中山大学计算机院本科生实验报告

（2023 学年春季学期）

课程名称：并行程序设计　　批改人：

实验	CUDA 矩阵转置	专业（方向）	计算机科学与技术
学号	21307082	姓名	赖耿桂
Email	laigg@mail2.sysu.edu.cn	完成日期	2024.5.25

1. 实验目的（200 字以内）

①由多个线程并行输出“Hello World!”，观察输出，并回答线程输出顺序是否有规律。

②使用 CUDA 完成并行矩阵转置，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（600 字左右，图文并茂）

实验一

首先，定义一个函数 Print_Hello_World(注意使用__global__关键字来将其定义为全局函数)，用于每个线程打印“Hello World from Thread (x, y) in Block block_id!”。由于需要输出二维块内线程号和线程块编号，可以通过 blockIdx(dim3 类型)和 threadIdx(dim3 类型)来获取当前线程块号在网格中的索引和当前线程在当前线程块中的相对位置。

然后在 main 函数中，输入参数 n、m、k，然后输出“Hello World from the host!”，然后调用内核函数 Print_Hello_World，再使用 cudaDeviceSynchronize()进行同步，确保内核执行完毕，最后 cudaDeviceReset()释放所有资源。

实验一的文件为 CUDA_Hello_World.cu，完整代码如下：

```
# include <stdio.h>

__global__ void PrintHelloWorld(void){
    int blockID = blockIdx.x;
    int x = threadIdx.x;
    int y = threadIdx.y;
    printf("Hello World from Thread (%d, %d) in Block %d!\n",x,y,blockID);
}

```

```
int main(){
    int n, m, k;
    printf("请输入 n、m、k:\n");
    scanf("%d %d %d",&n,&m,&k);
    printf("Hello World from the host!\n");
    PrintHelloWorld<<<n,dim3(m,k)>>>();
    cudaDeviceSynchronize();
    cudaDeviceReset();
    return 0;
}
```

实验二

定义一些常量：

```
#define N 512
#define BLOCK_DIM_X 4
#define BLOCK_DIM_Y 4
#define TILE_SIZE 4
```

N 为矩阵规模参数(矩阵大小为 N*N)，BLOCK_SIZE 为线程块大小(线程块内的线程数)，BLOCK_DIM_X 和 BLOCK_DIM_Y 为线程块二维块内参数(如 BLOCK_SIZE 为 16 时，为一个(4,4)的块)，TILE_SIZE 为共享内存大小参数(一个 TILE_SIZE 行、(TILE_SIZE+1)列的二维数组；该参数仅在使用共享内存进行矩阵转置时有用)。

以上参数均可以进行调整。

定义在主进程随机初始化矩阵的函数 init(0-9 的随机整数)和打印矩阵的函数 printMatrix：

```
// 随机初始化 size*size 的矩阵
__host__ void init(int* matrix, int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            matrix[i * size + j] = rand() % 10;
        }
    }
}

// 打印 size 行 size 列的矩阵
__host__ void printMatrix(int* matrix, int size) {
    for (int i = 0; i < size; ++i) {
```

```
        for (int j = 0; j < size; ++j) {
            printf("%d ", matrix[i * size + j]);
        }
        printf("\n");
    }
}
```

然后定义矩阵转置函数，我分别在两个文件中定义了不同的矩阵转置函数，主要区别在于一个使用的是全局内存，另一个使用的是共享内存：

- 使用全局内存：

```
// 使用全局内存进行矩阵转置
__global__ void transpose_global(int *original, int *transposed){
    int bx = blockDim.x * blockIdx.x, by = blockDim.y * blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;
    int x = bx + tx, y = by + ty;
    if(x < N && y < N){
        transposed[y*N+x] = original[x*N+y];
    }
}
```

