1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**ЛАБОРАТОРНАЯ РАБОТА № 3**

1. «**Программирование на GPU**»
2. по дисциплине «Языки программирования»
3. Выполнил
4. студент гр. 4851003/00002 Вологдин М.В.

<*подпись*>

1. Преподаватель
2. асс. преподавателя Малышев Е.В.

<*подпись*>

1. Санкт-Петербург
2. 2021
3. **Цель работы**

Практическое ознакомление с фреймворком для написания компьютерных программ, связанных с параллельными вычислениями OpenCL на основе программы, вычисляющей обратную матрицу

1. **Ход работы**

В ходе работы был реализован алгоритм, который начальную матрицу (введённую вручную или же сгенерированную случайно) преобразует в обратную для начальной.

Общий алгоритм построения обратной матрицы:

1. Заменить элементы матрицы их алгебраическими дополнениями
2. Транспонировать матрицу
3. Разделить каждый элемент на определитель исходной матрицы

В качестве алгоритма для рассчёта алгебраических дополнений был выбран метод Гаусса. Матрица приводится с помощью элементарных преобразований к верхней треугольной, и далее её определитель считается как произведение элементов на главной диагонали.

В функции det программа вычисляет определитель матрицы, что пригодится для будущих вычислений. Вычисление также производится с помощью метода Гаусса. Программа выполняет 2 вложенных цикла, по строкам матрицы. Высчитывается коэфициент на который нужно умножить первую строку и сложить со второй, чтобы все до i-ого элемента обнулились. После этого циклом по длине строки первая, умноженная на коэфициент и вторая складываются. После всех операций в переменную res программа записывает итоговый результат перемножения всех диагональных элементов.

Листинг алгоритма представлен в Приложении А.

Рассмотрим программу на GPU. Алгоритм используется аналогичный, за исключением некоторых строк, которые были изменены для корректной работы программы, однако общая суть остаётся такой же.

Для реализации программирования на видеокарте используется библиотека OpenCL.lib и <cl.h>.

С помощью clGetPlatformIDs программа определяет количество доступных платформ и сохраняется их список

С помощью clGetDeviceIDs программа определяет количество доступных устройств, а также ищет устройство типа GPU на нашей платформе

С помощью clCreateContext программа создаёт контекст выполнения нашего алгоритма

С помощью clCreateCommandQueueWithProperties программа создаёт очередь выполняемых команд

Далее с помощью clCreateBuffer программа выделяет память для device, а c помощью clEnqueueWriteBuffer копирует данные с указателей в выделенную память.

После этого программа создаёт программу, которая будет исполняться, с помощью clCreateProgramWithSource и компилирует благодаря clBuildProgram.

Следующий шаг – создание ядра с помощью clCreateKernel, а также определение входных аргументов, которые устанавливаются с помощью clSetKernelArg

С помощью clEnqueueNDRangeKernel программа запускает ядро, и чтобы получить выходные данные программа копирует данные с помощью clEnqueueReadBuffer

После всех операций вся выделенная память освобождается в обратном порядке выделения

Текст программы исполняемой на gpu записывается в массив char

В исходной функции \_\_kernel void obr\_matr программа определяет индекс данного потока int index = get\_global\_id(0);

Далее по этому индексу программа определяет i и j для удобства представления в матричном виде, создаётся копия входного массива A, и текущий поток вычисляет определённый элемент выходного массива B с помощью вышеописанной функции dop. Листинг алгоритма представлен в Приложении Б.

Сложность данного алгоритма – O(n3) так как это тройной вложенный цикл:  
два по количеству строк, а один по длине строки, в случае квадратной матрицы все эти три измерения одинаковы и равны n.

