Министерство науки и высшего образования РФ

Федеральное государственное автономное

образовательное учреждение высшего образования

**«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

Институт космических и информационных технологий

Кафедра вычислительной техники

**ОТЧЁТ О ПРАКТИЧЕСКОЙ РАБОТЕ №2**

**по дисциплине**

**«Гибридные вычислительные системы»**

|  |  |  |
| --- | --- | --- |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_ | С. А. Тарасов |
|  | подпись, дата | инициалы, фамилия |
| Студент КИ22-07Б, 032212677 | \_\_\_\_\_\_\_\_\_\_\_\_ | Л. А. Глушков |
| номер группы, зачетной книжкой | подпись, дата | инициалы, фамилия |
| Студент КИ22-07Б, 032215583 | \_\_\_\_\_\_\_\_\_\_\_\_ | А. М. Коробков |
| номер группы, зачетной книжкой | подпись, дата | инициалы, фамилия |
| Студент КИ22-07Б, 032214653 | \_\_\_\_\_\_\_\_\_\_\_\_ | И. О. Бердин |
| номер группы, зачетной книжкой | подпись, дата | инициалы, фамилия |

**СОДЕРЖАНИЕ**

[Задание 3](#_Toc211381046)

[Ключевые фрагменты кода 4](#_Toc211381047)

[Результат выполнения программы 4](#_Toc211381048)

[Заключение 7](#_Toc211381049)

[Приложение А 8](#_Toc211381050)

[Приложение Б 10](#_Toc211381051)

# **Задание**

1. Реализовать класс Matrix (см. рис. 1), который представляет собой

фасад, объединяющий:

• агрегацию с классом Data через std::shared\_ptr (для разделения данных между матрицами);

• композицию с классом MatrixView (каждый экземпляр Matrix имеет собственное представление);

• единый упрощённый интерфейс для работы с матрицами.

2. Реализовать класс MatrixView, предоставляющий интерфейс для доступа к элементам матрицы на устройстве (чтение и запись по индексам, rowmajor). Класс должен быть тривиально-копируемым.

3. Реализовать кёрнел kernel\_matmul\_naive, который принимает объекты MatrixView по значению и вычисляет произведение матриц без использования разделяемой памяти.

4. Перегрузить оператор operator\* для класса Matrix, который вызывает kernel\_matmul\_naive.

5. Используя фреймворк Google Test, разработать модульные тесты для operator\* со следующими размерами матриц: A (m × k) и B (k × n), где m, n, k ∈ {1, 2, 3, 127, 128, 129, 512}. В качестве эталона для сравнения использовать результат аналогичной операции для Eigen::MatrixXf; для верификации результатов применять метод Eigen::MatrixXf::isApprox с абсолютной точностью 10−5 .

6. Используя фреймворк Google Benchmark, разработать бенчмарки для operator\* со следующими размерами матриц: n×n, где n ∈ {16, 32, 64, 128, 256, 512, 1024}. Бенчмарки должны игнорировать время, затраченное на выделение, копирование и освобождение памяти. Для корректного измерения времени выполнения CUDA-кода необходимо использовать CUDA Events API.

7. Построить график реальной вычислительной сложности (real complexity, пример на рис. 2) operator\* для умножения матриц типа Matrix и аналогичный график для умножения матриц типа Eigen::MatrixXf.

8. Построить график ускорения (speedup, пример на рис. 3) operator\* для умножения матриц типа Matrix относительно operator\* для Eigen::MatrixXf.

9. Сравнить экспериментальные результаты с теоретическими оценками вычислительной сложности.

10. Подготовить отчёт, содержащий:

• ключевые фрагменты реализованного кода;

• ссылку на репозиторий с полной реализацией;

• графики результатов измерений;

• анализ и интерпретацию полученных результатов.

# **Ключевые фрагменты кода**

Ключевые моменты кода:

* Реализация класса матрицы (исходный код приведен в приложении А);
* Реализация класса представления матрицы (исходный код приведен в приложении Б).

# **Результат выполнения программы**

На рисунках 1 и 2 представлены графики, построенные на основании выполненной программы.

