中山大学计算机院本科生实验报告

（2024 学年春季学期）

课程名称：并行程序设计　　批改人：

实验	10-CUDA 并行矩阵乘法	专业（方向）	计算机科学与技术
学号	21307174	姓名	刘俊杰
Email	liujj255@mail2.sysu.edu.cn	完成日期	2024/5/29

1. 实验目的

CUDA 实现并行通用矩阵乘法，并通过实验分析不同线程块大小，访存方式、数据/任务划分方式对并行性能的影响。

输入：$\boldsymbol{m},\boldsymbol{n},\boldsymbol{k}$三个整数，每个整数的取值范围均为[128, 2048]

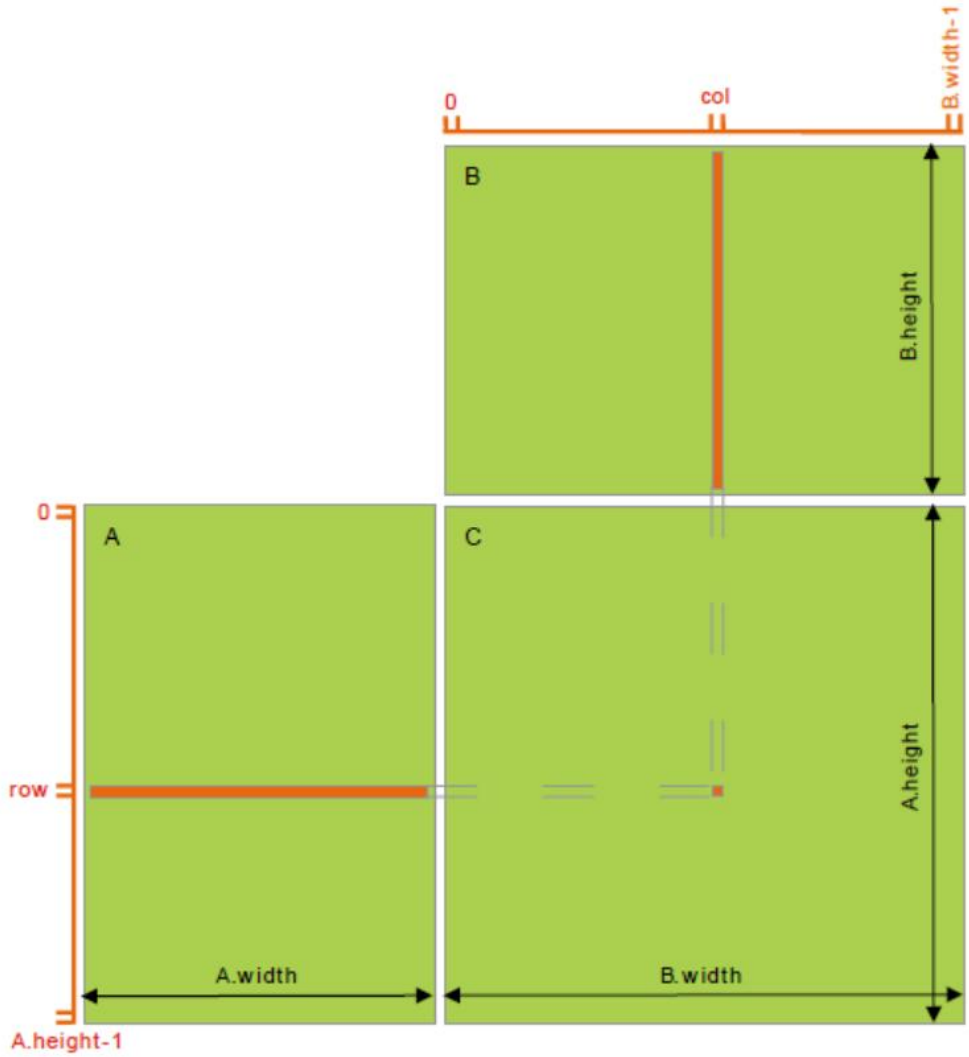
问题描述：随机生成$\boldsymbol{m}\times\boldsymbol{n}$的矩阵$\boldsymbol{A}$及$\boldsymbol{n}\times\boldsymbol{k}$的矩阵$\boldsymbol{B}$，并对这两个矩阵进行矩阵乘法运算，得到矩阵$\boldsymbol{C}$.

