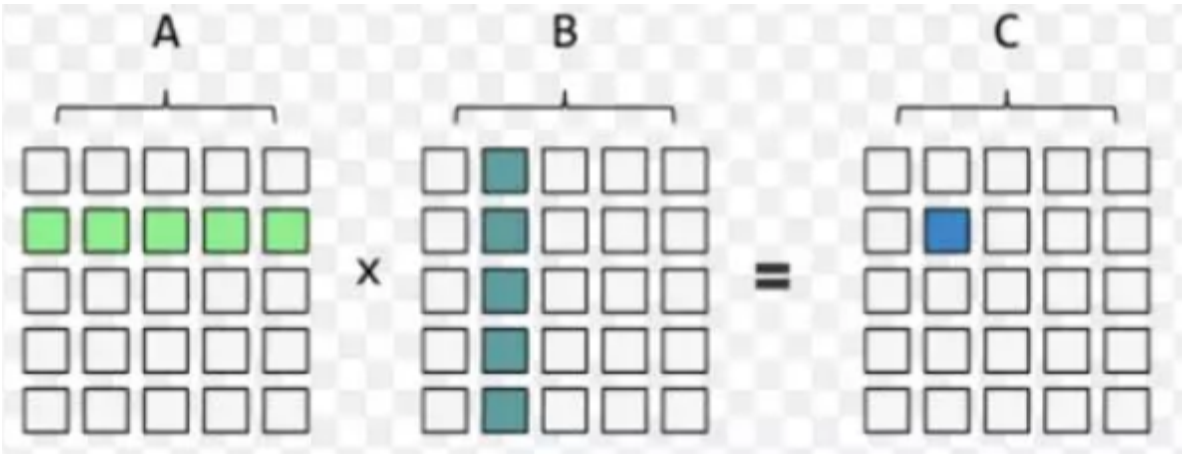
学习笔记2——矩阵相乘与共享内存

1、矩阵乘法 CPU 实现



$A_{m\times k}\cdot B_{k\times n}=C_{m\times n}$

CPU程序通过三层循环实现：

```
void matrixMulCpu(float* fpMatrixA, float* fpMatrixB, float* fpMatrixC, int m,
int n, int k){
    float sum = 0.0f;
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            for(int l = 0; l < k; l++){
                sum += fpMatrixA[i * k + l] * fpMatrixB[l * n + j];
            }
            fpMatrixC[i * n + j] = sum;
            sum = 0.0f;
        }
    }
}
```

- 通过上面CPU代码的实验观察可以看出，总共的计算次数为：$m\times n\times k$
- 时间复杂度为：$O(N^3)$

2、GPU实现矩阵乘法

获得 C 矩阵的计算方法都是相同的，只不过使用的是矩阵 A、B 不同的元素来进行计算，即不同数据的大量相同计算操作，这种计算是特别适合使用GPU来计算，因为GPU拥有大量简单重复的计算单元，通过并行就能极大的提高计算效率。

