作业 1

18340146 计算机科学与技术 宋渝杰

题目要求：

矩阵相加：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：

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

程序整体逻辑：

由于本次实现的是“矩阵相加”，还是比较简单的，程序的整体流程为：

- 初始化两个矩阵
- 实现矩阵相加（同时计时）
- 输出运算时间（有需要的话同时输出矩阵）

因此设计对应的函数和主函数：

矩阵设计：这里用一个结构体 `Matrix` 表示一个矩阵，具体的属性如下：

```
struct Matrix{
    int x, y;               // 矩阵的长和宽
    int** v;                // 二维数组表示矩阵
};
```

矩阵初始化：这里对矩阵采取随机初始化的方式：

```
void init(Matrix* m) {
    for (int i = 0; i < m->x; i++)
        for (int j = 0; j < m->y; j++)
            m->v[i][j] = rand() % 10;
}
```

矩阵相加（cuda）：这里采用一个线程计算一步 $C[i,j] = A[i,j] + B[i,j]$

```
__global__ void add(Matrix* a, Matrix* b, Matrix* c) {
    int x = blockIdx.y * blockDim.y + threadIdx.y, y = blockIdx.x * blockDim.x +
threadIdx.x; // 定位线程ID，处理对应位置的C[i,j]
    if (x < c->x && y < c->y) // 防止矩阵越界
        c->v[x][y] = a->v[x][y] + b->v[x][y];
}
```

输出矩阵：这里同样采用朴素的二维 for 循环进行矩阵遍历并输出：

```
void print(Matrix* m) {
    for (int i = 0; i < m->x; i++) {
        for (int j = 0; j < m->y; j++) printf("%d ", m->v[i][j]);
        printf("\n");
    }
}
```

主函数：主函数的逻辑为：接收矩阵大小输入 -> 申请内存 -> 初始化矩阵 -> 矩阵相加 -> 输出时间（和矩阵）：

```
int main(int argc, char* argv[]) {
    int m = strtol(argv[1], NULL, 10), n = strtol(argv[2], NULL, 10); // 接收矩阵大小输入
    timeval t1, t2;
    Matrix *a, *b, *c;
    cudaMallocManaged((void**)&a, sizeof(Matrix)); // 申请内存
    cudaMallocManaged((void**)&b, sizeof(Matrix));
    cudaMallocManaged((void**)&c, sizeof(Matrix));
    a->x = m; a->y = n; b->x = m; b->y = n; c->x = m; c->y = n;
    cudaMallocManaged((void**)&a->v, a->x * sizeof(int*));
    cudaMallocManaged((void**)&b->v, b->x * sizeof(int*));
    cudaMallocManaged((void**)&c->v, c->x * sizeof(int*));
    for (int i = 0; i < m; i++) {
        cudaMallocManaged((void**)&a->v[i], a->y * sizeof(int));
        cudaMallocManaged((void**)&b->v[i], b->y * sizeof(int));
        cudaMallocManaged((void**)&c->v[i], c->y * sizeof(int));
    }
    init(a); init(b); // 初始化矩阵
    dim3 block(sizex, sizey), grid(m / sizex, n / sizey);

    gettimeofday(&t1, NULL); // 计时开始
    add <<< grid, block >>> (a, b, c); // 矩阵相加
    cudaDeviceSynchronize();
    gettimeofday(&t2, NULL); // 计时结束

    printf("Time: %.4fs\n", (t2.tv_sec-t1.tv_sec+(t2.tv_usec-t1.tv_usec)/1.0e6)); // 输出时间
    return 0;
}
```

讨论性能：

基准性能：这里的基准程序为【二维矩阵、线程块大小为 8*8、每个线程处理一个元素求和、矩阵大小为 4096*4096】

命令行输入 `nvcc -w cuda1.cu -o test1` 编译并生成 test1 可执行程序，再输入 `./test1 4096 4096` 执行程序，结果如下：

```
jovyan@jupyter-songyj9-40mail2-2esysu-2eedu-2ecn:~/multi$ nvcc -w cuda1.cu -o test1
jovyan@jupyter-songyj9-40mail2-2esysu-2eedu-2ecn:~/multi$ ./test1 4096 4096
Time: 0.0553s
```

