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
Пахомов Владислав Андреевич

Проверили:
доц. Островский Алексей Мичеславович

Белгород 2025 г.

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.

Условие индивидуального задания:

Реализовать параллельные алгоритмы для обучения и предсказания с помощью логистической регрессии. Дан набор данных dataset[N][D], где N — количество обучающих примеров, D — размерность признакового пространства (загрузка из файла). Даны метки labels[N] (выходные значения 0 или 1). Модель: логистическая регрессия с функцией активации sigmoid. Предсказание вычисляется по формуле:

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

где w_i — веса модели, b — смещение. Требуется: реализовать параллельный прямой проход (forward pass), где 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. После обучения вывести предсказания на обучающем наборе данных и на новом тестовом примере. Сравнить производительность (по времени выполнения) между реализациями на GPU и CPU. Все данные использовать в формате float. Загрузка данных из файла.

Ход выполнения работы

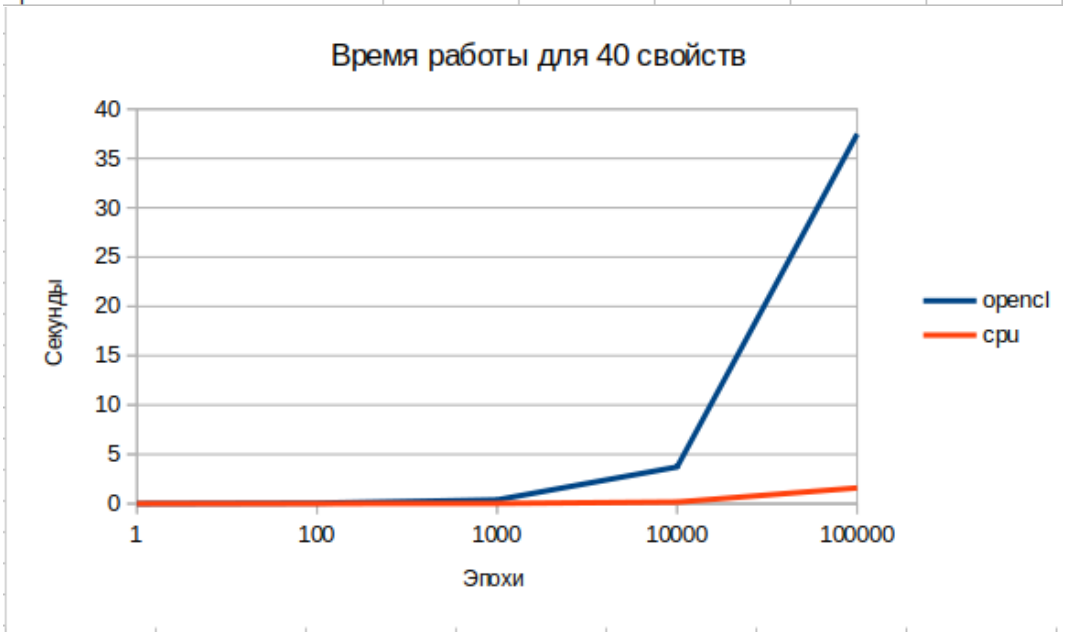
Описание архитектурных решений Шейдер для вычисления предсказаний использует разбиения по группам для просчёта предсказания для одного объекта. Размер группы представляет собой количество свойств объекта наиболее близкое к 2^N (в большую сторону). В конце используется редукция сумма для получения итогового предсказания. Шейдер может работать как со множествами объектами, так и с одним объектом.

