中山大学计算机学院

本科生实验报告

（2025学年春季学期）

课程名称: 并行程序设计

实验	CUDA 矩阵转置	**专业(方向)**	计算机科学与技术
学号	--	**姓名**	--
Email	--	**完成日期**	2025.5.25

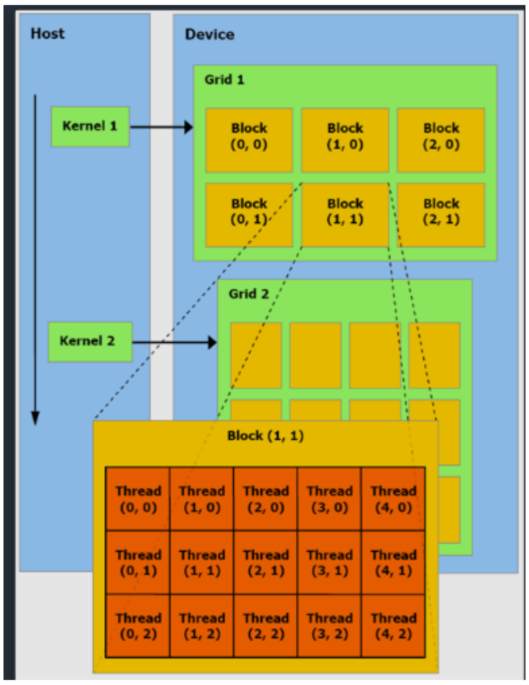
实验目的

- CUDA Hello World: CUDA入门练习, 由多个线程并行输出"Hello World!"
- 使用CUDA堆矩阵进行转置

实验过程和核心代码

CUDA Hello World

CUDA线程模型如下



本实验需要创建n个线程块, 即线程块维度为 n $\times$ 1
将线程块数设置为n, 每个线程块维度设置为m $\times$ k. 然后根据此配置去执行核函数.
核心代码如下

```
__global__ void hello_world_kernel() {
    printf("Hello World from thread (%2d,%2d) in Block %2d!\n", threadIdx.x, threadIdx.y, blockIdx.x);
}

int main() {
    int n, m, k;

    std::cout << "Please input three integers n,m,k(Range from 1 to 32): " << std::endl;
    std::cin >> n >> m >> k;

    int blockNum = n;
    dim3 blockDim(m, k);

    // Launch the kernel
    hello_world_kernel<<<blockNum, blockDim>>>();

    // Wait for GPU to finish before accessing on host
    cudaDeviceSynchronize();

    std::cout << "Hello World from the host!" << std::endl;
    return 0;
}
```

CUDA 矩阵转置

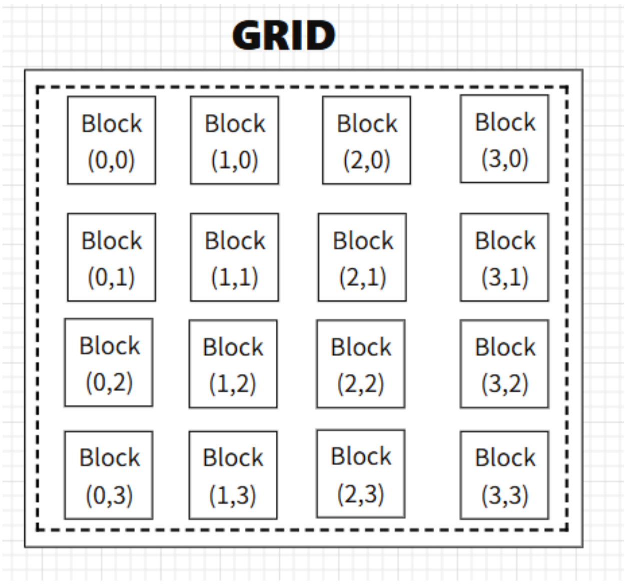
首先在host分配空间并初始化矩阵, 然后使用 `cudaMalloc` 在GPU显存上分配空间, 并使用 `cudaMemcpy` 将初始化的矩阵值复制到显存矩阵中

```
// initialize matrix
float* mat = new float[n * n];
float* ans = new float[n * n];
for(int i = 0; i < n * n; ++i) mat[i] = rand() % 100;
// cuda malloc
float* cuda_mat;
float* cuda_ans;
cudaMalloc(&cuda_mat, n * n * sizeof(float));
cudaMalloc(&cuda_ans, n * n * sizeof(float));
cudaMemcpy(cuda_mat, mat, n * n * sizeof(float), cudaMemcpyHostToDevice);
```

