中山大学数据科学与计算机院本科生实验报告

（2020学年秋季学期）

课程名称：**高性能计算程序设计**　　任课教师：**黄聃**　　批改人：

年级+班级	**18级计7**	**专业（方向）**	**计科**
学号	18340166	**姓名**	王若琪
Email	wangrq29@mail2.sysu.edu.cn	**完成日期**	2020.12.12

1. 实验目的

- **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。

  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 。

- **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。

- **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. 实验过程和核心代码

2.1 CUDA 通用矩阵乘法

- 定义并行矩阵乘法函数：

```
__global__ static void matMultCUDA(const float *a, const float *b, float *c,
int M, int N, int K, int BLOCK_SIZE)
```

  先得到目前的 thread 是第几个 thread，目前是在第几个 block 中，再据此计算出当前需要计算的 row 和 block：

```
    //表示目前的 thread 的编号
    const int tid = threadIdx.x;
    //表示目前在第几个 block 中
    const int bid = blockIdx.x;
    //计算出当前的 row 和 column
    const int idx = bid * BLOCK_SIZE + tid;
    const int row = idx / M;
    int column = idx % M;
```

再循环进行通用矩阵乘法：

```
    do
    {
        //矩阵乘法
        if (row < M && column < K)
        {
            float t = 0;
            for (int i = 0; i < N; i++)
            {
                t += a[row * N + i] * b[i * K + column];
            }
            c[row * K + column] = t;
        }
        column += M;
    } while (column < K);
```

- 计算 block 的数目：

```
const int blocks_num = (M * M + BLOCK_SIZE - 1) / BLOCK_SIZE;
```

- 初始化cuda，先判断并找出支持 cuda 的设备，并设置报错信息：

```
//CUDA 初始化
bool InitCUDA()
{
    //...此处省略，具体请见源代码 1.cu
    cudaSetDevice(i);
    return true;
}
```

- 定义矩阵并分配内存（代码展示略），再随机生成 a，b 两个矩阵：

```
    init_matrix(a, M, N);
    init_matrix(b, N, K);
```

- 使用函数 cudaMalloc()，在显存中给矩阵 a, b, c 分配空间：

```
    cudaMalloc((void **)&cuda_a, sizeof(float) * M * N);
    cudaMalloc((void **)&cuda_b, sizeof(float) * N * K);
    cudaMalloc((void **)&cuda_c, sizeof(float) * M * K);
```

- 使用函数 cudaMemcpy()，将 a，b 矩阵复制到显存中，其中 cudaMemcpyHostToDevice 表示从内存复制到显卡内存：

```
    //cudaMemcpy 将矩阵复制到显存中
    cudaMemcpy(cuda_a, a, sizeof(float) * M * N, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_b, b, sizeof(float) * N * K, cudaMemcpyHostToDevice);
```

- 在 cuda 中执行函数：

```
    // 在CUDA 中执行函数
    matMultCUDA<<<blocks_num, BLOCK_SIZE, 0>>>(cuda_a, cuda_b, cuda_c, M, N,
K, BLOCK_SIZE);
```

- 使用函数 cudaMemcpy()，将结果从显存中复制会内存：

```
    //cudaMemcpy 将结果从显存中复制回内存
    cudaMemcpy(c, cuda_c, sizeof(float) * M * K, cudaMemcpyDeviceToHost);
```

2.2 OpenMp + CUDA 通用矩阵乘法

在任务一的基础上，将矩阵 a 划分，用 OpenMp 并行，并行任务再调用 CUDA 矩阵乘法：

- 先设置线程：

```
omp_set_num_threads(OpenMp_THREAD_NUM);
```

- 用并行指令开始并行：

```
#pragma omp parallel private(cuda_a, cuda_b, cuda_c)
```

- 划分 a 矩阵，获得每个 OpenMp 线程分到的 a 矩阵部分的开始位置，同时划分结果 c 矩阵：

```
        //获取线程号
        int OpenMp_thread_id = omp_get_thread_num();
        //计算各个线程分到的 a 矩阵开始位置
        float *sub_a = a + OpenMp_thread_id * M * N / OpenMp_THREAD_NUM;
        //计算各个线程分到的 c 矩阵开始位置
        float *sub_c = c + OpenMp_thread_id * M * K / OpenMp_THREAD_NUM;
```

