中山大学数据科学与计算机院本科生实验报告

(2020 秋季学期)

课程名称：高性能程序设计　**任课老师**：黄聃　**批改人**：

年级 + 班级	18 级计算机	年级 (方向)	计算机科学
学号	18340236	姓名	朱煜
Email	zhuy85@mail2.sysu.edu.cn	完成日期	2020.12.14

1 实验目的

1.1 任务 1

通过 CUDA 实现通用矩阵乘法（Lab1）的并行版本，CUDA Thread Block size 从 32 增加至 512，矩阵规模从 512 增加至 8192。

通用矩阵乘法 **GEMM** 通常定义为：

$$C = AB$$

$$C_{m,n} = \sum_{n=1}^{N} A_{m,n} B_{n,k}$$

输入：M，N，K 三个整数 (512∼8192)

问题描述：随机生成 M * N 和 N * K 的两个矩阵 A，B，对这两个矩阵做乘法得到矩阵 C

输出：A，B，C 三个矩阵以及矩阵计算的时间

1.2 任务 2

将任务 1 改造成基于 OpenMP+CUDA 的多层次并行矩阵乘法。矩阵被主进程切分成子矩阵分配给 OpenMP 并行线程计算，并行进程调用任务 1 的 CUDA 版本矩阵乘法计算子矩阵，汇总并行进程的计算结果，并打印结果和运行时间，并行线程数：1，2，4，8。

1.3 任务 3

通过 NVDIA 的矩阵计算函数库 CUBLAS 计算矩阵相乘，矩阵规模从 512 增加至 8192，并与任务 1 和任务 2 的矩阵乘法进行性能比较和分析，如果性能不如 CUBLAS，思考并文字描述可能的改进方法（参考《计算机体系结构-量化研究方法》第四章）。

CUBLAS 参考资料《CUBLAS_Library.pdf》，CUBLAS 矩阵乘法参考第 70 页内容。CUBLAS 矩阵乘法例子，参考附件《matrixMulCUBLAS》

2 实验过程

2.1 任务 1

cuda 通过将一组线程组成一个 thread block，并一次执行若干个 thread block 的形式来进行并行计算，形成一种网格式的并行结构。在矩阵计算中，计算结果 C 矩阵为 $M * K$ 的矩阵，要想最终结果的每个元素采用一个 thread 进行计算，设 thread block 的线程数为 $ThreadBlockSize$，则对 C 矩阵进行划分后的块大小也应该为 $ThreadBlockSize$，为了方便说明,这里假设 K 可以被 $ThreadBlockSize$ 整除,则可简单地将每一行按 $ThreadBlockSize$ 拆分成块，则每一块中都有 $ThreadBlockSize$ 数目的线程与对应所计算的唯一元素，这样每个线程只需计算一个位置的元素值，划分方式如图1。

