

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：张劲瞰

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) × 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++
1  // 核函数
2  __global__ void vectorAdd(const float* A, const float* B, float* C, int length)
3  {
4      int i = blockDim.x * blockIdx.x + threadIdx.x;
5
6      if(i < length)
7      {
8          C[i] = A[i] + B[i];
9      }
10 }
11 // 并行调用
```

```

12  size_t size = lengths[i] * sizeof(float);
13
14  float* host_A = (float*)malloc(size);
15  float* host_B = (float*)malloc(size);
16  float* host_C = (float*)malloc(size);
17  float* host_C_Serial = (float*)malloc(size);
18
19  for(int j = 0; j < lengths[i]; j++)
20  {
21      host_A[j] = rand() / MY RAND_MAX;
22      host_B[j] = rand() / MY RAND_MAX;
23  }
24
25  float* device_A = NULL;
26  float* device_B = NULL;
27  float* device_C = NULL;
28
29  cudaMalloc((void*)&device_A, size);
30  cudaMalloc((void*)&device_B, size);
31  cudaMalloc((void*)&device_C, size);
32
33  int threadsPerBlock = 1024;
34  int blocksPerGrid = (lengths[i] + threadsPerBlock - 1) / threadsPerBlock;
35
36  gettimeofday(&beginTime, NULL);
37  //-----
38  cudaMemcpy(device_A, host_A, size, cudaMemcpyHostToDevice);
39  cudaMemcpy(device_B, host_B, size, cudaMemcpyHostToDevice);
40  vectorAdd<<<blocksPerGrid, threadsPerBlock>>>>(device_A, device_B, device_C, lengths[i]);
41  cudaMemcpy(host_C, device_C, size, cudaMemcpyDeviceToHost);
42  //-----
43  gettimeofday(&endTime, NULL);
44
45  int cudaTime_us = (endTime.tv_sec - beginTime.tv_sec) * 1e6 + (endTime.tv_usec - beginTime.tv_usec);
46

```

结果：结果正确

```

''' shell
1  [test001@G101 WulingYan]$ nvcc ./vectorAdd.cu -o ./vectorAdd
2  [test001@G101 WulingYan]$ ./vectorAdd
3  length:      100000, Speedup ratio = 1.044118
4  length:      200000, Speedup ratio = 1.291619
5  length:      1000000, Speedup ratio = 1.559444
6  length:      2000000, Speedup ratio = 1.730396
7  length:      10000000, Speedup ratio = 1.524786
8  length:      20000000, Speedup ratio = 1.642759
9  [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++
1 // 核函数
2 __global__ void matrixMultiply(float* C,
3 const float* A,
4 const float* B,
5 const int widthA,
6 const int widthB
7)
8 {
9 int block_x = blockIdx.x;
10 int block_y = blockIdx.y;
11
12 int thread_x = threadIdx.x;
13 int thread_y = threadIdx.y;
14 /*****
15 int A_start = widthA * BLOCK_SIZE * block_y;
16 int A_end = A_start + widthA - 1;
17 int A_step = BLOCK_SIZE;
18 int B_start = BLOCK_SIZE * block_x;
19 int B_step = BLOCK_SIZE * widthB;
20
21 float C_submatrix = 0.0;
22
23 for(int a = A_start, b = B_start; a <= A_end; a += A_step, b += B_step)
24 {
25 __shared__ float shared_A[BLOCK_SIZE][BLOCK_SIZE];
26 __shared__ float shared_B[BLOCK_SIZE][BLOCK_SIZE];
27
28 shared_A[thread_y][thread_x] = A[widthA * thread_y + thread_x + a];
29 shared_B[thread_y][thread_x] = B[widthB * thread_y + thread_x + b];
30
31 __syncthreads();
32
33 // 循环展开
34 #pragma unroll
35
36 for(int k = 0; k < BLOCK_SIZE; k++)
37 {
38 C_submatrix += shared_A[thread_y][k] * shared_B[k][thread_x];
39 }
40
41 __syncthreads();
42 }
43 int blo_bias = (widthB * block_y + block_x) * BLOCK_SIZE;
44 int ele_bias = (widthB * thread_y + thread_x);
45 C[blo_bias + ele_bias] = C_submatrix;
46 /*****
47 }
48 // 并行调用
```

