1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**ЛАБОРАТОРНАЯ РАБОТА № 3**

1. «**Программирование на GPU**»
2. по дисциплине «Языки программирования»
3. Выполнил
4. студент гр. №4851003/00002 Скрипко И.А.

<*подпись*>

1. Преподаватель
2. Малышев Е.В.

<*подпись*>

1. Санкт-Петербург
2. 2021
3. **Цель работы**

Научиться выполнять расчеты при помощи GPU и сравнить зависимость времени от входных данных при счете на CPU и GPU. Для этого реализовать решение задачи из линейной алгебры на CPU и GPU.

1. **Постановка задачи**
2. Реализовать нахождение обратной матрицы при помощи CPU;
3. Реализовать нахождение обратной матрицы при помощи GPU.
4. **Ход работы**

Была написана реализация программы для нахождения обратной матрицы с использованием CPU. Листинги программы представлен в приложениях А и Б. Также было реализовано решение этой же задачи, но с использование счета на GPU. Листинг данного решения представления в приложении В.

* 1. **Алгоритм работы программы на CPU**

В программе пользователь изначально вводит размер матрицы (обратная матрица существует только для квадратной матрицы, поэтому пользователь вводит только 1 число). Для хранения матрицы были созданы двумерные векторы. Далее матрицы заполняется случайными числами с помощью метода push\_back(), после чего пользователю показывается для какой матрицы будет находиться обратная.

Обратная матрицы будет находиться методом Гаусса-Жордана. Данный метод состоит из нескольких этапов:

1. Составить расширенную матрицу, где слева будет введенная, а справа единичная;
2. Привести левую матрицу расширенной матрицу к виду верхней треугольной матрицы;
3. Привести левую матрицу расширенной к виду диагональной матрицы;
4. Поделить строки на ведущие элементы левой диагональной матрицы, чтобы получилась единичная матрицы.

После данных действий правая матрицы расширенной матрицы и есть искомая обратная матрица.

Расширенная матрица не составлялась специально. Функции каждый раз принимали на вход 2 матрицы (левую и правую).

Для приведения матрицы в вид верхней треугольной был написан следующий код:

for (int i = 0; i < n - 1; i++)

{

// Обмен строк

pivotize(matrix, identity, n, i);

// Обнулить элементы ниже ведущего

subtract\_below(matrix, identity, n, i);

}

Функция pivotize принимает на вход расширенную матрицу, размер матрицы и индекс элемента главной диагонали, под которым надо обнулить матрицу. Данная функция ищет в i-ом столбце самое большое число (рассматриваются числа, начиная с i-ой строки). Далее строки меняются местами с помощью функции std::swap().

Функция substract\_below принимает на вход расширенную матрицу, размер матрицы и индекс элемента главной диагонали, под которым нужно обнулить элементы. В данной функции элемент под элементом (i,i) делится на элемент (i,i), тем самым находится коэффициент, на который нужно умножить i-ую строку, чтобы при добавлении к другой строке элемент под (i,i), который сейчас рассматривается, обнулился.

Соответственно, пройдя этот цикл, получится верхняя треугольная матрица.

Ниже представлен код для приведения матрицы в диагональный вид:

for (int i = n - 1; i > 0; i--)

// Обнулить элементы выше ведущего

subtract\_above(matrix, identity, n, i);

Функция substract\_above принимает на вход расширенную матрицу, размер матрицы и индекс элемента главной диагонали, над которым надо обнулить матрицу. Данная функция работает с право налево, так как последняя строка содержит только 1 число – справа. Остальная равны 0 и не будет менять числа выше при добавлении нижней строки к высшим. В данной функции находится коэффициент для каждой строки и строка (i,i) домножается на него и добавляется к другим строкам. В результате работы данного фрагмента кода матрица примет вид диагональной.

