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 П.А. Маштаков

(*подпись*)

Преподаватель

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. Рассчитать ускорение параллельных программ относительно последовательного варианта.**

Таблица 1 – Исходные данные на ЛР № 4

|  |  |
| --- | --- |
| Тип | float |
| N | 6300000 (/3/12) |
| Параметры сетки | [(1024, 1024);(256, 1024);(1024, 512);(256, 512);  (1024, 256);(256, 256)] |

ВВЕДЕНИЕ

Программная модель CUDA, разработанная компанией Nvidia в 2006 году, является комплексом аппаратного и программного обеспечения, который упрощает создание программ для графических адаптеров. Nvidia гарантирует, что все её графические карты, независимо от серии, будут поддерживать архитектуру, совместимую с CUDA. Эта технология включает в себя необходимые инструменты для разработки, такие как расширения языка C, компилятор, API для взаимодействия с графическими адаптерами и библиотеки. В API предусмотрены как синхронные, так и асинхронные функции. Синхронные вызовы блокируют выполнение, в то время как асинхронные позволяют выполнять другие задачи параллельно [1]. Асинхронные процессы, включая копирование, запуск ядра и инициализацию памяти, требуют синхронизации перед использованием результатов.

В рамках лабораторной работы проводится сравнение времени, необходимого для сложения элементов векторов, при использовании разного количества потоков.

ЭКСПЕРИМЕНТАЛЬНЫЕ ИССЛЕДОВАНИЯ

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

На рисунке 1 представлен скрин запуска и работы программы.

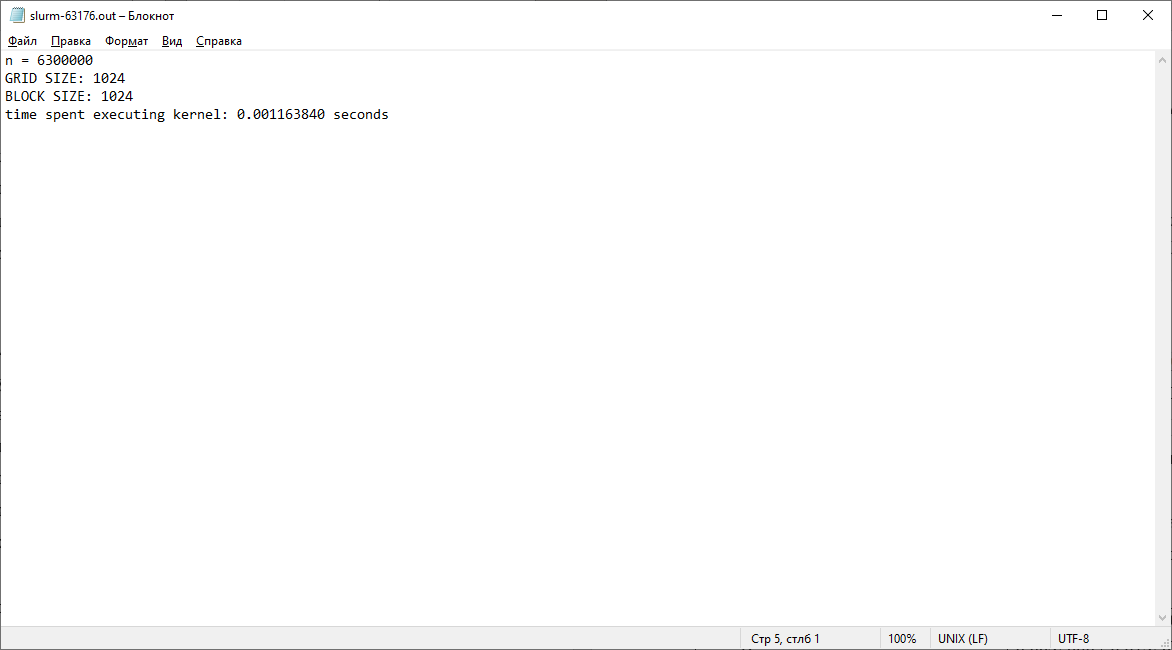


Рисунок 1 – Пример работы программы c технологией CUDA для следующих параметров: N = 6300000, [GridDim, BlockDim] = [1024,1024].

Последовательная программа представляла собой поиск суммы элементов векторов.

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

Таблица 2 – Время работы программ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | | 525000 | 2100000 | 6300000 |
| Время последовательного, с | | 0,0025000 | 0,0125000 | 0,0358330 |
| Время параллельного на [GridDim, BlockDim], с | (1024,1024) | 0,0002094 | 0,0004797 | 0,0011638 |
| (256,1024) | 0,0001428 | 0,0004036 | 0,0010681 |
| (1024,512) | 0,0001330 | 0,0003692 | 0,0009907 |
| (256,512) | 0,0001154 | 0,0003550 | 0,0010008 |
| (1024,256) | 0,0001176 | 0,0003520 | 0,0009764 |
| (256,256) | 0,0001129 | 0,0003743 | 0,0010635 |

Таблица 3 – Ускорение параллельных программ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | | 525000 | 2100000 | 6300000 |
| Ускорение параллельного на [GridDim, BlockDim] | (1024,1024) | 11,9388 | 26,0579 | 30,7896 |
| (256,1024) | 17,5070 | 30,9712 | 33,5483 |
| (1024,512) | 18,7969 | 33,8569 | 36,1693 |
| (256,512) | 21,6637 | 35,2112 | 35,8043 |
| (1024,256) | 21,2585 | 35,5113 | 36,6990 |
| (256,256) | 22,1434 | 33,3956 | 33,6934 |

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.