```

49 struct timeval beginTime, endTime;
50
51 size_t size_A = sizeof(float) * WIDTH_A * HEIGHT_A;
52 size_t size_B = sizeof(float) * WIDTH_B * HEIGHT_B;
53 size_t size_C = sizeof(float) * WIDTH_B * HEIGHT_A;
54
55 float* host_A = (float*) malloc(size_A);
56 float* host_B = (float*) malloc(size_B);
57 float* host_C = (float*) malloc(size_C);
58 float* host_C_Serial = (float*) malloc(size_C);
59
60 float* device_A = NULL;
61 float* device_B = NULL;
62 float* device_C = NULL;
63
64 cudaMalloc((void**)&device_A, size_A);
65 cudaMalloc((void**)&device_B, size_B);
66 cudaMalloc((void**)&device_C, size_C);
67
68
69 for(int j = 0; j < WIDTH_A * HEIGHT_A; j++)
70 {
71 host_A[j] = rand() / MY_RAND_MAX;
72 }
73 for(int j = 0; j < WIDTH_B * HEIGHT_B; j++)
74 {
75 host_B[j] = rand() / MY_RAND_MAX;
76 }
77
78 dim3 block(BLOCK_SIZE, BLOCK_SIZE);
79 dim3 grid(WIDTH_B / BLOCK_SIZE, HEIGHT_A / BLOCK_SIZE);
80
81 gettimeofday(&beginTime, NULL);
82 //-----
83 cudaMemcpy(device_A, host_A, size_A, cudaMemcpyHostToDevice);
84 cudaMemcpy(device_B, host_B, size_B, cudaMemcpyHostToDevice);
85 matrixMultiply<<<grid, block>>>(device_C, device_A, device_B, WIDTH_A, WIDTH_B);
86 cudaMemcpy(host_C, device_C, size_C, cudaMemcpyDeviceToHost);
87 //-----
88 gettimeofday(&endTime, NULL);
89
90 int cudaTime_us = (endTime.tv_sec - beginTime.tv_sec) * 1e6 + (endTime.tv_usec - beginTime.tv_usec);

```

结果：结果正确

```

''' shell
1 [test001@G101 WulingYan]$ nvcc ./matrixMultiply.cu -o ./matrixMultiply
2 [test001@G101 WulingYan]$./matrixMultiply
3 Speedup ratio = 1037.384033
4 [test001@G101 WulingYan]$

```

## ## 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++
1  #include <stdio.h>
2  #include <sys/time.h>
3  #include <cuda_runtime.h>
4
5  #define MY_RAND_MAX 100.0
6
7  __global__ void vectorAdd(const float* A, const float* B, float* C, int length)
8  {
9      int i = blockDim.x * blockIdx.x + threadIdx.x;
10
11      if(i < length)
12      {
13          C[i] = A[i] + B[i];
14      }
15  }
16
17  int main(void)
18  {
19      struct timeval beginTime, endTime;
20      int lengths[6] = {(int)1e5, (int)2e5, (int)1e6, (int)2e6, (int)1e7, (int)2e7};
21
22      for(int i = 0; i < 6; i++)
23      {
24          size_t size = lengths[i] * sizeof(float);
25
26          float* host_A = (float*)malloc(size);
27          float* host_B = (float*)malloc(size);
28          float* host_C = (float*)malloc(size);
29          float* host_C_Serial = (float*)malloc(size);
30
31          for(int j = 0; j < lengths[i]; j++)
32          {
33              host_A[j] = rand() / MY_RAND_MAX;
34              host_B[j] = rand() / MY_RAND_MAX;
35          }
36
37          float* device_A = NULL;
38          float* device_B = NULL;
39          float* device_C = NULL;
40
41          cudaMalloc((void**)&device_A, size);
42          cudaMalloc((void**)&device_B, size);
43          cudaMalloc((void**)&device_C, size);
44
45          int threadsPerBlock = 1024;
46          int blocksPerGrid = (lengths[i] + threadsPerBlock - 1) / threadsPerBlock;
47
48          gettimeofday(&beginTime, NULL);
49          //-----
50          cudaMemcpy(device_A, host_A, size, cudaMemcpyHostToDevice);
51          cudaMemcpy(device_B, host_B, size, cudaMemcpyHostToDevice);
52          vectorAdd<<<blocksPerGrid, threadsPerBlock>>>>(device_A, device_B, device_C, lengths[i]);
53          cudaMemcpy(host_C, device_C, size, cudaMemcpyDeviceToHost);

