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

**ЛАБОРАТОРНАЯ РАБОТА № 3**

1. «**Расчеты при помощи GPU**»
2. по дисциплине «Языки программирования»
3. Выполнили
4. студенты гр. 5131001/20001 Маронова К.Д.  
    Тайманов Д.М.

<*подпись*>

1. Преподаватель

Малышев Е.В.

<*подпись*>

1. Санкт-Петербург
2. 2023
3. **1. Цель работы**

В лабораторной работе необходимо научиться выполнять расчеты при помощи GPU (CUDA, OpenCL).

**2. Ход работы**

Для выполнения лабораторной работы была выбрана версия CUDA 12.3.1. Необходимо установить CUDA с официального сайта и запустить проект в Visual Studio 2022.

**Алгоритм, который мы использовали для распараллеливания в своей работе**, основан на запуске ядра CUDA с использованием блоков и потоков. В функции main() запускается ядро с параметрами блоков и потоков, которые позволяют выполнять обработку данных *параллельно*:

\_\_global\_\_ void determinant(double\* mat)

{

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

// Приведение к треугольной матрице

int i = 0;

while (i < N - 1) {

// Нахождение коэффициента для обнуления нижней строки

if (tid > i) {

double coeff = mat[tid \* N + i] / mat[i \* N + i];

// Обнуление нижней строки

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

{

mat[tid \* N + j] -= coeff \* mat[i \* N + j];

}

}

i++;

}

}

Каждый блок содержит N потоков, где N - размер матрицы. Каждый поток выполняет вычисления *для одной строки матрицы*. Таким образом, все строки матрицы обрабатываются параллельно благодаря использованию множества потоков.

Каждый поток получает уникальный идентификатор tid, который вычисляется как сумма threadIdx.x (**идентификатор потока в блоке**) и blockDim.x \* blockIdx.x (**идентификатор блока**).

Внутри ядра CUDA происходит приведение матрицы к треугольному виду путем обнуления нижней части матрицы. Это также выполняется параллельно с использованием различных потоков для различных строк матрицы.

После завершения выполнения ядра CUDA, результат вычислений копируется обратно на хост и используется для вычисления детерминанта путем умножения значений на диагонали матрицы:

cudaMemcpy(host\_mat, dev\_mat, N\* N \* sizeof(double), cudaMemcpyDeviceToHost);

for (int i = 0; i < N; i++) {

host\_det \*= host\_mat[i \* N + i];

if (host\_det == 0) break;

}

**Краткое описание аналогичного алгоритма для CPU:**

Алгоритм начинается с инициализации матрицы случайными значениями. Затем запускается функция determinant, которая приводит матрицу к треугольному виду путем обнуления нижней части матрицы. Это выполняется с использованием циклов и вычисления коэффициентов для обнуления строк (Метод Гаусса). После этого вычисляется определитель матрицы и выводится на экран (см. Приложение).

**Для вычисления количества блоков и потоков используется следующая схема:**

Количество блоков определяется как (N+64)/64, что означает, что мы разбиваем матрицу на блоки по 64 потока каждый, и если размерность матрицы не делится на 64, то добавляем еще один блок для обработки оставшихся элементов.

Количество потоков в каждом блоке задано как N, что означает, что каждый блок будет обрабатывать N элементов матрицы (построчно).

Это позволяет эффективно распараллеливать вычисления на GPU и ускорить выполнение операций над матрицей.

Правильность работы алгоритма была проверена с помощью онлайн калькулятора, высчитывающего определитель матрицы (алгоритм на CPU и GPU работает верно). Результаты приведены ниже:

