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 6408

Студент Д.О. Абрамов

(*подпись*)

Преподаватель,

к.ф.-м.н. Е.С. Козлова

(*подпись*)

Самара 2022

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 | double |
| N | 8 200 000 |
| Параметры сетки | [(4096, 1024); (2048,1024); (4096,256);  (2048, 256); (4096, 64); (2048, 64)] |

ВВЕДЕНИЕ

Первопричиной создания компьютеров была необходимость быстрого проведения вычислительных работ, со временем запросы на данную деятельность только росли и распространялись, что привело к повсеместному использованию компьютерной техники для разного рода задач.

В настоящее время на любом домашнем компьютере стоит многоядерный процессор, который проводит вычисления параллельно. Параллельные вычисления – это использование нескольких или многих вычислительных устройств для одновременного выполнения разных частей одной программы. Основная цель параллельных вычислений – уменьшение времени решения задачи, создание ресурса параллелизма в процессах решения задач с целью достижения наибольшей эффективности использования многопроцессорной вычислительной техники [1].

Параллельные вычисления охватывают вопросы, касающиеся создания ресурсов параллелизма в процессах решения задач и обеспечения его эффективной реализации. Разделяют параллелизм на уровне задач, на уровне данных, алгоритмов. Параллелизм на уровне задач предполагает разбиение задач на подзадачи, которые реализуются одновременно. Параллелизм на уровне данных реализуется компилятором и заключается в замене множества однотипных операций одной операцией. Параллелизм на уровне алгоритмов предполагает замену последовательных алгоритмов некоторых вычислений на параллельные вычисления [2].

На практике часто имеют дело с последовательными программами, где распараллеливание не используется, но с его помощью можно ускорить работу программы. Например, параллельная программа для вычисления приближения к определённому интегралу, смысл в использовании свойства аддитивности интеграла и подсчёта большого количества меньших интегралов. Что приводит к ускорению за счёт параллелизма [3].

Для решения сложных прикладных задач, с большим объёмом вычислений, требуются всё большие вычислительные мощности, такие проблемы и решает создание параллельных вычислительных систем. Сложность подобных задач со временем только растёт, что требует постоянного развития данной отрасли [4].

Можно с уверенностью сказать, что проблемы обработки информации в параллельных вычислениях играют важную роль в современных тенденциях развития информационных технологий, что подтверждается большим количеством современных статей на данную тематику.

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.

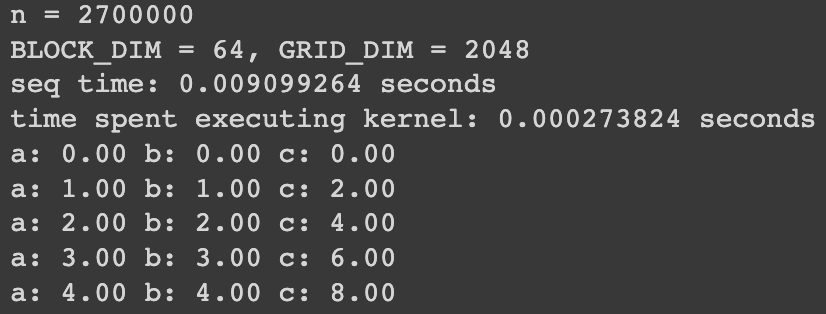


Рисунок 1 – Пример работы программы c технологией CUDA для следующих параметров: N = 2700000, [GridDim, BlockDim] = [2048, 64].

Последовательная программа представляла собой программу для подсчёта суммы двух векторов одним процессом/потоком

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

Таблица 2 – Время работы программ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | | 8200 000 | 2700 000 | 900 000 |
| Время последовательного, мкс | | 27254.367 | 8779.872 | 2945.024 |
| Время параллельного на [GridDim, BlockDim], мкс | 4096, 1024 | 822.656 | 286.128 | 139.072 |
| 2048, 1024 | 804.640 | 284.816 | 107.328 |
| 4096, 256 | 791.104 | 281.584 | 101.312 |
| 2048, 256 | 789.536 | 278.752 | 97.888 |
| 4096, 64 | 788.288 | 275.360 | 95.20 |
| 2048, 64 | 764.192 | 272.032 | 94.72 |

Таблица 3 – Ускорение параллельных программ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | | 8200 000 | 2700 000 | 900 000 |
| Ускорение параллельного на [GridDim, BlockDim] | 4096, 1024 | 33.12972493 | 30.68511994 | 21.17625402 |
| 2048, 1024 | 33.87150402 | 30.82647042 | 27.43947525 |
| 4096, 256 | 34.45105447 | 31.18029433 | 29.06885660 |
| 2048, 256 | 34.51947346 | 31.49707267 | 30.08564890 |
| 4096, 64 | 34.58289810 | 31.88506682 | 30.93512605 |
| 2048, 64 | 35.66429248 | 32.27514410 | 31.09189189 |

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.