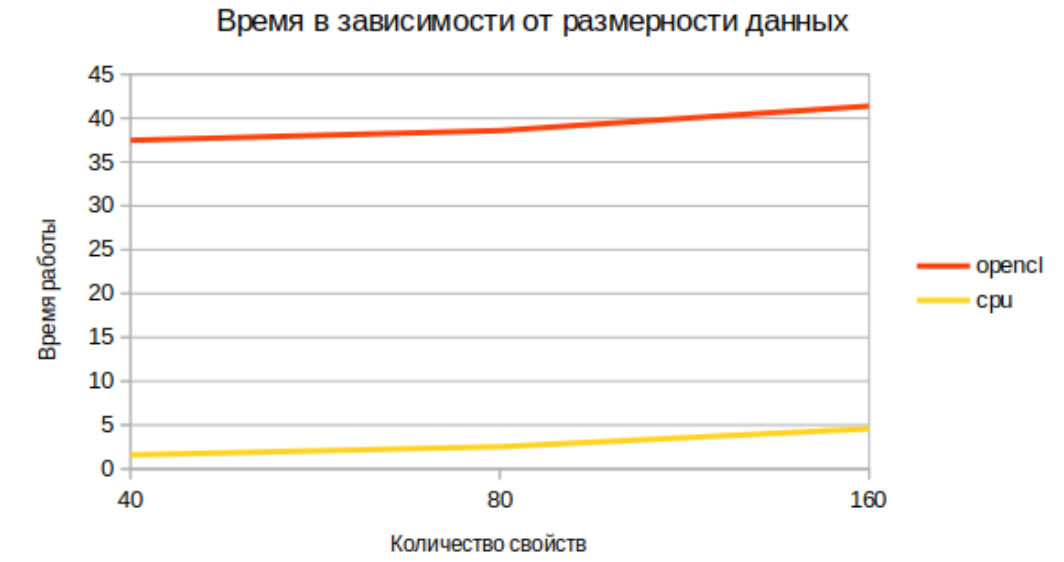
Шейдер для перерасчёта смещения и весов использует несколько другую композицию. В нём в группе объединяется один признак объекта, a local_id представляет собой номер объекта. Соответственно размер группы будет количеством объектов наиболее близкое к 2^N (в большую) сторону. В конце производится редукция градиентов и производится перерасчёт весов. А также производится редукция для смещения и производится его перерасчёт.

Для пересчёта градиентов использовался другой подход, более подходящий под задачу параллелизации. (Asynchronous SGD) Так, в примере градиент пересчитывался каждый раз с новыми данными и веса обновлялись на лету. При новом подходе градиенты считаются для всех предсказаний сразу и редуцируются. Такой подход может привести к потенциально худшим результатам, однако позволяет распараллелить задачу.

Графики ускорения и зависимостей. Ниже приведено время выполнения программы при разных количествах эпох и свойств:

	Эпохи				
Устройство	1	100	1000	10000	100000
OpenCL (AMD Radeon 6800 XT)	0,003	0,05	0,382	3,72	37,48
Процессор (Intel Core i5-12600k)	0,00003	0,002	0,02	0,16	1,59
При 40 свойствах					
	Эпохи				
Устройство	1	100	1000	10000	100000
OpenCL (AMD Radeon 6800 XT)	0,003	0,05	0,4	3,87	38,59
Процессор (Intel Core i5-12600k)	0,00005	0,002	0,036	0,253	2,53
При 80 свойствах					
	Эпохи				
Устройство	1	100	1000	10000	100000
OpenCL (AMD Radeon 6800 XT)	0,003	0,06	0,4	4,12	41,38
Процессор (Intel Core i5-12600k)	0,00007	0,004	0,058	0,425	4,58
При 160 свойствах					