Далее, для приведения левой матрицы расширенной матрицы к виду единичной матрицы, была написан следующий код:

// Разделить строки на ведущие элементы

for (int i = 0; i < n; i++)

{

devide(identity, n, i, matrix[i][i]);

devide(matrix, n, i, matrix[i][i]);

}

Функция devide принимает матрицу, размерность, номер строки и коэффициент, на сколько надо делить. Далее делит строку матрицы на этот коэффициент. В результате работы этого цикла, левая матрица расширенной матрицы примет вид единичной. Матрицы identity будет хранить результат работы программы: найденную обратную матрицу.

Сложность алгоритма является квадратичной – . В начале идет работа с заполнением матрицы, далее в цикле прохода строк матрицы вызываются функции, которые работают со строками матрицы. Матрица является квадратичной, поэтому работа со строками и столбцами будет проходить одинаковое количество элементов.

Данный алгоритм был протестирован на разных входных данных с замером времени. Результаты представлены в таблице 1.

Таблица 1 – Зависимость времени работы программы от размера входных данных.

|  |  |
| --- | --- |
| Входные данные: | Время работы программы: |
| 50 | 0.017 |
| 100 | 0.128 |
| 150 | 0.429 |
| 200 | 1.018 |
| 250 | 2.003 |
| 300 | 3.5 |
| 350 | 6.024 |
| 400 | 9.538 |
| 500 | 17.106 |

* 1. **Алгоритм работы программы на GPU**

На GPU данная задача будет решаться с использованием одномерного массива. Пользователю предлагается ввести размер матрицы, далее выделяется память, и матрица заполняется случайными числами. Далее заполняется единичная матрица.

Далее выделяется память на GPU и туда копируются матрицы с CPU. Затем матрица приводится к виду верхней треугольной матрицы тем же методом, что и в программе на CPU.

for (int i = 0; i < matrix\_size; i++)

{

// Обменять строку со строкой с наибольшим ведущим элементом и обнулить

// элементы под ним

//ищем максимальный элемент столбца (запускается 1 блок с 1 потоком внути)

max\_in\_column << <1, 1 >> > (dev\_matrix, matrix\_size, i);

//меняем i-ую строку со строкой с максимальным элементом в i-ом столбце местами (запускается 1 блок с matrix\_size потоками внутри)

swap\_lines << <1, matrix\_size >> > (dev\_matrix, dev\_identity, matrix\_size, i);

//обнуляем элементы под i-ым в i-ом столбце

make\_zero\_below << < 1, matrix\_size >> > (dev\_matrix, dev\_identity, matrix\_size, i);

}

Ядро max\_in\_column принимает расширенную матрицу и индекс столбца. Далее находит в этом столбце максимальный элемент и сохраняет в переменную max\_element\_index (переменная помечена как device – может вызываться только на GPU). Данный процесс можно было бы распараллелить, но могла возникнуть проблема синхронизации, так как все потоки бы имели доступ к переменной max\_element\_index. Данную проблему можно было бы решить применением атомарных операций при присваивании данной переменной нового значения.

Ядро swap\_lines меняет местами 2 строки. Данный процесс можно распараллелить, что и сделано в коде. У 2 строк матрицы одновременно меняются все элементы местами.

Ядро make\_zero\_below с помощью элементарных преобразований над матрицами обнуляет все элементы под элементом (i,i). Ко всем строкам, ниже i-ой, будет прибавляться i-ая строка, домноженная на коэффициент. Сама i-ая строка меняться не будет, поэтому данный процесс можно распараллелить. Ядро запускает количество потоков, равное количеству строк, после чего каждая строка параллельно меняется.

После приведения матрицы к виду верхней треугольной матрицы в обнуленных элементах могло что-то остаться, в виде десятитысячных долей, за счет округления при счете (на числа выделено определенное количество бит). Поэтому выполняется функция, которая даст всем этим элементам значение точного нуля:

//в элементах могли остаться какие-нибудь тысячные доли, поэтому обнулим их

nullify\_below << <1,matrix\_size >> > (dev\_matrix, matrix\_size);

Далее матрица приводится к виду диагональной:

for (int i = matrix\_size - 1; i >= 0; i--)

// Обнулить элементы выше ведущего

subtract\_above << <1, matrix\_size >> > (dev\_matrix, dev\_identity, matrix\_size, i);

i-ая строка домножается на коэффициент и прибавляется к строкам выше, обнуляя элементы, выше (i,i). Данный процесс также распараллеливается, так как i-ая строка не меняется.

