**НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ УКРАИНЫ**

**«КИЕВСКИЙ ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ ИМЕНИ ИГОРЯ СИКОРСКОГО»**

**КАФЕДРА ТЕХНИЧЕСКОЙ КИБЕРНЕТИКИ**

**СОВРЕМЕННЫЕ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ**

Лабораторная работа

Проверил:

Студент группы ЗПИ-ЗП6115

Макиян С.А.

Проверил:

Доцент кафедры ТК

Корнага Я. И.

Киев - 2019

Лабораторная работа №1

“Обработка одномерных массивов и матриц при помощи CUDA”

**Цель работы:** научиться использовать современную технологию параллельного программирования CUDA для обработки одномерных массивов и матриц.

Требуется создать вектор или несколько матриц (векторов) одинаковой размерности 230 заполнить их считанными из текстового файла значениями. Текстовые файлы следует предварительно подготовить, заполнив их случайными числами. Затем реализовать одномерный массив C = B+3\*A–8, используя обработку данных на CUDA. В конце вывести результаты вычислений опять в текстовый файл.

Интерфейс программы – консольное приложение. В самом начале программа спрашивает имя входного и выходного файлов. Затем, после выполнения всех вычислений, печатает общее время работы (в секундах) на экране в виде дробного числа.

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <iostream>

#include <fstream>

#include <time.h>

#include <numeric>

#include <random>

#include <ctime>

#include <chrono>

cudaError\_t addWithCuda(int \*c, const int \*a, const int \*b, unsigned int size);

\_\_global\_\_ void addKernel(int \*c, const int \*a, const int \*b)

{

int i = threadIdx.x;

c[i] = a[i]\*3 + b[i] -8;

}

int main(int argc, char\*\* argv)

{

// generate 2 <<30 random integers

const int CH\_BUF\_LEN = 32\*32;

std::ifstream infileA("file\_a.txt", std::ifstream::binary);

if (!infileA.is\_open())

{

std::cerr << "no file test\_a.txt" << std::endl;

return 1;

}

std::ifstream infileB("file\_b.txt", std::ifstream::binary);

if (!infileB.is\_open())

{

std::cerr << "no file test\_b.txt" << std::endl;

return 2;

}

std::ofstream outfile("new.txt", std::ofstream::binary);

if (!outfile.is\_open())

{

std::cerr << "cannot create new.txt" << std::endl;

return 2;

}

cudaError\_t cudaStatus = cudaSetDevice(0);

if (cudaStatus != cudaSuccess) {

std::cerr<< "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?" <<std::endl;

return 12;

}

auto start = std::chrono::system\_clock::now();

std::time\_t start\_time = std::chrono::system\_clock::to\_time\_t(start);

#pragma warning(suppress : 4996)

std::cout << "start at " << std::ctime(&start\_time) << std::endl;

// get size of file

infileA.ignore(std::numeric\_limits<std::streamsize>::max());

std::streamsize sizeA = infileA.gcount();

infileA.clear(); // Since ignore will have set eof.

infileA.seekg(0, std::ios\_base::beg);

infileB.ignore(std::numeric\_limits<std::streamsize>::max());

std::streamsize sizeB = infileA.gcount();

infileB.clear(); // Since ignore will have set eof.

infileB.seekg(0, std::ios\_base::beg);

if (sizeA != sizeB)

{

std::cerr << "file\_a.txt size not match file\_b.txt size" << std::endl;

return 4;

}

int tmpBufA[CH\_BUF\_LEN] = { 0 };

int tmpBufB[CH\_BUF\_LEN] = { 0 };

int tmpBufC[CH\_BUF\_LEN] = { 0 };

for (int i = 0; i < sizeA; i += CH\_BUF\_LEN \* sizeof(int))

{

infileA.seekg(i);

infileB.seekg(i);

// read content of infile

infileA.read((char\*)tmpBufA, CH\_BUF\_LEN \* sizeof(int));

infileA.read((char\*)tmpBufB, CH\_BUF\_LEN \* sizeof(int));

// write to outfile

// Add vectors in parallel.

cudaStatus = addWithCuda(tmpBufC, tmpBufA , tmpBufC, CH\_BUF\_LEN);

if (cudaStatus != cudaSuccess) {

std::cout << "addWithCuda failed!" << std::endl;;

return 1;

}

outfile.write((char\*)tmpBufC, CH\_BUF\_LEN \* sizeof(int));

}

cudaStatus = cudaDeviceReset();

if (cudaStatus != cudaSuccess) {

std::cerr << "cudaDeviceReset failed!" << std::endl;

return 1;

