**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ   
РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ**

**ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**

**(БГТУ им. В.Г. Шухова)**



ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

**Лабораторная работа №4**

по дисциплине: Параллельное программирование

тема: «Параллельное программирование с использованием OpenCL»

Выполнил: ст. группы ПВ-223

Игнатьев Артур Олегович

Проверили:

доц. Островский Алексей Мичеславович

Белгород 2025 г.

**Лабораторная работа №4  
Параллельное программирование с использованием OpenCL.**

**Цель работы**: Изучить основы параллельного программирования с использованием OpenCL, реализовать вычислительные задачи с применением графического ускорителя (GPU), оценить производительность и масштабируемость решений при выполнении вычислений.

**Цель работы обуславливает постановку и решение следующих задач:**

1. Изучить архитектуру и программную модель OpenCL, включая понятия платформ, устройств, контекстов, очередей команд, ядер (kernels), рабочих элементов и групп.

2. Освоить процесс компиляции и запуска OpenCL-программ в среде Linux, включая настройку окружения, установку драйверов и инструментов, проверку доступности устройств и базовую сборку кода.

3. Реализовать задачу-пример с использованием OpenCL, реализующую умножение двух квадратных матриц большого размера.

4. Научиться загружать и исполнять OpenCL-ядра, передавать данные между CPU (хост) и GPU (устройство), управлять памятью, синхронизацией и чтением результатов.

5. Изучить подходы к декомпозиции вычислительной задачи для GPU, включая:

5.1 Разбиение задачи на элементарные операции (work-items),

5.2 Разбиение массива или данных на блоки (work-groups).

5.3 Определение схемы обращения к данным (coalesced memory access).

5.4 Минимизацию конфликтов доступа и использование локальной памяти устройства.

6. Выполнить индивидуальное задание, связанное с реализацией вычислительной задачи на OpenCL: декомпозировать задачу на параллельно исполняемые фрагменты (work-items и work-groups), обеспечить эффективное распределение вычислений между элементами устройства, обратить внимание на балансировку нагрузки, схему обращения к памяти и взаимодействие между уровнями иерархии памяти (глобальная, локальная, приватная).

7. Провести сравнение производительности OpenCL-программ с однопоточными реализациями на CPU, выявить выигрыш в скорости, определить факторы, влияющие на масштабируемость (размер входных данных, конфигурация устройства, количество work- items).

8. Оформить отчёт с выводами по эффективности параллельного подхода, включая:

8.1 Описание архитектурных решений.

8.2 Анализ времени выполнения.

8.3 Графики ускорения и зависимостей.

8.4 Оценку применимости OpenCL к подобным задачам в практической деятельности.

**Индивидуальное задание**

**Вариант 3**

Реализовать параллельные алгоритмы для обучения и предсказания с помощью модели

линейной регрессии в пространстве базисных функций, используя OpenCL. Дан набор

данных dataset[N][D], где N — количество обучающих примеров, D — размерность

признакового пространства. Даны метки labels[N] (вещественные значения для задачи

регрессии). Модель регрессии имеет вид:

где φ(x) — некоторая базисная функция (например, синус), wi — веса модели, — b смещение. Требуется реализовать параллельный прямой проход (forward pass), когда каждая параллельная нить (thread) вычисляет предсказание y\_pred для одного обучающего примера; реализовать параллельное вычисление локальных градиентов по каждому примеру; выполнить редукцию локальных градиентов для обновления весов и смещения; обеспечить эффективную работу на GPU с использованием OpenCL; вывести предсказания на обучающем наборе данных; вывести предсказание на новом тестовом примере. Сравнить производительность (по времени выполнения) между реализациями на GPU и CPU. Использовать базовый тип данных float. Данные загружать из файла. (См. однопоточный прототип в файле 3.py)

**Ход выполнения лабораторной работы**

Реализация набора данных (random\_dataset.py)

import numpy as np

N, D = 1000, 500

dataset = np.random.rand(N, D) \* 10

labels = np.sum(np.sin(dataset), axis=1) \* 0.1 + np.random.randn(N) \* 0.1

with open("data.txt", "w") as f:

    f.write(f"{N} {D}\n")

    for i in range(N):

        f.write(" ".join(map(str, dataset[i])) + f" {labels[i]}\n")

Реализация линейной регрессии с использованием OpenCL на языке C (main.c)

#define CL\_TARGET\_OPENCL\_VERSION 200

#include <CL/cl.h>

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#define EPSILON 0.01

#define LEARNING\_RATE 0.001

#define EPOCHS 1000

typedef struct {

    cl\_context context;

    cl\_command\_queue queue;

    cl\_program program;

    cl\_kernel forward\_kernel;

    cl\_kernel gradient\_kernel;

    cl\_kernel reduce\_kernel;

    cl\_device\_id device;

} OpenCLContext;

// Проверка ошибок OpenCL

int check\_error(cl\_int err, const char\* msg) {

    if (err != CL\_SUCCESS) {

        fprintf(stderr, "%s failed: %d\n", msg, err);

        return 0;

    }

    return 1;

}

// Настройка OpenCL

int setup\_opencl(OpenCLContext\* cl\_ctx) {

    cl\_platform\_id platform;

    cl\_int err;

    int ok = 1;

    err = clGetPlatformIDs(1, &platform, NULL);

    ok &= check\_error(err, "clGetPlatformIDs");

    if (ok) {

        err = clGetDeviceIDs(platform, CL\_DEVICE\_TYPE\_GPU, 1, &cl\_ctx->device, NULL);

        ok &= check\_error(err, "clGetDeviceIDs");

    }

    if (ok) {

        cl\_ctx->context = clCreateContext(NULL, 1, &cl\_ctx->device, NULL, NULL, &err);

        ok &= check\_error(err, "clCreateContext");

    }

    if (ok) {

        cl\_command\_queue\_properties props[] = {CL\_QUEUE\_PROPERTIES, CL\_QUEUE\_PROFILING\_ENABLE, 0};

        cl\_ctx->queue = clCreateCommandQueueWithProperties(cl\_ctx->context, cl\_ctx->device, props, &err);

        ok &= check\_error(err, "clCreateCommandQueueWithProperties");

    }

    if (ok) {

        const char\* kernel\_source =

            "\_\_kernel void forward\_pass(\_\_global const float\* dataset, \_\_global const float\* weights, "

            "\_\_global const float\* bias, \_\_global float\* predictions, const unsigned int D) {"

            "    int gid = get\_global\_id(0);"

            "    float sum = 0.0f;"

            "    for (unsigned int i = 0; i < D; i++) {"

            "        sum += weights[i] \* sin(dataset[gid \* D + i]);"

            "    }"

            "    predictions[gid] = sum + \*bias;"

            "}"

            "\_\_kernel void compute\_gradients(\_\_global const float\* dataset, \_\_global const float\* labels, "

            "\_\_global const float\* predictions, \_\_global float\* grad\_weights, \_\_global float\* grad\_bias, "

            "const unsigned int D, const unsigned int N) {"

            "    int gid = get\_global\_id(0);"

            "    if (gid < N) {"

            "        float error = predictions[gid] - labels[gid];"

            "        for (unsigned int i = 0; i < D; i++) {"

            "            grad\_weights[gid \* D + i] = error \* sin(dataset[gid \* D + i]);"

            "        }"

            "        grad\_bias[gid] = error;"

            "    }"

            "}"

            "\_\_kernel void reduce\_gradients(\_\_global const float\* grad\_weights, \_\_global float\* weights, "

            "\_\_global const float\* grad\_bias, \_\_global float\* bias, const unsigned int D, const unsigned int N, "

            "const float lr) {"

            "    int gid = get\_global\_id(0);"

            "    if (gid < D) {"

            "        float sum = 0.0f;"

            "        for (unsigned int i = 0; i < N; i++) {"

            "            sum += grad\_weights[i \* D + gid];"

            "        }"

            "        weights[gid] -= lr \* sum / N;"

            "    }"

            "    if (gid == 0) {"

            "        float sum\_bias = 0.0f;"

            "        for (unsigned int i = 0; i < N; i++) {"

            "            sum\_bias += grad\_bias[i];"

