19335030-陈至雪-作业1

作业要求

> 1. (coding) 利用CUDA实现矩阵相加
> – 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
> • $C[i,j] = A[i,j] + B[i,j]$
> note :
> – 矩阵的读取、写入代码已提供，只需实现矩阵相加部分代码
> – 提供的测试数据中，input1.bin为*88*维的矩阵，*input2.bin*为*2048*2048维的矩阵
> 可通过如下的脚本进行调用（默认调用input1.bin）：
> ./main input1.bin output1.bin
> ./main input2.bin output2.bin
> 最终应生成output1.bin，output2.bin 两个文件。
> – 最终提交的代码中要求对于长宽都大于0，总的大小不超过显存大小的矩阵都能正常运行。
> 2. (writing) 回答以下问题
> – 2.1 介绍程序整体逻辑，包含的函数，每个函数完成的内容（10分）
> – 2.2 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
> – 2.3 使用OpenMP实现并与CUDA版本进行对比（30分）
> • 可根据矩阵大小讨论
>
> submission:
> 作业提交内容：需提交一个.zip文件，需包括main.cu，output1.bin， output2.bin 以及实验报告pdf。
> zip文件命名格式：姓名学号homework1; 如需提交不同版本，则命名格式：姓名学号homework1_v2等。
> 作业提交方式: https://www.scholat.com/course/muticore2022
> 作业提交截止时间： 5月15日晚11时59分

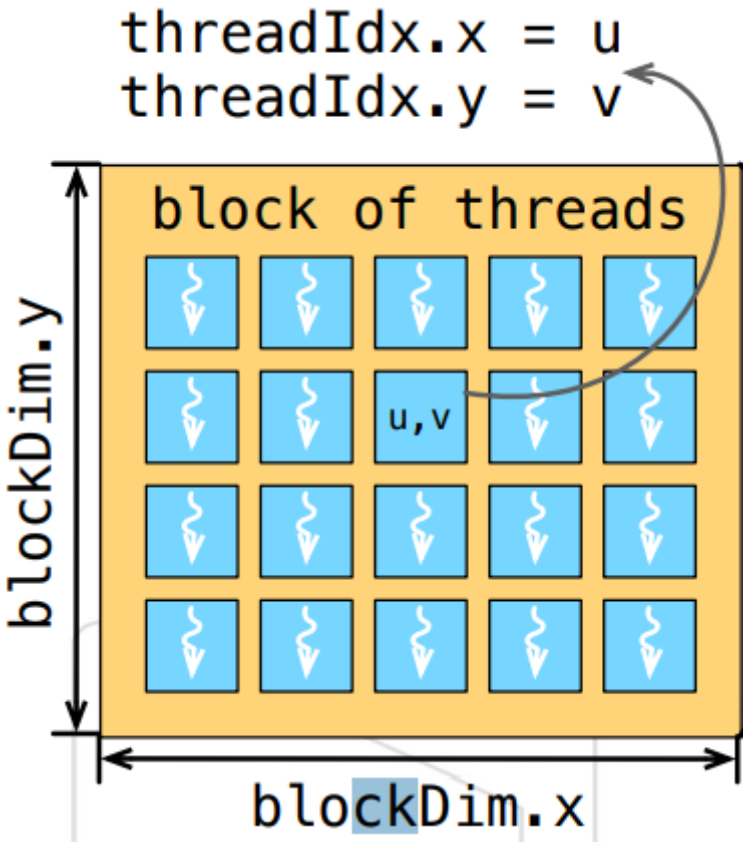
程序实现

CUDA 一维

矩阵加法函数

```
__global__ void Matrix_add(float* A, float* B, float* C, int n){
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if( tid <  n)
        C[tid] = A[tid] + B[tid];
}
```

`__global__`限制函数从主机端调用，在设备端执行。由于此处的矩阵存储是一维的，先用公式`tid = blockDim.x * blockIdx.x + threadIdx.x`找出线程id，然后进行加法操作`C[tid] = A[tid] + B[tid]`.