После этого элементы также принимают значения абсолютного нуля:

make\_zero\_above << <1, matrix\_size >> > (dev\_matrix, matrix\_size);

Далее расширенная матрица делится на коэффициенты главной диагонали:

devide\_identity << <1, matrix\_size >> > (dev\_matrix, dev\_identity, matrix\_size);

После этого, результат копируется с GPU на CPU и происходит вывод результата. Далее освобождается задействованная память на GPU и CPU.

Разработанная программа была протестирована на различных входных данных. Результаты тестирования представлены в таблице 2.

Таблица 2 – Зависимость времени работы программы от входных данных

|  |  |  |
| --- | --- | --- |
| Входные данные: | Время работы программы: | Коэффициент ускорения |
| 50 | 0.153 | 0.11 |
| 100 | 0.162 | 0.79 |
| 150 | 0.391 | 1.15 |
| 200 | 0.749 | 1.36 |
| 250 | 0.754 | 2.65 |
| 300 | 0.773 | 4.52 |
| 350 | 0.787 | 7.65 |
| 400 | 0.82 | 11.63 |
| 500 | 0.873 | 19.59 |

На основании данных из таблиц 1 и 2 был построен график (рисунок 1).

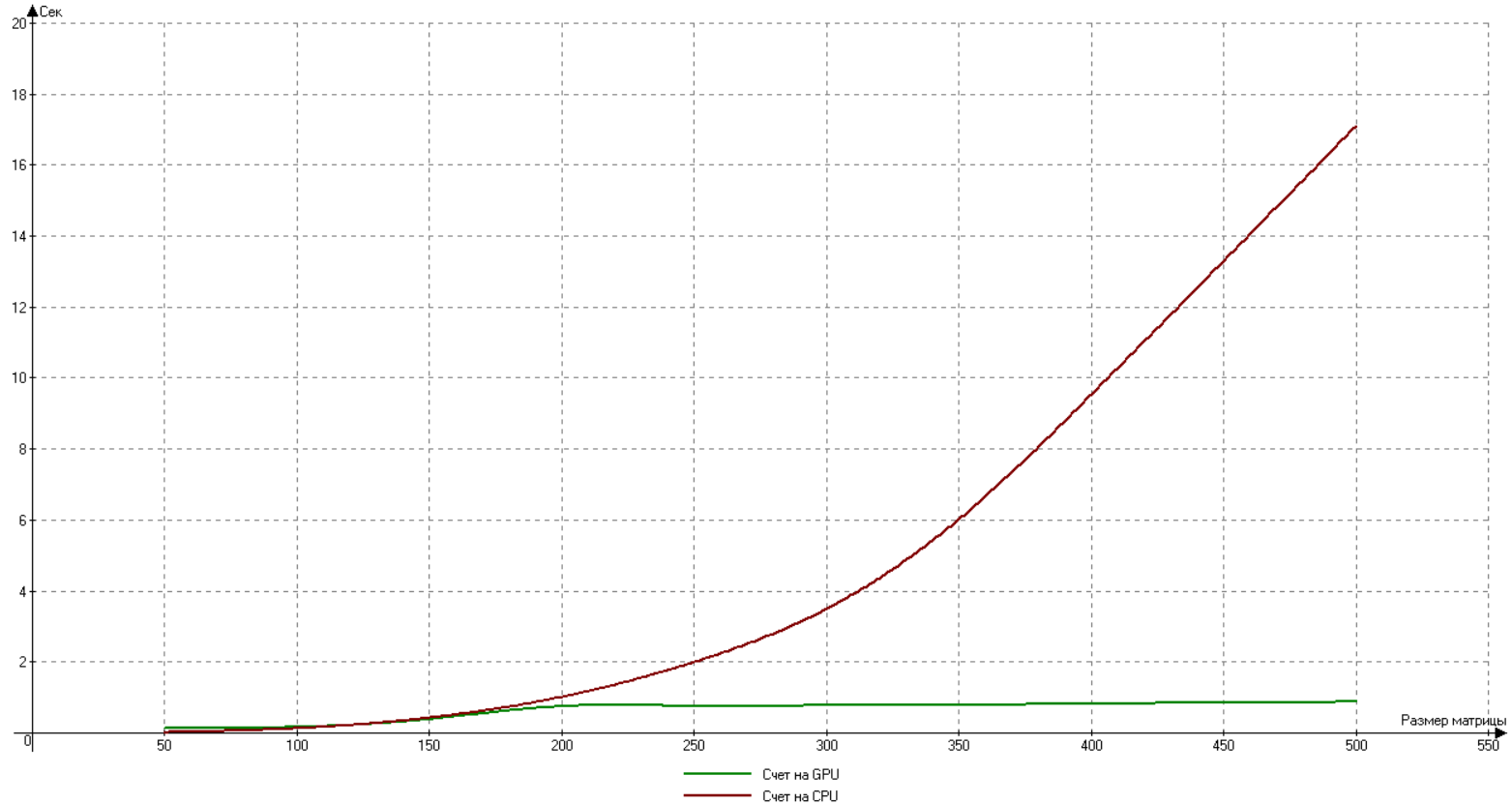


Рисунок 1 – Зависимость времени работы программы от входных данных

По графику видно, что на маленьких входных значениях счет на GPU проигрывает по времени счету на CPU. Это объясняется дополнительными операции копирования из памяти хоста на память девайса перед счетом и обратно после счета. Коэффициент выигрыша по времени начинает расти, начиная с входных данных равных 150. Далее время работы счета на GPU является почти константным – независимо от размера матрицы, элементарные преобразования над каждым элементом строки будет параллельным, следовательно, сложность счета на GPU будет константной.

1. **Результаты работы программы**

Результаты работы программы представлены на рисунках 2-4. Интерфейсы и результаты работы с счетом на CPU и GPU одинаковые – используется один и тот же алгоритм.