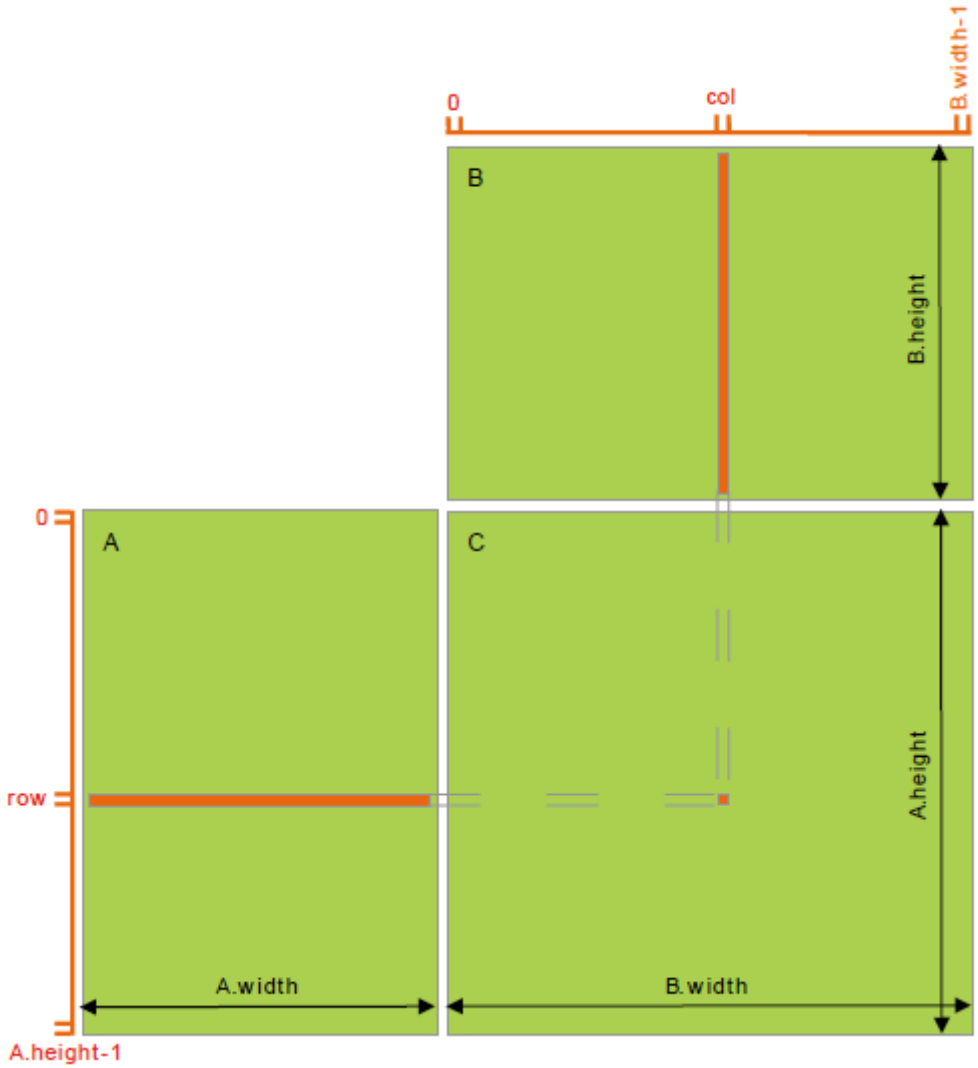
在 GPU 中执行矩阵乘法运算操作：

1. 在 Global Memory 中分别为矩阵 A、B、C 分配存储空间；
2. 由于矩阵 C 中每个元素的计算均相互独立，NVIDIA GPU 采用的 SIMT (单指令多线程)的体系结构来实现并行计算的, 因此在并行度映射中，让每个 thread 对应矩阵 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，

$$gridSize.x \times blockSize.x = ngridSize.x \times blockSize.x = n$$

$$gridSize.y \times blockSize.y = mgridSize.y \times blockSize.y = m$$

CUDA的kernel函数实现如下：



每个 thread 需要执行的 workflow 为：

- 从矩阵 A 中读取一行向量 (长度为width) ==> A[Row * width + i]
- 从矩阵 B 中读取一列向量 (长度为width（图中为height）) ==> B[i * width + Col]
- 对这两个向量做点积运算 (单层 width 次循环的乘累加) ==> A[Row * width + i] * B[i * width + Col]
- 最后将结果写回矩阵 C。 ==> C[Row * width + Col] = Pervalue

```
//核函数的具体实现
__global__ void matMul_GlobalKernel(int *A,int *B,int *C,int width){
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int Col = bx * blockDim.x + tx;
    int Row = by * blockDim.y + ty;

    int perValue = 0;
    for(int i = 0; i < width; i++){
        perValue += A[Row * width + i] * B[i * width + Col];
    }
    C[Row * width + Col] = Pervalue;
}
```

下面来分析一下该 kernel 函数中 A、B、C 三个矩阵对 Global memory 的读取和写入情况：

读取 Global Memory：

- 对于矩阵 C 中每一个元素计算，需要读取矩阵 A 中的一行元素；

  对于矩阵 C 中同一行的 width 个元素，需要重复读取矩阵 A 中同一行元素 width 次；
- 对于矩阵 C 中每一个元素计算，需要读取矩阵 B 中的一列元素；

  对于矩阵 C 中同一列的 width 个元素，需要重复读取矩阵 B 中同一列元素 width 次；

写入 Global Memory：

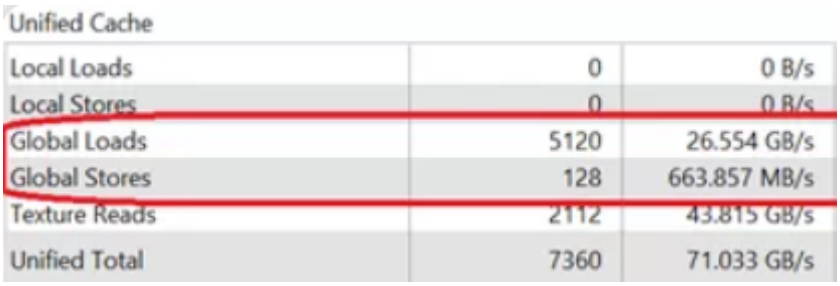
- 矩阵 C 中的所有元素只需写入一次

由此可见：

- 对 A 矩阵重复读取 width 次，共计 $width \times width \times width$ 次 32 bit Global Memory Load操作；
- 对 B 矩阵重复读取 width 次，共计 $width \times width \times width$ 次32 bit Global Memory Load操作；
- 对 C 矩阵共计 $width \times width$ 次 32 bit Global Memory Store操作。

在上述分析中是将每一个 32 bit 元素 (或者说每个thread)对 Global memory 的访问 (access)独立对待的；但实际情况是如此吗?

