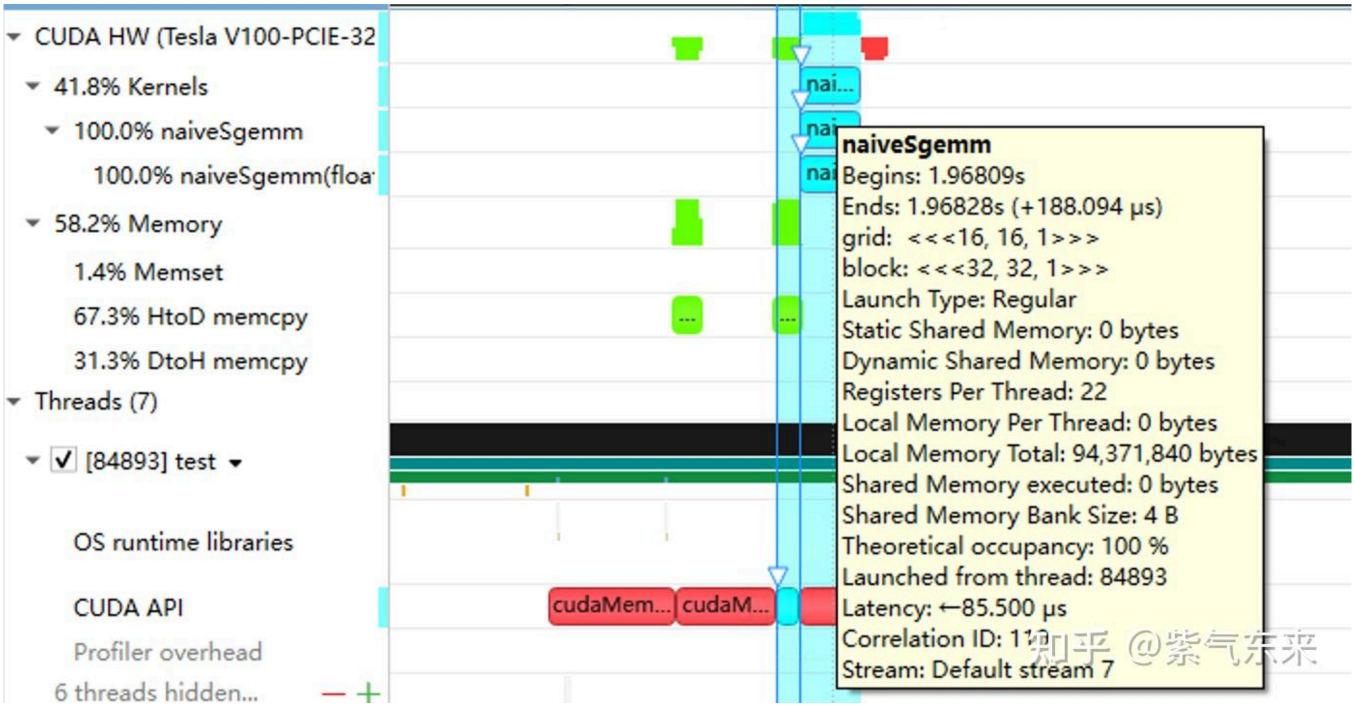
$$gridSize.x \times blockSize.x = N \\ gridSize.y \times blockSize.y = M$$

每个 thread 需要执行的 workflow 为：从矩阵 $\boldsymbol{A}$ 中读取一行向量 (长度为K), 从矩阵 $\boldsymbol{B}$ 中读取一列向量 (长度为K), 对这两个向量做点积运算 (单层 K 次循环的乘累加), 最后将结果写回矩阵 $\boldsymbol{C}$ 。由此可以计算出矩阵 $\boldsymbol{C}$ 的所有元素，读取矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$ 分别执行了 $K\times M \times N \times 4$Byte 的 load 操作，写回矩阵 $\boldsymbol{C}$ 需要执行 $M \times N \times 4$Byte 的 store 操作。



实际上，由于 GPU 的指令执行的最小的单元是 warp（32个 thread），同一 warp内的 thread 读写操作可以部分合并，具体是：对于 1 个 warp 中的 32 个 thread, 在每 1 次循环中, 需要读取矩阵 \boldsymbol{A} 同一个元素 (1 次 transaction)，以及矩阵 \boldsymbol{B} 连续的 32 个元素 (假设是理想的可合并访问的, 至少需要 4 次 transaction)，共发生 5 次 transaction。K 次循环总共需要 k×5 次 transactions. 对于 M×N 个 thread, 共有 M\times N/32 个 warp，总共的 Global Memory Load Transaction 数目为： M\times N/32 \times K \times 5 (注意, 并不是前文的 K\times M \times N \times 2 次)。

由此我们可以计算得到计算访存比为 2KMN/(KMN/32 \times 5 \times 4) = 3.2 OP/byte，由于实测带宽为 763GB/s (官方文档为900GB/s)，由此可以得到这种方式下理论算力最高可达到 64/20*763=2442 TFLOPS。



nsys 记录 的 naive 版本的 profiling

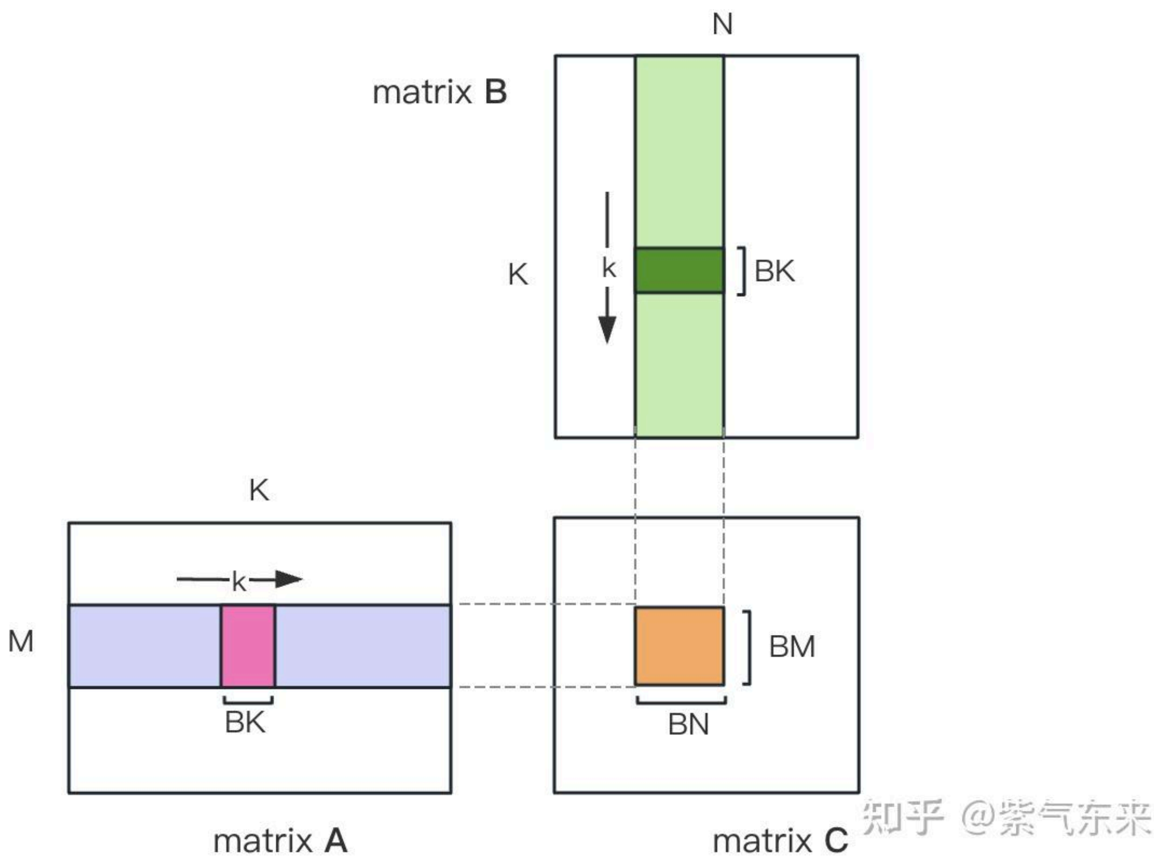
二、GEMM的优化探究

前文仅仅在功能上实现了 GEMM，性能上还远远不及预期，本节将主要研究 GEMM 性能上的优化。

2.1 矩阵分块利用Shared Memory

上述的计算需要两次Global Memory的load才能完成一次乘累加运算，计算访存比极低，没有有效的数据复用。所以可以用 Shared Memory 来减少重复的内存读取。

首先把矩阵 $\boldsymbol{C}$ 等分为$BM \times BN$大小的分块，每个分块由一个 Block 计算，其中每个Thread负责计算矩阵 $\boldsymbol{C}$ 中的 $TM \times TN$ 个元素。之后计算所需的数据全部从 smem 中读取，就消除了一部分重复的 $\boldsymbol{A}, \boldsymbol{B}$ 矩阵内存读取。考虑到 Shared Memory 容量有限，可以在K维上每次读取BK大小的分块，这样的循环一共需要K / BK次以完成整个矩阵乘法操作，即可得到 Block 的结果。其过程如下图所示：