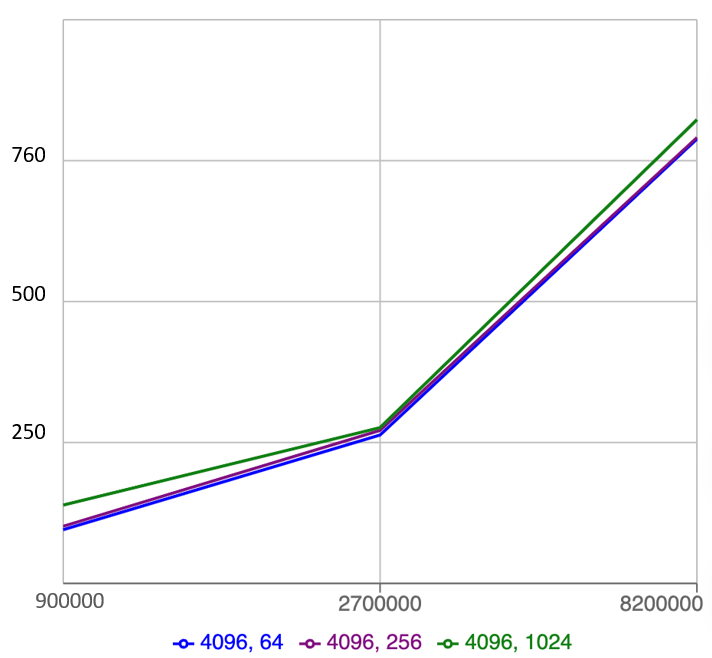


Рисунок 2 – Время работы программ

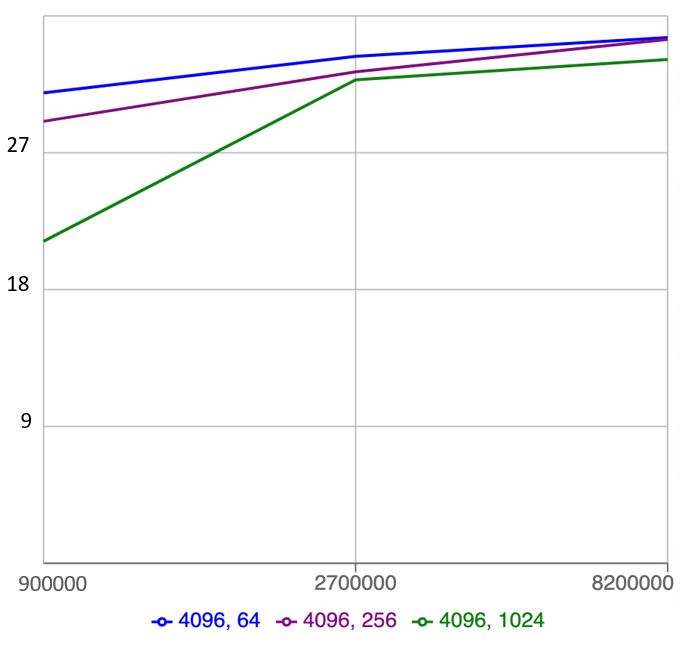


Рисунок 3 – Ускорение программ

**ВЫВОДЫ:**

Из полученных результатов видно, что:

1. Размерность блока и грида для разных N показала различные результаты, слишком большая сетка для малой размерности векторов приводила к уменьшению замедлению программы, что проявлялось в уменьшении темпов ускорении и его падении.
2. Максимальное ускорение программы получилось для размерности [GridDim, BlockDim] = [4096, 1024] и N = 8 200 000. Для меньших размерностей вектора и большой сетки наблюдается меньшее ускорение и увеличение времени работы, что говорит о том, что реализация большой виртуальной топологии для относительно малых задач не целесообразно.
3. Увеличение размерности сетки для N = 8 200 000 приводило к росту ускорения, в случае с N = 900 000 ускорение сначала замедлялось, позже падало. Это происходит из-за временных затрат на реализацию большой виртуальной топологии для относительно малой задачи.

