# 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 |

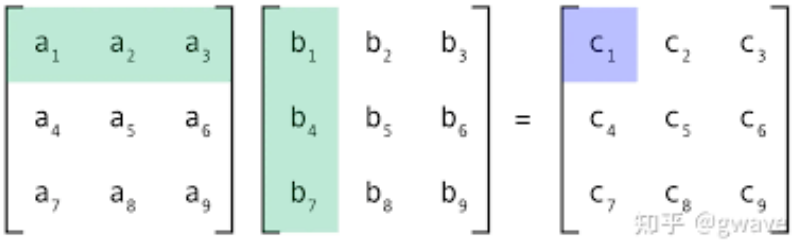
## 1. 实验要求

> - 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,

$$\begin{aligned} \text{gridSize}.x \times \text{blockSize}.x &= n \\ \text{gridSize}.y \times \text{blockSize}.y &= m \end{aligned} \tag{1}$$

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 $\mathrm{k}$ 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:

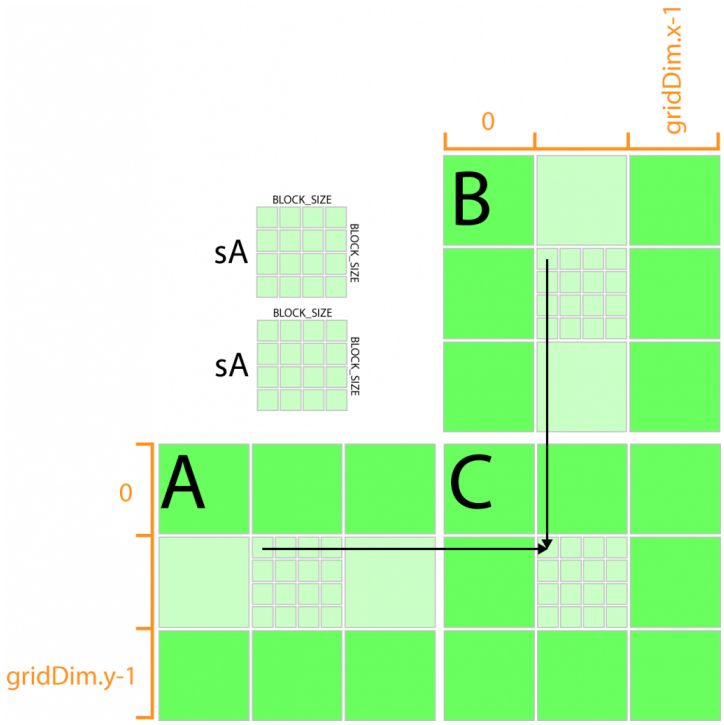
- 矩阵 $\mathrm{C}$ 中的所有元素只需写入一次.

由此可见:

- 对 A 矩阵重复读取 $\mathrm{n}$ 次, 共计 $m \times k \times n$ 次 32bit Global Memory Load操作;
- 对 $\mathrm{B}$ 矩阵重复读取 $\mathrm{m}$ 次, 共计 $k \times n \times m$ 次 32bit Global Memory Load操作;
- 对 C 矩阵共计 $m \times n$ 次 32bit 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 A 的行方向和矩阵 B 的列方向上滑动，直到完成所有 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 A 在矩阵 A 的行方向上移动 k / 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 k 次循环，变为两层循环：外层循环次数为 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:

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP10$ ./mm_cuda.out
CPU                     Execution Time elapsed 3.850786 sec
CUDA                    Execution Time elapsed 0.000076 sec
CUDA(shared mem)        Execution Time elapsed 0.000009 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:

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP10$ ./mm_cuda.out
CPU                     Execution Time elapsed 3.850786 sec
CUDA                    Execution Time elapsed 0.000076 sec
CUDA(shared mem)        Execution Time elapsed 0.000009 sec
```

Block_size = 16:

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP10$ ./mm_cuda.out
CPU                     Execution Time elapsed 3.855901 sec
CUDA                    Execution Time elapsed 0.000081 sec
CUDA(shared mem)        Execution Time elapsed 0.000007 sec
```

