多核程序设计与实践

作业一：矩阵加法

中山大学计算机学院　计算机科学与技术

19335174　施天予

目录

一、题目要求

1. 给定两个大小相等的矩阵 A, B，计算矩阵 C，其每一个元素均为 A，B 中相应元素之和:

$$C[i,j] = A[i,j] + B[i,j]$$

   注: 矩阵大小不小于 1000×1000

2. 介绍程序整体逻辑, 包含的函数, 每个函数完成的内容 (10 分)
3. 讨论矩阵大小及线程组织对性能的影响, 可考虑但不限于以下因素 (60 分):
   - 一维 VS 二维
   - 线程块大小对性能的影响
   - 每个线程计算的元素数量对性能的影响
   - 以上配置在处理不同大小的矩阵时, 性能可能的差异
4. 使用 OpenMP 实现并与 CUDA 版本进行对比, 可根据矩阵大小讨论 (30 分)

二、程序逻辑

由于所给的 8×8 和 2048×2048 矩阵不太容易分析 CUDA 程序的性能，我就自己写了一个随机初始化任意大小矩阵的函数。随机初始化两个矩阵，并为加和矩阵分配 host 端空间。

```
void Generate(float **a, float **b, float **c, int m, int n) {
    *a = new float[m*n], *b = new float [m*n], *c = new float [m*n];
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j) {
            (*a)[i*n+j] = 10.0 * rand()/(RAND_MAX+1.0);
            (*b)[i*n+j] = 10.0 * rand()/(RAND_MAX+1.0);
        }
}
```

Evaluate 函数用于检验矩阵加法结果的正确性，也可以看作是矩阵加法的 CPU 串行版本。

```
void Evaluate(float *a, float *b, float *c, int m, int n) {
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
            if ((fabs(a[i*n+j] + b[i*n+j]- c[i*n+j]) / c[i*n+j]) > 1e-4) {
                printf("Computation Error In %d Row %d Col!\n", i, j);
                return;
            }
    printf("Computation Correct!\n");
}
```

1. CUDA 矩阵加法

(i) 显存分配与数据拷贝

在 device 端分配显存，并将 2 个矩阵从 host 端复制到 device 端

```
// 显存分配
float *da, *db, *dc;
CHECK(cudaMalloc((void**)&da, size*sizeof(float)));
CHECK(cudaMalloc((void**)&db, size*sizeof(float)));
CHECK(cudaMalloc((void**)&dc, size*sizeof(float)));
// 数据拷贝
CHECK(cudaMemcpy((void*)da, (void*)input1, size*sizeof(float),
    ↪ cudaMemcpyHostToDevice));
CHECK(cudaMemcpy((void*)db, (void*)input2, size*sizeof(float),
    ↪ cudaMemcpyHostToDevice));
```

(ii) 参数设置

设置 blockSize 和 gridSize。b1 是 block 一维大小，b2 是 block 二维大小，grid_dim 是网格维度，1 或者 2。sz 用于指示每个线程计算的元素个数，一维则每个线程计算 sz^2 个元素，二维则每个线程计算 $sz \times sz$ 的矩阵。我们将 blockSize 和 gridSize 用 dim3 类型保存，用于核函数调用。注意 gridSize 可以由 blockSize 和 sz 确定。

由于我在给定相关文件前已经实现完成，所以在 cuda_add.cu 中用命令行获取这些参数，而在给定的 main.cu 中为不改变原本的代码就在程序中设置这些参数。

```
int b1 = strtol(argv[3], NULL, 10);   // blockSize一维
int b2 = strtol(argv[4], NULL, 10);   // blockSize二维（1代表一维块）
int grid_dim = strtol(argv[5], NULL, 10); // grid维度
int sz = strtol(argv[6], NULL, 10);
if (grid_dim == 1)
    sz = sz * sz;
dim3 blockSize(b1, b2);
dim3 gridSize;
// block一维，grid一维
if (b2 == 1 && grid_dim == 1)
    gridSize = dim3(divup(divup(size, sz), blockSize.x));
// block一维，grid二维 或 block二维，grid二维
else if (grid_dim == 2)
    gridSize = dim3(divup(divup(width, sz), blockSize.x), divup(divup(height,
        ↪ sz), blockSize.y));
```

(iii) 核函数设计

一维块核函数如下：

```
__global__ void MatrixMul_1d(float *a, float *b, float *c, int m, int n, int sz) {
    int id = (threadIdx.x + blockIdx.x * blockDim.x) * sz;
    for (int i = id; i < id + sz; ++i)
        if (i < m * n)
            c[i] = a[i] + b[i];
}
```