然后设置线程块的维度, 并据此计算出grid的大小, 使每个线程负责计算一个矩阵位置的转置

```
// 设置线程块大小与grid大小, 每个线程执行一个转置操作
dim3 blockDim(block_size, block_size);
dim3 gridDim((n + block_size - 1) / block_size, (n + block_size - 1) / block_size);
```

首先是普通的转置核函数, 根据线程的 `blockIdx` 与 `threadIdx` 来计算此线程负责的矩阵位置, 然后将 `ans` 矩阵该位置的值赋值为转置后的值



如图, 每个block负责矩阵的一块区域, 容易得出线程负责的位置计算公式为

$$(\text{blockIdx.x} \times \text{blocksize} + \text{threadIdx.x}, \text{blockIdx.y} \times \text{blocksize} + \text{threadIdx.y})$$

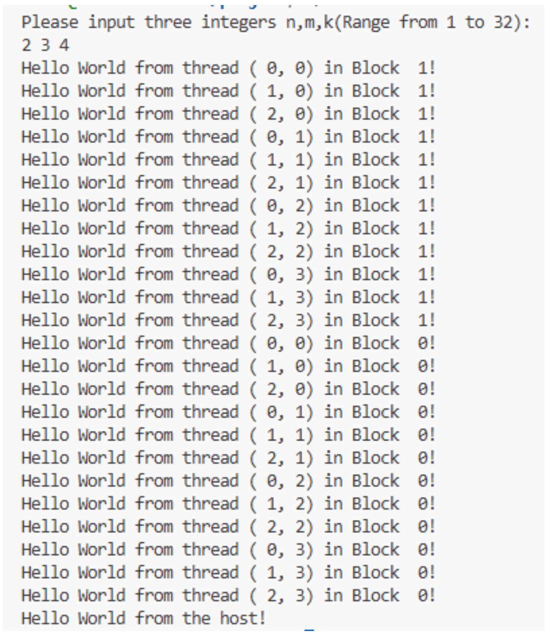
```
__global__ void transposeKernel(float* mat, float* ans, int n) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;
    if(row < n && col < n) {
        ans[row * n + col] = mat[col * n + row];
    }
}
```

接下来是使用共享内存的转置方案, 分为两个步骤. 第一步是将转置位置的原始矩阵值放到共享的内存区域, 然后需要等待同一个线程块的所有线程执行完第一步, 再从共享区域取值赋值到结果矩阵的对应位置.

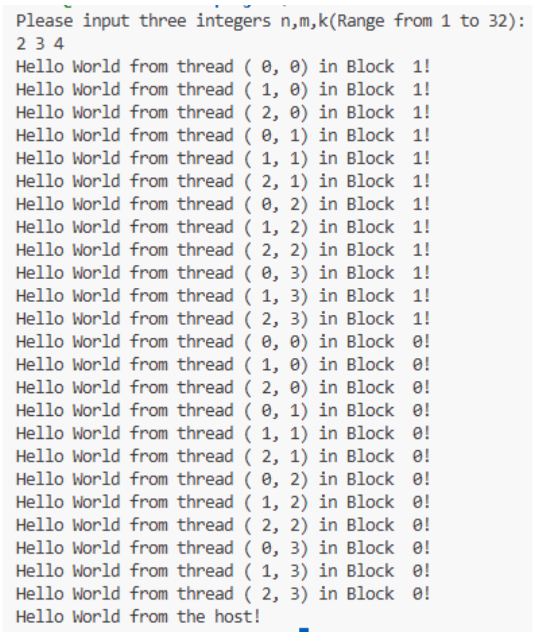
```
__global__ void transpose_shared(float* mat, float* ans, int n) {
    __shared__ float tile[32][32];

    int row_t = blockIdx.x * blockDim.x + threadIdx.x;
    int col_t = blockIdx.y * blockDim.y + threadIdx.y;

    if (row_t < n && col_t < n) {
        tile[threadIdx.y][threadIdx.x] = mat[col_t * n + row_t];
    }
    // 等待所有的线程第一步执行完毕
    __syncthreads();

    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;
    if (row < n && col < n) {
        ans[row * n + col] = tile[threadIdx.y][threadIdx.x];
    }
}
```