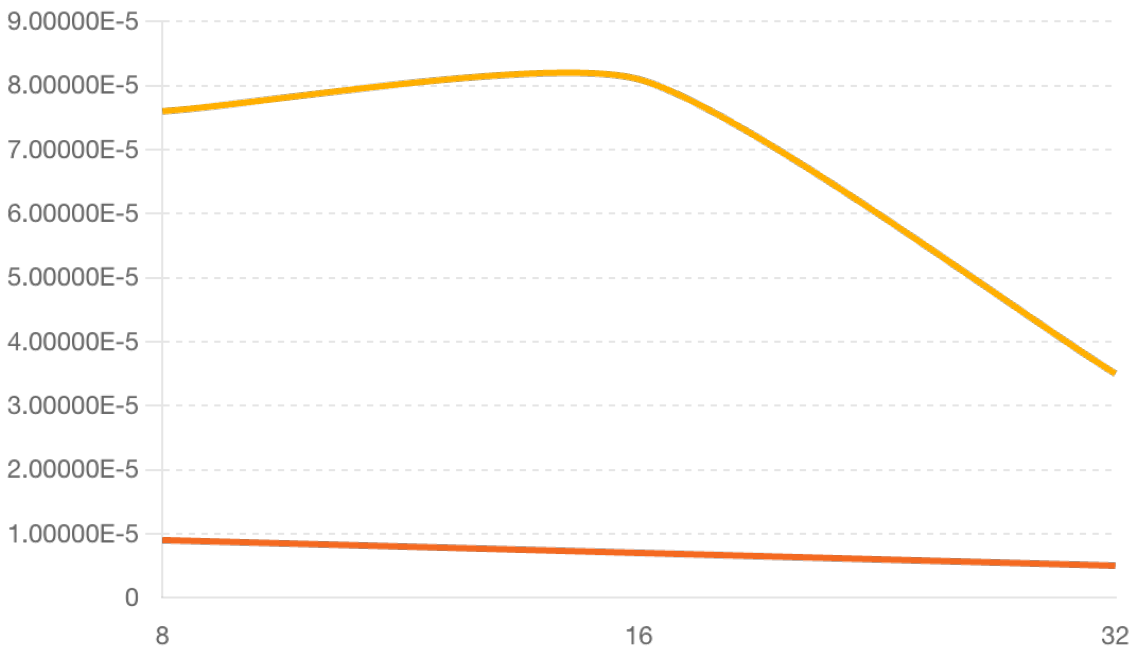
Block_size = 32:

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP10$ ./mm_cuda.out
CPU                     Execution Time elapsed 3.858319 sec
CUDA                    Execution Time elapsed 0.000035 sec
CUDA(shared mem)        Execution Time elapsed 0.000005 sec
```

表格：

| \Block size | 8 | 16 | 32 |
|---|---|---|---|
| Global Mem | 0.000076 | 0.000081 | 0.000035 |
| Shared Mem | 0.000009 | 0.000007 | 0.000005 |

可视化：



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，随着block size变大，矩阵乘法的效率也同样增大了。

## 3.3 矩阵规模

默认线程块大小为8

N = 128:

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP10$ ./mm_cuda.out
CPU                     Execution Time elapsed 0.007748 sec
CUDA                    Execution Time elapsed 0.000022 sec
CUDA(shared mem)        Execution Time elapsed 0.000005 sec
```

N = 256:

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP10$ ./mm_cuda.out
CPU                     Execution Time elapsed 0.060277 sec
CUDA                    Execution Time elapsed 0.000029 sec
CUDA(shared mem)        Execution Time elapsed 0.000004 sec
```

N = 512:

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP10$ ./mm_cuda.out
CPU                     Execution Time elapsed 0.491059 sec
CUDA                    Execution Time elapsed 0.000022 sec
CUDA(shared mem)        Execution Time elapsed 0.000006 sec
```

N = 1024:

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP10$ ./mm_cuda.out
CPU                     Execution Time elapsed 3.890929 sec
CUDA                    Execution Time elapsed 0.000097 sec
CUDA(shared mem)        Execution Time elapsed 0.000005 sec
```

N = 2048:

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP10$ ./mm_cuda.out
CPU                     Execution Time elapsed 30.761449 sec
CUDA                    Execution Time elapsed 0.000077 sec
CUDA(shared mem)        Execution Time elapsed 0.000006 sec
```

表格：

| \N | 128 | 256 | 512 | 1024 | 2048 |
| --- | --- | --- | --- | --- | --- |
| Global Mem | 0.000022 | 0.000029 | 0.000022 | 0.000097 | 0.000077 |
| Shared Mem | 0.000005 | 0.000004 | 0.000006 | 0.000005 | 0.000006 |

可视化：

可以看到，随着矩阵的规模增大。CPU的用时确实是增大的。但是，出于cuda的多线程，“核多力量大”，时间几乎没怎么增大。

# 4. 实验感想

在本次并行程序设计与算法实验中，通过使用CUDA实现并行矩阵乘法，并分析了不同因素对性能的影响，我有以下几点感想。首先，使用Shared Memory优化Global Memory访问显著提高了矩阵乘法的效率。Shared Memory的快速访问显著减少了数据从Global Memory读取的次数，从而降低了内存访问延迟，尤其是在计算较大规模的矩阵时，这一优化效果尤为明显。通过调整线程块的大小，我发现随着块大小的增加，矩阵乘法的执行时间明显缩短。这是因为较大的块大小可以更好地利用共享内存，从而减少全局内存的访问次数，提高计算效率。

此外，我还测试了不同矩阵规模下的执行时间。结果显示，随着矩阵规模的增大，CPU的执行时间显著增加，但CUDA并行计算的执行时间几乎没有显著增加，这得益于CUDA的多线程并行计算能力，使得其在处理大规模矩阵时依然保持高效。在实现CUDA矩阵乘法的过程中，如何有效地划分数据和任务、合理使用同步机制等都是关键挑战。通过对代码进行优化，尤其是对共享内存的利用，以及正确使用`__syncthreads()`函数，确保了数据在共享内存中的正确性和同步性，从而实现了高效的并行计算。这次实验不仅加深了我对CUDA并行编程的理解，也让我认识到了共享内存在提高计算效率中的重要作用，为今后在实际项目中应用这些知识打下了坚实的基础。