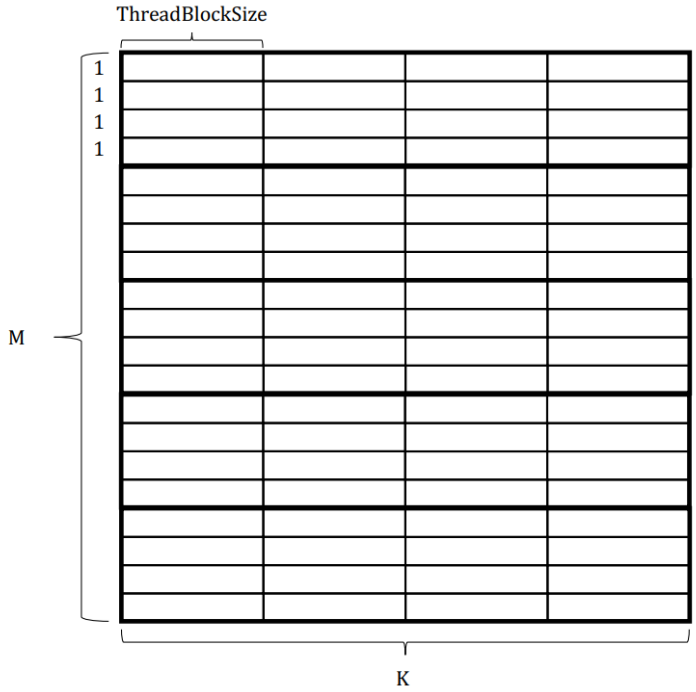

图 1: C 矩阵块的划分

根据块的划分形式，设计出我们的核函数，每个线程根据自己所属的块号与线程号得到对应元素的位置，根据位置的行号与列号直接计算。

```
1 __global__ void MatrixMulCUDA(const float *A, const float *B, float *C, int
  ↪  m, int n, int k, int ThreadBlockSize)
2 {
```

```
    //      计算元素的行号 由于每行都进行划分结果直接为网格的 x 值
    const int row = blockIdx.x;
    //      计算元素的列号 每个块有 ThreadBlockSize 个元素 则计算网格的第
    ↪  blockIdx.y 块的第 threadIdx.x 个线程
    const int col = blockIdx.y * ThreadBlockSize + threadIdx.x;
    float temp = 0;
    if (row < m && col < k)
    {
        for (int i = 0; i < n; ++i)
            temp += A[row * n + i] * B[i * k + col];
        C[row * k + col] = temp;
    }
}
```

按照设计的核函数，简单地将生成的 A B 矩阵从 CPU 转移到 GPU 中，根据如上的划分方式调用核函数，再将结果 C 矩阵从 GPU 转回我们的 CPU 中，得到最后的结果，实现代码如下。

```
    //      申请空间
    cudaMalloc((void **)&cuda_A, sizeof(float) * m * n);
    cudaMalloc((void **)&cuda_B, sizeof(float) * n * k);
    cudaMalloc((void **)&cuda_C, sizeof(float) * m * k);
    //      将 A、B 矩阵从 CPU 转移到 GPU
    cudaMemcpy(cuda_A, A, sizeof(float) * m * n, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_B, B, sizeof(float) * n * k, cudaMemcpyHostToDevice);
    //      定义的结构网格
    dim3 grid(m, k / ThreadBlockSize);
    //      矩阵乘法
    MatrixMulCUDA<<<grid, ThreadBlockSize>>>(cuda_A, cuda_B, cuda_C, m, n,
    ↪  k, ThreadBlockSize);

    //      将结果 C 矩阵从 GPU 转移回 CPU
    cudaMemcpy(C, cuda_C, sizeof(float) * m * k, cudaMemcpyDeviceToHost);
    cudaFree(cuda_A);
    cudaFree(cuda_B);
    cudaFree(cuda_C);
```

矩阵的初始化与打印代码与之前相同,此处不做赘述,详细见源码文件"MatrixMul_cuda.cu"。为验证其正确性，选择小规模的矩阵进行计算并打印结果，如图2，可知以上实现的矩阵运算正确。

```
jovyan@jupyter-zhuzhuzhu:~/ZY/code$ ./MatrixMul_cuda 8 8 8 4
Calculation time is 0.3445119858 ms
Matrix A:
1.000000 7.000000 0.000000 7.000000 5.000000 7.000000 1.000000 3.000000
6.000000 1.000000 5.000000 4.000000 5.000000 7.000000 5.000000 4.000000
6.000000 0.000000 7.000000 1.000000 8.000000 8.000000 6.000000 6.000000
8.000000 8.000000 8.000000 4.000000 1.000000 1.000000 5.000000 0.000000
0.000000 3.000000 5.000000 3.000000 1.000000 7.000000 4.000000 7.000000
6.000000 0.000000 0.000000 2.000000 5.000000 4.000000 5.000000 2.000000
2.000000 3.000000 2.000000 1.000000 1.000000 8.000000 8.000000 0.000000
5.000000 5.000000 4.000000 4.000000 6.000000 0.000000 5.000000 6.000000
Matrix B:
2.000000 8.000000 7.000000 3.000000 4.000000 2.000000 0.000000 0.000000
0.000000 0.000000 2.000000 6.000000 2.000000 5.000000 6.000000 5.000000
7.000000 6.000000 6.000000 8.000000 5.000000 3.000000 6.000000 2.000000
8.000000 1.000000 6.000000 6.000000 8.000000 0.000000 1.000000 1.000000
7.000000 0.000000 3.000000 2.000000 0.000000 1.000000 2.000000 1.000000
8.000000 3.000000 5.000000 2.000000 6.000000 0.000000 7.000000 2.000000
7.000000 2.000000 8.000000 1.000000 6.000000 5.000000 1.000000 5.000000
4.000000 6.000000 0.000000 4.000000 6.000000 2.000000 3.000000 2.000000
Matrix C:
168.000000 56.000000 121.000000 124.000000 140.000000 53.000000 118.000000 72.000000
221.000000 137.000000 188.000000 133.000000 179.000000 70.000000 116.000000 71.000000
255.000000 163.000000 202.000000 142.000000 187.000000 83.000000 139.000000 81.000000
154.000000 129.000000 192.000000 169.000000 156.000000 106.000000 114.000000 88.000000
178.000000 104.000000 124.000000 124.000000 163.000000 65.000000 127.000000 77.000000
138.000000 84.000000 129.000000 61.000000 106.000000 46.000000 51.000000 44.000000
153.000000 69.000000 145.000000 72.000000 128.000000 66.000000 97.000000 77.000000
171.000000 114.000000 151.000000 142.000000 148.000000 90.000000 93.000000 80.000000
```

