Федеральное государственное бюджетное образовательное учреждение высшего образования.
«Национально исследовательский университет «Московский энергетический институт»
Кафедра ВМСС

Лабораторная работа №3

ПРОГРАММИРОВАНИЕ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ СРЕДСТВАМИ NVIDIA CUDA. ВВЕДЕНИЕ В ИСПОЛЬЗОВАНИЕ GPU В КАЧЕСТВЕ СОПРОЦЕССОРА ДЛЯ УСКОРЕНИЯ ВЫЧИСЛЕНИЙ

Курс: Вычислительные системы

Группа: А-07м-23

Выполнили: Балашов С.А.,
Кретов Н.В.

Проверил: Филатов А.В.

Москва 2023 г.

Задание (2 вариант):

2	$Y_i = (A_i + C_i)/(B^m{}_i - D_i)$

1. Параметры GPU:
Device 0: "NVIDIA GeForce RTX 2060”
Версия драйвера CUDA: 12.3
Версия CUDA Runtime: 12.0
Оценка CUDA возможностей видеокарты от NVIDIA: 7.5
Всего видеопамяти: 6144 МБ (6442123264 Б)
Максимальная частота карты: 1680 МГц (1.68 ГГц)
Максимальная частота памяти: 7001 МГц
Ширина шины памяти: 192-битная
Размер кэша L2: 3145728 байт
Максимальный размер текстур (x,y,z): 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
Размер многослойной текстуры: 1D=(32768), 2048 слоев, 2D=(32768, 32768), 2048 слоев
Всего постоянной памяти: 65536 байт
Всего общей памяти на блок: 49152 байт
Всего общей памяти на мультипроцессор: 65536 байт
Всего регистров на блок: 65536
Размер варпа: 32
Всего потоков на мультипроцессор: 1024
Всего потоков на блок: 1024
Максимальная размерность блока (x,y,z): (1024, 1024, 64)
Максимальная размерность сетки (x,y,z): (2147483647, 65535, 65535)

2. Времена выполнения на CPU
Полный код программы представлен в приложении А.

Таблица 1.

Время выполнения расчётов на CPU

Размер массивов	Время (мс)
16384	0.4851
65536	1.8836

3. Времена выполнения на 1 блоке с 1 потоком
16384: 50.9027 мс
65536: 148.887 мс

4. Времена выполнения при разных размерах

Таблица 2.

Время выполнения на GPU

Размер массивов	Размер блока	Количество блоков	Время (мс)
16384	4	1	14.7146
16384	32	1	1.8824
16384	64	1	1.00269
16384	256	1	0.673888
16384	512	1	0.6776
16384	1	2	28.6583
16384	4	2	10.2727
16384	32	2	0.9728
16384	64	2	0.464896
16384	256	2	0.413696
16384	512	2	0.39264
16384	1	4	17.1703
16384	4	4	3.01686
16384	32	4	0.443552
16384	64	4	0.239616
16384	256	4	0.215328
16384	512	4	0.202752
16384	1	8	5.97747
16384	4	8	1.5352
16384	32	8	0.254464
16384	64	8	0.168704
16384	256	8	0.108544
16384	512	8	0.139264
16384	1	16	4.00355
16384	4	16	0.786432
16384	32	16	0.1392
16384	64	16	0.097824

16384	256	16	0.084
16384	512	16	0.070848
65536	4	1	56.9535
65536	32	1	7.14384
65536	64	1	2.83034
65536	256	1	2.68902
65536	512	1	1.86557
65536	1	2	81.5273
65536	4	2	20.1626
65536	32	2	2.62451
65536	64	2	1.21037
65536	256	2	0.961728
65536	512	2	0.955424
65536	1	4	40.1797
65536	4	4	9.23226
65536	32	4	1.34518
65536	64	4	0.623968
65536	256	4	0.502496
65536	512	4	0.49152
65536	1	8	20.5279
65536	4	8	4.21888
65536	32	8	1.71766
65536	64	8	0.343872
65536	256	8	0.251264
65536	512	8	0.259328
65536	1	16	9.20029
65536	4	16	2.12138
65536	32	16	0.319488
65536	64	16	0.229216
65536	256	16	0.139808
65536	512	16	0.174208

5. Коэффициенты ускорения

Таблица 4.

