# 并行程序设计与算法实验 10

| 实验 | CUDA实现矩阵乘法 | 专业 | 计算机科学与技术 |
| --- | --- | --- | --- |
| 学号 | 21311525 | 姓名 | 马梓培 |
| Email | mazp@mail2.sysu.edu.cn | 完成日期 | 2024/05/27 |

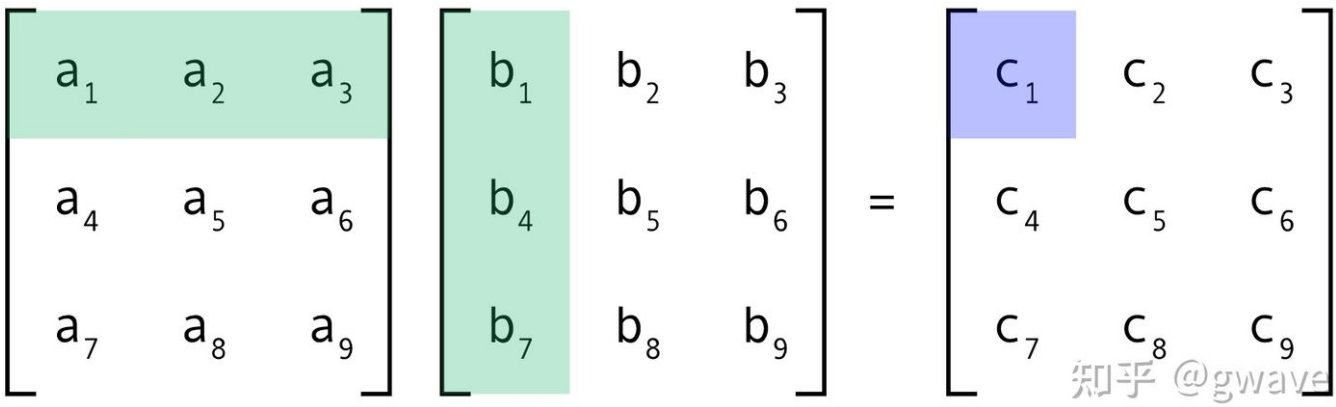
## 1. 实验要求

> - 使用CUDA实现并行通用矩阵乘法
> - 分析不同因素性能的影响
>   - 线程块大小、矩阵规模：如何提高占用率?
>   - 访存方式：何时使用何种存储?
>   - 数据/任务划分方式：按行，列，数据块划分，等

## 2. 矩阵乘法

代码详见附件或github

### 2.1 矩阵乘法的CPU实现




老生常谈了，代码：

```
void mm_cpu(float *A_host, float *B_host, float *C_host) {
   for (int i = 0; i < N; i ++) {
       for (int k = 0; k < N; k ++) {
           for (int j = 0; j < N; j ++) {
               C_host[i * N + j] += A_host[i * N + k] * B_host[k * N + j];}
       }
   }
}
```

一个小小的优化，我调换了`j`和`k`两个index，有助于提高cache的命中率。

## 2.2 Global Memory矩阵乘法

虽然, 师兄布置了是方阵的矩阵乘法。但是, 这里的分析, 用A.shape $= [m, k]$, B.shape $= [k, n], C.shape = [m, n]$来分析更加清楚。否则仅有一个N, 很容易混淆。

在 GPU 中执行矩阵乘法运算操作:

1. 在 Global Memory 中分别为矩阵 A、B、C 分配存储空间.
2. 由于矩阵 C 中每个元素的计算均相互独立, NVIDIA GPU 采用的 SIMT (单指令多线程)的体系结构来实现并行计算的, 因此在并行度映射中, 我们让每个 thread 对应矩阵 C 中1 个元素的计算.
3. 执行配置中 gridSize 和 blockSize 均有 $x$ (列向)、 $y$ (行向)两个维度. 其中,

$$\begin{aligned} \text{gridSize}.x \times \text{blockSize}.x &= n \\ \text{gridSize}.y \times \text{blockSize}.y &= m \end{aligned} \tag{4}$$

每个 thread 需要执行的 workflow 为：从矩阵 $A$ 中读取一行向量 (长度为k)，从矩阵 $B$ 中读取一列向量 (长度为 $k$ ), 对这两个向量做点积运算 (单层 k 次循环的乘累加), 最后将结果写回矩阵 C.