}

outfile.close();

//infileA.close();

//infileB.close();

auto end = std::chrono::system\_clock::now();

std::time\_t end\_time = std::chrono::system\_clock::to\_time\_t(end);

std::chrono::duration<double> elapsed\_seconds = end - start;

#pragma warning(suppress : 4996)

std::cout << "finished at " << std::ctime(&end\_time) << "elapsed time: " << elapsed\_seconds.count() << "s" << std::endl;

return 0;

}

// Helper function for using CUDA to add vectors in parallel.

cudaError\_t addWithCuda(int \*c, const int \*a, const int \*b, unsigned int size)

{

int \*dev\_a = 0;

int \*dev\_b = 0;

int \*dev\_c = 0;

cudaError\_t cudaStatus;

// Allocate GPU buffers for three vectors (two input, one output) .

cudaStatus = cudaMalloc((void\*\*)&dev\_c, size \* sizeof(int));

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

cudaStatus = cudaMalloc((void\*\*)&dev\_a, size \* sizeof(int));

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

cudaStatus = cudaMalloc((void\*\*)&dev\_b, size \* sizeof(int));

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

// Copy input vectors from host memory to GPU buffers.

cudaStatus = cudaMemcpy(dev\_a, a, size \* sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

cudaStatus = cudaMemcpy(dev\_b, b, size \* sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

addKernel<<<256, size>>>(dev\_c, dev\_a, dev\_b);

// Check for any errors launching the kernel

cudaStatus = cudaGetLastError();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));

goto Error;

}

// cudaDeviceSynchronize waits for the kernel to finish, and returns

// any errors encountered during the launch.

cudaStatus = cudaDeviceSynchronize();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaStatus);

goto Error;

}

// Copy output vector from GPU buffer to host memory.

cudaStatus = cudaMemcpy(c, dev\_c, size \* sizeof(int), cudaMemcpyDeviceToHost);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

Error:

cudaFree(dev\_c);

cudaFree(dev\_a);

cudaFree(dev\_b);

return cudaStatus;

}

Код программы для генерации случайных значений и записи их в файл:

// lab1\_generator.cpp : Defines the entry point for the console application.

//

#include "stdafx.h"

#include <stdio.h>

#include <iostream>

#include <fstream>

#include <time.h>

#include <numeric>

#include <random>

#include <chrono>

#include <ctime>

int main(int argc, char\*\* argv)

{

// generate 2 <<30 random integers

const int CH\_BUF\_LEN = 32\*4;

int totalBuffSize = 1 << 30;

int tmpBuf[CH\_BUF\_LEN] = { 0 };

std::cout << "start gen" << std::endl;

auto start = std::chrono::system\_clock::now();

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(0, std::numeric\_limits<int>::max() / 4);

std::ofstream outfile("file\_a.txt", std::ofstream::binary);

if (!outfile.is\_open())

{

return -1;

}

for (int i = 0; i < totalBuffSize; i += CH\_BUF\_LEN)

{

for (int j = 0; j < CH\_BUF\_LEN; ++j)

{

tmpBuf[j] = dis(gen);

}

size\_t tt = sizeof(tmpBuf);

outfile.write((char\*)(&tmpBuf), tt);

}

outfile.close();

auto end = std::chrono::system\_clock::now();

std::chrono::duration<double> elapsed\_seconds = end - start;

