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	9-CUDA 矩阵转置	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	2024/5/27

1. 实验目的

1.1. CUDA Hello World

本实验为 CUDA 入门练习，由多个线程并行输出“Hello World! ”。

输入：三个整数n, m, k，其取值范围为[1,32]

问题描述：创建n个线程块，每个线程块的维度为$m \times k$，每个线程均输出线程块编号、二维块内线程编号及 Hello World!（如，“Hello World from Thread (1, 2) in Block 10!”）。主线程输出“Hello World from the host!”。

要求：完成上述内容，观察输出，并回答线程输出顺序是否有规律？

1.2. CUDA 矩阵转置

使用 CUDA 对矩阵进行并行转置。

输入：整数n，其取值范围均为[512, 2048]

问题描述：随机生成$n \times n$的矩阵A，对其进行转置得到A^T。转置矩阵中第i行j列上的元素为原矩阵中j行i列元素，即$A^T_{ij} = A_{ji}$。

输出：矩阵A及其转置矩阵A^T，及计算所消耗的时间t。

要求：使用 CUDA 实现并行矩阵转置，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 CUDA Hello World

核函数:计算线程块编号,并输出线程在线程块内的编号以及线程块的编号:

```
// Kernel函数，每个线程块中的每个线程执行一次
__global__ void helloWorld(int m, int k) {
    int blockId = blockIdx.y * gridDim.x + blockIdx.x; // 计算线程块编号
    // int threadId = threadIdx.y * blockDim.x + threadIdx.x; // 计算线程在块内的编号

    // 输出线程块编号和线程在块内的编号
    printf("Hello World from Thread (%d, %d) in Block %d!\n", threadIdx.x, threadIdx.y, blockId);
}
```

主函数:确定线程块的维度$m \times k$，和线程块的数目,启动内核函数，主线程在内核函数结束后输出主机输出内容:

```
int main() {
    int m = 2; // 线程块维度 m
    int k = 3; // 线程块维度 k
    int n = 4; // 线程块数目 n
    dim3 gridDim(n); // 线程块网格维度
    dim3 blockDim(m, k); // 线程块维度

    // 启动内核函数
    helloWorld<<<gridDim, blockDim>>>(m, k);

    // 同步线程，等待内核函数执行完成
    cudaDeviceSynchronize();

    // 主线程输出信息
    printf("Hello World from the host!\n");

    return 0;
}
```

2.2 CUDA 矩阵转置

2.2.1 全局内存实现矩阵转置

```
// Kernel函数，每个线程块中的每个线程执行一次矩阵转置操作
// 每个线程处理一个位置的转置
__global__ void matrixTranspose_global(float *A, float *AT, int n) {
    // 计算线程在矩阵中的索引
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    // 检查索引是否超出矩阵的范围
    if (i < n && j < n) {
        // 计算转置后的索引并进行赋值
        AT[j * n + i] = A[i * n + j];
    }
}
```

利用每一个线程来负责对转置矩阵的一个位置进行赋值：

①首先计算每个线程负责的转置矩阵的索引位置。

②如果索引超出范围，则不进行处理。

③否则每个线程对对应的位置进行赋值。

2.2.2 共享内存实现矩阵转置

```cpp
// Kernel函数，每个线程块中的每个线程执行一次矩阵转置操作
// 每个线程块处理一个数据块的转置
__global__ void matrixTranspose_shared(float *A, float *AT, int n) {
    // 声明共享内存数组，用于存储输入矩阵的部分数据
    __shared__ float sharedMemory[ROW][COL];

    // 计算当前线程在输入矩阵中的索引
    int i1 = blockIdx.x * ROW + threadIdx.x;
    int j1 = blockIdx.y * COL + threadIdx.y;

    // 将输入矩阵中的数据复制到共享内存中
    if (i1 < n && j1 < n) {
        sharedMemory[threadIdx.y][threadIdx.x] = A[j1 * n + i1];
    }

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

    // 计算当前线程在输出矩阵中的索引，以进行转置操作
    int i2 = blockIdx.y * COL + threadIdx.x;
    int j2 = blockIdx.x * ROW + threadIdx.y;

    // 将共享内存中的数据写回到输出矩阵中，完成转置操作
    if (i2 < n && j2 < n) {
        AT[j2 * n + i2] = sharedMemory[threadIdx.x][threadIdx.y];
    }
}
```