CUDA的kernel函数实现如下:

```
__global__ void mm_cuda(float *A_dev, float *B_dev, float *C_dev) {
    int nRow = blockIdx.y * blockDim.y + threadIdx.y;
    int nCol = blockIdx.x * blockDim.x + threadIdx.x;
    float c_sum = 0.f;

    for (int i = 0; i < N; i ++) {
        c_sum += A_dev[nRow * N + i] * B_dev[i * N + nCol];
    }
    C_dev[nRow * N + nCol] = c_sum;
}
```

下面来分析一下该 kernel 函数中 $A$、$B$、$C$ 三个矩阵对 global memory 的读取和写入情况:
读取 Global Memory:

- 对于矩阵 C 中每一个元素计算, 需要读取矩阵 A 中的一行元素;

对于矩阵 C 中同一行的 $n$ 个元素, 需要重复读取矩阵 $A$ 中同一行元素 $n$ 次;

- 对于矩阵 $C$ 中每一个元素计算, 需要读取矩阵 $B$ 中的一列元素;

对于矩阵 $C$ 中同一列的 $m$ 个元素, 需要重复读取矩阵 $B$ 中同一列元素 $m$ 次;
写入 Global Memory:

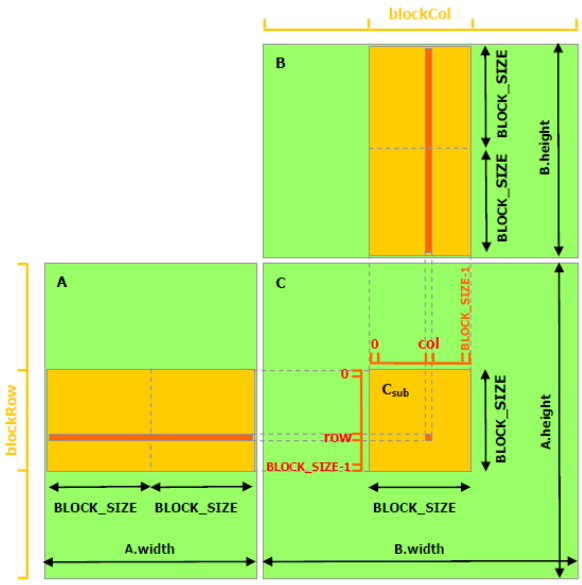
- 矩阵 C 中的所有元素只需写入一次.

由此可见:

- 对 A 矩阵重复读取 n 次, 共计 $m \times k \times n$ 次 32 bit Global Memory Load操作;
- 对 B 矩阵重复读取 m 次, 共计 $k \times n \times m$ 次 32 bit Global Memory Load操作;
- 对 C 矩阵共计 $m \times n$ 次 32 bit Global Memory Store操作.

读取的次数太多，global memory的速度又很慢，我们可以引入shared memory来提高提取速度。

## 2.3 Shared Memory 矩阵乘法




如上图所示, 利用 Shared Memory 优化 Global Memory 访问的基本原理是充分利用数据的局部性。具体方法是让一个 block 内的线程先从 Global Memory 中读取子矩阵块的数据（大小为 BLOCK_SIZE*BLOCK_SIZE）并写入 Shared Memory; 在计算过程中, 重复从 Shared Memory 读取数据进行乘法和累加, 从而避免每次都从 Global Memory 获取数据所带来的高延迟。然后, 让子矩阵块分别在矩阵 $\mathrm{A}$ 的行方向和矩阵 $\mathrm{B}$ 的列方向上滑动, 直到完成所有 $\mathrm{k}$ 个元素的乘法和累加。使用 Shared Memory 优化后的 kernel 代码如下所示:

```
__global__ void mm_shared_mem(float *A_dev, float *B_dev, float *C_dev) {
    int nRow = blockIdx.y * blockDim.y + threadIdx.y;
    int nCol = blockIdx.x * blockDim.x + threadIdx.x;

    float c_sum = 0.f;

    __shared__ float A_tile[block_size][block_size];
    __shared__ float B_tile[block_size][block_size];

    int nIter = (N + block_size - 1) / block_size;      // 将矩阵一个维度拆成 nIter 块
    for (int i = 0; i < nIter; i ++) {
        A_tile[threadIdx.y][threadIdx.x] = A_dev[nRow * N + i * block_size + threadIdx.x];
        B_tile[threadIdx.y][threadIdx.x] = B_dev[(i * block_size + threadIdx.y) * N + nCol];
        // 同步block中不同warp
        __syncthreads();

        for (int iter = 0; iter < block_size; iter ++) {
            c_sum += A_tile[threadIdx.y][iter] * B_tile[iter][threadIdx.x];
        }
        // 同步block中不同warp
        __syncthreads();
    }
    C_dev[nRow * N + nCol] = c_sum;
}
```