数据的读取

```
    if (argc == 3) {
        inputPath = argv[1];
        outputPath = argv[2];
    }

    // Open the input file
    FILE *stream = fopen(inputPath, "rb");
    if (stream == NULL) {
        printf("failed to open the data file %s\n", inputPath);
        return -1;
    }

    // Open a stream to write out results in text
    FILE *outStream = fopen(outputPath, "wb");
    if (outStream == NULL) {
        printf("failed to open the output file %s\n", outputPath);
        return -1;
    }

// Read in and process the input matrix one-by-one
    int width, height, size;
    float *input1, *input2, *result;
    clock_t start,end;
    loadMatrix(stream, &width, &height, &input1);
    loadMatrix(stream, &width, &height, &input2);
    size = width * height;
    result = (float*)malloc(sizeof(float) * size);
```

从命令行中读取文件名信息，然后用`loadMatrix`函数加载矩阵到input1和input2中。`size = width * height`计算矩阵的大小，`result = (float*)malloc(sizeof(float) * size);`为输出矩阵申请在主机端上的内存空间。

设备端内存申请

```
    float *input1_d, *input2_d, *result_d;
    int n_bytes = sizeof(float)*size;
    cudaMalloc((void**)&input1_d, sizeof(float)*size);
    cudaMalloc((void**)&input2_d, sizeof(float)*size);
    cudaMalloc((void**)&result_d, sizeof(float)*size);
    start=clock();//开始计时
    cudaMemcpy(input1_d, input1, n_bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(input2_d, input2, n_bytes, cudaMemcpyHostToDevice);
```

在设备端用`cudaMalloc`函数申请内存空间，用`cudaMencpy`函数将主机端的数据拷贝到申请好内存空间的设备端变量上。在拷贝内存之前，进行计时。

执行配置

```
    int block_size;
    printf("Please enter block_size!\n");
    scanf("%d",&block_size);
    Matrix_add<<<divup(size, block_size), block_size>>>(input1_d, input2_d, result_d, size);
```

读取`block_size`，然后调用`Matrix_add`函数，<<<divup(size, block_size), block_size>>>指明grid大小为`divup(size, block_size)`, block大小为block_size。

```
    cudaDeviceSynchronize();
    cudaMemcpy( result, result_d, n_bytes, cudaMemcpyDeviceToHost);
    end = clock();//计时结束
```

调用完`Matrix_add`函数后，调用`cudaDeviceSynchronize`函数进行同步，待所有线程执行完成后，将结果拷贝回主机的结果矩阵中，再次记录时间。

释放GPU内存

```
    cudaFree(result_d);
    cudaFree(input1_d);
    cudaFree(input2_d);
```

将结果保存到输出文件中

```
    saveMatrix(outStream, &width, &height, &result);
```

输出运行时间

```
    float time1=(float)(end-start)/CLOCKS_PER_SEC;
    printf("Time of GPU: %f\n", time1);
```

释放主机内存，关闭文件

```
    free(input1);
    free(input2);
    input1 = input2 = NULL;
    free(result);
    result = NULL;
    // Close the output stream
    fclose(outStream);
```

CUDA 二维

二维的实现和一维的实现的差别在于执行的配置和矩阵加法函数的实现。故下面只对这两部分进行说明。

矩阵加法函数

```
//2维
__global__ void Matrix_add(float* A, float* B, float* C, int n, int m){
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;
    if (y < n && x < m ){
        C[y * m + x] = A[y * m + x] + B[y * m + x];
    }
}
```

用公式`x=blockDim.x * blockIdx.x + threadIdx.x`和`y = blockDim.y * blockIdx.y + threadIdx.y`定位线程，然后进行加法操作。

执行配置

```
    int w, h;
    printf("Please enter block_w and block_h!\n");
    scanf("%d%d",&w, &h);
    dim3 block(w, h, 1);
    dim3 grid(divup(width, w), divup(height, h), 1);

    start=clock();//开始计时
    cudaMemcpy(input1_d, input1, n_bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(input2_d, input2, n_bytes, cudaMemcpyHostToDevice);
    Matrix_add<<<grid, block>>>(input1_d, input2_d, result, height, width);

    cudaDeviceSynchronize();
    cudaMemcpy( result, result_d, n_bytes, cudaMemcpyDeviceToHost);
    end = clock();//计时结束
```

输入block的长和宽，即 w 和 h，用语句`dim3 block(w, h, 1);`创建三维的block。给定 block 的长和宽之后，用语句`divup(width,w)`和`divup(height,h)`计算grid的长和宽，然后用语句`dim3 grid(divup(width, w), divup(height, h), 1);`创建三维的grid。再用语句`Matrix_add<<<grid, block>>>(input1_d, input2_d, result, height, width);`传递相关参数，调用矩阵加法函数进行矩阵加法运算。

运算完后进行同步，最后将结果拷贝回主机内存变量。

最后记录时间，并对运行时间进行输出。

OpenMP

读取数据的方式和上面的一样，区别在于矩阵加法的运算。

```
    clock_t t1 = clock(); //开始计时
    int thread_num = 1;
    cout << "Please enter the number of threads" << endl;
    cin >> thread_num;
    #pragma omp parallel for num_threads(thread_num)
    #pragma omp parallel for num_threads(32)
    for( int row = 0; row < height; row ++ ){
        for( int col = 0; col < width; col++ ){
            result[ row * width + col ] = input1[ row * width + col ] +
input2[ row * width + col ];
        }
    }
    clock_t t2 = clock(); //结束计时
    cout << "OpenMP time: " << (float)(t2-t1)/CLOCKS_PER_SEC << endl;
    saveMatrix(outStream, &width, &height, &result);
```