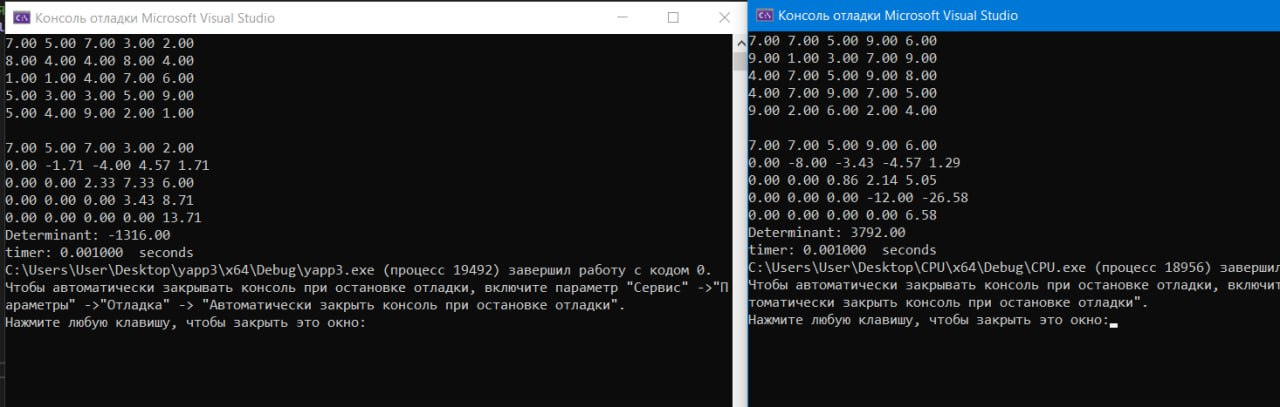


Рисунок 1. «Результат работы программы»

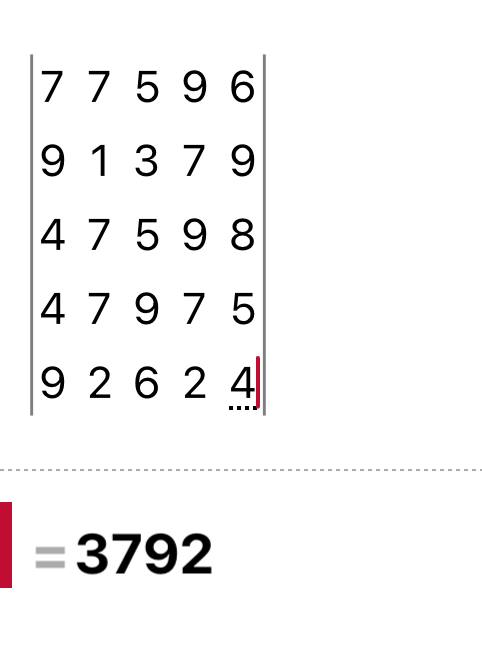
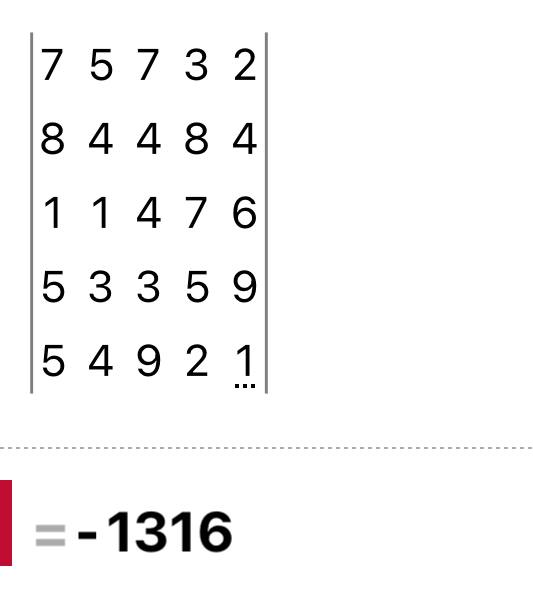


Рисунок 2. «Результат вычислений онлайн калькулятора»

**Зависимости времени работы программ на CPU и GPU в зависимости от размеров входных данных:**

|  |  |  |
| --- | --- | --- |
| Значение N | Время выполнения программы на GPU, мс | Время выполнения программы на CPU, мс |
| 10 | 1 | 0 |
| 100 | 32 | 1 |
| 500 | 115 | 138 |
| 1000 | 356 | 1382 |
| 2000 | 1530 | 8187 |
| 3000 | 3389 | 26599 |
| 4000 | 6480 | 64892 |
| 5000 | 10231 | 128000 |

