ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЯ И ИНФОРМАТИКИ»

КАФЕДРА ПМиК

Расчетно-графическая работа  
по дисциплине «Программирование графических процессоров»

Выполнил: студент 3-го курса,  
группы ИП-014  
Обухов.А.И.

Проверил: Старший преподаватель  
Кафедры прикладной математики и кибернетики  
Нужнов А.В.

Новосибирск, 2023

Оглавление

[Постановка задачи 3](#_Toc135132045)

[Описание архитектуры CUDA и «сырого» кода CUDA C 4](#_Toc135132046)

[Описание библиотеки Thrust 5](#_Toc135132047)

[Описание библиотеки cuBLAS 6](#_Toc135132048)

[Описание OpenGL 7](#_Toc135132049)

[Описание реализации 8](#_Toc135132050)

[Результаты работы программы 10](#_Toc135132051)

[Листинг программы 13](#_Toc135132052)

# Постановка задачи

**Текст задания:** Сравнение производительности программ на основе интерфейса CUDA (пункт 1) и на основе OpenGL - вычислительных шейдеров (оценка “отлично”).

Цель данного проекта – провести сравнительный анализ производительности программ, реализующих алгоритмы линейной алгебры с использованием библиотек Thrust, cuBLAS, "сырого" CUDA C кода и на основе OpenGL(вычислительные шейдеры). Для этого необходимо рассмотреть следующие задачи:

1. Изучить основные принципы работы библиотек Thrust и cuBLAS, а также "сырого" CUDA C кода.
2. Реализовать несколько алгоритмов линейной алгебры с использованием каждой из библиотек и "сырого" CUDA C кода.
3. Провести тестирование разработанных программ на различных объемах данных и сравнить полученные результаты в терминах времени выполнения операций линейной алгебры.
4. Проанализировать полученные результаты и сделать выводы о производительности каждого подхода и эффективности использования библиотек Thrust, cuBLAS, "сырым" CUDA C кодом и OpenGL.

# Описание архитектуры CUDA и «сырого» кода CUDA C

CUDA (Compute Unified Device Architecture) — это архитектура параллельных вычислений, разработанная компанией NVIDIA для использования графических процессоров (GPU) в качестве вычислительных устройств. Она предоставляет программистам возможность использовать вычислительные возможности GPU для выполнения параллельных вычислений.

CUDA позволяет программистам писать приложения на языке программирования C или C++, которые могут выполняться на графическом процессоре. Это делает возможным ускорение вычислений в таких областях, как научные исследования, машинное обучение, анализ больших данных и другие.

"Сырой" код CUDA — это прямой доступ к возможностям графического процессора через использование языка CUDA C. Он позволяет программистам написать свои собственные ядра (kernel), которые могут выполняться на GPU. Код ядер CUDA C напоминает обычный C-код, но включает дополнительные функции, такие как спецификаторы качества, которые указывают, как ядро должно выполняться на GPU.

"Сырой" код CUDA предоставляет более гибкий и точный контроль над выполнением кода на GPU. Он позволяет программистам оптимизировать свой код для специфических характеристик архитектуры GPU, что может привести к ускорению вычислений и более эффективному использованию ресурсов GPU. Однако написание "сырого" кода CUDA может потребовать большего времени и усилий, чем использование библиотек, таких как cuBLAS и Thrust.

# Описание библиотеки Thrust

Thrust — это высокоуровневая библиотека для обобщенного программирования на языке C++ с использованием графических процессоров NVIDIA CUDA. Библиотека предоставляет множество алгоритмов, контейнеров и функций, которые упрощают написание эффективных параллельных программ на CUDA C.

Thrust предоставляет широкий спектр функций для работы с массивами данных, включая сортировку, свертку, поиск, перемешивание и т.д. Она также предоставляет контейнеры, такие как векторы, матрицы и многомерные массивы, которые могут быть использованы для хранения и манипулирования данными. Кроме того, Thrust поддерживает различные операции над контейнерами, такие как слияние, преобразование, фильтрацию и другие.

Одной из главных преимуществ Thrust является ее интеграция с языком C++, что позволяет программистам использовать знакомый синтаксис языка и библиотеки STL для работы с данными на GPU. Это позволяет легко переносить существующий код на CUDA на платформу, поддерживаемую Thrust.

Thrust также предоставляет удобный интерфейс для работы с различными типами данных, включая пользовательские типы, что позволяет использовать ее в широком диапазоне задач вычислительной математики, машинного обучения и других областях.

В целом, Thrust является мощным инструментом для разработки эффективных параллельных программ на CUDA C, позволяя программистам сосредоточиться на решении задач вычислительной математики и машинного обучения, а не на деталях параллельного программирования на GPU.

# Описание библиотеки cuBLAS

cuBLAS — это библиотека линейной алгебры для GPU, разработанная компанией NVIDIA и оптимизированная для использования с графическими процессорами NVIDIA CUDA. Библиотека предоставляет реализации на GPU стандартных алгоритмов линейной алгебры, таких как умножение матриц, решение систем линейных уравнений, нахождение собственных значений и векторов, QR-разложение и другие.

cuBLAS имеет широкий функционал для работы с различными типами матриц и векторов, в том числе с плотными и разреженными матрицами. Библиотека обеспечивает максимальную производительность вычислений на GPU, используя оптимизированные алгоритмы и обходя ограничения, которые могут возникнуть при использовании центрального процессора (CPU).

Одним из главных преимуществ cuBLAS является ее оптимизация под специфичные характеристики архитектуры GPU, такие как высокая параллельность и высокая пропускная способность. Библиотека позволяет получить существенное ускорение вычислений линейной алгебры, что делает ее незаменимым инструментом для разработки многих приложений в области научных исследований, машинного обучения и других областей.

В целом, cuBLAS является мощным инструментом для разработки эффективных приложений на CUDA, обеспечивающим максимальную производительность и оптимизацию для работы с матрицами и векторами на GPU.

# Описание OpenGL

Основным принципом работы OpenGL является получение наборов векторных графических примитивов в виде точек, линий и треугольников с последующей математической обработкой полученных данных и построением растровой картинки на экране и/или в памяти. Векторные трансформации и растеризация выполняются графическим конвейером (graphics pipeline), который по сути представляет собой дискретный автомат. Абсолютное большинство команд OpenGL попадает в одну из двух групп: либо они добавляют графические примитивы на вход в конвейер, либо конфигурируют конвейер на различное исполнение трансформаций.

OpenGL является низкоуровневым процедурным API, что вынуждает программиста диктовать точную последовательность шагов, чтобы построить результирующую растровую графику (императивный подход). Это является основным отличием от дескрипторных подходов, когда вся сцена передается в виде структуры данных (чаще всего дерева), которое обрабатывается и строится на экране. С одной стороны, императивный подход требует от программиста глубокого знания законов трёхмерной графики и математических моделей, с другой стороны — даёт свободу внедрения различных инноваций.

На базовом уровне OpenGL — это просто *спецификация*, то есть документ, описывающий набор функций и их точное поведение. Производители оборудования на основе этой спецификации создают *реализации* — библиотеки функций, соответствующих набору функций спецификации. Реализация призвана эффективно использовать возможности оборудования.

# Описание реализации

Данная программа представляет собой реализацию операций над матрицами, таких как транспонирование, умножение и скалярное произведение. Она использует библиотеки thrust, cuBLAS и CUDA, для оптимального использования ресурсов GPU.

Программа, реализованная при помощи OpenGL, находится отдельно.

Для проверки правильности транспонирования и умножения, программа включает функции transpose\_check и multiply\_check, которые сравнивают результаты выполнения соответствующих операций с эталонными значениями.

Для транспонирования матриц используются следующие функции:

* transpose\_kernel, который выполняет транспонирование на GPU с помощью CUDA ядра.
* kernel\_transpose, который также использует CUDA ядро, но в отличие от предыдущей функции, он возвращает время выполнения операции.
* thrust\_transpose, который использует библиотеку thrust для выполнения транспонирования на GPU.
* cublas\_transpose, который использует библиотеку cuBLAS для выполнения транспонирования на GPU.