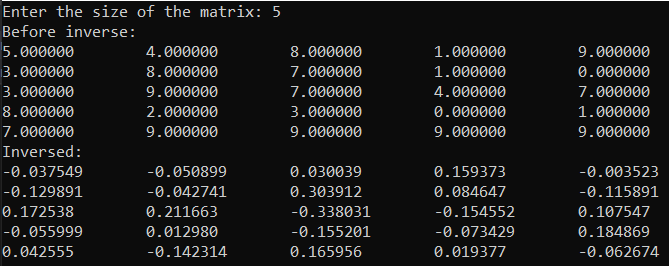


Рисунок 2 – Результат работы программы

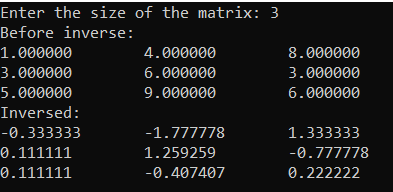


Рисунок 3 – Результат работы программы

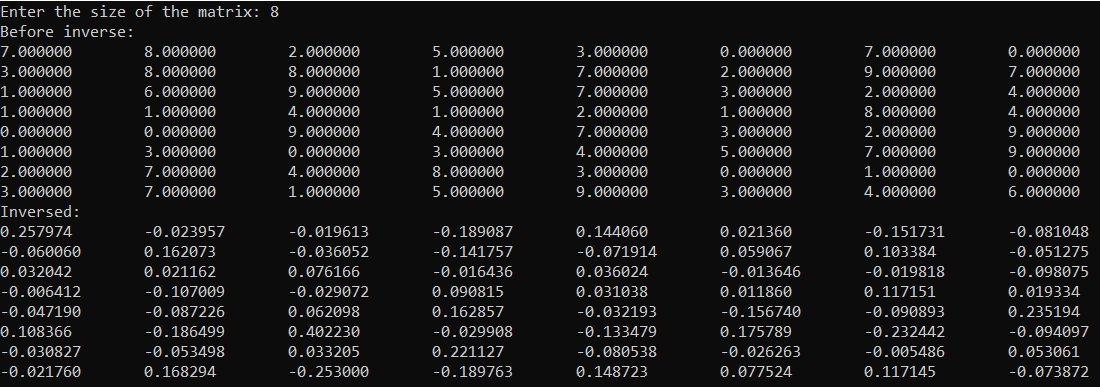


Рисунок 4 – Результат работы программы

1. **Выводы**

В результате выполнения данной лабораторной работы были получены навыки выполнения расчетов на GPU – написание ядер и проведение расчетов на них. Также были сравнены зависимости времени работы программы на CPU и GPU, результаты показали, что чем больше входные данные, тем больше коэффициент ускорения

**ПРИЛОЖЕНИЕ А**

Листинг файла «main.cpp»

#include "main.h"

int main()

{

//ввод размера матрицы

cout << "Enter the size of the matrix: ";

int matrix\_size;

cin >> matrix\_size;

vector<vector<double>> matrix;

vector<vector<double>> identity;

matrix.resize(matrix\_size);

identity.resize(matrix\_size);

//заполняем матрицы

srand(time(NULL));

for (int i = 0; i < matrix\_size; i++)

for (int j = 0; j < matrix\_size; j++)

{

identity[i].push\_back(i == j ? 1.0 : 0.0);

matrix[i].push\_back(rand() % 10);

}

cout << "Before inverse:\n";

print(matrix, matrix\_size);

//нахождение обратной матрицы

inverse(matrix, identity, matrix\_size);

cout << "Inversed:\n";

print(identity, matrix\_size);

return 0;

}

void print(vector<vector<double>> &matrix, int n)

{

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

cout <<fixed<< matrix[i][j] << "\t";

cout << endl;

}

}

// Поставить на место row строку с наибольшим ведущим коэффициентом

void pivotize(vector<vector<double>>& matrix, vector<vector<double>>& identity, int n, int row)

{

int max\_index = row;

double max\_value = fabs(matrix[row][row]);

double current\_value;

for (int i = row + 1; i < n; i++)

{

current\_value = fabs(matrix[i][row]);

if (current\_value > max\_value)

{

max\_index = i;

max\_value = current\_value;

}

}

//меняем строки местами

if (row != max\_index)

{

swap(matrix[row], matrix[max\_index]);

swap(identity[row], identity[max\_index]);

}

}

// Разделить строку матрицы на значение

void devide(vector<vector<double>>& matrix, int n, int i, double denominator)

{

for (int j = 0; j < n; j++)

matrix[i][j] /= denominator;

}

// Обнуление элементов под ведущим элементом в строке x

void subtract\_below(vector<vector<double>>& matrix, vector<vector<double>>& identity, int n, int x)

{

double coeff;

for (int i = x + 1; i < n; i++)

{

coeff = matrix[i][x] / matrix[x][x];

for (int j = x; j < n; j++)

matrix[i][j] -= coeff \* matrix[x][j];

for (int j = 0; j < n; j++)

identity[i][j] -= coeff \* identity[x][j];

}

}

// Обнуление элементов над ведущим элементом в строке x

void subtract\_above(vector<vector<double>>& matrix, vector<vector<double>>& identity, int n, int x)

{

double coeff;

for (int i = x - 1; i >= 0; i--)

{

//обнуляем элементы выде главной диагонали матрицы для левой части расширенной матрицы

coeff = matrix[i][x] / matrix[x][x];

for (int j = x; j >= 0; j--)

matrix[i][j] -= coeff \* matrix[x][j];