图 2: MatrixMul_cuda 运行结果

调节线程数与矩阵规模进行输入，测试时输入的矩阵为方阵，则矩阵规模为 512 时表示对应的 m, n, k 为都为 512。其运算时间结果（单位：ms）如表1。可见在相同矩阵规模下，Thread Block size 的大小并不影响运算时间。这是由于采用的划分方式为每一个线程计算一个矩阵位置的值，则无论 Thread Block size 的大小为多少，每个线程分配到的计算量都是相同的。而矩阵规模的扩大，会扩大每个线程在计算对应位置元素值的循环的大小，因此运算时间大致随着规模增大呈三次函数增大。

表 1: MatrixMul_cuda 运算时间 (ms) 与 Thread Block size、矩阵规模的关系

矩阵规模 / Thread Block size	**512**	**1024**	**2048**	**4096**	**8192**
32	2.72	5.93	35.41	221.68	1574.29
64	2.62	5.76	35.84	245.56	2462.93
128	2.64	6.01	40.89	282.26	1997.61
256	2.65	6.54	39.61	281.74	2167.25
512	2.62	6.57	44.37	291.69	4225.00

2.2 任务 2

2.2.1 按行划分

由于任务 1 中 C 矩阵块行的划分方式为逐行划分，因此考虑在使用 OpenMP 划分矩阵时可使用按行划分，这样可简单地使用任务 1 的矩阵运算核函数，A 矩阵划分方式如图3。