Для умножения матриц используются следующие функции:

* matrix\_multiply\_kernel, который выполняет умножение матриц на GPU с помощью CUDA ядра.
* kernel\_matrix\_multiply, который также использует CUDA ядро, но в отличие от предыдущей функции, он возвращает время выполнения операции.
* thrust\_matrix\_multiply, который использует библиотеку thrust для выполнения умножения матриц на GPU.
* cublas\_matrix\_multiply, который использует библиотеку cuBLAS для выполнения умножения матриц на GPU.

Для скалярного произведения используются следующие функции:

* scalar\_multiply\_kernel, который выполняет скалярное произведение на GPU с помощью CUDA ядра.
* kernel\_scalar\_multiply, который также использует CUDA ядро, но в отличие от предыдущей функции, он возвращает время выполнения операции.
* thrust\_scalar\_multiply, который использует библиотеку thrust для выполнения скалярного произведения на GPU.
* cublas\_scalar\_multiply, который использует библиотеку cuBLAS для выполнения скалярного произведения на GPU.

Функционал для OpenGL:

const char\* matrixMulShaderSource – инициализация шейдера для умножения.

const char\* matrixTransposeShaderSource – инициализация шейдера для транспонирования.

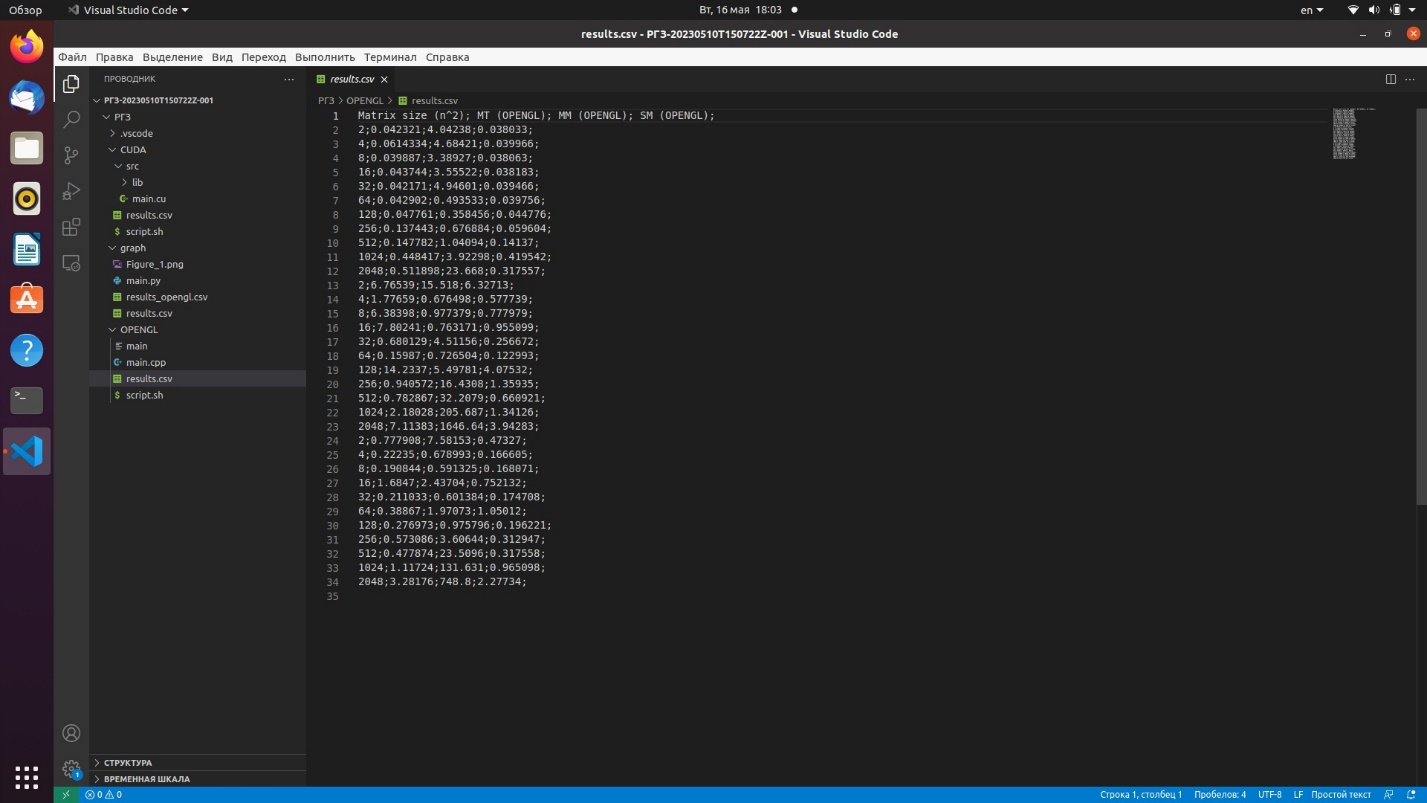
const char\* matrixScalarMulShaderSource – инициализация шейдера для скалярного умножения.

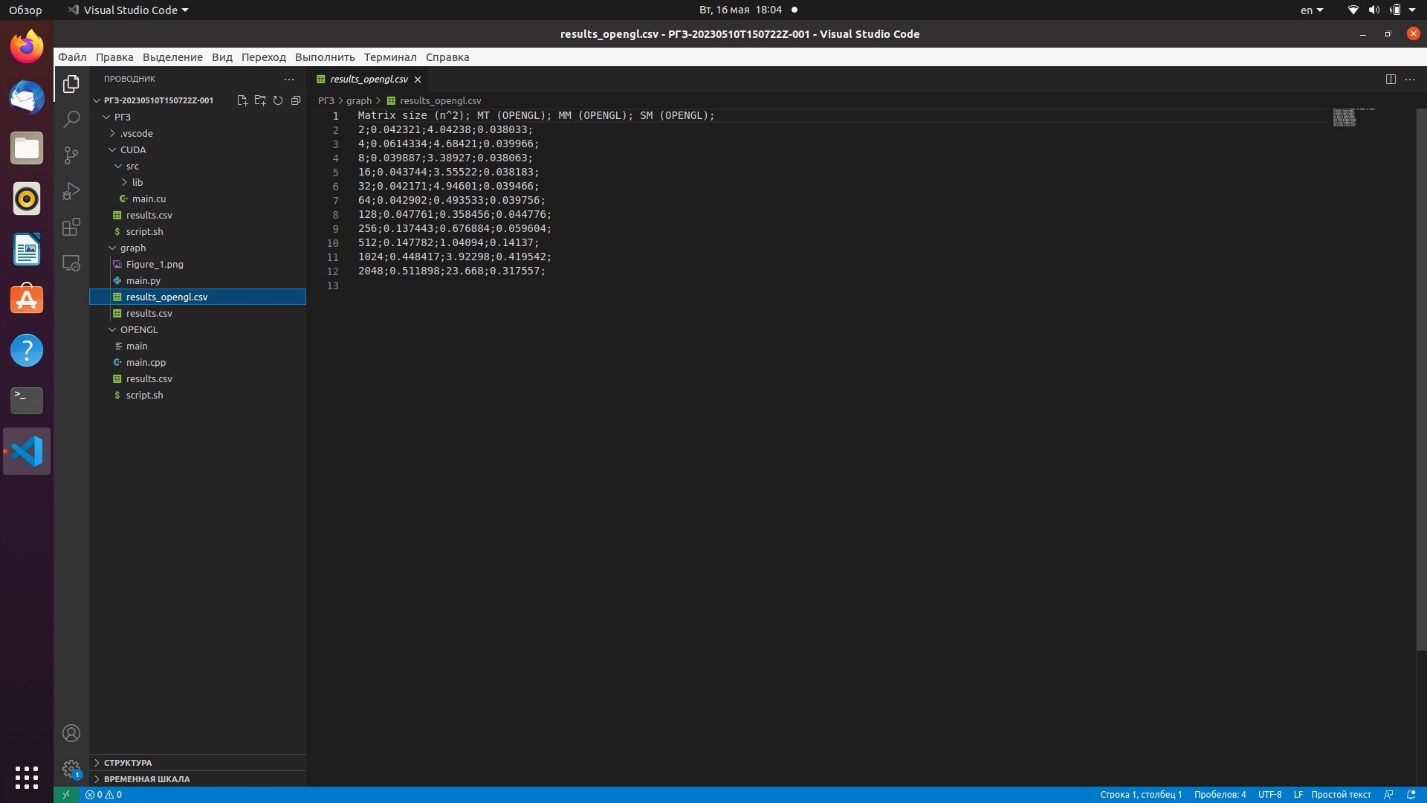
void checkShaderError(GLuint shader) – функция проверки ошибки компиляции шейдера.