输入想要的thread_num设置，使用`#pragma omp parallel for num_threads(thread_num)`进行并行，在开始和结束时用clock()计时，然后输出运行时间。

结果分析

cuda的编译命令：

`nvcc -o main.out main.cu`

cuda文件的执行命令：

`./main.out input1.bin input2.out`

openMP的编译命令：

`g++ -fopenmp omp.cpp -o omp`

openMP的执行命令：

`./omp`

线程组织：

线程组织	矩阵规模 8 * 8 (s)	矩阵规模 2048 * 2048 (s)
一维	0.000060	0.01702
二维	0.01623	0.03884

取block_size分别为 8 * 8， 16 * 16， 32 * 32，运行时间取平均。

可以看到无论矩阵规模大还是小，一维的线程组织形式都比二维的线程组织形式更快。导致的原因可能是，二维线程组织形式导致数据的访问不连续。

线程块大小：

线程块大小	矩阵规模 8 * 8 (s)	矩阵规模 2048 * 2048 (s)
1 * 2	0.000056	0.023070
2 * 2	0.000058	0.019947
4 * 4	0.000055	0.017770
8 * 8	0.000060	0.017063
16 * 16	0.000060	0.016913
32 * 32	0.000062	0.017092

由于GPU实际设计中，线程块大小最多为1024，因此这里只将最大值设置为了1024。另外，这里用的是一维的线程组织方式。

可以看到在矩阵规模过小时，线程块越大，运行时间反而越长，说明对少量数据，大的线程块是不必要的。

当矩阵规模较大时，可以看到随着线程块规模的增大，运行时间随之降低，但是当线程规模过大时，运行时间不再增大。

每个线程计算的元素数量：

每个线程计算的元素数量	矩阵规模 8 * 8 (s)	矩阵规模 2048 * 2048 (s)
4	0.000060	0.0157
8	0.000063	0.0188
16	0.000065	0.0829
32	0.000058	0.0128
64	0.000075	0.0148
128		0.0195

从表中可以看到，矩阵规模为 8 * 8的计算量太小了，随着每个线程计算的元素个数增加，运行时间不一定增加，这可能和主机外部因素影响。

从矩阵规模为 2048 * 2048 的运行时间可以看到也存在着这个现象，但当线程计算的元素数量增大到 128 时，运行时间很明显比其他设置慢。

考虑到数据存放的局部性，让一个线程处理一个数据的方式可能并不是最好的。如果对数据的读取操作的消耗大于对数据计算的消耗，那可以让线程负责多一些计算，避免频繁的数据读取。

处理不同矩阵规模的性能差异：

从上面的表格中可以看到，对于小规模的矩阵，简易的配置（如，一维配置，一个线程处理多个数据计算）往往能取得不错的性能，因为它们对计算的需求不高。

对于大规模矩阵，计算时间明显会比小规模矩阵长。并且让一个线程负责的数据计算越少，大规模矩阵计算的时间也会越短。可见，当计算规模大到一定程度时，配置的好坏在一定程度上影响计算性能。此时配置需要根据具体的计算量以及数据的存储方式进行设定。

OpenMP与CUDA版本比较：

方式	矩阵规模 8 * 8 (s)	矩阵规模 2048 * 2048 (s)
CUDA	0.000060	0.017100
OpenMP	0.000108	0.090296

为OpenMP设置不同的线程数（1，2，4，32，64，128，256），找到表现最好的设置（小规模为1，大规模为64），CUDA每个线程负责一个数据的处理，各运行100遍，计算平均运行时间，比较CPU和GPU的性能。

可以看到无论在小规模的矩阵还是在大规模的矩阵，在此类计算任务上，GPU的表现性能都比OpenMP好，并且矩阵规模越大，与OpenMP相比CUDA的计算性能越显优势——小规模矩阵上，OpenMp的计算时间约是CUDA的1.8倍，大规模矩阵上，OpenMP的计算时间约是CUDA计算时间的5.28倍。

注释：

CUDA一维代码见`1dmain.cu`，输出文件见`1dinput1.out`，`1dinput2.out`

CUDA二维代码见`2dmian.cu`，输出文件见`2dinput1.out`，`2dinput2.out`

OpenMP代码见`omp.cpp`，输出文件见`OMPoutput1.bin`，`OMPoutput2.bin`