《并行计算》上机报告

姓名：	魏钊	学号：	PB18111699	日期：	2021/5/25
上机题目：	GPU 并行编程实验				

实验环境：
CPU： i7-8750H; 内存： 16GB; 操作系统 WIN10; 软件平台:Visual Studio 2017+CUDA11;

一、算法设计与分析：

题目一:
GPU 上矩阵乘法：
每一个线程计算 C 矩阵中的一个元素。
每一个线程从全局存储器读入 A 矩阵的一行和 B 矩阵的一列。
A 矩阵和 B 矩阵中每个元素都被方位 N =BLOCK_SIZE 次
题目二：
GPU 上矩阵向量乘：
Ax=b
A 为矩阵，x,b 为列向量，类似矩阵乘法不再赘述。

二、核心代码：

题目一:

```
cudaMalloc((void**)&g_mat1, sizeof(int) * M_SIZE);
cudaMalloc((void**)&g_mat2, sizeof(int) * M_SIZE);
cudaMalloc((void**)&g_mat_result, sizeof(int) * M_SIZE);

cudaMemcpy(g_mat1, mat1, sizeof(int) * M_SIZE, cudaMemcpyHostToDevice);
cudaMemcpy(g_mat2, mat2, sizeof(int) * M_SIZE, cudaMemcpyHostToDevice);

/*并行方法*/
startTime = clock();//计时开始

mat_mul <<<BLOCK_NUM, THREAD_NUM >>> (g_mat1, g_mat2, g_mat_result);

cudaMemcpy(result, g_mat_result, sizeof(int) * M_SIZE, cudaMemcpyDeviceToHost);
```

```cpp
__global__ void mat_mul(int* mat1, int* mat2, int* result)
{
    const int bid = blockIdx.x; //块 id
    const int tid = threadIdx.x; //进程 id
    // 每个线程计算一行
    const int row = bid * THREAD_NUM + tid; //计算当前进程所需计算的行数
    for (int c = 0; c < R_SIZE; c++)
    {
        for (int n = 0; n < R_SIZE; n++)
        {
            result[row * R_SIZE + c] += mat1[row * R_SIZE + n] * mat2[n * R_SIZE + c];
        }
    }
}
```