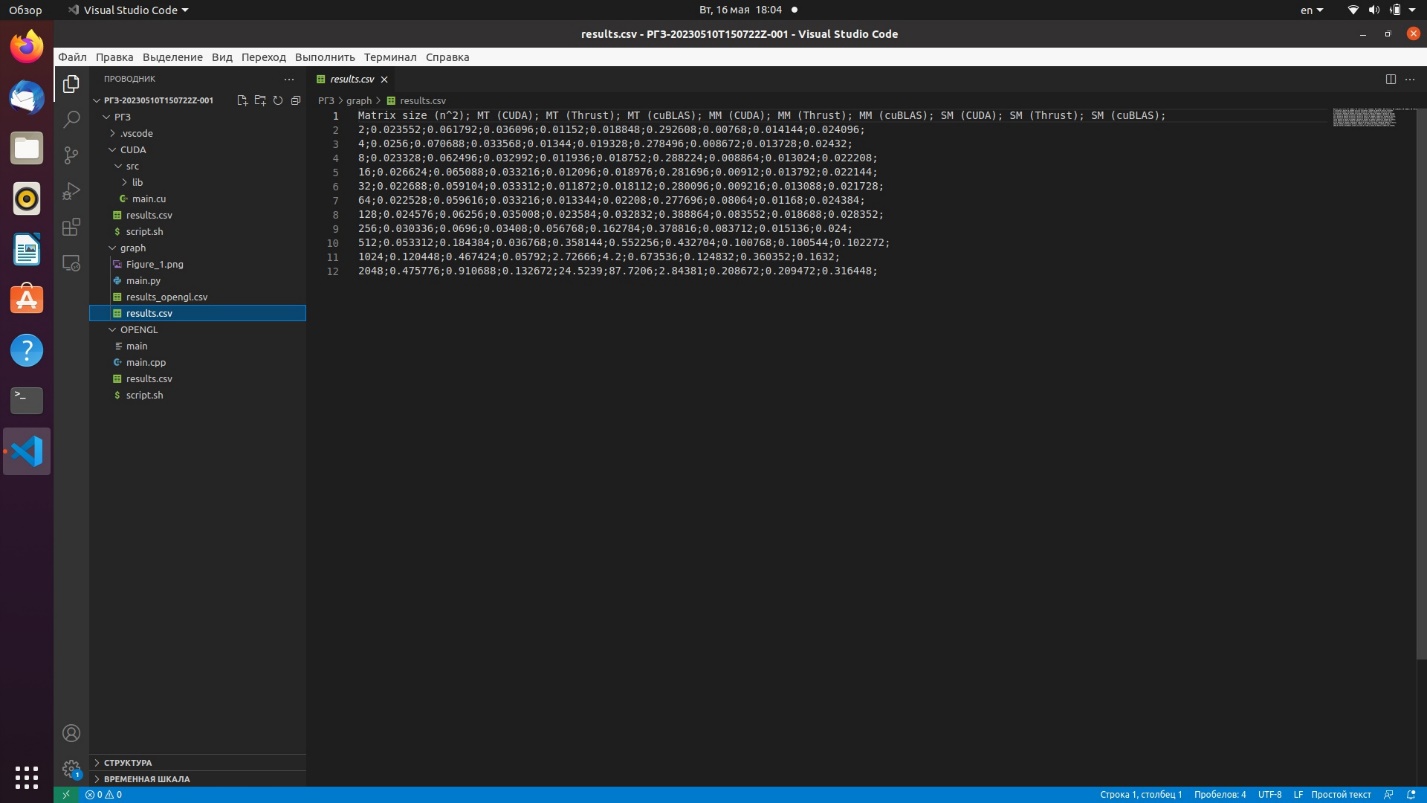
void checkProgramError(GLuint program) – функция проверки ошибки линковки.

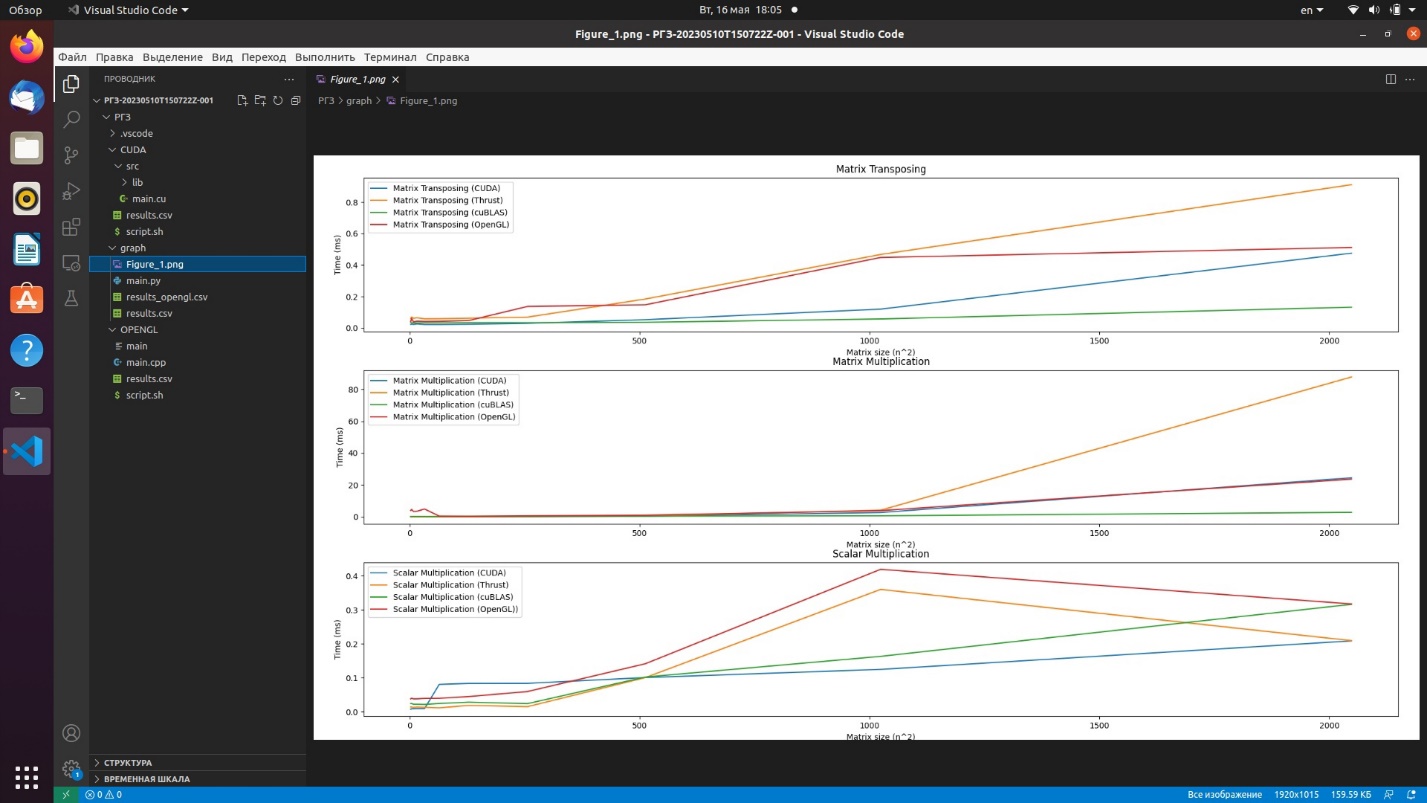
В функции main() происходит инициализация матриц для всех операций, создание и заполнение буферов, запуск вычислительного шейдера.