- 使用共享内存：

```
// 使用共享内存进行矩阵转置
__global__ void transpose_shared(int* original, int* transposed, int size) {
    __shared__ int smem[TILE_SIZE][TILE_SIZE + 1];//避免bank冲突，设置TILE_SIZE+1列
    int bx = blockIdx.x * TILE_SIZE, by = blockIdx.y * TILE_SIZE;
    int tx = threadIdx.x, ty = threadIdx.y;
    int x = bx + tx, y = by + ty;

    if (x < size && y < size) {
        smem[ty][tx] = original[y * size + x];
    }
    __syncthreads();
    if (bx + ty < size && by + tx < size) {
        transposed[(bx + ty) * size + (by + tx)] = smem[tx][ty];
    }
}
```

在 main 函数中，首先定义两个 int 型的指针作为数组，分别存储原矩阵和转置后的矩阵的数据，然后调用 cudaMallocHost 为它们分配空间，再对原矩阵进行初始化后进行输出：

```
int *original_matrix, *transposed_matrix;
cudaMallocHost((void **)&original_matrix, sizeof(int) * N * N);   // 使用 cudaMallocHost 分配原矩阵内存
cudaMallocHost((void **)&transposed_matrix, sizeof(int) * N * N); // 使用 cudaMallocHost 分配转置后的矩阵内存
init(original_matrix, N);                                         // 初始化原矩阵

printf("原矩阵:\n");
printMatrix(original_matrix, N);
```

接下来计算调用 cuda 核函数所需的块内信息 dimBlock 和描述网格的配置信息 dimGrid，并创建事件用于计时：

```
dim3 dimBlock(BLOCK_DIM_X, BLOCK_DIM_Y, 1);
dim3 dimGrid((N + BLOCK_DIM_X - 1) / BLOCK_DIM_X, (N + BLOCK_DIM_Y - 1) / BLOCK_DIM_Y, 1);

// 创建事件
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);
```

调用核函数并进行计时，然后输出线程块大小、矩阵规模、访存方式等相关信息和所用的时间：

全局内存：

```
cudaEventRecord(start, 0); // 记录事件

transpose_global<<<dimGrid, dimBlock>>>(original_matrix, transposed_matrix);

cudaEventRecord(end, 0);   // 记录事件
cudaEventSynchronize(end); // 同步

float global_time = 0;
cudaEventElapsedTime(&global_time, start, end); // 计时
printf("使用全局内存进行转置后的矩阵:\n");
printMatrix(transposed_matrix, N);
printf("线程块大小:(%d,%d) 矩阵规模:%d*%d 访存方式:全局内存 using time:%f ms\n", BLOCK_DIM_X, BLOCK_DIM_Y, N, N, global_time);
```

共享内存：

```
cudaEventRecord(start, 0); // 记录事件
transpose_shared<<<dimGrid, dimBlock>>>(original_matrix, transposed_matrix, N);
cudaEventRecord(end, 0);   // 记录事件
cudaEventSynchronize(end); // 同步
float shared_time = 0;
cudaEventElapsedTime(&shared_time, start, end); // 计时

printf("使用共享内存进行转置后的矩阵:\n");
printMatrix(transposed_matrix, N);
printf("线程块大小:(%d,%d) 矩阵规模:%d*%d 访存方式:共享内存 using time:%f ms\n", BLOCK_DIM_X, BLOCK_DIM_Y, N, N, shared_time);
```

```
最后释放内存：
cudaFreeHost(original_matrix);
cudaFreeHost(transposed_matrix);
cudaEventDestroy(start);
cudaEventDestroy(end);
```

3. 实验结果（500 字左右，图文并茂）

实验一

- 编译

```
nvcc -g -o CUDA_Hello_World CUDA_Hello_World.cu
```

- 运行

```
./CUDA_Hello_World
```

- 运行截图

```
jovyan@jupyter-21307082:~$ nvcc -g -o CUDA_Hello_World CUDA_Hello_World.cu
jovyan@jupyter-21307082:~$ ./CUDA_Hello_World
请输入n、m、k:
3 4 2
Hello World from the host!
Hello World from Thread (0, 0) in Block 2!
Hello World from Thread (1, 0) in Block 2!
Hello World from Thread (2, 0) in Block 2!
Hello World from Thread (3, 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 (2, 1) in Block 2!
Hello World from Thread (3, 1) in Block 2!
Hello World from Thread (0, 0) in Block 1!
Hello World from Thread (1, 0) in Block 1!
Hello World from Thread (2, 0) in Block 1!
Hello World from Thread (3, 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 (2, 1) in Block 1!
Hello World from Thread (3, 1) in Block 1!
Hello World from Thread (0, 0) in Block 0!
Hello World from Thread (1, 0) in Block 0!
Hello World from Thread (2, 0) in Block 0!
Hello World from Thread (3, 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 (2, 1) in Block 0!
Hello World from Thread (3, 1) in Block 0!
jovyan@jupyter-21307082:~$
```