一维 or 二维：在基准程序的基础上，采用一维数组代替二维矩阵，具体的代替方法为：

$$V_{new}[i * m_y + j] = V_{old}[i][j]$$

其中 m_y 代表矩阵的宽度。测试结果如下：

```
jovyan@jupyter-songyj9-40mail2-2esysu-2eedu-2ecn:~/multi$ nvcc -w cuda2.cu -o test2
jovyan@jupyter-songyj9-40mail2-2esysu-2eedu-2ecn:~/multi$ ./test2 4096 4096
Time: 0.0481s
```

可以看出一维矩阵运算略微快于二维矩阵。

线程块大小：在基准程序的基础上，修改线程块大小为 32*32，测试结果如下：

```
jovyan@jupyter-songyj9-40mail2-2esysu-2eedu-2ecn:~/multi$ nvcc -w cuda3.cu -o test3
jovyan@jupyter-songyj9-40mail2-2esysu-2eedu-2ecn:~/multi$ ./test3 4096 4096
Time: 0.0432s
```

可以看出线程块变大时，矩阵相加的运算也稍微变快。

每个线程计算的元素数量：在基准程序的基础上，修改 cuda 核函数，使得其一个函数处理矩阵一行的数据：

```
__global__ void add(Matrix* a, Matrix* b, Matrix* c) {
    int x = blockIdx.y * blockDim.y + threadIdx.y, y = blockIdx.x * blockDim.x +
threadIdx.x;
    if (x < c->x && y == 0)
        for (int i = 0; i < c->y; i++)
            c->v[x][i] = a->v[x][i] + b->v[x][i];
}
```

测试结果如下：

```
jovyan@jupyter-songyj9-40mail2-2esysu-2eedu-2ecn:~/multi$ nvcc -w cuda4.cu -o test4
jovyan@jupyter-songyj9-40mail2-2esysu-2eedu-2ecn:~/multi$ ./test4 4096 4096
Time: 0.0438s
```

这里有些出乎个人预料：线程数变少了，每个线程数的工作量变大了，运算的时间反而变短了，多次测试之后结果也是如此，也许是基准程序 cuda 创建太多线程开销更大。

矩阵大小：修改矩阵大小为 8*4096，再对比修改“每个线程计算的元素数量”前后的结果，即测试命令行为 `./test1 8 4096` 和 `./test4 8 4096`，测试结果如下：

```
jovyan@jupyter-songyj9-40mail2-2esysu-2eedu-2ecn:~/multi$ ./test1 8 4096
Time: 0.0003s
jovyan@jupyter-songyj9-40mail2-2esysu-2eedu-2ecn:~/multi$ ./test4 8 4096
Time: 0.0010s
```

这次线程数变少，每个线程数的工作量变大之后，运算时间变长，符合一般“并行程度降低，运算时间变长”的规律。

矩阵相加（openmp）：对于矩阵加法可以采用朴素的二维 for 循环进行矩阵遍历并相加，同时采用 `#pragma omp parallel for num_threads(4)` 对 for 循环进行并行化处理

```
void add(Matrix* a, Matrix* b, Matrix* c) {
    #pragma omp parallel for num_threads(4) // 4个线程
    for (int i = 0; i < a->x; i++)
        for (int j = 0; j < a->y; j++)
            c->v[i][j] = a->v[i][j] + b->v[i][j];
}
```

openmp 使用 4 线程，对比测试结果如下：（由于服务器不支持 CPU 多线程，因此在本地测试）

```
C:\Users\Song\Desktop\S & W\大三下\多核\lab1\src>g++ openmp.cpp -o test5 -w -fopenmp

C:\Users\Song\Desktop\S & W\大三下\多核\lab1\src>test5.exe 4096 4096
Time: 0.022318s

C:\Users\Song\Desktop\S & W\大三下\多核\lab1\src>test5.exe 8 4096
Time: 0.001004s
```

在本次实验中，opemmp 在矩阵规模为 4096*4096 时比 cuda 更快，而在 8*4096 时比 cuda 基准程序要慢