Коэффициент ускорения к CPU (16384)

Размер блока	Количество блоков	Время (мс)	Ускорение
CPU	CPU	0.4851	1
256	1	0.673888	0.719852557101477
512	1	0.6776	0.7159
256	2	0.413696	1.17260017017327
512	2	0.39264	1.23548288508557
256	4	0.215328	2.2528421756576
512	4	0.202752	2.392578125
256	8	0.108544	4.46915536556604
512	8	0.139264	3.48331227022059

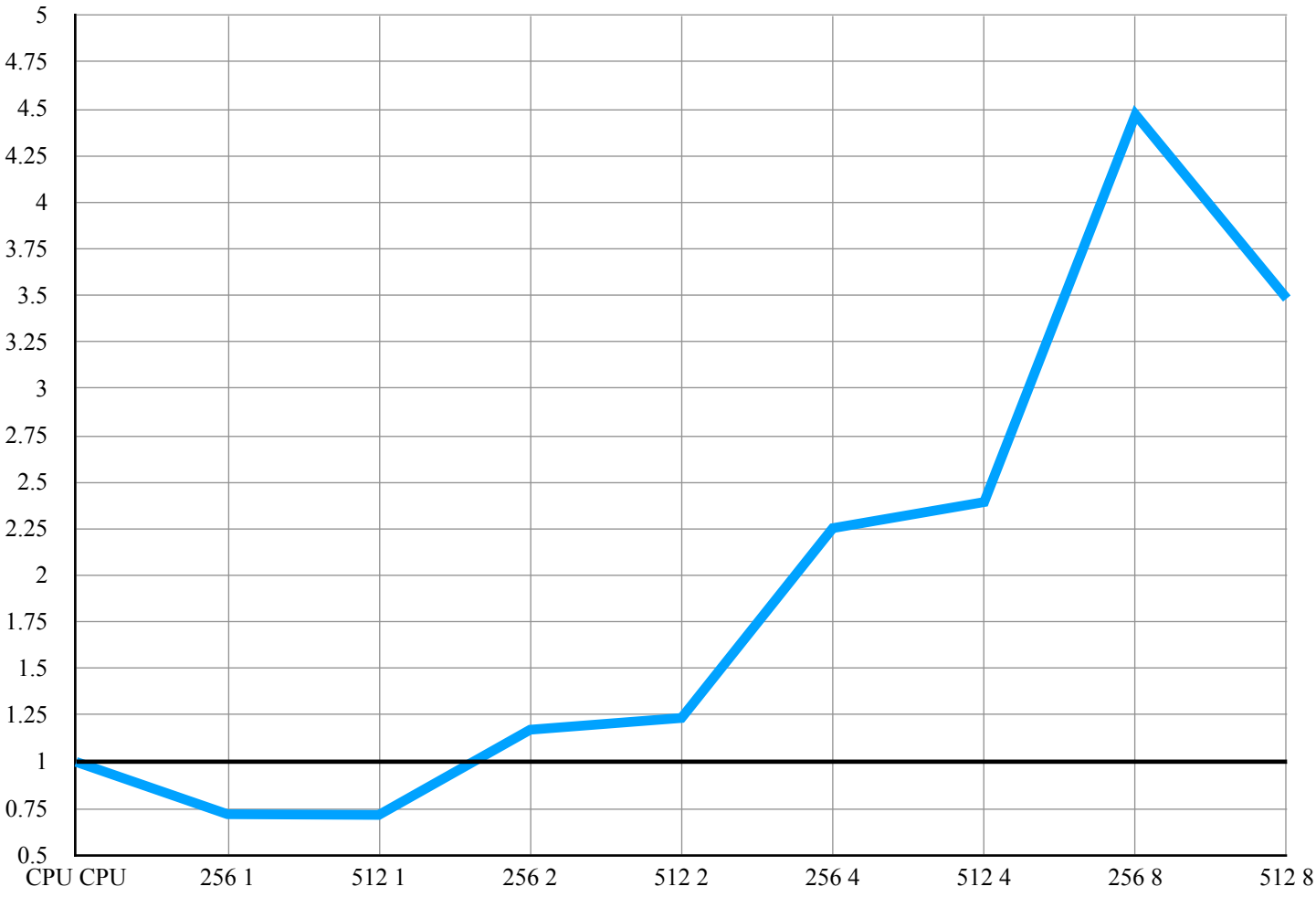

Рис. 1. Коэффициент ускорения GPU к CPU при размере 16384

Таблица 5.

