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 6407

Студент Н.В. Носов

(*подпись*)

Преподаватель,

к.ф.-м.н. Е.С. Козлова

(*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 | double |
| N | 3 200 000 (/4/8) |
| Параметры сетки | [(1024,128);(512,128);(1024,64);(512,64);(1024,32);(512,32)] |

ВВЕДЕНИЕ

GPU является программируемым устройством, отвечающим за графический рендеринг. В нынешнее время GPU всё больше применяются не только для рендеринга графики, но и для вычислительных задач.

GPU, в отличие от CPU, работает на низкой тактовой частоте и имеет на порядок больше вычислительных элементов. Также GPU состоит преимущественно из АЛУ, что способствует росту производительности по сравнению с CPU.

CUDA является кроссплатформенной системой, совмещающей в себе работу как на CPU, так и на GPU: вся последовательная часть выполняется на CPU, а параллельная выносится на GPU.

В данной работе представлены результаты времени выполнения программы для поиска суммы векторов при параллельном запуске, на различном количестве исполняющих нитей с использованием технологии CUDA.

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.

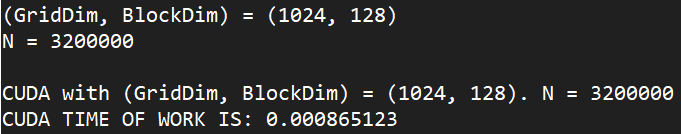


Рисунок 1 – Пример работы программы c технологией CUDA для следующих параметров: N = 3200000, [GridDim, BlockDim] = [1024, 128].

Последовательная программа представляла собой суммирование элементов двух массивов и запись результата в соответствующую ячейку результирующего массива.

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

Таблица 2 – Время работы программ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | | 3 200 000 | 800 000 | 400 000 |
| Время последовательного, с | | 0.012500 | 0.002500 | 0.000833 |
| Время параллельного на [GridDim, BlockDim], с | (1024, 128) | 0.000865123 | 0.000217251 | 0.000112101 |
| (512, 128) | 0.000874688 | 0.000223032 | 0.000115309 |
| (1024, 64) | 0.000897829 | 0.000231397 | 0.000123515 |
| (512, 64) | 0.000900427 | 0.000234085 | 0.000124757 |
| (1024, 32) | 0.001365480 | 0.000350728 | 0.000185101 |
| (512, 32) | 0.001355691 | 0.000350616 | 0.000184240 |

Таблица 3 – Ускорение параллельных программ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | | 3 200 000 | 800 000 | 400 000 |
| Ускорение параллельного на [GridDim, BlockDim] | (1024, 128) | 14.44881 | 57.53713 | 111.5066 |
| (512, 128) | 14.29081 | 56.04577 | 108.4044 |
| (1024, 64) | 13.92247 | 54.01972 | 101.2023 |
| (512, 64) | 13.8823 | 53.39941 | 100.1948 |
| (1024, 32) | 9.15429 | 35.64015 | 67.5307 |
| (512, 32) | 9.22039 | 35.65154 | 67.84629 |

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. Для каждой размерности массива размерности сетки и блока влияют по-разному. Так, например, для 6000000 самой быстрой программой оказалась программа с размерностями (1024, 256) – её время составило 0,000904 секунд. Для 3000000 самой быстрой оказалась программа с (1024, 128) – она выполнилась за 0,000471 секунд. А для 1500000 элементов самой быстрой стала программа с (512, 128) и временем выполнения 0,000248 секунд. Самой медленной стала программа с 6000000 элементов, размерностью (512, 64) – её время выполнения составило 0,001283 секунд. Такая вариативность времени для каждого числа элементов может объясняться тем, что минимальное время работы достигается при оптимальной размерности – когда создаётся такая сетка с такой размерностью блоков, что все нити выполняют приблизительно равное количество операций и это количество стремится к минимуму. Время изменяется некратно в виду того, что нагрузка между нитями распределяется неравномерно.
2. Максимальное ускорение получено для программы при размерности (1024, 128) на 3000000 элементов и составило 22,990482. Минимальное ускорение получено для 6000000 элементов при размерности (512, 64) и составило 8,299998. Это можно объяснить тем, что для 3000000 при данной размерности не создаётся огромное количество лишних нитей и каждая из них выполняет примерно равное количество операций, которое невелико. Для 6000000 элементов ускорение минимальное, так как на общем минимальном количестве нитей обрабатывается максимальное количество элементов – каждая нить выполняет большое количество операций из-за чего происходит замедление.
3. С изменением размерности массивов время выполнения на одинаковых размерностях [GridDim, BlockDim] для трёх разных длин массивов изменяется соответственно кратности. То есть для 6000000 на (1024, 256) получаем 0,000904 секунды, на такой же размерности для 1500000 получаем 0,000250, что примерно в 4 раза меньше, как и количество элементов. Ускорение изменяется примерно кратно только для 6000000 и 3000000 элементов. Это можно объяснить тем, что при 1500000 элементах последовательная программа работает быстрее, чем на предыдущих размерностях массива – 0,004167 против 0,010833. Из-за этого ускорение при 1500000 в ~1,5 раза больше, чем на 6000000 элементов и в ~1,5 раза меньше, чем при 3000000 элементов.
4. Сравнивая эффективность CUDA с работой опции schedule OpenMP и MPI на одинаковом количестве элементов, получаем, что лучшее время на CUDA для 6000000 элементов – 0,000904 секунды при размерности (1024, 256), для OpenMP лучшее время 0,004 секунды для 6 процессов при static, dynamic и guided (см. отчёт по 3 ЛР); у MPI лучшее время – 0,09 секунд для кратного количества элементов (см. отчёт по 3 ЛР). Можно сделать вывод, что CUDA работает быстрее за счёт того, что все вычисления происходят на GPU, где данные обрабатывают, в данном случае, 262144 процесса, в то время когда OpenMP и MPI работают на CPU при 6 процессах.

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 время выполнения программы с технологией CUDA имеет гораздо меньший порядок, чем время выполнения на обычном процессоре.

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 написание функции ядра для суммирования массивов. Интерес вызвало изучение технологии CUDA и написание программы с использованием этой технологии.

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. оеводин, 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] / B. II. Гергель. – М.: Интуит. 2016. - 500 с.
5. 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.
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

// Здесь объяснение почему код ядра именно такой:

// https://stackoverflow.com/questions/16619274/cuda-griddim-and-blockdim/29076010#29076010

\_\_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 with (GridDim, BlockDim) = (%d, %d). 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

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

//NMAX = [6000000, 3000000, 1500000]

#define NMAX 6000000

#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("\nLinear with N = %d", NMAX);

printf("\nLINEAR TIME OF WORK IS %f", (end\_time - start\_time) / CLOCKS\_PER\_SEC / ITERATIONS);

return 0;

}

ПРИЛОЖЕНИЕ В

Скрипт запуска CUDA программы

#!/bin/bash

#SBATCH --job-name=myjob\_gpu

#SBATCH --nodes=1

#SBATCH --gres=gpu

#SBATCH --time=00:03:00

module load cuda/8.0

nvcc -g -G -O0 -DGRID\_SIZE=2048 -DBLOCK\_SIZE=256 -DNMAX=6000000 –DITERATIONS=12 -lcublas -o mainGPU.bin mainGPU.cu

./mainGPU.bin

ПРИЛОЖЕНИЕ Г

Скрипт запуска последовательной программы

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