1. **Тестирование**

Таблица 1 – Результаты тестирования программы

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 9 | 16 | 25 | 64 | 100 | 400 | 900 | 1225 | 1600 | 2500 | 6400 | 10000 |
| TGPU,c | 0.505 | 0.54 | 0.568 | 0.612 | 0.626 | 1.371 | 7.135 | 0.965 | 0.785 | 0.834 | 1.66 | 3.17 |
| TCPU,c | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.006 | 0.044 | 0.113 | 0.185 | 0.585 | 5.91 | 91.375 |

Рисунок 1 – График зависимости времени выполнения от количества элементов

1. **Выводы**

В ходе работы были изучены основные функции, использующиеся в фреймворке OpenCL, и их применение. Также была написана программа, вычисляющая обратную матрицу, и было проведено сравнение работы алгоритма на CPU и GPU, которое показало, что при большом количестве потоков, в нашем случае от 10000, программа на gpu является более эффективной, и может выполнять быстрые расчёты для десятков тысяч потоков и более.

**ПРИЛОЖЕНИЕ А**

Листинг программы “CPU”

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#define VECTOR\_SIZE 100

#define SIZE 10

#define RAND

long det(int\* A, int size, int del\_i, int del\_j)

{

float tmp[VECTOR\_SIZE];

memset(tmp, 0, sizeof(tmp));

if (!(del\_i == -1 && del\_j == -1))

{

int z = 0;

for (int k = 0; k < size \* size; k++)

{

if (k / size == del\_i || k % size == del\_j)

continue;

tmp[z] = A[k];

z++;

}

size--;

}

else

memcpy(tmp, A, sizeof(tmp));

float koef;

for (int i = 0; i < size - 1; i++)

{

for (int j = i + 1; j < size; j++)

{

koef = (float)tmp[j \* size + i] / tmp[i \* size + i];

for (int ind = 0; ind < size; ind++)

tmp[j \* size + ind] -= tmp[i \* size + ind] \* koef;

}

/\*for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

printf("%2f ", tmp[j + size \* i]);

printf("\n");

}

printf("\n");\*/

}

float res = 1;

for (int i = size - 1; i >= 0; i--)

res \*= tmp[(size + 1) \* i];

return res;

}

int main()

{

int A[VECTOR\_SIZE];

float B[VECTOR\_SIZE];

memset(B, 0, sizeof(B));

srand(time(0));

for (int i = 0; i < VECTOR\_SIZE; i++)

{

#ifdef RAND

A[i] = 1 + rand() % 10;

#else

//scanf("%f", &A[i]);

A[0] = 2; A[1] = 10; A[2] = 8; A[3] = 7; A[4] = 9; A[5] = 6; A[6] = 7; A[7] = 17; A[8] = 2; A[9] = 3;

A[10] = 15; A[11] = 8; A[12] = 20; A[13] = 19; A[14] = 9; A[15] = 7; A[16] = 2; A[17] = 14; A[18] = 10;

A[19] = 11; A[20] = 5; A[21] = 18; A[22] = 20; A[23] = 8; A[24] = 17;

#endif

}

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

{

for (int j = 0; j < SIZE; j++)

printf("%2ld ", A[j + SIZE \* i]);

printf("\n");

}

printf("\n");

clock\_t start = clock();

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

for (int j = 0; j < SIZE; j++)

{

B[i \* SIZE + j] = det(A, SIZE, i, j);

if ((i + j) % 2 != 0)

B[i \* SIZE + j] = -B[i \* SIZE + j];

}

double timeСPUTotal = (double)(clock() - start) / CLOCKS\_PER\_SEC;

printf("Time: %lf\n", timeСPUTotal);

printf("RESULT:\n");

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

{

for (int j = 0; j < SIZE; j++)

{

printf("%f ", B[i + SIZE \* j]);

}

printf("\n");

}

}

**ПРИЛОЖЕНИЕ Б**

Листинг программы “GPU”

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include <CL/cl.h>

#include <time.h>

#define VECTOR\_SIZE 100

#define SIZE 10

#define RAND

#define MAX\_SOURCE\_SIZE 4096

float det(int\* A, int size);

// 10000 3.17

// 6400 1.66

// 2500 0.834

// 1600 0.785

// 1225 0.965

// 900 7.135

// 400 1.371

// 100 0.626

// 64 0.612

// 25 0.568

// 16 0.54

// 9 0.505

int main(void) {

int len = sqrt(VECTOR\_SIZE);

int size\_small = (SIZE - 1) \* (SIZE - 1);

int i, j;

