中山大学计算机学院

并行程序设计

本科生实验报告

（2024学年春季学期）
课程名称：Parallel Programming
批改人：

实验	CUDA矩阵转置	专业(方向)	计算机科学与技术
学号	21307185	姓名	张礼贤
Email	zhanglx78@mail2.sysu.edu.cn	完成日期	2024.5.21

- 1. 实验目的
- 2. 实验过程与代码
  - 2.1 CUDA Hello World
  - 2.2 CUDA 矩阵转置
    - 2.2.1 核函数(共享内存)
    - 2.2.2 核函数(全局内存)
    - 2.2.3 网格和线程块的划分
    - 2.2.4 调用执行
- 3. 实验结果与分析
  - 3.1 CUDA Hello World
  - 3.2 CUDA 矩阵转置
- 4. 实验感想

1. 实验目的

1. 了解CUDA编程模型。
2. 掌握CUDA编程的基本方法。
3. 掌握CUDA编程解决矩阵转置问题的方法。

4. 学会分析CUDA程序在不同的访存方式、线程块大小下的性能差异。

---

2. 实验过程与代码

2.1 CUDA Hello World

1. **编写CUDA核函数：**

```
__global__ void helloWorld(int n, int m, int k) {
    int blockId = blockIdx.x + blockIdx.y *
gridDim.x;
    int threadId = threadIdx.x + threadIdx.y *
blockDim.x;

    printf("Hello World from Thread (%d, %d) in Block
%d!\n", threadIdx.x, threadIdx.y, blockId);
}
```

2. **根据输入构造线程块和线程并运行：**

```
int n = atoi(argv[1]);  // 线程块数量
int m = atoi(argv[2]);  // 每个线程块的维度
int k = atoi(argv[3]);  // 每个线程块的维度

helloWorld<<<n, dim3(m, k)>>>(n, m, k);

cudaDeviceSynchronize();

printf("Hello World from the host!\n");
```

2.2 CUDA 矩阵转置

2.2.1 核函数(共享内存)

CUDA核函数 `matrixTranspose` 主要分为三个部分：

1. **计算线程的全局索引**

```
int x = blockIdx.x * TILE_DIM + threadIdx.x;
int y = blockIdx.y * TILE_DIM + threadIdx.y;
```

每个线程的全局索引 `(x, y)` 是通过块索引 `blockIdx` 和块内线程索引 `threadIdx` 计算的：

- `blockIdx.x` 和 `blockIdx.y` 分别表示当前线程块在网格中的x和y位置。
- `threadIdx.x` 和 `threadIdx.y` 分别表示当前线程在块内的x和y位置。
- `TILE_DIM` 是块的尺寸，通常是32（即32x32的线程块）。

这样，每个线程计算出的全局索引 `(x, y)` 可以唯一标识在整个矩阵中的位置。

2. **数据加载**：

```
if (x < width && y < height)
{
    int index_in = y * width + x;

    for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS) {
        if (y + i < height) {
            tile[threadIdx.y + i][threadIdx.x] =
d_in[index_in + i * width];
        }
    }
}
```

- 首先检查 `(x, y)` 是否在输入矩阵的范围内，防止越界访问。
- `index_in` 计算当前线程在全局输入矩阵中的线性索引。
- 使用一个循环，每个线程在 `BLOCK_ROWS` 步长内加载 `TILE_DIM / BLOCK_ROWS` 个元素：
- `tile[threadIdx.y + i][threadIdx.x]` 是共享内存中的位置。
- `d_in[index_in + i * width]` 是全局内存中的位置。

3. **数据存储**：

```
__syncthreads(); // 等待所有线程加载完毕

x = blockIdx.y * TILE_DIM + threadIdx.x;
y = blockIdx.x * TILE_DIM + threadIdx.y;

if (x < height && y < width) {
    int index_out = y * height + x;

    for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS) {
        if (y + i < width) {
            d_out[index_out + i * height] =
tile[threadIdx.x][threadIdx.y + i];
        }
    }
}
```

- 每个线程从共享内存 `tile` 中加载数据，并将其写入全局内存中的输出矩阵 `d_out`。
- 注意这里的坐标 `(x, y)` 与之前不同，因为在每个线程块中已经实现了转置，所以 x, y 尾部 + threadIdx.x, threadIdx.y。

