并行计算 上机报告

> 上机题目：
>
> 1. 向量加法。定义A,B两个一维数组,编写GPU程序将A和B对应项相加,将结果保存在数组C中。分别测试数组规模为10W、20W、100W、200W、1000W、2000W时其与CPU加法的运行时间之比。
> 2. 矩阵乘法。定义A,B两个二维数组。使用GPU实现矩阵乘法。并对比串行程序,给出加速比。
>
> 姓名：张劲暾
>
> 学号：PB16111485
>
> 日期：2019年5月20日
>
> 实验环境：
>
> CPU：Intel® Xeon® CPU E5-2650 v4 @ 2.20GHz × 47
>
> GPU: NVIDIA Corporation GP102 [GeForce GTX 1080 Ti] (rev a1) x 8
>
> 内存：263858944 KB (264 GB)
>
> 操作系统：Linux G101 3.10.0-693.21.1.el7.x86_64
>
> 软件平台：nvcc (NVIDIA® Cuda compiler driver) 8.0 V8.0.44

算法设计与分析

题目一

向量加法。定义A,B两个一维数组,编写GPU程序将A和B对应项相加,将结果保存在数组C中。分别测试数组规模为10W、20W、100W、200W、1000W、2000W时其与CPU加法的运行时间之比。

设计：

1. 每个kernel函数只计算一个位的加法
2. 在device上动态分配内存空间
3. 将主机上的数据copy到device
4. device并行计算
5. 将device上的结果copy回主机

完整实验代码见附录

```c++
// 核函数
__global__ void vectorAdd(const float* A, const float* B, float* C, int length)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i < length)
    {
        C[i] = A[i] + B[i];
    }
}
// 并行调用
```

```
size_t size = lengths[i] * sizeof(float);

float* host_A = (float*)malloc(size);
float* host_B = (float*)malloc(size);
float* host_C = (float*)malloc(size);
float* host_C_Serial = (float*)malloc(size);

for(int j = 0; j < lengths[i]; j++)
{
        host_A[j] = rand() / MY_RAND_MAX;
        host_B[j] = rand() / MY_RAND_MAX;
}

float* device_A = NULL;
float* device_B = NULL;
float* device_C = NULL;

cudaMalloc((void**)&device_A, size);
cudaMalloc((void**)&device_B, size);
cudaMalloc((void**)&device_C, size);

int threadsPerBlock = 1024;
int blocksPerGrid = (lengths[i] + threadsPerBlock - 1) / threadsPerBlock;

gettimeofday(&beginTime, NULL);
//-------------------------------------------------------------------------------------
cudaMemcpy(device_A, host_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(device_B, host_B, size, cudaMemcpyHostToDevice);
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(device_A, device_B, device_C, lengths[i]);
cudaMemcpy(host_C, device_C, size, cudaMemcpyDeviceToHost);
//-------------------------------------------------------------------------------------
gettimeofday(&endTime, NULL);

int cudaTime_us = (endTime.tv_sec - beginTime.tv_sec) * 1e6 + (endTime.tv_usec - beginTime.tv_usec);

```

结果：结果正确

```shell
[test001@G101 WulingYan]$ nvcc ./vectorAdd.cu -o ./vectorAdd
[test001@G101 WulingYan]$ ./vectorAdd
length:     100000, Speedup ratio = 1.044118
length:     200000, Speedup ratio = 1.291619
length:    1000000, Speedup ratio = 1.559444
length:    2000000, Speedup ratio = 1.730396
length:   10000000, Speedup ratio = 1.524786
length:   20000000, Speedup ratio = 1.642759
[test001@G101 WulingYan]$
```

题目二

矩阵乘法。定义A,B两个二维数组。使用GPU实现矩阵乘法。并对比串行程序,给出加速比。

设计：

1. 每个kernel函数只计算一个矩阵元素
2. 在device上动态分配内存空间
3. 将主机上的数据copy到device

4. 在每个block静态分配shared memory
5. 所有线程并行合作，将计算所需要的子矩阵加载到shared memory
6. 同步进程，保证所有数据到达shared memory
7. 利用shared memory中的数据计算对应矩阵中的元素
8. 同步进程，保证所有线程同时进入下一个循环，不存在抢先修改上一个循环shared memory内容的问题
9. 将计算结果写回device
10. 将device上的结果copy回主机

