Федеральное государственное бюджетное образовательное учреждение высшего образования

«Уфимский Государственный Авиационный Технический Университет»

Кафедра ВВТиС

Отчет по дисциплине «Суперкомпьютерные технологии» по лабораторной работе №4

Выполнила студентка группы ПРО

Проверил старший преподаватель

Юлдашев А.В.

Уфа 2017

**Задание**

Реализовать сложение на GPU двух векторов с целочисленными элементами. Предусмотреть проверку корректности вычислений, обработку ошибок и замер времени выполнения функции-ядра.

**Практическая часть**

Параметры графического процессора, на котором проводились вычисления:

* Название чипа – nVidia GeForce GTX 750
* Объём видеопамяти – 2 Гбайта
* Количество CUDA-ядер – 640
* Пропускная способность памяти – 86,4 Гб/с
* Частота ядра – 1,02 ГГц
* Архитектура чипа – Maxwell™

Замер времен для различных размерностей N:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Время (мc),  производительность  (гигафлопс) / размерность | 10^5 | 10^6 | 10^7 | 10^8 | MAX (152600000) |
| Время расчета на GPU | 0.02 | 0.19 | 1.76 | 17.69 | 27.35 |
| Производительность | 5 | 5.26 | 5.68 | 5.65 | 5.58 |
| Время расчета и копирования на GPU c UM | 0.06 | 0.25 | 1.87 | 17.97 | 28.73 |
| Время пересылок device to host | 0.19 | 1.26 | 9.38 | 93.92 | 139.03 |
| Время пересылок host to device | 0.21 | 1.78 | 12.74 | 122.7 | 185.94 |

График зависимости времени показывает, что при увеличении размерности время расчета на GPU увеличивается.

График зависимости производительности показывает, что при увеличении размерности производительность возрастает.

**Заключение**

Разработаны простейшие параллельные программы средствами CUDA C на примере задачи параллельного сложения векторов. Используемые элементы CUDA:

- реализация и вызов вычислительного ядра;

- функции выделения памяти на GPU и копирования массивов между CPU и GPU.

Также была изучена зависимость времени расчета на GPU и производительности от размерности векторов.

Листинг разработанной программы:

#define N 150000000

#define M 1024 // THREADS\_PER\_BLOCK

#include<stdio.h>

#include<stdlib.h>

\_\_global\_\_ void add(int \*a, int \*b, int \*c) {

int index = threadIdx.x + blockIdx.x \* blockDim.x;

if (index < N) c[index] = a[index] + b[index];

}

void random\_ints(int \*a, int n)

{

srand(time(NULL));

for(int i=0;i<n;i++)

{

a[i]=rand();

}

}

void check\_results(int \*a,int \*b,int \*c,int n)

{

for(int i=0;i<n;i++)

{

if(!(a[i]+b[i]==c[i]))

{

printf("Error");

break;

}

}

}

int main(void) {

int \*a, \*b, \*c; // host copies of a, b, c

int \*d\_a, \*d\_b, \*d\_c; // device copies of a, b, c

int size = N \* sizeof(int);

printf("N=150000000 \n");

// Alloc space for device copies of a, b, c

cudaMalloc((void \*\*)&d\_a, size);

cudaMalloc((void \*\*)&d\_b, size);

cudaMalloc((void \*\*)&d\_c, size);

// Alloc space for host copies of a, b, c and setup input values

a = (int \*)malloc(size); random\_ints(a, N);

b = (int \*)malloc(size); random\_ints(b, N);

c = (int \*)calloc(N, sizeof(int));

// Copy inputs to device

cudaEvent\_t start1, stop1;

float time = 0;

cudaEventCreate(&start1);

cudaEventCreate(&stop1);

cudaEventRecord(start1 , 0);

cudaMemcpy(d\_a, a, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, b, size, cudaMemcpyHostToDevice);

cudaEventRecord( stop1,0);

cudaEventSynchronize(stop1);

cudaEventElapsedTime(&time, start1, stop1);

printf("Elapsed time : %.2f ms\n", time);

cudaEventDestroy(start1);

cudaEventDestroy (stop1);

cudaEvent\_t start, stop;

time = 0;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start , 0);

add<<<(N + M - 1) / M, M>>>(d\_a, d\_b, d\_c); // Launch add() kernel on GPU with N blocks

cudaEventRecord( stop,0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);

printf("Elapsed time : %.2f ms\n", time);

cudaEventDestroy(start);

cudaEventDestroy (stop);

cudaEvent\_t start2, stop2;

time = 0;

cudaEventCreate(&start2);

cudaEventCreate(&stop2);

cudaEventRecord(start2 , 0);

cudaMemcpy(c, d\_c, size, cudaMemcpyDeviceToHost); // Copy result back to host

cudaEventRecord( stop2,0);

cudaEventSynchronize(stop2);

cudaEventElapsedTime(&time, start2, stop2);

printf("Elapsed time : %.2f ms\n", time);

cudaEventDestroy(start2);

cudaEventDestroy (stop2);

check\_results(a,b,c, N);

cudaDeviceSynchronize();

cudaError\_t error = cudaGetLastError();

if(error != cudaSuccess) {

printf("CUDA error: %s\n", cudaGetErrorString(error));

exit(-1);

}

// Cleanup

free(a); free(b); free(c);

cudaFree(d\_a); cudaFree(d\_b); cudaFree(d\_c);

return 0;

}

Листинг программы с UM:

#define N 100000

#define M 1024 // THREADS\_PER\_BLOCK

#include<stdio.h>

#include<stdlib.h>

\_\_global\_\_ void add(int \*a, int \*b, int \*c) {

int index = threadIdx.x + blockIdx.x \* blockDim.x;

if (index < N) c[index] = a[index] + b[index];

}

void random\_ints(int \*a, int n)

{

srand(time(NULL));

for(int i=0;i<n;i++)

{

a[i]=rand();

}

}

void check\_results(int \*a,int \*b,int \*c,int n)

{

for(int i=0;i<n;i++)

{

if(!(a[i]+b[i]==c[i]))

{

printf("Error");

break;

}

}

printf("Correct \n");

}

int main(void) {

int \*a, \*b, \*c; // host copies of a, b, c

cudaMallocManaged(&a, N);

cudaMallocManaged(&b, N);

cudaMallocManaged(&c, N);

cudaEvent\_t start, stop;

float time = 0;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start , 0);

add<<<(N + M - 1) / M, M>>>(a, b, c); // Launch add() kernel on GPU with N blocks

cudaEventRecord( stop,0);

cudaEventSynchronize(stop);

cudaDeviceSynchronize();

cudaEventElapsedTime(&time, start, stop);

printf("Elapsed time : %.2f ms\n", time);

cudaEventDestroy(start);

cudaEventDestroy (stop);

check\_results(a,b,c, N);

// Cleanup

cudaFree(a); cudaFree(b); cudaFree(c);

return 0;

}