- 将划分过的 a 矩阵复制到显存中的对应位置，并复制整个 b 矩阵：

```
        //cudaMemcpy 将矩阵复制到显存中
        cudaMemcpy(cuda_a + OpenMp_thread_id * M * N / OpenMp_THREAD_NUM,
sub_a, sizeof(float) * M * N / OpenMp_THREAD_NUM, cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_b, b, sizeof(float) * N * K,
cudaMemcpyHostToDevice);
```

- 各个 OpenMp 线程分别执行矩阵乘法函数，负责各部分结果的计算：

```
        // 在CUDA 中执行函数
        matMultCUDA<<<blocks_num, BLOCK_SIZE, 0>>>(cuda_a + OpenMp_thread_id
* M * N / OpenMp_THREAD_NUM, cuda_b, cuda_c, M / OpenMp_THREAD_NUM, N, K,
BLOCK_SIZE);
```

- 最后将各个 OpenMp 线程结果从显存中复制回内存的对应位置：

```
        //cudaMemcpy 将结果从显存中复制回内存
        cudaMemcpy(sub_c, cuda_c, sizeof(float) * M * K / OpenMp_THREAD_NUM,
cudaMemcpyDeviceToHost);
```

2.3 CUBLAS计算矩阵相乘，并比较分析

cublas 的 cublasSgemm 函数完成 $C = \alpha op(A) op(B) + \beta C$ 的计算，当需要计算C=AB时，显然可以直接设置 $\alpha = 1, \beta = 0$ 。其中op操作对决定矩阵是否转置，即决定该矩阵是按照行优先还是列优先。当我们选择CUBLAS_OP_N时表示不转置，按列优先存储；当我们选择CUBLAS_OP_T时表示需要转置，按行优先存储。

cublasSgemm() 函数各个参数的含义解释如下：

- 参数handle：cublas的句柄，通过cublasCreate创建；
- 参数transA：矩阵B是否转置，转置为CUBLAS_OP_T，否则为CUBLAS_OP_N；
- 参数transB：矩阵A是否转置；
- 参数M：表示矩阵B的列数，若转置则为B的行数；
- 参数N：表示矩阵A的行数，若转置则为A的列数；
- 参数K：表示 B的行数或A的列数（A的列数=B的行数），即为被吃掉的维度；
- 参数alpha和beta：计算公式C = alpha*op(A)*op(B) + beta*C中的两个参数的引用；
- 参数lda：表示矩阵B的leading dimension，在列主序下，B转置表示B的列否则为B的行，值均为K，和参数M相反；
- 参数ldb：表示矩阵A的leading dimension，在列主序下，A转置表示A的行否则为A的列，值均为K，和参数N相反；
- 参数ldc：表示矩阵C的leading dimension，在列主序下始终为N，表示C的列。

本实验中，用 CUBLAS 计算矩阵乘法的核心代码如下：

```
    //调用API
    cublasHandle_t handle;
    cublasCreate(&handle);
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, K, M, N, &alpha, cuda_b, K,
cuda_a, N, &beta, cuda_c, K);
```

对三种矩阵乘法方式的性能的比较分析见实验结果部分，这里不做赘述。

可能的改进方法也请见实验结果部分。

---

3. 实验结果

3.1 CUDA 并行矩阵乘法的结果

测试不同的矩阵规模和 cuda Block size 大小，运行时间如下表：

矩阵规模\block size	32	64	128	256	512
512	0.091028 s	0.089948 s	0.087901 s	0.089482 s	0.095167 s
1024	0.092586 s	0.093660 s	0.092434 s	0.101220 s	0.103837 s
2048	0.139740 s	0.134835 s	0.134172 s	0.129117 s	0.129010 s
4096	0.621529 s	0.529688 s	0.542882 s	0.536705 s	0.532161 s
8192	3.971817 s	4.101717 s	3.977295 s	4.059693 s	3.893630 s