二维块核函数如下：

```
__global__ void MatrixMul_2d(float *a, float *b, float *c, int m, int n, int sz) {
    int idx = (threadIdx.x + blockIdx.x * blockDim.x) * sz;
    int idy = (threadIdx.y + blockIdx.y * blockDim.y) * sz;
    for (int y = idy; y < idy + sz; ++y)
        for (int x = idx; x < idx + sz; ++x)
            if (y < m && x < n) {
                int id = y * n + x;
                c[id] = a[id] + b[id];
            }
}
```

这里主要需要注意下标越界的问题，其它用 for 循环即可完成。一维线程块每个线程计算 sz 个元素，二位线程块每个线程计算 sz^2 个元素。

(iv) 核函数调用

指定 gridSize 和 blockSize 即可调用核函数。我们用 getTime 函数来计算矩阵加法的运行时间，而不用计算显存分配及拷贝的开销。

```
    long st, et;
    st = getTime();
    // 调用核函数
    if (grid_dim == 1)
        MatrixMul_1d <<<gridSize, blockSize>>> (da, db, dc, height, width, sz);
    else
        MatrixMul_2d <<<gridSize, blockSize>>> (da, db, dc, height, width, sz);
    et = getTime();
    printf("Time cost (CUDA): %.3f ms\n", (et - st) / 1e6);
```

(v) 释放显存

最后将结果返回 host 端并释放显存。

```
    // 拷贝回host并释放显存
    CHECK(cudaMemcpy((void*)result, (void*)dc, size*sizeof(float),
        ↪ cudaMemcpyDeviceToHost));
    CHECK(cudaFree(da));
    CHECK(cudaFree(db));
```

```
    CHECK(cudaFree(dc));
```

2. OpenMP 矩阵加法

OpenMP 的实现比较简单，指定线程数，用 for 循环即可。

```
    int thread_num = strtol(argv[3], NULL, 10);
    double start = omp_get_wtime();
#   pragma omp parallel for num_threads(thread_num)
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j) {
                int id = i * n + j;
                c[id] = a[id] + b[id];
            }
    double end = omp_get_wtime( );
    printf("Time cost(OpenMP): %.3f ms\n", (end - start) * 1000);
```

三、实验过程

1. 实验环境

CPU	Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz
GPU	NVIDIA GeForce RTX 3090
CUDA	Cuda compilation tools, release 9.1, V9.1.85

表 1: 软硬件环境

2. 编译运行

CUDA 矩阵加法，例如矩阵大小 10000×10000，块大小 32×32, grid 维数 2, 单线程计算元素个数为 1×1:

```
$ nvcc -w cuda_add.cu -o cuda_add
$ ./cuda_add 10000 10000 32 32 2 1
```

所给定的文件 main.cu 不用指定参数，只需对应的文件路径:

```
$ nvcc -w main.cu -o main
$ ./main ../data/input2.bin ../data/output2.bin
```

OpenMP 矩阵加法，例如矩阵大小 10000×10000，线程数 32:

```
$ g++ -fopenmp omp_add.cpp -o omp_add
$ ./omp_add 10000 10000 32
```

3. 实验结果

设置不同参数，每组参数测试 5 次取均值，结果如下（单位 ms）：

blockSize / sz	One Dim Block				Two Dim Block			
	16×1	64×1	256×1	1024×1	4×4	8×8	16×16	32×32
1×1	0.184	0.067	0.069	0.067	0.183	0.068	0.065	0.073
2×2	0.067	0.069	0.066	0.066	0.087	0.063	0.072	0.062
4×4	0.104	0.437	0.454	0.468	0.068	0.074	0.075	0.072
8×8	0.172	0.931	1.068	0.911	0.102	0.105	0.104	0.131

表 2: 矩阵大小 2000×2000

blockSize / sz	One Dim Block				Two Dim Block			
	16×1	64×1	256×1	1024×1	4×4	8×8	16×16	32×32
1×1	1.082	0.359	0.361	0.366	1.080	0.380	0.363	0.387
2×2	0.358	0.364	0.371	0.378	0.513	0.363	0.382	0.359
4×4	0.622	2.617	3.039	2.991	0.376	0.411	0.466	0.375
8×8	1.032	6.009	6.156	6.016	0.591	1.456	1.405	0.925

表 3: 矩阵大小 5000×5000

blockSize / sz	One Dim Block				Two Dim Block			
	16×1	64×1	256×1	1024×1	4×4	8×8	16×16	32×32
1×1	4.035	1.402	1.452	1.460	4.307	1.449	1.406	1.509
2×2	1.417	1.423	1.442	1.432	2.004	1.429	1.510	1.413
4×4	2.439	10.350	12.250	11.916	1.432	1.527	1.734	1.477
8×8	4.102	24.373	24.877	24.398	2.186	3.523	4.766	3.077

