МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное
образовательное учреждение высшего образования
«Самарский национальный исследовательский университет
имени академика С.П. Королева»
(Самарский университет)

Институт информатики и кибернетики
Кафедра технической кибернетики

ЛАБОРАТОРНАЯ РАБОТА № 4

по курсу
Параллельное программирование

Группа 6409
Студент Д.К. Чернышова
(*подпись*)

Преподаватель,
к.ф.-м.н. В.Д. Зайцев
(*подпись*)

Самара 2023

ЗАДАНИЕ

Произвести запуск программы для поиска суммы векторов, которая использует технологию CUDA, на различном количестве нитей. В ходе анализа работы программы оценить время ее выполнения на различном количестве исполняющих нитей.

Реализовать последовательный вариант программы, оценить время ее выполнения и рассчитать ускорение параллельных программ относительно последовательного варианта.

Таблица 1 – Исходные данные на ЛР № 4

Тип	int
N	4 500 000 (/10/15)
Параметры сетки	[(2048,512);(512,512);(2048,128);(512,128);(2048,32);(512,32)]

ВВЕДЕНИЕ

GPU изначально отвечал только за обработку трехмерной графики, но в настоящее время используются не только для этого, но и для выполнения вычислительных задач.

В отличие от центрального процессора, графический процессор работает на низкой частоте и имеет значительно больше вычислительных элементов. Основная часть GPU состоит из арифметико-логических блоков (ALU), что увеличивает производительность по сравнению с CPU.

CUDA – это кроссплатформенная система, которая позволяет выполнять последовательную часть программы на CPU, а параллельную часть на GPU.

В ходе данной лабораторной работы необходимо произвести запуск программы для поиска суммы векторов, которая использует технологию CUDA, на различном количестве нитей и оценить время ее выполнения на различном количестве исполняющих нитей.

ЭКСПЕРИМЕНТАЛЬНЫЕ ИССЛЕДОВАНИЯ

В ходе исследования времени работы программ здесь и далее проводилось усреднение по 12 запускам.

На рисунке 1 представлен скриншот запуска и работы программы.

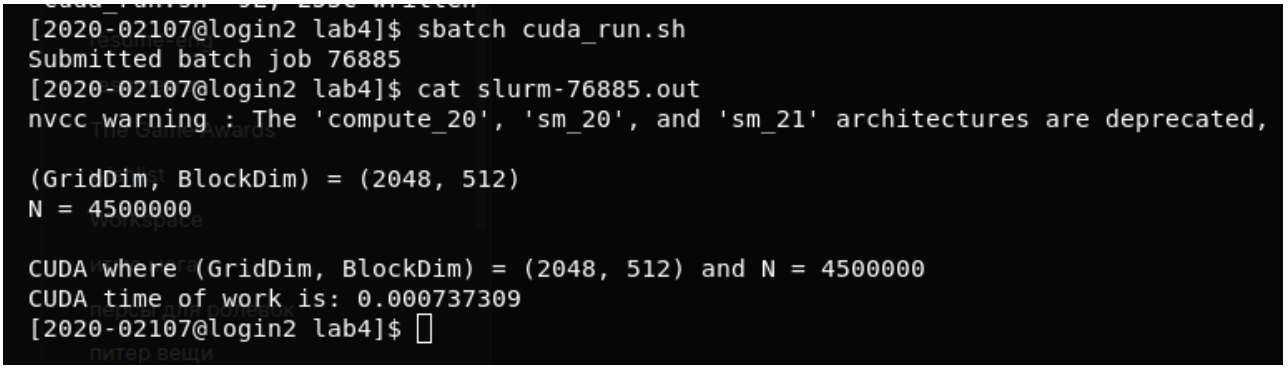

Рисунок 1 – Пример работы программы с технологией CUDA для следующих параметров: N = 4 500 000, [GridDim, BlockDim] = [2048, 512].