            "        }"

            "        \*bias -= lr \* sum\_bias / N;"

            "    }"

            "}";

        cl\_ctx->program = clCreateProgramWithSource(cl\_ctx->context, 1, &kernel\_source, NULL, &err);

        ok &= check\_error(err, "clCreateProgramWithSource");

    }

    if (ok) {

        err = clBuildProgram(cl\_ctx->program, 1, &cl\_ctx->device, NULL, NULL, NULL);

        if (err != CL\_SUCCESS) {

            size\_t log\_size;

            clGetProgramBuildInfo(cl\_ctx->program, cl\_ctx->device, CL\_PROGRAM\_BUILD\_LOG, 0, NULL, &log\_size);

            char\* log = (char\*)malloc(log\_size);

            clGetProgramBuildInfo(cl\_ctx->program, cl\_ctx->device, CL\_PROGRAM\_BUILD\_LOG, log\_size, log, NULL);

            fprintf(stderr, "Build log:\n%s\n", log);

            free(log);

            return 0;

        }

        ok &= check\_error(err, "clBuildProgram");

    }

    if (ok) {

        cl\_ctx->forward\_kernel = clCreateKernel(cl\_ctx->program, "forward\_pass", &err);

        ok &= check\_error(err, "clCreateKernel forward\_pass");

        cl\_ctx->gradient\_kernel = clCreateKernel(cl\_ctx->program, "compute\_gradients", &err);

        ok &= check\_error(err, "clCreateKernel compute\_gradients");

        cl\_ctx->reduce\_kernel = clCreateKernel(cl\_ctx->program, "reduce\_gradients", &err);

        ok &= check\_error(err, "clCreateKernel reduce\_gradients");

    }

    return ok;

}

// Очистка ресурсов

void cleanup(OpenCLContext\* cl\_ctx) {

    clReleaseKernel(cl\_ctx->forward\_kernel);

    clReleaseKernel(cl\_ctx->gradient\_kernel);

    clReleaseKernel(cl\_ctx->reduce\_kernel);

    clReleaseProgram(cl\_ctx->program);

    clReleaseCommandQueue(cl\_ctx->queue);

    clReleaseContext(cl\_ctx->context);

}

// CPU-реализация

void calculate\_cpu(float\* dataset, float\* labels, float\* weights, float\* bias, unsigned int N, unsigned int D, float\* predictions) {

    for (unsigned int i = 0; i < N; i++) {

        float sum = 0.0f;

        for (unsigned int j = 0; j < D; j++) {

            sum += weights[j] \* sinf(dataset[i \* D + j]);

        }

        predictions[i] = sum + \*bias;

    }

}

void train\_cpu(float\* dataset, float\* labels, float\* weights, float\* bias, unsigned int N, unsigned int D, float lr) {

    float\* predictions = (float\*)malloc(N \* sizeof(float));

    float\* grad\_weights = (float\*)malloc(D \* sizeof(float));

    float grad\_bias = 0.0f;

    for (unsigned int epoch = 0; epoch < EPOCHS; epoch++) {

        calculate\_cpu(dataset, labels, weights, bias, N, D, predictions);

        for (unsigned int j = 0; j < D; j++) {

            grad\_weights[j] = 0.0f;

        }

        grad\_bias = 0.0f;

        for (unsigned int i = 0; i < N; i++) {

            float error = predictions[i] - labels[i];

            for (unsigned int j = 0; j < D; j++) {

                grad\_weights[j] += error \* sinf(dataset[i \* D + j]);

            }

            grad\_bias += error;

        }

        for (unsigned int j = 0; j < D; j++) {

            weights[j] -= lr \* grad\_weights[j] / N;

        }

        \*bias -= lr \* grad\_bias / N;

    }

    free(predictions);

    free(grad\_weights);

}

// GPU-реализация

int run\_kernels(OpenCLContext\* cl\_ctx, float\* dataset, float\* labels, float\* weights, float\* bias, unsigned int N, unsigned int D, float\*\* predictions\_out) {

    int ok = 1;

    const size\_t LOCAL\_SIZE = 256;

    size\_t global\_size = ((N + LOCAL\_SIZE - 1) / LOCAL\_SIZE) \* LOCAL\_SIZE;

    size\_t reduce\_global\_size = ((D + LOCAL\_SIZE - 1) / LOCAL\_SIZE) \* LOCAL\_SIZE;

    cl\_mem dataset\_buf = clCreateBuffer(cl\_ctx->context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, sizeof(float) \* N \* D, dataset, NULL);

    cl\_mem labels\_buf = clCreateBuffer(cl\_ctx->context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, sizeof(float) \* N, labels, NULL);

    cl\_mem weights\_buf = clCreateBuffer(cl\_ctx->context, CL\_MEM\_READ\_WRITE | CL\_MEM\_COPY\_HOST\_PTR, sizeof(float) \* D, weights, NULL);

    cl\_mem bias\_buf = clCreateBuffer(cl\_ctx->context, CL\_MEM\_READ\_WRITE | CL\_MEM\_COPY\_HOST\_PTR, sizeof(float), bias, NULL);

    cl\_mem predictions\_buf = clCreateBuffer(cl\_ctx->context, CL\_MEM\_WRITE\_ONLY, sizeof(float) \* N, NULL, NULL);

    cl\_mem grad\_weights\_buf = clCreateBuffer(cl\_ctx->context, CL\_MEM\_WRITE\_ONLY, sizeof(float) \* N \* D, NULL, NULL);

    cl\_mem grad\_bias\_buf = clCreateBuffer(cl\_ctx->context, CL\_MEM\_WRITE\_ONLY, sizeof(float) \* N, NULL, NULL);

    for (unsigned int epoch = 0; epoch < EPOCHS; epoch++) {

        // Прямой проход

        ok &= check\_error(clSetKernelArg(cl\_ctx->forward\_kernel, 0, sizeof(cl\_mem), &dataset\_buf), "set forward arg 0");

        ok &= check\_error(clSetKernelArg(cl\_ctx->forward\_kernel, 1, sizeof(cl\_mem), &weights\_buf), "set forward arg 1");

        ok &= check\_error(clSetKernelArg(cl\_ctx->forward\_kernel, 2, sizeof(cl\_mem), &bias\_buf), "set forward arg 2");

        ok &= check\_error(clSetKernelArg(cl\_ctx->forward\_kernel, 3, sizeof(cl\_mem), &predictions\_buf), "set forward arg 3");

        ok &= check\_error(clSetKernelArg(cl\_ctx->forward\_kernel, 4, sizeof(unsigned int), &D), "set forward arg 4");

        ok &= check\_error(clEnqueueNDRangeKernel(cl\_ctx->queue, cl\_ctx->forward\_kernel, 1, NULL, &global\_size, &LOCAL\_SIZE, 0, NULL, NULL), "enqueue forward kernel");

        // Вычисление градиентов

        ok &= check\_error(clSetKernelArg(cl\_ctx->gradient\_kernel, 0, sizeof(cl\_mem), &dataset\_buf), "set gradient arg 0");

        ok &= check\_error(clSetKernelArg(cl\_ctx->gradient\_kernel, 1, sizeof(cl\_mem), &labels\_buf), "set gradient arg 1");

        ok &= check\_error(clSetKernelArg(cl\_ctx->gradient\_kernel, 2, sizeof(cl\_mem), &predictions\_buf), "set gradient arg 2");

        ok &= check\_error(clSetKernelArg(cl\_ctx->gradient\_kernel, 3, sizeof(cl\_mem), &grad\_weights\_buf), "set gradient arg 3");

        ok &= check\_error(clSetKernelArg(cl\_ctx->gradient\_kernel, 4, sizeof(cl\_mem), &grad\_bias\_buf), "set gradient arg 4");

        ok &= check\_error(clSetKernelArg(cl\_ctx->gradient\_kernel, 5, sizeof(unsigned int), &D), "set gradient arg 5");

        ok &= check\_error(clSetKernelArg(cl\_ctx->gradient\_kernel, 6, sizeof(unsigned int), &N), "set gradient arg 6");

        ok &= check\_error(clEnqueueNDRangeKernel(cl\_ctx->queue, cl\_ctx->gradient\_kernel, 1, NULL, &global\_size, &LOCAL\_SIZE, 0, NULL, NULL), "enqueue gradient kernel");