实验结果

CUDA Hello World

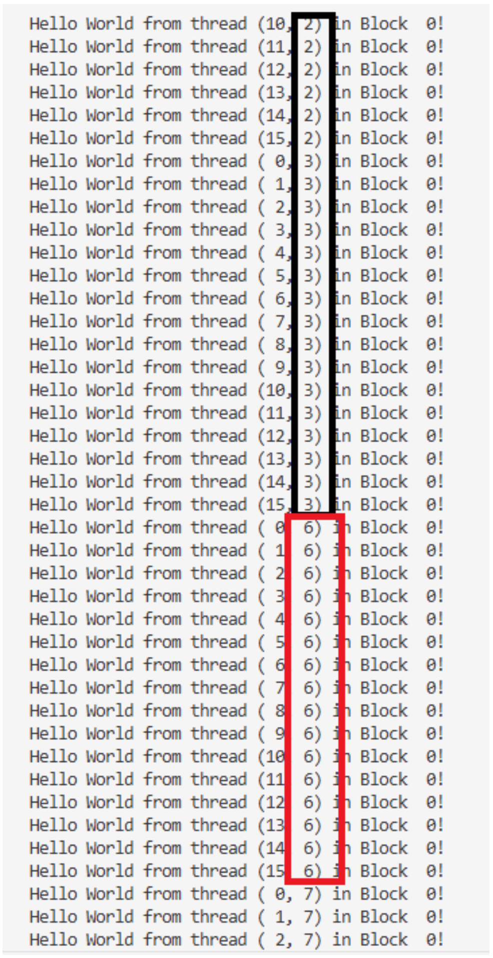


```
Please input three integers n,m,k(Range from 1 to 32):
2 3 4
Hello World from thread ( 0, 0) in Block  1!
Hello World from thread ( 1, 0) in Block  1!
Hello World from thread ( 2, 0) in Block  1!
Hello World from thread ( 0, 1) in Block  1!
Hello World from thread ( 1, 1) in Block  1!
Hello World from thread ( 2, 1) in Block  1!
Hello World from thread ( 0, 2) in Block  1!
Hello World from thread ( 1, 2) in Block  1!
Hello World from thread ( 2, 2) in Block  1!
Hello World from thread ( 0, 3) in Block  1!
Hello World from thread ( 1, 3) in Block  1!
Hello World from thread ( 2, 3) 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 ( 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 ( 0, 2) in Block  0!
Hello World from thread ( 1, 2) in Block  0!
Hello World from thread ( 2, 2) in Block  0!
Hello World from thread ( 0, 3) in Block  0!
Hello World from thread ( 1, 3) in Block  0!
Hello World from thread ( 2, 3) in Block  0!
Hello World from the host!
```



```
Please input three integers n,m,k(Range from 1 to 32):
2 3 4
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 ( 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 ( 0, 2) in Block  1!
Hello World from thread ( 1, 2) in Block  1!
Hello World from thread ( 2, 2) in Block  1!
Hello World from thread ( 0, 3) in Block  1!
Hello World from thread ( 1, 3) in Block  1!
Hello World from thread ( 2, 3) 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 ( 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 ( 0, 2) in Block  0!
Hello World from thread ( 1, 2) in Block  0!
Hello World from thread ( 2, 2) in Block  0!
Hello World from thread ( 0, 3) in Block  0!
Hello World from thread ( 1, 3) in Block  0!
Hello World from thread ( 2, 3) in Block  0!
Hello World from the host!
```