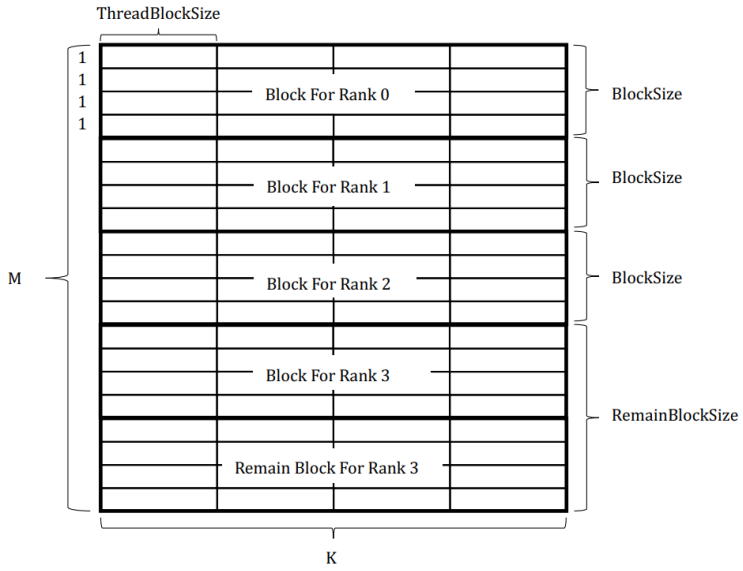

图 3: A 矩阵按行划分

由于 OpenMP 为共享内存，则只需在各线程运行时计算出划分矩阵块的初始起点与元素个数则可简单地进行矩阵的划分，实现代码如下。

```
    int tid;
    int block_size = block_size = m / thread_count;
    omp_set_num_threads(thread_count);
    int my_block_size;
    float elapsedTime = 0;
#pragma omp parallel private(tid, my_block_size)
    {
        float my_elapsedTime = 0;
        tid = omp_get_thread_num();
        my_block_size = block_size;
//          最后的线程要计算矩阵划分的剩余部分
```

```
12          if ((thread_count * block_size < m) && (tid == thread_count - 1))
13              my_block_size = m - (thread_count - 1) * block_size;
14
15          float * cuda_A_block, *cuda_B, *cuda_C_block;
16          cudaEvent_t start, stop;
17          cudaEventCreate(&start);
18          cudaEventCreate(&stop);
19          cudaEventRecord(start, 0);
20
21          cudaMalloc((void **)&cuda_A_block, sizeof(float) * my_block_size *
            ↪  n);
22          cudaMalloc((void **)&cuda_B, sizeof(float) * n * k);
23          cudaMalloc((void **)&cuda_C_block, sizeof(float) * my_block_size *
            ↪  k);
24  //          根据线程号拷贝 A 矩阵
25          cudaMemcpy(cuda_A_block, A + (block_size * tid) * n, sizeof(float)
            ↪  * my_block_size * n, cudaMemcpyHostToDevice);
26  //          B 矩阵全部拷贝
27          cudaMemcpy(cuda_B, B, sizeof(float) * n * k,
            ↪  cudaMemcpyHostToDevice);
28  //          采用与任务 1 相同的网格定义
29          dim3 grid(my_block_size , k / ThreadBlockSize);
30  //          矩阵乘法
31          MatrixMulCUDA<<<grid, ThreadBlockSize>>>(cuda_A_block, cuda_B,
            ↪  cuda_C_block, my_block_size, n, k, ThreadBlockSize);
32  //          返回结果
33          cudaMemcpy(C + (block_size * tid) * k, cuda_C_block, sizeof(float)
            ↪  * my_block_size * k, cudaMemcpyDeviceToHost);
34          cudaFree(cuda_A_block);
35          cudaFree(cuda_B);
36          cudaFree(cuda_C_block);
37          cudaEventRecord(stop, 0);
38          cudaEventSynchronize(stop);
39          cudaEventElapsedTime(&my_elapsedTime, start, stop);
40  //          计算各线程最长的运算时间
41          #pragma omp critical
```

```
42          {
43              if (my_elapsedTime > elapsedTime)
44                  elapsedTime = my_elapsedTime;
45          }
46      }
```

矩阵的初始化与打印代码与之前相同,此处不做赘述,详细见源码文件"MatrixMul_omp.cu"。为验证其正确性，选择小规模的矩阵进行计算并打印结果，如图4，可知以上实现的矩阵运算正确。

```
jovyan@jupyter-zhuzhuzhu:~/ZY/code$ ./MatrixMul_omp 8 8 8 4 4
Calculation time is 0.7417920232 ms
Matrix A:
1.000000 7.000000 0.000000 7.000000 5.000000 7.000000 1.000000 3.000000
6.000000 1.000000 5.000000 4.000000 5.000000 7.000000 5.000000 4.000000
6.000000 0.000000 7.000000 1.000000 8.000000 8.000000 6.000000 6.000000
8.000000 8.000000 8.000000 4.000000 1.000000 1.000000 5.000000 0.000000
0.000000 3.000000 5.000000 3.000000 1.000000 7.000000 4.000000 7.000000
6.000000 0.000000 0.000000 2.000000 5.000000 4.000000 5.000000 2.000000
2.000000 3.000000 2.000000 1.000000 1.000000 8.000000 8.000000 0.000000
5.000000 5.000000 4.000000 4.000000 6.000000 0.000000 5.000000 6.000000
Matrix B:
2.000000 8.000000 7.000000 3.000000 4.000000 2.000000 0.000000 0.000000
0.000000 0.000000 2.000000 6.000000 2.000000 5.000000 6.000000 5.000000
7.000000 6.000000 6.000000 8.000000 5.000000 3.000000 6.000000 2.000000
8.000000 1.000000 6.000000 6.000000 8.000000 0.000000 1.000000 1.000000
7.000000 0.000000 3.000000 2.000000 0.000000 1.000000 2.000000 1.000000
8.000000 3.000000 5.000000 2.000000 6.000000 0.000000 7.000000 2.000000
7.000000 2.000000 8.000000 1.000000 6.000000 5.000000 1.000000 5.000000
4.000000 6.000000 0.000000 4.000000 6.000000 2.000000 3.000000 2.000000
Matrix C:
168.000000 56.000000 121.000000 124.000000 140.000000 53.000000 118.000000 72.000000
221.000000 137.000000 188.000000 133.000000 179.000000 70.000000 116.000000 71.000000
255.000000 163.000000 202.000000 142.000000 187.000000 83.000000 139.000000 81.000000
154.000000 129.000000 192.000000 169.000000 156.000000 106.000000 114.000000 88.000000
178.000000 104.000000 124.000000 124.000000 163.000000 65.000000 127.000000 77.000000
138.000000 84.000000 129.000000 61.000000 106.000000 46.000000 51.000000 44.000000
153.000000 69.000000 145.000000 72.000000 128.000000 66.000000 97.000000 77.000000
171.000000 114.000000 151.000000 142.000000 148.000000 90.000000 93.000000 80.000000
```