Исходный код

```
mod neurals;

use crate::neurals::neural::NeuralNetwork;

const TRAIN_DATA_PATH: &str = "./lab4/datasets/train_data_160";
const LABELS_PATH: &str = "./lab4/datasets/labels";
const EPOCHS: usize = 100000;
const LEARNING_RATE: f32 = 0.1;

fn load_data() -> Result<(Vec<Vec<f32>>, Vec<f32>), String> {
    // Загрузить объекты
    let mut result_train_data: Vec<Vec<f32>> = vec![];
    for train_data in std::fs::read_to_string(TRAIN_DATA_PATH)
        .expect("train data file is not accessible")
        .lines()
    {
        let mut train_object: Vec<f32> = vec![];
        for prop in train_data.split(" ") {
            train_object.push(prop.parse::<f32>().expect("invalid float variable in train data file"));
```

```
        }

        if result_train_data.len() > 0 {
            if result_train_data[0].len() != train_object.len() {
                return Err(format!("objects have different props amount"));
            }
        }

        result_train_data.push(train_object);
    }

    // Загрузить метки
    let mut result_labels: Vec<f32> = vec![];
    for labels_data in std::fs::read_to_string(LABELS_PATH)
        .expect("label lines file is not accessible")
        .lines()
    {
        result_labels.push(labels_data.parse::<f32>().expect("invalid float variable in labels file"));
    }

    if result_train_data.len() != result_labels.len() {
        return Err(format!("train data and labels amounts are different"));
    }

    Ok((result_train_data, result_labels))
}

fn main() {
    let (train_data, train_labels) = load_data().expect("invalid dataset");

    // Тренировка при помощи GPU
    let mut nn = neurals::opencl::OpenCLNeuralNetwork::new().expect("failed GPU Accelerated train");
    nn.fit(&train_data, &train_labels, EPOCHS, LEARNING_RATE)
        .expect("failed GPU Accelerated train");

    // Тренировка при помощи CPU
    let mut nn = neurals::cpu::CPUNeuralNetwork::new().expect("failed CPU train");
    nn.fit(&train_data, &train_labels, EPOCHS, LEARNING_RATE)
        .expect("failed CPU train");
}
```

```
/// Описывает нейронную сеть
pub trait NeuralNetwork {
    /// Обучает нейронную сеть, пересчитывая веса и смещение.
    /// После обучения для предсказания принимает данные только
    /// той же размерности, что и тренировочный датасет.
    ///
    /// * `train_data` - тренировочный датасет.
    /// * `labels` - метки.
    /// * `epochs` - количество итераций для обучения.
    /// * `learning_rate` - коэффициент скорости обучения
    ///
```

```rust
    /// Возвращает обученную нейронную сеть или ошибку.
    fn fit(
        &mut self,
        train_data: &Vec<Vec<f32>>,
        labels: &Vec<f32>,
        epochs: usize,
        learning_rate: f32,
    ) -> Result<&impl NeuralNetwork, String>;

    /// Выполняет предсказание на основе раннее выполненного обучения
    /// (смотри метод `fit`).
    ///
    /// * `train_data` - набор свойств, для которых необходимо выполнить обучение.
    ///
    /// Возвращает предсказание для заданного набора свойств.
    fn predict(&self, train_data: &Vec<Vec<f32>>) -> Result<Vec<f32>, String>;
}
```

```rust
use crate::neurals::neural::NeuralNetwork;
use rand::Rng;
use std::time::Instant;

pub struct CPUNeuralNetwork {
    _weights: Vec<f32>,
    _bias: [f32; 1],
}

impl CPUNeuralNetwork {
    /// Конструирует нейронную сеть с поддержкой процессора.
    pub fn new() -> Result<CPUNeuralNetwork, String> {
        let mut rng = rand::rng();

        let result = CPUNeuralNetwork {
            _bias: [rng.random_range(-1.0..1.0)],
            _weights: vec![],
        };

        Ok(result)
    }

    fn _sigmoid(x: f32) -> f32 {
        1. / (1. + (-x).exp())
    }
}

impl NeuralNetwork for CPUNeuralNetwork {
    fn fit(
        &mut self,
        train_data: &Vec<Vec<f32>>,
        labels: &Vec<f32>,
        epochs: usize,
        learning_rate: f32,
    ) -> Result<&impl NeuralNetwork, String> {
```

```
if train_data.len() == 0 {
    return Err(format!("no data to train provided"));
}
if train_data.len() != labels.len() {
    return Err(format!("train and label lengths are different"));
}

// Обновить веса
let mut rng = rand::rng();

self._weights = vec![0.0; train_data[0].len()];
for weight_index in 0..self._weights.len() {
    self._weights[weight_index] = rng.random_range(-1.0..1.0);
}

let now = Instant::now();
println!("Starting CPU only training\n\
          ================================");

for epoch in 0..epochs {
    let mut weights_new = self._weights.clone();
    let mut bias_new = self._bias[0];

    for dataset_index in 0..train_data.len() {
        let y_pred = self.predict(&vec![train_data[dataset_index].clone()])?[0];
        let error = labels[dataset_index] - y_pred;

        for prop_index in 0..train_data[dataset_index].len() {
            weights_new[prop_index] += learning_rate * (
                error * y_pred * (1f32 - y_pred) * train_data[dataset_index][prop_index]
            );
        }
        bias_new += learning_rate * error * y_pred * (1f32 - y_pred);
    }

    self._weights = weights_new;
    self._bias[0] = bias_new;

    if epoch % 10000 == 0 {
        println!("Current epoch: {}\n\
        Current weights: {:?}\n\
        Current bias: {}", epoch, self._weights, self._bias[0]);
