Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт космических и информационных технологий
Кафедра вычислительной техники

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №1

Преподаватель _____ Тарасов С.А.
подпись, дата инициалы, фамилия

Студент КИ22-06б, 032210841 _____ Федченко А.О.
номер группы, зачетной книжки подпись, дата инициалы, фамилия

Красноярск 2025

Цель работы:

Освоить базовые навыки программирования CUDA: работу с одномерными сетка ми нитей и динамической памятью устройства. Изучить паттерн проектирования data + view.

Задание:

Изучить основы модели параллельного программирования CUDA, реализовать и сравнить производительность функции на центральном и графическом процессорах, а также проанализировать полученные результаты для понимания преимущества и особенностей GPU-вычислений. Сумма векторов.

1 ИСХОДНЫЙ КОД ПРОГРАММЫ

Фрагменты кода представлены на рисунках 1-4.

```cpp
#ifndef HSYS_VECTOR_CUH
#define HSYS_VECTOR_CUH

#include "data_block.cuh"
#include "vector_accessor.cuh"

namespace hsys {

template <AtomKind AtomT>
struct Vector {
  struct hsys_vector_feature {};

 private:
  DataBlock<AtomT> block_;

 public:
  Vector(std::size_t size)
      : block_(size) {}

  [[nodiscard]] std::size_t size() const {
    return block_.size();
  }

  DataBlock<AtomT>& block() {
    return block_;
  }

  const DataBlock<AtomT>& block() const {
    return block_;
  }

  VectorAccessor<AtomT> accessor() {
    return VectorAccessor<AtomT>(block_.data(), size());
  }

  const VectorAccessor<AtomT> accessor() const {
    return VectorAccessor<AtomT>(const_cast<AtomT*>(block_.data()), size());
  }
};

}  // namespace hsys

#endif  // HSYS_VECTOR_CUH
```

Рисунок 1 – Код файла vector

```
#ifndef HSYS_KERNEL_VECTOR_ADD
#define HSYS_KERNEL_VECTOR_ADD

#include "../vector_accessor.cuh"

namespace hsys {

template <AtomKind AtomT>
__global__ void kernel_vector_add(VectorAccessor<AtomT> c,
    const VectorAccessor<AtomT> a,
    const VectorAccessor<AtomT> b) {
  std::size_t i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < a.size()) c[i] = a[i] + b[i];
}

}  // namespace hsys

#endif  // HSYS_KERNEL_VECTOR_ADD
```

Рисунок 2 – Код файла kernel_vecadd

```cpp
#ifndef HSYS_DATA_BLOCK
#define HSYS_DATA_BLOCK

#include "kinds.cuh"

// TODO: check CUDA errors

namespace hsys {

template <AtomKind AtomT>
struct DataBlock {
  struct hsys_data_block_feature {};

 private:
  std::size_t size_;
  AtomT* data_;

 public:
  using atom_t = AtomT;

  DataBlock(std::size_t size)
      : size_(size)
      , data_(nullptr) {
    cudaMalloc(&data_, size * sizeof(AtomT));
  }

  DataBlock(const DataBlock& other)
      : size_(other.size_)
      , data_(nullptr) {
    cudaMalloc(&data_, size_ * sizeof(AtomT));
    cudaMemcpy(data_, other.data_, size_ * sizeof(AtomT), cudaMemcpyDeviceToDevice);
  }

  DataBlock(DataBlock&& other) noexcept
      : size_(other.size_)
      , data_(other.data_) {
    other.data_ = nullptr;
  }

  DataBlock& operator=(const DataBlock& other) {
    if (this != &other) {
      if (data_) cudaFree(data_);
      size_ = other.size_;
      cudaMalloc(&data_, size_ * sizeof(AtomT));
      cudaMemcpy(data_, other.data_, size_ * sizeof(AtomT), cudaMemcpyDeviceToDevice);
    }
    return *this;
  }

  DataBlock& operator=(DataBlock&& other) noexcept {
    if (this != &other) {
      if (data_) cudaFree(data_);
      size_ = other.size_;
      data_ = other.data_;
      other.data_ = nullptr;
    }
    return *this;
  }

  AtomT* data() {
    return data_;
  }

  const AtomT* data() const {
    return data_;
  }

  [[nodiscard]] std::size_t size() const {
    return size_;
  }

  void copy_to_host(AtomT* host_ptr) const {
    cudaMemcpy(host_ptr, data_, size_ * sizeof(AtomT), cudaMemcpyDeviceToHost);
  }

  void copy_from_host(const AtomT* host_ptr) {
    cudaMemcpy(data_, host_ptr, size_ * sizeof(AtomT), cudaMemcpyHostToDevice);
  }

  ~DataBlock() noexcept {
    if (data_) cudaFree(data_);
  }
};

}  // namespace hsys

#endif  // HSYS_DATA_BLOCK
```