        // Редукция градиентов

        ok &= check\_error(clSetKernelArg(cl\_ctx->reduce\_kernel, 0, sizeof(cl\_mem), &grad\_weights\_buf), "set reduce arg 0");

        ok &= check\_error(clSetKernelArg(cl\_ctx->reduce\_kernel, 1, sizeof(cl\_mem), &weights\_buf), "set reduce arg 1");

        ok &= check\_error(clSetKernelArg(cl\_ctx->reduce\_kernel, 2, sizeof(cl\_mem), &grad\_bias\_buf), "set reduce arg 2");

        ok &= check\_error(clSetKernelArg(cl\_ctx->reduce\_kernel, 3, sizeof(cl\_mem), &bias\_buf), "set reduce arg 3");

        ok &= check\_error(clSetKernelArg(cl\_ctx->reduce\_kernel, 4, sizeof(unsigned int), &D), "set reduce arg 4");

        ok &= check\_error(clSetKernelArg(cl\_ctx->reduce\_kernel, 5, sizeof(unsigned int), &N), "set reduce arg 5");

        float lr = LEARNING\_RATE;

        ok &= check\_error(clSetKernelArg(cl\_ctx->reduce\_kernel, 6, sizeof(float), &lr), "set reduce arg 6");

        ok &= check\_error(clEnqueueNDRangeKernel(cl\_ctx->queue, cl\_ctx->reduce\_kernel, 1, NULL, &reduce\_global\_size, &LOCAL\_SIZE, 0, NULL, NULL), "enqueue reduce kernel");

    }

    // Чтение результатов

    \*predictions\_out = (float\*)malloc(N \* sizeof(float));

    ok &= check\_error(clEnqueueReadBuffer(cl\_ctx->queue, predictions\_buf, CL\_TRUE, 0, sizeof(float) \* N, \*predictions\_out, 0, NULL, NULL), "read predictions");

    // Чтение весов и смещения

    ok &= check\_error(clEnqueueReadBuffer(cl\_ctx->queue, weights\_buf, CL\_TRUE, 0, sizeof(float) \* D, weights, 0, NULL, NULL), "read weights");

    ok &= check\_error(clEnqueueReadBuffer(cl\_ctx->queue, bias\_buf, CL\_TRUE, 0, sizeof(float), bias, 0, NULL, NULL), "read bias");

    // Вывод предсказаний

    printf("Предсказания на обучающем наборе (GPU):\n");

    for (unsigned int i = 0; i < N && i < 10; i++) {

        printf("Пример %u: %.6f\n", i, (\*predictions\_out)[i]);

    }

    // Освобождение ресурсов

    clReleaseMemObject(dataset\_buf);

    clReleaseMemObject(labels\_buf);

    clReleaseMemObject(weights\_buf);

    clReleaseMemObject(bias\_buf);

    clReleaseMemObject(predictions\_buf);

    clReleaseMemObject(grad\_weights\_buf);

    clReleaseMemObject(grad\_bias\_buf);

    return ok;

}

// Измерение времени

double measure\_time(void (\*func)(void\*), void\* arg) {

    struct timespec start, end;

    clock\_gettime(CLOCK\_MONOTONIC, &start);

    func(arg);

    clock\_gettime(CLOCK\_MONOTONIC, &end);

    return (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9;

}

typedef struct {

    float\* dataset;

    float\* labels;

    float\* weights;

    float\* bias;

    unsigned int N, D;

} CpuTaskArgs;

typedef struct {

    OpenCLContext\* cl\_ctx;

    float\* dataset;

    float\* labels;

    float\* weights;

    float\* bias;

    unsigned int N, D;

    int\* ok;

    float\*\* predictions;

} GpuTaskArgs;

void cpu\_task(void\* args) {

    CpuTaskArgs\* cpu\_args = (CpuTaskArgs\*)args;

    train\_cpu(cpu\_args->dataset, cpu\_args->labels, cpu\_args->weights, cpu\_args->bias, cpu\_args->N, cpu\_args->D, LEARNING\_RATE);

}

void gpu\_task(void\* args) {

    GpuTaskArgs\* gpu\_args = (GpuTaskArgs\*)args;

    \*(gpu\_args->ok) = run\_kernels(gpu\_args->cl\_ctx, gpu\_args->dataset, gpu\_args->labels, gpu\_args->weights, gpu\_args->bias, gpu\_args->N, gpu\_args->D, gpu\_args->predictions);

}

// Загрузка данных из файла

void load\_data(const char\* filename, float\*\* dataset, float\*\* labels, unsigned int\* N, unsigned int\* D) {

    FILE\* file = fopen(filename, "r");

    if (!file) {

        fprintf(stderr, "Не удалось открыть файл %s\n", filename);

        exit(1);

    }

    fscanf(file, "%u %u", N, D);

    \*dataset = (float\*)malloc((\*N) \* (\*D) \* sizeof(float));

    \*labels = (float\*)malloc((\*N) \* sizeof(float));

    for (unsigned int i = 0; i < \*N; i++) {

        for (unsigned int j = 0; j < \*D; j++) {

            fscanf(file, "%f", &(\*dataset)[i \* (\*D) + j]);

        }

        fscanf(file, "%f", &(\*labels)[i]);

    }

    fclose(file);

}

int main() {

    unsigned int N, D;

    float \*dataset, \*labels, \*weights\_cpu, \*weights\_gpu, bias\_cpu = 0.0f, bias\_gpu = 0.0f;

    // Загрузка данных

    load\_data("data.txt", &dataset, &labels, &N, &D);

    // Инициализация весов

    weights\_cpu = (float\*)malloc(D \* sizeof(float));

    weights\_gpu = (float\*)malloc(D \* sizeof(float));

    for (unsigned int i = 0; i < D; i++) {

        weights\_cpu[i] = weights\_gpu[i] = 0.0f;

    }

    // CPU выполнение

    CpuTaskArgs cpu\_args = {dataset, labels, weights\_cpu, &bias\_cpu, N, D};

    double cpu\_time = measure\_time(cpu\_task, &cpu\_args);

    // GPU выполнение

    OpenCLContext\* cl\_ctx = (OpenCLContext\*)malloc(sizeof(OpenCLContext));

    int ok = setup\_opencl(cl\_ctx);

    if (ok) {

        float\* predictions\_gpu = NULL;

        GpuTaskArgs gpu\_args = {cl\_ctx, dataset, labels, weights\_gpu, &bias\_gpu, N, D, &ok, &predictions\_gpu};

        double gpu\_time = measure\_time(gpu\_task, &gpu\_args);

        // Вывод времени

        printf("Затраты CPU: %.6f секунд\n", cpu\_time);

        printf("Затраты GPU: %.6f секунд\n", gpu\_time);

        printf("Ускорение: %.2fx\n", cpu\_time / gpu\_time);

        // Вывод весов и смещения для диагностики

        printf("Веса GPU: ");

        for (unsigned int i = 0; i < D && i < 10; i++) { // Ограничиваем вывод первыми 10 весами

            printf("%.6f ", weights\_gpu[i]);

        }

        printf("\nСмещение GPU: %.6f\n", bias\_gpu);

        // Проверка корректности

        float\* predictions\_cpu = (float\*)malloc(N \* sizeof(float));

        calculate\_cpu(dataset, labels, weights\_cpu, &bias\_cpu, N, D, predictions\_cpu);

        printf("Проверка результатов (CPU vs GPU):\n");

        int correct = 1;

        for (unsigned int i = 0; i < N && i < 10; i++) {

            printf("Пример %u: CPU=%.6f, GPU=%.6f\n", i, predictions\_cpu[i], predictions\_gpu[i]);

            if (fabs(predictions\_cpu[i] - predictions\_gpu[i]) > EPSILON) {

                correct = 0;

            }

        }

        printf("Проверка: %s\n", correct ? "OK" : "ОШИБКА");