输出：$\boldsymbol{A},\boldsymbol{B},\boldsymbol{C}$三个矩阵，及矩阵计算所消耗的时间$\boldsymbol{t}$。

要求：使用 CUDA 实现并行矩阵乘法，分析不同线程块大小，矩阵规模，访存方式，任务/数据划分方式，对程序性能的影响。

2. 实验过程和核心代码

2.1 全局内存 每个线程负责计算 C 中一个位置的值



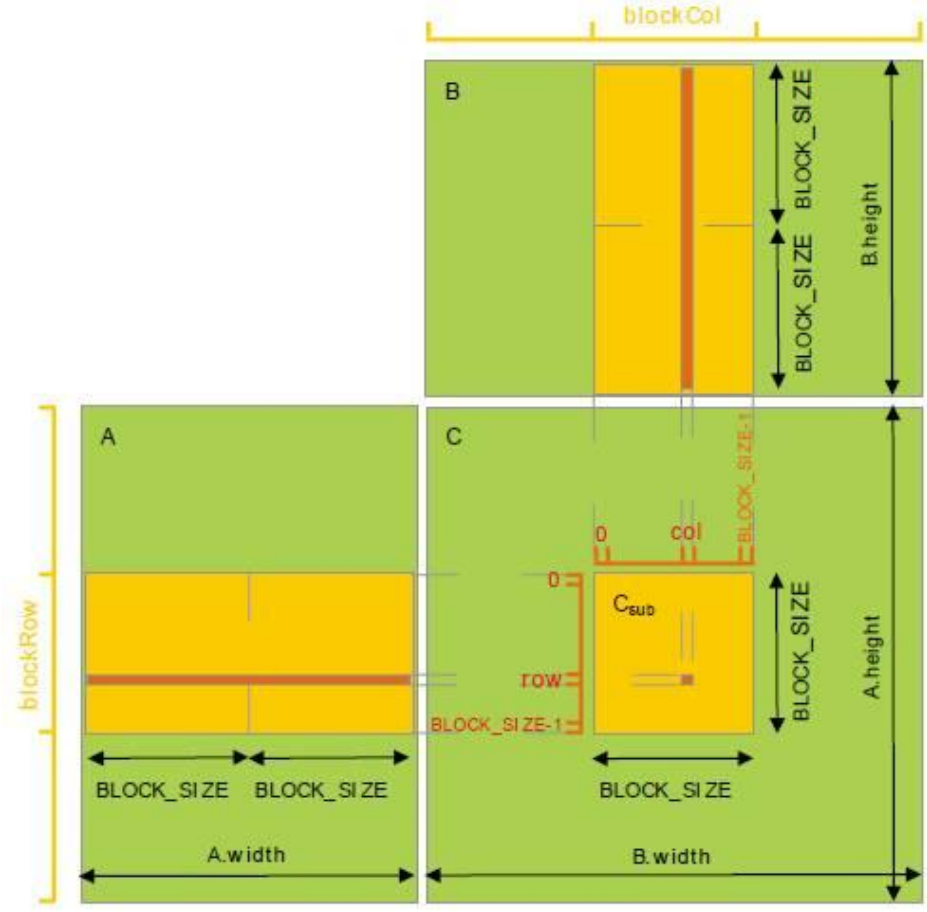
2.1.1 实验思路

①首先根据 blockID、blockDim 和 threadIDx 计算出每个线程在全局上的索引，同时也是其负责计算 C 位置上的值的索引。
②检查负责的索引是否在矩阵 C 的范围内。
③若在，则该线程计算出该位置的值，并赋值给 C 对应的位置。