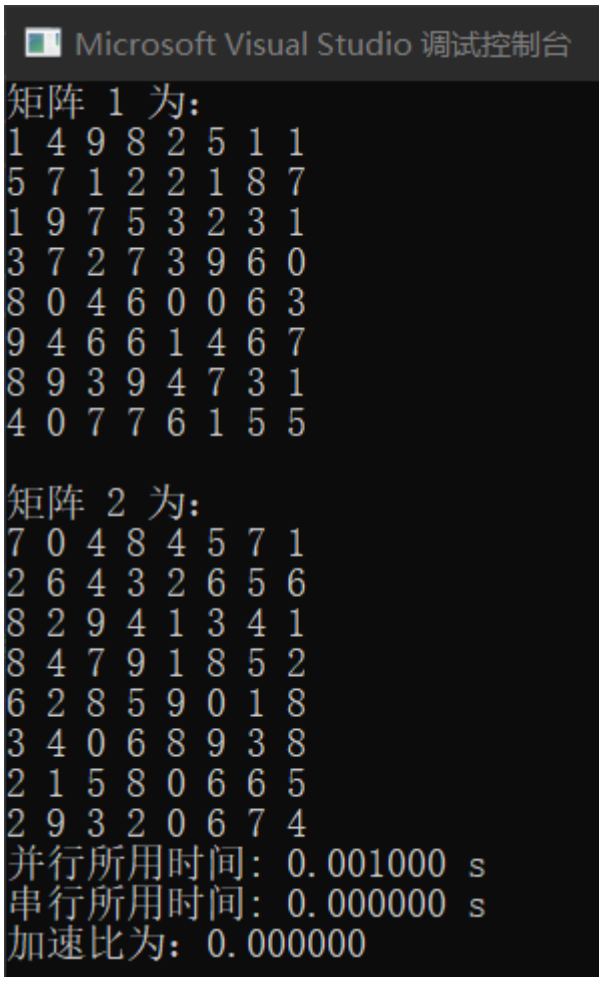
题目二：

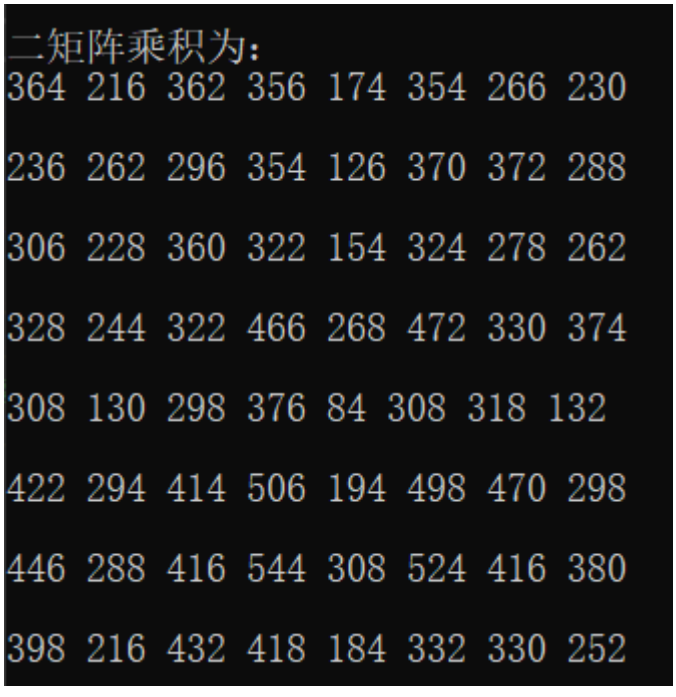
```cpp
// Kernal:
__global__ void MatrixMultiply(double *a, double * b, double *c, int N) {
    int tx = threadIdx.x + blockIdx.x * blockDim.x;
    if (tx < N) {
        double sum = 0;
        for (int k = 0; k < N; ++k) {
            sum += a[tx * N + k] * b[k];
        }
        c[tx] = sum;
    }
}
```

```cpp
cudaError_t matrixMultiplyWithCuda(double *a, double *b, double *c, size_t N) {
    double *dev_a = 0;
    double *dev_b = 0;
    double *dev_c = 0;
    cudaError_t cudaStatus;
    cudaStatus = cudaMalloc((void**)&dev_a, N * N * sizeof(double));
    cudaStatus = cudaMalloc((void**)&dev_b, N * sizeof(double));
    cudaStatus = cudaMalloc((void**)&dev_c, N * sizeof(double));
    cudaStatus = cudaMemcpy(dev_a, a, N * N * sizeof(double), cudaMemcpyHostToDevice);
    cudaStatus = cudaMemcpy(dev_b, b, N * sizeof(double), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        printf("错误②! \n");
        goto Error;
    }
    // kernal invocation
    dim3 threadPerBlock(500, 1, 1);
    dim3 numBlocks(N / threadPerBlock.x + 1, 1, 1);
    MatrixMultiply <<<numBlocks, threadPerBlock >>> (dev_a, dev_b, dev_c, N);
    if (cudaStatus != cudaSuccess) {
        printf("计算错误\n");
        goto Error;
    }
    cudaStatus = cudaMemcpy(c, dev_c, N * sizeof(double), cudaMemcpyDeviceToHost);
```

