Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет

информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра Информатики

Дисциплина «Архитектура вычислительных систем»

**ОТЧЕТ**

к практической работе №3

на тему:

**«работа с графическими процессорами различных семейств и поколений, исследуя их производительность в режиме вычислений»**

БГУИР 6-05-0612-02 01

|  |
| --- |
| Выполнила студентка группы 353504  АНТОНОВА Лидия Сергеевна |
|  |
| (дата, подпись студента) |
| Проверила  КАЛИНОВСКАЯ Анастасия Александровна |
|  |
| (дата, подпись преподавателя) |

Минск 2025

# Цель работы

Научиться работать с графическими процессорами различных семейств и поколений, исследуя их производительность в режиме вычислений.

# Индивидуальное задание

Выбрав вариант задания, а именно, заданную функцию осуществить написание программы на любом удобном языке программирования, с использованием CUDA или OpenCL. Выполнение задачи должно осуществляться в операционной системе без графической оболочки (Linux Ubuntu, Debian и др. – можно в LiveCD режиме, можно посредством установки второй системы на ноутбук). Реализуйте параллельные вычисления, используя возможности GPU/iGPU, чтобы ускорить выполнение задачи. Убедитесь, что ваша программа может обрабатывать более 5000 итераций для получения достоверных результатов.

# Выполнение работы

Для выполнения задачи была разработана программа, реализующая вычисление значения функции, суммы и количества итераций до достижения заданной точности. Программа была написана на языке C++ с использованием фреймворка OpenCL. Время выполнения каждого цикла программы замерялось с использованием clGetEventProfilingInfo. Основной листинг программы:

#define CL\_TARGET\_OPENCL\_VERSION 120

#include <CL/cl.h>

#include <iostream>

#include <chrono>

#include <fstream>

#include <cmath>

#include <vector>

#define CHECK\_ERROR(err, msg) \

if (err != CL\_SUCCESS) { \

std::cerr << msg << " Error: " << err << std::endl; \

exit(1); \

}

const char\* kernelSource = R"CLC(

#pragma OPENCL EXTENSION cl\_khr\_fp64 : enable

\_\_kernel void solve\_all(const double a,

const double h,

const int count,

const double epsilon,

\_\_global int\* iterations,

\_\_global double\* results)

{

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

if (i >= count) return;

double x = a + i \* h;

double Y = -log(fabs(2.0 \* sin(x / 2.0)));

double result = 0.0;

int iter = 1;

while (fabs(result - Y) >= epsilon) {

double kx = iter \* x;

double term = cos(kx) / iter;

result += term;

iter++;

}

iterations[i] = iter;

results[i] = result;

}

)CLC";

std::ofstream file("data.txt", std::ios::out);

double SolveY(double x){

return -(log(fabs(2 \* sin(x / 2))));

}

int main() {

double a, b, h, epsilon;

a = 0.01;

b = 1;

h = 0.01;

epsilon = 0.0005;

int count = static\_cast<int>((b - a) / h) + 1;

std::vector<int> iterations(count);

std::vector<double> results(count);

// --- Настройка OpenCL ---

cl\_int err;

cl\_uint numPlatforms;

err = clGetPlatformIDs(0, nullptr, &numPlatforms);

CHECK\_ERROR(err, "Не удалось получить список платформ");

std::vector<cl\_platform\_id> platforms(numPlatforms);

err = clGetPlatformIDs(numPlatforms, platforms.data(), nullptr);

CHECK\_ERROR(err, "Не удалось получить идентификаторы платформ");

// Выбираем первую платформу

cl\_platform\_id platform = platforms[0];

cl\_uint numDevices;

err = clGetDeviceIDs(platform, CL\_DEVICE\_TYPE\_ALL, 0, nullptr, &numDevices);

CHECK\_ERROR(err, "Не удалось получить количество устройств");

std::vector<cl\_device\_id> devices(numDevices);

err = clGetDeviceIDs(platform, CL\_DEVICE\_TYPE\_ALL, numDevices, devices.data(), nullptr);

CHECK\_ERROR(err, "Не удалось получить идентификаторы устройств");

cl\_device\_id device = devices[0];

// Создаём контекст

cl\_context\_properties props[] = { CL\_CONTEXT\_PLATFORM, (cl\_context\_properties)platform, 0 };

cl\_context context = clCreateContext(props, 1, &device, nullptr, nullptr, &err);

CHECK\_ERROR(err, "Ошибка при создании контекста");

// Создаём командную очередь

cl\_command\_queue\_properties propsQue = CL\_QUEUE\_PROFILING\_ENABLE;

cl\_command\_queue queue = clCreateCommandQueue(context, device, propsQue, &err);

CHECK\_ERROR(err, "Ошибка при создании командной очереди");

// Создаём объект программы из исходного кода

cl\_program program = clCreateProgramWithSource(context, 1, &kernelSource, nullptr, &err);

CHECK\_ERROR(err, "Ошибка при создании программы");

// Компилируем программу

err = clBuildProgram(program, 1, &device, nullptr, nullptr, nullptr);

if (err != CL\_SUCCESS) {

// В случае ошибки компиляции выводим лог

size\_t log\_size;

clGetProgramBuildInfo(program, device, CL\_PROGRAM\_BUILD\_LOG, 0, nullptr, &log\_size);

std::vector<char> log(log\_size);

clGetProgramBuildInfo(program, device, CL\_PROGRAM\_BUILD\_LOG, log\_size, log.data(), nullptr);

std::cerr << "Ошибка сборки программы:\n" << log.data() << std::endl;

exit(1);

}

// Создаём ядро

cl\_kernel kernel = clCreateKernel(program, "solve\_all", &err);

CHECK\_ERROR(err, "Ошибка при создании ядра");

