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 №3

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

Красноярск 2025

СОДЕРЖАНИЕ

Задание

1. Разработать кёрнел kernel_matmul_shmem, который принимает объекты MatrixView по значению и вычисляет произведение матриц, используя разделяемую память в качестве буфера для обмена данными между нитями. Для синхронизации нитей блока использовать __syncthreads; не использовать CUDA Cooperative Groups;

2. Перегрузить оператор operator* для класса Matrix, используя указанный кёрнел.

3. 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 .

4. 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.

5. Построить график реальной вычислительной сложности умножения матриц типа Matrix с помощью новой реализации operator*, а также аналогичный график для предыдущей версии operator*.

6. Построить график ускорения (speedup) новой реализации operator* относительно предыдущей версии.

7. 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 перегрузки оператора * (исходный код приведен в приложении Б).

Результат выполнения программы

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

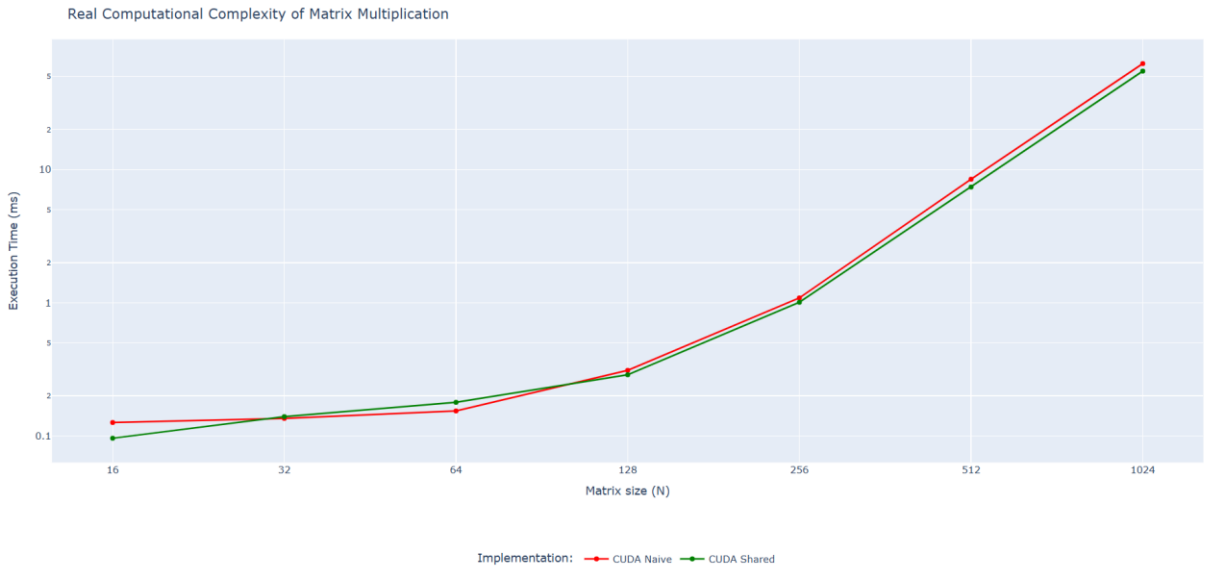

Рисунок 1 – График реальной вычислительной сложности

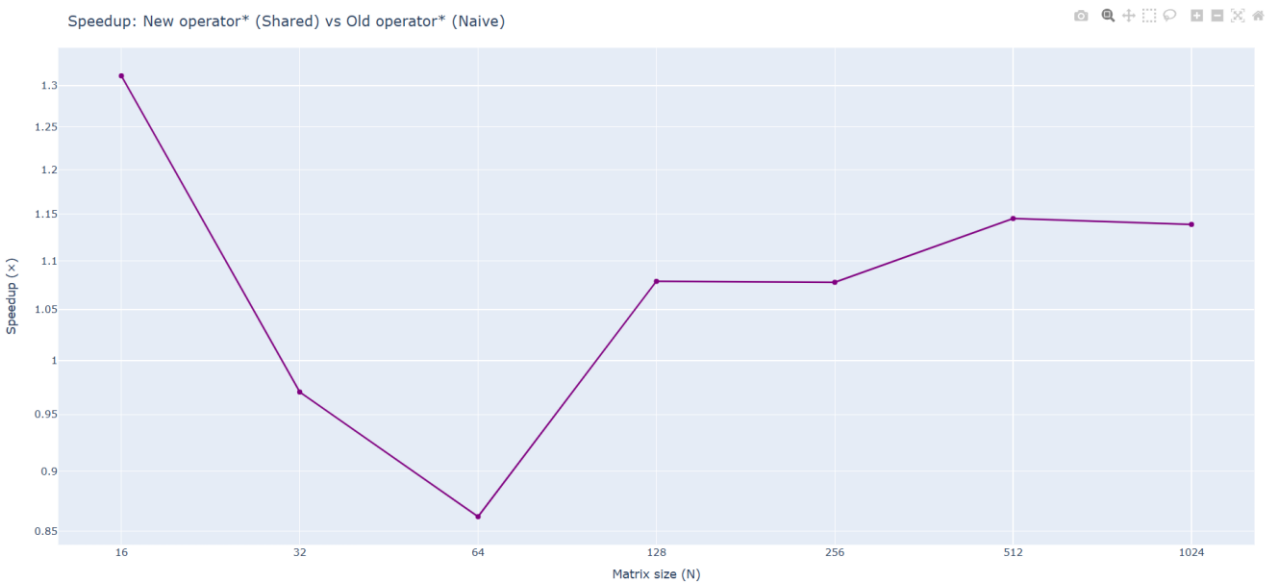

Рисунок 2 – График ускорения

ЗАКЛЮЧЕНИЕ

В результате выполнения данной работы были закреплены базовые навыки работы с двумерными сетками нитей CUDA. Была освоена работа с разделяемой памятью и примитивами синхронизации нитей.

ПРИЛОЖЕНИЕ А

```cpp
#pragma once
#include "data.cuh"
#include "matrixView.cuh"
#include "matrixOperationsKernel.cuh"
#include <memory>
#include <stdexcept>
#include <iostream>

enum class MultiplyAlgorithm {
    naive,
    shared
};

template<typename T, MultiplyAlgorithm Algorithm = MultiplyAlgorithm::shared>
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;

    template<MultiplyAlgorithm OtherAlgorithm>
    Matrix(const Matrix<T, OtherAlgorithm>& other) : data_(std::make_shared<Data<T>>(other.rows() * other.cols())), view_(data_->data(), other.rows(), other.cols()
        for (std::size_t i = 0; i < size(); ++i) {
