МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Кафедра вычислительной математики и программирования

**спецкурс «Параллельные и распределенные вычисления»**

**ОТЧЕТ**

**Лабораторная работа № 1**

**«Программирование CUDA»**

Выполнил: Савельев А.С.

Группа: М8О-107-22

Преподаватель: Семенов С. А.

Москва, 2022

Содержание

[1. Постановка задачи 2](#_Toc7492)

[2. Описание решения 2](#_Toc16227)

[4. Основные моменты кода 3](#_Toc5215)

[5. Результат работы программы 3](#_Toc10579)

[7. Выводы 5](#_Toc8193)

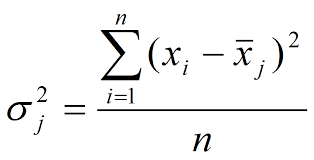
[8. Приложения 5](#_Toc23729)

# 1. Постановка задачи

Вариант 13 (Сложный): Найти среднеквадратичное отклонение.

# Описание решения

Среднеквадратичное отклонение определяется как корень дисперсии. Дисперсией случайной величины называют среднее взвешенное квадратов отклонений реальных (массив *a*) значений от ожидаемых (массив *b*).



Оптимальный алгоритм решения сводится к задаче редукции: частичные суммы рассчитываются на GPU, затем производиться итоговый рассчёт на CPU.

Длинна массива частичных сумм должна быть, с одной стороны достаточно мала, чтобы не перегружать CPU, а с другой - достаточно велика, чтобы должным образом занять его вычислениями. Эмпирически была определена длинна массива в 32 значения. Таким образом максимальное кол-во блоков сетки должно быть меньше или равно 32.

Результатами выполнения программы являются значения массива *reduced* частичных сумм, полученных на GPU, и результирующего среднеквадратичного отклонения *dev*, полученной на CPU.

# Основные моменты кода

Кол-во блоков должно быть меньше или равно 32:

const int threadsPerBlock = 512;

const int blocksPerGrid = imin(32, (N + threadsPerBlock - 1) / threadsPerBlock);

Если кол-во элементов массива превышает общее кол-во нитей, задействованных в вычислениях - осуществляется шаг на gridDim.x \* blockDim.x. И так до тех пор, пока не будут обработаны все элементы массива:

\_\_global\_\_ void calcDeviation(int\* a, int\* b, int\* reduced)

{

\_\_shared\_\_ int cache[threadsPerBlock];

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

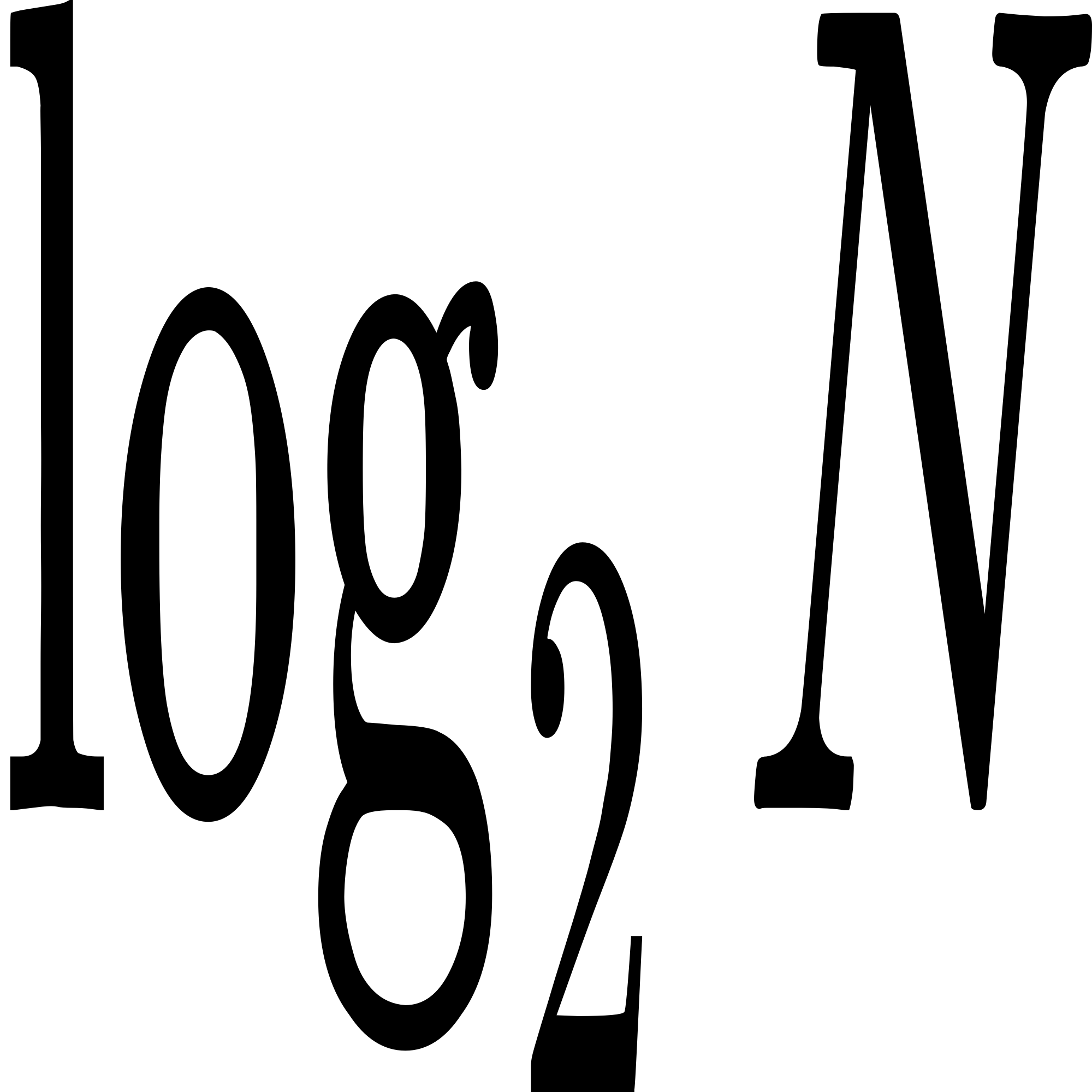
// Вычисление суммы разностей полученных результатов от ожидаемых

