ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

**БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ**

**ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

(НИУ «БелГУ»)

ИНСТИТУТ ИНЖЕНЕРНЫХ И ЦИФРОВЫХ ТЕХНОЛОГИЙ

КАФЕДРА МАТЕМАТИЧЕСКОГО И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

**Отчет по лабораторной работе 6**

**по дисциплине: «Модели параллельного программирования»**

**Тема работы «Программирование графических процессоров (NVIDIA CUDA)»**

студента очного отделения

2 курса 12001801 группы

Капустина Виктора Сергеевича

Проверил(а):

Петров Денис Васильевич

Белгород 2020

**Цель:** Ознакомиться с технологией NVIDIA CUDA. Научиться компилировать и запускать программы, содержащие CUDA- код.

**Индивидуальное задание 17 вариант** Необходимо написать программу согласно варианту, при этом реализовать 2 функции: одну для выполнения на процессоре, вторую для выполнения на видеокарте. Затем сравнить результаты (возвращаемые значения) и скорость работы.

Разработайте программу, для вычисления значения функции f(x) на отрезке [1,N+1] с шагом h=N/ k, где N – номер варианта и составьте таблицу:  
  
 **Теоретическая часть**

Модель программирования в CUDA предполагает группирование потоков. Потоки объединяются в блоки потоков (thread block) — одномерные или двумерные сетки потоков, взаимодействующих между собой при помощи разделяемой памяти и точек синхронизации. Программа (ядро, kernel) исполняется над сеткой (grid) блоков потоков (thread blocks). Одновременно исполняется одна сетка. Каждый блок может быть одно-, двух- или трехмерным по форме, и может состоять из 512 потоков на текущем аппаратном обеспечении.

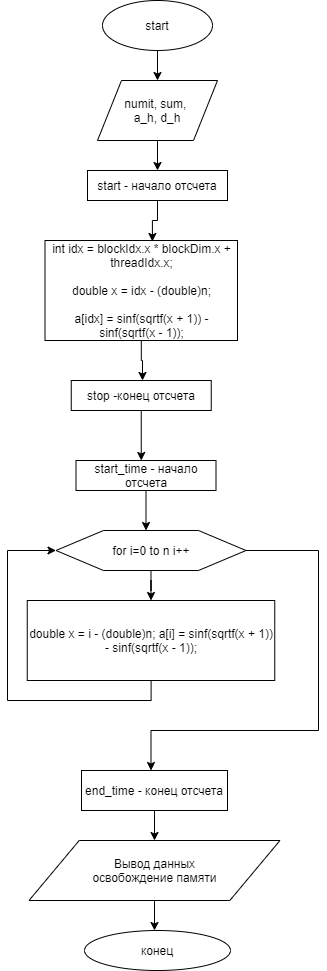
Использующая технологию CUDA программа имеет два типа функций: функции host-кода - набор инструкций, который выполняет центральный процессор; и функций device-кода - набор инструкций, который выполняет процессором графического адаптера. Фактически, технология позволяет программисту включать в программу специальные функции на C подобном языке, которые при запуске программы будут выполнены графическим процессором.

Общий алгоритм работы на СUDA:  
1. Получить данные для расчетов.  
 2. Скопировать эти данные в GPU память.  
 3. Произвести вычисление в GPU через функцию ядра.  
 4. Скопировать вычисленные данные из GPU памяти в ОЗУ.   
5. Посмотреть результаты.  
 6. Высвободить используемые ресурсы.