利用 Shared Memory 优化后，对每一个分块，可得：

计算量：$BM \times BN \times K \times 2$

访存量：$(BM+BN)\times K \times 4Byte$

计算访存比：$\frac{BM\cdot BN}{2(BM+BN)}=\frac{1}{2(\frac{1}{BN}+\frac{1}{BM})}$

由上式可知BM和BN越大，计算访存比越高，性能就会越好。但是由于 Shared Memory 容量的限制(V100 1个SM仅96KB)，而一个Block需要占用 BK * (BM + BN) * 4 Bytes大小。

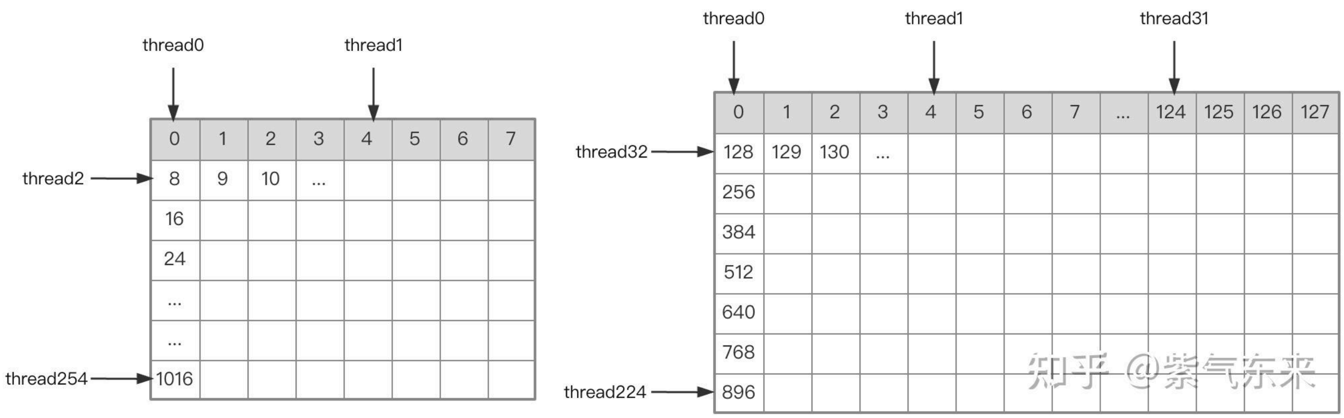
TM和TN的取值也受到两方面限制，一方面是线程数的限制，一个Block中有BM / TM * BN / TN个线程，这个数字不能超过1024，且不能太高防止影响SM内Block间的并行；另一方面是寄存器数目的限制，一个线程至少需要TM * TN个寄存器用于存放矩阵 $\boldsymbol{C}$ 的部分和，再加上一些其它的寄存器，所有的寄存器数目不能超过256，且不能太高防止影响SM内同时并行的线程数目。

最终选取 BM = BN = 128，BK = 8，TM = TN = 8，则此时计算访存比为32。根据V100的理论算力15.7TFLOPS，可得 15.7TFLOPS/32 = 490GB/s，根据实测的HBM带宽为763GB/s，可知此时带宽不再会限制计算性能。

根据以上分析，kernel 函数实现过程如下，完整代码参见 sgemm_v1.cu，主要步骤包括：

1）将矩阵分块 $A_{[BM,BK]}$, $B_{[BK,BN]}$ 存入 Shared Memory 中，

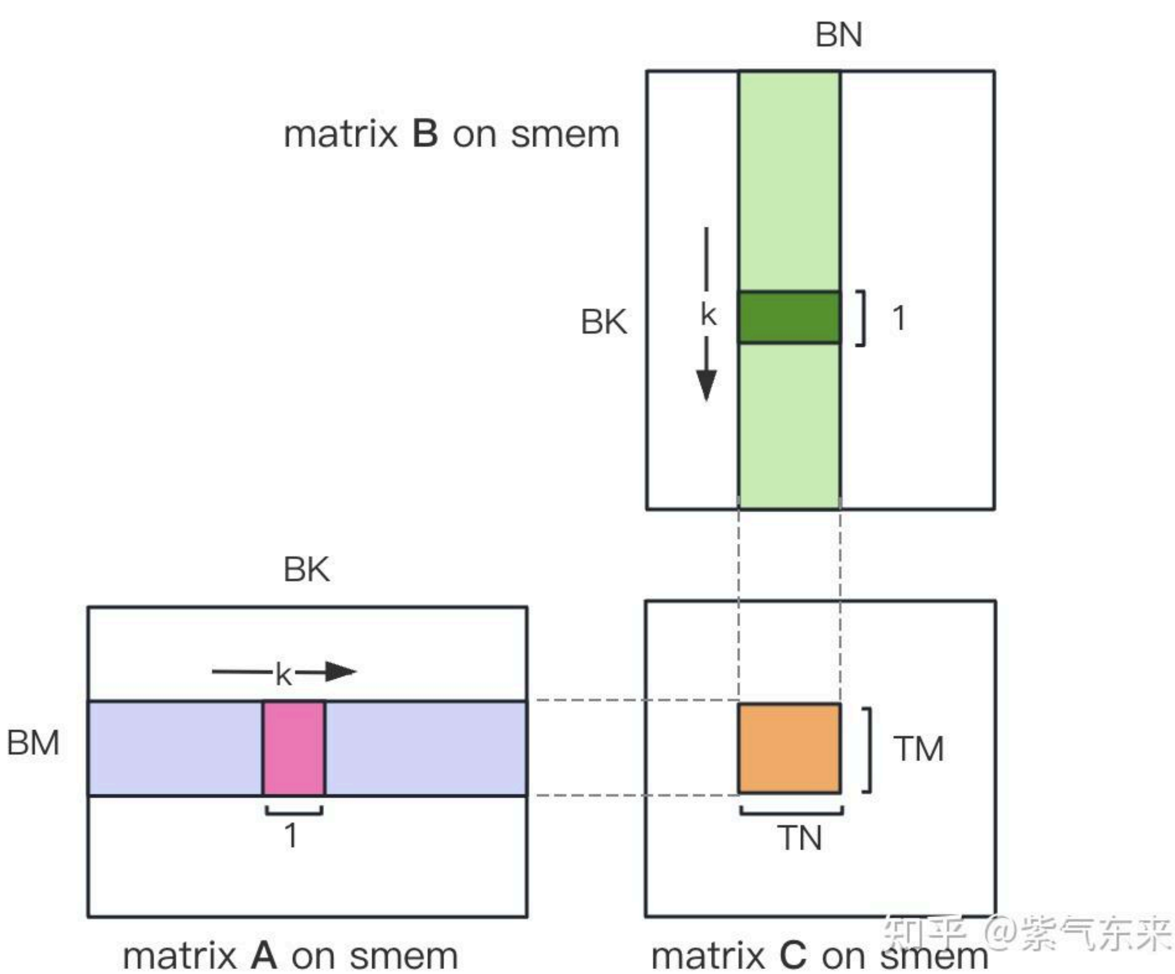
详细分析这一过程：`blockDim(BN / TN, BM / TM)` 即每个block有 $\frac{BM*BN}{TM*TN}$ 个 Thread，那么对于矩阵分块 $A_{[BM,BK]}$则每个Thread需要搬运 $\frac{BK*TM*TN}{BN}$ 个浮点数，在该例中该值为 4，刚好可以用 `FLOAT4` 函数来操作，对于[128,8] 的分块，Thread 的索引关系如下图左所示，代码中`load_a_smem_m = tid / 2 = tid >> 1` 表示s_a 的行号，`load_a_smem_k = (tid % 2 == 0) ? 0 : 4 = (tid & 1) << 2` 表示s_a 的列号。同理可分析矩阵分块 $B_{[BK,BN]}$ 的情况，不再赘述。



A B 矩阵分块的线程索引关系

确定好单个block的执行过程，接下来需要确定多block处理的不同分块在Global Memory中的对应关系，仍然以\boldsymbol{A} 为例进行说明。由于分块 A_{[BM,BK]} 沿着行的方向移动，那么首先需要确定行号，根据 Grid 的二维全局线性索引关系，`by * BM` 表示该分块的起始行号，同时我们已知`load_a_smem_m` 为分块内部的行号，因此全局的行号为`load_a_gmem_m = by * BM + load_a_smem_m` 。由于分块沿着行的方向移动，因此列是变化的，需要在循环内部计算，同样也是先计算起始列号`bk * BK` 加速分块内部列号`load_a_smem_k` 得到 `load_a_gmem_k = bk * BK + load_a_smem_k` ，由此我们便可以确定了分块在原始数据中的位置`OFFSET(load_a_gmem_m, load_a_gmem_k, K)` 。同理可分析矩阵分块 B_{[BK,BN]} 的情况，不再赘述。

2）计算矩阵分块 C_{[TM,TN]} ，在得到`s_a`，`s_b` 之后就可以按照定义计算对应的 `r_c` ，注意这里是更小的分块 TM*TN，其过程如下图所示