Коэффициент ускорения к CPU (65536)

Размер блока	Количество блоков	Время (мс)	Ускорение
CPU	CPU	1.8836	1
256	1	2.68902	0.700478241143614
512	1	1.86557	1.00966460652776
256	2	0.961728	1.95855792906102
512	2	0.955424	1.97148072478816
256	4	0.502496	3.74848755014965
512	4	0.49152	3.83219401041667
256	8	0.251264	7.49649770759042
512	8	0.259328	7.26338845014808

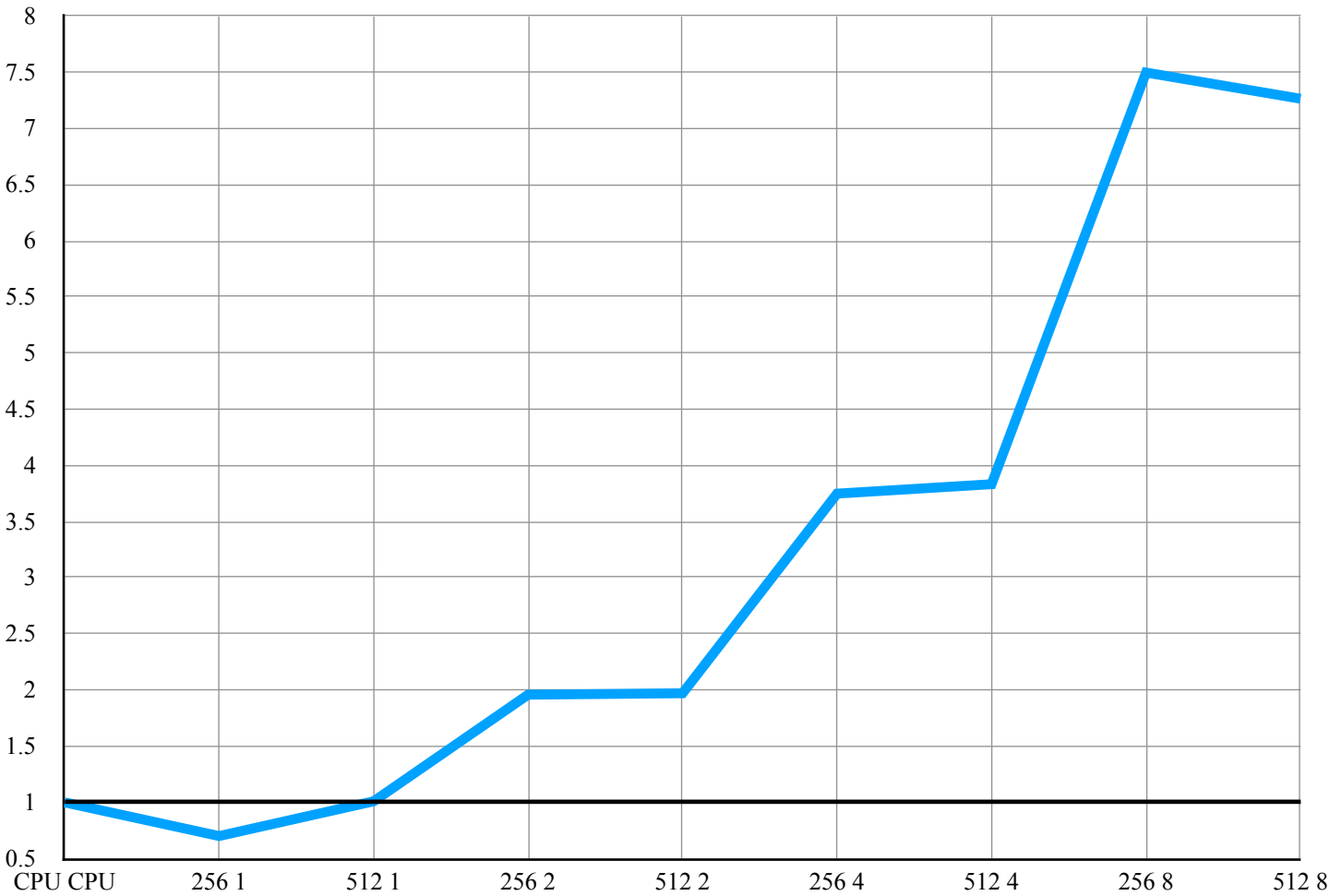

Рис. 2. Коэффициент ускорения GPU к CPU при размере 65536

Таблица 6.

Коэффициент ускорения к DEVICE (16384)

Размер блока	Количество блоков	Время (мс)	Ускорение
1	1	0.4851	1
256	1	0.673888	0.719852557101477
512	1	0.6776	0.7159
256	2	0.413696	1.17260017017327
512	2	0.39264	1.23548288508557
256	4	0.215328	2.2528421756576
512	4	0.202752	2.392578125
256	8	0.108544	4.46915536556604
512	8	0.139264	3.48331227022059