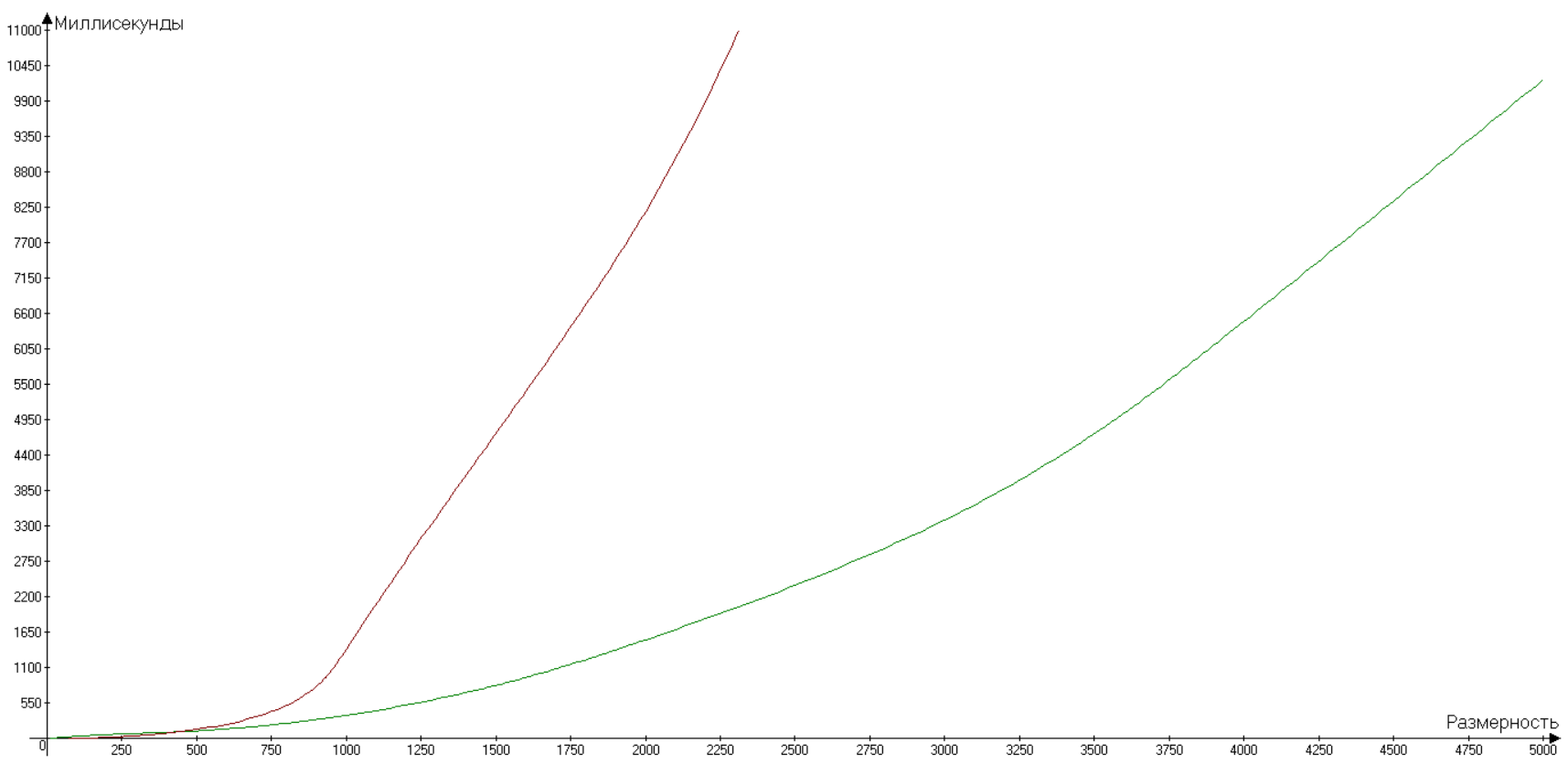
****

Рисунок 3. «График зависимости времени (мс) от размеров матрицы для CPU и GPU»

**4. Вывод**

Из результатов лабораторной работы видно, что время выполнения программы на GPU остается примерно постоянным при увеличении размера входных данных N, в то время как время выполнения программы на CPU существенно увеличивается. Это говорит о том, что параллельные вычисления на GPU позволяют эффективно обрабатывать большие объемы данных и значительно ускоряют выполнение операций. Таким образом, использование CUDA для параллельных вычислений на GPU может быть целесообразным при работе с большими объемами данных.