- 每个 block 可以看作是一个子矩阵 $C$, 并且是一个方阵;
- 从 Global Memory 读取的子矩阵 A 和子矩阵 B 的大小均等于子矩阵 C 的维度大小, 并存储在 Shared Memory 中;
- 子矩阵 $\mathrm{A}$ 在矩阵 $\mathrm{A}$ 的行方向上移动 $\mathrm{k}/\mathrm{BLOCK}SIZE$ 次, 子矩阵 B 在矩阵 B 的列方向上移动 $k$ / BLOCK_SIZE 次;
- 每个线程的计算过程由原先的单层 $\mathrm{k}$ 次循环, 变为两层循环：外层循环次数为 $\mathrm{k}/$ BLOCK_SIZE（假设能整除）, 任务是从 Global Memory 中读取数据到 Shared Memory 中; 内层循环次数为 BLOCK_SIZE, 任务是从 Shared Memory 中读取数据进行乘累加计算;
- 代码中有两次 _syncthreads() 操作：第一次在 Shared Memory 数据写入之后和计算开始之前进行同步, 确保所有线程都更新了 Shared Memory 中的数据; 第二次在计算完成后和 Shared Memory 写入之前进行同步, 确保 block 内所有线程的计算都已完成, 然后进行 Shared Memory 数据的更新。

# 3. 实验结果

## 3.1 正确性

在进行指标的测量前, 我们需要先验证一下程序的正确性。
我在`include/cuda_tool.h`中实现了 `checkResult()`用于比较两个结果 (CPU串行运算结果与CUDA并行运算结果) 是否相同。

简单验证一下, N = 128:

```
CPU                     Execution Time elapsed 0.007697 sec
CUDA                    Execution Time elapsed 0.000031 sec
CUDA(shared mem)        Execution Time elapsed 0.000012 sec
------------------------------------------------
CUDA:                   Check result success!
CUDA(shared mem):       Check result success!
```

Global Mem和Shared Mem实现的并行运算结果都是正确的。

## 3.2 线程块大小

默认矩阵大小为1024。

Block_size = 8:

```
CPU                     Execution Time elapsed 3.893570 sec
CUDA                    Execution Time elapsed 0.001276 sec
CUDA(shared mem)        Execution Time elapsed 0.001011 sec
------------------------------------------------
CUDA:                   Check result success!
CUDA(shared mem):       Check result success!
```

Block_size = 16:

```
CPU                     Execution Time elapsed 3.943985 sec
CUDA                    Execution Time elapsed 0.001034 sec
CUDA(shared mem)        Execution Time elapsed 0.000770 sec
-----------------------------------------------
CUDA:                   Check result success!
CUDA(shared mem):       Check result success!
```