```

```

54 //-----
55     gettimeofday(&endTime, NULL);
56
57     int cudaTime_us = (endTime.tv_sec - beginTime.tv_sec) * 1e6 + (endTime.tv_usec -
beginTime.tv_usec);
58
59     gettimeofday(&beginTime, NULL);
60 //-----
61     for(int j = 0; j < lengths[i]; j++)
62     {
63         host_C_Serial[j] = host_A[j] + host_B[j];
64     }
65 //-----
66     gettimeofday(&endTime, NULL);
67
68     int serialTime_us = (endTime.tv_sec - beginTime.tv_sec) * 1e6 + (endTime.tv_usec -
beginTime.tv_usec);
69
70     for(int j = 0; j < lengths[i]; j++)
71     {
72         if(fabs(host_C[j] - host_C_Serial[j]) > 1e-5)
73         {
74             printf("Seems Wrong.\n");
75         }
76     }
77     printf("length: %10d, Speedup ratio = %.6f\n", lengths[i], (float)serialTime_us /
(float)cudaTime_us );
78
79     cudaFree(device_A);
80     cudaFree(device_B);
81     cudaFree(device_C);
82
83     free(host_A);
84     free(host_B);
85     free(host_C);
86     free(host_C_Serial);
87 }
88 return 0;
89 }

```

矩阵乘法matrixMultiply.cu

```

''' c++
1  #include <stdio.h>
2  #include <sys/time.h>
3  #include <cuda_runtime.h>
4
5  #define MY_RAND_MAX 100000000.0
6
7  #define WIDTH_A      (int)640
8  #define HEIGHT_A     (int)640
9  #define WIDTH_B      (int)640
10 #define HEIGHT_B     (int)640
11 #define BLOCK_SIZE   16
12
13 __global__ void matrixMultiply( float* C,
14                                const float* A,
15                                const float* B,

```

```

16         const int widthA,
17         const int widthB
18     )
19 {
20     int block_x = blockIdx.x;
21     int block_y = blockIdx.y;
22
23     int thread_x = threadIdx.x;
24     int thread_y = threadIdx.y;
25     /*****
26     int A_start = widthA * BLOCK_SIZE * block_y;
27     int A_end   = A_start + widthA - 1;
28     int A_step  = BLOCK_SIZE;
29     int B_start = BLOCK_SIZE * block_x;
30     int B_step  = BLOCK_SIZE * widthB;
31
32     float C_submatrix = 0.0;
33
34     for(int a = A_start, b = B_start; a <= A_end; a += A_step, b += B_step)
35     {
36         __shared__ float shared_A[BLOCK_SIZE][BLOCK_SIZE];
37         __shared__ float shared_B[BLOCK_SIZE][BLOCK_SIZE];
38
39         shared_A[thread_y][thread_x] = A[widthA * thread_y + thread_x + a];
40         shared_B[thread_y][thread_x] = B[widthB * thread_y + thread_x + b];
41
42         __syncthreads();
43
44         // 循环展开
45         #pragma unroll
46
47         for(int k = 0; k < BLOCK_SIZE; k++)
48         {
49             C_submatrix += shared_A[thread_y][k] * shared_B[k][thread_x];
50         }
51
52         __syncthreads();
53     }
54     int blo_bias = ( widthB * block_y  + block_x ) * BLOCK_SIZE;
55     int ele_bias = ( widthB * thread_y + thread_x );
56     C[ blo_bias + ele_bias ] = C_submatrix;
57     /*****
58 }
59
60 int main()
61 {
62     struct timeval beginTime, endTime;
63
64     size_t size_A = sizeof(float) * WIDTH_A * HEIGHT_A;
65     size_t size_B = sizeof(float) * WIDTH_B * HEIGHT_B;
66     size_t size_C = sizeof(float) * WIDTH_B * HEIGHT_A;
67
68     float* host_A      = (float*) malloc( size_A );
69     float* host_B      = (float*) malloc( size_B );
70     float* host_C      = (float*) malloc( size_C );
71     float* host_C_Serial = (float*) malloc( size_C );
72
73     float* device_A = NULL;
74     float* device_B = NULL;
75     float* device_C = NULL;