假设矩阵规模为 width=32，执行配置 blockSize=(32, 32, 1)，gridSize=(1, 1, 1)，使用上述的 Kernel 函数进行计算，在 NVVP 中 Memory Bandwidth Analysis 结果如下：



Unified Cache		
Local Loads	0	0 B/s
Local Stores	0	0 B/s
Global Loads	5120	26.554 GB/s
Global Stores	128	663.857 MB/s
Texture Reads	2112	43.815 GB/s
Unified Total	7360	71.033 GB/s

按照前面的计算方式，Global Memory Load 次数为$32 \times 32 \times 32 \times 2 = 65536$次，Store 次数为 1024 次；而 NVVP 显示的读取次数为 5120 次；写入次数为 128 次, 这到底是为什么呢?

别有洞天之 Warp

GPU 编程中最重要的概念之一是 warp，每个 warp 包含 32 个 thread，而 GPU 的指令发射是以 warp 为最小单元的。当 warp 中的一条指令发射一次，称为 1 次 “transaction”，重复发射一次, 称为 1 次 “reply”。

对于 Global Memory 的访问，warp 内的指令需要几次 transaction，或者说是否会发生 reply，取决于地址对齐及可合并访问的情况。

Global Memory 的读/写访问均是以 32 Byte 为单元的，称为 1 个 segment，即 1 transaction 可访问 32 Byte 数据 (假设 L1 cache 为 non-caching)。假设 1 个 warp 中的每个 thread 需要访问 1 个 32 bit (=4 Byte)数据，且访问地址是 32 Byte 对齐的，则总共需要访问 4 个 segments，即产生 4 次 transaction (即 3 次 reply)指令发射。

接下来重新分析矩阵乘法中Global Memory访问的情况：

- `Global Memory Load`：对于 1 个 warp 中的 32 个 thread，在每 1 次循环中，需要读取矩阵 A 同一个元素 (1 次 transaction)，以及矩阵 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 (注意，并不是前文的 32+32 次)。K 次循环总共需要 k×5次 transactions。对于 $width \times width$ 个 thread, 共有 $m \times n \div 32$个 warp，总共的 Global Memory Load Transaction 数目为：$width \times width \div 32 \times k \times 5$ (注意，并不是前文的 $width \times width \times width \times 2$次).
- `Global Memory Store`：矩阵 C 的写入过程中，每个 warp 中的 32 thread 可连续写入 32 个 32 bit 元素 (4次 transaction)，对于$width \times width$个 thread，共有$width \times width \div 32$个 warp，总共的 Global Memory Store Transaction 数目为：$width \times width \div 32 \times 4$次 transaction。

对于前面验证矩阵, 其规模为width=32, 执行配置blockSize=(32,32,1), gridSize=(1,1,1),

- `Global Memory Load Transaction`数目为: width×width÷32×width×5=32×32÷32×32×5=5120

分析结果与 NVVP 中 GPU 实际执行结果是完全吻合的。

width*width*width\Blocksize 对计算性能的影响

width*width*width\blocksize	8*8	16*16	32*32
8*8*8	0.197	NA	NA
16*16*16	1.890	1.921	NA
32*32*32	15.675	12.810	13.409
64*64*64	117.014	101.927	71.828
128*128*128	633.439	649.165	346.142
256*256*256	1125.477	1268.391	**1511.790**
512*512*512	1301.495	**1332.036**	1391.029
1024*1024*1024	**1386.940**	1295.104	1361.126
2028*2048*2048	1215.287	1144.767	1237.371
4096*4096*4096	799.716	1091.926	1153.420

结果分析:

- 随着矩阵规模增大，计算性能不断提升，到达峰值后又略有下降。在矩阵规模较小时，由于block数量不够多，无法填满所有SM单元，此时的性能瓶颈为Latency Bound（由于低Occupancy导致GPU计算资源的利用率低，延迟无法被很好的隐藏）；随着矩阵规模增加，block数量增加，每个SM中的active warps数量随之增大，此时Latency不再是性能瓶颈，转而受限于Memory Bound（过多的高延迟、低带宽的全局内存访问），在无法提升memory访问效率的情况下，性能无法进一步提升；
- 不同的blockSize对性能的影响不大（这里仅限于讨论8*8、16*16、32*32三种情况）。究其原因，是因为选择的几种block维度设计（每行分别有8/16/32个thread），对1个warp内访问Global Memory时（Load或Store）transaction的数量没有变化。