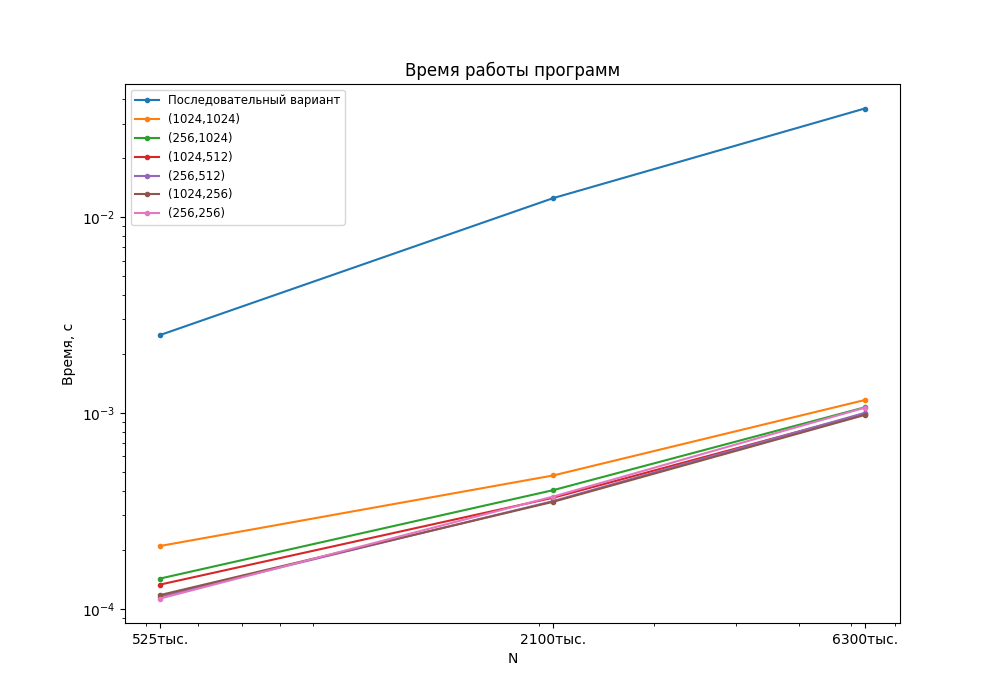


Рисунок 2 – Время работы программ

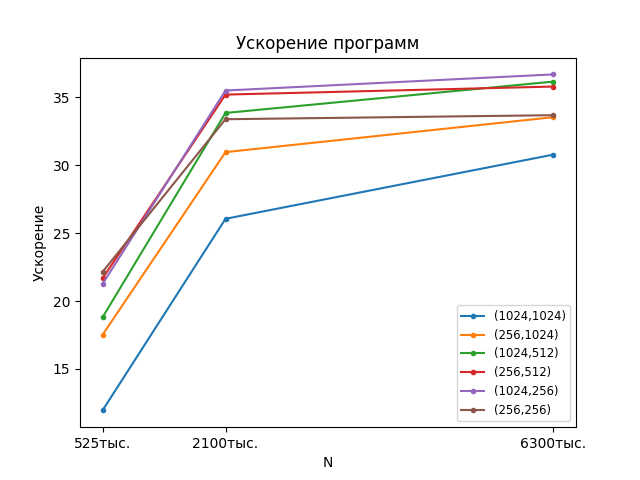


Рисунок 3 – Ускорение программ

**ВЫВОДЫ:**

Из полученных результатов видно, что:

1. Нельзя установить линейную зависимость между параметрами GridDim, BlockDim и ускорением параллельных программ, поскольку для разного количества задач оптимальны индивидуальные параметры.
2. Наибольшее ускорение (36,6990) было получено параллельным вариантом с параметрами GridDim = 1024, BlockDim = 256 и массивом задач, размерностью 6300000, минимальное (30,7896) – параллельным вариантом с параметрами GridDim = 1024, BlockDim = 1024, и массивом задач с размерностью 525000. Это связано с тем, что эффективность программ меняется в зависимости от размера массива, а также размерности грида и блока.
3. С увеличением размерности задачи, увеличивается ускорение программ. Это связано с тем, что с увеличением задач, время, затраченное на управление потоками, составляет все меньшую часть от общего времени работы программы.

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, в виду того, что программы с использованием CUDA дают большое ускорение относительно последовательных программ.