图一

```
jovyan@jupyter-21307082:~$ ./CUDA_Hello_World 3 4 2
请输入n、m、k:
3 4 2
Hello World from the host!
Hello World from Thread (0, 0) in Block 2!
Hello World from Thread (1, 0) in Block 2!
Hello World from Thread (2, 0) in Block 2!
Hello World from Thread (3, 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 (2, 1) in Block 2!
Hello World from Thread (3, 1) in Block 2!
Hello World from Thread (0, 0) in Block 0!
Hello World from Thread (1, 0) in Block 0!
Hello World from Thread (2, 0) in Block 0!
Hello World from Thread (3, 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 (2, 1) in Block 0!
Hello World from Thread (3, 1) 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 (2, 0) in Block 1!
Hello World from Thread (3, 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 (2, 1) in Block 1!
Hello World from Thread (3, 1) in Block 1!
jovyan@jupyter-21307082:~$
```

图二

- 分析

可以看到线程输出顺序是有规律的。输入的 n、m、k 分别为 3、4、2，即有 3 个线程块，每个线程块有 4*2 个线程。可以看到有以下规律：

①同一线程块内的线程会一起输出。可以看到图一先输出了 Block 2 的所有“Hello World!”，再输出了 Block 1 的，最后输出了 Block 0 的。而图二则是先输出了 Block 2 的，再输出 Block 0 的，最后输出 Block 1 的，虽然块号顺序不同，但是都遵循同一块内的线程一起执行。

②然后可以看到，先固定 threadIdx.y，遍历了一遍 threadIdx.x(从 0 到 3，从小到大)。在遍历完一遍 threadIdx.x 后，更新 threadIdx.y，然后继续从头开始遍历 threadIdx.x，直到所有线程块的所有线程均执行完毕。

实验二

- 编译：

- 全局内存：

```
nvcc -g -o CUDA_Matrix_Transpose_global CUDA_Matrix_Transpose_global.cu
```

- 共享内存：

```
nvcc -g -o CUDA_Matrix_Transpose_shared CUDA_Matrix_Transpose_shared.cu
```

- 运行：

- 全局内存：

```
./CUDA_Matrix_Transpose_global
```

- 共享内存：

```
./CUDA_Matrix_Transpose_shared
```

- 运行截图，用于展示程序正确性：

- 全局内存：

```
jovyan@jupyter-21307082:~$ nvcc -g -o CUDA_Matrix_Transpose_global1 CUDA_Matrix_Transpose_global1.cu
jovyan@jupyter-21307082:~$ ./CUDA_Matrix_Transpose_global1
原矩阵:
3 6 7 5 3 5 6 2
9 1 2 7 0 9 3 6
0 6 2 6 1 8 7 9
2 0 2 3 7 5 9 2
2 8 9 7 3 6 1 2
9 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7
使用全局内存进行转置后的矩阵:
3 9 0 2 2 9 5 2
6 1 6 0 8 3 0 0
7 2 2 2 9 1 3 6
5 7 6 3 7 9 6 1
3 0 1 7 3 4 1 5
5 9 8 5 6 7 0 5
6 3 7 9 1 8 6 4
2 6 9 2 2 4 3 7
线程块大小:(2,2) 矩阵规模:8*8 访存方式:全局内存 using time:0.011168 ms
jovyan@jupyter-21307082:~$
```