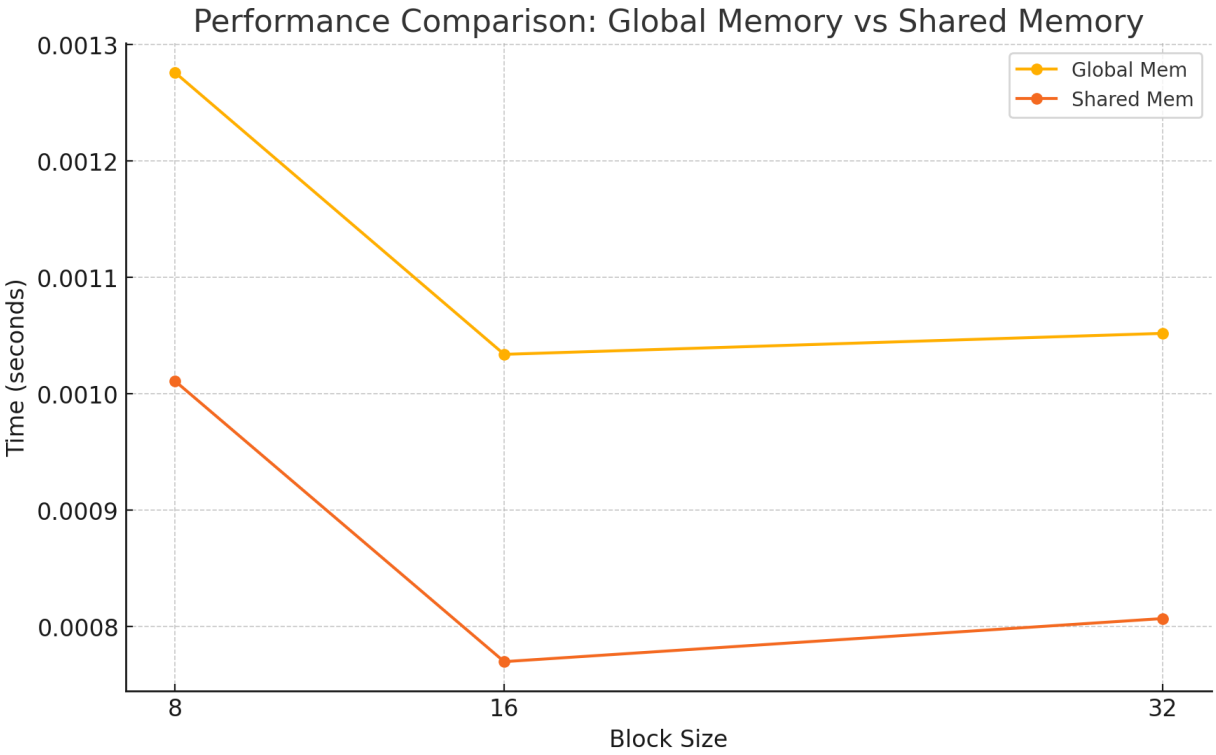
Block_size = 32:

```
CPU                     Execution Time elapsed 3.924863 sec
CUDA                    Execution Time elapsed 0.001052 sec
CUDA(shared mem)        Execution Time elapsed 0.000807 sec
-----------------------------------------------
CUDA:                   Check result success!
CUDA(shared mem):       Check result success!
```

表格：

| \Block size | 8 | 16 | 32 |
|---|---|---|---|
| CPU | 3.893570 | 3.943985 | 3.924863 |
| Global Mem | 0.001276 | 0.001034 | 0.001052 |
| Shared Mem | 0.001011 | 0.000770 | 0.000807 |

可视化（忽略CPU，不是一个数量级）：




可以看到，thanks to Shared Mem的快速访问，矩阵乘法的效率大大提高。此外，block size = 16时，global mem和shared mem访存时间都是最小的。

## 3.3 矩阵规模

默认线程块大小为8
N = 128:

```
CPU                     Execution Time elapsed 0.007890 sec
CUDA                    Execution Time elapsed 0.000042 sec
CUDA(shared mem)        Execution Time elapsed 0.000013 sec
-----------------------------------------------
CUDA:                   Check result success!
CUDA(shared mem):       Check result success!
```

N = 256:

```
CPU                     Execution Time elapsed 0.060837 sec
CUDA                    Execution Time elapsed 0.000053 sec
CUDA(shared mem)        Execution Time elapsed 0.000025 sec
-----------------------------------------------
CUDA:                   Check result success!
CUDA(shared mem):       Check result success!
```

N = 512:

```
CPU                     Execution Time elapsed 0.485761 sec
CUDA                    Execution Time elapsed 0.000171 sec
CUDA(shared mem)        Execution Time elapsed 0.000111 sec
-----------------------------------------------
CUDA:                   Check result success!
CUDA(shared mem):       Check result success!
```

N = 1024:

```
CPU                     Execution Time elapsed 3.934249 sec
CUDA                    Execution Time elapsed 0.001060 sec
CUDA(shared mem)        Execution Time elapsed 0.000773 sec
-----------------------------------------------
CUDA:                   Check result success!
CUDA(shared mem):       Check result success!
```