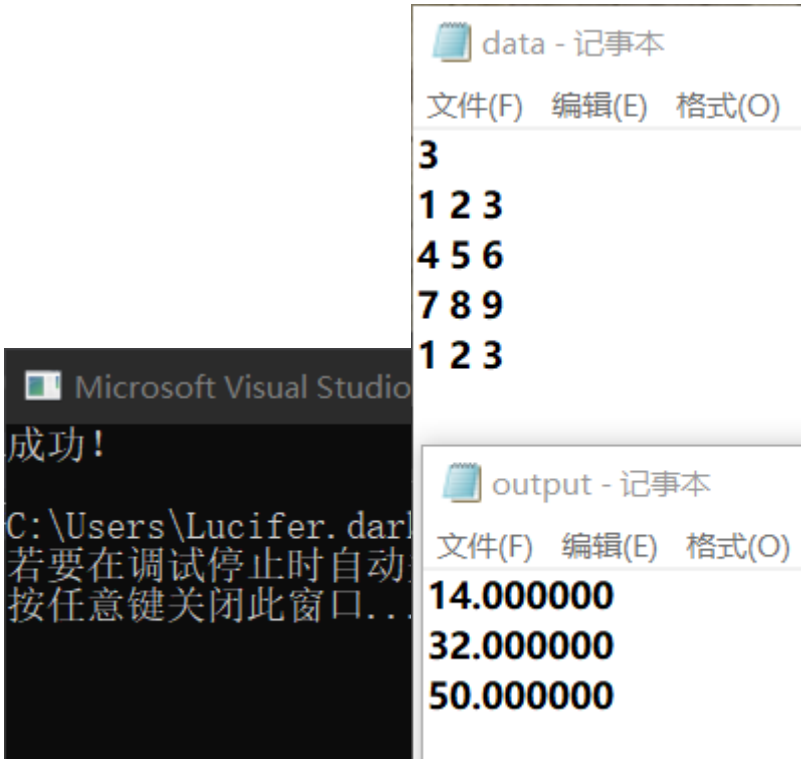
三、结果与分析：

题目一:


Microsoft Visual Studio 调试控制台
矩阵 1 为:
1 4 9 8 2 5 1 1
5 7 1 2 2 1 8 7
1 9 7 5 3 2 3 1
3 7 2 7 3 9 6 0
8 0 4 6 0 0 6 3
9 4 6 6 1 4 6 7
8 9 3 9 4 7 3 1
4 0 7 7 6 1 5 5
矩阵 2 为:
7 0 4 8 4 5 7 1
2 6 4 3 2 6 5 6
8 2 9 4 1 3 4 1
8 4 7 9 1 8 5 2
6 2 8 5 9 0 1 8
3 4 0 6 8 9 3 8
2 1 5 8 0 6 6 5
2 9 3 2 0 6 7 4
并行所用时间: 0.001000 s
串行所用时间: 0.000000 s
加速比为: 0.000000


二矩阵乘积为:
364 216 362 356 174 354 266 230
236 262 296 354 126 370 372 288
306 228 360 322 154 324 278 262
328 244 322 466 268 472 330 374
308 130 298 376 84 308 318 132
422 294 414 506 194 498 470 298
446 288 416 544 308 524 416 380
398 216 432 418 184 332 330 252

题目二：



四、备注（* 可选）：

有可能影响结论的因素：

GPU 型号不同结果可能略有差异

总结：

GPU 在大规模数据并行时有明显的优势

附录（源代码）

算法源代码（C/C++/JAVA 描述）

①矩阵乘法

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include<cuda.h>
#include<string.h>
#include<ctime>

#define BLOCK_NUM 4   // 块数量
#define THREAD_NUM 2 // 每个块中的线程数
#define R_SIZE (BLOCK_NUM * THREAD_NUM) // 矩阵行列数
#define M_SIZE (R_SIZE * R_SIZE) //矩阵规模

__global__ void mat_mul(int* mat1, int* mat2, int* result)
{
    const int bid = blockIdx.x; //块 id
    const int tid = threadIdx.x; //进程 id
```

```
    // 每个线程计算一行
    const int row = bid * THREAD_NUM + tid; //计算当前进程所需计算的行数
    for (int c = 0; c < R_SIZE; c++)
    {
        for (int n = 0; n < R_SIZE; n++)
        {
            result[row * R_SIZE + c] += mat1[row * R_SIZE + n] * mat2[n * R_SIZE + c];
        }
    }
}

int main(int argc, char* argv[])
{
    int* mat1, *mat2, *result;
    int* g_mat1, *g_mat2, *g_mat_result;
    double time_pc, time_normal;

    clock_t startTime, endTime;

    // 用一位数组表示二维矩阵
    mat1 = (int*)malloc(M_SIZE * sizeof(int));
    mat2 = (int*)malloc(M_SIZE * sizeof(int));
    result = (int*)malloc(M_SIZE * sizeof(int));

    // initialize
    for (int i = 0; i < M_SIZE; i++)
    {
        mat1[i] = rand() % 10;
        mat2[i] = rand() % 10;
        result[i] = 0;
    }

    printf("矩阵 1 为: \n");
    for (int i = 0; i < M_SIZE; i++)
        if((i + 1) % R_SIZE == 0)
            printf("%d\n", mat1[i]);
        else
            printf("%d ", mat1[i]);

    printf("\n矩阵 2 为: \n");
    for (int i = 0; i < M_SIZE; i++)
        if ((i + 1) % R_SIZE == 0)
            printf("%d\n", mat2[i]);
```

```
        else
            printf("%d ", mat2[i]);


    cudaMalloc((void**)&g_mat1, sizeof(int) * M_SIZE);
    cudaMalloc((void**)&g_mat2, sizeof(int) * M_SIZE);
    cudaMalloc((void**)&g_mat_result, sizeof(int) * M_SIZE);

    cudaMemcpy(g_mat1, mat1, sizeof(int) * M_SIZE,
cudaMemcpyHostToDevice);
    cudaMemcpy(g_mat2, mat2, sizeof(int) * M_SIZE,
cudaMemcpyHostToDevice);