- 共享内存：

```
jovyan@jupyter-21307082:~$ nvcc -g -o CUDA_Matrix_Transpose_shared CUDA_Matrix_Transpose_shared.cu
jovyan@jupyter-21307082:~$ ./CUDA_Matrix_Transpose_shared
原矩阵:
3 6 7 5 3 5 6 2
9 1 2 7 0 9 3 6
0 6 2 6 1 8 7 9
2 0 2 3 7 5 9 2
2 8 9 7 3 6 1 2
9 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7
使用共享内存进行转置后的矩阵:
3 9 0 2 2 9 5 2
6 1 6 0 8 3 0 0
7 2 2 2 9 1 3 6
5 7 6 3 7 9 6 1
3 0 1 7 3 4 1 5
5 9 8 5 6 7 0 5
6 3 7 9 1 8 6 4
2 6 9 2 2 4 3 7
线程块大小:(2,2) 矩阵规模:8*8 访存方式:共享内存 using time:0.012128 ms
jovyan@jupyter-21307082:~$
```

➢ 实验结果：

使用全局内存：

线程块大小	矩阵规模		
	512*512	1024*1024	2048*2048
16	0.529216 ms 1.543648 ms 0.896000 ms	4.522336 ms 4.796352 ms 5.865696 ms	23.414207 ms 54.921249 ms 27.241505 ms
64	0.279584 ms 1.503168 ms 1.476224 ms	1.461856 ms 10.090688 ms 10.495168 ms	9.371296 ms 50.894142 ms 45.538177 ms
256	0.211488 ms 1.422048 ms 1.352416 ms	0.645568 ms 14.175840 ms 10.160928 ms	4.863936 ms 51.571362 ms 41.515518 ms
1024	0.134944 ms 1.637184 ms 1.379392 ms	0.598080 ms 14.329664 ms 10.427264 ms	1.704096 ms 51.984928 ms 28.914593 ms

(说明：每个条目有 3 个时间分别是线程块维度为(N½*N½)、(N*1)、(1*N)的时间)

使用共享内存：

线程块大小	矩阵规模		
	512*512	1024*1024	2048*2048
16	0.452608 ms	1.807552 ms	23.459904 ms
64	0.273248 ms	0.998816 ms	6.453248 ms
256	0.166880 ms	0.606688 ms	2.156416 ms
1024	0.116512 ms	0.459200 ms	2.092224 ms

(说明：每个条目的时间为线程块维度为(N½,N½)、TILE_SIZE 也为 N½的时间)

- 分析

1) 线程块大小：

可以看到无论是使用全局内存还是共享内存，无论矩阵规模有多大，无论使用何种维度，总体上来说，随着线程块大小的增大，所用时间是减少的。这是因为随着线程块的增大，并行度提高，效率也随之提高。

2) 矩阵规模：

可以看到在线程块大小相同、线程块维度相同、访存方式相同的情况下，随着矩阵规模的增大，所用时间增加。显然是因为随着矩阵的增大，要处理的数据增加，时间自然也就增加了。

3) 访存方式：

可以看到在矩阵规模相同、线程块大小相同的情况下，使用共享内存的效率比使用全局内存的效率要高。这是因为共享内存是 GPU 的一种稀缺资源，它位于芯片上，只被同一个线程块内的线程所共享(而不是和全局内存一样被所有线程块所共享)，当没有发生 bank 冲突时，访问共享内存和访问寄存器一样快，所以共享内存空间要比本地和全局内存空间快得多。

4) 任务/数据划分方式：

可以看到在矩阵规模相同、都使用全局内存的情况下，总体上来说，不同线程块维度的效率大小为：(N½*N½) > (1*N) > (N*1)。可能的原因为：线程以二维块的形式访问内存(而不是像其他两种一样为行或列)，负载较为均衡，能够更好地实现内存合并，能够很好地匹配 GPU 的 SM（Streaming Multiprocessor）资源分配，确保较高的占用率和调度效率。

4. 实验感想（300 字左右）

通过此次实验，我初步掌握了 CUDA 编程，了解了 CUDA 中的线程块的各个参数如 blockIdx、threadIdx、blockDim、gridDim 的意义和它们之间的关系。此外，通过实验我还知道了不同访存方式和线程块维度对效率的影响以及使用共享内存中的 bank 冲突和解决方式。