3、Shared Memory 优化矩阵乘法

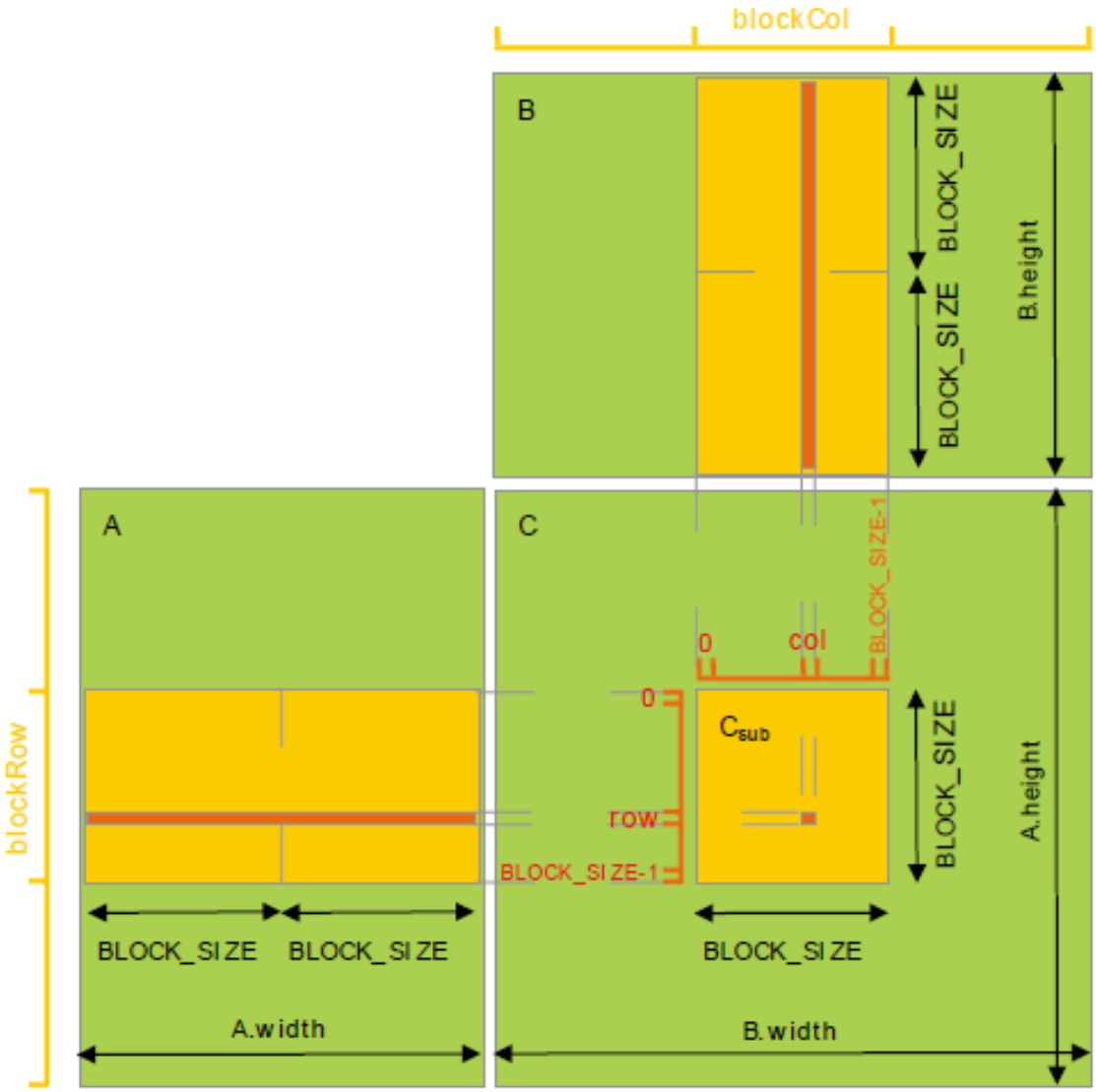
虽然 warp 内对 Global Memory 的访问均已最大的实现了合并访问，但在 A、B 矩阵的读取操作中仍然有很多重复访问，例如：

对于矩阵 A 的读取操作，通过合并访问（32 个 thread 访问 Global Memory 的同一地址，合并为一次访问），实际重复读取次数是(n/32)；

对于矩阵 B 的读取操作，通过合并访问（8 个 thread 访问 32 Byte 数据可合并为一次），实际重复读取次数为(m/8)次。

在不改变这种数据读取方式的前提下又如何优化性能呢?

在GPU中除了 Global Memory 还有 Shared Memory，这部分 Memory 是在芯片内部的，相较于 Global Memory 400~600 个时钟周期的访问延迟，Shared Memory 延时小 20-30 倍、带宽高 10 倍，具有低延时、高带宽的特性。因此性能优化的问题可以转变为如何利用 Shared Memory 代替 Global Memory 来实现数据的重复访问。