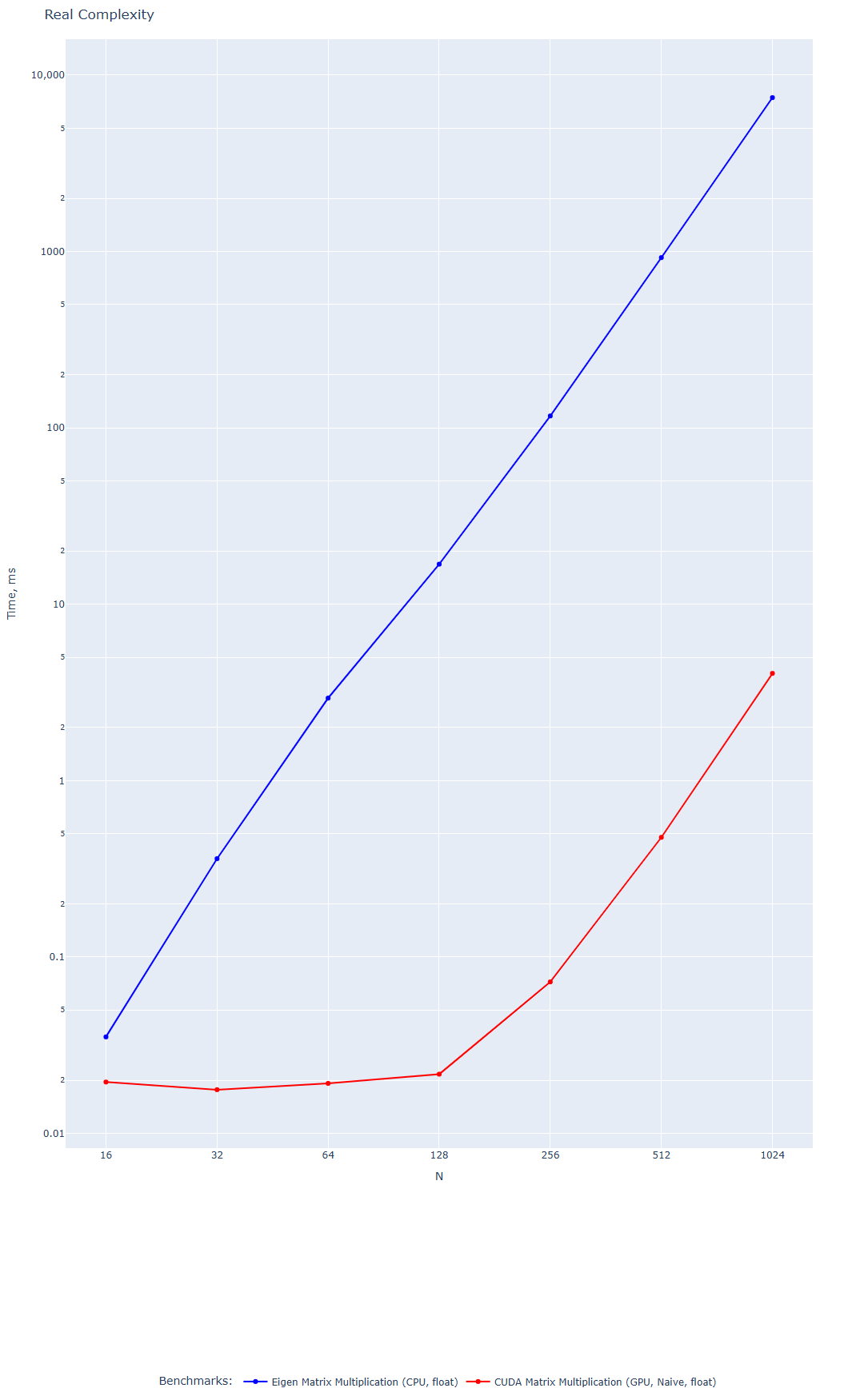


Рисунок 1 – График реальной вычислительной сложности

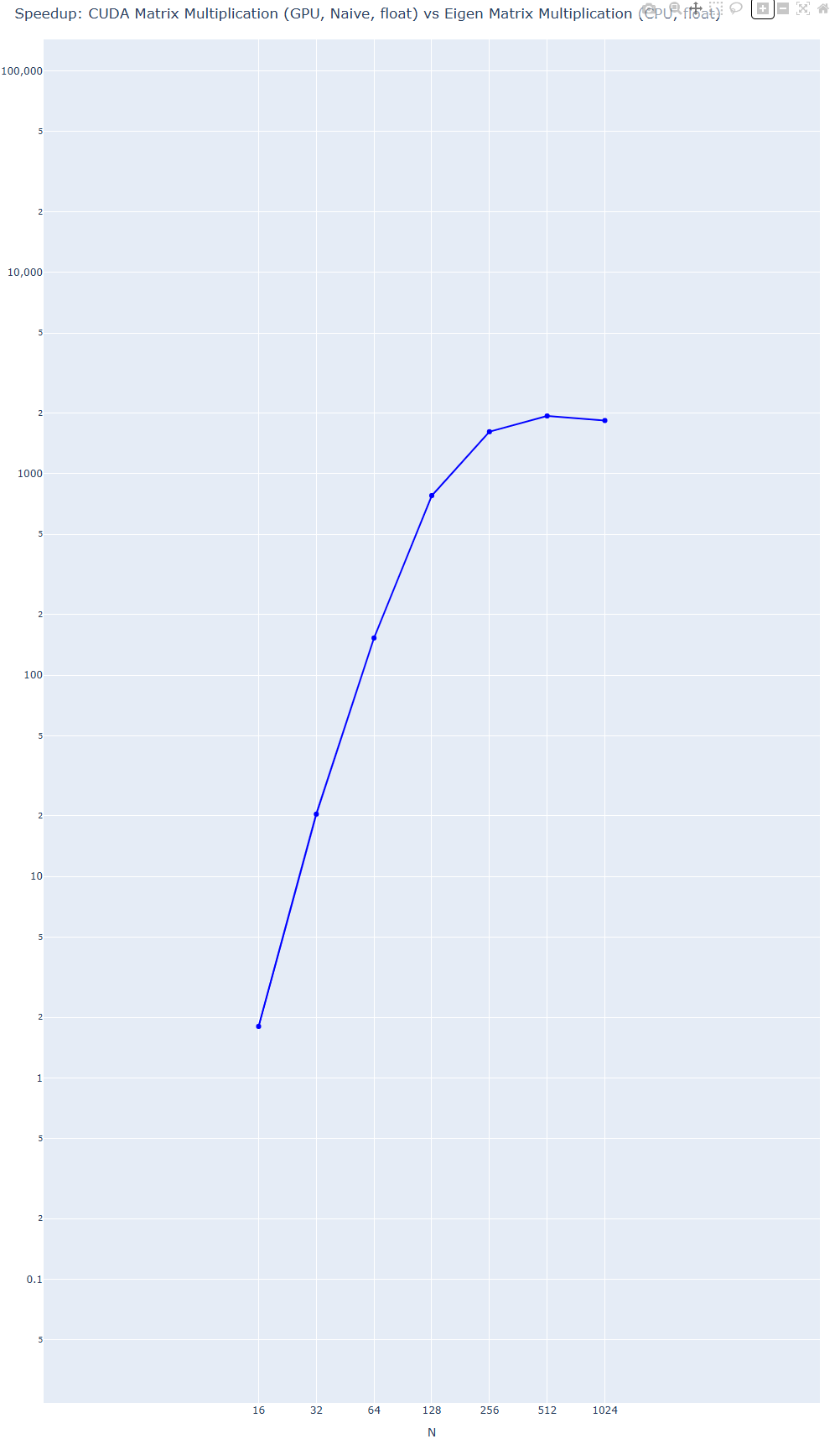


Рисунок 2 – График ускорения

На основании полученных графиков можно сделать вывод, что при малых размерах матриц (N < 128) GPU-ядра не полностью загружены, т.к. время вычисления не меняется.

# **ЗАКЛЮЧЕНИЕ**

В результате выполнения данной работы были закреплены базовые навыки программирования CUDA. Были освоены принципы работы с двумерными сетками нитей.