使用共享内存实现矩阵转置,让每个线程块负责处理一个数据块的转置:

①计算当前线程在输入矩阵中负责的索引。

②将输入矩阵中的数据复制到共享内存中。

③使用 __syncthreads() 同步同一线程块里的所有线程，确保共享内存中的数据加载完成。

④ 计算当前线程在输出矩阵中的索引，以进行转置操作。

⑤将共享内存中的数据写回到输出矩阵中，完成转置操作。

3. 实验结果

3.1 CUDA Hello World 实验结果

(m=2 k=3 n=4)

```
jovyan@jupyter-21307174:~$ nvcc Hello_CUDA.cu -o Hello_CUDA
jovyan@jupyter-21307174:~$ ./Hello_CUDA
Hello World from Thread (0, 0) in Block 0!
Hello World from Thread (1, 0) in Block 0!
Hello World from Thread (0, 1) in Block 0!
Hello World from Thread (1, 1) in Block 0!
Hello World from Thread (0, 2) in Block 0!
Hello World from Thread (1, 2) in Block 0!
Hello World from Thread (0, 0) in Block 1!
Hello World from Thread (1, 0) in Block 1!
Hello World from Thread (0, 1) in Block 1!
Hello World from Thread (1, 1) in Block 1!
Hello World from Thread (0, 2) in Block 1!
Hello World from Thread (1, 2) in Block 1!
Hello World from Thread (0, 0) in Block 3!
Hello World from Thread (1, 0) in Block 3!
Hello World from Thread (0, 1) in Block 3!
Hello World from Thread (1, 1) in Block 3!
Hello World from Thread (0, 2) in Block 3!
Hello World from Thread (1, 2) in Block 3!
Hello World from Thread (0, 0) in Block 2!
Hello World from Thread (1, 0) in Block 2!
Hello World from Thread (0, 1) in Block 2!
Hello World from Thread (1, 1) in Block 2!
Hello World from Thread (0, 2) in Block 2!
Hello World from Thread (1, 2) in Block 2!
Hello World from the host!
```

可以从实验结果中看出,有关 BlockI 的输出是不确定,说明线程块的执行顺序是不确定的,但是 Block 中线程的输出顺序是一样的，说明线程块内的线程的执行顺序是确定的，即从上到下、从左往右。

3.2 CUDA 矩阵转置 实验结果

3.2.1 验证实验正确性

```
jovyan@jupyter-21307174:~$ nvcc Transpose.cu -o program
jovyan@jupyter-21307174:~$ ./program 5
Matrix Size: 5 x 5
The matrix before transposing:
0.840188 0.394383 0.783099 0.798440 0.911647
0.197551 0.335223 0.768230 0.277775 0.553970
0.477397 0.628871 0.364784 0.513401 0.952230
0.916195 0.635712 0.717297 0.141603 0.606969
0.016301 0.242887 0.137232 0.804177 0.156679

Grid Size (global): (1, 1)
Block Size (global): (16, 16)
Time to compute matrix transpose(global): 0.02048 milliseconds
The matrix transpose:
0.840188 0.197551 0.477397 0.916195 0.016301
0.394383 0.335223 0.628871 0.635712 0.242887
0.783099 0.768230 0.364784 0.717297 0.137232
0.798440 0.277775 0.513401 0.141603 0.804177
0.911647 0.553970 0.952230 0.606969 0.156679

The matrix before transposing:
0.840188 0.394383 0.783099 0.798440 0.911647
0.197551 0.335223 0.768230 0.277775 0.553970
0.477397 0.628871 0.364784 0.513401 0.952230
0.916195 0.635712 0.717297 0.141603 0.606969
0.016301 0.242887 0.137232 0.804177 0.156679

Grid Size (shared): (1, 1)
Block Size (shared): (16, 16)
Time to compute matrix transpose(shared): 0.00797 milliseconds
The matrix transpose:
0.840188 0.197551 0.477397 0.916195 0.016301
0.394383 0.335223 0.628871 0.635712 0.242887
0.783099 0.768230 0.364784 0.717297 0.137232
0.798440 0.277775 0.513401 0.141603 0.804177
0.911647 0.553970 0.952230 0.606969 0.156679

jovyan@jupyter-21307174:~$
```

