МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Кафедра вычислительной математики и программирования

**спецкурс «Параллельные и распределенные вычисления»**

**ОТЧЕТ**

**Лабораторная работа № 1.1**

**«Освоение программного обеспечения среды программирования NVIDIA»**

Выполнил: Малахов А. В.

Группа: М8О-103М-20

Преподаватель: Семенов С. А.

Москва, 2020

Содержание

[1. Постановка задачи 2](#_Toc52570380)

[2. Описание решения 2](#_Toc52570381)

[3. Аппаратное обеспечение и ПО 2](#_Toc52570382)

[4. Основные моменты кода 2](#_Toc52570383)

[5. Результат работы программы 2](#_Toc52570384)

[6. Сравнение скорости выполнения на CPU и GPU 2](#_Toc52570385)

[7. Выводы 3](#_Toc52570386)

[8. Приложения 3](#_Toc52570387)

# 1. Постановка задачи

# Сложение 1000000 матриц (сравнение производительности с CPU)

# 2. Описание решения

Сложение матриц (сумма матриц) A + B есть операция вычисления матрицы C, все элементы которой равны попарной сумме всех соответствующих элементов матриц A и B, то есть каждый элемент матрицы C равен:

сij = aij + bij

# 3. Аппаратное обеспечение и ПО

Вычисления производились на NVIDIA GeForce GTX 1660 и Intel Core i5 9400f

# 4. Основные моменты кода

Создадим нужное количество матриц через рандом

vector<int\*> matrixs;

for (int i = 0; i < MATRIX\_COUNT; i++) {

printf("%d\n", i);

matrixs.push\_back(new int[M \* K]);

for (int j = 0; j < M \* K; j++)

matrixs[i][j] = rand() % 100 - 50;

}

Так как матрица двумерная, то будем использовать проход по индексу как   
index = col \* row\_size + row

Функция будет принимать две матрицы, которые надо сложить, а также матрицу, в которую будет записан результат. Также параметры M и K будут хранить размерность матрицы.  
Таким образом, функция сложения будет иметь следующий вид:

\_\_global\_\_ void matrixAdd(int\* A, int\* B, 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;

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

B[index] += A[index];

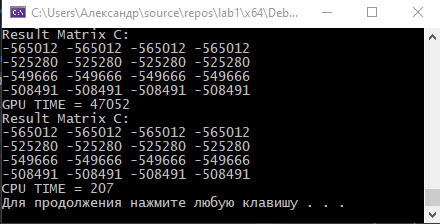
}

}

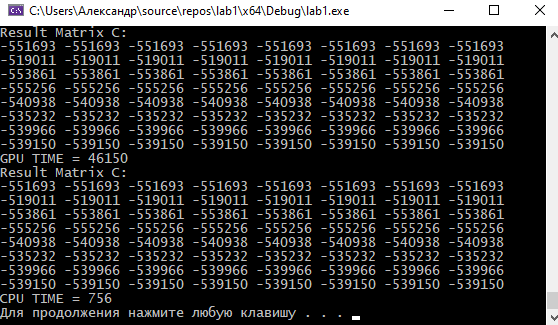
# 5. Результат работы программы

Результат работы программы является вывод результирующей матрицы вычисленной на GPU и на CPU, также выводится затраченное время

Для матрицы 4\*4 результаты следующие:



Для матрицы 8\*8 результаты следующие:



# 6. Сравнение скорости выполнения на CPU и GPU

При запуске программы с различными значениями размерности матрицы видно, что вычисления на видеокарте произвелись медленнее, чем на процессоре компьютера. Это связно с тем, что операция сложения является простой. А вот копирование данных в память GPU оказалось долгим процессом, забирающим большую часть времени выполнения.

Однако стоит отметить, что при увеличении размерности матрицы время вычисления на видеокарте почти не изменилось, в то время как на CPU заметно начало замедляться.