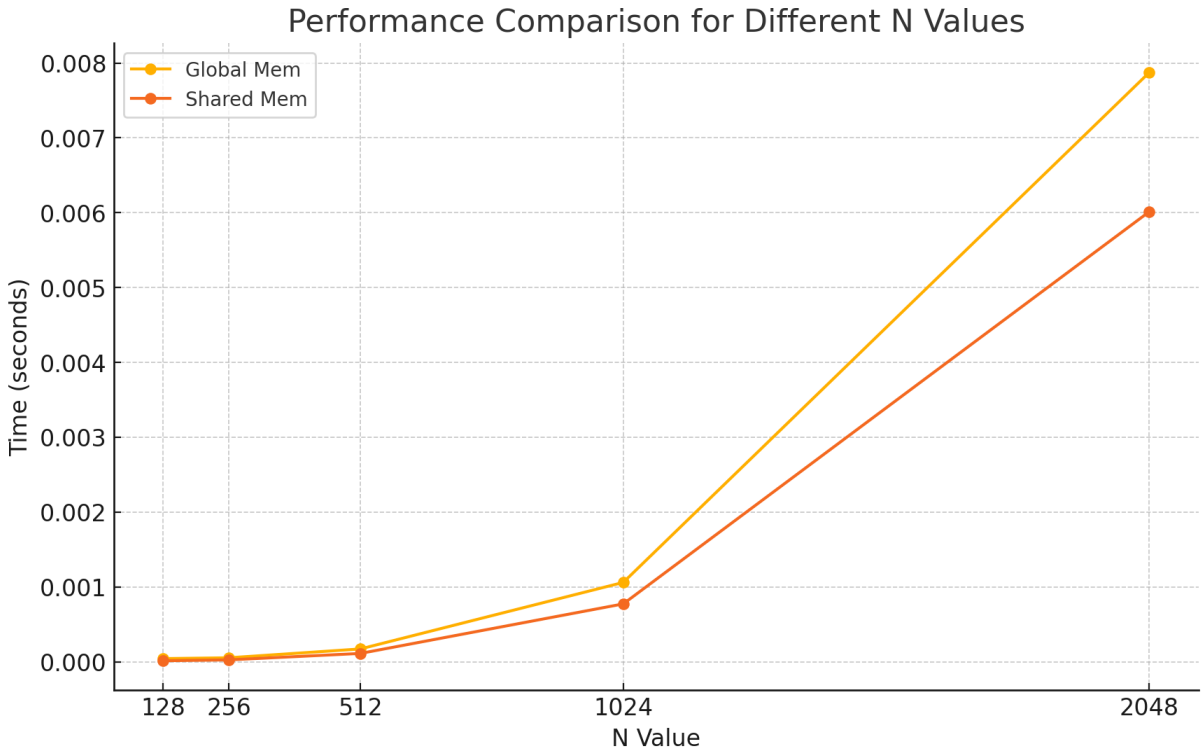
N = 2048:

```
CPU                     Execution Time elapsed 31.048180 sec
CUDA                    Execution Time elapsed 0.007873 sec
CUDA(shared mem)        Execution Time elapsed 0.006013 sec
-----------------------------------------------
CUDA:                   Check result success!
CUDA(shared mem):       Check result success!
```

表格：

| \N | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|
| CPU | 0.007890 | 0.060837 | 0.485761 | 3.934249 | 31.048180 |
| Global Mem | 0.000042 | 0.000053 | 0.000171 | 0.001060 | 0.007873 |
| Shared Mem | 0.000013 | 0.000025 | 0.000111 | 0.000773 | 0.006013 |

可视化（同样，忽略了CPU）：




可以看到，shared mem的访存方式是一直优于global mem的。并且，随着矩阵规模的增大，差距越来越明显。

# 4. 实验感想

在这次并行程序设计与算法实验中，我通过使用CUDA实现了矩阵乘法，深入理解了并行计算的基本概念和实际应用。实验中，分别采用了全局内存和共享内存两种方式进行矩阵乘法计算，并通过调整线程块大小和矩阵规模，详细分析了这些因素对程序性能的影响。实验结果表明，使用共享内存显著提升了计算效率，特别是在处理大规模矩阵时，性能优势更加明显。

通过这次实验，我不仅掌握了CUDA编程的基本技巧，还认识到在并行计算中，合理的存储方式和任务划分对提升性能至关重要。这些经验为我今后在并行计算领域的学习和研究提供了宝贵的实践参考。同时，通过实验中的优化尝试，我也更深刻地理解了计算机硬件和软件之间的协同作用。