    }
}

println!("================================\n\
GPU Accelerated train completed!\n\
  Total epochs: {}\n\
  Result weights: {:?}\n\
  Result bias: {:?}\n\
  Elapsed time: {:.2?}
",
epochs, self._weights, self._bias[0], now.elapsed());
```

```
        Ok(self)
    }

    fn predict(&self, data: &Vec<Vec<f32>>) -> Result<Vec<f32>, String> {
        if data.len() == 0 {
            return Ok([].to_vec());
        }

        if self._weights.len() != data[0].len() {
            return Err(format!(
                "data and train set props lengths ({} and {}) are different",
                data.len(),
                self._weights.len()
            ));
        }

        let mut results = vec![0f32; data.len()];
        for dataset_index in 0..data.len() {
            for prop_index in 0..data[dataset_index].len() {
                results[dataset_index] += data[dataset_index][prop_index] * self._weights[prop_index];
            }

            results[dataset_index] = CPUNeuralNetwork::_sigmoid(results[dataset_index] + self._bias[0]);
        }

        Ok(results)
    }
}
```

```
use opencl3::command_queue::{CommandQueue, CL_QUEUE_PROFILING_ENABLE};
use opencl3::context::Context;
use opencl3::device::{get_all_devices, Device, CL_DEVICE_TYPE_GPU};
use opencl3::kernel::{ExecuteKernel, Kernel};
use opencl3::memory::{Buffer, CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY};
use opencl3::program::Program;
use opencl3::types::{cl_event, cl_float, CL_BLOCKING, CL_NON_BLOCKING};
use rand::Rng;
use std::ptr;
use crate::neurals::neural::NeuralNetwork;
use std::time::Instant;

const PREDICTION_COMP_SHADER_PATH: &str = "./lab4/shaders/prediction.cl";
const PREDICTION_COMP_SHADER_NAME: &str = "prediction";
const RECALC_WEIGHTS_COMP_SHADER_PATH: &str = "./lab4/shaders/recalc_weights.cl";
const RECALC_WEIGHTS_COMP_SHADER_NAME: &str = "recalc_weights";

/// Структура для нейронного обучения с
/// помощью OpenCL устройства
pub struct OpenCLNeuralNetwork {
    _weights: Vec<f32>,
    _bias: [f32; 1],
    _queue: CommandQueue,
    _context: Context,
```

```
    _program_prediction: Program,
    _kernel_prediction: Kernel,
    _program_recalc_weights: Program,
    _kernel_recalc_weights: Kernel,
}

impl OpenCLNeuralNetwork {
    /// Конструирует нейронную сеть с поддержкой OpenCL устройства.
    /// Подключает устройства, компилирует шейдеры.
    /// Если успешно, возвращает нейронную сеть,
    /// иначе - ошибку
    pub fn new() -> Result<OpenCLNeuralNetwork, String> {
        // Найти устройство
        let device_id = *get_all_devices(CL_DEVICE_TYPE_GPU)?
            .first()
            .expect("no device found in platform");
        let device = Device::new(device_id);

        // Создать контекст
        let context = Context::from_device(&device)?;

        // Создать очередь команд
        let queue = CommandQueue::create_default(&context, CL_QUEUE_PROFILING_ENABLE)?;

        // Загрузить вычислительные шейдеры
        let program_prediction = Program::create_and_build_from_source(
            &context,
            std::fs::read_to_string(PREDICTION_COMP_SHADER_PATH)
                .expect("can't open file for prediction shader")
                .as_str(),
            "",
        )?;
        let kernel_prediction = Kernel::create(&program_prediction, PREDICTION_COMP_SHADER_NAME)?;
        let program_recalc_weights = Program::create_and_build_from_source(
            &context,
            std::fs::read_to_string(RECALC_WEIGHTS_COMP_SHADER_PATH)
                .expect("can't open file for recalc weights shader")
                .as_str(),
            "",
        )?;
        let kernel_recalc_weights =
            Kernel::create(&program_recalc_weights, RECALC_WEIGHTS_COMP_SHADER_NAME)?;

        let mut rng = rand::thread_rng();

        let result = OpenCLNeuralNetwork {
            _bias: [rng.random_range(-1.0..1.0)],
            _weights: vec![],
            _queue: queue,
            _context: context,
            _program_prediction: program_prediction,
            _kernel_prediction: kernel_prediction,
            _program_recalc_weights: program_recalc_weights,
            _kernel_recalc_weights: kernel_recalc_weights,
```

```
    };

    Ok(result)
}

fn _predict(&self, data: &Vec<Vec<f32>>, train_data_aligned: &Vec<f32> ) -> Result<Vec<f32>, String> {
    if data.len() == 0 {
        return Ok([].to_vec());
    }

    if self._weights.len() != data[0].len() {
        return Err(format!(
            "data and train set props lengths ({} and {}) are different",
            data.len(),
            self._weights.len()
        ));
    }

    let nodes_amount: u32 = self._weights.len() as u32;

    let mut oclbuf_input_x = unsafe {
        Buffer::<cl_float>::create(
            &self._context,
            CL_MEM_READ_ONLY,
            data[0].len() * data.len(),
            ptr::null_mut(),
        )?
    };
    let mut oclbuf_input_weights = unsafe {
        Buffer::<cl_float>::create(
            &self._context,
            CL_MEM_READ_ONLY,
            self._weights.len(),
            ptr::null_mut(),
        )?
    };
    let oclbuf_predictions = unsafe {
        Buffer::<cl_float>::create(
            &self._context,
            CL_MEM_WRITE_ONLY,