完整实验代码见附录

```c++
// 核函数
__global__ void matrixMultiply( float* C,
                                const float* A,
                                const float* B,
                                const int widthA,
                                const int widthB
                                )
{
    int block_x = blockIdx.x;
    int block_y = blockIdx.y;

    int thread_x = threadIdx.x;
    int thread_y = threadIdx.y;
/********************************************************************************/
    int A_start = widthA * BLOCK_SIZE * block_y;
    int A_end   = A_start + widthA - 1;
    int A_step  = BLOCK_SIZE;
    int B_start = BLOCK_SIZE * block_x;
    int B_step  = BLOCK_SIZE * widthB;

    float C_submatrix = 0.0;

    for(int a = A_start, b = B_start; a <= A_end; a += A_step, b += B_step)
    {
        __shared__ float shared_A[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float shared_B[BLOCK_SIZE][BLOCK_SIZE];

        shared_A[thread_y][thread_x] = A[widthA * thread_y + thread_x + a];
        shared_B[thread_y][thread_x] = B[widthB * thread_y + thread_x + b];

        __syncthreads();

        // 循环展开
        #pragma unroll

        for(int k = 0; k < BLOCK_SIZE; k++)
        {
            C_submatrix += shared_A[thread_y][k] * shared_B[k][thread_x];
        }

        __syncthreads();
    }
    int blo_bias = ( widthB * block_y  + block_x  ) * BLOCK_SIZE;
    int ele_bias = ( widthB * thread_y + thread_x );
    C[ blo_bias + ele_bias ] = C_submatrix;
/********************************************************************************/
}
// 并行调用
```

```
struct timeval beginTime, endTime;

size_t size_A = sizeof(float) * WIDTH_A * HEIGHT_A;
size_t size_B = sizeof(float) * WIDTH_B * HEIGHT_B;
size_t size_C = sizeof(float) * WIDTH_B * HEIGHT_A;

float* host_A        = (float* )malloc( size_A );
float* host_B        = (float* )malloc( size_B );
float* host_C        = (float* )malloc( size_C );
float* host_C_Serial = (float* )malloc( size_C );

float* device_A = NULL;
float* device_B = NULL;
float* device_C = NULL;

cudaMalloc( (void**)&device_A, size_A );
cudaMalloc( (void**)&device_B, size_B );
cudaMalloc( (void**)&device_C, size_C );


for(int j = 0; j < WIDTH_A * HEIGHT_A; j++)
{
        host_A[j] = rand() / MY_RAND_MAX;
}
for(int j = 0; j < WIDTH_B * HEIGHT_B; j++)
{
        host_B[j] = rand() / MY_RAND_MAX;
}

dim3 block( BLOCK_SIZE, BLOCK_SIZE) ;
dim3 grid(  WIDTH_B / BLOCK_SIZE, HEIGHT_A / BLOCK_SIZE);

gettimeofday(&beginTime, NULL);
//--------------------------------------------------------------------------------
cudaMemcpy(device_A, host_A, size_A, cudaMemcpyHostToDevice);
cudaMemcpy(device_B, host_B, size_B, cudaMemcpyHostToDevice);
matrixMultiply<<<grid, block>>>(device_C, device_A, device_B, WIDTH_A, WIDTH_B);
cudaMemcpy(host_C, device_C, size_C, cudaMemcpyDeviceToHost);
//--------------------------------------------------------------------------------
gettimeofday(&endTime, NULL);

int cudaTime_us = (endTime.tv_sec - beginTime.tv_sec) * 1e6 + (endTime.tv_usec - beginTime.tv_usec);
```

结果：结果正确

```shell
[test001@G101 WulingYan]$ nvcc ./matrixMultiply.cu -o ./matrixMultiply
[test001@G101 WulingYan]$ ./matrixMultiply
Speedup ratio = 1037.384033
[test001@G101 WulingYan]$
```