计算完 C_{[TM,TN]} 后，还需要将其存入 Global Memory 中，这就需要计算其在 Global Memory 中的对应关系。由于存在更小的分块，则行和列均由3部分构成：全局行号`store_c_gmem_m`等于大分块的起始行号`by * BM`+小分块的起始行号`ty * TM`+小分块内部的相对行号`i`。列同理。

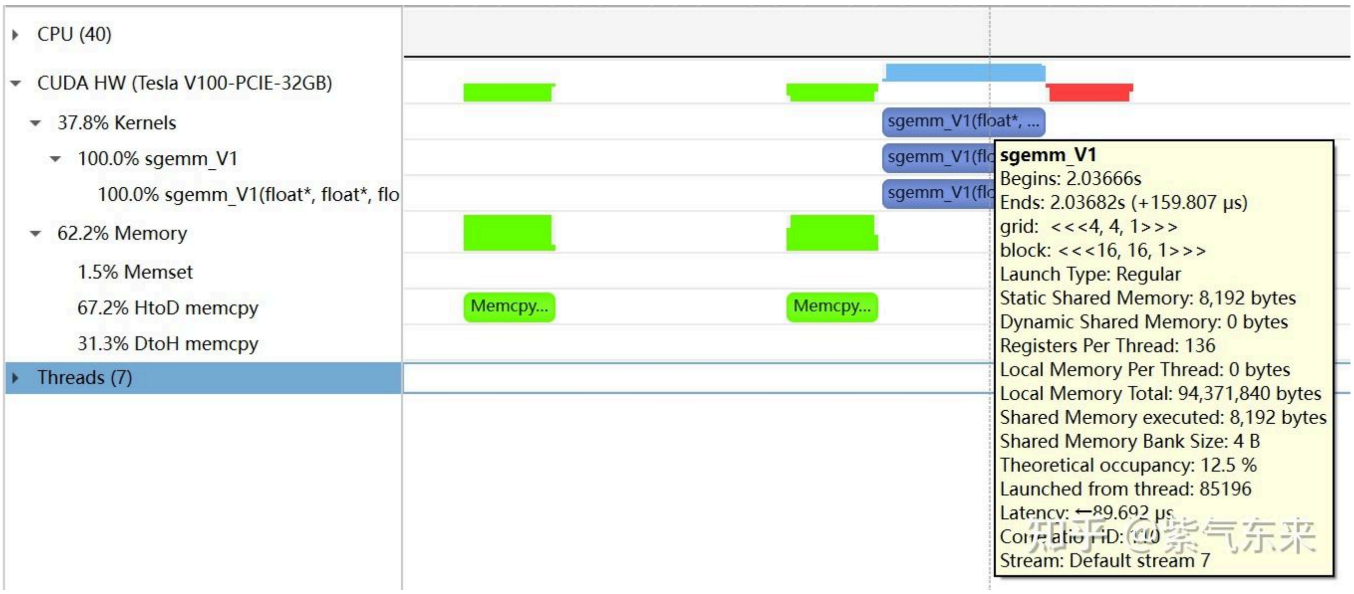
```
__global__ void sgemm_V1(
    float * __restrict__ a, float * __restrict__ b, float * __restrict__ c,
    const int M, const int N, const int K) {

    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;
    const int ty = threadIdx.y;
    const int tid = ty * blockDim.x + tx;

    __shared__ float s_a[BM][BK];
    __shared__ float s_b[BK][BN];

    float r_c[TM][TN] = {0.0};

    int load_a_smem_m = tid >> 1;  // tid/2, row of s_a
    int load_a_smem_k = (tid & 1) << 2;  // (tid % 2 == 0) ? 0 : 4, col of s_a
    int load_b_smem_k = tid >> 5;   // tid/32, row of s_b
    int load_b_smem_n = (tid & 31) << 2;  // (tid % 32) * 4, col of s_b

    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

    for (int bk = 0; bk < (K + BK - 1) / BK; bk++) {
        int load_a_gmem_k = bk * BK + load_a_smem_k;   // global col of a
        int load_a_gmem_addr = OFFSET(load_a_gmem_m, load_a_gmem_k, K);
        FLOAT4(s_a[load_a_smem_m][load_a_smem_k]) = FLOAT4(a[load_a_gmem_addr]);
        int load_b_gmem_k = bk * BK + load_b_smem_k;   // global row of b
        int load_b_gmem_addr = OFFSET(load_b_gmem_k, load_b_gmem_n, N);
        FLOAT4(s_b[load_b_smem_k][load_b_smem_n]) = FLOAT4(b[load_b_gmem_addr]);
        __syncthreads();

        #pragma unroll
        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;
                    r_c[m][n] += s_a[comp_a_smem_m][k] * s_b[k][comp_b_smem_n];
                }
            }
        }

        __syncthreads();
    }

    #pragma unroll
    for (int i = 0; i < TM; i++) {
        int store_c_gmem_m = by * BM + ty * TM + i;
        #pragma unroll
        for (int j = 0; j < TN; j += 4) {
            int store_c_gmem_n = bx * BN + tx * TN + j;
            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]);
        }
    }
}
```

计算结果如下，性能达到了理论峰值性能的51.7%：

```
M N K =    128    128   1024, Time =   0.00031578   0.00031727   0.00032288 s, AVG Performance =    98.4974 Gflops
M N K =    192    192   1024, Time =   0.00031638   0.00031720   0.00031754 s, AVG Performance =   221.6661 Gflops
M N K =    256    256   1024, Time =   0.00031488   0.00031532   0.00031606 s, AVG Performance =   396.4287 Gflops
M N K =    384    384   1024, Time =   0.00031686   0.00031814   0.00032080 s, AVG Performance =   884.0425 Gflops
```

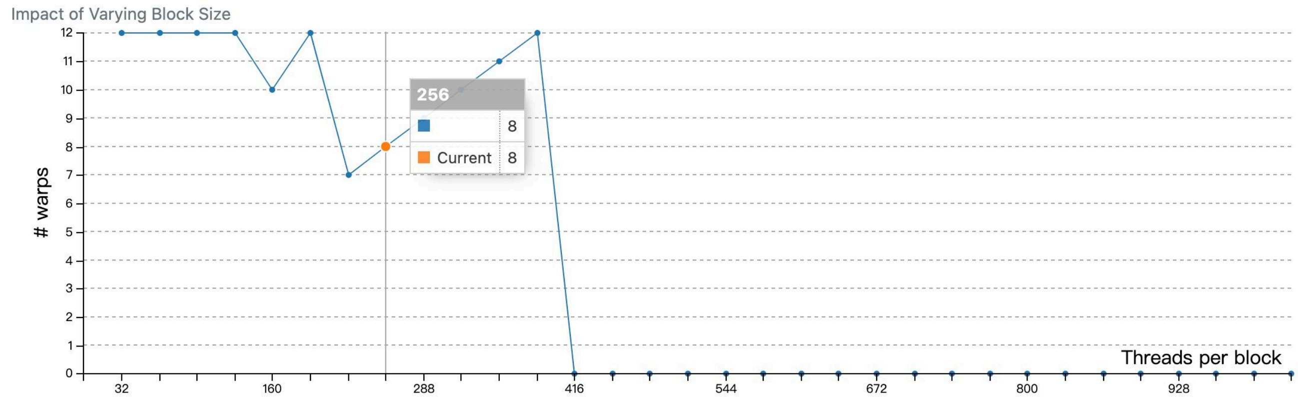
```
M N K =    512    512   1024, Time =   0.00031814   0.00032007   0.00032493 s, AVG Performance =   1562.1563 Gflops
M N K =    768    768   1024, Time =   0.00032397   0.00034419   0.00034848 s, AVG Performance =   3268.5245 Gflops
M N K =   1024   1024   1024, Time =   0.00034570   0.00034792   0.00035331 s, AVG Performance =   5748.3952 Gflops
M N K =   1536   1536   1024, Time =   0.00068797   0.00068983   0.00069094 s, AVG Performance =   6523.3424 Gflops
M N K =   2048   2048   1024, Time =   0.00136173   0.00136552   0.00136899 s, AVG Performance =   5858.5604 Gflops
M N K =   3072   3072   1024, Time =   0.00271910   0.00273115   0.00274006 s, AVG Performance =   6590.6331 Gflops
M N K =   4096   4096   1024, Time =   0.00443805   0.00445964   0.00446883 s, AVG Performance =   7175.4698 Gflops
M N K =   6144   6144   1024, Time =   0.00917891   0.00950608   0.00996963 s, AVG Performance =   7574.0999 Gflops
M N K =   8192   8192   1024, Time =   0.01628838   0.01645271   0.01660790 s, AVG Performance =   7779.8733 Gflops
M N K =  12288  12288   1024, Time =   0.03592557   0.03597434   0.03614323 s, AVG Performance =   8005.7066 Gflops
M N K =  16384  16384   1024, Time =   0.06304122   0.06306373   0.06309302 s, AVG Performance =   8118.7715 Gflops
```