```

```

76
77     cudaMalloc( (void**)&device_A, size_A );
78     cudaMalloc( (void**)&device_B, size_B );
79     cudaMalloc( (void**)&device_C, size_C );
80
81
82     for(int j = 0; j < WIDTH_A * HEIGHT_A; j++)
83     {
84         host_A[j] = rand() / MY_RAND_MAX;
85     }
86     for(int j = 0; j < WIDTH_B * HEIGHT_B; j++)
87     {
88         host_B[j] = rand() / MY_RAND_MAX;
89     }
90
91     dim3 block( BLOCK_SIZE, BLOCK_SIZE );
92     dim3 grid( WIDTH_B / BLOCK_SIZE, HEIGHT_A / BLOCK_SIZE);
93
94     gettimeofday(&beginTime, NULL);
95     //-----
96     cudaMemcpy(device_A, host_A, size_A, cudaMemcpyHostToDevice);
97     cudaMemcpy(device_B, host_B, size_B, cudaMemcpyHostToDevice);
98     matrixMultiply<<<grid, block>>>(device_C, device_A, device_B, WIDTH_A, WIDTH_B);
99     cudaMemcpy(host_C, device_C, size_C, cudaMemcpyDeviceToHost);
100    //-----
101    gettimeofday(&endTime, NULL);
102
103    int cudaTime_us = (endTime.tv_sec - beginTime.tv_sec) * 1e6 + (endTime.tv_usec - beginTime.tv_usec);
104
105    gettimeofday(&beginTime, NULL);
106    //-----
107    for(int i = 0; i < HEIGHT_A; i++)
108    {
109        for(int j = 0; j < WIDTH_B; j++)
110        {
111            host_C_Serial[i * WIDTH_B + j] = 0.0;
112            for(int k = 0; k < WIDTH_A; k++)
113            {
114                host_C_Serial[i * WIDTH_B + j] += host_A[i * WIDTH_A + k] * host_B[k * WIDTH_B + j];
115            }
116        }
117    }
118    //-----
119    gettimeofday(&endTime, NULL);
120
121    int serialTime_us = (endTime.tv_sec - beginTime.tv_sec) * 1e6 + (endTime.tv_usec -
beginTime.tv_usec);
122
123    for(int i = 0; i < HEIGHT_A; i++)
124    {
125        for(int j = 0; j < WIDTH_B; j++)
126        {
127
128            if(fabs(host_C[i * WIDTH_B + j] - host_C_Serial[i * WIDTH_B + j]) > 1e-1)
129            {
130                printf("Seems Wrong at %d, %d , %f, %f.\n", i, j, host_C[i * WIDTH_B +
j], host_C_Serial[i * WIDTH_B + j]);
131                exit(0);
132            }
133        }

```



```
134     }
135
136     printf("Speedup ratio = %.6f\n", (float)serialTime_us / (float)cudaTime_us );
137
138     cudaFree(device_A);
139     cudaFree(device_B);
140     cudaFree(device_C);
141
142     free(host_A);
143     free(host_B);
144     free(host_C);
145     free(host_C_Serial);
146
147     return 0;
148 }
```