Последовательная программа представляла собой заполнение двух массивов случайными значениями, поэлементное суммирование массивов 12 раз для усреднения результатов, суммирование производилось с помощью одного цикла for.

В таблицах 2-3 представлено время выполнения параллельных программ и их ускорение по сравнению с последовательным вариантом.

Таблица 2 – Время работы программ

N		4 500 000	450 000	300 000
Время последовательного, с		0.014167	0.000833	0.000833
Время параллельного на [GridDim, BlockDim], мкс	(2048,512)	0,000737	0,000150	0,000127
	(512,512)	0,000688	0,000097	0,000076
	(2048,128)	0,000692	0,000100	0,000077
	(512,128)	0,000701	0,000089	0,000065
	(2048,32)	0,001633	0,000186	0,000133
	(512,32)	0,001710	0,000186	0,000130

Таблица 3 – Ускорение параллельных программ

N	4 500 000	450 000	300 000

Ускорение параллельного на [GridDim, BlockDim]	(2048,512)	19,222524	5,553333	6,559055
	(512,512)	20,591570	8,587629	10,960526
	(2048,128)	20,472543	8,330000	10,818182
	(512,128)	20,209700	9,359551	12,815385
	(2048,32)	8,675444	4,478495	6,263158
	(512,32)	8,284795	4,478495	6,407692

На рисунке 2 приведен график зависимости времени работы программ от размерности вектора при различных параметрах сетки. На рисунке 3 приведен график зависимости ускорения программ от размерности вектора при различных параметрах сетки.