2.2.2 核函数(全局内存)

全局访存方式的 CUDA 核函数利用并行线程计算，将输入矩阵按行列互换（转置）存储到输出矩阵中，每个线程负责一个元素的位置交换操作。

```
// 全局内存访存的CUDA核函数
__global__ void matrixTransposeGlobal(float *d_out, float
*d_in, int width, int height) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < width && y < height) {
        int index_in = y * width + x;
        int index_out = x * height + y;
        d_out[index_out] = d_in[index_in];
    }
}
```

2.2.3 网格和线程块的划分

1. **线程块大小**

```
dim3 threads(TILE_DIM, BLOCK_ROWS);
```

   - `TILE_DIM = 32`：每个线程块在 x 方向上有 32 个线程。
   - `BLOCK_ROWS = 8`：每个线程块在 y 方向上有 8 个线程。
   - 每个线程块包含 `32 x 8 = 256` 个线程。

2. **网格大小**

```
dim3 grid((n + TILE_DIM - 1) / TILE_DIM, (m +
TILE_DIM - 1) / TILE_DIM);
```

   - `(n + TILE_DIM - 1) / TILE_DIM`：矩阵在 x 方向上需要多少个线程块。
   - `(m + TILE_DIM - 1) / TILE_DIM`：矩阵在 y 方向上需要多少个线程块。

   例如，对于一个 `m x n` 的矩阵，假设 `m = 1024`，`n = 1024`：

   - `grid.x = (1024 + 32 - 1) / 32 = 32`
   - `grid.y = (1024 + 32 - 1) / 32 = 32`

   所以网格的大小是 `32 x 32`，共包含 `32 x 32 = 1024` 个线程块。

2.2.4 调用执行

1. **启动 CUDA 核函数**

```
// 启动CUDA核函数，进行矩阵转置
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

matrixTranspose<<<grid, threads>>>(d_transposed,
d_matrix, n, m);
```

2. **进行时间统计：**

```
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);

// 将转置后的矩阵数据从GPU内存复制回主机内存
cudaMemcpy(h_transposed, d_transposed, bytes,
cudaMemcpyDeviceToHost);
```

3. **结果检验：**

```
bool correct = true;
for (int i = 0; i < m; i++)
{
    for (int j = 0; j < n; j++)
    {
        if (h_transposed[i * n + j] != h_matrix[j * m
+ i])
        {
            correct = false;
        }
    }
}
if (correct)
{
    printf("Matrix Transpose is correct!\n");
}
else
{
    printf("Matrix Transpose is incorrect!\n");
}
```

为了避免庞大的矩阵输出，这里只输出了转置结果是否正确。

---

3. 实验结果与分析

3.1 CUDA Hello World

1. **编译运行：**

```
nvcc helloWorld.cu -o helloWorld
./helloWorld 4 3 2
```

2. **结果展示：**

```
jovyan@jupyter-21307185:~$ ./hello 4 3 2
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 (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 (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, 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, 0) in Block 3!
Hello World from Thread (1, 0) in Block 3!
Hello World from Thread (2, 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 (2, 1) in Block 3!
Hello World from the host!
jovyan@jupyter-21307185:~$
```

3. **实验分析：**

   通过多次执行程序，可以发现 `BlockIdx` 的输出是不确定的，因为线程块的执行顺序是不确定的。但是每个线程块内的线程执行顺序是确定的，所以每个线程块内的输出是有序的：从左至右，从上至下。

3.2 CUDA 矩阵转置

1. **编译运行：**

```bash
#!/bin/bash

# 编译 CUDA 程序
nvcc -o transpose_matrix_v2 transpose_matrix_v2.cu

# 结果文件
output_file="results_v2.txt"

# 清空之前的结果文件
echo "" > $output_file

# 设置 m 和 n 的范围
start=512
end=2048
step=256

# 遍历 m 和 n 的组合
for n in $(seq $start $step $end); do
    # 运行 CUDA 程序并将结果追加到结果文件
    echo "Running for m=$n, n=$n" | tee -a $output_file
    ./transpose_matrix_v2 $n $n >> $output_file
    echo "--------------" >> $output_file
done


echo "All tests completed. Results are saved in $output_file"
```