        // Тестовый пример

        float\* test\_point = (float\*)malloc(D \* sizeof(float));

        for (unsigned int i = 0; i < D; i++) {

            test\_point[i] = (float)(i + 1); // Пример: 1.0, 2.0, ..., D

        }

        float test\_pred = 0.0f;

        for (unsigned int i = 0; i < D; i++) {

            test\_pred += weights\_gpu[i] \* sinf(test\_point[i]);

        }

        test\_pred += bias\_gpu;

        printf("Предсказание на тестовом примере: %.6f\n", test\_pred);

        free(predictions\_cpu);

        free(predictions\_gpu);

        free(test\_point);

        cleanup(cl\_ctx);

    }

    // Освобождение памяти

    free(dataset);

    free(labels);

    free(weights\_cpu);

    free(weights\_gpu);

    free(cl\_ctx);

    return ok ? 0 : 1;

}

Реализация линейной регрессии с использованием OpenCL на языке Rust (main.rs)

use ocl::{Buffer, Context, Device, Kernel, Platform, Program, Queue}; // OpenCL библиотека

use std::fs::File; // Работа с файлами

use std::io::{self, BufRead, BufReader}; // Чтение файлов

use std::time::Instant; // Измерение времени

// Константы для обучения

const EPSILON: f32 = 0.01; // Порог для проверки предсказаний CPU и GPU

const LEARNING\_RATE: f32 = 0.001; // Скорость обучения

const EPOCHS: usize = 1000; // Количество эпох обучения

// Структура для хранения OpenCL контекста и ядер

#[allow(dead\_code)] // Подавление предупреждений о неиспользуемых полях

struct OpenCLContext {

    context: Context, // OpenCL контекст

    queue: Queue, // Очередь команд

    program: Program, // Программа с ядрами

    forward\_kernel: Kernel, // Ядро для прямого прохода

    gradient\_kernel: Kernel, // Ядро для вычисления градиентов

    reduce\_kernel: Kernel, // Ядро для редукции градиентов

}

impl OpenCLContext {

    // Инициализация OpenCL: платформа, устройство, контекст, очередь и ядра

    fn new() -> Result<Self, ocl::Error> {

        let platform = Platform::default(); // Выбор платформы по умолчанию

        let device = Device::first(platform)?; // Первое доступное устройство

        let context = Context::builder()

            .platform(platform)

            .devices(device)

            .build()?;

        let queue = Queue::new(&context, device, None)?;

        // Код ядер OpenCL

        let kernel\_source = r#"

            // Прямой проход: вычисляет предсказания как sum(weights[i] \* sin(dataset[i])) + bias

            \_\_kernel void forward\_pass(\_\_global const float\* dataset, \_\_global const float\* weights,

                                      \_\_global const float\* bias, \_\_global float\* predictions,

                                      const unsigned int D) {

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

                float sum = 0.0f;

                for (unsigned int i = 0; i < D; i++) {

                    sum += weights[i] \* sin(dataset[gid \* D + i]);

                }

                predictions[gid] = sum + \*bias;

            }

            // Вычисление градиентов: grad\_weights = error \* sin(dataset), grad\_bias = error

            \_\_kernel void compute\_gradients(\_\_global const float\* dataset, \_\_global const float\* labels,

                                           \_\_global const float\* predictions, \_\_global float\* grad\_weights,

                                           \_\_global float\* grad\_bias, const unsigned int D,

                                           const unsigned int N) {

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

                if (gid < N) {

                    float error = predictions[gid] - labels[gid];

                    for (unsigned int i = 0; i < D; i++) {

                        grad\_weights[gid \* D + i] = error \* sin(dataset[gid \* D + i]);

                    }

                    grad\_bias[gid] = error;

                }

            }

            // Редукция градиентов: обновляет веса и смещение по среднему градиенту

            \_\_kernel void reduce\_gradients(\_\_global const float\* grad\_weights, \_\_global float\* weights,

                                           \_\_global const float\* grad\_bias, \_\_global float\* bias,

                                           const unsigned int D, const unsigned int N, const float lr) {

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

                if (gid < D) {

                    float sum = 0.0f;

                    for (unsigned int i = 0; i < N; i++) {

                        sum += grad\_weights[i \* D + gid];

                    }

                    weights[gid] -= lr \* sum / N;

                }

                if (gid == 0) {

                    float sum\_bias = 0.0f;

                    for (unsigned int i = 0; i < N; i++) {

                        sum\_bias += grad\_bias[i];

                    }

                    \*bias -= lr \* sum\_bias / N;

                }

            }

        "#;

        // Компиляция программы OpenCL

        let program = Program::builder()

            .src(kernel\_source)

            .devices(device)

            .build(&context)?;

        // Создание ядра для прямого прохода

        let forward\_kernel = Kernel::builder()

            .program(&program)

            .name("forward\_pass")

            .queue(queue.clone())

            .arg(None::<&Buffer<f32>>)

            .arg(None::<&Buffer<f32>>)

            .arg(None::<&Buffer<f32>>)

            .arg(None::<&Buffer<f32>>)

            .arg(0u32)

            .build()?;

        // Создание ядра для вычисления градиентов

        let gradient\_kernel = Kernel::builder()

            .program(&program)

            .name("compute\_gradients")

            .queue(queue.clone())

            .arg(None::<&Buffer<f32>>)

            .arg(None::<&Buffer<f32>>)

            .arg(None::<&Buffer<f32>>)

            .arg(None::<&Buffer<f32>>)

            .arg(None::<&Buffer<f32>>)

            .arg(0u32)

            .arg(0u32)

            .build()?;

        // Создание ядра для редукции градиентов

        let reduce\_kernel = Kernel::builder()

            .program(&program)

            .name("reduce\_gradients")

            .queue(queue.clone())

            .arg(None::<&Buffer<f32>>)

            .arg(None::<&Buffer<f32>>)

            .arg(None::<&Buffer<f32>>)

            .arg(None::<&Buffer<f32>>)

            .arg(0u32)

            .arg(0u32)

            .arg(0.0f32)

            .build()?;

        Ok(OpenCLContext {

            context,

            queue,

            program,

            forward\_kernel,

            gradient\_kernel,

            reduce\_kernel,

        })

    }

}

// Вычисление предсказаний на CPU: sum(weights[i] \* sin(dataset[i])) + bias

fn calculate\_cpu(dataset: &[f32], \_labels: &[f32], weights: &[f32], bias: f32, n: usize, d: usize, predictions: &mut [f32]) {

    for i in 0..n {

        let mut sum = 0.0;

        for j in 0..d {

            sum += weights[j] \* dataset[i \* d + j].sin();

        }

        predictions[i] = sum + bias;

    }

}

// Обучение на CPU: градиентный спуск для линейной регрессии с sin

fn train\_cpu(dataset: &[f32], labels: &[f32], weights: &mut [f32], bias: &mut f32, n: usize, d: usize) {

    let mut predictions = vec![0.0; n]; // Буфер для предсказаний

    let mut grad\_weights = vec![0.0; d]; // Градиенты весов

    let mut grad\_bias; // Градиент смещения

    for \_ in 0..EPOCHS {

        calculate\_cpu(dataset, labels, weights, \*bias, n, d, &mut predictions);

        grad\_weights.fill(0.0); // Сброс градиентов

        grad\_bias = 0.0;

        // Вычисление градиентов

        for i in 0..n {

            let error = predictions[i] - labels[i];

            for j in 0..d {

                grad\_weights[j] += error \* dataset[i \* d + j].sin();

            }

            grad\_bias += error;

        }

        // Обновление весов и смещения

        for j in 0..d {

            weights[j] -= LEARNING\_RATE \* grad\_weights[j] / n as f32;

        }

        \*bias -= LEARNING\_RATE \* grad\_bias / n as f32;

    }

}

// Обучение на GPU: выполняет прямой проход, вычисление и редукцию градиентов

fn run\_kernels(

    cl\_ctx: &OpenCLContext,

    dataset: &[f32],

    labels: &[f32],

    weights: &mut [f32],

    bias: &mut f32,

    n: usize,

    d: usize,

) -> Result<Vec<f32>, ocl::Error> {

    let local\_size = 512; // Размер локальной рабочей группы

    let global\_size = ((n + local\_size - 1) / local\_size) \* local\_size; // Глобальный размер для N

    let reduce\_global\_size = ((d + local\_size - 1) / local\_size) \* local\_size; // Глобальный размер для D