            data.len(),
            ptr::null_mut(),
        )?
    };

    let _writing_event = unsafe {
        self._queue.enqueue_write_buffer(
            &mut oclbuf_input_x,
            CL_NON_BLOCKING,
            0,
            &train_data_aligned,
            &[],
        )?
    };
```

```
        let _writing_event = unsafe {
            self._queue.enqueue_write_buffer(
                &mut oclbuf_input_weights,
                CL_NON_BLOCKING,
                0,
                &self._weights,
                &[],
            )?
        };

        let kernel_event = unsafe {
            ExecuteKernel::new(&self._kernel_prediction)
                .set_arg(&oclbuf_input_x)
                .set_arg(&oclbuf_input_weights)
                .set_arg(&self._bias[0])
                .set_arg(&nodes_amount)
                .set_arg(&oclbuf_predictions)
                .set_arg_local_buffer((nodes_amount.next_power_of_two() as usize) * size_of::<f32>())
                .set_local_work_size(nodes_amount.next_power_of_two() as usize)
                .set_global_work_size((nodes_amount.next_power_of_two() as usize) * data.len())
                .enqueue_nd_range(&self._queue)?
        };

        let mut events: Vec<cl_event> = Vec::default();
        events.push(kernel_event.get());

        let mut predictions = vec![0.0 as cl_float; data.len()];
        let read_event = unsafe {
            self._queue.enqueue_read_buffer(
                &oclbuf_predictions,
                CL_BLOCKING,
                0,
                &mut predictions,
                &events,
            )?
        };

        read_event.wait()?;

        Ok(predictions)
    }
}

impl NeuralNetwork for OpenCLNeuralNetwork {
    fn fit(
        &mut self,
        train_data: &Vec<Vec<f32>>,
        labels: &Vec<f32>,
        epochs: usize,
        learning_rate: f32,
    ) -> Result<&impl NeuralNetwork, String> {
        if train_data.len() == 0 {
            return Err(format!("no data to train provided"));
```

```
    }
    if train_data.len() != labels.len() {
        return Err(format!("train and label lengths are different"));
    }

    // Обновить веса
    let mut rng = rand::thread_rng();

    self._weights = vec![0.0; train_data[0].len()];
    for weight_index in 0..self._weights.len() {
        self._weights[weight_index] = rng.random_range(-1.0..1.0);
    }

    let mut train_data_aligned = vec![0f32; train_data[0].len() * train_data.len()];
    for i in 0..train_data.len() {
        for j in 0..train_data[0].len() {
            train_data_aligned[j + i * train_data[0].len()] = train_data[i][j];
        }
    }

    let objects_amount: u32 = train_data.len() as u32;

    let now = Instant::now();
    println!("Starting GPU Accelerated training\n\
              ================================");

    for epoch in 0..epochs {
        let predictions = self._predict(&train_data, &train_data_aligned)?;

        let mut oclbuf_input_labels = unsafe {
            Buffer::<cl_float>::create(
                &self._context,
                CL_MEM_READ_ONLY,
                labels.len(),
                ptr::null_mut(),
            )?
        };

        let mut oclbuf_input_predictions = unsafe {
            Buffer::<cl_float>::create(
                &self._context,
                CL_MEM_READ_ONLY,
                predictions.len(),
                ptr::null_mut(),
            )?
        };

        let mut oclbuf_input_x = unsafe {
            Buffer::<cl_float>::create(
                &self._context,
                CL_MEM_READ_ONLY,
                train_data[0].len() * train_data.len(),
                ptr::null_mut(),
            )?
```

```
        };

        let mut oclbuf_output_weights = unsafe {
            Buffer::<cl_float>::create(
                &self._context,
                CL_MEM_READ_ONLY,
                self._weights.len(),
                ptr::null_mut(),
            )?
        };

        let mut oclbuf_output_bias = unsafe {
            Buffer::<cl_float>::create(
                &self._context,
                CL_MEM_READ_ONLY,
                self._bias.len(),
                ptr::null_mut(),
            )?
        };

        let _writing_event = unsafe {
            self._queue.enqueue_write_buffer(
                &mut oclbuf_input_labels,
                CL_NON_BLOCKING,
                0,
                &labels,
                &[],
            )?
        };

        let _writing_event = unsafe {
            self._queue.enqueue_write_buffer(
                &mut oclbuf_input_predictions,
                CL_NON_BLOCKING,
                0,
                &predictions,
                &[],
            )?
        };

        let _writing_event = unsafe {
            self._queue.enqueue_write_buffer(
                &mut oclbuf_input_x,
                CL_NON_BLOCKING,
                0,
                &train_data_aligned,
                &[],
            )?
        };

        let _writing_event = unsafe {
            self._queue.enqueue_write_buffer(
                &mut oclbuf_output_weights,
                CL_NON_BLOCKING,
```

```
        0,
        &self._weights,
        &[],
    )?
};

let _writing_event = unsafe {
    self._queue.enqueue_write_buffer(