图 4: MatrixMul_omp 运行结果

调节线程数与矩阵规模进行输入，固定 Thread Block size 大小为 32。测试时输入的矩阵为方阵，则矩阵规模为 512 时表示对应的 m, n, k 为都为 512。其运算时间结果（单位：ms）如表2。可见在相同矩阵规模下，线程数的大小并不影响运算时间。这是由于采用的划分方式为每一个线程计算一个矩阵位置的值，则无论线程数的大小为多少，虽然矩阵经过了 OpenMP 的划分，但每个线程分配到的计算量仍是相同的。一个位置的计算量仍是 A 矩阵的一行乘以 B 矩阵的一列，而行的大小并不会因为矩阵按行划分而变小，因此以下考虑对矩阵按列重新划分。

表 2: MatrixMul_omp 运算时间 (ms) 与线程数、矩阵规模的关系

线程数 \ 矩阵规模	**512**	**1024**	**2048**	**4096**	**8192**
1	3.28	5.82	35.52	222.94	1574.77
2	5.62	9.83	24.55	241.75	1638.74
4	7.33	16.10	50.59	248.26	1335.38
8	4.48	27.41	76.23	272.96	1189.66

2.2.2 按列划分

根据以上按行划分的分析，由于采用的 cuda 并行为每一 cuda 线程计算一个矩阵位置的数值，计算方法为 A 矩阵的一行乘以 B 矩阵的一列，因此要想提高 $OpenMP$ 的并行速度，需要减少 cuda 每一线程的计算量。考虑将 A 矩阵按列划分，同时 B 矩阵按行划分，每一部分的矩阵块对应相乘，则 cuda 线程的计算量减少，最后将 OpenMP 的线程计算结果累加起来，则是最终的 C 矩阵运算结果，A 矩阵划分方式如图5