    // Создание буферов OpenCL

    let dataset\_buf = Buffer::builder()

        .queue(cl\_ctx.queue.clone())

        .flags(ocl::flags::MEM\_READ\_ONLY | ocl::flags::MEM\_COPY\_HOST\_PTR)

        .len(n \* d)

        .copy\_host\_slice(dataset)

        .build()?;

    let labels\_buf = Buffer::builder()

        .queue(cl\_ctx.queue.clone())

        .flags(ocl::flags::MEM\_READ\_ONLY | ocl::flags::MEM\_COPY\_HOST\_PTR)

        .len(n)

        .copy\_host\_slice(labels)

        .build()?;

    let weights\_buf = Buffer::builder()

        .queue(cl\_ctx.queue.clone())

        .flags(ocl::flags::MEM\_READ\_WRITE | ocl::flags::MEM\_COPY\_HOST\_PTR)

        .len(d)

        .copy\_host\_slice(weights)

        .build()?;

    let bias\_buf = Buffer::builder()

        .queue(cl\_ctx.queue.clone())

        .flags(ocl::flags::MEM\_READ\_WRITE | ocl::flags::MEM\_COPY\_HOST\_PTR)

        .len(1)

        .copy\_host\_slice(&[\*bias])

        .build()?;

    let predictions\_buf = Buffer::<f32>::builder()

        .queue(cl\_ctx.queue.clone())

        .flags(ocl::flags::MEM\_WRITE\_ONLY)

        .len(n)

        .build()?;

    let grad\_weights\_buf = Buffer::<f32>::builder()

        .queue(cl\_ctx.queue.clone())

        .flags(ocl::flags::MEM\_WRITE\_ONLY)

        .len(n \* d)

        .build()?;

    let grad\_bias\_buf = Buffer::<f32>::builder()

        .queue(cl\_ctx.queue.clone())

        .flags(ocl::flags::MEM\_WRITE\_ONLY)

        .len(n)

        .build()?;

    // Цикл обучения

    for \_ in 0..EPOCHS {

        // Прямой проход

        cl\_ctx.forward\_kernel.set\_arg(0, &dataset\_buf)?;

        cl\_ctx.forward\_kernel.set\_arg(1, &weights\_buf)?;

        cl\_ctx.forward\_kernel.set\_arg(2, &bias\_buf)?;

        cl\_ctx.forward\_kernel.set\_arg(3, &predictions\_buf)?;

        cl\_ctx.forward\_kernel.set\_arg(4, d as u32)?;

        unsafe {

            cl\_ctx

                .forward\_kernel

                .cmd()

                .global\_work\_size(global\_size)

                .local\_work\_size(local\_size)

                .enq()?;

        }

        // Вычисление градиентов

        cl\_ctx.gradient\_kernel.set\_arg(0, &dataset\_buf)?;

        cl\_ctx.gradient\_kernel.set\_arg(1, &labels\_buf)?;

        cl\_ctx.gradient\_kernel.set\_arg(2, &predictions\_buf)?;

        cl\_ctx.gradient\_kernel.set\_arg(3, &grad\_weights\_buf)?;

        cl\_ctx.gradient\_kernel.set\_arg(4, &grad\_bias\_buf)?;

        cl\_ctx.gradient\_kernel.set\_arg(5, d as u32)?;

        cl\_ctx.gradient\_kernel.set\_arg(6, n as u32)?;

        unsafe {

            cl\_ctx

                .gradient\_kernel

                .cmd()

                .global\_work\_size(global\_size)

                .local\_work\_size(local\_size)

                .enq()?;

        }

        // Редукция градиентов и обновление весов

        cl\_ctx.reduce\_kernel.set\_arg(0, &grad\_weights\_buf)?;

        cl\_ctx.reduce\_kernel.set\_arg(1, &weights\_buf)?;

        cl\_ctx.reduce\_kernel.set\_arg(2, &grad\_bias\_buf)?;

        cl\_ctx.reduce\_kernel.set\_arg(3, &bias\_buf)?;

        cl\_ctx.reduce\_kernel.set\_arg(4, d as u32)?;

        cl\_ctx.reduce\_kernel.set\_arg(5, n as u32)?;

        cl\_ctx.reduce\_kernel.set\_arg(6, LEARNING\_RATE)?;

        unsafe {

            cl\_ctx

                .reduce\_kernel

                .cmd()

                .global\_work\_size(reduce\_global\_size)

                .local\_work\_size(local\_size)

                .enq()?;

        }

    }

    // Чтение предсказаний

    let mut predictions = vec![0.0; n];

    predictions\_buf.read(&mut predictions).enq()?;

    // Чтение обновлённых весов и смещения

    weights\_buf.read(weights).enq()?;

    let mut bias\_vec = vec![0.0; 1];

    bias\_buf.read(&mut bias\_vec).enq()?;

    \*bias = bias\_vec[0];

    // Вывод первых 10 предсказаний

    println!("Предсказания на обучающем наборе (GPU):");

    for i in 0..10.min(n) {

        println!("Пример {}: {:.6}", i, predictions[i]);

    }

    Ok(predictions)

}

// Вычисление предсказаний на GPU без обучения

fn predict\_gpu(

    cl\_ctx: &OpenCLContext,

    dataset: &[f32],

    weights: &[f32],

    bias: f32,

    n: usize,

    d: usize,

) -> Result<Vec<f32>, ocl::Error> {

    let local\_size = 512; // Размер локальной рабочей группы

    let global\_size = ((n + local\_size - 1) / local\_size) \* local\_size; // Глобальный размер

    // Создание буферов OpenCL

    let dataset\_buf = Buffer::builder()

        .queue(cl\_ctx.queue.clone())

        .flags(ocl::flags::MEM\_READ\_ONLY | ocl::flags::MEM\_COPY\_HOST\_PTR)

        .len(n \* d)

        .copy\_host\_slice(dataset)

        .build()?;

    let weights\_buf = Buffer::builder()

        .queue(cl\_ctx.queue.clone())

        .flags(ocl::flags::MEM\_READ\_ONLY | ocl::flags::MEM\_COPY\_HOST\_PTR)

        .len(d)

        .copy\_host\_slice(weights)

        .build()?;

    let bias\_buf = Buffer::builder()

        .queue(cl\_ctx.queue.clone())

        .flags(ocl::flags::MEM\_READ\_ONLY | ocl::flags::MEM\_COPY\_HOST\_PTR)

        .len(1)

        .copy\_host\_slice(&[bias])

        .build()?;

    let predictions\_buf = Buffer::<f32>::builder()

        .queue(cl\_ctx.queue.clone())

        .flags(ocl::flags::MEM\_WRITE\_ONLY)

        .len(n)

        .build()?;

    // Выполнение прямого прохода

    cl\_ctx.forward\_kernel.set\_arg(0, &dataset\_buf)?;

    cl\_ctx.forward\_kernel.set\_arg(1, &weights\_buf)?;

    cl\_ctx.forward\_kernel.set\_arg(2, &bias\_buf)?;

    cl\_ctx.forward\_kernel.set\_arg(3, &predictions\_buf)?;

    cl\_ctx.forward\_kernel.set\_arg(4, d as u32)?;

    unsafe {

        cl\_ctx

            .forward\_kernel

            .cmd()

            .global\_work\_size(global\_size)

            .local\_work\_size(local\_size)

            .enq()?;

    }

    // Чтение предсказаний

    let mut predictions = vec![0.0; n];

    predictions\_buf.read(&mut predictions).enq()?;

    Ok(predictions)

}

// Измерение времени выполнения в секундах

fn measure\_time<F: FnOnce()>(f: F) -> f64 {

    let start = Instant::now();

    f();

    let duration = start.elapsed();

    duration.as\_secs() as f64 + duration.subsec\_nanos() as f64 / 1\_000\_000\_000.0

}

// Загрузка данных из файла: N, D, dataset (N x D), labels (N)

fn load\_data(filename: &str) -> io::Result<(Vec<f32>, Vec<f32>, usize, usize)> {

    let file = File::open(filename)?;

    let reader = BufReader::new(file);

    let mut lines = reader.lines();

    // Чтение первой строки: N (кол-во примеров), D (размерность)

    let first\_line = lines.next().ok\_or\_else(|| io::Error::new(io::ErrorKind::InvalidData, "Empty file"))??;

    let dims: Vec<usize> = first\_line

        .split\_whitespace()

        .map(|s| s.parse().unwrap())

        .collect();

    let n = dims[0];

    let d = dims[1];

    let mut dataset = vec![0.0; n \* d]; // Матрица данных

    let mut labels = vec![0.0; n]; // Метки

    // Чтение данных и меток

    for (i, line) in lines.enumerate() {

        let values: Vec<f32> = line?