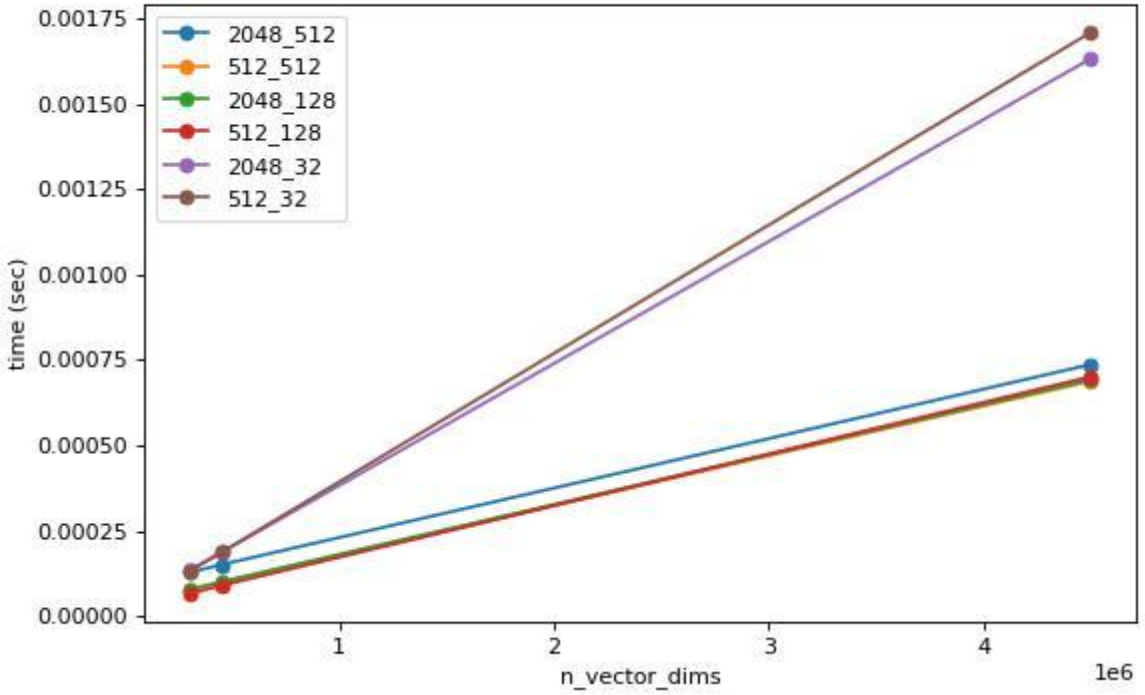

Рисунок 2 – Время работы программ

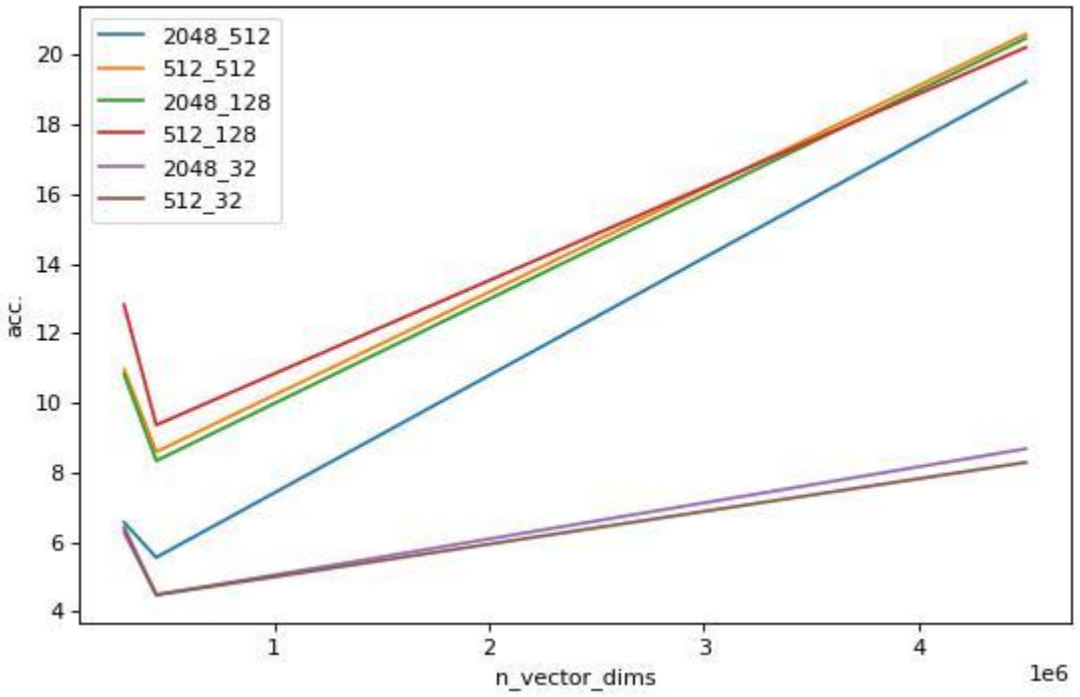

Рисунок 3 – Ускорение программ

ВЫВОДЫ:

Из полученных результатов видно, что:

1. Наибольшее время работы для 4500000 элементов составило 0,00171 секунд, для 450000 элементов – 0,000186 секунд, для 300000 элементов – 0,000133 секунд. Во всех трех случаях работ программ с наибольшим временем работы размер блоков был 32, но если посмотреть на другие значения, то не будет закономерности. Это можно объяснить тем, что нагрузка между нитями распределяется неравномерно, поэтому время изменяется некратно.
2. Наибольшее ускорение было при (512,512) для 4500000 элементов – 20,59157, наибольшее для 450000 элементов – 9,359551, для 300000 элементов – 12,815385. Наименьшее ускорение для 4500000 составило 8,284795, для 450000 элементов наименьшее ускорение было равно 4,478495, а для 300000 элементов равнялось 6,263158.
3. Чем больше размерность задачи, тем больше время выполнения программы, так как с увеличением размерности задачи программе нужно обработать массивы с большей размерностью. Время

выполнения программы с увеличением размерности задачи изменяется примерно кратно. Ускорение кратно не изменяется.

ЗАКЛЮЧЕНИЕ

Цель лабораторной работы – написать параллельную программу поиска суммы двух векторов с использованием технологии CUDA и сравнить время выполнения с длительностью последовательной программы достигнута. Показано, что использование параллельных технологий для данного типа программ обосновано, ввиду того, что ускорение во всех случаях получилось значительным.