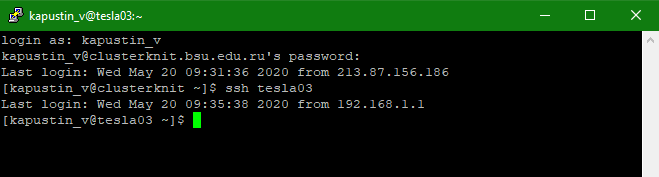
в CUDA требуется не просто выделить локальную память, но также сделать это и на видеокарте.Так с помощью функции cudaMalloc(). после этого можно перенести данные из оперативной памяти в память видеокарты функцией cudaMemcpy() c указанием размеров копируемых данных и место копирования DeviceToHost или HostToDevice. Далее, если необходимо, задаем количество блоков и количество потоков в одном блоке с помощью функций dim3 grid() и dim3 threads() соответственно. Далее вызываем функцию ядра устройства, которая была объявлена как global с указанием на переменные, содержащие количество блоков и потоков в блоке. Теперь остается скопировать результат расчета из видеопамяти в память хоста. Но у функций ядра при этом есть особенность – асинхронное исполнение, то есть, если после вызова ядра начал работать следующий участок кода, то это ещё не значит, что GPU выполнил расчеты. Для завершения работы заданной функции ядра необходимо использовать средства синхронизации. Для непосредственной синхронизации потоков существует функция \_\_syncthreads(), которая ждет, пока все запущенные потоки отработают до этой точки. Она необходима, когда данные, обрабатываемые одним потоком, затем используются другими потоками. Либо существуют так называемые event. Которые объявляются с помощью определенного типа cudaEvent\_t и функций cudaCreateEvent(). Функция cudaEventRecord служит для задания места, прохождение которого должен сигнализировать данный event. Если параметр stream не равен нулю, то отслеживается только завершение выполнения всех операций в данном потоке.

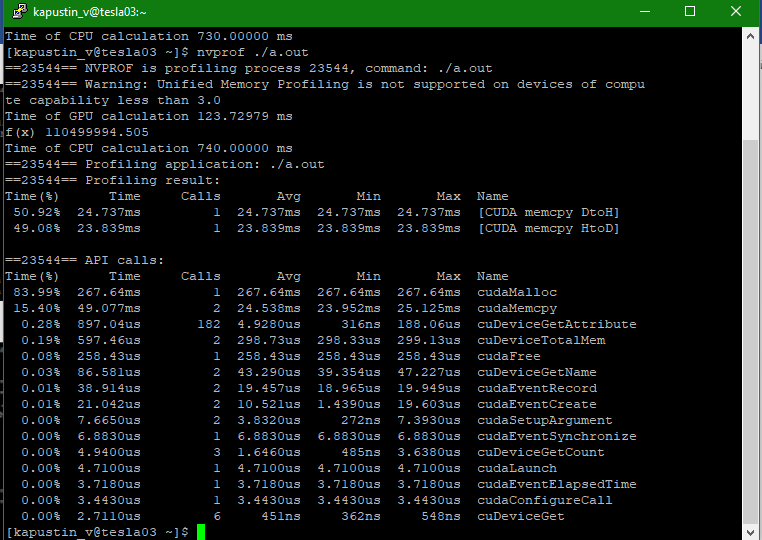
**Задание №1**

****

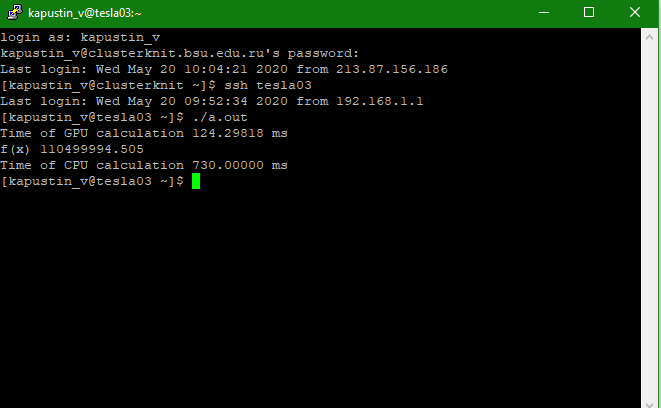


Расчеты по вычислению функции будем производить на putty.exe.