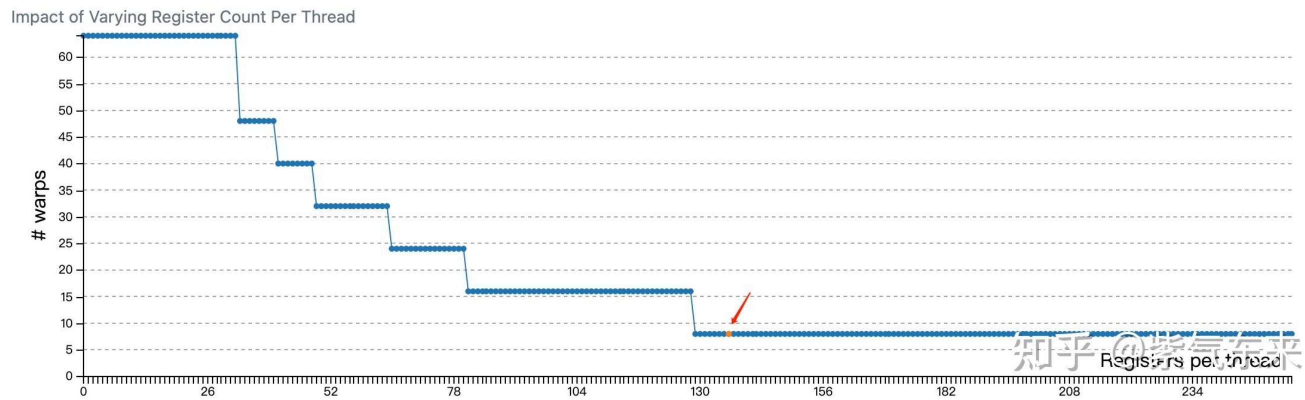
下面仍以 M=512, K=512, N=512 为例，分析一下结果。首先通过 profiling 可以看到 Shared Memory 占用为 8192 bytes，这与理论上 (128+128)\times 8 \times 4 完全一致。



nsys 记录 的 V1 版本的 profiling

profiling 显示 Occupancy 为 12.5%，可以通过 cuda-calculator 加以印证，该例中 threads per block = 256, Registers per thread = 136, 由此可以计算得到每个SM中活跃的 warp 为8，而对于V100，每个SM中的 warp 总数为64，因此 Occupancy 为 8/64 = 12.5%。

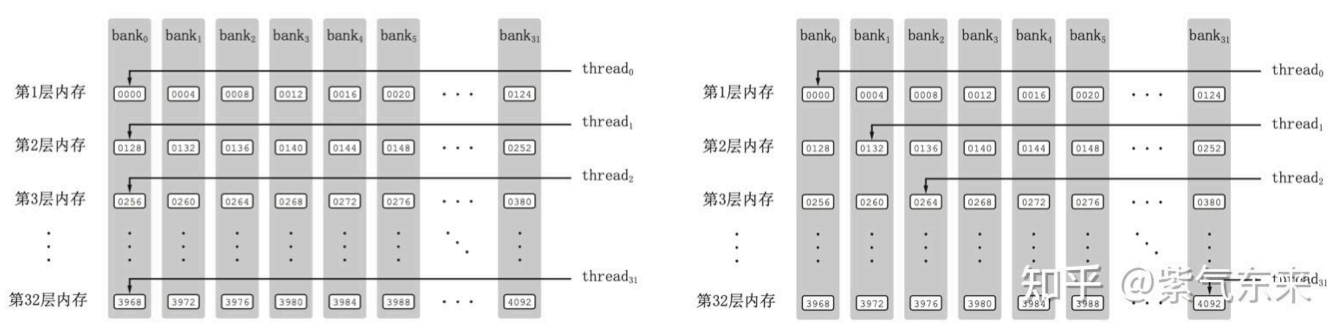




2.2 解决 Bank Conflict 问题

上节通过利用 Shared Memory 大幅提高了访存效率，进而提高了性能，本节将进一步优化 Shared Memory 的使用。

Shared Memory一共划分为32个Bank，每个Bank的宽度为4 Bytes，如果需要访问同一个Bank的多个数据，就会发生Bank Conflict。例如一个Warp的32个线程，如果访问的地址分别为0、4、8、...、124，就不会发生Bank Conflict，只占用Shared Memory一拍的时间；如果访问的地址为0、8、16、...、248，这样一来地址0和地址128对应的数据位于同一Bank、地址4和地址132对应的数据位于同一Bank，以此类推，那么就需要占用Shared Memory两拍的时间才能读出。



有 Bank Conflict VS 无 Bank Conflict

再看 V1 版本计算部分的三层循环，每次从Shared memory中取矩阵 \boldsymbol{A} 的长度为TM的向量和矩阵 \boldsymbol{B} 的长度为TN的向量，这两个向量做外积并累加到部分和中，一次外积共TM * TN次乘累加，一共需要循环BK次取数和外积。

接下来分析从Shared Memory load的过程中存在的Bank Conflict：

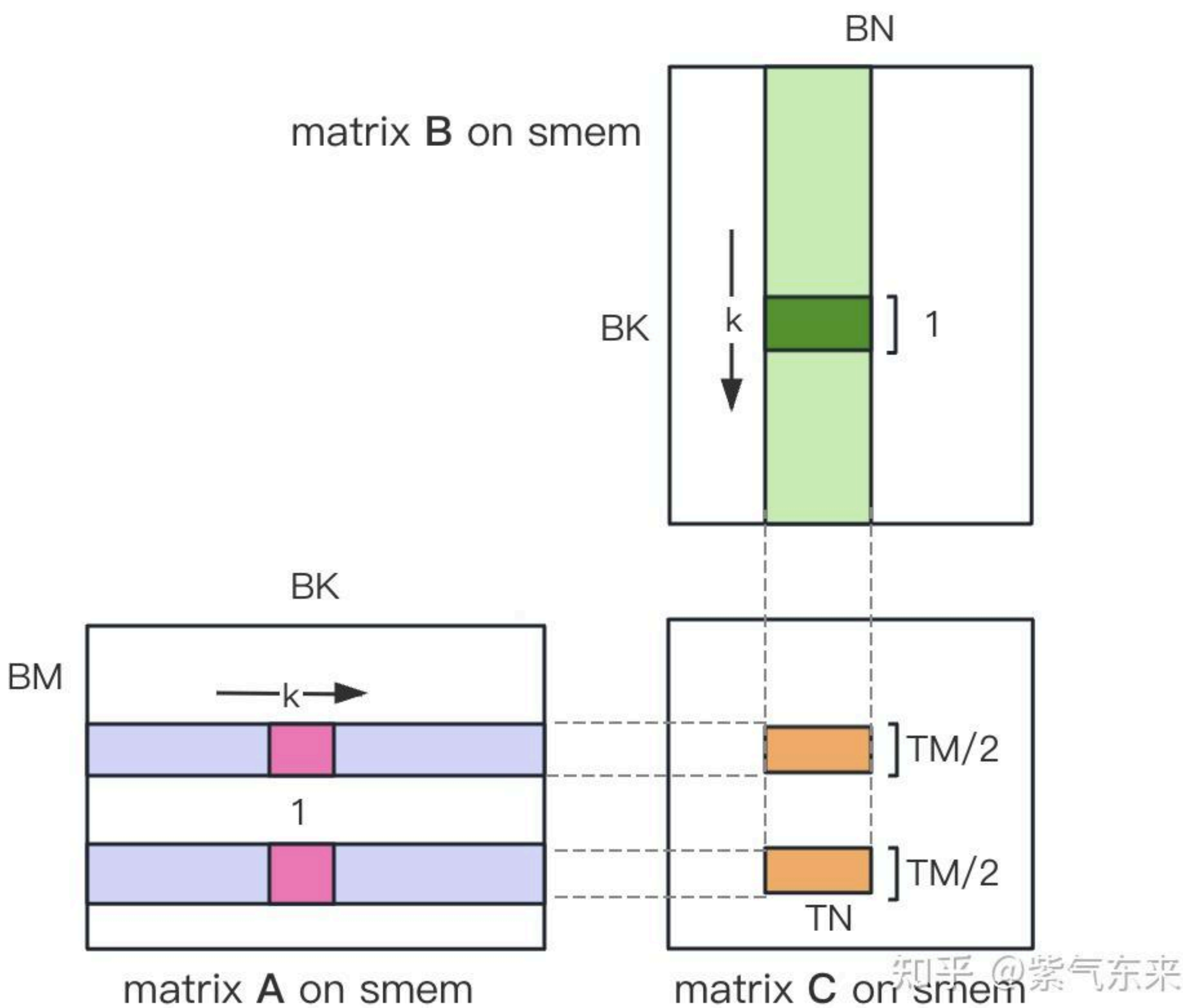
i) 取矩阵 \boldsymbol{A} 需要取一个列向量，而矩阵 \boldsymbol{A} 在Shared Memory中是按行存储的；

ii) 在TM = TN = 8的情况下，无论矩阵A还是矩阵B，从Shared Memory中取数时需要取连续的8个数，即使用LDS.128指令一条指令取四个数，也需要两条指令，由于一个线程的两条load指令的地址是连续的，那么同一个Warp不同线程的同一条load指令的访存地址就是被间隔开的，便存在着 Bank Conflict。