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 использование ресурсов графического процесса для распараллеливания поиска суммы больших векторов даёт большое ускорение относительно последовательной програмы.

В ходе выполнения лабораторной работы я изучил основы CUDA, приобрел навыки по написанию параллельных программ с использованием этой технологии. Наиболее сложной частью выполнения лабораторной работы было реализация функции ядра. Интерес вызвали технология Cuda и зависимость ускорения времени выполнения от размерности блока и грида.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Параллельные вычисления: Учебное пособие [Текст] / Ф.М. Гафаров, А.Ф. Галимянов. – Казань.: Казан. Ун-та, 2018. – 149 с.
2. Параллельные методы и алгоритмы: Учебное пособие [Электронный ресурс] / А.В. Волосова – М.: МАДИ, 2020. – 176 с. – URL.: https://lib.madi.ru/fel/fel1/fel20E533.pdf (дата обращения 18.09.2022).
3. Богачёв К.Ю. Основы параллельного программирования: Учебное пособие, 2-е изд. [Текст] / К. Ю. Богачёв – М.: БИНОМ. Лаборатория знаний, 2013. – 344 с.
4. Козлова, Е.С. Лабораторные работы по курсу «Параллельное программирование»: Методические указания [Текст] / Сост. Е.С. Козлова, А.С. Широканев − Самара, 2019. – 61 с.
5. оеводин, В. В. Параллельные вычисления [Текст] / В. В. Воеводин, Вл. В. Воеводин. — СПб.: БХВ-Петербург, 2002. — 608 с.
6. Богачёв К.Ю. Основы параллельного программирования: учебное пособие, 2-е изд. [Текст] / К. Ю. Богачёв ‑ М. : БИНОМ. Лаборатория знаний, 2013. ‑ 344 с.
7. Гергель, В. П. Теория и практика параллельных вычислений, 2-е изд. [Текст] / B. II. Гергель. — М.: Интуит. 2016. - 500 с.
8. 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.
9. 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

%%cu  
  
#include **<cublas\_v2.h>**#include **<malloc.h>**#include **<stdio.h>**#include **<stdlib.h>** \_\_global\_\_ **void** addKernel(**double**\* c, **double**\* a, **double**\* b, **unsigned int** size) {  
 **for** (**int** i = blockIdx.x \* blockDim.x + threadIdx.x; i < size; i += blockDim.x \* gridDim.x)  
 c[i] = a[i] + b[i];  
}  
  
**int** main(**int** argc, **char**\* argv[])  
{  
 **int** GRID\_DIM = 2048;  
 **int** BLOCK\_DIM = 256;  
 **int** n = 8200000;  
 printf(**"n = %d\n"**, n);  
 printf(**"BLOCK\_DIM = %d, GRID\_DIM = %d\n"**, BLOCK\_DIM, GRID\_DIM);  
 **int** n2b = n \* **sizeof**(**double**);  
  
 **double**\* a = (**double**\*)calloc(n, **sizeof**(**double**));  
 **double**\* b = (**double**\*)calloc(n, **sizeof**(**double**));  
 **double**\* c = (**double**\*)calloc(n, **sizeof**(**double**));  
 **double**\* c\_ = (**double**\*)calloc(n, **sizeof**(**double**));  
  
 **for** (**int** i = 0; i < n; i++) {  
 a[i] = **double**(i);  
 b[i] = **double**(i);  
 }  
  
 cudaEvent\_t start\_p, stop\_p;  
 **float** cpuTime = 0.0f;  
 cudaError\_t cuerr = cudaEventCreate(&start\_p);  
 **if** (cuerr != cudaSuccess) {  
 fprintf(**stderr**, **"Cannot create CUDA start event: %s\n"**,  
 cudaGetErrorString(cuerr));  
 **return** 0;  
 }  
 cuerr = cudaEventCreate(&stop\_p);  
 **if** (cuerr != cudaSuccess) {  
 fprintf(**stderr**, **"Cannot create CUDA end event: %s\n"**,  
 cudaGetErrorString(cuerr));  
 **return** 0;  
 }  
 cuerr = cudaEventRecord(start\_p, 0);  
 **if** (cuerr != cudaSuccess) {  
 fprintf(**stderr**, **"Cannot record start\_p CUDA event: %s\n"**,  
 cudaGetErrorString(cuerr));  
 **return** 0;  
 }  
  