可以看到使用全局内存和共享内存都实现了矩阵的转置。

3.2.2 比较全局内存和共享内存对程序性能的影响

```
jovyan@jupyter-21307174:~$ nvcc Transpose_test.cu -o run
jovyan@jupyter-21307174:~$ ./run
-------------------------
Matrix Size: 16 x 16
Grid Size (global): (1, 1)
Block Size (global): (16, 16)
Time to compute matrix transpose(global): 0.01946 milliseconds
Grid Size (shared): (1, 1)
Block Size (shared): (16, 16)
Time to compute matrix transpose(shared): 0.00778 milliseconds
-------------------------
-------------------------
Matrix Size: 32 x 32
Grid Size (global): (2, 2)
Block Size (global): (16, 16)
Time to compute matrix transpose(global): 0.00819 milliseconds
Grid Size (shared): (2, 2)
Block Size (shared): (16, 16)
Time to compute matrix transpose(shared): 0.00688 milliseconds
-------------------------
-------------------------
Matrix Size: 64 x 64
Grid Size (global): (4, 4)
Block Size (global): (16, 16)
Time to compute matrix transpose(global): 0.00717 milliseconds
Grid Size (shared): (4, 4)
Block Size (shared): (16, 16)
Time to compute matrix transpose(shared): 0.00659 milliseconds
-------------------------
-------------------------
Matrix Size: 128 x 128
Grid Size (global): (8, 8)
Block Size (global): (16, 16)
Time to compute matrix transpose(global): 0.00512 milliseconds
Grid Size (shared): (8, 8)
Block Size (shared): (16, 16)
Time to compute matrix transpose(shared): 0.00653 milliseconds
-------------------------
```