**Листинг:**

**GPU:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <time.h>

#include <locale.h>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

// Определяем размер матрицы

#define N 10

// Ядро CUDA для нахождения определителя

\_\_global\_\_ void determinant(double\* mat)

{

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

int i = 0;

if (tid < N) {

while (i < N - 1)

{

// Нахождение коэффициента для обнуления нижней строки

if (tid > i) {

double coeff = mat[tid \* N + i] / mat[i \* N + i];

// Обнуление нижней строки

//printf("BITCH1:%d\n", tid);

for (int j = i; j < N; j++)

{

//printf("BITCH2:%d\n", tid);

mat[tid \* N + j] -= coeff \* mat[i \* N + j];

}

}

i++;

}

}

}

int main()

{

double\* dev\_mat;

double host\_det = 1.0;

srand(time(NULL)); // инициализация функции rand значением функции time

// Выделение памяти на устройстве

cudaMalloc((void\*\*)&dev\_mat, N \* N \* sizeof(double));

double\* host\_mat = (double\*)malloc(N \* N \* sizeof(double));

//cudaMalloc((void\*\*)&host\_mat, N \* N \* sizeof(double));

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

host\_mat[i\*N+j] = rand() % 9+1;

//printf("%0.2f ", host\_mat[i\*N+j]);

}

//printf("\n");

}

//printf("\n");

cudaMemcpy(dev\_mat, host\_mat, N \* N \* sizeof(double), cudaMemcpyHostToDevice);

// Запуск ядра CUDA

clock\_t start\_time = clock();

determinant <<<(N+64)/64, 64 >>> (dev\_mat);

cudaThreadSynchronize();

// Копирование результата на хост

cudaMemcpy(host\_mat, dev\_mat, N \* N \*sizeof(double), cudaMemcpyDeviceToHost);

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

for (int j = 0; j < N; j++) {

printf("%0.2f ", host\_mat[i\*N+j]);

}

printf("\n");

}\*/

for (int i = 0; i < N; i++) {

host\_det \*= host\_mat[i\*N+i];

//if (host\_det == 0) break;

}

clock\_t end\_time = clock();

// Вывод результата

printf("Determinant: %0.2f\n", host\_det);

printf("timer: %0.1f mseconds", (double)(end\_time - start\_time));

// Освобождение памяти

cudaFree(dev\_mat);

free(host\_mat);

return 0;

}

**CPU:**

double determinant(double\* mat, double det)

{

// Приведение к треугольной матрице

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

{

// Проверка на ноль диагонального элемента

if (mat[i + i \* N] == 0)

{

det = 0;

return det;

}

int j = 0;

while (j < N - 1) {

// Нахождение коэффициента для обнуления нижней строки

if (i > j) {

double coeff = mat[i \* N + j] / mat[j \* N + j];

// Обнуление нижней строки

for (int z = 0; z < N; z++)

{

mat[i \* N + z] -= coeff \* mat[j \* N + z];

}

}

j++;

}

}

for (int i = 0; i < N; i++) {

det \*= mat[i + N \* i];

}

return det;

}

int main()

{

double det = 1.0;

srand(time(NULL)); // инициализация функции rand значением функции time

double\* mat = (double\*)malloc(N \* N \* sizeof(double));

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

if (rand() % 2 == 1) mat[i \* N + j] = rand() % 9 + 1;

else mat[i \* N + j] = rand() % 9 + 1;

}

}

clock\_t start\_time = clock();

det = determinant(mat, det);

clock\_t end\_time = clock();

// Вывод результата

printf("Determinant: %0.2f\n", det);

printf("timer: %f seconds", (float)(end\_time - start\_time) / CLOCKS\_PER\_SEC);

// Освобождение памяти

free(mat);

return 0;

}