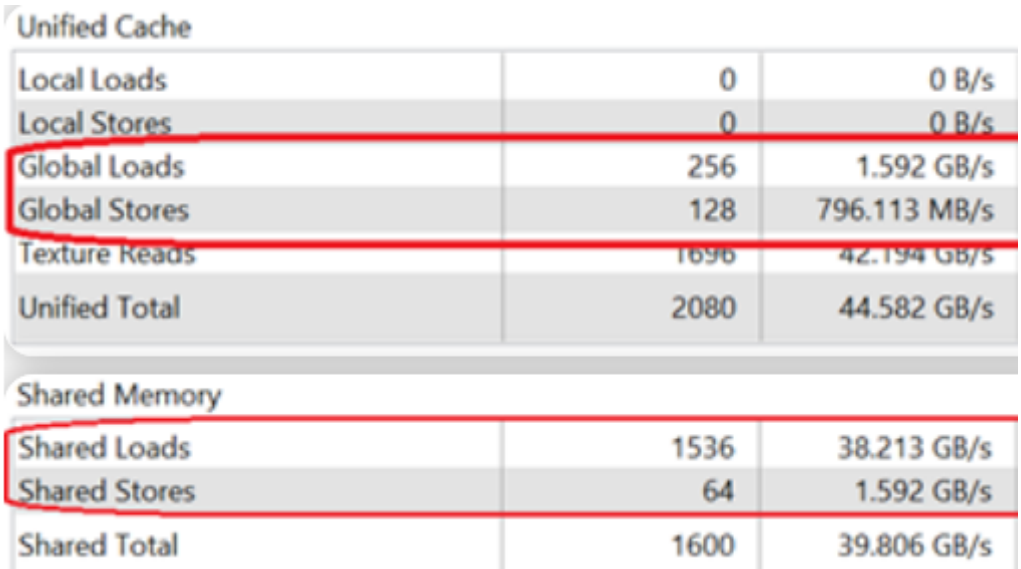


如上图所示，使用 Shared Memory 优化 Global Memory 访问的基本思想是充分利用数据的局部性。

让一个 block 内的 thread 先从 Global Memory 中读取子矩阵块数据（大小为 BLOCK_SIZE × BLOCK_SIZE）并写入 Shared Memory 中；在计算时，从 Shared Memory 中（重复）读取数据做乘累加，从而避免每次都到 Global 中取数据带来的高延迟影响。接下来让子矩阵块分别在矩阵 A 的行向以及矩阵 B 的列向上滑动，直到计算完所有 width 个元素的乘累加。使用 Shared Memory 优化后的 kernel 代码如下所示：

```
//核函数的具体实现
__global__ void matmul_ShareMemory(int *M,int *N,int *P,int width){
    __shared__ float Mds[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Nds[BLOCK_SIZE][BLOCK_SIZE];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int Col = bx * BLOCK_SIZE + tx;
    int Row = by * BLOCK_SIZE + ty;

    int Pervalue = 0;
    //有多少个BLOCK_SIZE，每个循环计算一个块的大小
    for(int i = 0;i < width / BLOCK_SIZE;i++){
        Mds[ty][tx] = M[Row * width + (i * BLOCK_SIZE + tx)];
        Nds[ty][tx] = N[Col + (i * BLOCK_SIZE + ty) * width];
        __syncthreads();

        //BLOCK_SIZE相乘
        for(int k = 0;k < BLOCK_SIZE;k++)
            Pervalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    P[Row * width + Col] = Pervalue;
}
```

- 1 个 block 即看做 1 个子矩阵 C，且为方阵；
- 读取子矩阵 A 和子矩阵 B 的 Shared Memory 的大小均等于子矩阵 C 的维度大小；
- 子矩阵 A 在矩阵 A 的行向上移动 width/BLOCK_SIZE 次，子矩阵 B 在矩阵 B 的列向上移动 width / BLOCK_SIZE 次；
- 每个 thread 的计算过程，由原来的单层 k 次循环，变为了两层循环：外层循环次数为 width / BLOCK_SIZE（假设能够整除），其任务是从 Global Memory 读取数据到 Shared Memory；内存循环次数为 BLOCK_SIZE，其任务是读取 Shared Memory 中的数据做乘累加计算；
- 有 2 次 `__syncthreads()` 操作：第一次表示同步 Shared Memory 中数据写入之后，在进入计算之前，保证block内所有Shared Memory中数据已更新；第二次表示同步计算之后以及 Shared Memory 写入之前，保证 block 内所有 thread 的计算已完成，可以进行 Shared Memory 的数据更新。



Unified Cache		
Local Loads	0	0 B/s
Local Stores	0	0 B/s
Global Loads	256	1.592 GB/s
Global Stores	128	796.113 MB/s
Texture Reads	1696	42.194 GB/s
Unified Total	2080	44.582 GB/s
Shared Memory		
Shared Loads	1536	38.213 GB/s
Shared Stores	64	1.592 GB/s
Shared Total	1600	39.806 GB/s

Shared Memory 性能分析

width* width* width\blocksize	8*8	16*16	32*32
8*8*8	0.229	–	–
16*16*16	1.592	1.878	–
32*32*32	12.242	15.269	15.334
64*64*64	102.352	116.184	95.163
128*128*128	501.691	730.729	486.994
256*256*256	1560.865	2001.799	2194.816
512*512*512	**1758.115**	2993.400	3004.314
1024*1024*1024	1618.022	**3120.376**	3821.940
2048*2048*2048	1213.414	2893.394	3800.826
4096*4096*4096	875.712	2708.951	**3824.857**