            data()[i] = other.data()[i];
        }
    }
```

Рисунок 3 – Реализация класса матрицы

```cpp
    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;
        }
    }
```

Рисунок 4 – Реализация класса матрицы

```
59          void print(const char* name = "") const {
60              std::cout << name << " (" << rows() << "x" << cols() << "):\n";
61              for (std::size_t i = 0; i < rows(); ++i) {
62                  for (std::size_t j = 0; j < cols(); ++j) {
63                      std::cout << (*this)(i, j) << " ";
64                  }
65                  std::cout << "\n";
66              }
67              std::cout << std::endl;
68          }
69
70          template<MultiplyAlgorithm OtherAlgorithm>
71          Matrix operator*(const Matrix<T, OtherAlgorithm>& other) const;
72      };
```

Рисунок 5 – Реализация класса матрицы

ПРИЛОЖЕНИЕ Б

```
6   template<typename T, MultiplyAlgorithm Algorithm>
7   template<MultiplyAlgorithm OtherAlgorithm>
8   Matrix<T, Algorithm> Matrix<T, Algorithm>::operator*(const Matrix<T, OtherAlgorithm>& other) const {
9       if (cols() != other.rows()) {
10          throw std::invalid_argument("Invalid matrix for multiply");
11      }
12
13      Matrix<T, Algorithm> result(rows(), other.cols());
14
15      constexpr std::size_t BLOCK_SIZE = 16;
16      dim3 blockDim(BLOCK_SIZE, BLOCK_SIZE);
17      dim3 gridDim((other.cols() + BLOCK_SIZE - 1) / BLOCK_SIZE,
18                   (rows() + BLOCK_SIZE - 1) / BLOCK_SIZE);
19
20      if constexpr (Algorithm == MultiplyAlgorithm::shared) {
21          matrixMultiplyKernelShared<T><<<gridDim, blockDim>>>(view(), other.view(), result.view());
22      } else if constexpr (Algorithm == MultiplyAlgorithm::naive) {
23          matrixMultiplyKernel<T><<<gridDim, blockDim>>>(view(), other.view(), result.view());
24      }
25
26
27      cudaError_t err = cudaGetLastError();
28      if (err != cudaSuccess) {
29          throw std::runtime_error("CUDA error: " + std::string(cudaGetErrorString(err)));
30      }
31      cudaDeviceSynchronize();
32
33      return result;
34  }
```

Рисунок 6 – Реализация перегрузки оператора *