为了解决上述的两点Shared Memory的Bank Conflict，采用了一下两点优化：

i) 为矩阵 $\boldsymbol{A}$ 分配Shared Memory时形状分配为[BK][BM]，即让矩阵 $\boldsymbol{A}$ 在Shared Memory中按列存储

ii) 将原本每个线程负责计算的TM * TN的矩阵 $\boldsymbol{C}$ ，划分为下图中这样的两块TM/2 * TN的矩阵 $\boldsymbol{C}$ ，由于TM/2=4，一条指令即可完成A的一块的load操作，两个load可同时进行。



kernel 函数的核心部分实现如下，完整代码见 sgemm_v2.cu 。

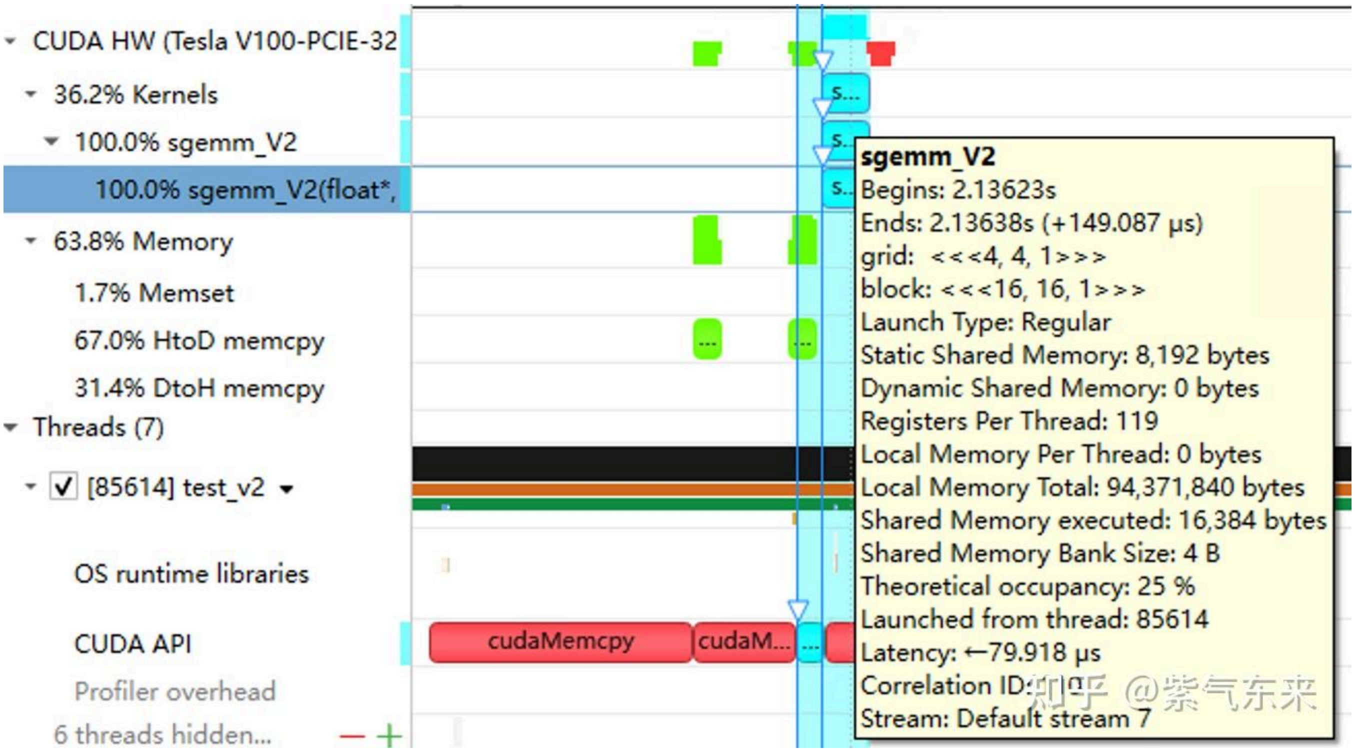
```
__shared__ float s_a[BK][BM];
__shared__ float s_b[BK][BN];

float r_load_a[4];
float r_load_b[4];
float r_comp_a[TM];
float r_comp_b[TN];
float r_c[TM][TN] = {0.0};

int load_a_smem_m = tid >> 1;
int load_a_smem_k = (tid & 1) << 2;
int load_b_smem_k = tid >> 5;
int load_b_smem_n = (tid & 31) << 2;

int load_a_gmem_m = by * BM + load_a_smem_m;
int load_b_gmem_n = bx * BN + load_b_smem_n;
```

```
    for (int bk = 0; bk < (K + BK - 1) / BK; bk++) {

        int load_a_gmem_k = bk * BK + load_a_smem_k;
        int load_a_gmem_addr = OFFSET(load_a_gmem_m, load_a_gmem_k, K);
        int load_b_gmem_k = bk * BK + load_b_smem_k;
        int load_b_gmem_addr = OFFSET(load_b_gmem_k, load_b_gmem_n, N);
        FLOAT4(r_load_a[0]) = FLOAT4(a[load_a_gmem_addr]);
        FLOAT4(r_load_b[0]) = FLOAT4(b[load_b_gmem_addr]);

        s_a[load_a_smem_k    ][load_a_smem_m] = r_load_a[0];
        s_a[load_a_smem_k + 1][load_a_smem_m] = r_load_a[1];
        s_a[load_a_smem_k + 2][load_a_smem_m] = r_load_a[2];
        s_a[load_a_smem_k + 3][load_a_smem_m] = r_load_a[3];
        FLOAT4(s_b[load_b_smem_k][load_b_smem_n]) = FLOAT4(r_load_b[0]);

        __syncthreads();

        #pragma unroll
        for (int tk = 0; tk < BK; tk++) {
            FLOAT4(r_comp_a[0]) = FLOAT4(s_a[tk][ty * TM / 2         ]);
            FLOAT4(r_comp_a[4]) = FLOAT4(s_a[tk][ty * TM / 2 + BM / 2]);
            FLOAT4(r_comp_b[0]) = FLOAT4(s_b[tk][tx * TN / 2         ]);
            FLOAT4(r_comp_b[4]) = FLOAT4(s_b[tk][tx * TN / 2 + BN / 2]);

            #pragma unroll
            for (int tm = 0; tm < TM; tm++) {
                #pragma unroll
                for (int tn = 0; tn < TN; tn++) {
                    r_c[tm][tn] += r_comp_a[tm] * r_comp_b[tn];
                }
            }
        }

        __syncthreads();
    }

    #pragma unroll
    for (int i = 0; i < TM / 2; i++) {
        int store_c_gmem_m = by * BM + ty * TM / 2 + i;
        int store_c_gmem_n = bx * BN + tx * TN / 2;
        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][0]);
        FLOAT4(c[store_c_gmem_addr + BN / 2]) = FLOAT4(r_c[i][4]);
    }
    #pragma unroll
    for (int i = 0; i < TM / 2; i++) {
        int store_c_gmem_m = by * BM + BM / 2 + ty * TM / 2 + i;
        int store_c_gmem_n = bx * BN + tx * TN / 2;
        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 + TM / 2][0]);
        FLOAT4(c[store_c_gmem_addr + BN / 2]) = FLOAT4(r_c[i + TM / 2][4]);
    }
```