FILE\* file = fopen("matr.txt", "r");

char\* code = (char\*)malloc(5120);

int len\_file = 0;

int c = fgetc(file);

while (c != EOF)

{

code[len\_file] = c;

len\_file++;

c = fgetc(file);

}

fclose(file);

code[len\_file] = '\0';

int\* A = (int\*)malloc(sizeof(int) \* VECTOR\_SIZE);

int\* B = (int\*)malloc(sizeof(int) \* VECTOR\_SIZE);

srand(time(0));

for (i = 0; i < VECTOR\_SIZE; i++)

{

#ifdef RAND

A[i] = 1 + rand() % 10;

#else

scanf("%d", &A[i]);

#endif

}

clock\_t start = clock();

// Get platform and device information

cl\_platform\_id\* platforms = NULL;

cl\_uint num\_platforms;

//Set up the Platform

cl\_int clStatus = clGetPlatformIDs(0, NULL, &num\_platforms);

platforms = (cl\_platform\_id\*)malloc(sizeof(cl\_platform\_id) \* num\_platforms);

clStatus = clGetPlatformIDs(num\_platforms, platforms, NULL);

char buf[1024];

//clGetPlatformInfo(platforms[0], CL\_PLATFORM\_NAME, sizeof(buf), buf, NULL);

//printf("Platform: %s\n", buf);

//Get the devices list and choose the device

cl\_device\_id\* device\_list = NULL;

cl\_uint num\_devices;

clStatus = clGetDeviceIDs(platforms[0], CL\_DEVICE\_TYPE\_GPU, 0, NULL, &num\_devices);

device\_list = (cl\_device\_id\*)malloc(sizeof(cl\_device\_id) \* num\_devices);

clStatus = clGetDeviceIDs(platforms[0], CL\_DEVICE\_TYPE\_GPU, num\_devices, device\_list, NULL);

//clGetDeviceInfo(device\_list[0], CL\_DEVICE\_NAME, sizeof(buf), buf, 0);

//printf("Device: %s\n", buf);

// Create one OpenCL context for each device in the platform

cl\_context context;

context = clCreateContext(NULL, num\_devices, device\_list, NULL, NULL, &clStatus);

// Create a command queue

cl\_command\_queue command\_queue = clCreateCommandQueueWithProperties(context, device\_list[0], 0, &clStatus);

// Create a program from the kernel source

cl\_program program = clCreateProgramWithSource(context, 1, (const char\*\*)&code, NULL, &clStatus);

// Build the program

clStatus = clBuildProgram(program, 1, device\_list, NULL, NULL, NULL);

// Create the OpenCL kernel

cl\_kernel kernel = clCreateKernel(program, "del\_extra", &clStatus);

// Create memory buffers on the device for each vector

cl\_mem A\_clmem = clCreateBuffer(context, CL\_MEM\_READ\_ONLY, VECTOR\_SIZE \* sizeof(int), NULL, &clStatus);

cl\_mem B\_clmem = clCreateBuffer(context, CL\_MEM\_WRITE\_ONLY, VECTOR\_SIZE \* sizeof(int), NULL, &clStatus);

// Copy the Buffer A to the device

clStatus = clEnqueueWriteBuffer(command\_queue, A\_clmem, CL\_TRUE, 0, VECTOR\_SIZE \* sizeof(int), A, 0, NULL, NULL);

clStatus = clEnqueueWriteBuffer(command\_queue, B\_clmem, CL\_TRUE, 0, VECTOR\_SIZE \* sizeof(int), B, 0, NULL, NULL);

// Set the arguments of the kernel

clStatus = clSetKernelArg(kernel, 0, sizeof(cl\_mem), (void\*)&A\_clmem);

clStatus = clSetKernelArg(kernel, 1, sizeof(cl\_mem), (void\*)&B\_clmem);