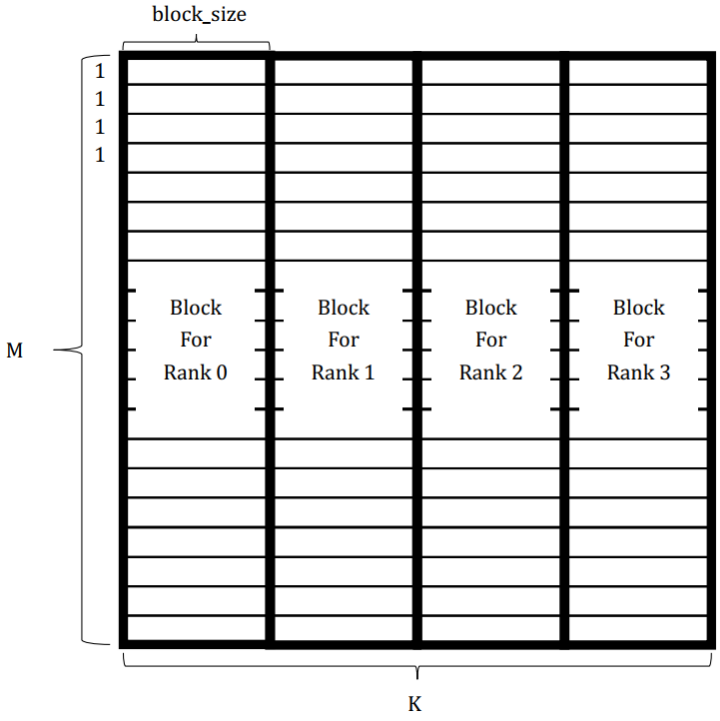

图 5: A 矩阵按列划分

由于矩阵保存时按行连续存储，因此需要将矩阵的列取出重新排序，为了提高效率，在

cuda 中实现矩阵的按列划分。简单地为矩阵的每一位置分配一个 cuda 线程，线程判断是否属于划分矩阵块的列中，进行赋值，实现代码如下：

```
__global__ void MatrixDivideCUDA(const float *matrix, float *result_matrix,
↪  int m, int n, int col_begin, int my_block_size)
{
    const int row = blockIdx.x;
    const int col = blockIdx.y;
    //    判断是否在划分的块内，根据计算得到新的索引
    if (row < m && col < col_begin + my_block_size && col >= col_begin)
        result_matrix[row * my_block_size + col - col_begin] += matrix[row
        ↪  * n + col];
}
```

与 A 矩阵的划分对应，B 矩阵需要按行进行划分，这样 cuda 每个线程计算量便减少，增加了将最后结果累加的步骤，为了提高并行效率，使用 cuda 实现各结果矩阵的累加。由于每一个 OpenMP 线程需要累加 CPU 中 C 矩阵的结果，因此需要考虑互斥问题，实现代码如下。

```
__global__ void MatrixAddCUDA(const float *matrix, float *result_matrix,
↪  int m, int k)
{
    const int row = blockIdx.x;
    const int col = blockIdx.y;
    // 矩阵求和
    if (row < m && col < k)
        result_matrix[row * k + col] += matrix[row * k + col];
}


        #pragma omp critical
        {
            float *cuda_C_now;
            cudaMalloc((void **)&cuda_C_now, sizeof(float) * m * k);
            cudaMemcpy(cuda_C_now, C, sizeof(float) * m * k,
            ↪  cudaMemcpyHostToDevice);
```

```
16              dim3 grid_temp(m, k);
17              MatrixAddCUDA<<<grid_temp, 1>>>(cuda_C_now, cuda_C, m, k);
18              cudaMemcpy(C, cuda_C, sizeof(float) * m * k,
                ↪  cudaMemcpyDeviceToHost);
19              cudaFree(cuda_C_now);
20          }
```

矩阵的初始化与打印代码与之前相同,此处不做赘述,详细见源码文件"MatrixMul_omp_2.cu"。为验证其正确性，选择小规模的矩阵进行计算并打印结果，如图6，可知以上实现的矩阵运算正确。

```
jovyan@jupyter-zhuzhuzhu:~/ZY/code$ ./MatrixMul_omp_2 8 8 8 4 4
Calculation time of thread 1 is 4.5445117950 ms
Calculation time of thread 0 is 0.0102399997 ms
Calculation time of thread 3 is 4.5414400101 ms
Calculation time of thread 2 is 0.0081920000 ms
Total calculation time is 18.2580795288 ms
Matrix A:
1.000000 7.000000 0.000000 7.000000 5.000000 7.000000 1.000000 3.000000
6.000000 1.000000 5.000000 4.000000 5.000000 7.000000 5.000000 4.000000
6.000000 0.000000 7.000000 1.000000 8.000000 8.000000 6.000000 6.000000
8.000000 8.000000 8.000000 4.000000 1.000000 1.000000 5.000000 0.000000
0.000000 3.000000 5.000000 3.000000 1.000000 7.000000 4.000000 7.000000
6.000000 0.000000 0.000000 2.000000 5.000000 4.000000 5.000000 2.000000
2.000000 3.000000 2.000000 1.000000 1.000000 1.000000 8.000000 8.000000 0.000000
5.000000 5.000000 4.000000 4.000000 6.000000 0.000000 5.000000 6.000000
Matrix B:
2.000000 8.000000 7.000000 3.000000 4.000000 2.000000 0.000000 0.000000
0.000000 0.000000 2.000000 6.000000 2.000000 5.000000 6.000000 5.000000
7.000000 6.000000 6.000000 8.000000 5.000000 3.000000 6.000000 2.000000
8.000000 1.000000 6.000000 6.000000 8.000000 0.000000 1.000000 1.000000
7.000000 0.000000 3.000000 2.000000 0.000000 1.000000 2.000000 1.000000
8.000000 3.000000 5.000000 2.000000 6.000000 0.000000 7.000000 2.000000
7.000000 2.000000 8.000000 1.000000 6.000000 5.000000 1.000000 5.000000
4.000000 6.000000 0.000000 4.000000 6.000000 2.000000 3.000000 2.000000
Matrix C:
168.000000 56.000000 121.000000 124.000000 140.000000 53.000000 118.000000 72.000000
221.000000 137.000000 188.000000 133.000000 179.000000 70.000000 116.000000 71.000000
255.000000 163.000000 202.000000 142.000000 187.000000 83.000000 139.000000 81.000000
154.000000 129.000000 192.000000 169.000000 156.000000 106.000000 114.000000 88.000000
178.000000 104.000000 124.000000 124.000000 163.000000 65.000000 127.000000 77.000000
138.000000 84.000000 129.000000 61.000000 106.000000 46.000000 51.000000 44.000000
153.000000 69.000000 145.000000 72.000000 128.000000 66.000000 97.000000 77.000000
171.000000 114.000000 151.000000 142.000000 148.000000 90.000000 93.000000 80.000000
```