对比m,n,k都为2,3,4的两次输出, 可以看到**线程块的执行顺序**是随机的.



```
Hello World from thread (10, 2) in Block  0!
Hello World from thread (11, 2) in Block  0!
Hello World from thread (12, 2) in Block  0!
Hello World from thread (13, 2) in Block  0!
Hello World from thread (14, 2) in Block  0!
Hello World from thread (15, 2) in Block  0!
Hello World from thread ( 0, 3) in Block  0!
Hello World from thread ( 1, 3) in Block  0!
Hello World from thread ( 2, 3) in Block  0!
Hello World from thread ( 3, 3) in Block  0!
Hello World from thread ( 4, 3) in Block  0!
Hello World from thread ( 5, 3) in Block  0!
Hello World from thread ( 6, 3) in Block  0!
Hello World from thread ( 7, 3) in Block  0!
Hello World from thread ( 8, 3) in Block  0!
Hello World from thread ( 9, 3) in Block  0!
Hello World from thread (10, 3) in Block  0!
Hello World from thread (11, 3) in Block  0!
Hello World from thread (12, 3) in Block  0!
Hello World from thread (13, 3) in Block  0!
Hello World from thread (14, 3) in Block  0!
Hello World from thread (15, 3) in Block  0!
Hello World from thread ( 0, 6) in Block  0!
Hello World from thread ( 1, 6) in Block  0!
Hello World from thread ( 2, 6) in Block  0!
Hello World from thread ( 3, 6) in Block  0!
Hello World from thread ( 4, 6) in Block  0!
Hello World from thread ( 5, 6) in Block  0!
Hello World from thread ( 6, 6) in Block  0!
Hello World from thread ( 7, 6) in Block  0!
Hello World from thread ( 8, 6) in Block  0!
Hello World from thread ( 9, 6) in Block  0!
Hello World from thread (10, 6) in Block  0!
Hello World from thread (11, 6) in Block  0!
Hello World from thread (12, 6) in Block  0!
Hello World from thread (13, 6) in Block  0!
Hello World from thread (14, 6) in Block  0!
Hello World from thread (15, 6) in Block  0!
Hello World from thread ( 0, 7) in Block  0!
Hello World from thread ( 1, 7) in Block  0!
Hello World from thread ( 2, 7) in Block  0!
```

观察m,n,k为4,16,8的输出, 可以看到在同**一个线程块内的线程输出的顺序也是随机的**.

综上, CUDA不同线程块执行顺序不确定, 同一块内的线程顺序也是不确定的.

CUDA 矩阵转置

详细结果见`result.txt`

时间单位: ms

Matrix Size	Block Size	normal running time	shared running time
512	4	0.0666	0.0772
1024	4	0.2083	0.2999
2048	4	0.7779	1.1620
512	8	0.0492	0.0538
1024	8	0.1470	0.1691
2048	8	0.5091	0.6241
512	16	0.0759	0.0791
1024	16	0.2610	0.2749
2048	16	0.9287	0.9801
512	32	0.0770	0.0806
1024	32	0.2541	0.2891
2048	32	0.9390	1.0590

线程块大小影响: 可以看到在相同矩阵大小和访存方式上都在线程块为8x8时达到了最短的运行时间. 在线程块size更小时会产生更多的线程块, 增加调度的开销. 在线程块size更大时可能超出SM资源限制, 导致资源竞争和性能下降.

矩阵规模: 显然, 矩阵规模越大所需要的运行时间越长. 矩阵大小增加一倍时, 矩阵的数据增加了4倍, 但运行时间并没有增加4倍. 这是由于GPU能够创建的线程非常多, 矩阵规模越大越能发挥出GPU并行的优势

访存方式: 可以看到使用共享内存的方案在各矩阵规模和线程块大小上都更慢. 这是因为任务比较简单, 任务本身只需要访问原矩阵一次. 而使用共享内存后还是需要先访问原矩阵一次, 将值复制到共享内存, 同时还需要进行块内线程的同步操作. 导致运行时间更长. 如果对比较复杂的任务, 先访问一次放在共享内存后, 后续访问多次直接通过共享内存才能发挥出共享内存的优势.

实验感想

在本此实验中, 我学习了简单的cuda编程. 并了解了Grid, Block的概念, 了解了GPU是如何组织线程的. 同时通过CUDA Hello World实验我还了解了CUDA线程的特性. 通过CUDA矩阵转置实验我了解了如何在Host和Device之间传输数据, 并且了解了共享内存的机制. 通过实验的结果我还探讨了在哪些情况下使用共享可能可以加速任务的执行.