结果如下，相对未解决 Bank Conflict 版(V1) 性能提高了 14.4%，达到了理论峰值的74.3%。

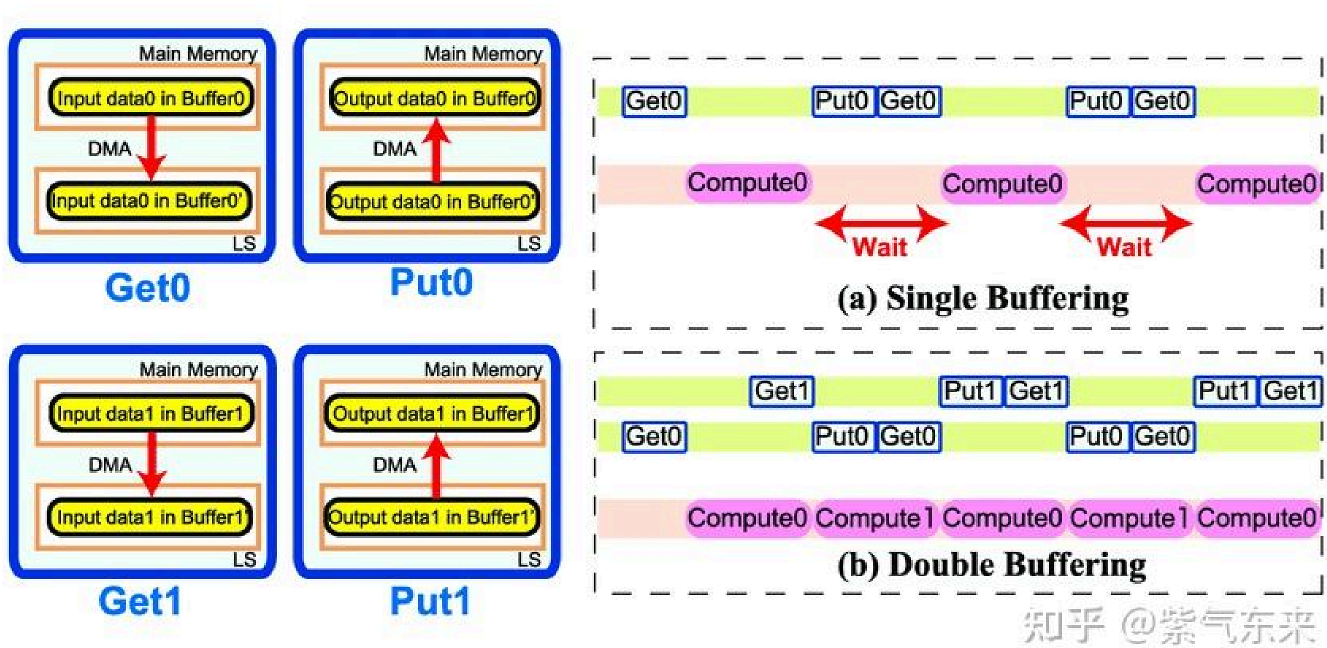
```
M N K =    128    128   1024, Time =   0.00029699   0.00029918   0.00030989 s, AVG Performance =   104.4530 Gflops
M N K =    192    192   1024, Time =   0.00029776   0.00029828   0.00029882 s, AVG Performance =   235.7252 Gflops
M N K =    256    256   1024, Time =   0.00029485   0.00029530   0.00029619 s, AVG Performance =   423.2949 Gflops
M N K =    384    384   1024, Time =   0.00029734   0.00029848   0.00030090 s, AVG Performance =   942.2843 Gflops
M N K =    512    512   1024, Time =   0.00029853   0.00029945   0.00030070 s, AVG Performance =  1669.7479 Gflops
M N K =    768    768   1024, Time =   0.00030458   0.00032467   0.00032790 s, AVG Performance =  3465.1038 Gflops
M N K =   1024   1024   1024, Time =   0.00032406   0.00032494   0.00032621 s, AVG Performance =  6155.0281 Gflops
M N K =   1536   1536   1024, Time =   0.00047990   0.00048224   0.00048461 s, AVG Performance =  9331.3912 Gflops
M N K =   2048   2048   1024, Time =   0.00094426   0.00094636   0.00094992 s, AVG Performance =  8453.4569 Gflops
M N K =   3072   3072   1024, Time =   0.00187866   0.00188096   0.00188538 s, AVG Performance =  9569.5816 Gflops
M N K =   4096   4096   1024, Time =   0.00312589   0.00319050   0.00328147 s, AVG Performance = 10029.7885 Gflops
M N K =   6144   6144   1024, Time =   0.00641280   0.00658940   0.00703498 s, AVG Performance = 10926.6372 Gflops
M N K =   8192   8192   1024, Time =   0.01101130   0.01116194   0.01122950 s, AVG Performance = 11467.5446 Gflops
M N K =  12288  12288   1024, Time =   0.02464854   0.02466705   0.02469344 s, AVG Performance = 11675.4946 Gflops
M N K =  16384  16384   1024, Time =   0.04385955   0.04387468   0.04388355 s, AVG Performance = 11669.5995 Gflops
```

分析一下 profiling 可以看到 Static Shared Memory 仍然是使用了8192 Bytes，奇怪的的是，Shared Memory executed 却翻倍变成了 16384 Bytes（知友如果知道原因可以告诉我一下）。



2.3 流水并行化：Double Buffering

Double Buffering，即双缓冲，即通过增加buffer的方式，使得 **访存-计算** 的串行模式流水线化，以减少等待时间，提高计算效率，其原理如下图所示:



Single Buffering VS Double Buffering

具体到 GEMM 任务中来，就是需要两倍的Shared Memory，之前只需要BK * (BM + BN) * 4 Bytes的Shared Memory，采用Double Buffering之后需要2BK * (BM + BN) * 4 Bytes的Shared Memory，然后使其 pipeline 流动起来。

代码核心部分如下所示，完整代码参见 sgemm_v3.cu 。有以下几点需要注意:

1）主循环从`bk = 1`开始，第一次数据加载在主循环之前，最后一次计算在主循环之后，这是pipeline 的特点决定的；

2）由于计算和下一次访存使用的Shared Memory不同，因此主循环中每次循环只需要一次__syncthreads()即可

3）由于GPU不能向CPU那样支持乱序执行，主循环中需要先将下一次循环计算需要的Gloabal Memory中的数据load 到寄存器，然后进行本次计算，之后再将load到寄存器中的数据写到Shared Memory，这样在LDG指令向Global Memory做load时，不会影响后续FFMA及其它运算指令的 launch 执行，也就达到了Double Buffering的目的。