  
**Скриншот 1. Подключение к tesla03**



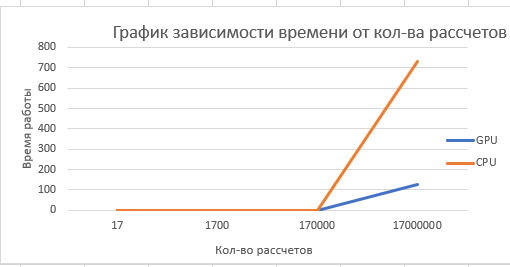
**Скриншот 2. Запуск программы в тестовом режиме**

****

**Скриншот 3. пример работы программы**

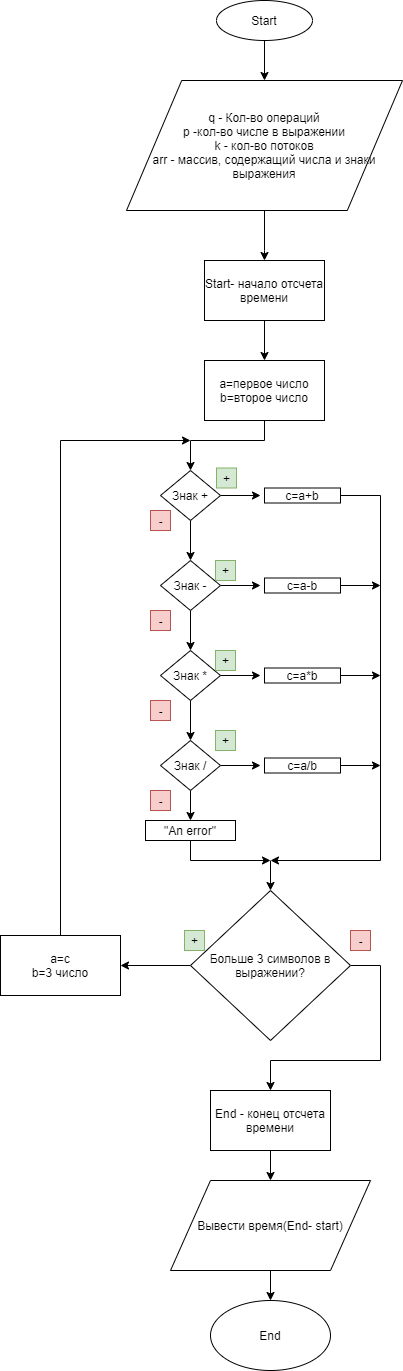
**Таблица №1. Время расчета значений функции.**

| № | P – количество  Расчетов | Время расчета на CPU | Время расчета на GPU | Сумма вычислений: |
| --- | --- | --- | --- | --- |
| 1 | 17 | 0 | 0.08411 | 0 |
| 2 | 1700 | 0 | 0.11181 | 0 |
| 3 | 170000 | 0 | 0.16016 | 0 |
| 4 | 17000000 | 730 | 124.29878 | 110499994.505 |

  
**График 1. График зависимости времени от количества расчетов**

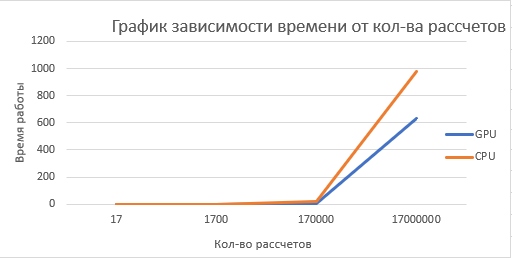
**Задание 2**

**17 вариант.** Дана последовательность арифметических выражений, операндами которых являются однозначные числа, а число операций не больше двух. Найти значения всех выражений.