//делаем тоже самое для правой матрицы расширенной матрицы

for (int j = 0; j < n; j++)

identity[i][j] -= coeff \* identity[x][j];

}

}

// Обращение матрицы методом Гаусса-Жордана

void inverse(vector<vector<double>>& matrix, vector<vector<double>>& identity, int n)

{

//с помощью элементарных преобразований над матрицами получим верхнюю треугольную матрицу

for (int i = 0; i < n - 1; i++)

{

// Обмен строк

pivotize(matrix, identity, n, i);

// Обнулить элементы ниже ведущего

subtract\_below(matrix, identity, n, i);

}

//с помощью элементарных преобразований получим диагональную матрицу

for (int i = n - 1; i > 0; i--)

// Обнулить элементы выше ведущего

subtract\_above(matrix, identity, n, i);

// Разделить строки на ведущие элементы

for (int i = 0; i < n; i++)

{

devide(identity, n, i, matrix[i][i]);

devide(matrix, n, i, matrix[i][i]);

}

}

**ПРИЛОЖЕНИЕ Б**

Листинг файла «main.h»

#ifndef MAIN\_H

#define MAIN\_H

#include <iostream>

#include <iomanip> //std::fixed

#include <cmath> //std::fabs

#include <time.h>

#include <algorithm> // std::swap

#include <vector> // std::vector

using namespace std;

void print(vector<vector<double>> &, int );

void pivotize(vector<vector<double>> &, vector<vector<double>> &, int , int );

void devide(vector<vector<double>> &, int , int , double );

void subtract\_below(vector<vector<double>> &, vector<vector<double>> &, int , int );

void subtract\_above(vector<vector<double>> &, vector<vector<double>> &, int , int );

void inverse(vector<vector<double>> &, vector<vector<double>> &, int );

#endif

**ПРИЛОЖЕНИЕ В**

Листинг программы «kernel.cu»

#include "device\_launch\_parameters.h"

#include <curand\_kernel.h>//выделение памяти

#include <cuda.h>

#include <iostream>

#include <cassert>

#include <cmath>

#include <time.h>

using namespace std;

//\_\_global\_\_ - вызывается CPU, но выполняется GPU\

\_\_device\_\_ - выполняется на GPU\

\_\_host\_\_ - выполняется на CPU

//переменная, в которой будет храниться результат функции max\_in\_column()\

может быть вызвана только из GPU

\_\_device\_\_ int max\_element\_index;

//поиск максимального элемента в столбце, исполняется на GPU

\_\_global\_\_ void max\_in\_column(double\* matrix, int matrix\_size, int x)

{

int max\_index = x;

double max\_value = fabs(matrix[x \* matrix\_size + x]);

double current\_value;

for (int i = x + 1; i < matrix\_size; i++)

{

//спускается на строку каждый раз

current\_value = fabs(matrix[i \* matrix\_size + x]);//fabs - взятие модуля

if (current\_value > max\_value)

{

max\_index = i;

max\_value = current\_value;

}

}

max\_element\_index = max\_index;//сохраняем максимальный индекс столбца

}

\_\_global\_\_ void swap\_lines(double\* matrix, double\* identity, int matrix\_size, int row) {

//если выбранная строка и так на своем месте, то ничего менять не будем

if (row == max\_element\_index)

return;

//размер блока(количество потоков в блоке)\*номер блока+номер потока

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

if (idx >= matrix\_size) return;//проверяем, что не выходим за пределы массива

double temp;

temp = matrix[row \* matrix\_size + idx];//сохраняем элемент строки

//ставим на место этого элемента соответствующий ему элемент другой строки

matrix[row \* matrix\_size + idx] = matrix[max\_element\_index \* matrix\_size + idx];

//ставим сохраненный элемент на соответствующее ему место в другой строке

matrix[max\_element\_index \* matrix\_size + idx] = temp;