分析表格数据，可以看出，矩阵规模越大，运行时间越长，但是在表中所示的矩阵规模和 block size 的范围内，很难看出 block size 大小对程序性能的影响，因为虽然 block size 有所不同，但是运行时间相差无几。

3.2 OpenMp + CUDA 结果

固定 cuda thread 为 256，测试不同矩阵规模和不同 OpenMp 线程数的运行结果，如下表：

矩阵规模\线程数	1	2	4	8
512	0.093532 s	0.086713 s	0.088437 s	0.097837 s
1024	0.095528 s	0.102634 s	0.100325 s	0.113955 s
2048	0.257789 s	0.123426 s	0.136060 s	0.176943 s
4096	0.533967 s	0.431233 s	0.490729 s	0.460124 s
8192	4.053119 s	3.491611 s	2.975388 s	3.005052 s

从表格可以看出，矩阵规模越大，运行时间越长。当矩阵规模较小时（512 和 1024），OpenMp 线程数的大小对运行时间几乎没有影响，但当矩阵规模较大时（2048、4096 和 8192），OpenMp 线程数增大时，运行时间总体上会稍稍缩短，性能有些许提升。

3.3 CUBLAS 与其他两种比较

因为要将三种方法作比较，所以对第一种 CUDA 并行方法，固定 CUDA block size 为256，对第二种 CUDA+OpenMp 的方法，固定 CUDA block size 为256，同时 OpenMp 线程数为 4。然后再将三种并行方法应用于不同规模的矩阵乘法计算，统计运行时间如下表：

矩阵规模\并行方式	CUDA	CUDA+OpenMp	CUBLAS
512	0.089482 s	0.088437 s	0.365077 s
1024	0.101220 s	0.100325 s	0.497235 s
2048	0.129117 s	0.136060 s	0.370552 s
4096	0.536705 s	0.490729 s	0.436916 s
8192	4.059693 s	2.975388 s	0.721208 s

从上表可以看出，当矩阵规模较小时（512、1024 和 2048），第一、二种方法的性能相差无几，第三种方法的性能远远差于前两种。当矩阵规模较大时（4096 和 8192），第三种方法运行速度最快，性能最好，第二种方法次之，第一种方法最慢，尤其是在矩阵规模为 8192 时，CUBLAS 方法的性能远胜于另外两种方法。

3.4 可能的改进方法

由于当矩阵规模较大时，自己编写的前两种并行矩阵乘法性能不如 CUBLAS API，所以提出以下几点可能的改进方法：

- 最大化内存吞吐量：
  - 尽可能减少 host 和 device 之间的内存传输，如可以将多次传输合并为一次等；
  - 尽量使用page-locked内存；
  - 尽可能考虑局部性原理，考虑对齐问题，降低访存所消耗的时间。
- 优化指令流：
  - 尽量减少条件分支；
  - 用更高效的单精度浮点乘除法；
  - 避免多于的同步；
  - 使用 CUDA 算数指令中的快速指令。
- 资源均衡：
  - 可通过调整每线程处理的数据量，调整 block 大小等，使资源均等分配。

---

4. 实验感想

- 本次实验是我第一次接触 C 语言的 CUDA 编程，所以在这次实验中，我遇到了许多问题，也一一解决了，在解决问题的过程中学到了许多新知识。我遇到的问题和解决方法如下：
- 任务一的难点应该就是在 __global__ 核函数中找到本线程要计算的行号与列号了。想要完成这一步骤，必须要对 CUDA 的结构有清晰的了解，比如 Block 是由多个线程组成的，各个 Block 之间是并行执行的，block之间无法通信。开始对列号的计算出错，导致结果矩阵中有很多 0，后来找到错误，多加了一个关于列号的循环，就解决了问题。
- 任务二就是对 OpenMp 的简单应用，所以在这个任务中几乎没有遇到困难，很顺利，像是对 OpenMp 编程的复习。
- 任务三的难点就是对 CUBLAS API 的使用方法，这个任务参考了网页 https://www.cnblogs.com/tiandsp/p/9463396.html。