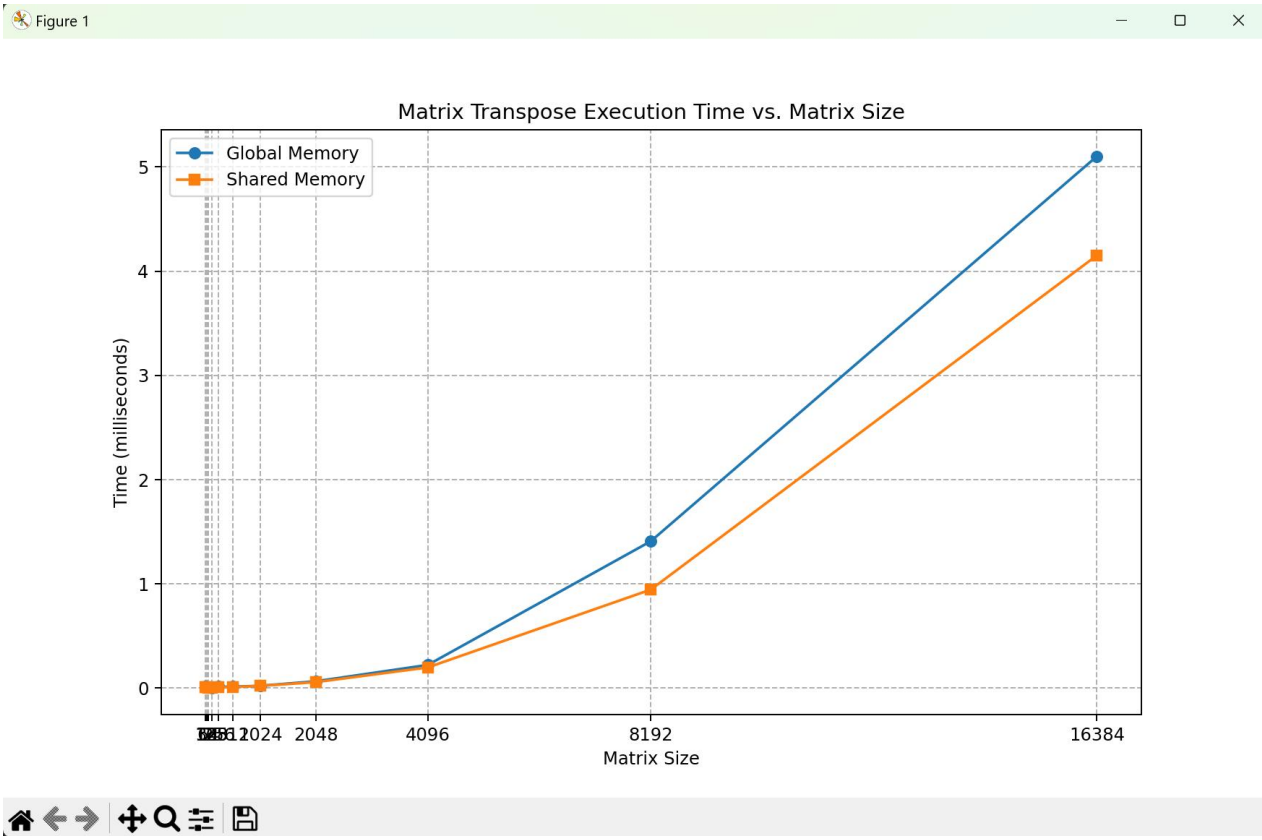
```
jovyan@jupyter-21307174:~$ nvcc Transpose.cu -o program
jovyan@jupyter-21307174:~$ ./program 5
Matrix Size: 5 x 5
The matrix before transposing:
0.840188 0.394383 0.783099 0.798440 0.911647
0.197551 0.335223 0.768230 0.277775 0.553970
0.477397 0.628871 0.364784 0.513401 0.952230
0.916195 0.635712 0.717297 0.141603 0.606969
0.016301 0.242887 0.137232 0.804177 0.156679

Grid Size (global): (1, 1)
Block Size (global): (16, 16)
Time to compute matrix transpose(global): 0.02048 milliseconds
The matrix transpose:
0.840188 0.197551 0.477397 0.916195 0.016301
0.394383 0.335223 0.628871 0.635712 0.242887
0.783099 0.768230 0.364784 0.717297 0.137232
0.798440 0.277775 0.513401 0.141603 0.804177
0.911647 0.553970 0.952230 0.606969 0.156679

The matrix before transposing:
0.840188 0.394383 0.783099 0.798440 0.911647
0.197551 0.335223 0.768230 0.277775 0.553970
0.477397 0.628871 0.364784 0.513401 0.952230
0.916195 0.635712 0.717297 0.141603 0.606969
0.016301 0.242887 0.137232 0.804177 0.156679

Grid Size (shared): (1, 1)
Block Size (shared): (16, 16)
Time to compute matrix transpose(shared): 0.00797 milliseconds
The matrix transpose:
0.840188 0.197551 0.477397 0.916195 0.016301
0.394383 0.335223 0.628871 0.635712 0.242887
0.783099 0.768230 0.364784 0.717297 0.137232
0.798440 0.277775 0.513401 0.141603 0.804177
0.911647 0.553970 0.952230 0.606969 0.156679

jovyan@jupyter-21307174:~$
```

```
--------------------------
Matrix Size: 512 x 512
Grid Size (global): (32, 32)
Block Size (global): (16, 16)
Time to compute matrix transpose(global): 0.01389 milliseconds
Grid Size (shared): (32, 32)
Block Size (shared): (16, 16)
Time to compute matrix transpose(shared): 0.01091 milliseconds
--------------------------
--------------------------
Matrix Size: 1024 x 1024
Grid Size (global): (64, 64)
Block Size (global): (16, 16)
Time to compute matrix transpose(global): 0.02294 milliseconds
Grid Size (shared): (64, 64)
Block Size (shared): (16, 16)
Time to compute matrix transpose(shared): 0.02202 milliseconds
--------------------------
--------------------------
Matrix Size: 2048 x 2048
Grid Size (global): (128, 128)
Block Size (global): (16, 16)
Time to compute matrix transpose(global): 0.06672 milliseconds
Grid Size (shared): (128, 128)
Block Size (shared): (16, 16)
Time to compute matrix transpose(shared): 0.05853 milliseconds
--------------------------
--------------------------
Matrix Size: 4096 x 4096
Grid Size (global): (256, 256)
Block Size (global): (16, 16)
Time to compute matrix transpose(global): 0.22294 milliseconds
Grid Size (shared): (256, 256)
Block Size (shared): (16, 16)
Time to compute matrix transpose(shared): 0.19901 milliseconds
--------------------------
```

```
--------------------------
Matrix Size: 8192 x 8192
Grid Size (global): (512, 512)
Block Size (global): (16, 16)
Time to compute matrix transpose(global): 1.40950 milliseconds
Grid Size (shared): (512, 512)
Block Size (shared): (16, 16)
Time to compute matrix transpose(shared): 0.94557 milliseconds
--------------------------
--------------------------
Matrix Size: 16384 x 16384
Grid Size (global): (1024, 1024)
Block Size (global): (16, 16)
Time to compute matrix transpose(global): 5.10138 milliseconds
Grid Size (shared): (1024, 1024)
Block Size (shared): (16, 16)
Time to compute matrix transpose(shared): 4.15069 milliseconds
--------------------------
```