# Результаты работы программы









Результаты показывают, что использование cuBLAS и Thrust может значительно ускорить выполнение операций с матрицами на GPU, по сравнению с ручной реализацией на CUDA, OpenGL, в свою очередь, проигрывает, поскольку CUDA использует свои оптимизации на видеокарте.

# Листинг программы

utility.h

#ifndef UTILITY

#define UTILITY

#define MATRIX\_SIZE 512

#define NUM\_ELEMENTS (MATRIX\_SIZE \* MATRIX\_SIZE)

#define BLOCK\_SIZE 16 // programm crashes if > 22

#define TILE\_DIM (BLOCK\_SIZE \* 2)

#define SCALAR 10

#define PRINT\_MODE false  // change between true/false

#define FILE\_OUTPUT false // change between true/false

#endif

matrix\_handler.h

#ifndef MATRIX\_HANDLER

#define MATRIX\_HANDLER

#include <iostream>

#include "utility.h"

#define GREEN "\033[32m"

#define RED "\033[31m"

#define STANDART\_COLOR "\033[0m"

using namespace std;

void init\_matrix(float \*matrix)

{

    srand(time(NULL));

    for (int i = 0; i < NUM\_ELEMENTS; i++)

        matrix[i] = (float)(rand() % 90 + 10);

}

void init\_zero\_matrix(float \*matrix)

{

    for (int i = 0; i < NUM\_ELEMENTS; i++)

        matrix[i] = 0.0f;

}

void print\_matrix(float \*matrix)

{

    for (int i = 0; i < MATRIX\_SIZE; i++)

    {

        for (int j = 0; j < MATRIX\_SIZE; j++)

            cout << matrix[i \* MATRIX\_SIZE + j] << " ";

        cout << endl;

    }

    cout << endl;

}

#endif

cuda\_handler.h

#ifndef CUDA\_HANDLER

#define CUDA\_HANDLER

#include <thrust/device\_vector.h>

#include <thrust/iterator/zip\_iterator.h>

#include <thrust/tuple.h>

#include <thrust/functional.h>

#include <thrust/transform.h>

#include <thrust/execution\_policy.h>

#include <iostream>

#include <iomanip>

#include <cstdlib>

#include <cmath>

#include <cuda\_runtime.h>

#include <cublas\_v2.h>

#include "utility.h"

#include "matrix\_handler.h"

using namespace std;

// TRANSPOSING PART

void transpose\_check(float \*input\_matrix, float \*transposed\_matrix)

{

    bool is\_equal = true;

    float \*expected\_matrix = (float \*)malloc(NUM\_ELEMENTS \* sizeof(float));

    for (int i = 0; i < MATRIX\_SIZE; i++)

        for (int j = 0; j < MATRIX\_SIZE; j++)

            expected\_matrix[j \* MATRIX\_SIZE + i] = input\_matrix[i \* MATRIX\_SIZE + j];

    if (PRINT\_MODE)

    {

        cout << "INPUT MATRIX:" << endl;

        print\_matrix(input\_matrix);

        cout << "EXPECTED MATRIX:" << endl;

        print\_matrix(expected\_matrix);

        cout << "ACTUAL MATRIX:" << endl;

        print\_matrix(transposed\_matrix);

    }

    bool found\_mismatch = false;

    for (int i = 0; i < MATRIX\_SIZE; i++)

    {

        for (int j = 0; j < MATRIX\_SIZE; j++)

            if (transposed\_matrix[i \* MATRIX\_SIZE + j] != expected\_matrix[i \* MATRIX\_SIZE + j])

            {

                is\_equal = false;

                found\_mismatch = true;

                cout << "Mismatch found at index (" << i << ", " << j << ")" << endl;

                cout << "Expected: " << expected\_matrix[i \* MATRIX\_SIZE + j] << endl;

                cout << "Actual: " << transposed\_matrix[i \* MATRIX\_SIZE + j] << endl;

                break;

            }

        if (found\_mismatch)

            break;

    }

    if (is\_equal)

        cout << GREEN << "SUCCESS: Transposed matrix equal to expected" << STANDART\_COLOR << endl;

    else

        cout << RED << "FAILURE: Transposed matrix doesn't equal to expected" << STANDART\_COLOR << endl;

}

\_\_global\_\_ void transpose\_kernel(float \*input\_matrix, float \*output\_matrix)

{

    int x = blockIdx.x \* blockDim.x + threadIdx.x;

    int y = blockIdx.y \* blockDim.y + threadIdx.y;

    if (x < MATRIX\_SIZE && y < MATRIX\_SIZE)

    {

        int index\_in = y \* MATRIX\_SIZE + x;

        int index\_out = x \* MATRIX\_SIZE + y;

        output\_matrix[index\_out] = input\_matrix[index\_in];

    }

}

float kernel\_transpose(float \*input\_matrix, float \*output\_matrix)

{

    float elapsed\_time = 0.0f;

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start);

    dim3 dimGrid((MATRIX\_SIZE - 1) / BLOCK\_SIZE + 1, (MATRIX\_SIZE - 1) / BLOCK\_SIZE + 1);

    dim3 dimBlock(TILE\_DIM, BLOCK\_SIZE, 1);

    transpose\_kernel<<<dimGrid, dimBlock>>>(input\_matrix, output\_matrix);

    cudaEventRecord(stop);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsed\_time, start, stop);

    return elapsed\_time;

}

float thrust\_transpose(float \*input\_matrix, float \*output\_matrix)

{

    float elapsed\_time = 0.0f;

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start);

    thrust::device\_vector<float> input(NUM\_ELEMENTS);

    thrust::device\_vector<float> output(NUM\_ELEMENTS);

    // Copy input data to input vector

    thrust::copy(input\_matrix, input\_matrix + NUM\_ELEMENTS, input.begin());

    thrust::counting\_iterator<int> first(0);

    thrust::counting\_iterator<int> last(NUM\_ELEMENTS);

    auto input\_iter = thrust::make\_permutation\_iterator(input.begin(),

                                                        thrust::make\_transform\_iterator(first,

                                                                                        [=] \_\_device\_\_(int i) mutable

                                                                                        {

                                                                                            int r = i / MATRIX\_SIZE;

                                                                                            int c = i % MATRIX\_SIZE;

                                                                                            return c \* MATRIX\_SIZE + r;

                                                                                        }));

    thrust::copy(input\_iter, input\_iter + NUM\_ELEMENTS, output.begin());

    // Copy output data back to output\_matrix

    thrust::copy(output.begin(), output.end(), output\_matrix);

    cudaEventRecord(stop);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsed\_time, start, stop);

    return elapsed\_time;

}

float cublas\_transpose(float \*input\_matrix, float \*output\_matrix)

{

    float alpha = 1.0f;

    float beta = 0.0f;

    float elapsed\_time = 0.0f;

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cublasHandle\_t handle;

    cublasCreate(&handle);

    cudaEventRecord(start);

    cublasSgeam(handle, CUBLAS\_OP\_T, CUBLAS\_OP\_N, MATRIX\_SIZE, MATRIX\_SIZE, &alpha, input\_matrix, MATRIX\_SIZE, &beta, input\_matrix, MATRIX\_SIZE, output\_matrix, MATRIX\_SIZE);

    cudaEventRecord(stop);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsed\_time, start, stop);

    cublasDestroy(handle);

    return elapsed\_time;

}

// MULTIPLYING PART

void multiply\_check(float \*input\_matrix\_1, float \*input\_matrix\_2, float \*result\_matrix)