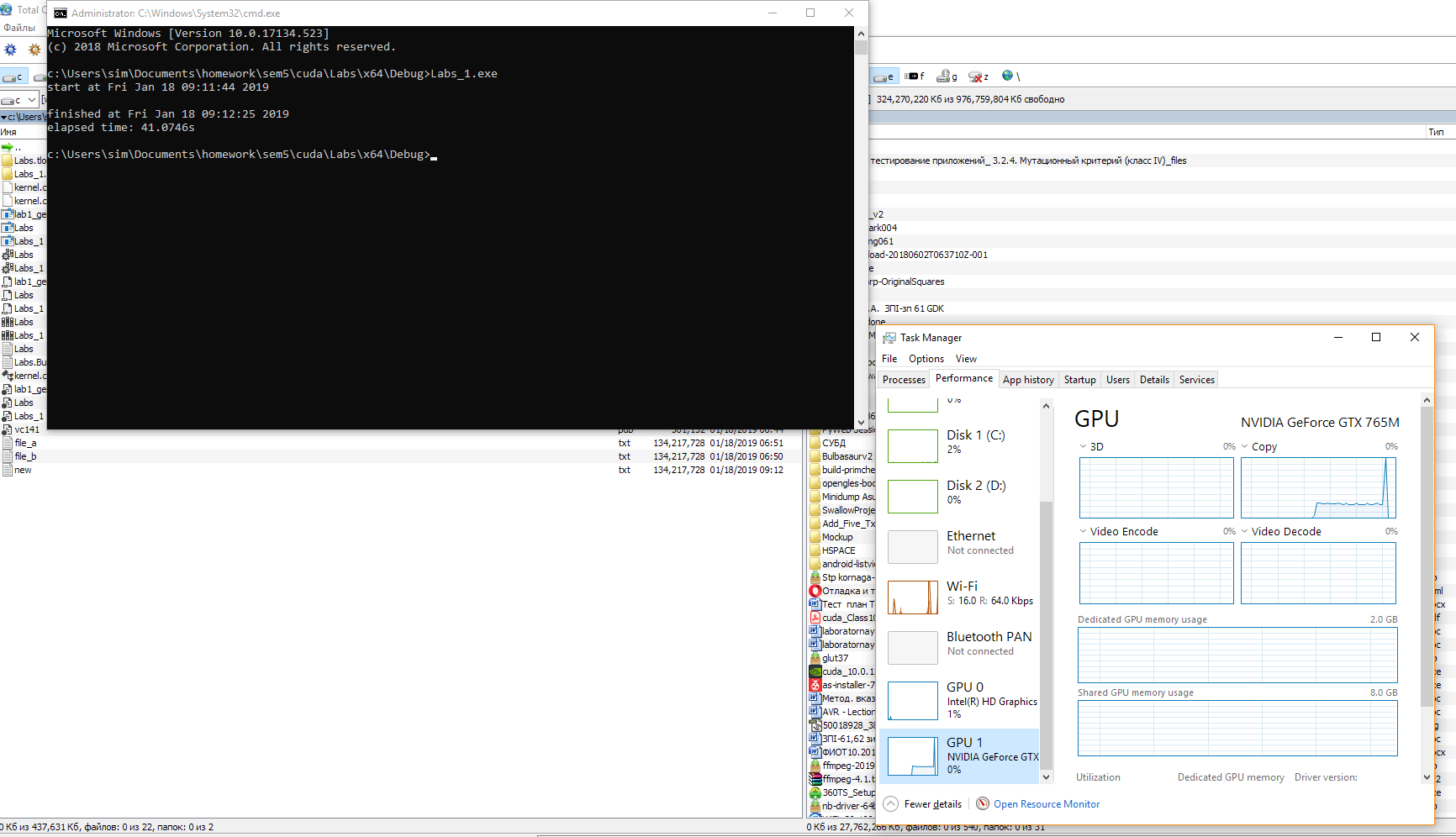
std::time\_t end\_time = std::chrono::system\_clock::to\_time\_t(end);

#pragma warning(suppress : 4996)

std::cout << "finished at " << std::ctime(&end\_time) << "elapsed time: " << elapsed\_seconds.count() << "s" << std::endl;

return 0;

}



Вывод:

В результате работы программы мы получили результирующий файл, сформированный из результата работы программе, с данными соотвествующими заданной формуле.

Расчеты на графичсеком адаптере с поддержкой технологии CUDA работают быстрее чем на процессоре, а счет распараллеливания задач, однако вследствие большого количества ограничений, накладываемых таким подходам, не являются универсальным методом, и подходят только под определнный круг задач.

Лабораторная работа №2

“Визуализация анимированных графиков функций при помощи CUDA”

**Цель работы:** научиться использовать современную технологию параллельного программирования CUDA для обработки графики и создания анимаций.

Cоздать приложение, которое строит анимацию для графика бесконечной периодической функции. При этом по оси X откладывается время, по оси Y – значения функции. Предполагается, что после запуска проекта на экране будет показано новое окно, в котором и отображается анимация. Управление анимацией осуществляет класс CPUAnimBitmap, который будет вызывать пользовательскую функцию всякий раз, когда потребуется сгенерировать новый кадр анимации.