//делаем аналогичную операцию, но с правой матрицей

temp = identity[row \* matrix\_size + idx];

identity[row \* matrix\_size + idx] = identity[max\_element\_index \* matrix\_size + idx];

identity[max\_element\_index \* matrix\_size + idx] = temp;

}

\_\_global\_\_ void make\_zero\_below(double\* matrix, double\* identity, int matrix\_size, int x)

{

//узнаем номер потока

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

int i, j;

//если это строка, под которой надо вычесть, то ничего не вычитаем, саму из себя строку вычитать нельзя

if (idx \* matrix\_size <= x \* matrix\_size) return;

//находим коэффициент

double coeff = matrix[idx \* matrix\_size + x] / matrix[x \* matrix\_size + x];

//проходим строку x с элемента matrix[x][x] до конца строки и из каждого элемента вычитаем\

соответствующий элемент строки x, умноженный на coeff

for (i = x; i < matrix\_size; i++)

matrix[idx \* matrix\_size + i] -= coeff \* matrix[x \* matrix\_size + i];

//делаем тоже самое для правой части расширенной матрицы

for (i=0; i < matrix\_size; i++)

identity[idx \* matrix\_size + i] -= coeff \* identity[x \* matrix\_size + i];

//cout << "haha\n";

}

\_\_global\_\_ void nullify\_below(double\* matrix, int matrix\_size)

{

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

if (idx == 0) return;

int cnt = 0;

for (int i = matrix\_size-idx; i < matrix\_size; i++)

matrix[matrix\_size\*idx+cnt++] = 0.0;

}

// Обнуление элементов над ведущим элементом в строке x

\_\_global\_\_ void subtract\_above(double\* matrix, double\* identity, int matrix\_size, int x)

{

//номер потока

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

if (idx == x) return;

double coeff = matrix[matrix\_size \* idx + x]/ matrix[matrix\_size \* x + x];

//в строке все элементы, кроме этого обнулены, так что в этой матрице ничего не изменится. \

Мы просто домножим на число и вычтем, получим 0

matrix [matrix\_size \* idx + x] = 0.0;

//прибавим строку к единичной матрице

for (int i = 0; i < matrix\_size; i++)

identity[matrix\_size \* idx + i] -= coeff \* identity[matrix\_size \* x + i];

}

\_\_global\_\_ void make\_zero\_above(double\* matrix, int matrix\_size)

{

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

if (idx == matrix\_size-1) return;

for (int i = idx+1; i <matrix\_size ; i++)

matrix[matrix\_size \* idx + i] = 0.0;

}

\_\_global\_\_ void devide\_identity(double\* matrix, double\* identity, int matrix\_size)

{

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

double coeff = matrix[idx \* matrix\_size + idx];

matrix[idx \* matrix\_size + idx] = 1.0;

for (int i = 0; i < matrix\_size; i ++)

identity[idx \* matrix\_size + i] /= coeff;

}

void print\_both(double\* matrix, double\* identity\_matrix, int matrix\_size);

void print(double\* matrix, int n);

\_\_host\_\_ int main()

{

//ввод размера матрицы

cout << "Enter the size of the matrix: ";

int matrix\_size;

cin >> matrix\_size;

int blocks = matrix\_size / 512;

if (blocks % 512 != 0) blocks++;

if (blocks > 384)

{

cout << "Error: size\n";

return 0;

}

//заполнение матрицы

//выделение памяти для данных на CPU

double\* matrix = new double[matrix\_size \* matrix\_size];

assert(matrix != 0);

double\* identity\_matrix = new double[matrix\_size \* matrix\_size];

assert(identity\_matrix != 0);

//заполняем таблицу случайными числами

srand(time(NULL));

for (int i = 0; i < matrix\_size \* matrix\_size; i++)

matrix[i] = 1 + rand() % 10;