В ходе выполнения лабораторной работы я изучил(а) основы CUDA, приобрел(а) навыки по написанию параллельных программ с использованием этой технологии. Наиболее сложной частью выполнения лабораторной работы был запуск программ на кластере. Интерес вызвала функция ядра, которая выполняет основной алгоритм на графическом ускорителе.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Сапелкин, А. С. CUDA и удалённый GPU / А. С. Сапелкин // Хабр : [сайт]. – Москва, 2018. – URL: https://habr.com/ru/articles/416127/ (дата обращения: 13.11.2023).
2. Козлова, Е.С. Лабораторные работы по курсу «Параллельное программирование»: Методические указания [Текст] / Сост. Е.С. Козлова, А.С. Широканев − Самара, 2019. – 61 с.
3. Воеводин, В. В. Параллельные вычисления [Текст] / В. В. Воеводин, Вл. В. Воеводин. — СПб.: БХВ-Петербург, 2002. — 608 с.
4. Богачёв К.Ю. Основы параллельного программирования: учебное пособие, 2-е изд. [Текст] / К. Ю. Богачёв ‑ М. : БИНОМ. Лаборатория знаний, 2013. ‑ 344 с.
5. Гергель, В. П. Теория и практика параллельных вычислений, 2-е изд. [Текст] / B. II. Гергель. — М.: Интуит. 2016. - 500 с.
6. Боресков А.В. Параллельные вычисления на GPU. Архитектура и программная модель CUDА Учеб. пособие [Текст] / А.В. Боресков ‑ М.: Издательство Московского университета, 2012. - 336 с.
7. 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>

#define BLOCK\_SIZE 1024

#define GRID\_SIZE 1024

\_\_global\_\_ void addKernel(float\* c, float\* a, float\* b, unsigned int size) {

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

while (index < size) {

c[index] = a[index] + b[index];

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

}

}

#define kernel addKernel

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

{

int n = 6300000;

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

int n2b = n \* sizeof(float);

int n2 = n;

float\* a = (float\*)calloc(n2, sizeof(float));

float\* b = (float\*)calloc(n2, sizeof(float));

float\* c = (float\*)calloc(n2, sizeof(float));

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

a[i] = 1.0;

b[i] = 2.0;

}

float\* 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;

}

float\* bdev = NULL;

cuerr = cudaMalloc((void\*\*)&bdev, n2b);

if (cuerr != cudaSuccess) {

fprintf(stderr, "Cannot allocate device array for b: %s\n",

cudaGetErrorString(cuerr));

return 0;

}

float\* cdev = NULL;

cuerr = cudaMalloc((void\*\*)&cdev, n2b);

if (cuerr != cudaSuccess) {

fprintf(stderr, "Cannot allocate device array for c: %s\n",

cudaGetErrorString(cuerr));

return 0;

}

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;

}

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;

}

cuerr = cudaEventRecord(start, 0);

if (cuerr != cudaSuccess) {

fprintf(stderr, "Cannot record CUDA event: %s\n",

cudaGetErrorString(cuerr));

return 0;

}

kernel <<< GRID\_SIZE, BLOCK\_SIZE >>> (cdev, adev, bdev, n);

cuerr = cudaGetLastError();

if (cuerr != cudaSuccess)

{

fprintf(stderr, "Cannot launch CUDA kernel: %s\n",

cudaGetErrorString(cuerr));

return 0;

}

cuerr = cudaDeviceSynchronize();

if (cuerr != cudaSuccess)

{

fprintf(stderr, "Cannot synchronize CUDA kernel: %s\n",

cudaGetErrorString(cuerr));

return 0;

}

cuerr = cudaEventRecord(stop, 0);

if (cuerr != cudaSuccess)

{

fprintf(stderr, "Cannot copy c array from device to host: %s\n",

cudaGetErrorString(cuerr));

return 0;

}

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;

}

cuerr = cudaEventElapsedTime(&gpuTime, start, stop);

printf("time spent executing %s: %.9f seconds\n", "kernel", gpuTime / 1000);

cudaEventDestroy(start);

cudaEventDestroy(stop);

cudaFree(adev);

cudaFree(bdev);

cudaFree(cdev);

free(a);

free(b);

free(c);

return 0;

}

ПРИЛОЖЕНИЕ Б

Код последовательной программы

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SIZE 6300000

#define Q 1

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

int count;

double total\_time = 0, start\_time, end\_time;

float\* a = (float\*)malloc(SIZE \* sizeof(float));

float\* b = (float\*)malloc(SIZE \* sizeof(float));

float\* c = (float\*)malloc(SIZE \* sizeof(float));

int i;

for (i = 0; i < SIZE; i++) {

a[i] = 1.0;

b[i] = 2.0;

}

for (count = 0; count < 12; count++) {

start\_time = clock();

int j = 0;

for (i = 0; i < SIZE; ++i) {

for (j = 0; j < Q; ++j) {

c[i] = a[i] + b[i];

}

}

end\_time = clock();

total\_time += end\_time - start\_time;

}

printf("\nSEQ TIME OF WORK IS %f ", (total\_time / 12) / CLOCKS\_PER\_SEC);

printf("\nSIZE: %d", SIZE);

free(a);

free(b);

free(c);

return 0;

}