Parallel-Programming Task9

刘森元, 21307289

中山大学计算机学院

Codes on https://github.com/Myocardial-infarction-Jerry/Parallel-Programming/tree/main/Task9.

Project built by CMake.

```
> cd Task9
> cmake . && make
> bin/CUDAMatTp <n>
> ctest -V
```

1 Environment

11th Gen Intel(R) Core(TM) i7-11700KF @ 3.60GHz

NVIDIA GeForce RTX 3080 Ti O12G

Windows Subsystem for Linux @ Ubuntu 22.04 LTS

2 Task

2.1 CUDA Hello World

> **Note**
>
> 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?

2.2 CUDA 矩阵转置

> **Note**
>
> 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AT。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$AijT=Aji$。
>
> **输出：**矩阵A及其转置矩阵AT，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（全局内存访问，共享内存访问），任务/数据划分和映射方式，对程序性能的影响。

3 Theory

3.1 并行计算理论与CUDA模型

1. **线程、块和网格**：
   - CUDA中的基本并行单位是线程。多个线程组合成一个线程块（Block），而多个线程块组合成一个网格（Grid）。这种层次的组织方式允许高度灵活的并行计算，适应不同规模和复杂度的计算任务。
   - 线程的索引由 `blockIdx`，`blockDim` 和 `threadIdx` 等确定。
2. **内存访问模式**：
   - **共享内存与全局内存**：CUDA中，共享内存的访问速度远快于全局内存。在这段代码中，矩阵元素直接从全局内存读取和写入，没有使用共享内存。这可能不是最优的内存访问策略，因为全局内存访问可能导致较高的延迟和内存带宽的瓶颈。
   - **内存访问模式的影响**：内存访问密集型的操作在写入操作时可能不是连续的，这可能会导致全局内存访问效率下降。
3. **核函数执行与同步**：
   - **核函数（Kernel）**：核函数由GPU上的多个线程并行执行。每次执行核函数时，都会在设定的网格和块的维度上启动线程。
   - **设备同步**：`cudaDeviceSynchronize()` 函数调用确保GPU完成所有线程的执行，之后才能进行时间计算和数据回传。这是必要的，因为GPU上的核函数执行是异步的。

3.2 性能考虑

- **数据传输**：数据从主机（CPU）到设备（GPU）的传输在性能上是一个瓶颈，因为它涉及到主存和GPU存储器之间的数据移动。在实际应用中，减少数据传输次数和优化数据传输模式是提高性能的关键。
- **并行粒度**：选择合适的块大小和形状对性能有重要影响。理想的块大小应该能够最大化处理器核心的利用率，同时减少线程间的冲突和等待。

4 Code

4.1 CUDA Hello World

> **Caution**
>
> 源代码详见 CUDAHelloWorld.cu

```cpp
__global__ void helloFromGPU(int n, int m, int k) {
    int idx = blockIdx.x;
    int idy = threadIdx.y + threadIdx.x * blockDim.y;

    if (idx < n && idy < m * k) {
        printf("Hello World from Thread (%d, %d) in Block %d!\n", threadIdx.x, threadIdx.y, blockIdx.x);
    }
}
```

