并行计算实验 3: GPU

傅申 PB20000051

1. 实验目的

使用 CUDA 编写在 GPU 上运行的并行程序.

向量加法 定义 $\boldsymbol{A}$, $\boldsymbol{B}$ 两个一维数组, 编写 GPU 程序将 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 对应项相加, 将结果保存在数组 $\boldsymbol{C}$ 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.

矩阵乘法 定义 $\boldsymbol{A}$, $\boldsymbol{B}$ 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.

2. 实验环境

本次实验在我自己的笔记本上运行, 硬件参数如下:

CPU Intel i5-1035G1, 4 核 8 进程.

内存 双通道 16 GB DDR4 3200 MHz 内存.

GPU NVIDIA GeForce MX330, 2 GB GDDR5 显存.

相应的软件环境如下:

OS GNU/Linux 6.1.31-2-MANJARO x86_64.

CUDA 12.1

编译器 nvcc

编译选项 `-O0`

3. 实验过程与结果

3.1. 向量加法

3.1.1. 实现

每个线程负责一个元素的加法, kernel 如下:

```
__global__ static void vecadd_kernel(const float *a, const float *b, float *c,
                                     size_t size)
{
    size_t index = blockDim.x * blockIdx.x + threadIdx.x;

    if (index < size) {
        c[index] = a[index] + b[index];
    }
}
```

在 `main` 函数中, 将依次进行不同数据规模的测试. 在每个测试中, 运行过程如下:

(1) 分配主机上的存储空间;

(2) 随机生成对应规模的向量;

(3) 分配设备 (GPU) 上的存储空间;

(4) 设置线程块和线程网络大小;

```
// #define NUM_THREADS 256
dim3 dim_block(NUM_THREADS, 1, 1);
int  num_block = (size - 1) / NUM_THREADS + 1;
dim3 dim_grid(num_block, 1, 1);
```

(5) 测试 GPU 上的向量加法, 并计时:

```
auto gpu_start = now();
cudaMemcpy(device_a, host_a, size * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(device_b, host_b, size * sizeof(float), cudaMemcpyHostToDevice);
cudaDeviceSynchronize();

auto kernel_start = now();
vecadd_kernel<<<dim_grid, dim_block>>>(device_a, device_b, device_c, size);
cudaDeviceSynchronize();
auto kernel_end = now();

cudaMemcpy(host_c_gpu, device_c, size * sizeof(float), cudaMemcpyDeviceToHost);
auto gpu_end = now();
```

(6) 测试 CPU 上的向量加法, 并计时;

(7) 检查结果是否正确;

- 若两个向量不相等, 则会输出错误信息并停止程序.

(8) 输出运行时间数据与加速比;

(9) 释放存储空间.

因为将向量从主机拷贝到设备上的时间开销较大, 所以 GPU 计算的总时间 (包括数据拷贝) 和 CPU 计算的总时间相比并没有优势, 但是只运行 kernel 的时间开销较小. 因此, 程序将会统计两种时间开销, 并分别计算加速比.

3.1.2. 结果

运行程序一次, 输出如下:

```
$ ./vecadd
```

size	running time (ms)			speedup	
	GPU total	kernel	CPU	GPU total	kernel
100000	0.69	0.04	0.40	0.58	10.58
200000	1.32	0.06	0.87	0.66	14.12
1000000	4.62	0.25	3.77	0.82	14.89
2000000	9.09	0.50	6.92	0.76	13.95
10000000	42.41	2.43	34.53	0.81	14.21
20000000	85.04	4.84	66.43	0.78	13.71

运行程序多次, 统计平均用时和加速比, 如下:

向量规模	运行时间/ms			加速比	
	GPU	kernel	CPU	GPU	kernel
100000	0.707	0.039	0.417	0.60	10.69
200000	1.355	0.061	0.809	0.60	13.26
1000000	4.859	0.256	3.625	0.75	14.16
2000000	9.220	0.501	6.362	0.69	12.70
10000000	42.922	2.433	34.292	0.80	14.09
20000000	83.975	4.851	67.256	0.80	13.86

3.2. 矩阵乘法

3.2.1. 实现

采用分块算法, 并且使用共享存储器优化, 每个线程块负责积的一个子矩阵的计算, kernel 如下:

```
__global__ static void matmul_kernel(const float *a, const float *b, float *c,
                                      size_t size)
{
    __shared__ float a_sub[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float b_sub[BLOCK_SIZE][BLOCK_SIZE];

    /* Block indices */
    const size_t block_x = blockIdx.x;
    const size_t block_y = blockIdx.y;

    /* Thread indices */
    const size_t thread_x = threadIdx.x;
    const size_t thread_y = threadIdx.y;

    /* Number of blocks */
    const size_t num_sub = (size + BLOCK_SIZE - 1) / BLOCK_SIZE;

    float c_sub = 0;

    for (size_t sub = 0; sub < num_sub; sub++) {
        /* Load the sub-matrices into the shared memory */
        /* Each thread loads one element of each sub-matrix */
        size_t a_x = block_x * BLOCK_SIZE + thread_x;
        size_t a_y = sub * BLOCK_SIZE + thread_y;
        if (a_x < size and a_y < size) {
            a_sub[thread_x][thread_y] = a[a_x * size + a_y];
        } else {
            a_sub[thread_x][thread_y] = 0;
        }

        size_t b_x = sub * BLOCK_SIZE + thread_x;
        size_t b_y = block_y * BLOCK_SIZE + thread_y;
        if (b_x < size and b_y < size) {
            b_sub[thread_x][thread_y] = b[b_x * size + b_y];
```

```
        } else {
            b_sub[thread_x][thread_y] = 0;
        }

        __syncthreads();

        /* Multiply the two sub-matrices */
        for (size_t k = 0; k < BLOCK_SIZE; k++) {
            c_sub += a_sub[thread_x][k] * b_sub[k][thread_y];
        }

        __syncthreads();
    }

    size_t c_x = block_x * BLOCK_SIZE + thread_x;
    size_t c_y = block_y * BLOCK_SIZE + thread_y;
    if (c_x < size and c_y < size) {
        c[c_x * size + c_y] = c_sub;
    }
}
```

程序的执行过程与向量加法的执行过程类似, 不再赘述.

3.2.2. 结果

运行程序一次, 输出如下:

```
$ ./matmul
```

size	running time (ms)			speedup	
	GPU total	kernel	CPU	GPU total	kernel
10	0.07	0.03	0.00	0.04	0.10
100	0.24	0.15	2.70	11.38	17.61
1000	61.74	57.24	2729.56	44.21	47.68

运行程序多次, 统计平均用时和加速比, 如下:

向量规模	运行时间/ms			加速比	
	GPU	kernel	CPU	GPU	kernel
10	0.070	0.030	0.003	0.04	0.10
100	0.242	0.151	2.682	11.08	17.76
1000	61.936	57.315	2728.517	44.05	47.60

4. 实验总结

本次实验中, 我了解了 GPU 的基本原理, 并使用 CUDA 编写了在 GPU 上运行的并行程序.