2. **结果展示：**

```
Running for m=512, n=512
Testing Global Memory Transpose:
Matrix Transpose is correct!
Matrix size: 512x512, Grid: (32,32), Threads: (16,16), Shared Memory: No
Time taken for matrix transpose: 0.039520 milliseconds
Testing Shared Memory Transpose:
Matrix Transpose is correct!
Matrix size: 512x512, Grid: (32,32), Threads: (16,16), Shared Memory: Yes
Time taken for matrix transpose: 0.022112 milliseconds
Testing Global Memory Transpose:
Matrix Transpose is correct!
Matrix size: 512x512, Grid: (16,64), Threads: (32,8), Shared Memory: No
Time taken for matrix transpose: 0.060864 milliseconds
Testing Shared Memory Transpose:
Matrix Transpose is correct!
Matrix size: 512x512, Grid: (16,64), Threads: (32,8), Shared Memory: Yes
Time taken for matrix transpose: 0.019776 milliseconds
Testing Global Memory Transpose:
Matrix Transpose is correct!
Matrix size: 512x512, Grid: (16,16), Threads: (32,32), Shared Memory: No
Time taken for matrix transpose: 0.064608 milliseconds
Testing Shared Memory Transpose:
Matrix Transpose is correct!
Matrix size: 512x512, Grid: (16,16), Threads: (32,32), Shared Memory: Yes
Time taken for matrix transpose: 0.025056 milliseconds
--------------
Running for m=768, n=768
Testing Global Memory Transpose:
Matrix Transpose is correct!
Matrix size: 768x768, Grid: (48,48), Threads: (16,16), Shared Memory: No
Time taken for matrix transpose: 0.058048 milliseconds
Testing Shared Memory Transpose:
Matrix Transpose is correct!
Matrix size: 768x768, Grid: (48,48), Threads: (16,16), Shared Memory: Yes
Time taken for matrix transpose: 0.039936 milliseconds
```

由于输出结果较多，利用sh文件将结果保存到results_v2.txt文件中。可查看results_v2.txt。

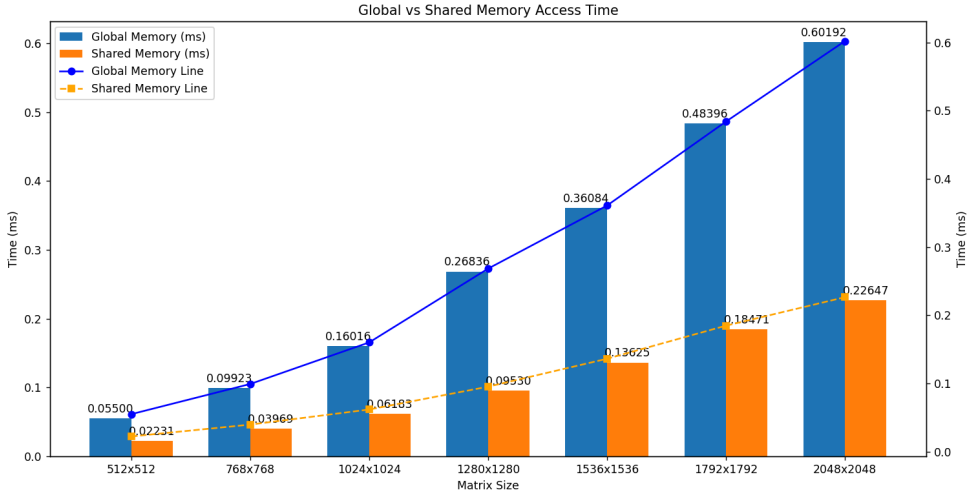
3. **实验分析：**

根据上面的实验结果，我们可以得到如下的表格：

矩阵大小	网格尺寸	线程块尺寸	全局内存访存时间 (ms)	共享内存访存时间 (ms)
512x512	(32, 32)	(16, 16)	0.039520	0.022112
	(16, 64)	(32, 8)	0.060864	0.019776
	(16, 16)	(32, 32)	0.064608	0.025056
768x768	(48, 48)	(16, 16)	0.058048	0.039936
	(24, 96)	(32, 8)	0.113888	0.037920
	(24, 24)	(32, 32)	0.125760	0.041216
1024x1024	(64, 64)	(16, 16)	0.088928	0.057216
	(32, 128)	(32, 8)	0.192064	0.060800
	(32, 32)	(32, 32)	0.199488	0.067488
1280x1280	(80, 80)	(16, 16)	0.167360	0.089664
	(40, 160)	(32, 8)	0.306880	0.093696