总结

1. 通过算法实现锻炼了并行思维，熟悉了Cuda编程环境的使用。
2. 验证了shared memory优化“几百倍加速”的理论

附录

向量加法vectorAdd.cu

```c++
#include <stdio.h>
#include <sys/time.h>
#include <cuda_runtime.h>

#define MY_RAND_MAX 100.0

__global__ void vectorAdd(const float* A, const float* B, float* C, int length)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i < length)
    {
        C[i] = A[i] + B[i];
    }
}

int main(void)
{
    struct timeval beginTime, endTime;
    int lengths[6] = {(int)1e5, (int)2e5, (int)1e6, (int)2e6, (int)1e7, (int)2e7};

    for(int i = 0; i < 6; i++)
    {
        size_t size = lengths[i] * sizeof(float);

        float* host_A = (float*)malloc(size);
        float* host_B = (float*)malloc(size);
        float* host_C = (float*)malloc(size);
        float* host_C_Serial = (float*)malloc(size);

        for(int j = 0; j < lengths[i]; j++)
        {
            host_A[j] = rand() / MY_RAND_MAX;
            host_B[j] = rand() / MY_RAND_MAX;
        }

        float* device_A = NULL;
        float* device_B = NULL;
        float* device_C = NULL;

        cudaMalloc((void**)&device_A, size);
        cudaMalloc((void**)&device_B, size);
        cudaMalloc((void**)&device_C, size);

        int threadsPerBlock = 1024;
        int blocksPerGrid = (lengths[i] + threadsPerBlock - 1) / threadsPerBlock;

        gettimeofday(&beginTime, NULL);
    //------------------------------------------------------------------------------------
        cudaMemcpy(device_A, host_A, size, cudaMemcpyHostToDevice);
        cudaMemcpy(device_B, host_B, size, cudaMemcpyHostToDevice);
        vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(device_A, device_B, device_C, lengths[i]);
        cudaMemcpy(host_C, device_C, size, cudaMemcpyDeviceToHost);
```

```
    //-------------------------------------------------------------------------------------
        gettimeofday(&endTime, NULL);

        int cudaTime_us = (endTime.tv_sec - beginTime.tv_sec) * 1e6 + (endTime.tv_usec - beginTime.tv_usec);

        gettimeofday(&beginTime, NULL);
    //-------------------------------------------------------------------------------------
        for(int j = 0; j < lengths[i]; j++)
        {
            host_C_Serial[j] = host_A[j] + host_B[j];
        }
    //-------------------------------------------------------------------------------------
        gettimeofday(&endTime, NULL);

        int serialTime_us = (endTime.tv_sec - beginTime.tv_sec) * 1e6 + (endTime.tv_usec - beginTime.tv_usec);

        for(int j = 0; j < lengths[i]; j++)
        {
            if(fabs(host_C[j] - host_C_Serial[j]) > 1e-5)
            {
                printf("Seems Wrong.\n");
            }
        }
        printf("length: %10d, Speedup ratio = %.6f\n", lengths[i], (float)serialTime_us / (float)cudaTime_us );

        cudaFree(device_A);
        cudaFree(device_B);
        cudaFree(device_C);

        free(host_A);
        free(host_B);
        free(host_C);
        free(host_C_Serial);
    }
    return 0;
}
```