表 4: 矩阵大小 10000×10000

四、性能分析

1. 一维 block vs 二维 block

如上表所示，当 block 中的线程个数相同时，观察表格左右两边可以发现：

当 $sz = 1\times1$ 时，每个线程只计算一个元素，1 维 block 和 2 维 block 性能接近。这是因为此时二维矩阵在内存都是连续存储的，所以 1 维 block 和 2 维 block 每个线程计算的元素位置也是完全一样的。

当 sz 增大时，对于一个 10000×10000 的矩阵和大小为 1024 的 block，一维用时 $24.398ms$，二维用时 $3.077ms$，明显加快。这可能是因为一维 block 和二维 block 不同的组织方式带来的

对访存方式的影响。

2. 线程块大小

不管是 1 维还是 2 维 block，随着线程块大小的增大，性能明显提升。但是并不是说单个 block 线程越多越好，比如对于 5000×5000 的矩阵，其性能峰值在大小为 64 的 block，1024 的 block 性能反而降低。另外，对于不同的 sz 大小或矩阵大小，性能的峰值会有所不同。

3. 单线程计算元素数量

通过三张表格我们可以发现，当线程块较小时，每个线程计算的元素数量越少，耗时越短。另外，不管是一维 block 还是二维 block，其性能峰值都在 2×2 时，而每个线程过少或过多的计算元素数量都反而会降低性能。这可能是由于加法任务较为简单，整体计算量不足的原因。

4. 矩阵大小

当 $sz = 1 \times 1$ 时，对于一个 5000×5000 的矩阵，其性能峰值在大小为 64 的 block，而矩阵增大后, 峰值性能在线程块大小为 1024 时。

当 sz 增大时，对于不同大小的矩阵，性能峰值几乎都在块大小为 1024 时。矩阵越大，加速比越大，更能充分发挥 CUDA 并行计算的优势。

5. 总结

通过上述分析, 我们可以看到线程块大小、矩阵大小、每个线程计算元素数量三个变量是，共同决定着当前 CUDA 程序的性能。首先我们知道线程块大小 * 每个线程计算元素数量决定着每个线程块可以计算的元素数量, 那么程序会根据这个值和矩阵的大小动态地计算出需要启动多少个 block, 即 gridSize。当每个线程块可以计算的元素数量增加时或者矩阵大小减小时, 启动的线程块的数量会减小，而当线程块的数量小于 GPU 硬件的 SM (Streaming Multiprocesser) 的数量时，此时流多处理器的占用率低，性能就会下降。

因此，矩阵大小、线程块大小、每个线程计算元素数量三个变量共同决定程序性能。另外线程块的数量、1 维或 2 维 block、寄存器占用情况等多种情况的影响也共同影响着 CUDA 程序的性能。

五、CUDA vs OpenMP

设置不同的线程数和矩阵大小，OpenMP 计算结果如下（单位 ms）:

$N \times N$ \ Thread Num	8	16	32	64
2000×2000	3.015	2.982	3.813	4.444
5000×5000	18.433	18.303	18.935	19.726
10000×10000	70.711	71.632	73.936	73.978

表 5: OpenMP 运行时间

我们将 CUDA 性能最高的配置和 OpenMP 性能最高的进行对比：

并行框架 \ $N \times N$	2000×2000	5000×5000	10000×10000
CUDA	0.062	0.358	1.402
OpenMP	2.982	18.303	70.711

表 6: CUDA vs OpenMP

可以发现，使用 RTX 3090 的 CUDA 进行矩阵加法比使用 OpenMP 性能提高了 50 倍左右。思考其中的原因，CPU 由专为顺序串行处理而优化的几个核心组成，而 GPU 则由数以千计的更小、更高效的核心组成，有着高吞吐量、高计算能力、高并发线程数等特点。GPU 可以高效地处理并行任务，也更加适合矩阵加法这种子任务数量庞大、无数据依赖等问题的任务，故 CUDA 程序能够取得更好的性能。

另外，我们对于 CUDA 矩阵加法的性能时，只考虑了其核函数调用的计算时间，而未考虑其创建显存和数据拷贝的额外开销。对于较大的矩阵，创建显存和拷贝数据的时间开销会远大于核函数计算的时间，此时总时间性能上 CUDA 反而会慢于 OpenMP。所以对于不同的任务需求，我们需要具体问题具体分析。