В ходе выполнения лабораторной работы я изучила основы CUDA, приобрела навыки по написанию параллельных программ с использованием этой технологии. Наиболее сложной частью выполнения лабораторной работы была подготовка отчета. Интерес вызвало знакомство с новой системой.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Козлова, Е.С. Лабораторные работы по курсу «Параллельное программирование»: Методические указания [Текст] / Сост. Е.С. Козлова, А.С. Широканев − Самара, 2019. – 61 с.

2 Воеводин, В. В. Параллельные вычисления [Текст] / В. В. Воеводин, Вл. В. Воеводин. — СПб.: БХВ-Петербург, 2002. — 608 с.

3 Богачёв К.Ю. Основы параллельного программирования: учебное пособие, 2-е изд. [Текст] / К. Ю. Богачёв - М. : БИНОМ. Лаборатория знаний, 2013. - 344 с.

4 Гергель, В. П. Теория и практика параллельных вычислений, 2-е изд. [Текст] / В. II. Гергель. — М.: Интуит. 2016. - 500 с.

5 Боресков А.В. Параллельные вычисления на GPU. Архитектура и программная модель CUDA Учеб. пособие [Текст] / А.В. Боресков - М.: Издательство Московского университета, 2012. - 336 с.

6 Библиографическое описание документа. Общие требования и правила составления [Электронный ресурс] / сост.: В.С. Крылова, С.М. Григорьевская, Е.Ю. Кичигина // Официальный интернет-сайт научной библиотеки Томского государственного университета. – Электрон. дан. – Томск, [2010]. – http://www.lib.tsu.ru/win/produkzija/metodichka/metodich.html (дата обращения: 10.09.2019).

