МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Лабораторная работа №1**

**по курсу «Программирование графических процессоров»**

**Изучение технологии CUDA**

Выполнил: Иларионов Д.А.

Группа: М8О-408Б-17

Преподаватели: Крашенинников К.Г.,

Морозов А.Ю.

Москва, 2020

**Условие**

Цель работы: Знакомство и установка ПО для работы с параллельными процессами и работой программ на GPU (CUDA). Реализация одной из простых операций над векторами в памяти графического процессора.

**Вариант: 4. Поэлементное нахождение минимума векторов.**

**Программное и аппаратное обеспечение**

**GPU:**

* Name: GeForce GTX 1060
* Compute capability: 6.1
* Частота видеопроцессора: 1404 – 1670 (Boost) МГц
* Частота памяти: 8000 МГц
* Графическая память: 6144 МБ
* Разделяемая память: отсутствует
* Количество регистров на блок: 65536
* Максимальное количество блоков: (2147483647, 65535, 65535)
* Максимальное количество нитей: (1024, 1024, 64)
* Количество мультипроцессоров: 10

**Сведения о системе:**

* Процессор: Intel Core i7-8750H 2.20GHz x 6
* Оперативная память: 16 ГБ
* SSD: 128 ГБ
* HDD: 1000 ГБ

**Программное обеспечение:**

* OS: Windows 10
* IDE: Visual Studio 2019
* Компилятор: nvcc

**Метод решения**

Заполняем оба массива. На каждый поток идет 1 элемент с каждого массива с одинаковым индексом. Идет сравнение и выбирается тот, который меньше. Данное число записывается в третий массив. Если элементов больше, чем потоков, то поток обрабатывает сразу несколько элементов, которые находятся на расстоянии = grimDim\*blockDim. Если полученное id потока больше, чем номер последнего элемента массивов, то данный элемент не обрабатывается. Поэтому, при большом количестве потоков и время выполнение программы дольше даже на небольших данных.

**Описание программы**

**Файл kernel.cu**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <iostream>

#include <fstream>

#include <string>

#include <algorithm>

using namespace std;

#define FIND\_ERR(call) { gpuAssert((call), \_\_FILE\_\_, \_\_LINE\_\_); }

inline void gpuAssert(cudaError\_t cudaStatus, const char\* file, int line, bool abort = true)

{

if (cudaStatus != cudaSuccess)

{

fprintf(stderr, "ERROR: CUDA failed in %s:%d: %s\n", file, line, cudaGetErrorString(cudaStatus));

exit(0);

}

}

\_\_global\_\_ void findMin(int N, double\* v1, double\* v2) {

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

while (tId < N) {

if (v1[tId] < v2[tId]) {

v2[tId] = v1[tId];

}

tId += blockDim.x \* gridDim.x;

}

}

//cpu

void findMinCPU(int N, double\* v1, double\* v2) {

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

v2[i] = min(v1[i], v2[i]);

}

}

int main()

{

std::ios::sync\_with\_stdio(false);

int N;

cin >> N;

if (N <= 0) {

return 0;

}

double\* vector1 = new double[N];

double\* vector2 = new double[N];

double\* fastV1, \*fastV2;

// Инициализируем память на GPU

FIND\_ERR(cudaMalloc(&fastV1, N \* sizeof(double)));

FIND\_ERR(cudaMalloc(&fastV2, N \* sizeof(double)));

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

cin >> vector1[i];

}

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

cin >> vector2[i];

}

//CPU FUNCT

//findMinCPU(N, vector1, vector2, result);

//Копируем в память GPU

FIND\_ERR(cudaMemcpy(fastV1, vector1, N \* sizeof(double), cudaMemcpyHostToDevice));

FIND\_ERR(cudaMemcpy(fastV2, vector2, N \* sizeof(double), cudaMemcpyHostToDevice));

//запись времени

/\*

cudaEvent\_t start, end;

FIND\_ERR(cudaEventCreate(&start));

FIND\_ERR(cudaEventCreate(&end));

FIND\_ERR(cudaEventRecord(start));

\*/

findMin <<<256, 512>>> (N, fastV1, fastV2); //функция

FIND\_ERR(cudaGetLastError()); //просмотр ошибок

/\*

FIND\_ERR(cudaEventRecord(end));

FIND\_ERR(cudaEventSynchronize(end));

float t;

FIND\_ERR(cudaEventElapsedTime(&t, start, end));

FIND\_ERR(cudaEventDestroy(start));

FIND\_ERR(cudaEventDestroy(end));\*/

//конец записи

//printf("time = %f\n", t);

// Копируем результат с ГПУ

FIND\_ERR(cudaMemcpy(vector2, fastV2, N \* sizeof(double), cudaMemcpyDeviceToHost));

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

printf("%.10e ", vector2[i]);

}

// Освобождение памяти

FIND\_ERR(cudaFree(fastV1));

FIND\_ERR(cudaFree(fastV2));

delete[] vector1;

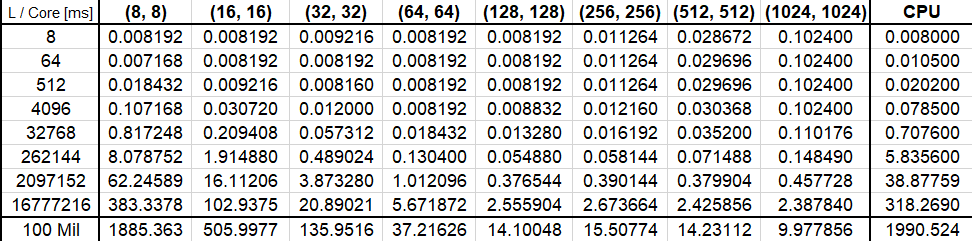
delete[] vector2;

return 0;

}

**Результаты**

Таблица работы программы с разными конфигурациями ядра GPU и с CPU (в миллисекундах)



**Выводы**

Данный алгоритм очень прост, мне не составило особого труда написать программу. Но данная ЛР заключается в другом – познакомиться с работой графического процессора, памятью видеокарты, основами работы в CUDA. Данная задача не сильно подходит для реализации ее на видеопроцессоре, потому что даже при 2 мил. данных не очень много и алгоритм быстрее реализуется на CPU. При работе с GPU тратится лишь дополнительно время на выделение потоков, поэтому и время работы больше в целом. Были глупые ошибки, из-за которых лаба долго не проходила на чекер. Например, в потоковой функции я вместо while почему-то написал int, в итоге, обрабатывались не все блоки. Еще пришлось отключить синхронизацию данных при вводе (std::ios::sync\_with\_stdio(false)), без этой строчки программа не проходила по времени, данный совет дал мне одногруппник. А в целом лаба не показалась мне особо сложной. Очень много времени ушло на замер тестов.