for (int i = tid; i < N; i += gridDim.x \* blockDim.x) {

cache[threadIdx.x] += (a[tid] - b[tid]) \* (a[tid] - b[tid]);

}

\_\_syncthreads();

Редукция выполняется за  итераций, результат помещается в нулевой элемент shared массива *cache.* Затем результаты записываются в результирующий массив *reduced* и затем передаются на CPU:

// Непосредственно редукция

for (int i = blockDim.x / 2; (i !=0) && (threadIdx.x < i); i /= 2){

cache[threadIdx.x] += cache[threadIdx.x + i];

\_\_syncthreads();

}

// Запись частичных сумм в массив меньшего размера (Результат выполнения редукции)

if (threadIdx.x == 0){

reduced[blockIdx.x] = cache[0];

}

}

Вычисление результирующего среднеквадратичного отклонения на CPU:

float dev = 0;

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

{

printf("\n Temp sum of %d block is %d", i, reduced[i]);

dev += reduced[i];

}

dev = sqrtf(dev / N);

# 5. Результат работы программы

# 

По правилу формирования значений массивов нетрудно определить корректность работы алгоритма:

// Заполнение входных массивов

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

{

a[i] = i;

b[i] = i + i % 2;

}

7. Выводы

В Лабораторной работе №1 были изучены возможности редукции в контексте задачи по вычислению среднеквадратичного отклонения. Рост производительности GPU относительно CPU коррелирует с ростом размера обрабатываемого массива значений.

# Приложения

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <time.h>

#include <cuda.h>

#include <cuda\_runtime\_api.h>

#include <device\_functions.h>

#include <cmath>

//#define N 1024

//#define N 2048

//#define N 4096

#define N 8192

//#define N 16384

//#define N 32768

//#define N 65536

using namespace std;

#define imin(a,b) (a<b?a:b)

const int threadsPerBlock = 512;

const int blocksPerGrid = imin(32, (N + threadsPerBlock - 1) / threadsPerBlock);

\_\_global\_\_ void calcDeviation(int\* a, int\* b, int\* reduced)

{

\_\_shared\_\_ int cache[threadsPerBlock];

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

// Вычисление суммы разностей полученных результатов от ожидаемых

for (int i = tid; i < N; i += gridDim.x \* blockDim.x) {

cache[threadIdx.x] += (a[tid] - b[tid]) \* (a[tid] - b[tid]);

}

\_\_syncthreads();

// Непосредственно редукция

for (int i = blockDim.x / 2; (i != 0) && (threadIdx.x < i); i /= 2) {

cache[threadIdx.x] += cache[threadIdx.x + i];

\_\_syncthreads();

}

// Запись частичных сумм в массив меньшего размера (Результат выполнения редукции)

if (threadIdx.x == 0) {

reduced[blockIdx.x] = cache[0];

}

}

int main()

{

// Инициализация массивов

int a[N], b[N], reduced[blocksPerGrid];

int\* dev\_a, \* dev\_b, \* dev\_reduced;

// Выделение памяти

cudaMalloc((void\*\*)&dev\_a, N \* sizeof(int));

cudaMalloc((void\*\*)&dev\_b, N \* sizeof(int));

cudaMalloc((void\*\*)&dev\_reduced, blocksPerGrid \* sizeof(int));

// Заполнение входных массивов

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

{

a[i] = i;

b[i] = i + i % 2;

}

// Передача данных устройству

cudaMemcpy(dev\_a, a, N \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b, b, N \* sizeof(int), cudaMemcpyHostToDevice);

// Исполнение на устройстве

calcDeviation << < blocksPerGrid, threadsPerBlock >> > (dev\_a, dev\_b, dev\_reduced);

// Передача данных хосту

cudaMemcpy(reduced, dev\_reduced, blocksPerGrid \* sizeof(int), cudaMemcpyDeviceToHost);

// Вывод результирующих значений

printf("\n Count of numbers: %d", N);

printf("\n Blocks per grid: %d", blocksPerGrid);

printf("\n Threads per block: %d", threadsPerBlock);

printf("\n ---");

float dev = 0;

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

{

printf("\n Temp sum of %d block is %d", i, reduced[i]);

dev += reduced[i];

}

dev = sqrtf(dev / N);

printf("\n ---");

printf("\n Deviation = %f\n", dev);

// Освобождение памяти

cudaFree(dev\_a);

cudaFree(dev\_b);

cudaFree(dev\_reduced);

return 0;

}