**Блок-схема  
**

**Таблица №2 Время расчета значений**

| № | P – Количество расчетов | Время расчета на CPU | Время расчета на GPU |
| --- | --- | --- | --- |
| 1 | 17 | 0 | 0,13658 |
| 2 | 1700 | 0 | 0,24262 |
| 3 | 170000 | 20 | 6,6601 |
| 4 | 17000000 | 980 | 632,54 |

**  
  
График 2. График зависимости времени от количества расчетов**

**Листинг 1**

#include <stdio.h>

#include <locale.h>

#include <stdlib.h>

#include <cuda\_runtime.h>

#include <cuda.h>

#include <ctime>

#include <cmath>

\_\_global\_\_ void cGPU(double\* a, long long n) {

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

double x = idx - (double)n;

a[idx] = sinf(sqrtf(x + 1)) - sinf(sqrtf(x - 1));

}

\_\_host\_\_ void cCPU(double\* a, long long n) {

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

double x = i - (double)n;

a[i] = sinf(sqrtf(x + 1)) - sinf(sqrtf(x - 1));

}

}

const long long numit = 17000000;

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

{

double\* a\_h;

double\* a\_d;

a\_h = (double\*)malloc(sizeof(double) \* numit);

cudaMalloc((void \*\*)&a\_d, sizeof(double) \* numit);

int blockSize = 4;

int blocks = numit / blockSize + (numit % blockSize == 0 ? 0 : 1);

double left = 1.0;

double right = 11.0 + 1.0;

double step = (right - left) / (double)numit;

int i = 0;

for (double x = left; x < right; x += step, i++)

{

a\_h[i] = x;

}

cudaEvent\_t start, stop;

float gpuTime = 0.0f;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

cudaMemcpy(a\_d, a\_h, sizeof(double) \* numit, cudaMemcpyHostToDevice);

cGPU <<< blocks, blockSize >>> (a\_d, numit);

cudaMemcpy(a\_h, a\_d, sizeof(double) \* numit, cudaMemcpyDeviceToHost);

double sum = 0;

for (int i = 0; i < numit; i++) {

sum += a\_h[i];

}

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Time of GPU calculation %.5f ms\n", gpuTime);

printf("f(x) %.3f \n", sum);

free(a\_h);

cudaFree(a\_d);

i = 0;

sum = 0.0;

double\* a\_hcpu = new double[numit];

for (double x = left; x < right; x += step, i++) {

a\_hcpu[i] = x;

}

double start\_time = clock();

cCPU(a\_hcpu, numit);

for (int i = 0; i < numit; i++) {

sum += a\_hcpu[i];

}

double end\_time = clock();

double cpuTime = ((end\_time - start\_time) / CLOCKS\_PER\_SEC) \* 1000;

printf("Time of CPU calculation %.5f ms\n", cpuTime);

return EXIT\_SUCCESS;

}

**Конец листинга 1**

**Листинг 2**

#include <iostream>

#include <iomanip>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <locale.h>

#include <stdlib.h>

#include <cuda.h>

#include <ctime>

#define BLOCK\_DIM 2 //размер субматрицы

int M, K;

using namespace std;

\_\_global\_\_ void matrixAdd(int\* A, int\* B, int\* C, int\* D, int\* E, int\* f, int\* n, int M, int K) {

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

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

int index = col \* M + row;

//сложение на GPU

if (col < M && row < K) {

if (f[index] == 1)

{

switch (D[index])

{

case 1: n[index] = A[index] + B[index]; break;

case 2: n[index] = A[index] - B[index]; break;

case 3: n[index] = A[index] \* B[index]; break;

case 4: n[index] = A[index] / B[index]; break;

}

else {switch (E[index])

{

case 1: n[index] = n[index] + C[index]; break;

case 2: n[index] = n[index] - C[index]; break;

case 3: n[index] = n[index] \* C[index]; break;

case 4: n[index] = n[index] / C[index]; break;

}

}

}

}

}

int main() {

cout << "M: ";

cin >> M;

K = 7;

int\* A = new int[M \* K];

int\* B = new int[M \* K];

int\* C = new int[M \* K];

int\* D = new int[M \* K];

int\* E = new int[M \* K];

int\* f = new int[M \* K];

int\* n = new int[M \* K];

//заполнение матриц

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

for (int j = 0; j < K; j++) {

A[i \* K + j] = rand() % 8 + 1;

B[i \* K + j] = rand() % 8 + 1;

C[i \* K + j] = rand() % 8 + 1;

D[i \* K + j] = rand() % 3 + 1;;