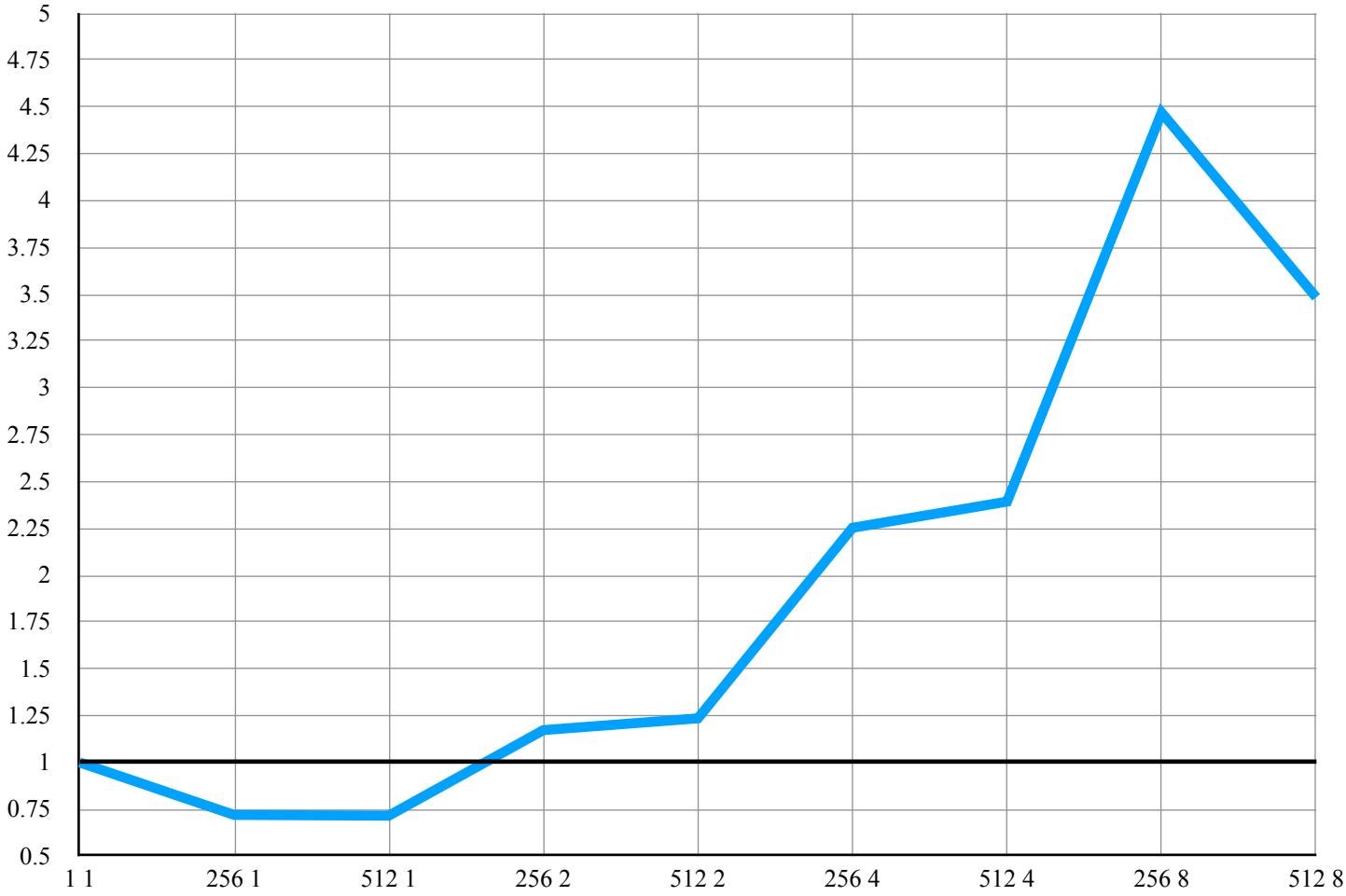

Рис. 3. Коэффициент ускорения GPU к GPU(1,1) при размере 16384

Таблица 7.

Коэффициент ускорения к DEVICE (65536)

Размер блока	Количество блоков	Время (мс)	Ускорение
1	1	1.8836	1
256	1	2.68902	0.700478241143614
512	1	1.86557	1.00966460652776
256	2	0.961728	1.95855792906102
512	2	0.955424	1.97148072478816
256	4	0.502496	3.74848755014965
512	4	0.49152	3.83219401041667
256	8	0.251264	7.49649770759042
512	8	0.259328	7.26338845014808

