МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет «Информационные технологии и прикладная математика»
Кафедра «Вычислительная математика и программирование»

Лабораторная работа №4
по курсу «Программирование графических процессоров»

Работа с матрицами. Метод Гаусса.

Студент: Лысенко Д.А.
Группа: 8О-408Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2019

Условие

1. *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.
2. *Вариант 1. Вычисление детерминанта матрицы.*

Входные данные. На первой строке задано число n -- размер матрицы. В

следующих n строках, записано по n вещественных чисел -- элементы матрицы.

n ≤ 10^4

Выходные данные. Необходимо вывести одно число -- детерминант матрицы.

Программное и аппаратное обеспечение

Computecapability: 5.0
Name: GeForce GTX 960M
Total Global Memory: -2147483648
Shared memory per block: 49152
Registers per block: 65536
Warp size: 32
Max threads per block: (1024, 1024, 64)
Max block: (2147483647, 65535, 65535)
Total constant memory: 65536
Multiprocessors count: 5
Процессор (CPU) –Intel Core i5-6300HQ 2.30GHz; 8 ГБ RAM;
OS Microsoft Windows 10 (x64)
CUDA V10.1

Метод решения

Для нахождения детерминанта воспользуемся тем свойством, что определитель треугольной матрицы равен произведению её диагональных элементов. Для приведения матрицы к диагональному виду применим метод Гаусса для строк, не забывая учитывать, что определитель меняет свой знак при обмене строк.

Описание программы

Функция поиска детерминанта, из неё вызываются функции для обмена строк и обнуления элементов стоящих ниже главной диагонали, выполняющиеся параллельно.

```
__host__ double findDet(double * __restrict__  matrix, const uint32_t matrixDim) {
    double det = 1;
    double *matrixDev;
    uint64_t devPitch, hostPitch;
    hostPitch = sizeof(double) * matrixDim;
    cudaMallocPitch(&matrixDev, &devPitch, matrixDim * sizeof(double), matrixDim);
    cudaMemcpy2D(matrixDev, devPitch, matrix, hostPitch, sizeof(double) * matrixDim, matrixDim,
cudaMemcpyHostToDevice);
    for (uint32_t i = 0; i < matrixDim; ++i) {
        thrust::device_ptr<double> currColumnPtr((double*)((char*)matrixDev + devPitch * i));
        thrust::device_ptr<double> start((double*)((char*)matrixDev + devPitch * i) + i);
        thrust::device_ptr<double> end((double*)((char*)matrixDev + devPitch * i) + matrixDim);
        thrust::device_ptr<double> maxInColumnPtr = thrust::max_element(start, end, cmp());
        uint64_t maxInColumnID = (uint64_t)(maxInColumnPtr - currColumnPtr);
```

```
        double maxInColumnVal = *maxInColumnPtr;
        det *= maxInColumnVal;
        if (fabs(maxInColumnVal) < 1e-7) {
            det = 0;
            break;
        }
        if (maxInColumnID != i) {
            det *= -1;
            rowsPermutation << <dim3(64), dim3(64) >> > (matrixDev, matrixDim, devPitch, i,
maxInColumnID);
        }

        if (i != matrixDim - 1) {
            updateBotRows << <dim3(32, 32), dim3(1, 512) >> > (matrixDev, matrixDim, devPitch,
i, maxInColumnVal);
        }

    }
    cudaFree(matrixDev);
    return det;
}
```

Функции обмена строк и обнуления элементов стоящих ниже главной диагонали, выполняющиеся на девайсе.

```
__global__ void rowsPermutation(double * __restrict__  matrix, const uint32_t matrixDim, const
uint64_t pitch,
    const uint64_t midInColumnID, const uint64_t maxInColumnID) {
    uint32_t idx = threadIdx.x + blockIdx.x * blockDim.x + midInColumnID;
    uint32_t offsetx = blockDim.x * gridDim.x;
    for (uint32_t i = idx; i < matrixDim; i += offsetx) {
        double tmp = *((double*)((char*)matrix + pitch * i) + midInColumnID);
        *((double*)((char*)matrix + pitch * i) + midInColumnID) = *((double*)((char*)matrix +
pitch * i) + maxInColumnID);
        *((double*)((char*)matrix + pitch * i) + maxInColumnID) = tmp;
    }
    return;
}

__global__ void updateBotRows(double * __restrict__  matrix, const uint32_t matrixDim, const
uint64_t pitch,
    const uint64_t midInColumnID, const double midInColumnVal) {
    uint32_t idx = threadIdx.x + blockIdx.x * blockDim.x + midInColumnID + 1;
    uint32_t idy = threadIdx.y + blockIdx.y * blockDim.y + midInColumnID + 1;
    uint32_t offsetx = blockDim.x * gridDim.x;
    uint32_t offsety = blockDim.y * gridDim.y;
    double factor;
    for (uint32_t j = idy; j < matrixDim; j += offsety) {
        factor = *((double*)((char*)matrix + pitch * midInColumnID) + j);
        if (fabs(factor) < 1e-7) continue;
        for (uint32_t i = idx; i < matrixDim; i += offsetx) {
            *((double*)((char*)matrix + pitch * i) + j) -= *((double*)((char*)matrix + pitch *
i) + midInColumnID) * factor / midInColumnVal;
        }
    }
    return;
}
```

Оценка производительности

Test size	CUDA <dim3(64), dim3(64)> <dim3(32, 32), dim3(1, 512) > TIME	CUDA <dim3(64), dim3(64)> <dim3(32, 32), dim3(32, 32) > TIME	CUDA <dim3(64), dim3(64)> <dim3(16, 16), dim3(16,16) > TIME	CUDA <dim3(64), dim3(64)> <dim3(1, 16), dim3(16, 64) > TIME	CPU TIME
1000	3597.18	5018.77	3377.14	3336.78	9892
100	189.76	278.26	150.78	150.76	6.0
30	59.64	81.22	57.05	50.17	-

Вывод

Метод гаусса имеет кубическую сложность, что делает вычисления как на GPU, так и на CPU достаточно трудоемким процессов. Как видно из тестов, больший размер блоков и сетки не всегда дает прирост производительности, это связанно, во-первых, с тем, что на небольших матрицах большинство нитей простаивают, не совершая полезных вычислений, во-вторых, во время работы алгоритма, для всех нитей с одинаковым адресом по оси y и разным по оси x нам приходится обращаться в одну и ту же область памяти, что создает конфликт и время работы алгоритма существенно увеличивается.