МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №4
по курсу «Программирование графических процессоров»

Работа с матрицам. Метод Гаусса.

Выполнил: М.А. Жерлыгин
Группа: 8О-408Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

Цель работы: Использование объединения запросов к глобальной памяти. Реализация метода Гаусса с выбором главного элемента по столбцу. Ознакомление с библиотекой алгоритмов для параллельных расчетов Thrust.

Вариант №2: Вычисление обратной матрицы.

Программное и аппаратное обеспечение

GPU:

1. Compute capability: 7.5;
2. Графическая память: 4294967296;
3. Разделяемая память: 49152;
4. Константная память: 65536;
5. Количество регистров на блок: 65536;
6. Максимальное количество блоков: (2147483647, 65535, 65535);
7. Максимальное количество нитей: (1024, 1024, 64);
8. Количество мультипроцессоров: 6.

Сведения о системе:

1. Процессор: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
2. Память: 16,0 ГБ;
3. HDD: 237 ГБ.

Программное обеспечение:

1. OS: Windows 10;
2. IDE: Visual Studio 2019;
3. Компилятор: nvcc.

Метод решения

Для того, чтобы использовать библиотеку ***Thrust*** для вычисления максимального элемента в столбце (это необходимо для алгоритма нахождения обратной матрицы), необходимо матрицу хранить по столбцам, а не строкам. В этом случае индексация по сохранённой в таком виде матрице будет выглядеть следующим образом: matrix[i + j * size], где i и j - переменные цикла, size - размер матрицы.
Также для использования метода нахождение максимального элемента необходимо реализовать компаратор, который передаётся в метод.
Сама задача моего варианта решается *методом элементарных преобразований.*

Описание программы

Компаратор:

```
struct compare_value {
    __device__ __host__ bool operator() (const double lhs,
const double rhs) {
        return fabs(lhs) < fabs(rhs);
    }
};
```

Перестановка двух строк местами:

```
__global__ void swap(double* matrix, double* identityMatrix,
int col, int max_idx, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int offset = gridDim.x * blockDim.x;

    for (int i = idx; i < size; i += offset) {
        double tempMatrixValue = matrix[col + i * size];
        matrix[col + i * size] = matrix[max_idx + i * size];
        matrix[max_idx + i * size] = tempMatrixValue;

        double tempIdentityMatrixValue = identityMatrix[col +
i * size];
        identityMatrix[col + i * size] =
identityMatrix[max_idx + i * size];
        identityMatrix[max_idx + i * size] =
tempIdentityMatrixValue;
    }
}
```

Зануление элементов ниже и выше главной диагонали:

```
__global__ void nullifyDown(double* matrix, double*
identityMatrix, int size, int col) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int offsetx = gridDim.x * blockDim.x;
    int offsety = gridDim.y * blockDim.y;

    for (int i = col + idx + 1; i < size; i += offsetx) {
        for (int j = col + idy + 1; j < size; j += offsety) {
            matrix[i + j * size] = - matrix[i + col * size] /
matrix[col + col * size]
```

```
                                        * matrix[col + j * size] +
matrix[i + j * size];
        }
        for (int j = idy; j < size; j += offsety) {
            identityMatrix[i + j * size] = - matrix[i + col *
size] / matrix[col + col * size]
                                                *
identityMatrix[col +  j * size] + identityMatrix[i + j *
size];
        }
    }
}

__global__ void nullifyUp(double* matrix, double*
identityMatrix, int size, int col) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int offsetx = gridDim.x * blockDim.x;
    int offsety = gridDim.y * blockDim.y;


    // Зануляем всё выше главной диагонали
    for (int i = col - idx - 1; i >= 0; i -= offsetx) {
        for (int j = idy; j < size; j += offsety) {
            identityMatrix[i + j * size] = - matrix[i + col *
size] / matrix[col + col * size]
                                                *
identityMatrix[col + j * size] + identityMatrix[i + j * size];
        }
    }
}
```

Деление элементов присоединённой матрицы на коэфицинты элементов в главной матрицы:

```
__global__ void divideIdentityMatrix(double* matrix, double*
identityMatrix, int size) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int offsetx = gridDim.x * blockDim.x;
    int offsety = gridDim.y * blockDim.y;

    // Делим строки присоединённой матрицы на соотв. элемент с
главной диагонали главной матрицы
```

```
    for (int i = idx; i < size; i += offsetx) {
        for (int j = idy; j < size; j += offsety) {
            identityMatrix[i + j * size] /= matrix[i + i *
size];
        }
    }
}
```

Результаты

CPU

n	10^2	10^3	$2 * 10^3$
time	0.010015	15.2192	159.141

GPU <<<(16, 16), (16, 16)>>>

n	10^3	$2 * 10^3$	$4 * 10^3$
time	1.07307	6.9942	57.6501

GPU <<<(32, 32), (32, 32)>>>

n	10^3	$2 * 10^3$	$4 * 10^3$
time	2.36444	10.1052	59.5529

GPU <<<(32, 16), (32, 16)>>>

n	10^3	$2 * 10^3$	$4 * 10^3$
time	1.11093	6.82851	56.2784

Выводы

Я реализовал алгоритм Гаусса, оптимизированный под GPU. Анализируя результаты, можно наблюдать ускорение примерно в 23 раза в отличие от CPU.

Эти результаты можно улучшить. В данной работе я использую глобальную память. Она очень медленная по сравнению с другими типами памяти на GPU.