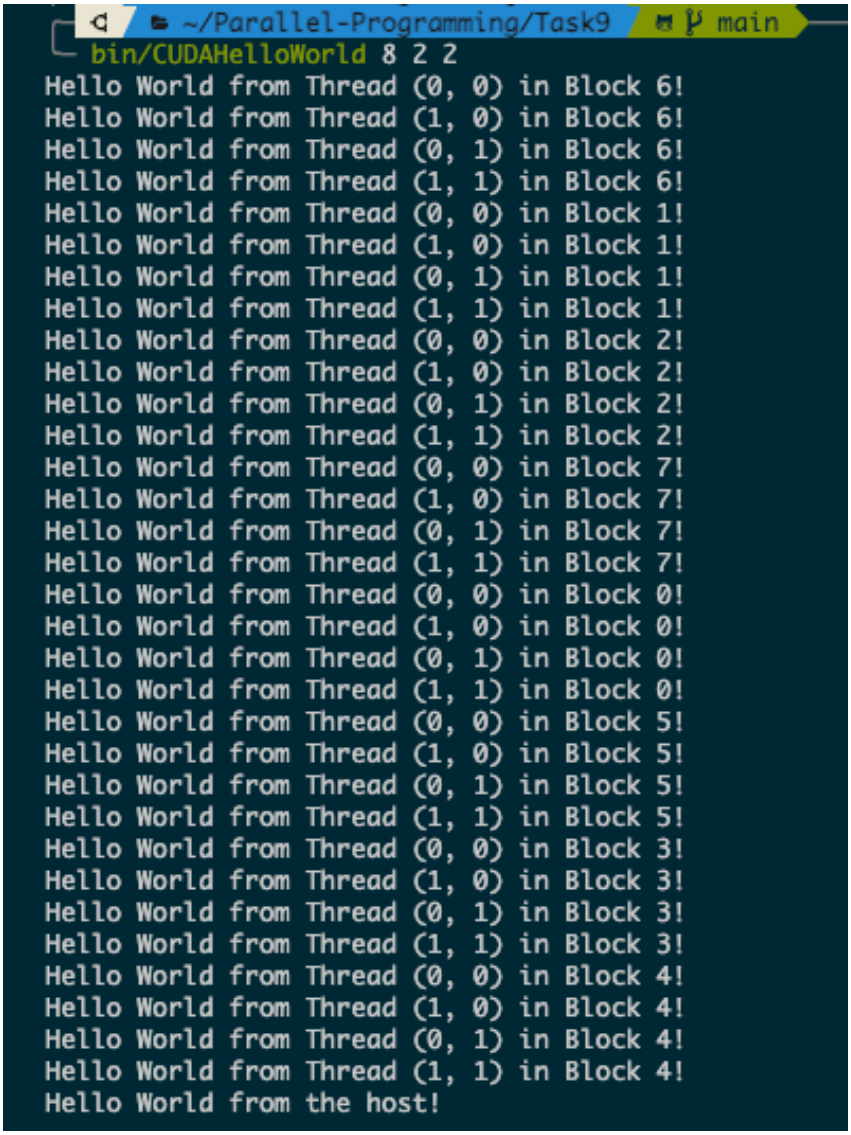
4.2 CUDA 矩阵转置

> **Caution**
>
> 源代码详见 CUDAMatTp.cu

```cpp
__global__ void matTranspose(float *d_A, float *d_B, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < n && j < n) {
        d_B[j * n + i] = d_A[i * n + j];
    }
}
```

5 Result

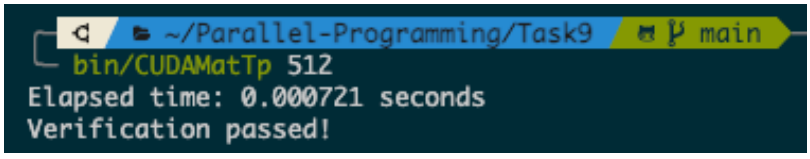
5.1 CUDA Hello World



在CUDA中，线程和线程块的调度并不保证按照特定的顺序执行。这是因为GPU调度器根据可用计算资源动态地管理线程块的执行。因此，尽管线程和线程块被分配了特定的索引，它们的实际执行顺序可以是非确定性的，通常取决于硬件的具体实现和当前的工作负载。

5.2 CUDA 矩阵转置

以 512 矩阵为例



有测试数据

Size	Time
512	0.529 ms
1024	0.862 ms
2048	1.482 ms

CUDA矩阵转置是一个广泛用来评估GPU性能的基准测试。以下将详细分析不同因素如线程块大小、矩阵规模、访存方式等如何影响并行矩阵转置的性能。

1. 线程块大小（Block Size）

   线程块大小是影响性能的关键参数之一，因为它直接关系到核心利用率和内存访问效率：

   - **小线程块**：可能无法充分利用GPU的并行性，因为活跃的线程数不足以覆盖内存访问延迟。
   - **大线程块**：虽然可以增加并行度，但可能会导致资源争用，如寄存器和共享内存的限制，从而降低效率。
   - **最优大小**：一般来说，使用16x16或32x32的线程块大小是效率和性能的一个良好折衷。这样的大小可以很好地映射到GPU的物理结构，同时保持较高的内存访问效率。

2. 矩阵规模（Matrix Size）

   矩阵的大小直接影响到并行算法的效率：

   - **小矩阵**：可能不需要太大的并行度，小线程块可能就足够了。
   - **大矩阵**：需要更高的并行度和更细致的内存访问策略来优化性能。

3. 访存方式（Memory Access Patterns）

   访问内存的方式对性能有显著影响，尤其是在矩阵转置这种频繁访问内存的操作中：

   - **全局内存访问**：直接从全局内存读取和写入数据。这种方式简单，但由于全局内存访问速度较慢，容易成为性能瓶颈。
   - **共享内存访问**：使用快速的共享内存来减少全局内存访问。一种常见的策略是使用共享内存作为块内数据的缓冲区，每个线程块先将全局内存中的一个矩阵块读入共享内存，转置后再写回全局内存。这种方式可以显著提高内存访问效率，尤其是对于大矩阵。

4. 任务/数据划分和映射方式

   任务如何在多个线程间划分也极为重要：

   - **一维划分**：每个线程处理一行或一列，这种方式简单但可能导致不均匀的内存访问和性能下降。
   - **二维划分**：更常见的是将矩阵划分为小块，每个线程块处理一个子块。这种划分方式可以更好地利用GPU的内存层次结构，减少内存访问延迟。

对CUDA矩阵转置性能的影响因素是多方面的。实际选择最佳策略时，通常需要根据具体应用的需求和GPU的硬件特性进行调整和优化。实验不同的配置和优化方法，测量它们对性能的具体影响，将有助于深入理解并行算法的性能瓶颈和优化空间。