            .split\_whitespace()

            .map(|s| s.parse().unwrap())

            .collect();

        for j in 0..d {

            dataset[i \* d + j] = values[j];

        }

        labels[i] = values[d];

    }

    Ok((dataset, labels, n, d))

}

// Основная функция: обучение, сравнение CPU и GPU, проверка результатов

fn main() -> Result<(), Box<dyn std::error::Error>> {

    // Загрузка данных

    let (dataset, labels, n, d) = load\_data("data.txt")?;

    // Обучение на CPU

    let mut weights\_cpu = vec![0.0; d];

    let mut bias\_cpu = 0.0;

    let cpu\_time = measure\_time(|| {

        train\_cpu(&dataset, &labels, &mut weights\_cpu, &mut bias\_cpu, n, d);

    });

    // Обучение на GPU

    let cl\_ctx = OpenCLContext::new()?;

    let mut weights\_gpu = vec![0.0; d];

    let mut bias\_gpu = 0.0;

    let gpu\_time = measure\_time(|| {

        let predictions = run\_kernels(&cl\_ctx, &dataset, &labels, &mut weights\_gpu, &mut bias\_gpu, n, d);

        predictions.expect("GPU kernels failed");

    });

    // Вывод времени и ускорения

    println!("Затраты CPU: {:.6} секунд", cpu\_time);

    println!("Затраты GPU: {:.6} секунд", gpu\_time);

    println!("Ускорение: {:.2}x", cpu\_time / gpu\_time);

    // Вывод весов и смещения

    println!("Веса GPU: {:?}", &weights\_gpu[..10.min(d)]);

    println!("Смещение GPU: {:.6}", bias\_gpu);

    // Проверка предсказаний CPU

    let mut predictions\_cpu = vec![0.0; n];

    calculate\_cpu(&dataset, &labels, &weights\_cpu, bias\_cpu, n, d, &mut predictions\_cpu);

    // Проверка предсказаний GPU

    let predictions\_gpu = predict\_gpu(&cl\_ctx, &dataset, &weights\_gpu, bias\_gpu, n, d)?;

    // Сравнение CPU и GPU

    println!("Проверка результатов (CPU vs GPU):");

    let mut correct = true;

    for i in 0..10.min(n) {

        println!(

            "Пример {}: CPU={:.6}, GPU={:.6}",

            i, predictions\_cpu[i], predictions\_gpu[i]

        );

        if (predictions\_cpu[i] - predictions\_gpu[i]).abs() > EPSILON {

            correct = false;

        }

    }

    println!("Проверка: {}", if correct { "OK" } else { "ОШИБКА" });

    // Вычисление MSE для GPU

    let mse: f32 = predictions\_gpu

        .iter()

        .zip(labels.iter())

        .map(|(p, l)| (p - l).powi(2))

        .sum::<f32>()

        / n as f32;

    println!("MSE на обучающем наборе (GPU): {:.6}", mse);

    // Тестовое предсказание

    let mut test\_point = vec![0.0; d];

    for i in 0..d {

        test\_point[i] = (i + 1) as f32;

    }

    let mut test\_pred = 0.0;

    for i in 0..d {

        test\_pred += weights\_gpu[i] \* test\_point[i].sin();

    }

    test\_pred += bias\_gpu;

    println!("Предсказание на тестовом примере: {:.6}", test\_pred);

    Ok(())

}

Компиляция и запуск:

Реализация Dataset на Python:

python3 random\_dataset.py

Реализация C:

gcc main.c -o main -lOpenCL -lm

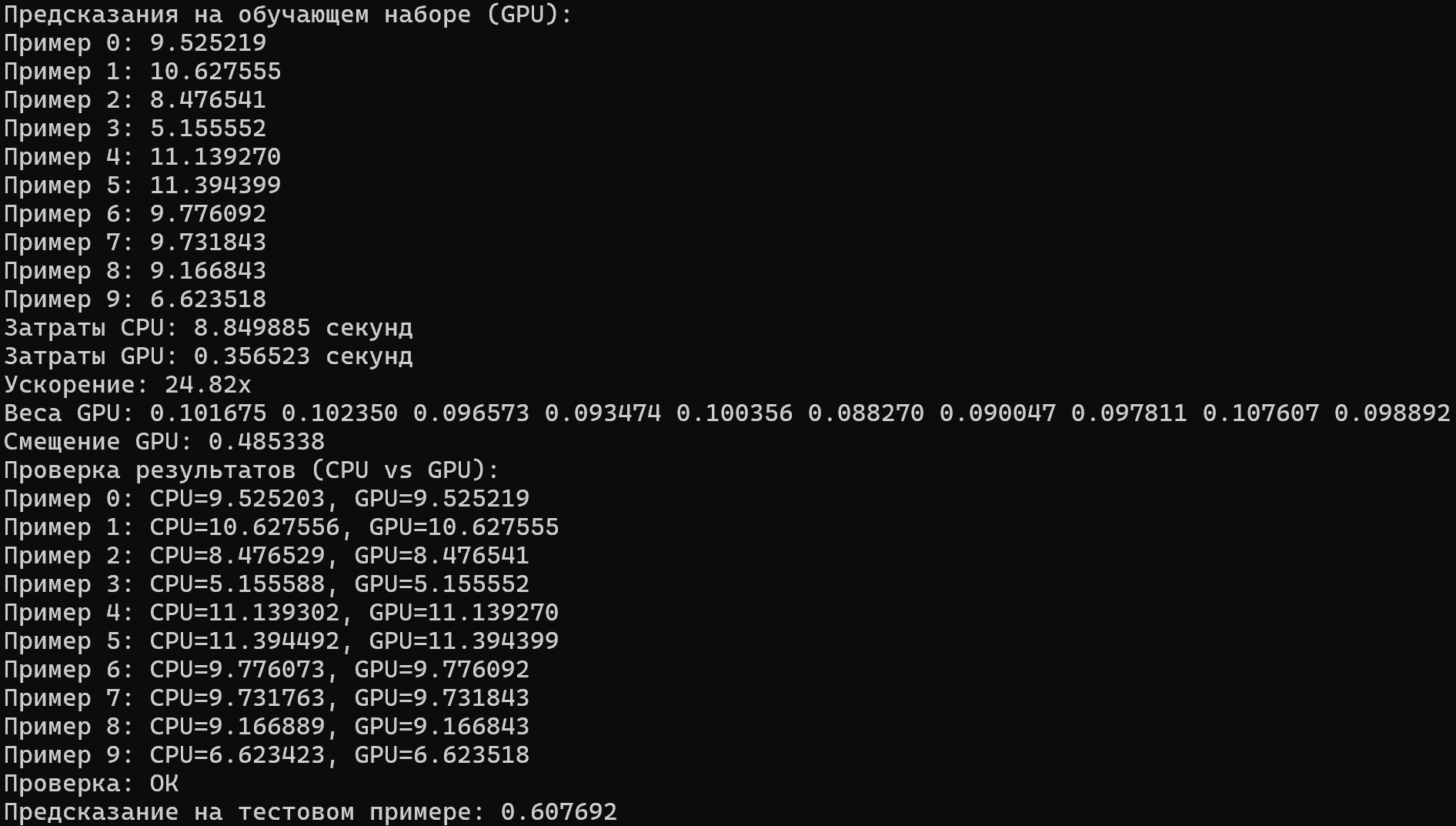
./main

Реализация Rust:

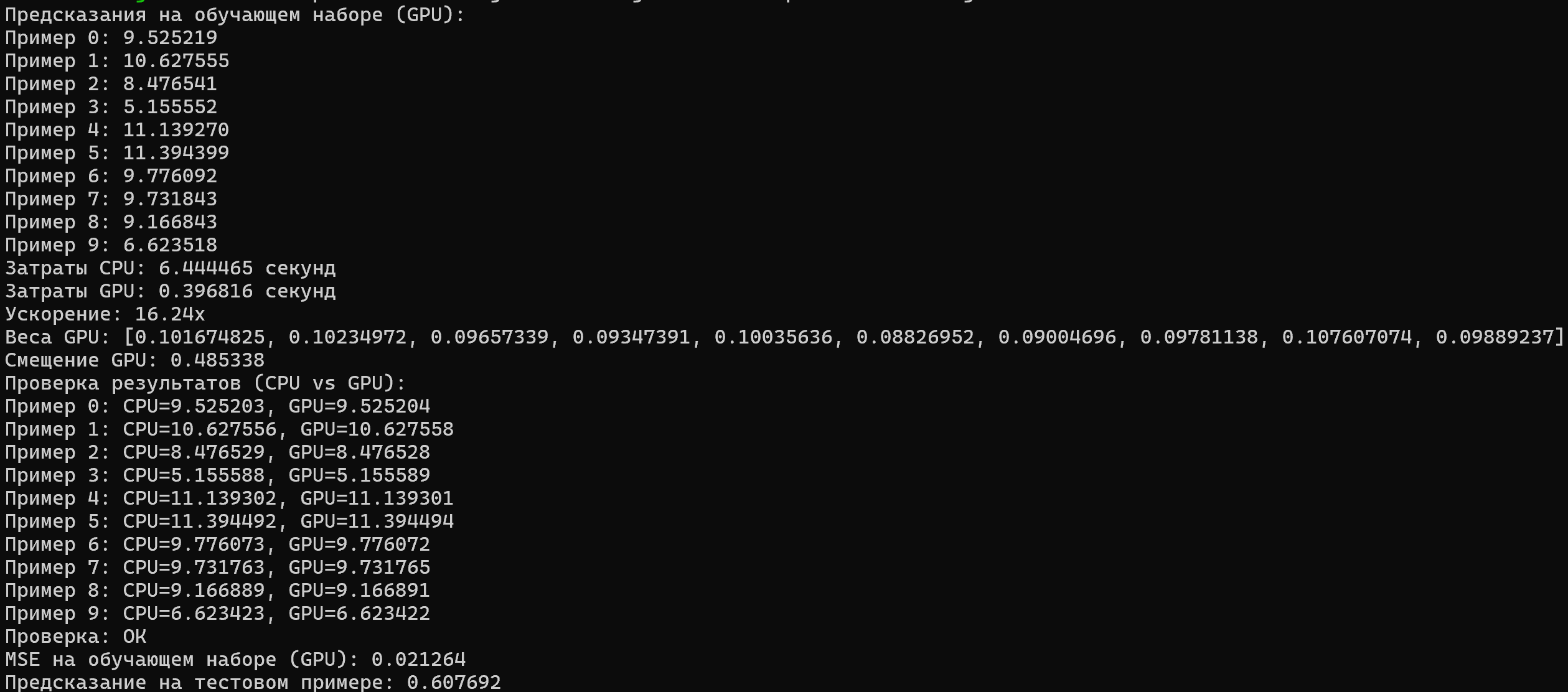
cargo run --release

Работа программы WSL:

C:

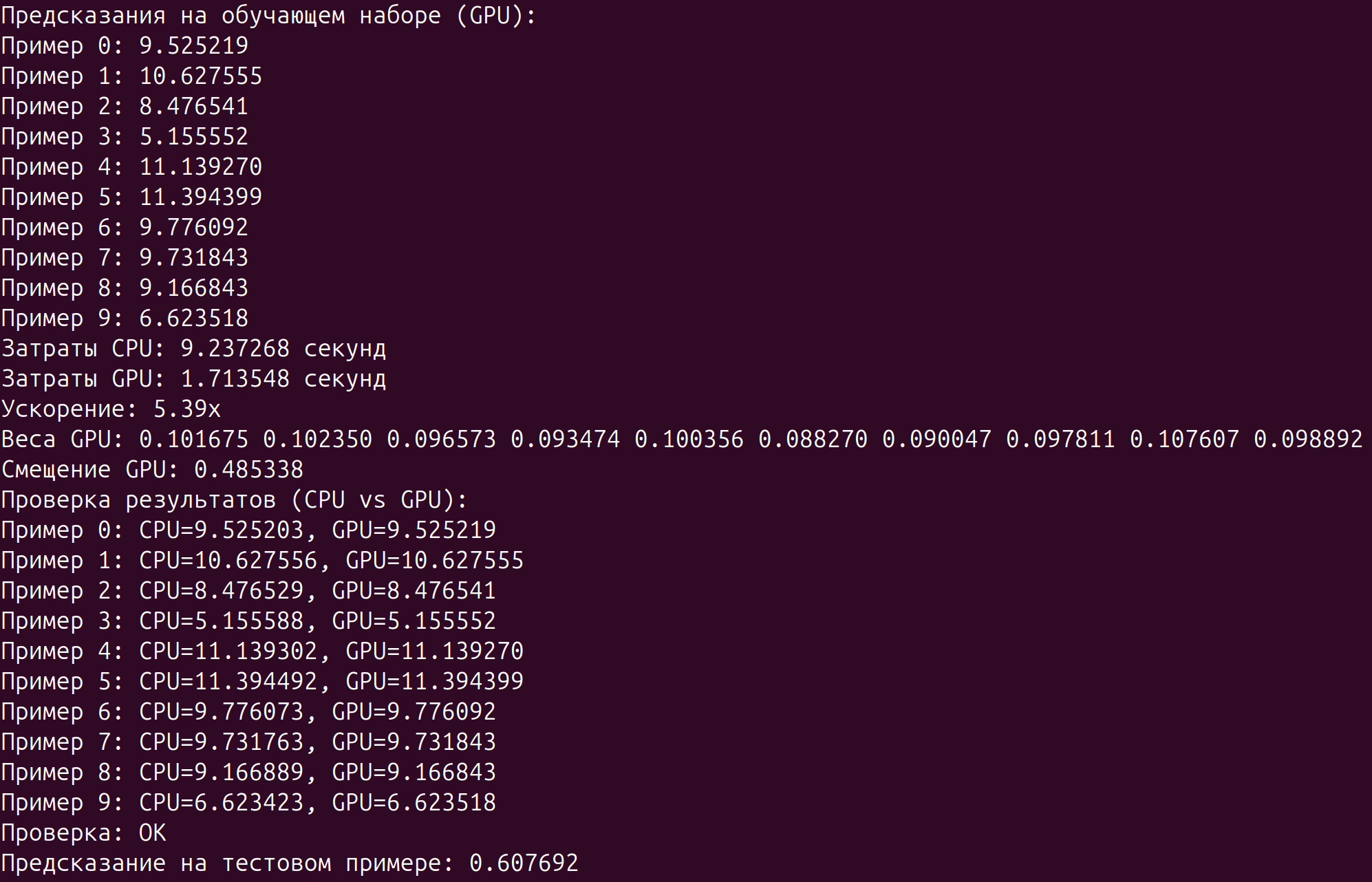


Rust:

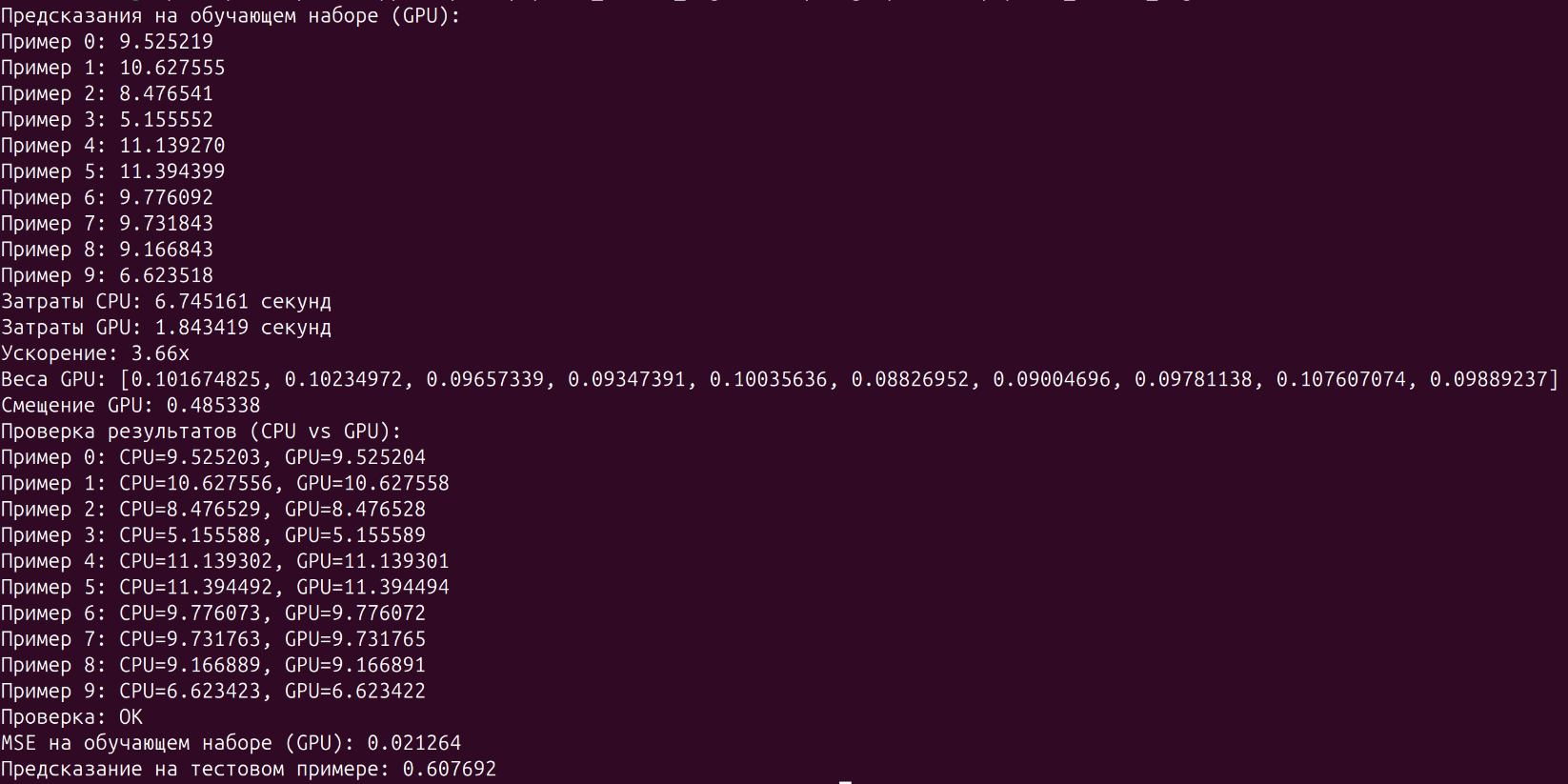


Работа программы Ubuntu

C:



Rust:

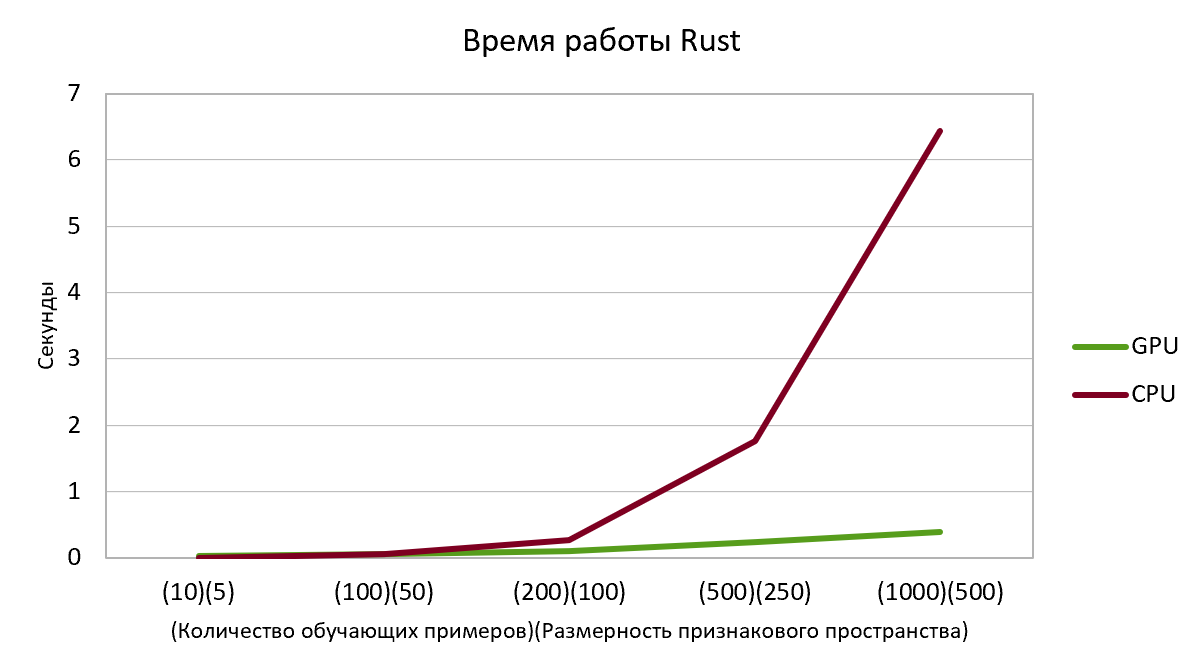


Результаты:

|  |  |  |
| --- | --- | --- |
| Замер времени выполнения, сек | | |
|  | WSL Ubuntu 24.04.2 LTS | Ubuntu 24.04.2 LTS |
| CPU (C) | 8.84985 | 9.237268 |
| CPU (Rust) | 6.444465 | 6.745161 |
| GPU (C) | 0.356523 | 1.7135448 |
| GPU (Rust) | 0.396816 | 1.843419 |

|  |  |  |
| --- | --- | --- |
| Сравнение результатов (≈) | | |
|  | WSL Ubuntu 24.04.2 LTS | Ubuntu 24.04.2 LTS |
| C | 25x | 5x |
| Rust | 16x | 4x |

|  |  |  |  |
| --- | --- | --- | --- |
| Результаты от размера датасета (Rust) | | | |
| N | D | CPU, сек | GPU, сек |
| 10 | 5 | 0.000283 | 0.035941 |
| 100 | 50 | 0.050700 | 0.064288 |
| 200 | 100 | 0.267465 | 0.101611 |
| 500 | 250 | 1.767052 | 0.232731 |
| 1000 | 500 | 6.444465 | 0.396816 |



**Вывод:** Программа реализует линейную регрессию с использованием OpenCL на GPU, демонстрируя ускорение 16x по сравнению с последовательной реализацией на CPU, что подтверждает преимущество параллельных вычислений, хотя результат ниже ожидаемого 25x из C-версии из-за возможных накладных расходов. Предсказания на обучающем наборе и тестовое предсказание стабильны и совпадают с C-версией, проверка корректности пройдена (OK), а MSE = 0.021264 указывает на хорошее качество модели.

Датасет использовался размером: 1000 — количество обучающих примеров, 500 — размерность признакового пространства. На графике видна огромная зависимость от размера датасета. Время затраченное CPU растёт экспоненциально, а GPU приближено к линейному.

Уменьшение скорости выполнения на Ubuntu вероятнее всего связано с новым графическим ядром процессора intel ultra 5 125h. Единственный видеодрайвер для Ubuntu датируется 09.01.2025.