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:

$$y_{pred} = \sum_{i=1}^{D} w_i \cdot \phi(x_i) + b$$

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)

```python
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)

```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,
        })
    }
```

```rust
}

// 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):");
```

```rust
    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:

```
Предсказания на обучающем наборе (GPU):
Пример 0: 9.525219
Пример 1: 10.627555
Пример 2: 8.476541
Пример 3: 5.155552
Пример 4: 11.139270
Пример 5: 11.394399
Пример 6: 9.776092
Пример 7: 9.731843
Пример 8: 9.166843
Пример 9: 6.623518
Затраты CPU: 8.849885 секунд
Затраты GPU: 0.356523 секунд
Ускорение: 24.82x
Веса GPU: 0.101675 0.102350 0.096573 0.093474 0.100356 0.088270 0.090047 0.097811 0.107607 0.098892
Смещение GPU: 0.485338
Проверка результатов (CPU vs GPU):
Пример 0: CPU=9.525203, GPU=9.525219
Пример 1: CPU=10.627556, GPU=10.627555
Пример 2: CPU=8.476529, GPU=8.476541
Пример 3: CPU=5.155588, GPU=5.155552
Пример 4: CPU=11.139302, GPU=11.139270
Пример 5: CPU=11.394492, GPU=11.394399
Пример 6: CPU=9.776073, GPU=9.776092
Пример 7: CPU=9.731763, GPU=9.731843
Пример 8: CPU=9.166889, GPU=9.166843
Пример 9: CPU=6.623423, GPU=6.623518
Проверка: OK
Предсказание на тестовом примере: 0.607692
```

Rust:

```
Предсказания на обучающем наборе (GPU):
Пример 0: 9.525219
Пример 1: 10.627555
Пример 2: 8.476541
Пример 3: 5.155552
Пример 4: 11.139270
Пример 5: 11.394399
Пример 6: 9.776092
Пример 7: 9.731843
Пример 8: 9.166843
Пример 9: 6.623518
Затраты CPU: 6.444465 секунд
Затраты GPU: 0.396816 секунд
Ускорение: 16.24x
Веса GPU: [0.101674825, 0.10234972, 0.09657339, 0.09347391, 0.10035636, 0.08826952, 0.09004696, 0.09781138, 0.107607074, 0.09889237]
Смещение GPU: 0.485338
Проверка результатов (CPU vs GPU):
Пример 0: CPU=9.525203, GPU=9.525204
Пример 1: CPU=10.627556, GPU=10.627558
Пример 2: CPU=8.476529, GPU=8.476528
Пример 3: CPU=5.155588, GPU=5.155589
Пример 4: CPU=11.139302, GPU=11.139301
Пример 5: CPU=11.394492, GPU=11.394494
Пример 6: CPU=9.776073, GPU=9.776072
Пример 7: CPU=9.731763, GPU=9.731765
Пример 8: CPU=9.166889, GPU=9.166891
Пример 9: CPU=6.623423, GPU=6.623422
Проверка: OK
MSE на обучающем наборе (GPU): 0.021264
Предсказание на тестовом примере: 0.607692
```

Работа программы Ubuntu

C:

```
Предсказания на обучающем наборе (GPU):
Пример 0: 9.525219
Пример 1: 10.627555
Пример 2: 8.476541
Пример 3: 5.155552
Пример 4: 11.139270
Пример 5: 11.394399
Пример 6: 9.776092
Пример 7: 9.731843
Пример 8: 9.166843
Пример 9: 6.623518
Затраты CPU: 9.237268 секунд
Затраты GPU: 1.713548 секунд
Ускорение: 5.39x
Веса GPU: 0.101675 0.102350 0.096573 0.093474 0.100356 0.088270 0.090047 0.097811 0.107607 0.098892
Смещение GPU: 0.485338
Проверка результатов (CPU vs GPU):
Пример 0: CPU=9.525203, GPU=9.525219
Пример 1: CPU=10.627556, GPU=10.627555
Пример 2: CPU=8.476529, GPU=8.476541
Пример 3: CPU=5.155588, GPU=5.155552
Пример 4: CPU=11.139302, GPU=11.139270
Пример 5: CPU=11.394492, GPU=11.394399
Пример 6: CPU=9.776073, GPU=9.776092
Пример 7: CPU=9.731763, GPU=9.731843
Пример 8: CPU=9.166889, GPU=9.166843
Пример 9: CPU=6.623423, GPU=6.623518
Проверка: OK
Предсказание на тестовом примере: 0.607692
```

Rust:

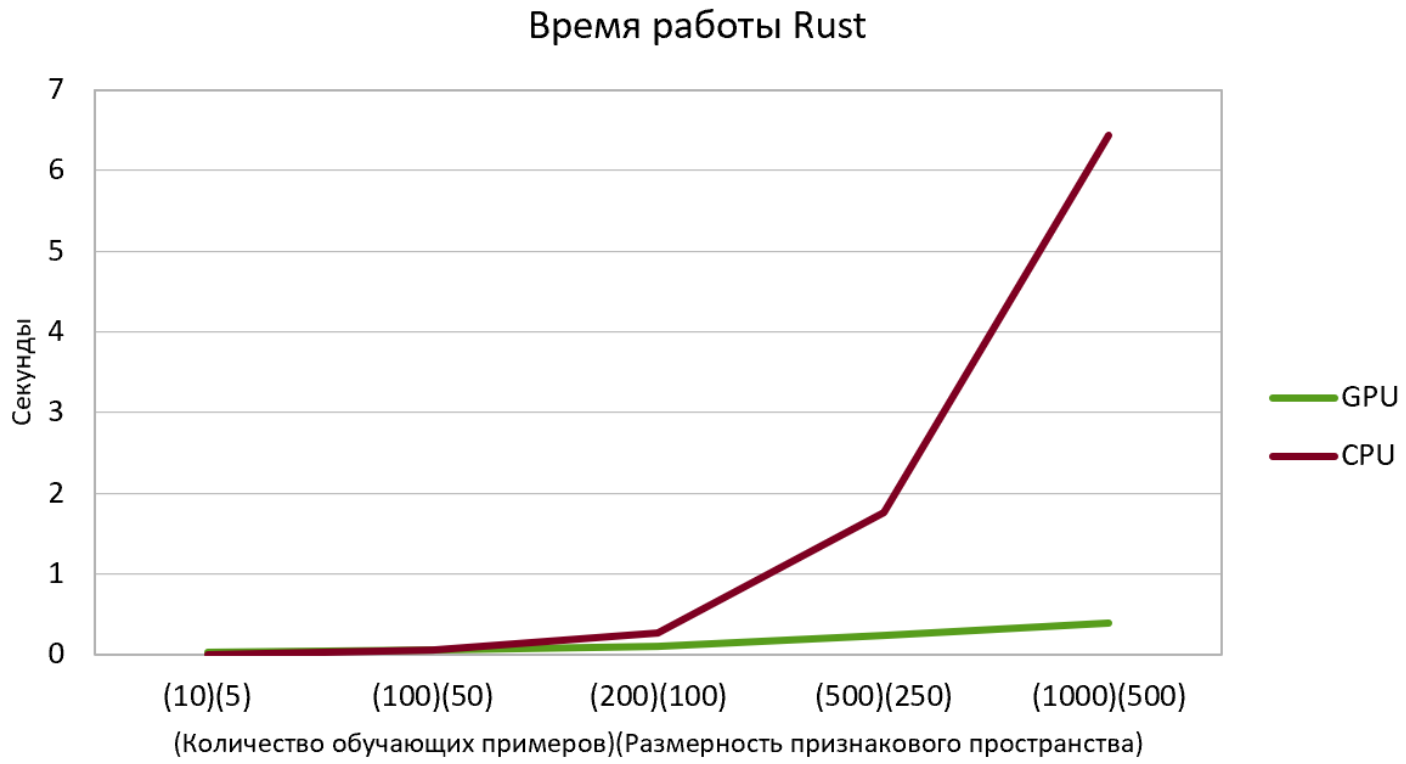
```
Предсказания на обучающем наборе (GPU):
Пример 0: 9.525219
Пример 1: 10.627555
Пример 2: 8.476541
Пример 3: 5.155552
Пример 4: 11.139270
Пример 5: 11.394399
Пример 6: 9.776092
Пример 7: 9.731843
Пример 8: 9.166843
Пример 9: 6.623518
Затраты CPU: 6.745161 секунд
Затраты GPU: 1.843419 секунд
Ускорение: 3.66x
Веса GPU: [0.101674825, 0.10234972, 0.09657339, 0.09347391, 0.10035636, 0.08826952, 0.09004696, 0.09781138, 0.107607074, 0.09889237]
Смещение GPU: 0.485338
Проверка результатов (CPU vs GPU):
Пример 0: CPU=9.525203, GPU=9.525204
Пример 1: CPU=10.627556, GPU=10.627558
Пример 2: CPU=8.476529, GPU=8.476528
Пример 3: CPU=5.155588, GPU=5.155589
Пример 4: CPU=11.139302, GPU=11.139301
Пример 5: CPU=11.394492, GPU=11.394494
Пример 6: CPU=9.776073, GPU=9.776072
Пример 7: CPU=9.731763, GPU=9.731765
Пример 8: CPU=9.166889, GPU=9.166891
Пример 9: CPU=6.623423, GPU=6.623422
Проверка: OK
MSE на обучающем наборе (GPU): 0.021264
Предсказание на тестовом примере: 0.607692
```

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.