{

    bool is\_equal = true;

    float \*expected\_matrix = (float \*)malloc(NUM\_ELEMENTS \* sizeof(float));

    for (int i = 0; i < MATRIX\_SIZE; i++)

        for (int j = 0; j < MATRIX\_SIZE; j++)

        {

            float sum = 0.0f;

            for (int k = 0; k < MATRIX\_SIZE; k++)

                sum += input\_matrix\_1[i \* MATRIX\_SIZE + k] \* input\_matrix\_2[k \* MATRIX\_SIZE + j];

            expected\_matrix[i \* MATRIX\_SIZE + j] = sum;

        }

    if (PRINT\_MODE)

    {

        cout << "INPUT MATRIX 1:" << endl;

        print\_matrix(input\_matrix\_1);

        cout << "INPUT MATRIX 2:" << endl;

        print\_matrix(input\_matrix\_2);

        cout << "EXPECTED MATRIX:" << endl;

        print\_matrix(expected\_matrix);

        cout << "ACTUAL MATRIX:" << endl;

        print\_matrix(result\_matrix);

    }

    bool found\_mismatch = false;

    for (int i = 0; i < MATRIX\_SIZE; i++)

    {

        for (int j = 0; j < MATRIX\_SIZE; j++)

            if (fabs(result\_matrix[i \* MATRIX\_SIZE + j] - expected\_matrix[i \* MATRIX\_SIZE + j]) > 1e-6f)

            {

                is\_equal = false;

                found\_mismatch = true;

                cout << "Mismatch found at index (" << i << ", " << j << ")" << endl;

                cout << "Expected: " << expected\_matrix[i \* MATRIX\_SIZE + j] << endl;

                cout << "Actual: " << result\_matrix[i \* MATRIX\_SIZE + j] << endl;

                break;

            }

        if (found\_mismatch)

            break;

    }

    if (is\_equal)

        cout << GREEN << "SUCCESS: Result matrix equal to expected" << STANDART\_COLOR << endl;

    else

        cout << RED << "FAILURE: Result matrix doesn't equal to expected" << STANDART\_COLOR << endl;

}

\_\_global\_\_ void matrix\_multiply\_kernel(float \*input\_matrix\_1, float \*input\_matrix\_2, float \*output\_matrix)

{

    int row = blockIdx.y \* blockDim.y + threadIdx.y;

    int col = blockIdx.x \* blockDim.x + threadIdx.x;

    if (row < MATRIX\_SIZE && col < MATRIX\_SIZE)

    {

        float sum = 0.0f;

        for (int i = 0; i < MATRIX\_SIZE; i++)

        {

            sum += input\_matrix\_1[row \* MATRIX\_SIZE + i] \* input\_matrix\_2[i \* MATRIX\_SIZE + col];

        }

        output\_matrix[row \* MATRIX\_SIZE + col] = sum;

    }

}

float kernel\_matrix\_multiply(float \*input\_matrix\_1, float \*input\_matrix\_2, float \*output\_matrix)

{

    float elapsed\_time = 0.0f;

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start);

    dim3 dimGrid((MATRIX\_SIZE - 1) / BLOCK\_SIZE + 1, (MATRIX\_SIZE - 1) / BLOCK\_SIZE + 1);

    dim3 dimBlock(TILE\_DIM, BLOCK\_SIZE, 1);

    matrix\_multiply\_kernel<<<dimGrid, dimBlock>>>(input\_matrix\_1, input\_matrix\_2, output\_matrix);

    cudaEventRecord(stop);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsed\_time, start, stop);

    return elapsed\_time;

}

float thrust\_matrix\_multiply(float \*input\_matrix\_1, float \*input\_matrix\_2, float \*output\_matrix)

{

    float elapsed\_time = 0.0f;

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start);

    thrust::device\_ptr<float> d\_input\_matrix\_1(input\_matrix\_1);

    thrust::device\_ptr<float> d\_input\_matrix\_2(input\_matrix\_2);

    thrust::device\_ptr<float> d\_output\_matrix(output\_matrix);

    thrust::counting\_iterator<int> idx\_first(0);

    thrust::counting\_iterator<int> idx\_last = idx\_first + MATRIX\_SIZE \* MATRIX\_SIZE;

    thrust::transform(

        idx\_first, idx\_last,

        d\_output\_matrix,

        [d\_input\_matrix\_1, d\_input\_matrix\_2] \_\_device\_\_(int idx) -> float

        {

            int row = idx / MATRIX\_SIZE;

            int col = idx % MATRIX\_SIZE;

            float sum = 0.0f;

            for (int i = 0; i < MATRIX\_SIZE; i++)

            {

                sum += d\_input\_matrix\_1[row \* MATRIX\_SIZE + i] \* d\_input\_matrix\_2[i \* MATRIX\_SIZE + col];

            }

            return sum;

        });

    cudaEventRecord(stop);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsed\_time, start, stop);

    return elapsed\_time;

}

float cublas\_matrix\_multiply(float \*input\_matrix\_1, float \*input\_matrix\_2, float \*output\_matrix)

{

    float alpha = 1.0f;

    float beta = 0.0f;

    cublasHandle\_t handle;

    cublasCreate(&handle);

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start);

    cublasSgemm(handle, CUBLAS\_OP\_N, CUBLAS\_OP\_N, MATRIX\_SIZE, MATRIX\_SIZE, MATRIX\_SIZE, &alpha, input\_matrix\_2, MATRIX\_SIZE, input\_matrix\_1, MATRIX\_SIZE, &beta, output\_matrix, MATRIX\_SIZE);

    cudaEventRecord(stop);

    cudaEventSynchronize(stop);

    float elapsed\_time;

    cudaEventElapsedTime(&elapsed\_time, start, stop);

    cublasDestroy(handle);

    return elapsed\_time;

}

/\*Scalar multiplication of matrices is the operation of multiplying each element of a matrix by a scalar value (a single number).

In other words, given a matrix A and a scalar k, the scalar multiplication of A and k is a new matrix C

where each element of A is multiplied by k to get the corresponding element of C. For example, if A = [1 2; 3 4] and k = 2,

then the scalar multiplication of A and k is C = [2 4; 6 8].

A scalar is a unit matrix with a necessary number on the main diagonal.

For example, if scalar k = 2 and width of matrix A is 2, then k => k \* E\_2 => [2 0; 0 2]\*/

void scalar\_check(float \*input\_matrix, float \*result\_matrix)

{

    bool is\_equal = true;

    float \*expected\_matrix = (float \*)malloc(NUM\_ELEMENTS \* sizeof(float));

    for (int i = 0; i < NUM\_ELEMENTS; i++)

        expected\_matrix[i] = SCALAR \* input\_matrix[i];

    if (PRINT\_MODE)

    {

        cout << "INPUT MATRIX:" << endl;

        print\_matrix(input\_matrix);

        cout << "SCALAR = " << SCALAR << endl;

        cout << "EXPECTED MATRIX:" << endl;

        print\_matrix(expected\_matrix);

        cout << "ACTUAL MATRIX:" << endl;

        print\_matrix(result\_matrix);

    }

    bool found\_mismatch = false;

    for (int i = 0; i < MATRIX\_SIZE; i++)

    {

        for (int j = 0; j < MATRIX\_SIZE; j++)

            if (result\_matrix[i \* MATRIX\_SIZE + j] != expected\_matrix[i \* MATRIX\_SIZE + j])

            {

                is\_equal = false;

                found\_mismatch = true;

                cout << "Mismatch found at index (" << i << ", " << j << ")" << endl;

                cout << "Expected: " << expected\_matrix[i \* MATRIX\_SIZE + j] << endl;

                cout << "Actual: " << result\_matrix[i \* MATRIX\_SIZE + j] << endl;

                break;

            }

        if (found\_mismatch)

            break;

    }

    if (is\_equal)

        cout << GREEN << "SUCCESS: Result matrix equal to expected" << STANDART\_COLOR << endl;

    else

        cout << RED << "FAILURE: Result matrix doesn't equal to expected" << STANDART\_COLOR << endl;