矩阵大小	网格尺寸	线程块尺寸	全局内存访存时间 (ms)	共享内存访存时间 (ms)
	(40, 40)	(32, 32)	0.330848	0.102528
1536x1536	(96, 96)	(16, 16)	0.199296	0.129568
	(48, 192)	(32, 8)	0.424896	0.128992
	(48, 48)	(32, 32)	0.458336	0.150176
1792x1792	(112, 112)	(16, 16)	0.250592	0.168736
	(56, 224)	(32, 8)	0.586048	0.172896
	(56, 56)	(32, 32)	0.615232	0.212512
2048x2048	(128, 128)	(16, 16)	0.315200	0.215232
	(64, 256)	(32, 8)	0.744256	0.216192
	(64, 64)	(32, 32)	0.746304	0.248000

对于不同的矩阵规模，其可视化图像如下：



从上面的实验结果可以看出：

1. 随着矩阵规模的增大，访存的时间也随之增大
    - 从数值的角度来看，全局内存访存的时间要比共享内存访存的时间要大
    - 从增量的角度来看，全局内存访存的时间走向相对较陡，而共享内存访存的时间走向相对较缓

    具体分析如下：

    1. **内存层次结构差异**：
        - **全局内存**：在GPU中，全局内存是所有线程都可以访问的内存，访问延迟较高，因为它位于设备内存层次结构的较低级别，需要通过较多的中间层（如缓存和内存控制器）进行访问。随着矩阵规模的增大，全局内存的访问次数和数据量也会显著增加，从而导致访问时间急剧上升。
        - **共享内存**：共享内存是位于每个SM（Streaming Multiprocessor）中的高速缓存，每个线程块的线程可以共享访问。共享内存的访问延迟较低，因为它是片内存储器，访问速度快且带宽高。即使矩阵规模增大，只要合理使用共享内存，访存时间的增长速度也会相对较慢。
    2. **内存访问模式**：
        - **全局内存**：访问全局内存时，如果访问模式不连续或不对齐，可能会导致非合并访问（non-coalesced access），这会进一步增加访存延迟。随着矩阵规模的增大，非合并

访问的概率也会增加，从而导致全局内存访存时间的陡增。
    - **共享内存**：共享内存的访问模式更容易优化为合并访问（coalesced access），即使矩阵规模增大，访存效率仍能保持较高。

3. **带宽限制**：

    - **全局内存**：全局内存的带宽较大，但当多个线程同时访问全局内存时，带宽会被迅速消耗。随着矩阵规模的增大，内存带宽的瓶颈会更加明显，从而导致访存时间成倍增长。
    - **共享内存**：共享内存的带宽非常高，并且访问延迟极低。即使线程数量和访存请求增加，只要访问模式合理，带宽瓶颈的影响较小。

   因此，随着矩阵规模的增大，全局内存访存时间增长更快主要是由于内存层次结构的差异、内存访问模式的效率差异、带宽限制以及缓存效果的不同。合理利用共享内存可以有效减少访存时间，提高内存访问效率。

2. 在同矩阵规模下：

    - **(16 x 16)的划分方式时间最短**：线程块较小，内存访问模式优化，并行度适中。
    - **(32 x 8)的划分方式时间次之**：线程块尺寸中等，但内存访问模式效率略低。
    - **(32 x 32)的划分方式时间最长**：线程块过大，内存访问不连续，并行度下降。

---

4. 实验感想

1. 在本次实验中，我学习到了简单的CUDA入门编程，了解了CUDA编程的基本方法和技巧
2. 通过实验，我掌握了CUDA编程解决矩阵转置问题的方法，学会了如何分析CUDA程序在不同的访存方式、线程块大小下的性能差异
3. 另外，在实验中我也遇到了很多问题：

1. 对于矩阵转置的 `index` 的计算没有很好的理解，后来经过手动绘图，明白了核函数于全局地址的计算关系
2. 对于共享内存的使用，一开始认为是块内共享，后来发现是线程块间共享，因此排查出了错误