//заполняем единичную матрицу

for (int i = 0; i < matrix\_size; i++)

for (int j = 0; j < matrix\_size; j++)

identity\_matrix[i \* matrix\_size + j] = (i == j ? 1.0 : 0.0);

cout << "Before:" << endl;

print\_both(matrix, identity\_matrix, matrix\_size);

double recording = clock();

//Выделение памяти для данных на GPU

double\* dev\_matrix;

double\* dev\_identity;

cudaMalloc((void\*\*)&dev\_matrix, sizeof(double) \* matrix\_size \* matrix\_size);

cudaMalloc((void\*\*)&dev\_identity, sizeof(double) \* matrix\_size \* matrix\_size);

//копируем с host на device

cudaMemcpy(dev\_matrix, matrix, sizeof(double) \* matrix\_size \* matrix\_size, cudaMemcpyHostToDevice);

cudaMemcpy(dev\_identity, identity\_matrix, sizeof(double) \* matrix\_size \* matrix\_size, cudaMemcpyHostToDevice);

//инициализируем переменные для замера времени счета на GPU

//на данном этапе все готово и можно приступать к преобразованиям\

с помощью элементарных преобразований обнулим матрицу под главной диагональю

//int i = 0;

for (int i = 0; i < matrix\_size; i++)

{

// Обменять строку со строкой с наибольшим ведущим элементом и обнулить

// элементы под ним

//ищем максимальный элемент столбца

max\_in\_column << <1, 1 >> > (dev\_matrix, matrix\_size, i);

//меняем i-ую строку со строкой с максимальным элементом в i-ом столбце местами

swap\_lines << <blocks, 512 >> > (dev\_matrix, dev\_identity, matrix\_size, i);

//обнуляем элементы под i-ым в i-ом столбце

make\_zero\_below << < blocks, 512 >> > (dev\_matrix, dev\_identity, matrix\_size, i);

}

//в элементах могли остаться какие-нибудь тысячные доли, поэтому обнулим их

nullify\_below << <blocks, 512 >> > (dev\_matrix, matrix\_size);

for (int i = matrix\_size - 1; i >= 0; i--)

// Обнулить элементы выше ведущего

subtract\_above << <blocks, 512 >> > (dev\_matrix, dev\_identity, matrix\_size, i);

make\_zero\_above << <blocks, 512 >> > (dev\_matrix, matrix\_size);

devide\_identity << <blocks, 512 >> > (dev\_matrix, dev\_identity, matrix\_size);

cudaThreadSynchronize();//синхронизация потоков

//копируем результат из памяти GPU на CPU

cudaMemcpy(matrix, dev\_matrix, sizeof(double) \* matrix\_size \* matrix\_size, cudaMemcpyDeviceToHost);

cudaMemcpy(identity\_matrix, dev\_identity, sizeof(double) \* matrix\_size \* matrix\_size, cudaMemcpyDeviceToHost);

recording = (clock() - recording) / CLOCKS\_PER\_SEC;

//вывод обратной матрицы

cout << "Invers matrix:" << endl;

print(identity\_matrix, matrix\_size);

cout << "Time of work: " << fixed << recording << endl;

//освобождаем память device

cudaFree(dev\_matrix);

cudaFree(dev\_identity);

//освобождаем память host

delete[] matrix;

delete[] identity\_matrix;

return 0;

}

\_\_host\_\_ void print\_both(double\* matrix, double\* identity\_matrix, int matrix\_size)

{

for (int i = 0; i < matrix\_size; i++)

{

int flag = 1;

for (int j = 0; j < 2 \* matrix\_size; j++)

if (j < matrix\_size)

cout << fixed << matrix[i \* matrix\_size + j] << "\t";

else

{

if (flag)

{

cout << "|\t";

flag--;

}

cout << fixed << identity\_matrix[i \* matrix\_size + j - matrix\_size] << "\t";

}

cout << endl;

}

cout << endl;

}

\_\_host\_\_ void print(double\* matrix, int n)

{

cout << scientific;

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

cout << fixed << matrix[i \* n + j] << "\t";

cout << endl;

}

}