}

\_\_global\_\_ void scalar\_multiply\_kernel(float \*input\_matrix, float \*output\_matrix)

{

    int x = blockIdx.x \* blockDim.x + threadIdx.x;

    int y = blockIdx.y \* blockDim.y + threadIdx.y;

    if (x < MATRIX\_SIZE && y < MATRIX\_SIZE)

        output\_matrix[y \* MATRIX\_SIZE + x] = input\_matrix[y \* MATRIX\_SIZE + x] \* SCALAR;

}

float kernel\_scalar\_multiply(float \*input\_matrix, float \*output\_matrix)

{

    float elapsed\_time = 0.0f;

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start);

    dim3 dimGrid((MATRIX\_SIZE - 1) / BLOCK\_SIZE + 1, (MATRIX\_SIZE - 1) / BLOCK\_SIZE + 1);

    dim3 dimBlock(TILE\_DIM, BLOCK\_SIZE, 1);

    scalar\_multiply\_kernel<<<dimGrid, dimBlock>>>(input\_matrix, output\_matrix);

    cudaEventRecord(stop);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsed\_time, start, stop);

    return elapsed\_time;

}

float thrust\_scalar\_multiply(float \*input\_matrix, float \*output\_matrix)

{

    float elapsed\_time = 0.0f;

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start);

    thrust::device\_ptr<float> d\_input\_matrix(input\_matrix);

    thrust::device\_ptr<float> d\_output\_matrix(output\_matrix);

    thrust::counting\_iterator<int> idx\_first(0);

    thrust::counting\_iterator<int> idx\_last = idx\_first + MATRIX\_SIZE \* MATRIX\_SIZE;

    thrust::transform(

        idx\_first, idx\_last,

        d\_output\_matrix,

        [d\_input\_matrix] \_\_device\_\_(int idx) -> float

        {

            return d\_input\_matrix[idx] \* SCALAR;

        });

    cudaEventRecord(stop);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsed\_time, start, stop);

    return elapsed\_time;

}

float cublas\_scalar\_multiply(float \*input\_matrix, float \*output\_matrix)

{

    cublasHandle\_t handle;

    cublasCreate(&handle);

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start);

    float scalar = static\_cast<float>(SCALAR);

    cublasSscal(handle, MATRIX\_SIZE \* MATRIX\_SIZE, &scalar, input\_matrix, 1);

    cudaMemcpy(output\_matrix, input\_matrix, MATRIX\_SIZE \* MATRIX\_SIZE \* sizeof(float), cudaMemcpyDeviceToDevice);

    cudaEventRecord(stop);

    cudaEventSynchronize(stop);

    float elapsed\_time;

    cudaEventElapsedTime(&elapsed\_time, start, stop);

    cublasDestroy(handle);

    return elapsed\_time;

}

#endif

main.cu

#include <iostream>

#include <iomanip>

#include <stdio.h>

#include <stdlib.h>

#include <cstdlib>

#include <chrono>

#include <thread>

#include <fstream>

#include "utility.h"

#include "cuda\_handler.h"

#include "matrix\_handler.h"

using namespace std;

int main()

{

    float \*input\_matrix\_1, \*input\_matrix\_2, \*output\_matrix;

    input\_matrix\_1 = (float \*)malloc(NUM\_ELEMENTS \* sizeof(float));

    input\_matrix\_2 = (float \*)malloc(NUM\_ELEMENTS \* sizeof(float));

    output\_matrix = (float \*)malloc(NUM\_ELEMENTS \* sizeof(float));

    init\_matrix(input\_matrix\_1);

    this\_thread::sleep\_for(chrono::seconds(1));

    init\_matrix(input\_matrix\_2);

    init\_zero\_matrix(output\_matrix);

    if (PRINT\_MODE)

    {

        cout << "BASE MATRICES:" << endl;

        cout << "input\_matrix\_1:" << endl;

        print\_matrix(input\_matrix\_1);

        cout << "input\_matrix\_2:" << endl;

        print\_matrix(input\_matrix\_2);

        cout << "output\_matrix:" << endl;

        print\_matrix(output\_matrix);

    }

    float \*device\_input\_matrix\_1, \*device\_input\_matrix\_2, \*device\_output\_matrix;

    cudaMalloc(&device\_input\_matrix\_1, NUM\_ELEMENTS \* sizeof(float));

    cudaMalloc(&device\_input\_matrix\_2, NUM\_ELEMENTS \* sizeof(float));

    cudaMalloc(&device\_output\_matrix, NUM\_ELEMENTS \* sizeof(float));

    cudaMemcpy(device\_input\_matrix\_1, input\_matrix\_1, NUM\_ELEMENTS \* sizeof(float), cudaMemcpyHostToDevice);

    cudaMemcpy(device\_input\_matrix\_2, input\_matrix\_2, NUM\_ELEMENTS \* sizeof(float), cudaMemcpyHostToDevice);

    cout << "KERNEL TRANSPOSING:" << endl;

    init\_zero\_matrix(output\_matrix);

    float elapsed\_time\_kernel\_transpose = 0.0f;

    elapsed\_time\_kernel\_transpose = kernel\_transpose(device\_input\_matrix\_1, device\_output\_matrix);

    cudaMemcpy(output\_matrix, device\_output\_matrix, NUM\_ELEMENTS \* sizeof(float), cudaMemcpyDeviceToHost);

    transpose\_check(input\_matrix\_1, output\_matrix);

    cout << "Time: " << elapsed\_time\_kernel\_transpose << " ms" << endl;

    cout << endl;

    cout << "THRUST TRANSPOSING:" << endl;

    init\_zero\_matrix(output\_matrix);

    float elapsed\_time\_thrust\_transpose = 0.0f;

    elapsed\_time\_thrust\_transpose = thrust\_transpose(device\_input\_matrix\_1, device\_output\_matrix);

    cudaMemcpy(output\_matrix, device\_output\_matrix, NUM\_ELEMENTS \* sizeof(float), cudaMemcpyDeviceToHost);

    transpose\_check(input\_matrix\_1, output\_matrix);

    cout << "Time: " << elapsed\_time\_thrust\_transpose << " ms" << endl;

    cout << endl;

    cout << "CUBLAS TRANSPOSING:" << endl;

    init\_zero\_matrix(output\_matrix);

    float elapsed\_time\_cublas\_transpose = 0.0f;

    elapsed\_time\_cublas\_transpose = cublas\_transpose(device\_input\_matrix\_1, device\_output\_matrix);

    cudaMemcpy(output\_matrix, device\_output\_matrix, NUM\_ELEMENTS \* sizeof(float), cudaMemcpyDeviceToHost);

    transpose\_check(input\_matrix\_1, output\_matrix);

    cout << "Time: " << elapsed\_time\_cublas\_transpose << " ms" << endl;

    cout << endl;

    cout << "KERNEL MATRIX MULTIPLICATION:" << endl;

    init\_zero\_matrix(output\_matrix);

    float elapsed\_time\_kernel\_matrix\_multiplication = 0.0f;

    elapsed\_time\_kernel\_matrix\_multiplication = kernel\_matrix\_multiply(device\_input\_matrix\_1, device\_input\_matrix\_2, device\_output\_matrix);

    cudaMemcpy(output\_matrix, device\_output\_matrix, NUM\_ELEMENTS \* sizeof(float), cudaMemcpyDeviceToHost);

    multiply\_check(input\_matrix\_1, input\_matrix\_2, output\_matrix);

    cout << "Time: " << elapsed\_time\_kernel\_matrix\_multiplication << " ms" << endl;

    cout << endl;

    cout << "THRUST MATRIX MULTIPLICATION:" << endl;

    init\_zero\_matrix(output\_matrix);

    float elapsed\_time\_thrust\_matrix\_multiplication = 0.0f;

    elapsed\_time\_thrust\_matrix\_multiplication = thrust\_matrix\_multiply(device\_input\_matrix\_1, device\_input\_matrix\_2, device\_output\_matrix);