2.1.2 核心代码
核心代码如下：

```cpp
// Kernel函数，执行矩阵乘法运算
__global__ void matrixMultiply(float *A, float *B, float *C, int N) {
    // 计算线程在矩阵C中的位置
    int col = blockIdx.y * blockDim.y + threadIdx.y;
    int row = blockIdx.x * blockDim.x + threadIdx.x;

    // 检查是否超出矩阵大小
    if (row < N && col < N) {
        // 计算C[row][col]的值
        float sum = 0.0;
        for (int k = 0; k < N; ++k) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

2.2 共享内存



2.2.1 实验思路

使用共享内存优化的基本思想是利用数据的局部性，通过将数据从全局内存读取到共享内存中，减少每次访问全局内存所需的延迟时间。

在矩阵乘法中，可以将每个线程块所需要的矩阵子块数据（例如 BLOCK_SIZE × BLOCK_SIZE 大小的子矩阵）从全局内存中读取到共享内存中。然后，每个线程在计算时可以从共享内存中读取数据，避免了每次计算都需要从全局内存中读取数据的高延迟。

具体实现时，可以将矩阵 A 的子块在行向上滑动，将矩阵 B 的子块在列向上滑动，以便每个线程块中的每个线程都能够计算完所有元素的乘累加。这样，每个线程块中的线程都可以重复利用共享内存中的数据，避免了频繁访问全局内存的开销。

2.2.2 核心代码

```
// Kernel函数，执行矩阵乘法运算
__global__ void matrixMultiply_shared(float *A, float *B, float *C, int N) {
    // 声明共享内存数组，用于存储部分矩阵数据
    __shared__ float shared_A[TILE_SIZE][TILE_SIZE];
    __shared__ float shared_B[TILE_SIZE][TILE_SIZE];

    // 计算线程在矩阵C中的位置
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float sum = 0.0;

    // 循环计算每个子矩阵的乘积并累加到结果中
    for (int t = 0; t < N / TILE_SIZE; ++t) {
        // 将子矩阵A和B从全局内存复制到共享内存中
        shared_A[threadIdx.y][threadIdx.x] = A[row * N + t * TILE_SIZE + threadIdx.x];
        shared_B[threadIdx.y][threadIdx.x] = B[(t * TILE_SIZE + threadIdx.y) * N + col];

        // 等待所有线程将数据加载到共享内存中
        __syncthreads();

        // 计算矩阵乘积的部分和
        for (int k = 0; k < TILE_SIZE; ++k) {
            sum += shared_A[threadIdx.y][k] * shared_B[k][threadIdx.x];
        }

        // 等待所有线程完成当前子矩阵的计算
        __syncthreads();
    }

    // 将计算结果写入矩阵C
    if (row < N && col < N) {
        C[row * N + col] = sum;
    }
}
```

3. 实验结果

3.1 验证实验正确性

运行结果：

```
Matrix A:
0.84 0.39 0.63 0.78
0.05 0.30 0.21 0.61
0.24 0.17 0.54 0.75
0.58 0.73 0.82 0.96

Matrix B:
0.58 0.21 0.40 0.31
0.70 0.14 0.14 0.38
0.82 0.88 0.83 0.45
0.73 0.15 0.14 0.66

Matrix C:
1.85 0.90 1.02 1.21
0.86 0.33 0.33 0.63
1.25 0.66 0.67 0.88
2.22 1.09 1.15 1.46

Time for computing C(global): 0.02 ms
Kernel for matrixMultiply_shared:
  Block dimensions: (2, 2)
  Grid dimensions: (2, 2)
Matrix A:
0.84 0.39 0.63 0.78
0.05 0.30 0.21 0.61
0.24 0.17 0.54 0.75
0.58 0.73 0.82 0.96