Рисунок 3 – Код файла Data

```cpp
#include <work1/vector.cuh>
#include <work1/vector_operators.cuh>

#define EIGEN_NO_CUDA

#include <Eigen/Dense>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>
#include <gtest/gtest.h>

class VectorAddTest : public ::testing::TestWithParam<std::pair<std::size_t, float>> {
 protected:
  bool vadd_test_impl(std::size_t size, float tol) {
    Eigen::VectorXf a_target = Eigen::VectorXf::Random(
        size);  // NOLINT(cppcoreguidelines-narrowing-conversions)

    Eigen::VectorXf b_target = Eigen::VectorXf::Random(
        size);  // NOLINT(cppcoreguidelines-narrowing-conversions)

    Eigen::VectorXf c_target = a_target + b_target;

    auto a = hsys::Vector<float>(size);
    a.block().copy_from_host(a_target.data());

    auto b = hsys::Vector<float>(size);
    b.block().copy_from_host(b_target.data());

    auto c = a + b;

    if (c.size() != size) return false;

    Eigen::VectorXf c_from_device = Eigen::VectorXf(size);
    c.block().copy_to_host(c_from_device.data());

    return c_target.isApprox(c_from_device, tol);
  }
};

TEST_P(VectorAddTest, vadd_test) {  // It's Ok, dumb clangd!
  auto [size, tol] = GetParam();
  EXPECT_TRUE(vadd_test_impl(size, tol));
}

// clang-format off
INSTANTIATE_TEST_SUITE_P(
    VectorAddTestSuite,
    VectorAddTest,
    ::testing::Values(
        std::make_pair(1, 1e-6),
        std::make_pair(2, 1e-6),
        std::make_pair(3, 1e-6),
        std::make_pair(127, 1e-6),
        std::make_pair(128, 1e-6),
        std::make_pair(129, 1e-6),
        std::make_pair(512, 1e-6),
        std::make_pair(513, 1e-6),
        std::make_pair(1023, 1e-6),
        std::make_pair(1024, 1e-6)
    )
);
```

Рисунок 4 – Код файла vector_add

2 РЕЗУЛЬТАТЫ ИЗМЕРЕНИЙ

График, показывающий зависимость размеров векторов от времени представлен на рисунке 5.

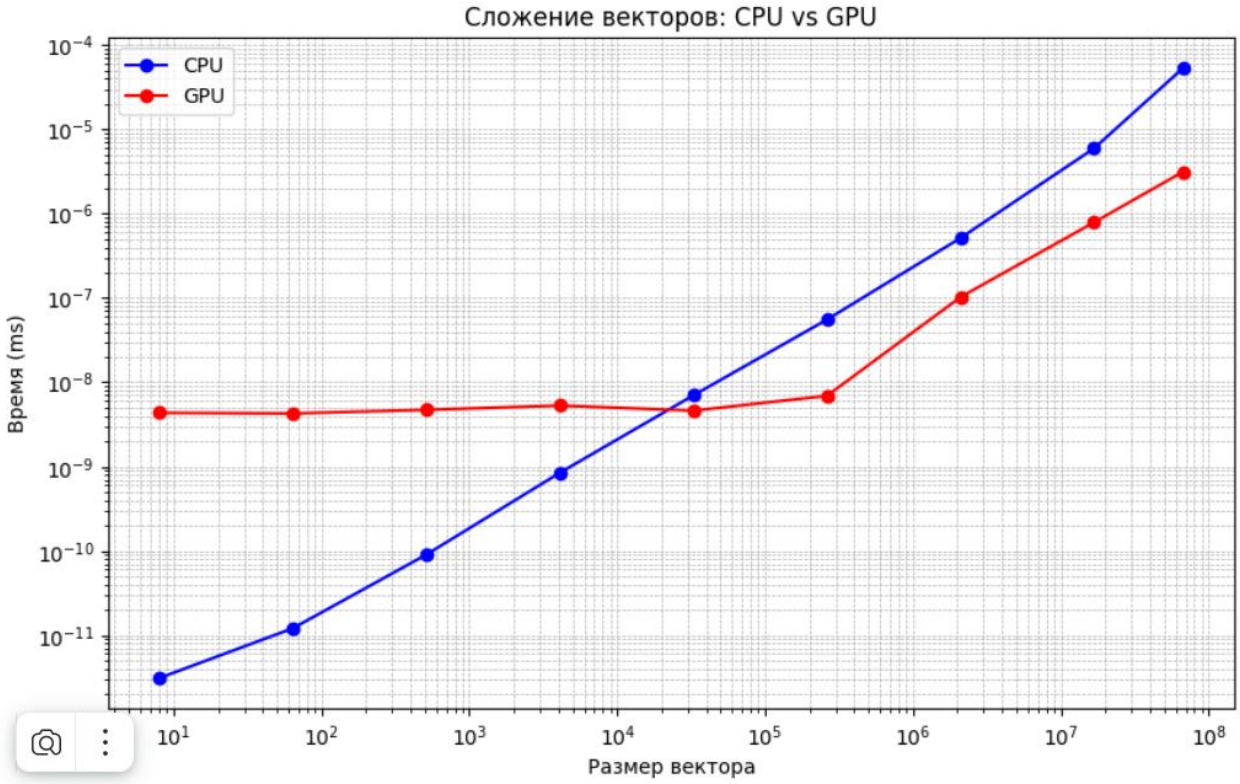

Рисунок 5 – График сложения векторов

Как мы видим по графику для небольших размеров векторов CPU эффективнее. Для больших размеров GPU значительно ускоряет сложение из-за параллельной обработки.