图 6: MatrixMul_omp_2 运行结果

调节线程数与矩阵规模进行输入，固定 Thread Block size 大小为 512。测试时输入的矩阵为方阵，则矩阵规模为 512 时表示对应的 m, n, k 为都为 512。其运算时间/运算 + 求和时间结果（单位：ms）如表3。据表可知，采用按列划分可减少 cuda 线程的计算量，因此每个线程的运算时间随着线程数的增加减少。但由于按列划分后的结果需要累加，在此过程中需要对 OpenMP 各线程进行互斥的访问，这极大地影响了汇总并行进程的计算结果的时间，导致效率低下，实现了 OpenMP 并行最终的负加速。

表 3: MatrixMul_omp_2 运算时间 (ms)/运算 + 求和时间 (ms) 与线程数、矩阵规模的关系

线程数 \ 矩阵规模	512	1024	2048	4096	8192
1	2.22/6.00	5.99/13.09	40.99/66.43	275.83/383.27	3891.62/4395.12
2	2.26/11.40	4.84/22.79	24.63/149.71	162.69/641.44	1170.94/3099.31
4	1.90/19.84	2.91/40.95	20.33/239.68	103.30/861.73	712.19/3960.62
8	2.3/132.48	2.52/301.21	16.40/421.96	77.47/1341.40	920.45/6059.24

2.3 任务 3

2.3.1 CUBLAS 实现与比较

实验中简单地采用了 CUBLAS 库中的 cublasSgemm 函数进行矩阵乘法运算，函数定义如图

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float           *C, int ldc)
```

图 7: cublasSgemm 函数

实现代码如下

```
1    float alpha = 1;
2    float beta = 0;
3    cublasHandle_t handle;
4    cublasCreate(&handle);
5    cublasSgemm(handle,
6                CUBLAS_OP_N,
7                CUBLAS_OP_N,
8                k, //矩阵 B 的列数
9                m, //矩阵 A 的行数
10               n, //矩阵 A 的列数
11               &alpha,
12               cuda_B,
13               k,
```