// Создаём буферы для результатов

cl\_mem iterBuffer = clCreateBuffer(context, CL\_MEM\_WRITE\_ONLY, sizeof(int) \* count, nullptr, &err);

CHECK\_ERROR(err, "Ошибка создания буфера для iterations");

cl\_mem resultsBuffer = clCreateBuffer(context, CL\_MEM\_WRITE\_ONLY, sizeof(double) \* count, nullptr, &err);

CHECK\_ERROR(err, "Ошибка создания буфера для results");

// Устанавливаем аргументы ядра

err = clSetKernelArg(kernel, 0, sizeof(double), &a);

err |= clSetKernelArg(kernel, 1, sizeof(double), &h);

err |= clSetKernelArg(kernel, 2, sizeof(int), &count);

err |= clSetKernelArg(kernel, 3, sizeof(double), &epsilon);

err |= clSetKernelArg(kernel, 4, sizeof(cl\_mem), &iterBuffer);

err |= clSetKernelArg(kernel, 5, sizeof(cl\_mem), &resultsBuffer);

CHECK\_ERROR(err, "Ошибка установки аргументов ядра");

// Запускаем ядро: глобальный размер равен количеству x-значений

size\_t globalSize = count;

auto start = std::chrono::high\_resolution\_clock::now();

err = clEnqueueNDRangeKernel(queue, kernel, 1, nullptr, &globalSize, nullptr, 0, nullptr, nullptr);

CHECK\_ERROR(err, "Ошибка enqueue ядра");

// Ждём завершения выполнения

clFinish(queue);

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsed = end - start;

// Читаем результаты из буферов

err = clEnqueueReadBuffer(queue, iterBuffer, CL\_TRUE, 0, sizeof(int) \* count, iterations.data(), 0, nullptr, nullptr);

CHECK\_ERROR(err, "Ошибка чтения буфера iterations");

err = clEnqueueReadBuffer(queue, resultsBuffer, CL\_TRUE, 0, sizeof(double) \* count, results.data(), 0, nullptr, nullptr);

CHECK\_ERROR(err, "Ошибка чтения буфера results");

std::cout << " x | Y(x) | S(x) | Итерации | Время (сек) " << std::endl;

std::cout << "-------------------------------------------------" << std::endl;

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

double cur\_x = a + i \* h;

err = clSetKernelArg(kernel, 0, sizeof(double), &cur\_x);

CHECK\_ERROR(err, "Ошибка установки аргумента");

size\_t globalSize = 1;

cl\_event event;

err = clEnqueueNDRangeKernel(queue, kernel, 1, nullptr, &globalSize,

nullptr, 0, nullptr, &event);

CHECK\_ERROR(err, "Ошибка enqueue ядра");

clFinish(queue);

cl\_ulong time\_start, time\_end;

err = clGetEventProfilingInfo(event, CL\_PROFILING\_COMMAND\_START, sizeof(time\_start), &time\_start, nullptr);

CHECK\_ERROR(err, "Ошибка получения времени старта");

err = clGetEventProfilingInfo(event, CL\_PROFILING\_COMMAND\_END, sizeof(time\_end), &time\_end, nullptr);

CHECK\_ERROR(err, "Ошибка получения времени завершения");

double exec\_time\_sec = (time\_end - time\_start) \* 1e-7;

std::cout << cur\_x << " | " << SolveY(cur\_x) << " | " << results[i] << " | " << iterations[i]

<< " | " << exec\_time\_sec << "\n";

file << iterations[i]<< ' ' << exec\_time\_sec << "\n";

clReleaseEvent(event);

}

file.close();

// Очистка ресурсов OpenCL

clReleaseMemObject(iterBuffer);

clReleaseMemObject(resultsBuffer);

clReleaseKernel(kernel);

clReleaseProgram(program);

clReleaseCommandQueue(queue);

clReleaseContext(context);

return 0;

}

# АНАлиз полученных результатов

На рисунке 1 показан график нагрузки на ядра процессора.

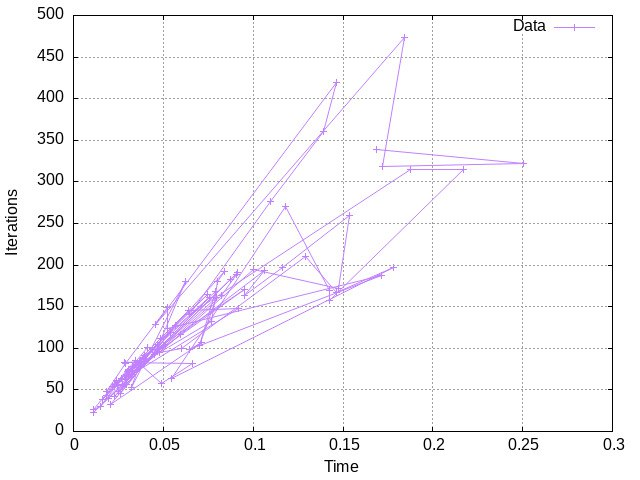


Рисунок 1

# выводы

В ходе работы была реализована и протестирована параллельная вычислительная программа с использованием OpenCL, предназначенная для эффективного выполнения математических вычислений на процессорах и графических ускорителях.

Использование OpenCL позволило значительно ускорить выполнение вычислений по сравнению с последовательным вариантом программы благодаря возможностям параллельной обработки данных. Оптимизация была достигнута за счёт разбиения задач на отдельные рабочие элементы (work-items), каждый из которых выполняется независимо в среде OpenCL.

При тестировании различных параметров платформы было выявлено, что производительность зависит от конкретного устройства (CPU/GPU) и доступной реализации OpenCL. В процессе работы были изучены особенности работы с OpenCL-платформами, механизмами создания контекста и буферов, а также профилирование выполнения отдельных вычислений.