《并行计算》上机报告

姓名:	李子旸	学号:	PB16110959	日期:	2019.5.25
上机题目:	GPU				

实验环境:
CPU:Intel 至强 E5-2680 v4, 56 核 GPU:GeForce GTX 1080 * 2
内存:128GB; 操作系统:Ubuntu;

一、算法设计与分析:

题目:

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。

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。

算法设计:

使用 cuda 进行向量加法和矩阵运算

二、核心代码:

向量加法:

```
__global__ void add_in_parallel(int *array_a, int *array_b, int *array_c)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  array_c[tid] = array_a[tid] + array_b[tid];
}



cudaMalloc((void **)&a_dev, arraysize * sizeof(int));
cudaMalloc((void **)&b_dev, arraysize * sizeof(int));
```

```
cudaMalloc((void **)&c_dev, arraysize * sizeof(int));

cudaMemcpy(a_dev, a_host, arraysize * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(b_dev, b_host, arraysize * sizeof(int), cudaMemcpyHostToDevice);

int blocksize = 512;
int blocknum = ceil(arraysize / double(blocksize));

dim3 dimBlock(blocksize, 1, 1);
dim3 dimGrid(blocknum, 1, 1);

add_in_parallel<<<dimGrid, dimBlock>>>(a_dev, b_dev, c_dev);
```

矩阵乘法：

```
__global__ void multiply(const int *a, const int *b, int *c, int n) {
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  int col = blockIdx.y * blockDim.y + threadIdx.y;

  int k;
  int sum = 0;

  if (row < n && col < n) {
    for (k = 0; k < n; k++) {
      sum += a[row * n + k] * b[k * n + col];
    }

    c[row * n + col] = sum;
  }
}



cudaMalloc((void **)&device_a, sizeof(int) * n * n);
cudaMalloc((void **)&device_b, sizeof(int) * n * n);
cudaMalloc((void **)&device_c, sizeof(int) * n * n);


cudaMemcpy(device_a, host_a, sizeof(int) * n * n, cudaMemcpyHostToDevice);
cudaMemcpy(device_b, host_b, sizeof(int) * n * n, cudaMemcpyHostToDevice);

double num = ceil(pow((double)n,2) / pow((double)BLOCKSIZE, 2));
```

```
int gridsize = (int)ceil(sqrt(num));

dim3 dimBlock(BLOCKSIZE, BLOCKSIZE, 1);
dim3 dimGrid(gridsize, gridsize, 1);

multiply<<<dimGrid, dimBlock>>>(device_a, device_b, device_c, n);
```

三、结果与分析:

向量加法:

单位为 ms

	CPU	GPU	CPU/GPU
10w:	0.5745	0.034096	16.8495
20w:	1.055	0.035456	29.7552
100w:	6.175	0.079648	77.5286
200w:	11.408	0.131808	86.5501
1000w:	53.384	0.54048	98.7715
2000w:	94.6495	1.030896	91.8128

矩阵加法:

n*n 矩阵
单位为 ms

n	CPU	GPU	CPU/GPU
10	0.005	0.078	0.064
20	0.045	0.098	0.4592
50	0.693	0.098	7.071
100	5.536	0.175	31.6343
200	43.692	0.459	95.189
500	705.813	4.145	169.798
1000	5636.48	26.025	216.579
2000	53003.927	209.775	252.670
5000	无结果	3743.796	

四、备注（* 可选）：

有可能影响结论的因素：

数据量的大小，CPU cache 的大小。

总结：

CPU 和 GPU 的加速比

在小数据量下，CPU 会比 GPU 快

在大数据量下，GPU 会比 CPU 快很多

性能瓶颈在访存上，只要数据能在 CPU 的 cache 里面存放，那么 CPU 比 GPU 快，放不下的话，GPU 更有优势。

附录（源代码）

算法源代码（C/C++/JAVA 描述）

向量加法：

```
#include <stdio.h>
#include <math.h>

__global__ void add_in_parallel(int *array_a, int *array_b, int *array_c)
{
int tid = blockIdx.x * blockDim.x + threadIdx.x;
array_c[tid] = array_a[tid] + array_b[tid];
}

int main()
{
// --------------------------------------------
printf("Begin...\n");
int arraysize = 100000;
int *a_host;
int *b_host;
int *c_host;
int *devresult_host;

a_host = (int *)malloc(arraysize * sizeof(int));
b_host = (int *)malloc(arraysize * sizeof(int));
c_host = (int *)malloc(arraysize * sizeof(int));
```

```
devresult_host = (int *)malloc(arraysize * sizeof(int));

for (int i = 0; i < arraysize; i++)
{
a_host[i] = i;
b_host[i] = i;
}

// ----------------------------------------------
printf("Allocating device memory...\n");
int *a_dev;
int *b_dev;
int *c_dev;

cudaMalloc((void **)&a_dev, arraysize * sizeof(int));
cudaMalloc((void **)&b_dev, arraysize * sizeof(int));
cudaMalloc((void **)&c_dev, arraysize * sizeof(int));

// ----------------------------------------------
cudaEvent_t start, stop;
float time_from_host_to_dev;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
cudaMemcpy(a_dev, a_host, arraysize * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(b_dev, b_host, arraysize * sizeof(int), cudaMemcpyHostToDevice);
cudaEventRecord(stop, 0);
cudaEventSynchronize(start);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_from_host_to_dev, start, stop);
printf("Copy host data to device, time used: %0.5g seconds\n", time_from_host_to_dev / 1000);

// ----------------------------------------------
float time_of_kernel;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
int blocksize = 512;
int blocknum = ceil(arraysize / double(blocksize));
```