    cudaMemcpy(output\_matrix, device\_output\_matrix, NUM\_ELEMENTS \* sizeof(float), cudaMemcpyDeviceToHost);

    multiply\_check(input\_matrix\_1, input\_matrix\_2, output\_matrix);

    cout << "Time: " << elapsed\_time\_thrust\_matrix\_multiplication << " ms" << endl;

    cout << endl;

    cout << "CUBLAS MATRIX MULTIPLICATION:" << endl;

    init\_zero\_matrix(output\_matrix);

    float elapsed\_time\_cublas\_matrix\_multiplication = 0.0f;

    elapsed\_time\_cublas\_matrix\_multiplication = cublas\_matrix\_multiply(device\_input\_matrix\_1, device\_input\_matrix\_2, device\_output\_matrix);

    cudaMemcpy(output\_matrix, device\_output\_matrix, NUM\_ELEMENTS \* sizeof(float), cudaMemcpyDeviceToHost);

    multiply\_check(input\_matrix\_1, input\_matrix\_2, output\_matrix);

    cout << "Time: " << elapsed\_time\_cublas\_matrix\_multiplication << " ms" << endl;

    cout << endl;

    cout << "KERNEL SCALAR MULTIPLICATION:" << endl;

    init\_zero\_matrix(output\_matrix);

    float elapsed\_time\_kernel\_scalar\_multiplication = 0.0f;

    elapsed\_time\_kernel\_scalar\_multiplication = kernel\_scalar\_multiply(device\_input\_matrix\_1, device\_output\_matrix);

    cudaMemcpy(output\_matrix, device\_output\_matrix, NUM\_ELEMENTS \* sizeof(float), cudaMemcpyDeviceToHost);

    scalar\_check(input\_matrix\_1, output\_matrix);

    cout << "Time: " << elapsed\_time\_kernel\_scalar\_multiplication << " ms" << endl;

    cout << endl;

    cout << "THRUST SCALAR MULTIPLICATION:" << endl;

    init\_zero\_matrix(output\_matrix);

    float elapsed\_time\_thrust\_scalar\_multiplication = 0.0f;

    elapsed\_time\_thrust\_scalar\_multiplication = thrust\_scalar\_multiply(device\_input\_matrix\_1, device\_output\_matrix);

    cudaMemcpy(output\_matrix, device\_output\_matrix, NUM\_ELEMENTS \* sizeof(float), cudaMemcpyDeviceToHost);

    scalar\_check(input\_matrix\_1, output\_matrix);

    cout << "Time: " << elapsed\_time\_thrust\_scalar\_multiplication << " ms" << endl;

    cout << endl;

    cout << "CUBLAS SCALAR MULTIPLICATION:" << endl;

    init\_zero\_matrix(output\_matrix);

    float elapsed\_time\_cublas\_scalar\_multiplication = 0.0f;

    elapsed\_time\_cublas\_scalar\_multiplication = cublas\_scalar\_multiply(device\_input\_matrix\_1, device\_output\_matrix);

    cudaMemcpy(output\_matrix, device\_output\_matrix, NUM\_ELEMENTS \* sizeof(float), cudaMemcpyDeviceToHost);

    scalar\_check(input\_matrix\_1, output\_matrix);

    cout << "Time: " << elapsed\_time\_cublas\_scalar\_multiplication << " ms" << endl;

    cout << endl;

    cout << "TOTAL RESULTS:" << endl;

    cout << "MATRIX SIZE = " << MATRIX\_SIZE << " x " << MATRIX\_SIZE << " (" << NUM\_ELEMENTS << " elements)" << endl;

    cout << setw(40) << "raw CUDA, ms" << setw(15) << "Thrust, ms"

         << setw(15) << "cuBLSAS, ms" << endl;

    cout << fixed;

    cout << setw(25) << "Matrix transposing" << setw(15) << elapsed\_time\_kernel\_transpose << setw(15) << elapsed\_time\_thrust\_transpose << setw(15) << elapsed\_time\_cublas\_transpose << endl;

    cout << setw(25) << "Matrix multiplication" << setw(15) << elapsed\_time\_kernel\_matrix\_multiplication << setw(15) << elapsed\_time\_thrust\_matrix\_multiplication << setw(15) << elapsed\_time\_cublas\_matrix\_multiplication << endl;

    cout << setw(25) << "Scalar multiplication" << setw(15) << elapsed\_time\_kernel\_scalar\_multiplication << setw(15) << elapsed\_time\_thrust\_scalar\_multiplication << setw(15) << elapsed\_time\_cublas\_scalar\_multiplication << endl;

    if (FILE\_OUTPUT)

    {

        ofstream file;

        file.open("results.csv", ios::app);

        if (file.tellp() == streampos(0))

        {

            file << "Matrix size (n^2); MT (CUDA); MT (Thrust); MT (cuBLAS); MM (CUDA); MM (Thrust); MM (cuBLAS); SM (CUDA); SM (Thrust); SM (cuBLAS);" << endl;

        }

        file << MATRIX\_SIZE << ";"

             << elapsed\_time\_kernel\_transpose << ";" << elapsed\_time\_thrust\_transpose << ";" << elapsed\_time\_cublas\_transpose << ";"

             << elapsed\_time\_kernel\_matrix\_multiplication << ";" << elapsed\_time\_thrust\_matrix\_multiplication << ";" << elapsed\_time\_cublas\_matrix\_multiplication << ";"

             << elapsed\_time\_kernel\_scalar\_multiplication << ";" << elapsed\_time\_thrust\_scalar\_multiplication << ";" << elapsed\_time\_cublas\_scalar\_multiplication << ";"

             << endl;

        file.close();

    }

    return EXIT\_SUCCESS;

}

//OpenGL

#include <iostream>

#include <vector>

#include <chrono>

#include <GL/glew.h>

#include <GLFW/glfw3.h>

#include <fstream>

using namespace std;

const char\* matrixMulShaderSource = R"(

#version 430

layout (local\_size\_x = 16, local\_size\_y = 16) in;

layout (std430, binding = 0) buffer A {

    float a[];

};

layout (std430, binding = 1) buffer B {

    float b[];

};

layout (std430, binding = 2) buffer C {

    float c[];

};

uniform int size;

void main() {

    ivec2 gid = ivec2(gl\_GlobalInvocationID.xy);

    if (gid.x < size && gid.y < size) {

        float sum = 0.0;

        for (int k = 0; k < size; k++) {

            sum += a[gid.y \* size + k] \* b[k \* size + gid.x];

        }

        c[gid.y \* size + gid.x] = sum;

    }

}

)";

const char\* matrixTransposeShaderSource = R"(

#version 430

layout (local\_size\_x = 16, local\_size\_y = 16) in;

layout (std430, binding = 0) buffer A {

    float a[];

};

layout (std430, binding = 1) buffer B {

    float b[];

};

uniform int size;

void main() {

    ivec2 gid = ivec2(gl\_GlobalInvocationID.xy);

    if (gid.x < size && gid.y < size) {

        b[gid.y \* size + gid.x] = a[gid.x \* size + gid.y];

    }

}

)";

const char\* matrixScalarMulShaderSource = R"(

#version 430

layout (local\_size\_x = 16, local\_size\_y = 16) in;

layout (std430, binding = 0) buffer A {

    float a[];

};

layout (std430, binding = 1) buffer B {

    float b[];

};

uniform float scalar;

uniform int size;

void main() {

    ivec2 gid = ivec2(gl\_GlobalInvocationID.xy);

    if (gid.x < size && gid.y < size) {

        b[gid.y \* size + gid.x] = a[gid.y \* size + gid.x] \* scalar;

    }

}

)";

void checkShaderError(GLuint shader) {

    GLint success;

    glGetShaderiv(shader, GL\_COMPILE\_STATUS, &success);

    if (!success) {

        char infoLog[512];

        glGetShaderInfoLog(shader, 512, nullptr, infoLog);

        std::cerr << "Shader compilation failed:\n" << infoLog << std::endl;