    /*并行方法*/
    startTime = clock();//计时开始

    mat_mul <<<BLOCK_NUM, THREAD_NUM >>> (g_mat1, g_mat2, g_mat_result);

    cudaMemcpy(result, g_mat_result, sizeof(int) * M_SIZE,
cudaMemcpyDeviceToHost);

    endTime = clock();//计时结束
    time_pc = (double)(endTime - startTime) / CLOCKS_PER_SEC;
    printf("并行所用时间: %lf s\n", time_pc);


    /*串行方法*/
    startTime = clock();//计时开始

    for (int r = 0; r < R_SIZE; r++) {
        for (int c = 0; c < R_SIZE; c++) {
            for (int n = 0; n < R_SIZE; n++) {
                result[r * R_SIZE + c] += mat1[r * R_SIZE + n] * mat2[n
* R_SIZE + c];
            }
        }
    }

    endTime = clock();//计时结束
    time_normal = (double)(endTime - startTime) / CLOCKS_PER_SEC;
    printf("串行所用时间: %lf s\n", time_normal);

    printf("加速比为：%lf\n", time_normal / time_pc);
```

```
    printf("\n二矩阵乘积为：\n");
    for (int i = 0; i < M_SIZE; i++)
        if ((i + 1) % R_SIZE == 0)
            printf("%d\n\n", result[i]);
        else
            printf("%d ", result[i]);
}
```

②矩阵向量乘

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>
#include <fstream>
#include <iomanip>
#include <stdio.h>

// Kernal:
__global__ void MatrixMultiply(double *a, double * b, double *c, int N) {
    int tx = threadIdx.x + blockIdx.x * blockDim.x;
    if (tx < N) {
        double sum = 0;
        for (int k = 0; k < N; ++k) {
            sum += a[tx * N + k] * b[k];
        }
        c[tx] = sum;
    }
}

cudaError_t matrixMultiplyWithCuda(double *a, double *b, double *c, size_t
size);

int main()
{
    std::ifstream in("data.txt");
    int N;
    in >> N;//矩阵阶数
    if (in.fail()) {
        printf("错误①！\n");
    }
    else {
        printf("成功！\n");
    }
    // host initial
    double *a = new double[N * N];
    double *b = new double[N];
```

```
    double *c = new double[N];

    // read
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j) in >> a[i * N + j];

    for (int i = 0; i < N; ++i) in >> b[i];

    cudaError_t cudaStatus = matrixMultiplyWithCuda(a, b, c, N);

    std::ofstream out("output.txt");
    for (int i = 0; i < N; ++i) {
        out << std::setiosflags(std::ios::fixed) << c[i] << " ";
        out << std::endl;
    }
    cudaStatus = cudaThreadExit();

    // host free
    delete[] a;
    delete[] b;
    delete[] c;
    return 0;
}
cudaError_t matrixMultiplyWithCuda(double *a, double *b, double *c, size_t
N) {
    double *dev_a = 0;
    double *dev_b = 0;
    double *dev_c = 0;
    cudaError_t cudaStatus;
    cudaStatus = cudaMalloc((void**)&dev_a, N * N * sizeof(double));
    cudaStatus = cudaMalloc((void**)&dev_b, N * sizeof(double));
    cudaStatus = cudaMalloc((void**)&dev_c, N * sizeof(double));
    cudaStatus = cudaMemcpy(dev_a, a, N * N * sizeof(double),
cudaMemcpyHostToDevice);
    cudaStatus = cudaMemcpy(dev_b, b, N * sizeof(double),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        printf("错误②! \n");
        goto Error;
    }
    // kernal invocation
    dim3 threadPerBlock(500, 1, 1);
    dim3 numBlocks(N / threadPerBlock.x + 1, 1, 1);
    MatrixMultiply <<<numBlocks, threadPerBlock >>> (dev_a, dev_b, dev_c,
```

```
N);
    if (cudaStatus != cudaSuccess) {
        printf("计算错误\n");
        goto Error;
    }
    cudaStatus = cudaMemcpy(c, dev_c, N * sizeof(double),
cudaMemcpyDeviceToHost);
Error:
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
    return cudaStatus;
}
```