        &mut oclbuf_output_bias,
        CL_NON_BLOCKING,
        0,
        &self._bias,
        &[],
    )?
};

let kernel_event = unsafe {
    ExecuteKernel::new(&self._kernel_recalc_weights)
        .set_arg(&oclbuf_input_labels)
        .set_arg(&oclbuf_input_predictions)
        .set_arg(&oclbuf_input_x)
        .set_arg(&oclbuf_output_weights)
        .set_arg(&oclbuf_output_bias)
        .set_arg(&objects_amount)
        .set_arg(&learning_rate)
        .set_arg_local_buffer((objects_amount.next_power_of_two() as usize) * size_of::<f32>())
        .set_local_work_size(objects_amount.next_power_of_two() as usize)
        .set_global_work_size((objects_amount.next_power_of_two() as usize) * train_data[0].len())
        .enqueue_nd_range(&self._queue)?
};

let mut events: Vec<cl_event> = Vec::default();
events.push(kernel_event.get());

let read_event = unsafe {
    self._queue.enqueue_read_buffer(
        &oclbuf_output_weights,
        CL_BLOCKING,
        0,
        &mut self._weights,
        &events,
    )?
};

read_event.wait()?;

let read_event = unsafe {
    self._queue.enqueue_read_buffer(
        &oclbuf_output_bias,
        CL_BLOCKING,
        0,
        &mut self._bias,
        &events,
    )?
```

```
            };

            read_event.wait()?;

            if epoch % 10000 == 0 {
                println!("Current epoch: {}\n\
                Current weights: {:?}\n\
                Current bias: {}", epoch, self._weights, self._bias[0]);
            }
        }

        println!("================================\n\
        GPU Accelerated train completed!\n\
          Total epochs: {}\n\
          Result weights: {:?}\n\
          Result bias: {:?}\n\
          Elapsed time: {:.2?}
        ",
        epochs, self._weights, self._bias[0], now.elapsed());

        Ok(self)
    }

    fn predict(&self, train_data: &Vec<Vec<f32>>) -> Result<Vec<f32>, String> {
        let mut train_data_aligned = vec![0f32; train_data[0].len() * train_data.len()];
        for i in 0..train_data.len() {
            for j in 0..train_data[0].len() {
                train_data_aligned[j + i * train_data[0].len()] = train_data[i][j];
            }
        }

        self._predict(train_data, &train_data_aligned)
    }
}
```

```
float sigmoid(float x) {
    return 1. / (1. + exp(-x));
}

__kernel void prediction(__global const float* input_x,
                         __global const float* input_weights,
                         const float bias,
                         unsigned int NODES_AMOUNT,
                         __global float* predictions,
                         __local float* local_data) {
    int group_id = get_group_id(0);
    int local_id = get_local_id(0);

    int group_size = get_local_size(0);

    float val = 0.0f;
    if (local_id < NODES_AMOUNT)
```

```
        val = input_x[group_id * NODES_AMOUNT + local_id] * input_weights[local_id];

    local_data[local_id] = val;

    barrier(CLK_LOCAL_MEM_FENCE);

    for (int stride = group_size >> 1; stride > 0; stride >>= 1) {
        if (local_id < stride)
            local_data[local_id] += local_data[local_id + stride];
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if (!local_id)
        predictions[group_id] = sigmoid(local_data[0] + bias);
}
```

```
float work_group_reduce_add(__local float* local_data) {
    int group_size = get_local_size(0);
    int local_id = get_local_id(0);
    float result = 0.0;

    barrier(CLK_LOCAL_MEM_FENCE);

    for (int stride = group_size >> 1; stride > 0; stride >>= 1) {
        if (local_id < stride)
            local_data[local_id] += local_data[local_id + stride];
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    return local_data[0];
}

__kernel void recalc_weights(__global const float* input_labels,
                             __global const float* input_predictions,
                             __global const float* input_x,
                             __global float* output_weights,
                             __global float* output_bias,
                             unsigned int OBJECTS_AMOUNT,
                             float learning_rate,
                             __local float* local_data) {
    int global_id = get_global_id(0);
    int group_id = get_group_id(0);
    int local_id = get_local_id(0);

    int NODES_AMOUNT = get_global_size(0) / get_local_size(0);

    // group_id - номер признака
    // local_id - номер тестируемого объекта

    float error = 0.0;
    float grad = 0.0;
    float bias = 0.0;
    if (local_id < OBJECTS_AMOUNT) {
```

```
        float y_pred = input_predictions[local_id];
        error = input_labels[local_id] - y_pred;

        grad = error * y_pred *
               (1 - y_pred) *
               input_x[local_id * NODES_AMOUNT + group_id];

        if (group_id == 0) {
            bias = error * y_pred * (1 - y_pred);
        }
    }

    local_data[local_id] = grad;

    float grad_sum = work_group_reduce_add(local_data);
    if (local_id == 0) {
        output_weights[group_id] += learning_rate * grad_sum;
    }

    // Скорректировать bias
    local_data[local_id] = bias;
    float bias_sum = work_group_reduce_add(local_data);

    if (group_id == 0 && local_id == 0) {
        output_bias[0] += bias_sum;
    }
}
```