    }

}

void checkProgramError(GLuint program) {

    GLint success;

    glGetProgramiv(program, GL\_LINK\_STATUS, &success);

    if (!success) {

        char infoLog[512];

        glGetProgramInfoLog(program, 512, nullptr, infoLog);

        std::cerr << "Shader program linking failed:\n" << infoLog << std::endl;

    }

}

int main(int argc, char\*\* argv) {

    if (argc < 2) {

        return -1;

    }

    int matrixSize = atoi(argv[1]);

    // Инициализация GLFW и GLEW

    if (!glfwInit()) {

        std::cerr << "GLFW initialization failed" << std::endl;

        return -1;

    }

    glfwWindowHint(GLFW\_VISIBLE, GLFW\_FALSE);

    GLFWwindow\* window = glfwCreateWindow(1, 1, "", nullptr, nullptr);

    if (!window) {

        std::cerr << "GLFW window creation failed" << std::endl;

        glfwTerminate();

        return -1;

    }

    glfwMakeContextCurrent(window);

    if (glewInit() != GLEW\_OK) {

        std::cerr << "GLEW initialization failed" << std::endl;

        glfwTerminate();

        return -1;

    }

    // Создание шейдерной программы

    GLuint shader = glCreateShader(GL\_COMPUTE\_SHADER);

    glShaderSource(shader, 1, &matrixMulShaderSource, nullptr);

    glCompileShader(shader);

    checkShaderError(shader);

    GLuint program = glCreateProgram();

    glAttachShader(program, shader);

    glLinkProgram(program);

    checkProgramError(program);

    glUseProgram(program);

    glDeleteShader(shader);

    // Размер матрицы и количество экспериментов

    // Инициализация матриц A и B с случайными значениями

    std::vector<float> a(matrixSize \* matrixSize), b(matrixSize \* matrixSize);

    for (int i = 0; i < matrixSize \* matrixSize; i++) {

        a[i] = static\_cast<float>(rand()) / RAND\_MAX;

        b[i] = static\_cast<float>(rand()) / RAND\_MAX;

    }

    // Создание и заполнение буферов

    GLuint buffers[3];

    glGenBuffers(3, buffers);

    glBindBufferBase(GL\_SHADER\_STORAGE\_BUFFER, 0, buffers[0]);

    glBufferData(GL\_SHADER\_STORAGE\_BUFFER, matrixSize \* matrixSize \* sizeof(float), a.data(), GL\_DYNAMIC\_DRAW);

    glBindBufferBase(GL\_SHADER\_STORAGE\_BUFFER, 1, buffers[1]);

    glBufferData(GL\_SHADER\_STORAGE\_BUFFER, matrixSize \* matrixSize \* sizeof(float), b.data(), GL\_DYNAMIC\_DRAW);

    glBindBufferBase(GL\_SHADER\_STORAGE\_BUFFER, 2, buffers[2]);

    glBufferData(GL\_SHADER\_STORAGE\_BUFFER, matrixSize \* matrixSize \* sizeof(float), nullptr, GL\_DYNAMIC\_DRAW);

    // Запуск вычислительного шейдера и измерение времени выполнения

    glUseProgram(program);

    glUniform1i(glGetUniformLocation(program, "size"), matrixSize);

    glMemoryBarrier(GL\_SHADER\_STORAGE\_BARRIER\_BIT);

    std::chrono::high\_resolution\_clock::time\_point start = std::chrono::high\_resolution\_clock::now();

    glDispatchCompute((matrixSize + 15) / 16, (matrixSize + 15) / 16, 1);

    glMemoryBarrier(GL\_SHADER\_STORAGE\_BARRIER\_BIT);

    glFinish();

    std::chrono::high\_resolution\_clock::time\_point end = std::chrono::high\_resolution\_clock::now();

    std::chrono::duration<double, std::milli> elapsed\_matrix\_multiplication = end - start;

    glDeleteProgram(program);

    glDeleteBuffers(3, buffers);

    shader = glCreateShader(GL\_COMPUTE\_SHADER);

    glShaderSource(shader, 1, &matrixTransposeShaderSource, nullptr);

    glCompileShader(shader);

    checkShaderError(shader);

    program = glCreateProgram();

    glAttachShader(program, shader);

    glLinkProgram(program);

    checkProgramError(program);

    glUseProgram(program);

    glDeleteShader(shader);

    glGenBuffers(2, buffers);

    glBindBufferBase(GL\_SHADER\_STORAGE\_BUFFER, 0, buffers[0]);

    glBufferData(GL\_SHADER\_STORAGE\_BUFFER, matrixSize \* matrixSize \* sizeof(float), a.data(), GL\_DYNAMIC\_DRAW);

    glBindBufferBase(GL\_SHADER\_STORAGE\_BUFFER, 1, buffers[1]);

    glBufferData(GL\_SHADER\_STORAGE\_BUFFER, matrixSize \* matrixSize \* sizeof(float), nullptr, GL\_DYNAMIC\_DRAW);

    glUseProgram(program);

    glUniform1i(glGetUniformLocation(program, "size"), matrixSize);

    glMemoryBarrier(GL\_SHADER\_STORAGE\_BARRIER\_BIT);

    start = std::chrono::high\_resolution\_clock::now();

    glDispatchCompute((matrixSize + 15) / 16, (matrixSize + 15) / 16, 1);

    glMemoryBarrier(GL\_SHADER\_STORAGE\_BARRIER\_BIT);

    glFinish();

    end = std::chrono::high\_resolution\_clock::now();

    std::chrono::duration<double, std::milli> elapsed\_matrix\_tanspose = end - start;

    glDeleteProgram(program);

    glDeleteBuffers(3, buffers);

    shader = glCreateShader(GL\_COMPUTE\_SHADER);

    glShaderSource(shader, 1, &matrixScalarMulShaderSource, nullptr);

    glCompileShader(shader);

    checkShaderError(shader);

    program = glCreateProgram();

    glAttachShader(program, shader);

    glLinkProgram(program);

    checkProgramError(program);

    glUseProgram(program);

    glDeleteShader(shader);

    glGenBuffers(2, buffers);

    glBindBufferBase(GL\_SHADER\_STORAGE\_BUFFER, 0, buffers[0]);

    glBufferData(GL\_SHADER\_STORAGE\_BUFFER, matrixSize \* matrixSize \* sizeof(float), a.data(), GL\_DYNAMIC\_DRAW);

    glBindBufferBase(GL\_SHADER\_STORAGE\_BUFFER, 1, buffers[1]);

    glBufferData(GL\_SHADER\_STORAGE\_BUFFER, matrixSize \* matrixSize \* sizeof(float), nullptr, GL\_DYNAMIC\_DRAW);

    const float scalar = 2.0f;

    glUseProgram(program);

    glUniform1f(glGetUniformLocation(program, "scalar"), scalar);

    glUniform1i(glGetUniformLocation(program, "size"), matrixSize);

    glMemoryBarrier(GL\_SHADER\_STORAGE\_BARRIER\_BIT);

    start = std::chrono::high\_resolution\_clock::now();

    glDispatchCompute((matrixSize + 15) / 16, (matrixSize + 15) / 16, 1);

    glMemoryBarrier(GL\_SHADER\_STORAGE\_BARRIER\_BIT);

    glFinish();

    end = std::chrono::high\_resolution\_clock::now();

    std::chrono::duration<double, std::milli> elapsed\_matrix\_scalar = end - start;

    ofstream file;

    file.open("results.csv", ios::app);

    if (file.tellp() == streampos(0))

    {

            file << "Matrix size (n^2); MT (OPENGL); MM (OPENGL); SM (OPENGL);" << "\n";

    }

    file

        << matrixSize << ";"

        << elapsed\_matrix\_tanspose.count() << ";"

        << elapsed\_matrix\_multiplication.count() << ";"

        << elapsed\_matrix\_scalar.count() << ";"

        << "\n";

    file.close();

    // Освобождение ресурсов и завершение работы

    glDeleteProgram(program);

    glDeleteBuffers(3, buffers);

    glfwTerminate();

    return 0;

}