Matrix B:
0.58 0.21 0.40 0.31
0.70 0.14 0.14 0.38
0.82 0.88 0.83 0.45
0.73 0.15 0.14 0.66

Matrix C:
1.85 0.90 1.02 1.21
0.86 0.33 0.33 0.63
1.25 0.66 0.67 0.88
2.22 1.09 1.15 1.46
```

可以看到使用共享内存和全局内存计算的结果都是正确的。

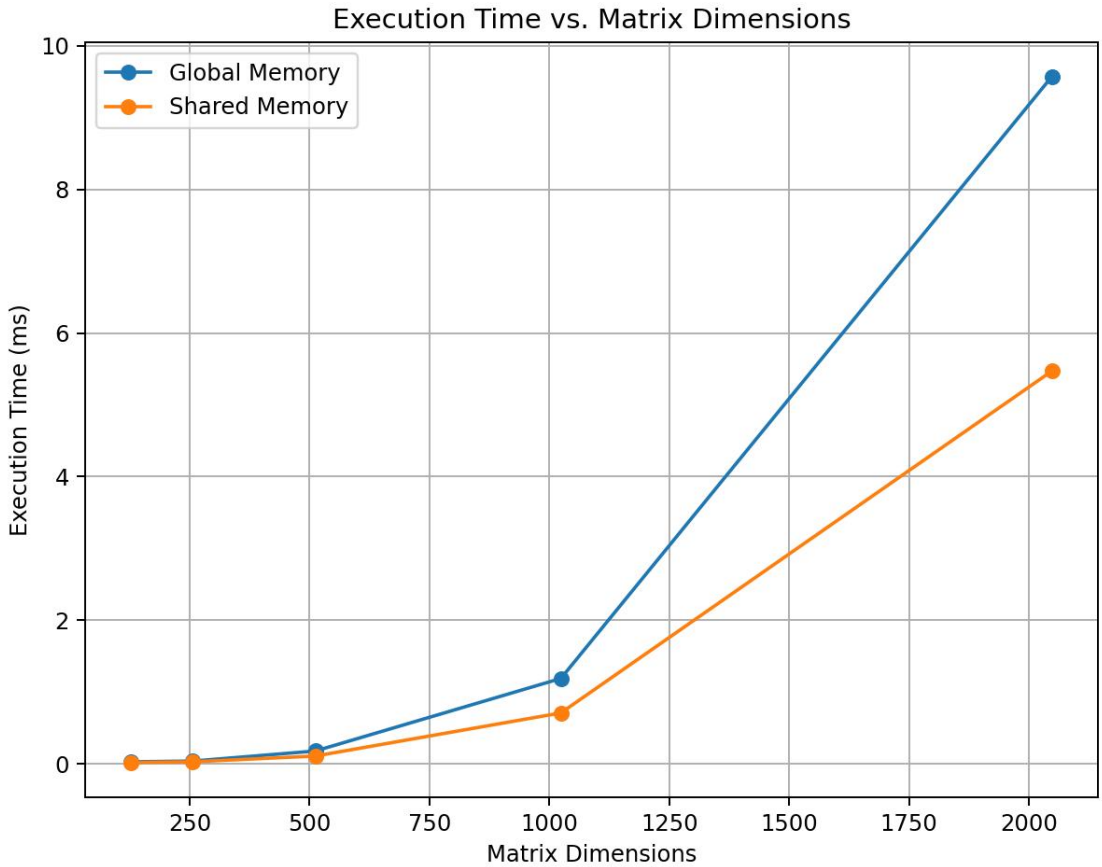
3.2 比较共享内存与全局内存对程序性能的影响(16 X 16 的线程块大小)

运行结果：

```
jovyan@jupyter-21307174:~$ nvcc matrixMultiply_test.cu -o run
jovyan@jupyter-21307174:~$ ./run
Matrix dimensions: 128 x 128