```
14                  cuda_A,
15                  n,
16                  &beta,
17                  cuda_C,
18                  k);
```

调节线程数与矩阵规模进行输入，测试时输入的矩阵为方阵，则矩阵规模为 512 时表示对应的 m, n, k 为都为 512。其运算时间结果（单位：ms）如表4。

表 4: MatrixMul_cubla 运算时间 (ms) 与矩阵规模的关系

矩阵规模	**512**	**1024**	**2048**	**4096**	**8192**
运算时间	230.62	227.62	248.64	312.25	581.00

调节矩阵规模进行输入，对实现的任务 1、任务 2 和任务 3 的程序测试运算时间，结果（单位：ms）如图8。由图可知，在小规模矩阵（<4096）的时候，实现的任务 1，任务 2 等程序表现较好，而在大规模 (>4096) 矩阵运算时，MatrixMul_cubla 实现的性能远高于任务 1，任务 2 的程序。

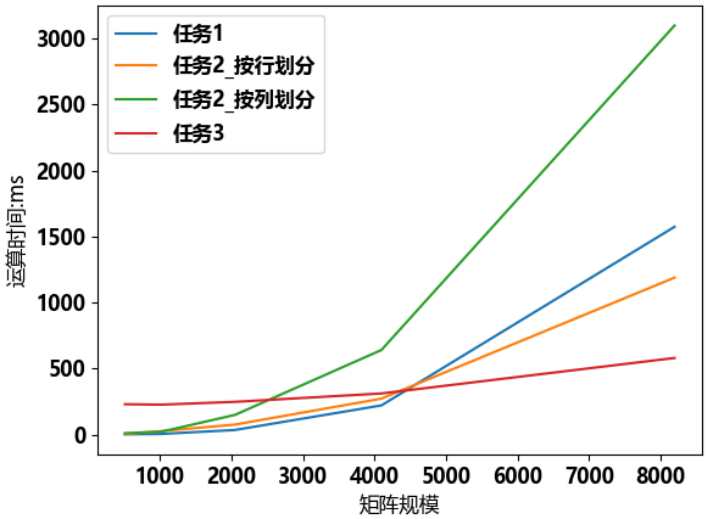

图 8: 实现的比较

2.3.2 改进方法

从比较中，程序还需进一步优化。在实验中，对于两个 $N*N$ 的矩阵 A 和 B 的相乘，并行方法是对于其输出矩阵 C 的每一个元素开一个线程，该线程载入 A 的一行和 B 的一列，然后对其做一次向量的内积。但问题是在 GPU 上访问显存的延时相当大。A 矩阵的一行因为在内存中是连续的还能够利用 GPU 的超大显存带宽一次载入多个元素平摊其载入时间以及缓存来降低延时。但对于大矩阵来说，B 的一个列中的元素的内存地址并不连续，意味着每次载入需要的那个列元素外大部分数据都是无用的，同时也降低了缓存效率。因此可以通过将 B 矩阵转置以提高数据的读取速度。

而为了提高矩阵读取效率，可将大矩阵划分成多个小矩阵进行运算，并在此过程中进行内存的重排，或是在算法层面上使用 Strassen 算法进行提高。在任务 2 中采用了按列划分的方式以提高效率，主要在于累加求和的互斥访问花费了大量时间，可以采用共享内存的方式提高求和速度。划分方式如图9，如果把小块看作一个元素，整个矩阵的规模相当于被缩小倍。对于每个小块的结果可以由一组线程负责，其中每个线程对应小块中的一个元素。这个线程组将 A 的行小片和 B 的列小片一一载入共享内存，在共享内存上对其做矩阵相乘，然后叠加在原有结果上。

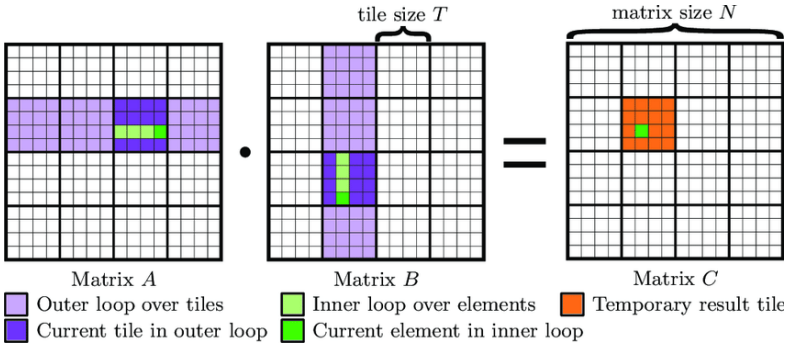

图 9: 矩阵划分

上述优化方法中，寄存器在线程间是不能共享的，如果每个线程要在自己的寄存器中保存所负责的划分的矩阵块的那部分，它也必须在寄存器中保存所用到的划分矩阵块中 A 的行和划分矩阵块 B 的列。结果是大量寄存器用于保存重复的数据。因此可以通过优化寄存器进一步提高效率。

3 实验感想

- 本次实验主要使用了 GPU 并行加速矩阵运算，GPU 的特点是集成大量的流处理器和计算单元, 具有并行处理的优势, 非常适合桌面和移动平台进行大规模超级计算。CUDA 是在 GPU 上实现的一种技术, 该技术可以将 GPU 由专用的处理器变为通用

的处理单元, 发挥 GPU 的强大的运算能力。

- 实验采用的 CUDA C 编写代码来并行矩阵运算，相较于之前实现的并行计算方式简单许多，在运算并行中 CUDA 的表现最佳。