Время выполнения программы при различных значениях *N*, где N – размерность матрицы:

|  |  |  |
| --- | --- | --- |
| **N** | **GPU**  **время выполнения, мс** | **CPU**  **время выполнения, мс** |
| 4\*4 | 47052 | 207 |
| 8\*8 | 46150 | 756 |
| 16\*16 | 41276 | 2848 |
| 32\*32 | 40425 | 11370 |

**Рис.** График зависимости времени выполнения программы от размера матрицы.

# 7. Выводы

В Лабораторной работе №1.1 проведен анализ работы программы по решению задачи Сложение 1000000 матриц (сравнение производительности с CPU). Сложение матриц на GPU оказалось медленнее и, исходя из графика, не имеет смысла на маленьких матрицах 32\*32, но возможно будет иметь смысл при матрицах размерностью больше 100\*100

# 8. Приложения

#include <iostream>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

#include <vector>

#include <cstdlib>

#include <ctime>

#define BLOCK\_DIM 4 //������ ����������+

#define MATRIX\_COUNT 1000000

int M = 32, K = 32;

using namespace std;

\_\_global\_\_ void matrixAdd(int\* A, int\* B, 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;

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

B[index] += A[index];

}

}

void cpu\_test(vector<int\*> matrixs) {

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

int size = M \* K;

for (int i = 0; i < M \* K; i++)

C[i] = 0;

for (int i = 0; i < MATRIX\_COUNT; i++) {

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

C[j] += matrixs[i][j];

}

}

printf("Result Matrix C:\n");

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

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

printf("%d\t", C[i]);

}

printf("\n");

}

free(C);

}

void gpu\_test(vector<int\*> matrixs) {

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

for (int i = 0; i < M \* K; i++)

C[i] = 0;

vector<int\*> cuda\_matrixs;

cuda\_matrixs.resize(MATRIX\_COUNT);

int\* cuda\_matrix\_result;

int size = M \* K \* sizeof(int);

for (int i = 0; i < MATRIX\_COUNT; i++) {

cudaMalloc((void\*\*)&cuda\_matrixs[i], size); //��������� ������

cudaMemcpy(cuda\_matrixs[i], matrixs[i], size, cudaMemcpyHostToDevice);

}

cudaMalloc((void\*\*)&cuda\_matrix\_result, size);

dim3 dimBlock(BLOCK\_DIM, BLOCK\_DIM); //����� ���������� ������

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

for (int i = 0; i < MATRIX\_COUNT; i++) {

matrixAdd << <dimGrid, dimBlock >> > (cuda\_matrixs[i], cuda\_matrix\_result, M, K); //����� ����

}

cudaDeviceSynchronize(); //�����������

cudaMemcpy(C, cuda\_matrix\_result, size, cudaMemcpyDeviceToHost);

//����� ����������

printf("Result Matrix C:\n");

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

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

printf("%d\t", C[i]);

}

printf("\n");

}

for (int i = 0; i < MATRIX\_COUNT; i++) {

cudaFree(cuda\_matrixs[i]);

}

cudaFree(cuda\_matrix\_result);

free(C);

}

int main() {

srand(static\_cast<unsigned int>(time(0)));

vector<int\*> matrixs;

for (int i = 0; i < MATRIX\_COUNT; i++) {

printf("%d\n", i);

matrixs.push\_back(new int[M \* K]);

for (int j = 0; j < M \* K; j++)

matrixs[i][j] = rand() % 100 - 50;

}

unsigned int start\_time1 = clock(); // ��������� �����

gpu\_test(matrixs);

unsigned int end\_time1 = clock(); // �������� �����

unsigned int search\_time1 = end\_time1 - start\_time1; // ������� �����

printf("GPU TIME = %d\n", search\_time1);

unsigned int start\_time2 = clock(); // ��������� �����

cpu\_test(matrixs);

unsigned int end\_time2 = clock(); // �������� �����

unsigned int search\_time2 = end\_time2 - start\_time2; // ������� �����

printf("CPU TIME = %d\n", search\_time2);

for (int i = 0; i < MATRIX\_COUNT; i++) {

free(matrixs[i]);

}

system("pause");

}