Time for computing C(global): 0.03 ms
---------------------------

Time for computing C(shared): 0.02 ms
---------------------------
------------------------------------------------------------
Matrix dimensions: 256 x 256

Time for computing C(global): 0.04 ms
---------------------------

Time for computing C(shared): 0.03 ms
---------------------------
------------------------------------------------------------
Matrix dimensions: 512 x 512

Time for computing C(global): 0.18 ms
---------------------------

Time for computing C(shared): 0.11 ms
---------------------------
------------------------------------------------------------
Matrix dimensions: 1024 x 1024

Time for computing C(global): 1.19 ms
---------------------------

Time for computing C(shared): 0.71 ms
---------------------------
------------------------------------------------------------
Matrix dimensions: 2048 x 2048

Time for computing C(global): 9.57 ms
---------------------------

Time for computing C(shared): 5.47 ms
---------------------------
```

对比结果：

Matrix Dimensions	Global Memory (ms)	Shared Memory (ms)
128 x 128	0.03	0.02
256 x 256	0.04	0.03
512 x 512	0.18	0.11
1024 x 1024	1.19	0.71
2048 x 2048	9.57	5.47

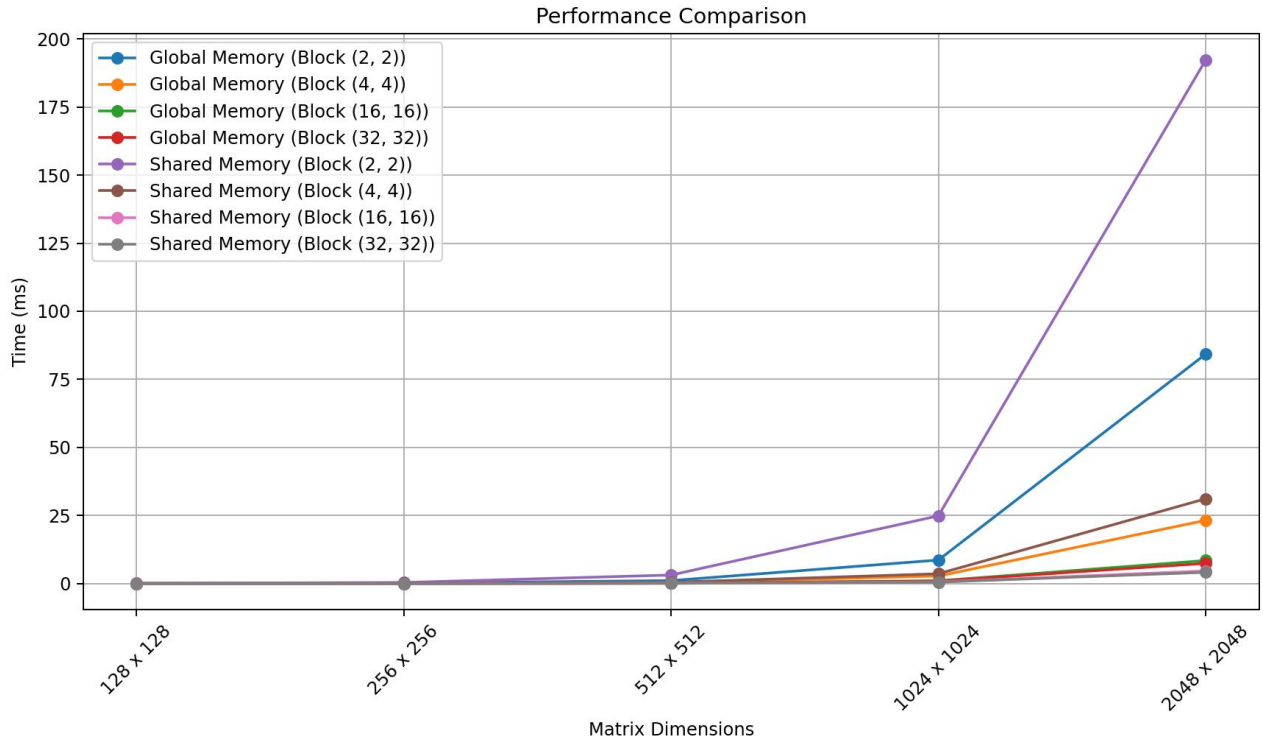


可以从实验结果中看到,使用共享内存的运行时间小于使用全局内存的运行时间,同时随着矩阵规模的增大,共享内存的加速效果越明显。

3.3 不同大小的线程块对程序性能的影响

实验结果对比:

Matrix Dimensions	Block Dimensions	Global Memory (ms)	Shared Memory (ms)