Вывод: в ходе лабораторной работы изучили основы параллельного программирования с использованием OpenCL, реализовать вычислительные задачи с применением графического ускорителя (GPU), оценить производительность и масштабируемость решений при выполнении вычислений.

В данной задаче победить реализации на OpenCL процессор не удалось, однако можно заметить, что время работы при выполнении на OpenCL при увеличении количества свойств практически не растёт по сравнению с однопоточной реализацией. Можно сделать вывод, что реализация OpenCL имеет смысл, если будут использоваться многомерные данные, при маломерных данных с небольшим количеством нейронов, можно использовать и однопоточный подход.

Также для обучения нейронных моделей можно использовать специализированные устройства, которые сегодня разрабатываются компанией Nvidia в частности. Эти устройства могут отличаться повышенным количеством ядер, памяти, возможности использования более широких групп с увеличенной локальной памятью.

Не стоит на месте и развитие стандартов. OpenCL, пусть и является одним из широкоподдерживаемых стандартов, на сегодняшний день находится в стадии ”угасания”. Рекомендуется использовать более современные поддерживаемые технологии, такие как OpenGL, Vulkan. Производители устройств предлагают специализированные решения для конкретных устройств, так для AMD существует ROCm, для Nvidia существует CUDA. Однако возникает проблема совместимости, но есть решение и для неё. Так AMD HIP предлагает компиляцию как для CUDA, так и для ROCm (C++). Можно использовать Rust - более безопасный язык программирования - для написания шейдеров, это возмож-

но благодаря компиляции Rust в промежуточный шейдерный код SPIR-V, поддерживаемый многими технологиями при помощи пакета rust-gpu. Пакет всё ещё развивается, так в нём пока что ещё не поддерживается компиляция в OpenCL, поэтому данная лабораторная работа и была выполнена с применением традиционного шейдера.

Современные тенденции развития графических ускорителей направлены на создание универсального языка для написания программ для графических ускорителей а также на увеличения производительности для решения актуальных задач: использование нейронных сетей а также их обучение.