

# Parallel-Programming Task10

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/Task10>.

Project built by CMake.

```
1 | > cd Task10
2 | > cmake . && make
3 | > bin/CUDAMatMul <size>
```

## 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: CUDA 并行矩阵乘法

CUDA实现并行通用矩阵乘法，并通过实验分析不同线程块大小，访存方式、数据/任务划分方式对并行性能的影响。

**输入：** $m, n, k$ 三个整数，每个整数的取值范围均为 $[128, 2048]$

**问题描述：**随机生成 $m \times n$ 的矩阵 $A$ 及 $n \times k$ 的矩阵 $B$ ，并对这两个矩阵进行矩阵乘法运算，得到矩阵 $C$ 。

**输出：** $A, B, C$ 三个矩阵，及矩阵计算所消耗的时间 $t$ 。

**要求：**使用CUDA实现并行矩阵乘法，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。

## 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

### ⚠ Caution

源代码详见 [CUDAMatMul.cu](#)

```
1  __global__ void matMul(float *A, float *B, float *C, int size) {
2      int row = blockIdx.y * blockDim.y + threadIdx.y;
3      int col = blockIdx.x * blockDim.x + threadIdx.x;
4
5      if (row >= size || col >= size)
6          return;
7
8      float sum = 0;
9      for (int i = 0; i < size; ++i)
10         sum += A[row * size + i] * B[i * size + col];
11
12     C[row * size + col] = sum;
13 }
14
15 __global__ void matMulShared(float *A, float *B, float *C, int size) {
16     __shared__ float s_A[DIM][DIM];
17     __shared__ float s_B[DIM][DIM];
18
19     int row = blockIdx.y * blockDim.y + threadIdx.y;
20     int col = blockIdx.x * blockDim.x + threadIdx.x;
21
22     float sum = 0;
23     for (int i = 0; i < size / DIM; ++i) {
24         s_A[threadIdx.y][threadIdx.x] = A[row * size + i * DIM + threadIdx.x];
```

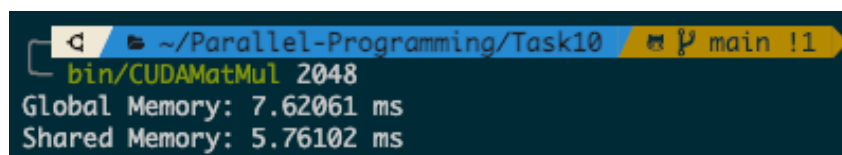
```

25     s_B[threadIdx.y][threadIdx.x] = B[(i * DIM + threadIdx.y) * size + col];
26     __syncthreads();
27
28     for (int j = 0; j < DIM; ++j)
29         sum += s_A[threadIdx.y][j] * s_B[j][threadIdx.x];
30     __syncthreads();
31 }
32
33 C[row * size + col] = sum;
34 }

```

## 5 Result

以 `dim(16, 16)` 以及 2048 矩阵为例



```

~/Parallel-Programming/Task10 main !1
bin/CUDAMatMul 2048
Global Memory: 7.62061 ms
Shared Memory: 5.76102 ms