E[i \* K + j] = rand() % 3 + 1;

f[i \* K + j] = rand() % 1 + 1;

n[i \* K + j] = 0;

}

double startTime = clock();

srand(time(NULL));

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

for (int j = 0; j < K; j++) {

if (f[i \* K + j] == 1)

{

switch (D[i \* K + j])

{

case 1: n[i \* K + j] = A[i \* K + j] + B[i \* K + j]; break;

case 2: n[i \* K + j] = A[i \* K + j] - B[i \* K + j]; break;

case 3: n[i \* K + j] = A[i \* K + j] \* B[i \* K + j]; break;

case 4: n[i \* K + j] = A[i \* K + j] / B[i \* K + j]; break;

}

else {switch (E[i \* K + j])

{

case 1: n[i \* K + j] = n[i \* K + j] + C[i \* K + j]; break;

case 2: n[i \* K + j] = n[i \* K + j] - C[i \* K + j]; break;

case 3: n[i \* K + j] = n[i \* K + j] \* C[i \* K + j]; break;

case 4: n[i \* K + j] = n[i \* K + j] / C[i \* K + j]; break;

}

}

}

}

double endTime = clock();

double cpuTime = (endTime - startTime) / CLOCKS\_PER\_SEC \* 1000;

int \*dev\_a, \*dev\_b, \*dev\_c, \*dev\_d, \*dev\_e, \*dev\_f, \*dev\_n ; //указатели на выделяемую память

int size = M \* K \* sizeof(int); //выделяемая память

cudaEvent\_t start, stop;

float gpuTime = 0.0f;

cudaMalloc((void\*\*)&dev\_a, size); //выделение памяти

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

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

cudaMalloc((void\*\*)&dev\_d, size);

cudaMalloc((void\*\*)&dev\_e, size);

cudaMalloc((void\*\*)&dev\_f, size);

cudaMalloc((void\*\*)&dev\_n, size);

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

cudaMemcpy(dev\_a, A, size, cudaMemcpyHostToDevice); //копирование на GPU

cudaMemcpy(dev\_b, B, size, cudaMemcpyHostToDevice);

cudaMemcpy(dev\_c, C, size, cudaMemcpyHostToDevice);

cudaMemcpy(dev\_d, D, size, cudaMemcpyHostToDevice);

cudaMemcpy(dev\_e, E, size, cudaMemcpyHostToDevice);

cudaMemcpy(dev\_f, f, size, cudaMemcpyHostToDevice);

cudaMemcpy(dev\_n, n, size, cudaMemcpyHostToDevice);

dim3 dimBlock(BLOCK\_DIM, BLOCK\_DIM); //число выделенных блоков

dim3 dimGrid((M + dimBlock.x - 1) / dimBlock.x, (K + dimBlock.y - 1) / dimBlock.y);

matrixAdd << <dimGrid, dimBlock >> > (dev\_a, dev\_b, dev\_c, dev\_d, dev\_e, dev\_f, dev\_n, M, K);

cudaDeviceSynchronize();

cudaMemcpy(n, dev\_n, size, cudaMemcpyDeviceToHost);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

cout << "Time on GPU is: " << std::setprecision(5) << gpuTime << " ms" << endl;

cout << "Time on CPU is: " << std::setprecision(5) << cpuTime << " ms" << endl;

cudaFree(dev\_a); //освобождение памяти

cudaFree(dev\_b);

cudaFree(dev\_c);

cudaFree(dev\_d);

cudaFree(dev\_e);

cudaFree(dev\_f);

cudaFree(dev\_n);

return 0;

}

**Конец листинга 2**

**Вывод:** В данной лабораторной работе я ознакомился с технологией CUDA, научился создавать и компилировать программы при помощи этой технологии, также были заполнены соответствующие таблицы и построены графики. При увеличении количества расчетов можно заметить, что скорость работы программы на CPU больше, чем на GPU. Если данных мало, то большую часть общего времени занимают вычисления, а не передача данных. На реальных данных гораздо большего размера время передачи данных занимает большую часть общего времени.