# **ПРИЛОЖЕНИЕ А**

#**pragma** once

#**include** "data.cuh"

#**include** "matrixView.cuh"

#**include** "matrixOperationsKernel.cuh"

#**include** <memory>

#**include** <stdexcept>

#**include** <iostream>

**template**<**typename** T>

**class** **Matrix** {

**private**:

std::shared\_ptr<Data<T>> data\_;

MatrixView<T> view\_;

**public**:

Matrix(std::size\_t rows, std::size\_t cols) : data\_(std::make\_shared<Data<T>>(rows \* cols)), view\_(data\_->data(), rows, cols) {}

Matrix(T\* externalData, std::size\_t rows, std::size\_t cols) : data\_(nullptr), view\_(externalData, rows, cols) {}

Matrix(const Matrix& other) = **default**;

Matrix& **operator**=(const Matrix& other) {

**if**(**this** != &other) {

data\_ = other.data\_;

view\_ = MatrixView<T>(other.view\_.data(), other.view\_.rows(), other.view\_.cols());

}

**return** \***this**;

}

~Matrix() = **default**;

MatrixView<T> **view**() const {**return** view\_;}

T\* **data**() {**return** view\_.data();}

const T\* **data**() const {**return** view\_.data();}

std::size\_t **rows**() const {**return** view\_.rows();}

std::size\_t **cols**() const {**return** view\_.cols();}

std::size\_t **size**() const {**return** view\_.size();}

T& **operator**()(std::size\_t row, std::size\_t col) {**return** view\_(row, col);}

const T& **operator**()(std::size\_t row, std::size\_t col) const {**return** view\_(row, col);}

bool **isSameSize**(const Matrix& other) const {**return** view\_.isSameSize(other.view\_);}

void **fill**(const T& value) {

**for** (std::size\_t i = 0; i < size(); i++) {

data()[i] = value;

}

}

void **print**(const char\* name = "") const {

std::cout << name << " (" << rows() << "x" << cols() << "):\n";

**for** (std::size\_t i = 0; i < rows(); ++i) {

**for** (std::size\_t j = 0; j < cols(); ++j) {

std::cout << (\***this**)(i, j) << " ";

}

std::cout << "\n";

}

std::cout << std::endl;

}

\_\_host\_\_ Matrix **operator**\*(const Matrix& other) const;

\_\_host\_\_ Matrix **operator**\*(const T& scalar) const;

\_\_host\_\_ Matrix **operator**+(const Matrix& other) const;

\_\_host\_\_ Matrix **operator**-(const Matrix& other) const;

Matrix **transpose**() const;

};

# **ПРИЛОЖЕНИЕ Б**

#**pragma** once

#**include** <cstddef>

#**include** <cuda\_runtime.h>

**template**<**typename** T>

**class** **MatrixView** {

**private**:

T\* data\_;

std::size\_t rows\_;

std::size\_t cols\_;

**public**:

\_\_host\_\_ \_\_device\_\_ **MatrixView**() : data\_(nullptr), rows\_(0), cols\_(0) {}

\_\_host\_\_ \_\_device\_\_ **MatrixView**(T\* data, std::size\_t rows, std::size\_t cols) : data\_(data), rows\_(rows), cols\_(cols) {}

\_\_host\_\_ \_\_device\_\_ **MatrixView**(const T\* data, std::size\_t rows, std::size\_t cols) : data\_(const\_cast<T\*>(data)), rows\_(rows), cols\_(cols) {}

\_\_host\_\_ \_\_device\_\_ ~MatrixView() = **default**;

\_\_host\_\_ \_\_device\_\_ **MatrixView**(const MatrixView&) = **default**;

\_\_host\_\_ \_\_device\_\_ MatrixView& **operator**=(const MatrixView&) = **default**;

\_\_host\_\_ \_\_device\_\_ T& **operator**()(std::size\_t row, std::size\_t col) {**return** data\_[row \* cols\_ + col];}

\_\_host\_\_ \_\_device\_\_ const T& **operator**()(std::size\_t row, std::size\_t col) const {**return** data\_[row \* cols\_ + col];}

\_\_host\_\_ \_\_device\_\_ T\* **data**() {**return** data\_;}

\_\_host\_\_ \_\_device\_\_ const T\* **data**() const {**return** data\_;}

\_\_host\_\_ \_\_device\_\_ std::size\_t **rows**() const {**return** rows\_;}

\_\_host\_\_ \_\_device\_\_ std::size\_t **cols**() const {**return** cols\_;}

\_\_host\_\_ \_\_device\_\_ std::size\_t **size**() const {**return** rows\_ \* cols\_;}

\_\_host\_\_ \_\_device\_\_ bool **isSameSize**(const MatrixView<T>& other) const {**return** rows\_ == other.rows\_ && cols\_ == other.cols\_;}

};