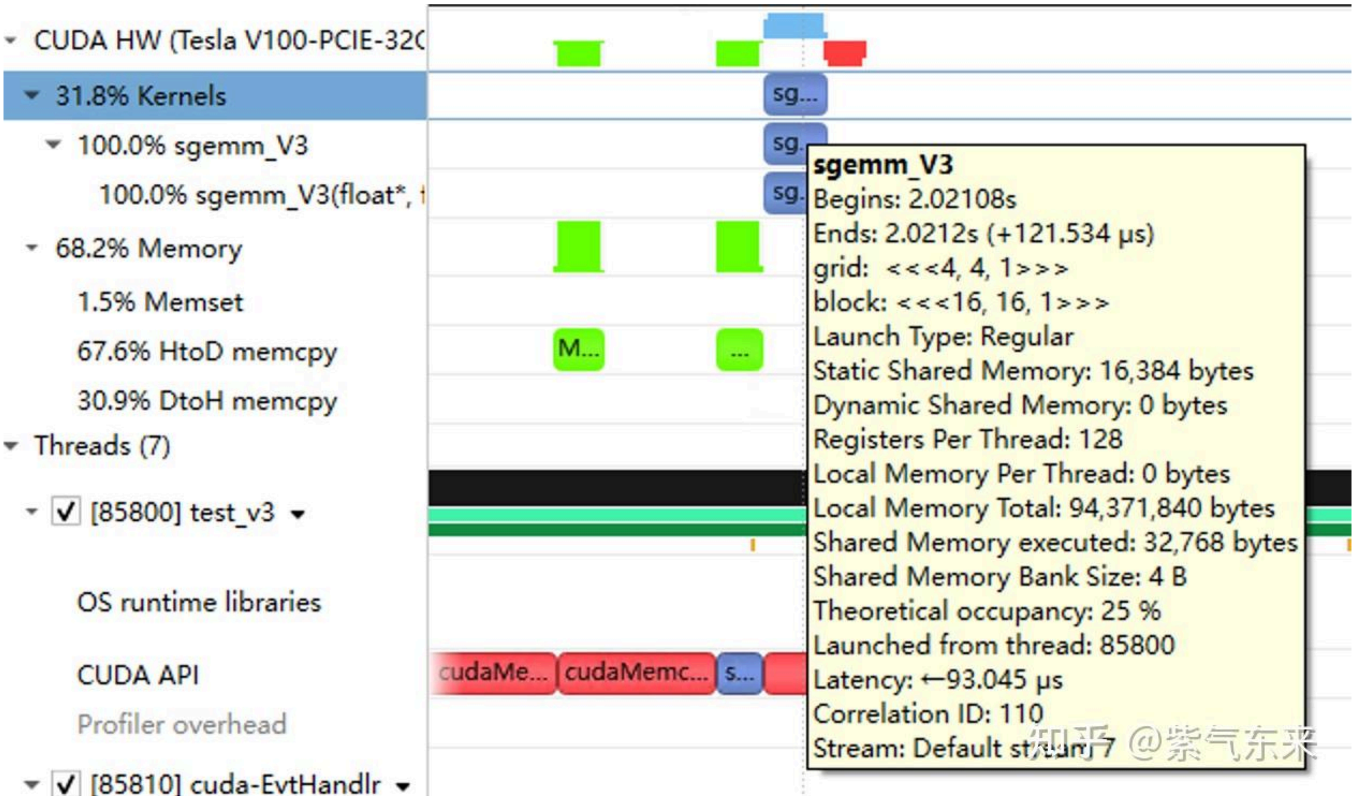
```
    __shared__ float s_a[2][BK][BM];
    __shared__ float s_b[2][BK][BN];

    float r_load_a[4];
    float r_load_b[4];
    float r_comp_a[TM];
    float r_comp_b[TN];
    float r_c[TM][TN] = {0.0};

    int load_a_smem_m = tid >> 1;
    int load_a_smem_k = (tid & 1) << 2;
    int load_b_smem_k = tid >> 5;
    int load_b_smem_n = (tid & 31) << 2;

    int load_a_gmem_m = by * BM + load_a_smem_m;
    int load_b_gmem_n = bx * BN + load_b_smem_n;

    {
        int load_a_gmem_k = load_a_smem_k;
        int load_a_gmem_addr = OFFSET(load_a_gmem_m, load_a_gmem_k, K);
        int load_b_gmem_k = load_b_smem_k;
        int load_b_gmem_addr = OFFSET(load_b_gmem_k, load_b_gmem_n, N);
        FLOAT4(r_load_a[0]) = FLOAT4(a[load_a_gmem_addr]);
        FLOAT4(r_load_b[0]) = FLOAT4(b[load_b_gmem_addr]);

        s_a[0][load_a_smem_k    ][load_a_smem_m] = r_load_a[0];
        s_a[0][load_a_smem_k + 1][load_a_smem_m] = r_load_a[1];
        s_a[0][load_a_smem_k + 2][load_a_smem_m] = r_load_a[2];
        s_a[0][load_a_smem_k + 3][load_a_smem_m] = r_load_a[3];
        FLOAT4(s_b[0][load_b_smem_k][load_b_smem_n]) = FLOAT4(r_load_b[0]);
    }

    for (int bk = 1; bk < (K + BK - 1) / BK; bk++) {

        int smem_sel = (bk - 1) & 1;
        int smem_sel_next = bk & 1;

        int load_a_gmem_k = bk * BK + load_a_smem_k;
        int load_a_gmem_addr = OFFSET(load_a_gmem_m, load_a_gmem_k, K);
        int load_b_gmem_k = bk * BK + load_b_smem_k;
        int load_b_gmem_addr = OFFSET(load_b_gmem_k, load_b_gmem_n, N);
        FLOAT4(r_load_a[0]) = FLOAT4(a[load_a_gmem_addr]);
        FLOAT4(r_load_b[0]) = FLOAT4(b[load_b_gmem_addr]);

        #pragma unroll
        for (int tk = 0; tk < BK; tk++) {
            FLOAT4(r_comp_a[0]) = FLOAT4(s_a[smem_sel][tk][ty * TM / 2         ]);
            FLOAT4(r_comp_a[4]) = FLOAT4(s_a[smem_sel][tk][ty * TM / 2 + BM / 2]);
            FLOAT4(r_comp_b[0]) = FLOAT4(s_b[smem_sel][tk][tx * TN / 2         ]);
            FLOAT4(r_comp_b[4]) = FLOAT4(s_b[smem_sel][tk][tx * TN / 2 + BN / 2]);

            #pragma unroll
            for (int tm = 0; tm < TM; tm++) {
                #pragma unroll
                for (int tn = 0; tn < TN; tn++) {
                    r_c[tm][tn] += r_comp_a[tm] * r_comp_b[tn];
                }
            }
        }

        s_a[smem_sel_next][load_a_smem_k    ][load_a_smem_m] = r_load_a[0];
        s_a[smem_sel_next][load_a_smem_k + 1][load_a_smem_m] = r_load_a[1];
        s_a[smem_sel_next][load_a_smem_k + 2][load_a_smem_m] = r_load_a[2];
```

```
        s_a[smem_sel_next][load_a_smem_k + 3][load_a_smem_m] = r_load_a[3];
        FLOAT4(s_b[smem_sel_next][load_b_smem_k][load_b_smem_n]) = FLOAT4(r_load_b[0]);

        __syncthreads();
    }

    #pragma unroll
    for (int tk = 0; tk < BK; tk++) {
        FLOAT4(r_comp_a[0]) = FLOAT4(s_a[1][tk][ty * TM / 2         ]);
        FLOAT4(r_comp_a[4]) = FLOAT4(s_a[1][tk][ty * TM / 2 + BM / 2]);
        FLOAT4(r_comp_b[0]) = FLOAT4(s_b[1][tk][tx * TN / 2         ]);
        FLOAT4(r_comp_b[4]) = FLOAT4(s_b[1][tk][tx * TN / 2 + BN / 2]);

        #pragma unroll
        for (int tm = 0; tm < TM; tm++) {
            #pragma unroll
            for (int tn = 0; tn < TN; tn++) {
                r_c[tm][tn] += r_comp_a[tm] * r_comp_b[tn];
            }
        }
    }

    #pragma unroll
    for (int i = 0; i < TM / 2; i++) {
        int store_c_gmem_m = by * BM + ty * TM / 2 + i;
        int store_c_gmem_n = bx * BN + tx * TN / 2;
        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][0]);
        FLOAT4(c[store_c_gmem_addr + BN / 2]) = FLOAT4(r_c[i][4]);
    }
    #pragma unroll
    for (int i = 0; i < TM / 2; i++) {
        int store_c_gmem_m = by * BM + BM / 2 + ty * TM / 2 + i;
        int store_c_gmem_n = bx * BN + tx * TN / 2;
        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 + TM / 2][0]);
        FLOAT4(c[store_c_gmem_addr + BN / 2]) = FLOAT4(r_c[i + TM / 2][4]);
    }
```

性能如下所示，达到了理论峰值的 80.6%。

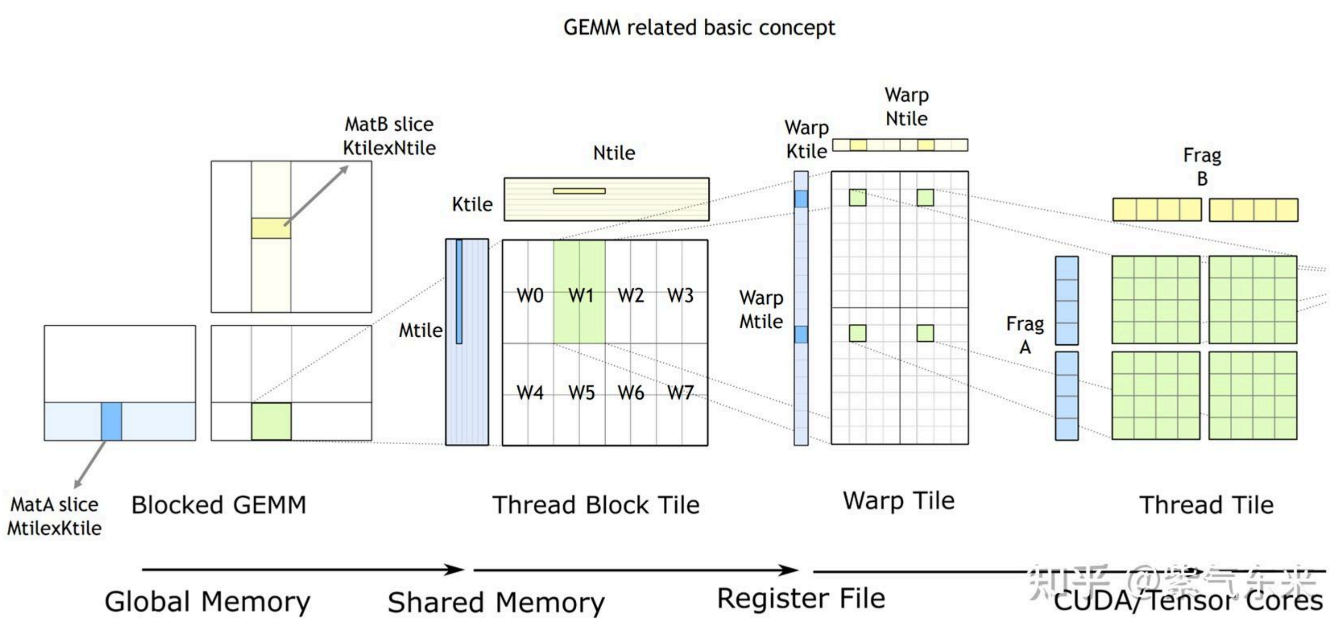
```
M N K =   128   128  1024, Time =   0.00024000   0.00024240   0.00025792 s, AVG Performance =   128.9191 Gflops
M N K =   192   192  1024, Time =   0.00024000   0.00024048   0.00024125 s, AVG Performance =   292.3840 Gflops
M N K =   256   256  1024, Time =   0.00024029   0.00024114   0.00024272 s, AVG Performance =   518.3728 Gflops
M N K =   384   384  1024, Time =   0.00024070   0.00024145   0.00024198 s, AVG Performance =  1164.8394 Gflops
M N K =   512   512  1024, Time =   0.00024173   0.00024237   0.00024477 s, AVG Performance =  2062.9786 Gflops
M N K =   768   768  1024, Time =   0.00024291   0.00024540   0.00026010 s, AVG Performance =  4584.3820 Gflops
M N K =  1024  1024  1024, Time =   0.00024534   0.00024631   0.00024941 s, AVG Performance =  8119.7302 Gflops
M N K =  1536  1536  1024, Time =   0.00045712   0.00045780   0.00045872 s, AVG Performance =  9829.5167 Gflops
M N K =  2048  2048  1024, Time =   0.00089632   0.00089970   0.00090656 s, AVG Performance =  8891.8924 Gflops
M N K =  3072  3072  1024, Time =   0.00177891   0.00178289   0.00178592 s, AVG Performance = 10095.9883 Gflops
M N K =  4096  4096  1024, Time =   0.00309763   0.00310057   0.00310451 s, AVG Performance = 10320.6843 Gflops
M N K =  6144  6144  1024, Time =   0.00604826   0.00619887   0.00663078 s, AVG Performance = 11615.0253 Gflops
M N K =  8192  8192  1024, Time =   0.01031738   0.01045051   0.01048861 s, AVG Performance = 12248.2036 Gflops
M N K = 12288 12288  1024, Time =   0.02283978   0.02285837   0.02298272 s, AVG Performance = 12599.3212 Gflops
M N K = 16384 16384  1024, Time =   0.04043287   0.04044823   0.04046151 s, AVG Performance = 12658.1556 Gflops
```