	`` dim3 dimBlock(blocksize, 1, 1); dim3 dimGrid(blocknum, 1, 1); add_in_parallel<<<dimGrid, dimBlock>>>(a_dev, b_dev, c_dev); cudaEventRecord(stop, 0); cudaEventSynchronize(start); cudaEventSynchronize(stop); cudaEventElapsedTime(&time_of_kernel, start, stop); printf("Add in parallel, time used: %0.5g seconds\n", time_of_kernel / 1000); // -- float time_from_dev_to_host; cudaEventCreate(&start); cudaEventCreate(&stop); cudaEventRecord(start, 0); cudaMemcpy(devresult_host, c_dev, arraysize * sizeof(int), cudaMemcpyDeviceToHost); cudaEventRecord(stop, 0); cudaEventSynchronize(start); cudaEventSynchronize(stop); cudaEventElapsedTime(&time_from_dev_to_host, start, stop); printf("Copy dev data to host, time used: %0.5g seconds\n", time_from_dev_to_host / 1000); // --- printf("Verify result...\n"); int status = 0; clock_t start_cpu, end_cpu; float time_cpu; start_cpu = clock(); for (int i = 0; i < arraysize; i++) { c_host[i] = a_host[i] + b_host[i]; } end_cpu = clock(); time_cpu = (double)(end_cpu - start_cpu) / CLOCKS_PER_SEC; for (int i = 0; i < arraysize; i++) { if (c_host[i] != devresult_host[i]) { status = 1; ```

	<pre>} } if (status) { printf("Failed vervified.\n"); } else { printf("Sucessdully verified.\n"); } // --- printf("Free dev memory\n"); cudaFree(a_dev); cudaFree(b_dev); cudaFree(c_dev); // --- printf("Free host memory\n"); free(a_host); free(b_host); free(c_host); // --- printf("\nPerformance: CPU vs. GPU\n"); printf("time cpu:%f\n", time_cpu); printf("time gpu(kernel):%f\n", time_of_kernel / 1000); return 1; }</pre> 矩阵乘法： <pre>#include <stdio.h> #include <stdlib.h> #include <time.h> #include <sys/time.h> #include <unistd.h> #include <cuda.h> #define RANDOM(x) (rand() % x) #define MAX 100000</pre>

```
#define BLOCKSIZE 16

__global__ void multiply(const int *a, const int *b, int *c, int n) {
int row = blockIdx.x * blockDim.x + threadIdx.x;
int col = blockIdx.y * blockDim.y + threadIdx.y;

int k;
int sum = 0;

if (row < n && col < n) {
for (k = 0; k < n; k++) {
sum += a[row * n + k] * b[k * n + col];
}

c[row * n + col] = sum;
}
}

int main(int argc, char **argv) {
int n = 512;
int i, j, k;
timeval start, finish;

if (argc == 2) {
n = atoi(argv[1]);
}

int *host_a = (int *)malloc(sizeof(int) * n * n);
int *host_b = (int *)malloc(sizeof(int) * n * n);
int *host_c = (int *)malloc(sizeof(int) * n * n);
int *host_c2 = (int *)malloc(sizeof(int) * n * n);

srand(time(NULL));

for (i = 0; i < n * n; i++) {
host_a[i] = RANDOM(MAX);
host_b[i] = RANDOM(MAX);
}

cudaError_t error = cudaSuccess;

int *device_a, *device_b, *device_c;
```

	`` error = cudaMalloc((void **)&device_a, sizeof(int) * n * n); error = cudaMalloc((void **)&device_b, sizeof(int) * n * n); error = cudaMalloc((void **)&device_c, sizeof(int) * n * n); if (error != cudaSuccess) { printf("Fail to cudaMalloc on GPU"); return 1; } //GPU parallel start gettimeofday(&start, 0); cudaMemcpy(device_a, host_a, sizeof(int) * n * n, cudaMemcpyHostToDevice); cudaMemcpy(device_b, host_b, sizeof(int) * n * n, cudaMemcpyHostToDevice); double num = ceil(pow((double)n,2) / pow((double)BLOCKSIZE, 2)); int gridsize = (int)ceil(sqrt(num)); dim3 dimBlock(BLOCKSIZE, BLOCKSIZE, 1); dim3 dimGrid(gridsize, gridsize, 1); multiply<<<dimGrid, dimBlock>>>(device_a, device_b, device_c, n); cudaThreadSynchronize(); cudaMemcpy(host_c, device_c, sizeof(int) * n * n, cudaMemcpyDeviceToHost); gettimeofday(&finish, 0); double t1 = 1000000 * (finish.tv_sec - start.tv_sec) + finish.tv_usec - start.tv_usec; printf("%lf ms\n", t1 / 1000); //GPU parallel finish //CPU serial start gettimeofday(&start, 0); for (i = 0; i < n; i++) { for (j = 0; j < n; j++) { host_c2[i * n + j] = 0; ```

	`` for (k = 0; k < n; k++) { host_c2[i * n + j] += host_a[i * n + k] * host_b[k * n + j]; } } } gettimeofday(&finish, 0); double t2 = 1000000 * (finish.tv_sec - start.tv_sec) + finish.tv_usec - start.tv_usec; printf("%lf ms\n", t2 / 1000); //CPU serial start printf("加速比：%lf\n", t2 / t1); //check int errorNum = 0; for (int i = 0; i < n * n; i++) { if (host_c[i] != host_c2[i]) { errorNum ++; printf("Error occurs at index: %d: c = %d, c2 = %d\n", i, host_c[i], host_c2[i]); } } if (errorNum == 0) { printf("Successfully run on GPU and CPU!\n"); } else { printf("%d error(s) occurs!\n", errorNum); } free(host_a); free(host_b); free(host_c); free(host_c2); cudaFree(device_a); cudaFree(device_b); cudaFree(device_c); return 0; } ```