128 x 128	(2, 2)	0.04	0.07
256 x 256	(2, 2)	0.16	0.41
512 x 512	(2, 2)	1.10	3.14
1024 x 1024	(2, 2)	8.64	24.88
2048 x 2048	(2, 2)	84.26	192.24
128 x 128	(4, 4)	0.03	0.02
256 x 256	(4, 4)	0.06	0.08
512 x 512	(4, 4)	0.38	0.47
1024 x 1024	(4, 4)	2.78	3.61
2048 x 2048	(4, 4)	23.22	31.15
128 x 128	(16, 16)	0.02	0.02
256 x 256	(16, 16)	0.03	0.03
512 x 512	(16, 16)	0.15	0.11
1024 x 1024	(16, 16)	1.02	0.61
2048 x 2048	(16, 16)	8.46	4.66
128 x 128	(32, 32)	0.03	0.02
256 x 256	(32, 32)	0.03	0.02
512 x 512	(32, 32)	0.15	0.09
1024 x 1024	(32, 32)	0.93	0.53
2048 x 2048	(32, 32)	7.44	4.15



可以看到使用不同的线程块大小对程序性能的影响是显著的,使用过小的线程块，程序的运行时间长,甚至当线程块大小过小时，全局内存的效果优于共享内存;从实验结果总体上看来,线程块大小越大,对程序性能的加速结果越明显,但当线程块过大时加速的提升就不是很大了。

4. 实验感想

4.1 问题即解决方案

在实验过程中,因为对 threadIdx.x 和 threadIdx.y 的理解错误导致矩阵乘法的计算结果一直不正确,这是因为误将 C++中 vector nums 中 nums[x][y]中的 x、y 的方向认为是与 CUDA threadIIx.x 和 threadIdx.y 是一致的。

而在 CUDA 编程中:
threadIdx.x 表示线程在其所属线程块中的 X 方向索引。
threadIdx.y 表示线程在其所属线程块中的 Y 方向索引。

4.2 实验感悟

在实现并行通用矩阵乘法并进行实验分析的过程中，我深刻体会到了 CUDA 并行计算的强大性能和灵活性。通过调整线程块大小、矩阵规模、访存方式，我发现这些因素对程序性能有着显著的影响。

首先，在调整线程块大小方面，我发现合适的线程块大小能够充分利用 GPU 的并行计算能力，提高计算效率。

其次，在矩阵规模方面，矩阵大小的选择会直接影响到并行计算的效率。较大的矩阵规模能够充分利用 GPU 的并行计算资源，提高计算效率，但也会增加访存和通信的开销。

再者，在访存方式方面，合理的内存访问模式能够减少内存访问延迟，提高计算效率。使用共享内存可以减少全局内存的访问次数，加速矩阵乘法运算。而优化的存储布局和内存访问模式也能够提高数据的局部性，减少内存访问冲突。

通过实验分析不同因素对并行性能的影响，我对 CUDA 并行计算的原理和优化技巧有了更深入的理解，也对如何优化并行程序性能有了更清晰的认识。在今后的工作中，我将继续探索并行计算领域，不断优化并行程序，提高计算效率。