从 profiling 可以看到双倍的 Shared Memory 的占用



三、cuBLAS 实现方式探究

本节我们将认识CUDA的标准库——cuBLAS， 即NVIDIA版本的基本线性代数子程序 (Basic Linear Algebra Subprograms, BLAS) 规范实现代码。它支持 Level 1 (向量与向量运算) ， Level 2 (向量与矩阵运算) ， Level 3 (矩阵与矩阵运算) 级别的标准矩阵运算。



cuBLAS/CUTLASS GEMM的基本过程

如上图所示，计算过程分解成**线程块片（thread block tile）**、**线程束片（warp tile）**和**线程片（thread tile）**的层次结构并将AMP的策略应用于此层次结构来高效率的完成基于GPU的拆分成tile的GEMM。这个层次结构紧密地反映了NVIDIA CUDA编程模型。可以看到从global memory到shared memory的数据移动（矩阵到thread block tile）；从shared memory到寄存器的数据移动（thread block tile到warp tile）；从寄存器到CUDA core的计算（warp tile到thread tile）。

cuBLAS 实现了单精度矩阵乘的函数cublasSgemm，其主要参数如下：

```
cublasStatus_t cublasSgemm( cublasHandle_t handle, // 调用 cuBLAS 库时的句柄
                            cublasOperation_t transa, // A 矩阵是否需要转置
                            cublasOperation_t transb, // B 矩阵是否需要转置
                            int m, // A 的行数
                            int n, // B 的列数
                            int k, // A 的列数
                            const float *alpha, // 系数 α, host or device pointer
                            const float *A, // 矩阵 A 的指针，device pointer
                            int lda, // 矩阵 A 的主维，if A 转置， lda = max(1, k), else max(1, m)
                            const float *B, // 矩阵 B 的指针, device pointer
                            int ldb, // 矩阵 B 的主维，if B 转置， ldb = max(1, n), else max(1, k)
                            const float *beta, // 系数 β, host or device pointer
                            float *C, // 矩阵 C 的指针，device pointer
                            int ldc // 矩阵 C 的主维，ldc >= max(1, m) );
```

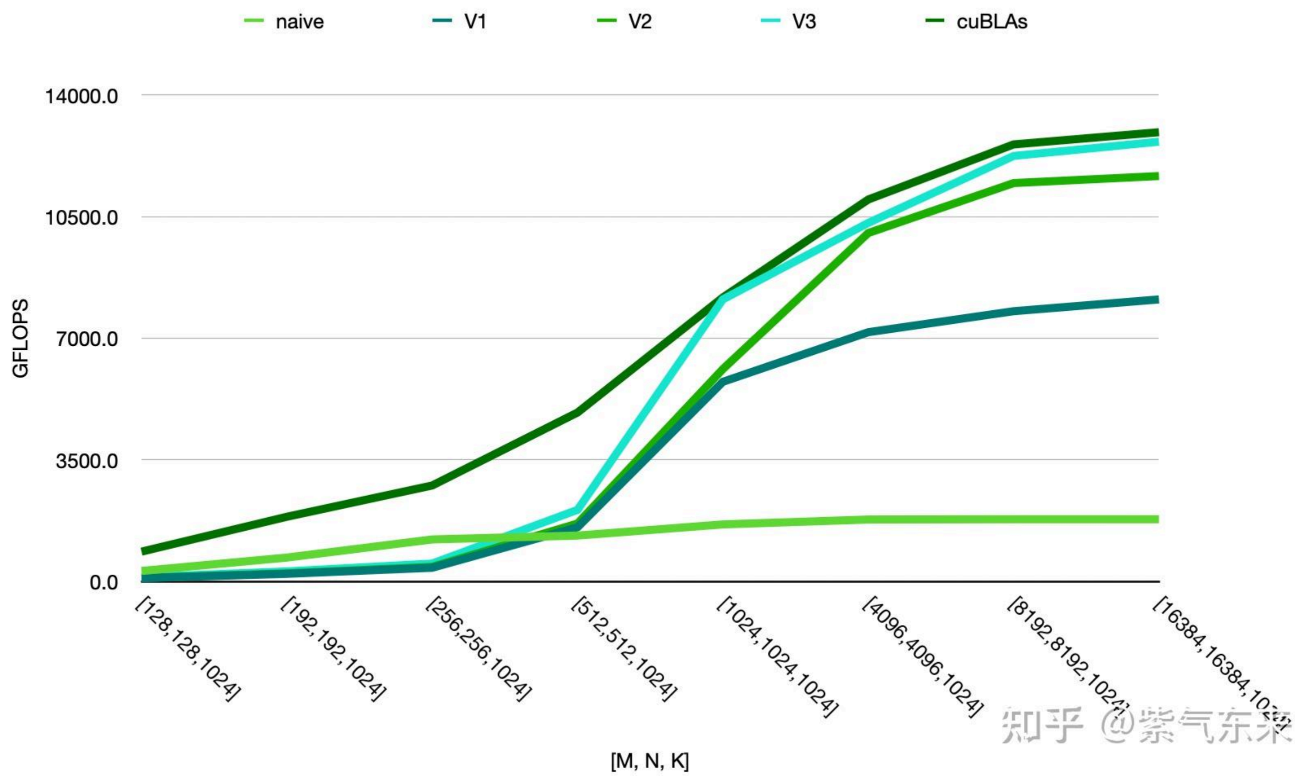
调用方式如下：

```
cublasHandle_t cublas_handle;
cublasCreate(&cublas_handle);
float cublas_alpha = 1.0;
float cublas_beta = 0;
cublasSgemm(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, &cublas_alpha, d_b, N, d_a, K, &cublas_beta, d_c, N);
```

性能如下所示，达到了理论峰值的 82.4%。

```
M N K =    128    128   1024, Time =   0.00002704   0.00003634   0.00010822 s, AVG Performance =   860.0286 Gflops
M N K =    192    192   1024, Time =   0.00003155   0.00003773   0.00007267 s, AVG Performance =  1863.6689 Gflops
M N K =    256    256   1024, Time =   0.00003917   0.00004524   0.00007747 s, AVG Performance =  2762.9438 Gflops
M N K =    384    384   1024, Time =   0.00005318   0.00005978   0.00009120 s, AVG Performance =  4705.0655 Gflops
M N K =    512    512   1024, Time =   0.00008326   0.00010280   0.00013840 s, AVG Performance =  4863.9646 Gflops
M N K =    768    768   1024, Time =   0.00014278   0.00014867   0.00018816 s, AVG Performance =  7567.1560 Gflops
M N K =   1024   1024   1024, Time =   0.00023485   0.00024460   0.00028150 s, AVG Performance =  8176.5614 Gflops
M N K =   1536   1536   1024, Time =   0.00046474   0.00047607   0.00051181 s, AVG Performance =  9452.3201 Gflops
M N K =   2048   2048   1024, Time =   0.00077930   0.00087862   0.00092307 s, AVG Performance =  9105.2126 Gflops
M N K =   3072   3072   1024, Time =   0.00167904   0.00168434   0.00171114 s, AVG Performance = 10686.6837 Gflops
M N K =   4096   4096   1024, Time =   0.00289619   0.00291068   0.00295904 s, AVG Performance = 10994.0128 Gflops
M N K =   6144   6144   1024, Time =   0.00591766   0.00594586   0.00596915 s, AVG Performance = 12109.2611 Gflops
M N K =   8192   8192   1024, Time =   0.01002384   0.01017465   0.01028435 s, AVG Performance = 12580.2896 Gflops
M N K =  12288  12288   1024, Time =   0.02231159   0.02233805   0.02245619 s, AVG Performance = 12892.7969 Gflops
M N K =  16384  16384   1024, Time =   0.03954650   0.03959291   0.03967242 s, AVG Performance = 12931.6086 Gflops
```

由此可以对比以上各种方法的性能情况，可见手动实现的性能已接近于官方的性能，如下：



参考资料：

[1] nicholaswilde：CUDA SGEMM矩阵乘法优化笔记——从入门到cublas

[3] a hgemm tvm schedule

[4] https://www.cnblogs.com/sinkinben/p/16244156.html

[5] Matrix Multiplication CUDA

[6] LustofLife：[CUDA] 并行计算优化策略

[7] https://xmartlabs.github.io/cuda-calculator/

[8] 李少侠：[施工中] CUDA GEMM 理论性能分析与 kernel 优化

[9] CUTLASS: Software Primitives for Dense Linear Algebra at All Levels and Scales within CUDA | NVIDIA On-Demand

[10] 我自己：CUTLASS: Fast Linear Algebra in CUDA C++

[11] 使用 CUTLASS 融合多个 GEMM 实现非凡性能 Use CUTLASS to Fuse Multiple GEMMs to Extreme Performance | NVIDIA On-Demand

> *今夜月明人尽望，不知秋思落谁家。—— 王建《十五夜望月》*