ПРИЛОЖЕНИЕ А

Код программы с технологией CUDA

```
#include <cublas_v2.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

// Код функции ядра
__global__ void addKernel(int* c, int* a, int* b, unsigned int size) {
    // blockDim – число нитей по x, y, z в блоке
    // gridDim – число блоков по x, y, z в сетке
    // Размер сетки по x
    int gridSize = blockDim.x * gridDim.x;
    int start = blockIdx.x * blockDim.x + threadIdx.x;

    for (int i = start; i < size; i += gridSize) {
        c[i] = a[i] + b[i];
    }
}


int main(int argc, char* argv[]) {
    printf("\n(GridDim, BlockDim) = (%d, %d)\n", GRID_SIZE, BLOCK_SIZE);
    printf("N = %d\n", NMAX);

    int n2b = NMAX * sizeof(int);
    // Выделение памяти на хосте
    int* a = (int*)calloc(NMAX, sizeof(int));
    int* b = (int*)calloc(NMAX, sizeof(int));
    int* c = (int*)calloc(NMAX, sizeof(int));

    // Инициализация массивов
    for (int i = 0; i < NMAX; i++) {
        a[i] = rand();
        b[i] = rand();
    }
```

```
// Выделение памяти на устройстве
int* adev = NULL;
cudaError_t cuerr = cudaMalloc((void**)&adev, n2b);
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot allocate device array for a: %s\n",
        cudaGetErrorString(cuerr));
    return 0;
}

int* bdev = NULL;
cuerr = cudaMalloc((void**)&bdev, n2b);
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot allocate device array for b: %s\n",
        cudaGetErrorString(cuerr));
    return 0;
}

int* cdev = NULL;
cuerr = cudaMalloc((void**)&cdev, n2b);
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot allocate device array for c: %s\n",
        cudaGetErrorString(cuerr));
    return 0;
}

// Создание обработчиков событий
cudaEvent_t start, stop;
float gpuTime = 0.0f;
cuerr = cudaEventCreate(&start);
if (cuerr != cudaSuccess)
{
```

```
    fprintf(stderr, "Cannot create CUDA start event: %s\n",
        cudaGetErrorString(cuerr));
    return 0;
}

cuerr = cudaEventCreate(&stop);
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot create CUDA end event: %s\n",
        cudaGetErrorString(cuerr));
    return 0;
}

// Копирование данных с хоста на девайс
cuerr = cudaMemcpy(adev, a, n2b, cudaMemcpyHostToDevice);
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot copy a array from host to device: %s\n",
        cudaGetErrorString(cuerr));
    return 0;
}

cuerr = cudaMemcpy(bdev, b, n2b, cudaMemcpyHostToDevice);
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot copy b array from host to device: %s\n",
        cudaGetErrorString(cuerr));
    return 0;
}

// Установка точки старта
cuerr = cudaEventRecord(start, 0);
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot record CUDA event: %s\n",
        cudaGetErrorString(cuerr));
```

```
    return 0;
}

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

    //Запуск ядра
    addKernel <<< GRID_SIZE, BLOCK_SIZE >>> (cdev, adev, bdev, NMAX);

    cuerr = cudaGetLastError();
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot launch CUDA kernel: %s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

    // Синхронизация устройств
    cuerr = cudaDeviceSynchronize();
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot synchronize CUDA kernel: %s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }
}

// Установка точки окончания
cuerr = cudaEventRecord(stop, 0);
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot copy c array from device to host: %s\n",
        cudaGetErrorString(cuerr));
    return 0;
}

// Копирование результата на хост
```

```
    cuerr = cudaMemcpy(c, cdev, n2b, cudaMemcpyDeviceToHost);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot copy c array from device to host: %s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

    // Расчет времени
    cuerr = cudaEventElapsedTime(&gpuTime, start, stop);
    printf("\nCUDA where (GridDim, BlockDim) = (%d, %d) and N = %d\n", GRID_SIZE,
BLOCK_SIZE, NMAX);
    printf("CUDA time of work is: %.9f\n", gpuTime / 1000 / ITERATIONS);

    // Очищение памяти
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    cudaFree(adev);
    cudaFree(bdev);
    cudaFree(cdev);
    free(a);
    free(b);
    free(c);



    return 0;
}
```

ПРИЛОЖЕНИЕ Б

Код последовательной программы

```
//NMAX = {9000000, 3000000, 1000000}


#include <stdio.h>
#include <stdlib.h>
#include <time.h>


#define NMAX 9000000
#define ITERATIONS 12


int main(int argc, char* argv[]) {
	int i, q, a[NMAX], b[NMAX], sum[NMAX], j;
	double start_time, end_time;
	//заполнение массива
	for (i = 0; i < NMAX; ++i) {
		a[i] = rand();
		b[i] = rand();
	}
	start_time = clock();
	for (j = 0; j < ITERATIONS; ++j) {
		for (i = 0; i < NMAX; ++i) {
			sum[i] = a[i] + b[i];
		}
	}
	end_time = clock();
	printf("\nSequentialy where N = %d", NMAX);
	printf("\nSequentialy time of work is %f", (end_time - start_time) /
CLOCKS_PER_SEC / ITERATIONS);
	return 0;
}
```

ПРИЛОЖЕНИЕ В

Скрипт для запуска CUDA

```
#!/bin/bash
#SBATCH --job-name=task
#SBATCH --nodes=1
#SBATCH --gres=gpu
#SBATCH --time=00:03:00
module load cuda/8.0
nvcc -g -G -O0 -DGRID_SIZE=128 -DBLOCK_SIZE=1024 -DNMAX=9000000 –DITERATIONS=12
-lcublas -o mainGPU.bin mainGPU.cu
./mainGPU.bin
```

ПРИЛОЖЕНИЕ Г

Скрипт для запуска последовательной программы

```
#!/bin/bash
#SBATCH --job-name=task
#SBATCH --time=00:03:00
#SBATCH --nodes=1
#SBATCH --mem=1gb
./lab4LinearResult
```