На рисунке 6 изображен график ускорения GPU относительно CPU.

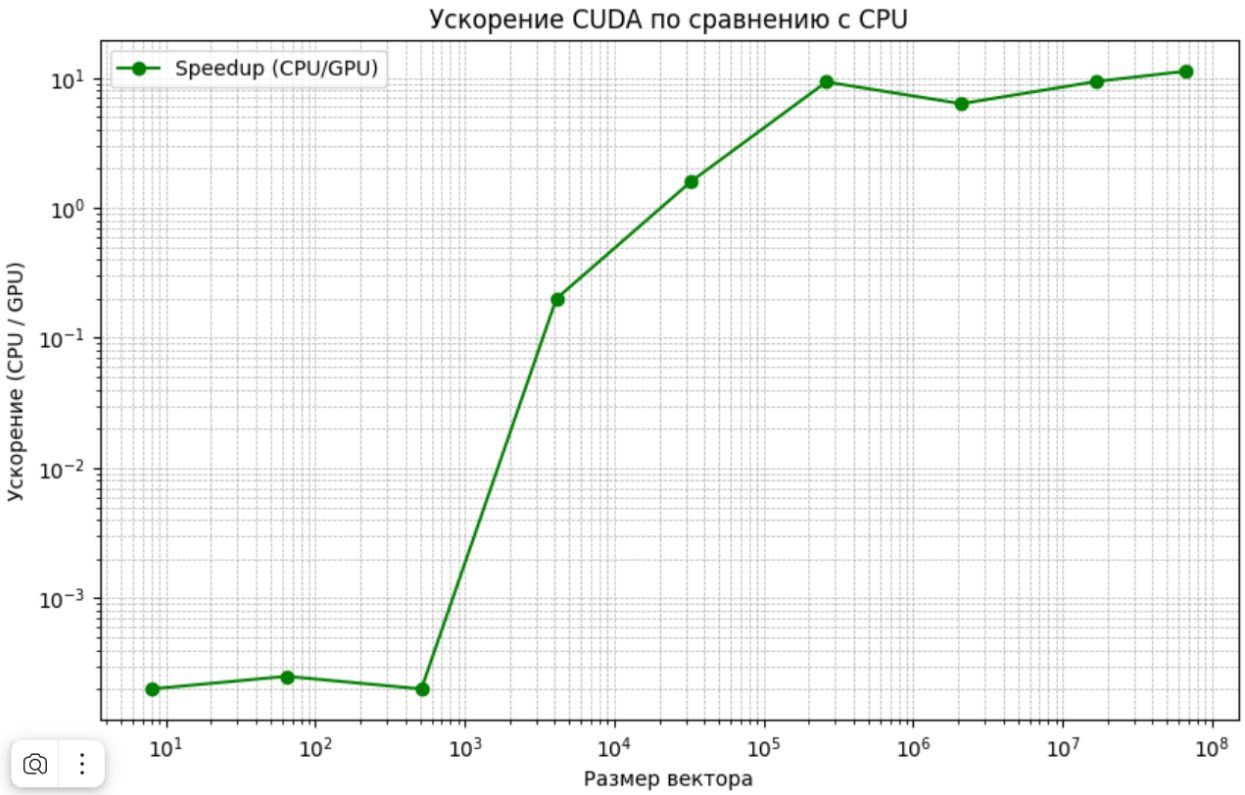

Рисунок 6 – График ускорения GPU и CPU

Как можем видеть по рисунке 6, ускорение GPU возрастает относительно размерам векторов. При больших размерах ускорение будет активно возрастать до определенного момента, после чего ускорение будет примерно одинаково.

3 ВЫВОД

В ходе работы была реализована система работы с векторами на GPU с использованием классов, а также кернела для сложения векторов. Были проведены тесты и бенчмарки, которые подтвердили корректность работы и показали значительное ускорение. В процессе работы были закреплены навыки управления памятью на GPU, организации удобного интерфейса для работы с данными, тестирования и анализа производительности вычислений.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. СТУ 7.5–07–2021. Стандарт университета «Система менеджмента качества. Общие требования к построению, изложению и оформлению документов учебной деятельности».

2. Andersan-tumry70. HistoricalEventPR1 [Электронный ресурс] // GitHub. URL: https://github.com/Andersan-tumry70/hsys (дата обращения: 05.10.2025).