程序性能对比:

Matrix Size	Grid Size (global)	Block Size (global)	Time (global) (ms)	Grid Size (shared)	Block Size (shared)	Time (shared) (ms)
16 x 16	(1, 1)	(16, 16)	0.01946	(1, 1)	(16, 16)	0.00778
32 x 32	(2, 2)	(16, 16)	0.00819	(2, 2)	(16, 16)	0.00688
64 x 64	(4, 4)	(16, 16)	0.00717	(4, 4)	(16, 16)	0.00659
128 x 128	(8, 8)	(16, 16)	0.00512	(8, 8)	(16, 16)	0.00653
256 x 256	(16, 16)	(16, 16)	0.01443	(16, 16)	(16, 16)	0.00864
512 x 512	(32, 32)	(16, 16)	0.01389	(32, 32)	(16, 16)	0.01091
1024 x 1024	(64, 64)	(16, 16)	0.02294	(64, 64)	(16, 16)	0.02202
2048 x 2048	(128, 128)	(16, 16)	0.06672	(128, 128)	(16, 16)	0.05853
4096 x 4096	(256, 256)	(16, 16)	0.22294	(256, 256)	(16, 16)	0.19901
8192 x 8192	(512, 512)	(16, 16)	1.40950	(512, 512)	(16, 16)	0.94557
16384 x 16384	(1024, 1024)	(16, 16)	5.10138	(1024, 1024)	(16, 16)	4.15069