- 随着矩阵规模增大，计算性能不断提升，到达峰值后又略有下降——这与未优化前使用 Global Memory 时的性能分析结果一致；
- 不同 blockSize 对性能影响很大。外层循环中从 Global Memory 读取数据写入到 Shared Memory 时，无论是读取 A 矩阵或 B 矩阵，当 warp 的组织形式（行x列）为 8x4 或 16x2，对于 Global Memory 的 load 操作，均可实现 32B 的合并访问（8 thread x 4B 及 4 次 transactions）。而对于 Shared Memory 的 Store 操作，则会出现 Bank Conflict 导致 reply 的发生；同样地，内层循环中读取 Shared Memory 时，当 warp 的组织形式为 8x4 或 16x2 时，则会出现 Bank Conflict，导致 Shared memory 读取时的 reply，从而影响性能。

4、Register 优化矩阵乘法

前面的算法设计中，每个线程只计算了矩阵 C 中的一个元素，每个线程每个内层循环需要从子矩阵 A 和子矩阵 B 中各读取一个 4 Byte 的元素（共取 8 Byte 数据执行2次浮点运算），实际上可以让每个线程读取一组 Shared Memory 数据后（放入寄存器中），计算更多的元素，从而减少 Shared Memory 的访问。

```
// using ILP 2 to improve the performance
__global__ void matrixMulSharedILPkernel(float* A, float* B, float* C, int
width){
    int row = blockIdx.y * blockDim.y * 2 + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float val[2] = {0.0f};

    __shared__ float shTileA[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float shTileB[BLOCK_SIZE][BLOCK_SIZE];

    int iter = (width + BLOCK_SIZE - 1) / BLOCK_SIZE;
    for(int i = 0; i < iter; i++){
        // read data from global memory to shared memory
        shTileA[threadIdx.y][threadIdx.x]=A[row *
width+i*BLOCK_SIZE+threadIdx.x];
        shTileA[threadIdx.y+16]
[threadIdx.x]=A[(row+16)*width+i*BLOCK_SIZE+threadIdx.x];
        shTileB[threadIdx.y]
[threadIdx.x]=B[(i*BLOCK_SIZE+threadIdx.y)*width+col];
        shTileB[threadIdx.y+16]
[threadIdx.x]=B[(i*BLOCK_SIZE+threadIdx.y+16)*width+col];
        __syncthreads();

        for(int j = 0; j < BLOCK_SIZE; j++){
            val[0] += shTileA[threadIdx.y][j] * shTileB[j][threadIdx.x];
            val[1] += shTileA[threadIdx.y + 16][j] * shTileB[j][threadIdx.x];
        }
        __syncthreads();
    }
    C[row * width + col] = val[0];
    C[(row + 16) * width + col] = val[1];
}
```

注意， kernel launch 时的 blocksize 需要变化为：blockSize.y = BLOCK_SIZE / 2。而 gridSize 不变。

上面的 kernel 函数中，注意观察内层循环：让 1 个 thread 分别从子矩阵 A 中读取 2 个数据，从子矩阵 B 中读取 1 个数据（注意2次取数据是同一地址！），然后同时计算2个元素 val[0] 和 val[1]。此时，通过读取 4B*3 个数据，实现了2次乘加共4次计算。减少了 shared memory 中子矩阵B一半的数据访问。

下面详细分析一下上述代码对 Shared Memory 的访问情况：

Shared Memory Store：每 1 次外层循环中对矩阵 A 和矩阵 B 有 2 次 load，总的 thread 个数减少了一半。但是实际上总的 load transactions 次数没有变化（假设 no bank conflict）。

Shared Memory Load：在每 1 次内层循环中，1 个 warp 内的 32 个 thread 需要从 shTileA 读取同 2 个元素，需要 2 次 Shared Memory Load Transactions，再从 shTileB 中读取连续的 32 个元素（假设没有 Bank Conflict，需要1次 Shared Memory Load Transactions）（注意val[0]和val[1]的计算中，shTileB 的地址是一样的），即总共需要 3 次 Shared Memory Load Transactions。$width \div BLOCK\_SIZE \times BLOCK\_SIZE$ 次循环总共需要 $width \times 3$ 次 Shared Memory Load Transactions。对于 $width \times width \div 2$ 个 threads，共有$width \times width \div 2 \div 32$个 warp，总共的 Shared Memory Load Transactions 数目为：$(width \times width \div 2) \div 32 \times width \times 3$。对比优化前的 Shared Memory Load Transactions 数目 $width \times width \div 32 \times width \times 2$。