```

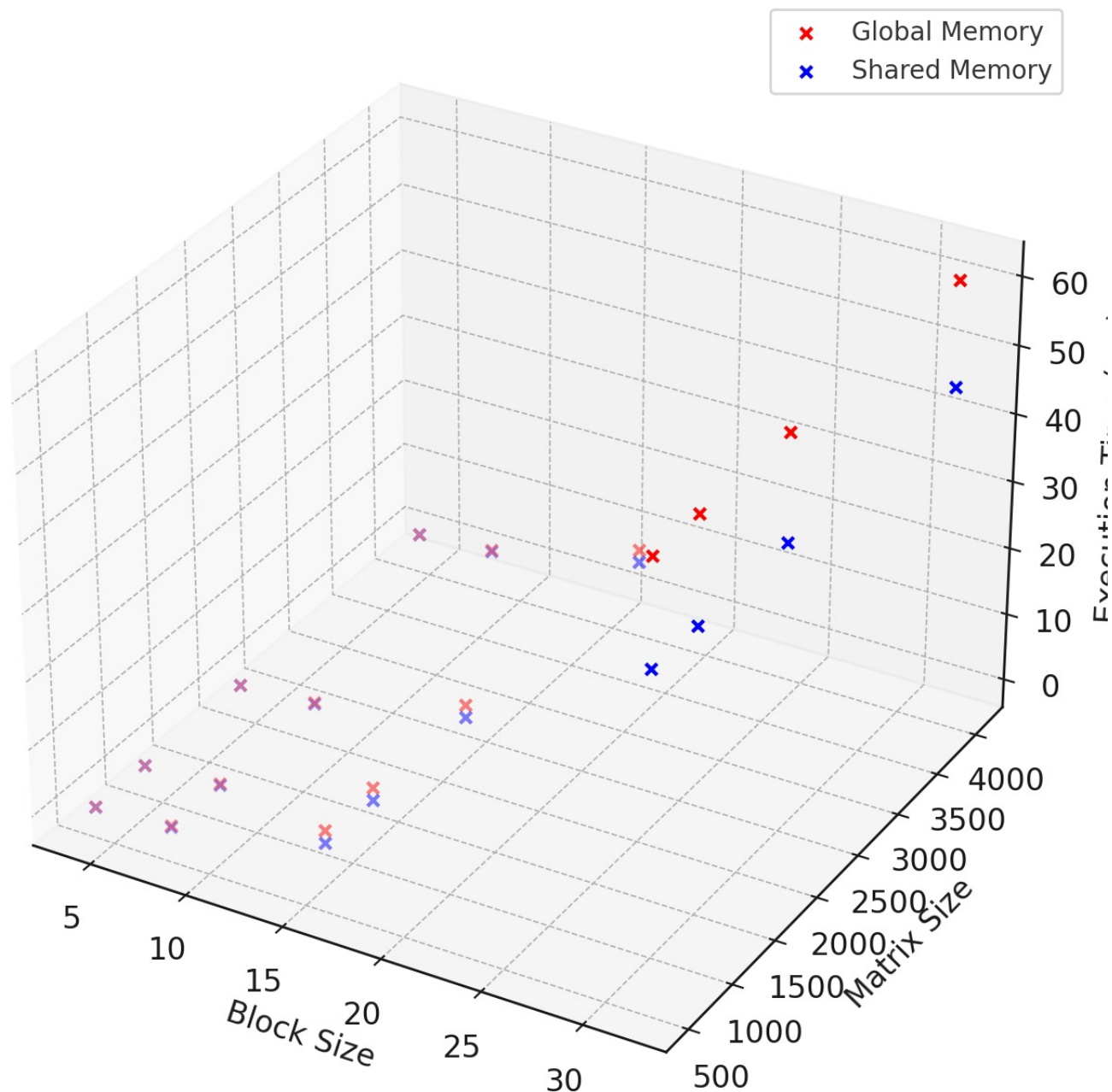
针对不同线程块大小，取 2048 矩阵

Dim Block Size	Global Mem	Shared Mem
4	20.1144	35.0394
8	9.51472	7.68102
16	7.61709	5.75898
32	8.42746	6.3191

针对矩阵规模，取 `dim(16, 16)`

Matrix Size	Global Mem	Shared Mem
512	0.130016	0.10016
1024	0.969728	0.745472
2048	7.61856	5.7559
4096	60.5256	45.0529

# 3D Plot of Execution Time vs. Block Size and Matrix Size



## 1. 线程块大小 (Thread Block Size)

项目使用不同的线程块大小来实现并行矩阵乘法。线程块的大小直接影响到核函数的执行效率和GPU资源的利用率。理想的线程块大小应该是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时间。选择过大或过小的线程块大小都可能导致性能不佳，因为太小可能导致GPU并行能力未被充分利用，太大则可能导致资源分配不均和更多的线程等待。

## 2. 矩阵规模 (Matrix Size)

矩阵的大小直接影响并行计算的复杂度。较大的矩阵需要更多的计算资源和内存带宽，但同时也提供了更高的并行度和可能的性能提升。矩阵规模的增加通常需要更精细的内存管理和计算策略，以避免内存访问瓶颈和资源浪费。

## 3. 访存方式 (Memory Access Patterns)

你的代码中提到了主要使用全局内存，并没有使用共享内存。全局内存的访问延迟较高，且如果访问模式不连续，可能导致显著的性能下降。使用共享内存可以显著减少访问延迟，特别是对于频繁访问的数据。优化内存访问模式，比如通过使用共享内存或优化全局内存访问的连续性，可以显著提高性能。

#### 4. 任务/数据划分方式 (Task/Data Partitioning)

合理的任务和数据划分对于提高CUDA程序的效率至关重要。数据应当按照能够最大化并行度和最小化跨线程通信的方式进行划分。不合理的划分可能导致某些核心空闲而其他核心过载，从而降低了整体的执行效率。