В свою очередь, пользовательскую функция запускает вычислительный процесс на CUDA, результатом которого является матрица цветов для каждого пикселя в текущем кадре анимации. Размер матрицы должен соответствовать размеру окна в пикселях. Каждый кадр анимации рассчитывается на основании текущего номера кадра.

| Функция | Размеры окна | Размеры графика | Скорость анимации |
| --- | --- | --- | --- |
|  | 360x270 | -1..1, 5 | 0.2 |

Текст программы:

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <iostream>

#include "cpu\_anim.h"

#define DIMX 360\*2

#define DIMY 270\*2

#define W DIMX //i have a large display so mul2

#define H DIMY

#define MAXX 3.1415628\*2 // Масштаб по оси X - 2\*PI

#define MAXY 1,5

#define MINY -1

#define DT 0.2

\_\_global\_\_ void kernel(unsigned char \*ptr, int ticks)

{

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

int y = threadIdx.y + blockIdx.y \* blockDim.y;

int offset = x + y \* blockDim.x \* gridDim.x;

float c0, c1, c2;

c0 = abs(H / (MAXY - MINY)\*(4\*cos(tan((x + 0.0) / W \* (MAXX + ticks \* DT))) - MINY) - y);

c1 = abs(H / (MAXY - MINY)\*(4\*cos(tan((x + 0.5) / W \* (MAXX + ticks \* DT))) - MINY) - y);

c2 = abs(H / (MAXY - MINY)\*(4\*cos(tan((x - 0.5) / W \* (MAXX + ticks \* DT))) - MINY) - y);

if (c0 <= 1 || c1 <= 1 || c2 <= 1)

ptr[offset \* 4 + 1] = ptr[offset \* 4 + 2] = 0;

else

ptr[offset \* 4 + 1] = ptr[offset \* 4 + 2] = 255;

ptr[offset \* 4 + 0] = 255;

ptr[offset \* 4 + 3] = 255;

}

struct DataBlock

{

unsigned char \*dev\_bitmap;

CPUAnimBitmap \*bitmap;

};

// Освободить выделенную память устройства

void cleanup(DataBlock \*d)

{

cudaFree(d->dev\_bitmap);

}

void generate\_frame(DataBlock \*d, int ticks)

{

dim3 blocks(DIMX / 16, DIMY / 16);

dim3 threads(16, 16);

kernel <<<blocks, threads >>>(d->dev\_bitmap, ticks);

cudaMemcpy(d->bitmap->get\_ptr(),

d->dev\_bitmap,

d->bitmap->image\_size(),

cudaMemcpyDeviceToHost

);

}

int main(void)

{

int count = 0;

cudaGetDeviceCount(&count);

if (!count)

{

std::cerr << "Not enough cuda devices =(" << std::endl;

return 1;

}

cudaDeviceProp prop;

cudaGetDeviceProperties(&prop, 0);

DataBlock data;

CPUAnimBitmap bitmap(DIMX, DIMY, &data);

data.bitmap = &bitmap;

cudaMalloc((void\*\*)&data.dev\_bitmap,

bitmap.image\_size()

);

bitmap.anim\_and\_exit((void(\*)(void\*, int))generate\_frame,

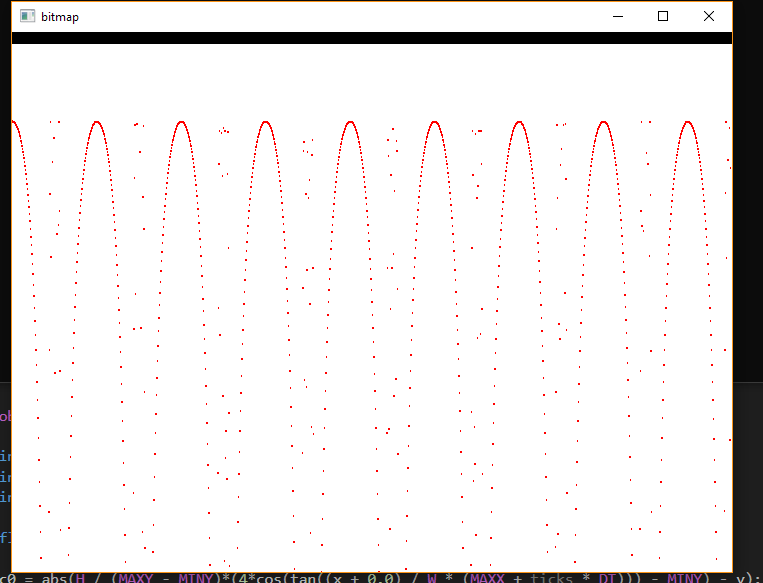
(void(\*)(void\*))cleanup

);

return 0;

}

Скриншот программы.



**Вывод.**

Программа отработала корректно, в результате было получено окно, с графиком заданной функции. Вследствие растягивания окна, появились разрывы между расчетными точками,