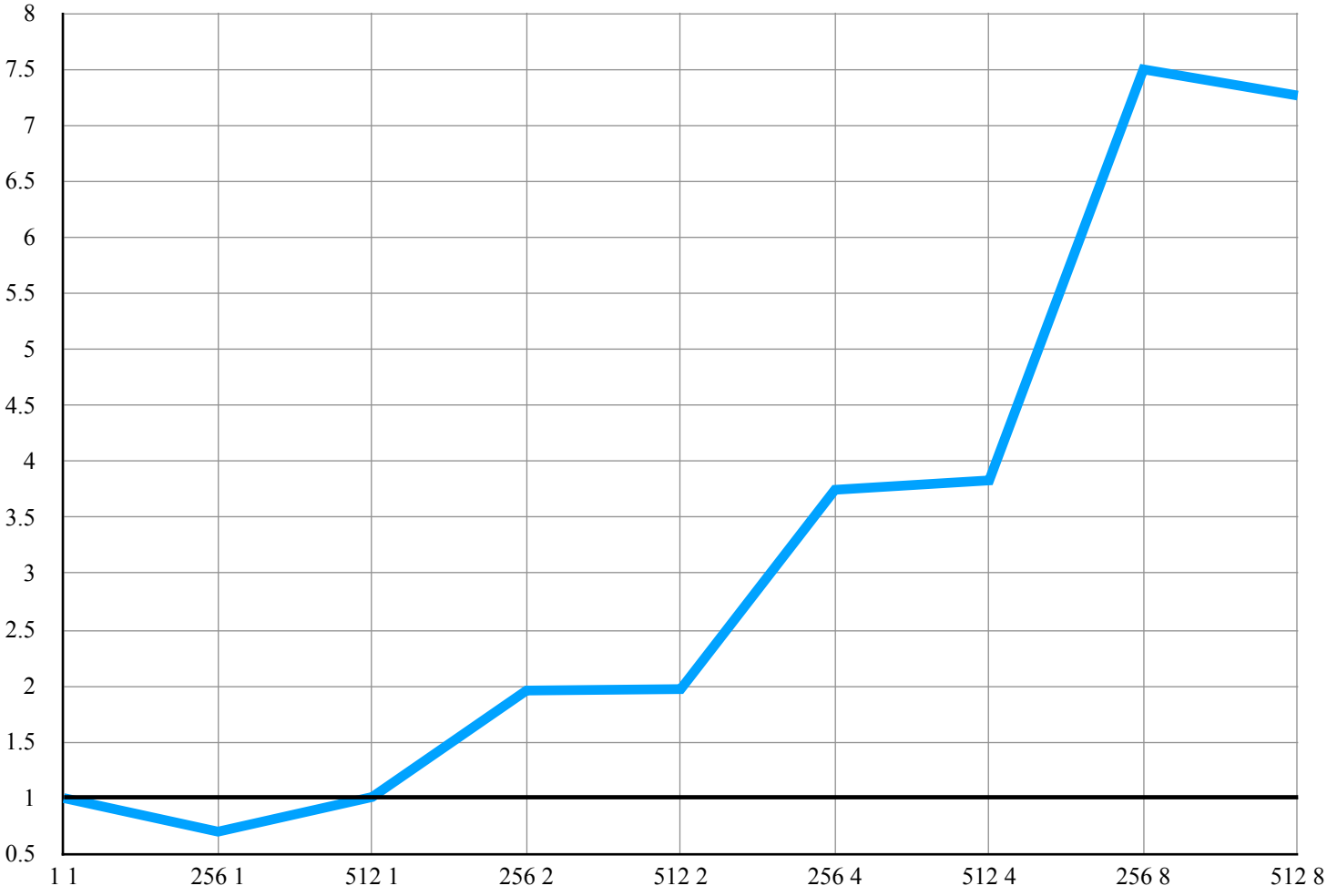

Рис. 4. Коэффициент ускорения GPU к GPU(1,1) при размере 65536

Приложение А. Код программы на CUDA

main.cu

```
#include <iostream>
#include <vector>
#include <fstream>
#include <chrono>
#include <cuda_runtime.h>
using namespace std;
using namespace std::chrono;
const int m = 3;                          // Значение степени по умолчанию

// Макрос и процедура для проверки выполнения CUDA кода
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char* file, int line, bool
abort = true)
{ if (code != cudaSuccess)
    { fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code),
file, line); if (abort) exit(code); } }

// Функция для расчёта времени выполнения функции
template<typename Func>
double measureExecutionTime(Func func) {
    auto startTime = high_resolution_clock::now();
    func();
    auto endTime = high_resolution_clock::now();
    auto duration = duration_cast<nanoseconds>(endTime -
startTime).count();
    return duration / 1e6;                // Перевод в миллисекунды
}

// Функция для создания вектора
double* createVector(int size) {
    auto* array = new double[size];// Динамическая инициализация матрицы
    for (int i = 0; i < size; i++) {
        array[i] = i + 1.5;
    }
    return array;
}
// Процедура для удаления вектора
void deleteVector(const double* array) {
    delete[] array;
}

// Процедура расчёта для CUDA
__global__ void kernel(const double* A, double* B, const double* C, const
double* D, double* Y, int size)
{
    // Условная координата в выходном векторе для текущего потока
    unsigned int idx_thread = blockIdx.x * blockDim.x + threadIdx.x;
    // Выполнение задачи
    for (int y = 0; y < size; y++)
    {
        Y[idx_thread * size + y] = (A[idx_thread * size + y] +
C[idx_thread * size + y]) / (pow(B[idx_thread * size + y], m) -
D[idx_thread * size + y]);
    }
}
```

```