 **for** (**int** i = 0; i < n; i++) {  
 c\_[i] = a[i] + b[i];  
 }  
  
 cuerr = cudaEventRecord(stop\_p, 0);  
 **if** (cuerr != cudaSuccess)  
 {  
 fprintf(**stderr**, **"Cannot record stop\_p CUDA event: %s\n"**,  
 cudaGetErrorString(cuerr));  
 **return** 0;  
 }  
 cuerr = cudaEventElapsedTime(&cpuTime, start\_p, stop\_p);  
 cudaEventDestroy(start\_p);  
 cudaEventDestroy(stop\_p);  
  
  
 **double**\* adev = **NULL**;  
 cuerr = cudaMalloc((**void**\*\*)&adev, n2b);  
 **if** (cuerr != cudaSuccess) {  
 fprintf(**stderr**, **"Cannot allocate device array for a: %s\n"**,  
 cudaGetErrorString(cuerr));  
 **return** 0;  
 }  
 **double**\* bdev = **NULL**;  
 cuerr = cudaMalloc((**void**\*\*)&bdev, n2b);  
 **if** (cuerr != cudaSuccess) {  
 fprintf(**stderr**, **"Cannot allocate device array for b: %s\n"**,  
 cudaGetErrorString(cuerr));  
 **return** 0;  
 }  
 **double**\* 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 start CUDA event: %s\n"**,  
 cudaGetErrorString(cuerr));  
 **return** 0;  
 }  
 addKernel <<< GRID\_DIM, BLOCK\_DIM >>> (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 record stop CUDA event: %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(**"seq time: %.9f seconds\n"**, cpuTime / 1000);  
 printf(**"time spent executing %s: %.9f seconds\n"**, **"kernel"**, gpuTime / 1000);  
 **for** (**int** i = 0; i < 5; i++) {  
 printf(**"a: %.2f b: %.2f c: %.2f\n"**, a[i], b[i], c[i]);  
 }  
 cudaEventDestroy(start);  
 cudaEventDestroy(stop);  
 cudaFree(adev);  
 cudaFree(bdev);  
 cudaFree(cdev);  
 free(a);  
 free(b);  
 free(c);  
 free(c\_);  
 **return** 0;  
}

ПРИЛОЖЕНИЕ Б

Код последовательной программы

*// Подключение необходимых библиотек*#include **<stdio.h>**#include **<stdlib.h>**#include **<time.h>  
  
int** main(**int** argc, **char**\* argv[])  
{  
 **int** vector\_size = 8200000;  
 **int** vector\_bytes = vector\_size \* **sizeof**(**double**);  
 **int** vector\_length = vector\_size;  
 *// Выделение памяти на хосте для векторов* **double**\* vector\_a = (**double**\*)calloc(vector\_length, **sizeof**(**double**));  
 **double**\* vector\_b = (**double**\*)calloc(vector\_length, **sizeof**(**double**));  
 **double**\* vector\_c = (**double**\*)calloc(vector\_length, **sizeof**(**double**));  
  
 *// Инициализация векторов* **for**(**int** i = 0; i < vector\_length; i++){  
 vector\_a[i] = (**double**)(i + 1);  
 vector\_b[i] = (**double**)((i + 1) \* (i + 1));  
 }  
  
 *// Суммирование векторов* clock\_t sum\_time;  
 clock\_t start\_time = clock();  
  
 *// Последовательное суммирование векторов* **for** (**int** k = 0; k < vector\_length; k++) {  
 vector\_c[k] = vector\_a[k] + vector\_b[k];  
 }  
  
 clock\_t end\_time = clock();  
 printf(**"\nSEQUENTIAL: %.4f second(s)\n"**, ((**double**) end\_time - start\_time) / ((**double**) **CLOCKS\_PER\_SEC**));  
  
 *// Вывод первых пяти элементов вектора c* **for**(**int** i = 0; i < 5; i++){  
 printf(**"a: %f + b: %f = c:%f\n"**, vector\_a[i], vector\_b[i], vector\_c[i]);  
 }  
  
 *// Освобождение памяти* free(vector\_a);  
 free(vector\_b);  
 free(vector\_c);  
 **return** 0;  
}