// Execute the OpenCL kernel on the list

size\_t global\_size = VECTOR\_SIZE; // Process the entire lists

clStatus = clEnqueueNDRangeKernel(command\_queue, kernel, 1, NULL, &global\_size, 0, 0, NULL, NULL);

// Read the cl memory B\_clmem on device to the host variable B

clStatus = clEnqueueReadBuffer(command\_queue, B\_clmem, CL\_TRUE, 0, VECTOR\_SIZE \* sizeof(int), B, 0, NULL, NULL);

double timeGPUTotal = (double)(clock() - start) / CLOCKS\_PER\_SEC;

printf("%lf\n", timeGPUTotal);

// Display the result to the screen

/\* printf("initial matrix:\n");

for (i = 0; i < len; i++)

{

for (j = 0; j < len; j++)

printf("%3ld ", a[i \* len + j]);

printf("\n");

}\*/

long long DET = det(A, SIZE);

/\*printf("%lld\n", DET);

printf("Inverse matrix:\n");

for (i = 0; i < len; i++)

{

for (j = 0; j < len; j++)

{

printf("%f ", (float)B[i \* SIZE + j]/DET);

}

printf("\n");

}\*/

// Finally release all OpenCL allocated objects and host buffers.

clStatus = clReleaseKernel(kernel);

clStatus = clReleaseProgram(program);

clStatus = clReleaseMemObject(A\_clmem);

clStatus = clReleaseMemObject(B\_clmem);

clStatus = clReleaseCommandQueue(command\_queue);

clStatus = clReleaseContext(context);

free(B);

free(A);

free(device\_list);

free(platforms);

return 0;

}

float det(int\* A, int size)

{

float tmp[VECTOR\_SIZE];

float koef;

for (int i = 0; i < VECTOR\_SIZE; i++)

tmp[i] = A[i];

for (int i = 0; i < size - 1; i++)

{

for (int j = i + 1; j < size; j++)

{

koef = (float)tmp[j \* size + i] / tmp[i \* size + i];

for (int ind = 0; ind < size; ind++)

tmp[j \* size + ind] -= tmp[i \* size + ind] \* koef;

}

}

float res = 1;

for (int i = size - 1; i >= 0; i--)

res \*= tmp[(size + 1) \* i];

return res;

}

**ПРИЛОЖЕНИЕ C**

Листинг программы “Ядро”

#define VECTOR\_SIZE 100

#define SIZE 10

float calc\_det(int\* a, int size)

{

float tmp[VECTOR\_SIZE];

float koef;

for (int i = 0; i < VECTOR\_SIZE; i++)

tmp[i] = a[i];

for (int i = 0; i < size - 1; i++)

{

for (int j = i + 1; j < size; j++)

{

koef = (float)tmp[j \* size + i] / tmp[i \* size + i];

for (int ind = 0; ind < size; ind++)

tmp[j \* size + ind] -= tmp[i \* size + ind] \* koef;

}

}

float res = 1;

for (int i = size - 1; i >= 0; i--)

res \*= tmp[(size + 1) \* i];

return res;

}

\_\_kernel

void del\_extra(\_\_global int\* A, \_\_global int\* B)

{

int index = get\_global\_id(0);

int a\_copy[VECTOR\_SIZE], b\_copy[VECTOR\_SIZE];

int sn = SIZE - 1;

int i = index / SIZE, j = index % SIZE;

int z = 0;

for (int k = 0; k < VECTOR\_SIZE; k++)

{

if (k / SIZE == i || k % SIZE == j)

continue;

b\_copy[z] = A[k];

z++;

}

double rew = calc\_det(b\_copy, sn);

double t = (int)rew + 0.5;

double tm = (int)rew - 0.5;

if (rew - t > 0 && rew > 0)

B[i + SIZE \* j] = (int)rew + 1;

if (rew - tm < 0 && rew < 0)

B[i + SIZE \* j] = (int)rew - 1;

else

B[i + SIZE \* j] = (int)rew;

if ((i + j) % 2 != 0)

B[i + SIZE \* j] = -B[i + SIZE \* j];

}