// Процедура инициализации, запуска и обработки процедуры для CUDA
void runCuda(const double* A, const double* B, const double* C, const
double* D, double* Y, int size, int GRID_SIZE, int BLOCK_SIZE)
{
// Переменные для расчёта времени на CUDA
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

// Указатели для выделения памяти в видеокарте
    double *d_A, *d_B, *d_C, *d_D, *d_Y;
// Размер кусков вектора
    const int forSize = size / (GRID_SIZE * BLOCK_SIZE);

// Выделение памяти в видеокарте
    gpuErrchk(cudaMalloc<double>(&d_A, size));
    gpuErrchk(cudaMalloc<double>(&d_B, size));
    gpuErrchk(cudaMalloc<double>(&d_C, size));
    gpuErrchk(cudaMalloc<double>(&d_D, size));
    gpuErrchk(cudaMalloc<double>(&d_Y, size));
// Копирование векторов в память видеокарты
    gpuErrchk(cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice));
    gpuErrchk(cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice));
    gpuErrchk(cudaMemcpy(d_C, C, size, cudaMemcpyHostToDevice));
    gpuErrchk(cudaMemcpy(d_D, D, size, cudaMemcpyHostToDevice));
    gpuErrchk(cudaMemcpy(d_Y, Y, size, cudaMemcpyHostToDevice));

// Определение размеров блоков и сетки
    const dim3 block(BLOCK_SIZE);  // Размер блока (одномерный)
    const dim3 grid(GRID_SIZE);    // Размер сетки из блоков (одномерная)
// Запуск таймера
    cudaEventRecord(start);
// Запуск процедуры
    kernel <<< grid, block >>>(d_A, d_B, d_C, d_D, d_Y, forSize);
// Проверка на ошибки
   gpuErrchk(cudaPeekAtLastError());
// Остановка таймера
cudaEventRecord(stop);

// Копирование из памяти видеокарты в память процессора
    cudaMemcpy(Y, d_Y, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    cudaFree(d_D);
    cudaFree(d_Y);

    cudaEventSynchronize(stop);    // Синхронизация таймера
    float time;                    // Время
    cudaEventElapsedTime(&time, start, stop);     // Расчёт времени
    ofstream gpuTime("../gpu.csv", std::ios::app);// Открытие файла