当然，我们还可以继续对 kernel 函数进行优化，让每个 thread 计算的元素个数从 2 个提高到 4/8/16/32 个，对比测试结果如下（m=n=k=1024, BLOCK_SIZE=32*32）：

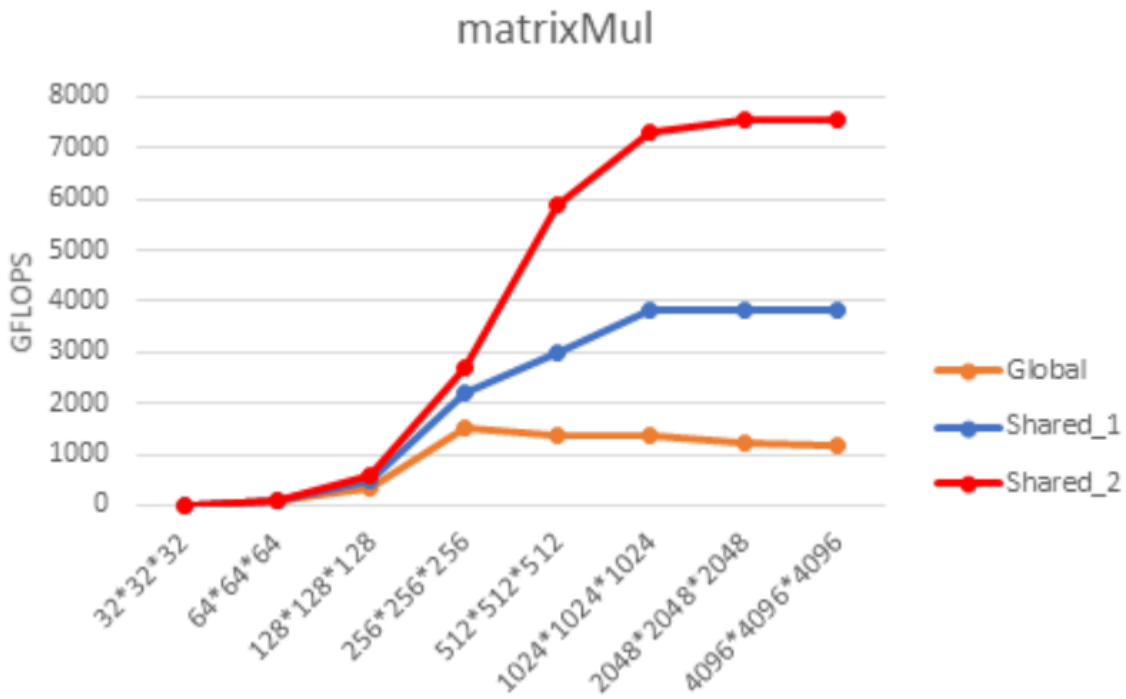
每个thread计算元素个数	计算性能
1	3.670 T-FLOPS
2	4.800 T-FLOPS
4	6.019 T-FLOPS
8	6.772 T-FLOPS
16	6.665 T-FLOPS
32	5.323 T-FLOPS

此时测出的峰值性能相比于优化前提升了约 82.5%。但是，为什么继续增大每个 thread 计算元素个数后，性能反而下降呢?

一方面是继续增大该指标后，每个 block 中 thread 的个数是在减少的，但是每个 block 中需要的 Shared Memory 数量没有减少。这将导致由于 Block 总数受限，降低 SM 中的 active threads 数量，即降低了 Occupancy。另外，每个 thread 计算更多元素会使用更多的 Registers。而每个 thread 中 Registers 的个数又会反过来影响 SM 中 active threads 的个数，进而影响 Occupancy。

具体 Shared Memory 或者 Registers 哪个会影响 Occupancy，取决于哪个指标先到达阈值。在没有换来同等指令集并行的情况下，Occupancy 的减少会导致计算性能受限。

终极性能对比



注：Shared_2代码中，每个thread计算 8 个元素。

上图为优化前后 3 个版本CUDA程序的性能差异，从图中可以得出：

- 在句子规模为 $4K \times 4K \times 4K$ 的情况下，第三个版本的方法达到的峰值性能超过 7T；
- 随着矩阵规模的增加，计算性能也逐渐增加；
- 通过利用 Shared Memory 和寄存器能有效的降低 IO 带宽对性能的影响，从而更加高效的利用 GPU 的硬件计算资源。