矩阵乘法matrixMultiply.cu

```c++
#include <stdio.h>
#include <sys/time.h>
#include <cuda_runtime.h>

#define MY_RAND_MAX 100000000.0

#define WIDTH_A     (int)640
#define HEIGHT_A    (int)640
#define WIDTH_B     (int)640
#define HEIGHT_B    (int)640
#define BLOCK_SIZE  16

__global__ void matrixMultiply( float* C,
                                const float* A,
                                const float* B,
```

```
                                const int widthA,
                                const int widthB
                                )
{
    int block_x = blockIdx.x;
    int block_y = blockIdx.y;

    int thread_x = threadIdx.x;
    int thread_y = threadIdx.y;
/********************************************************************************/
    int A_start = widthA * BLOCK_SIZE * block_y;
    int A_end   = A_start + widthA - 1;
    int A_step  = BLOCK_SIZE;
    int B_start = BLOCK_SIZE * block_x;
    int B_step  = BLOCK_SIZE * widthB;

    float C_submatrix = 0.0;

    for(int a = A_start, b = B_start; a <= A_end; a += A_step, b += B_step)
    {
        __shared__ float shared_A[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float shared_B[BLOCK_SIZE][BLOCK_SIZE];

        shared_A[thread_y][thread_x] = A[widthA * thread_y + thread_x + a];
        shared_B[thread_y][thread_x] = B[widthB * thread_y + thread_x + b];

        __syncthreads();

        // 循环展开
        #pragma unroll

        for(int k = 0; k < BLOCK_SIZE; k++)
        {
            C_submatrix += shared_A[thread_y][k] * shared_B[k][thread_x];
        }

        __syncthreads();
    }
    int blo_bias = ( widthB * block_y  + block_x  ) * BLOCK_SIZE;
    int ele_bias = ( widthB * thread_y + thread_x );
    C[ blo_bias + ele_bias ] = C_submatrix;
/********************************************************************************/
}

int main()
{
    struct timeval beginTime, endTime;

    size_t size_A = sizeof(float) * WIDTH_A * HEIGHT_A;
    size_t size_B = sizeof(float) * WIDTH_B * HEIGHT_B;
    size_t size_C = sizeof(float) * WIDTH_B * HEIGHT_A;

    float* host_A        = (float* )malloc( size_A );
    float* host_B        = (float* )malloc( size_B );
    float* host_C        = (float* )malloc( size_C );
    float* host_C_Serial = (float* )malloc( size_C );

    float* device_A = NULL;
    float* device_B = NULL;
    float* device_C = NULL;
```

```
    cudaMalloc( (void**)&device_A, size_A );
    cudaMalloc( (void**)&device_B, size_B );
    cudaMalloc( (void**)&device_C, size_C );


    for(int j = 0; j < WIDTH_A * HEIGHT_A; j++)
    {
        host_A[j] = rand() / MY_RAND_MAX;
    }
    for(int j = 0; j < WIDTH_B * HEIGHT_B; j++)
    {
        host_B[j] = rand() / MY_RAND_MAX;
    }

    dim3 block( BLOCK_SIZE, BLOCK_SIZE) ;
    dim3 grid(  WIDTH_B / BLOCK_SIZE, HEIGHT_A / BLOCK_SIZE);

    gettimeofday(&beginTime, NULL);
//-------------------------------------------------------------------------------------
    cudaMemcpy(device_A, host_A, size_A, cudaMemcpyHostToDevice);
    cudaMemcpy(device_B, host_B, size_B, cudaMemcpyHostToDevice);
    matrixMultiply<<<grid, block>>>(device_C, device_A, device_B, WIDTH_A, WIDTH_B);
    cudaMemcpy(host_C, device_C, size_C, cudaMemcpyDeviceToHost);
//-------------------------------------------------------------------------------------
    gettimeofday(&endTime, NULL);

    int cudaTime_us = (endTime.tv_sec - beginTime.tv_sec) * 1e6 + (endTime.tv_usec - beginTime.tv_usec);

    gettimeofday(&beginTime, NULL);
//-------------------------------------------------------------------------------------
    for(int i = 0; i < HEIGHT_A; i++)
    {
        for(int j = 0; j < WIDTH_B; j++)
        {
            host_C_Serial[i * WIDTH_B + j] = 0.0;
            for(int k = 0; k < WIDTH_A; k++)
            {
                host_C_Serial[i * WIDTH_B + j] += host_A[i * WIDTH_A + k] * host_B[k * WIDTH_B + j];
            }
        }
    }
//-------------------------------------------------------------------------------------
    gettimeofday(&endTime, NULL);

    int serialTime_us = (endTime.tv_sec - beginTime.tv_sec) * 1e6 + (endTime.tv_usec - beginTime.tv_usec);

    for(int i = 0; i < HEIGHT_A; i++)
    {
        for(int j = 0; j < WIDTH_B; j++)
        {

            if(fabs(host_C[i * WIDTH_B + j] - host_C_Serial[i * WIDTH_B + j]) > 1e-1)
            {
                printf("Seems Wrong at %d, %d , %f, %f.\n", i, j, host_C[i * WIDTH_B + j],host_C_Serial[i * WIDTH_B + j]);
                exit(0);
            }
        }
```

```
    }

    printf("Speedup ratio = %.6f\n", (float)serialTime_us / (float)cudaTime_us );

    cudaFree(device_A);
    cudaFree(device_B);
    cudaFree(device_C);

    free(host_A);
    free(host_B);
    free(host_C);
    free(host_C_Serial);

    return 0;
}
```