可以从实验结果中看到,随着矩阵规模的增大,使用使用共享内存相较于使用全局内存对程序性能的加速效果越明显。

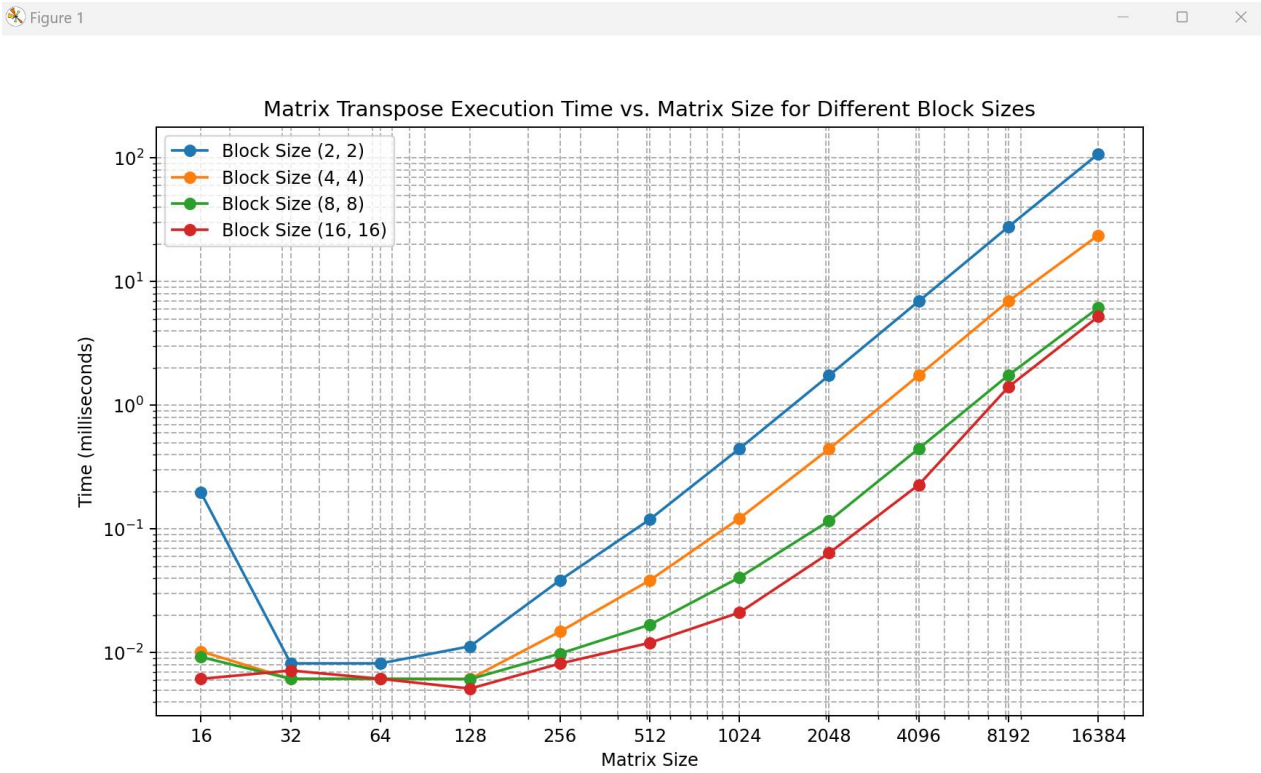
这是由于:

①共享内存位于每个线程块内部,与全局内存相比,共享内存的访问速度更快,延迟更低。这是因为共享内存是位于 GPU 芯片上的专用硬件单元,与处理器的寄存器文件相邻,访问速度非常快。

②共享内存是在线程块级别上共享的,意味着同一个线程块中的所有线程可以共同访问共享内存。这允许线程块内的线程在共享内存中共享数据,而无需通过全局内存进行通信,从而减少了数据传输的需求。

③共享内存允许数据在同一个线程块内被重复使用。当多个线程需要访问相同的数据时,如果这些数据已经存在于共享内存中,则可以直接从共享内存中读取,而不必每次都从全局内存中加载。这减少了对全局内存的访问次数,从而提高了性能。

3.2.3 不同线程块大小对程序性能的影响



可以看到线程块的维度越大，与程序性能的提高效果越明显,这是因为:

小线程块： 如果线程块太小，意味着每个线程块中的线程数量较少。这样会导致在某些情况下不能充分利用 GPU 的计算资源，因为在所有资源被充分利用之前，硬件可能会达到每个流多处理器（SM）的线程束数量的限制。小线程块可能会浪费硬件资源，降低程序的并行性和效率。

大线程块： 相反，如果线程块太大，每个线程块中的线程数量过多。这会导致在每个 SM 中每个线程的可用硬件资源较少。虽然较大的线程块可以提供更高的并行性，但是在某些情况下也会导致资源竞争和资源利用率下降，从而影响程序性能。

综上所述，选择适当的线程块大小是关键。通常，建议确保每个线程块中的线程数量是线程束大小（通常为 32）的倍数，同时避免线程块太小以及太大。

4. 实验感想

通过本次实验，我掌握了 CUDA 编程的入门基础，并且对于使用 CUDA 编程有了初步的了解。特别是在全局内存和共享内存的使用上，我对它们

的原理有了更深入的理解，并学会了如何在 CUDA 编程中灵活运用它们。通过对比全局内存和共享内存的使用情况，我也更清楚地认识到了它们对程序性能的影响。

在实验过程中，我发现通过合理使用不同的线程块大小，可以有效提高程序的性能。这让我意识到，在编写 CUDA 程序时，除了考虑算法的复杂度外，还要充分利用硬件资源和数据特性，以实现更高效的计算。

通过不断地实践和学习，我对 CUDA 编程有了更加丰富和深入的认识，我期待能够在以后的实践中进一步提升自己的技能，开发出更加高效和优秀的 CUDA 应用程序。