    gpuTime << size << "," << BLOCK_SIZE << "," << GRID_SIZE << "," <<
time << "\n";
    gpuTime.close();
}
// Процедура расчёта для CPU
```

```
void runCPU(const double* A, const double* B, const double* C, const
double* D, double* Y, int size)
{
    for (int i = 0; i < size; i++) {
        Y[i] = (A[i] + C[i]) / (pow(B[i], m) - D[i]);
    }
}

int main() {
    double* A, *B, *C, *D, *Y;           // Создание указателей векторов
    vector<int> sizes = {16384, 65536}; // Список размеров векторов
    // Размеры блоков и их количество
    vector<pair<int,int>> blocks = {
{1, 1},{1, 4}, {1, 32}, {1, 64}, {1, 256}, {1, 512},
{2, 1}, {2, 4}, {2, 32}, {2, 64}, {2, 256}, {2, 512},
{4, 1}, {4, 4}, {4, 32}, {4, 64}, {4, 256}, {4, 512},
{8, 1}, {8, 4}, {8, 32}, {8, 64}, {8, 256}, {8, 512},
{16, 1}, {16, 4}, {16, 32}, {16, 64}, {16, 256}, {16, 512}};
// Подготовка файлов для результатов
    ofstream gpuTime("../gpu.csv", std::ios::app);
    ofstream cpuTime("../cpu.csv", std::ios::app);
    gpuTime << "Размер массивов, размер блока, количество блоков, время
(мс)" << "\n";
    cpuTime << "Размер массивов, время (мс)" << "\n";
// Основная программа
    for (const auto &size : sizes) {
// Создание векторов
        A = createVector(size);
        B = createVector(size);
        C = createVector(size);
        D = createVector(size);
        Y = new double[size];
// Запуск на CUDA
        for (const auto &block : blocks) {
            runCuda(A, B, C, D, Y, size, block.first, block.second);
        }
// Запуск на CPU
        double time = measureExecutionTime([&]() {runCPU(A, B, C, D, Y,
size);});
        cpuTime << size << "," << time << "\n";

// Очистка памяти
        deleteVector(A);
        deleteVector(B);
        deleteVector(C);
        deleteVector(D);
        deleteVector(Y);
    }
// Закрытие файлов
    gpuTime.close();
    cpuTime.close();

    return 0;
}
```