完整代码如下：

```
#include <iostream>
#define BLOCK_SIZE 5

//基础版本
__global__ void matMul_GlobalKernel(int *A,int *B,int *C,int width){
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int Col = bx * blockDim.x + tx;
    int Row = by * blockDim.y + ty;

    int perValue = 0;
    for(int i = 0; i < width; i++){
        perValue += A[Row * width + i] * B[i * width + Col];
    }
    C[Row * width + Col] = Pervalue;
}

//优化1
__global__ void matmul_ShareMemory(int *M,int *N,int *P,int width){
    __shared__ float Mds[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Nds[BLOCK_SIZE][BLOCK_SIZE];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int Col = bx * BLOCK_SIZE + tx;
    int Row = by * BLOCK_SIZE + ty;

    int Pervalue = 0;
    //有多少个BLOCK_SIZE，每个循环计算一个块的大小
    for(int i = 0;i < width / BLOCK_SIZE;i++){
        Mds[ty][tx] = M[Row * width + (i * BLOCK_SIZE + tx)];
        Nds[ty][tx] = N[Col + (i * BLOCK_SIZE + ty) * width];
        __syncthreads();

        //BLOCK_SIZE相乘
        for(int k = 0;k < BLOCK_SIZE;k++)
            Pervalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    P[Row * width + Col] = Pervalue;
}

//优化2
__global__ void matrixMul_SharedILPkernel(float* A, float* B, float* C, int
width){
    int row = blockIdx.y * blockDim.y * 2 + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float val[2] = {0.0f};
```

```
    __shared__ float shTileA[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float shTileB[BLOCK_SIZE][BLOCK_SIZE];

    int iter = (width + BLOCK_SIZE - 1) / BLOCK_SIZE;
    for(int i = 0; i < iter; i++){
        // read data from global memory to shared memory
        shTileA[threadIdx.y][threadIdx.x]=A[row *
width+i*BLOCK_SIZE+threadIdx.x];
        shTileA[threadIdx.y+16]
[threadIdx.x]=A[(row+16)*width+i*BLOCK_SIZE+threadIdx.x];
        shTileB[threadIdx.y]
[threadIdx.x]=B[(i*BLOCK_SIZE+threadIdx.y)*width+col];
        shTileB[threadIdx.y+16]
[threadIdx.x]=B[(i*BLOCK_SIZE+threadIdx.y+16)*width+col];
        __syncthreads();

        for(int j = 0; j < BLOCK_SIZE; j++){
            val[0] += shTileA[threadIdx.y][j] * shTileB[j][threadIdx.x];
            val[1] += shTileA[threadIdx.y + 16][j] * shTileB[j][threadIdx.x];
        }
        __syncthreads();
    }
    C[row * width + col] = val[0];
    C[(row + 16) * width + col] = val[1];
}


int main(){
    const int x=15, y=15, z=15;
    int a[x][y], b[y][z], c[x][z];
    int *a_gpu, *b_gpu, *c_gpu;

    // step 1: init matrix data
    for(int i = 0; i < y; ++i){
        for(int j = 0; j < x; ++j){
            a[j][i] = 3;
        }
    }

    for(int i = 0; i < z; ++i){
        for(int j = 0; j < y; ++j){
            b[j][i] = 2;
        }
    }

    printf("\nmatrix a:\n");
    for(int i = 0; i < x; ++i){
        for(int j = 0; j < y; ++j){
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }

    printf("\nmatrix b:\n");
    for(int i = 0; i < y; ++i){
        for(int j = 0; j < z; ++j){
            printf("%d ", b[i][j]);
        }
```

```
        printf("\n");
    }

    // step 2: copy matrix to device
    cudaMalloc((void **)&a_gpu, x * y * sizeof(int));
    cudaMalloc((void **)&b_gpu, y * z * sizeof(int));
    cudaMalloc((void **)&c_gpu, x * z * sizeof(int));

    cudaMemcpy(a_gpu, a, x * y * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(b_gpu, b, y * z * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(c_gpu, c, x * z * sizeof(int), cudaMemcpyHostToDevice);

    // step 3: run kernel function
    const int numsThread = 5;

    dim3 gridSize(x / numsThread, y / numsThread);

    dim3 blockSize(numsThread, numsThread);
    //matMul_GlobalKernel<<<gridSize, blockSize>>>(a_gpu, b_gpu, c_gpu, x);
    matmul_ShareMemory<<<gridSize, blockSize>>>(a_gpu, b_gpu, c_gpu, x);
    //matrixMul_SharedILPkernel<<<gridSize, blockSize>>>(a_gpu, b_gpu, c_gpu,
x);

    // step 4: download result from device
    cudaMemcpy(c, c_gpu, x * z * sizeof(int), cudaMemcpyDeviceToHost);

    printf("\nmatrix a * matrix b = :\n");
    for(int a = 0; a < z; ++a){
        for(int b = 0; b < x; ++b){
            printf("%d ", c[b][a]);
        }
        printf("\n");
    }

    return 0;
}
```

CmakeList.txt内容如下：

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)
PROJECT(mat_multiply)
FIND_PACKAGE(CUDA REQUIRED)
CUDA_ADD_EXECUTABLE(mat_multiply main.cu)
TARGET_LINK_LIBRARIES(mat_multiply)
```

Linux系统执行指令如下：

```
$ mkdir build && cd build
$ make -j4
$ ./mat_multiply
```

参考

[1]. 矩阵乘法的 CUDA 实现、优化及性能分析