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 матрицами. Метод Гаусса.

Выполнил: И.Д. Черненко
Группа: 8О-406Б
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 задания: Вариант 7, Решение матричного уравнения..

Входные данные: На первой строке заданы числа n, m и k -- размеры матриц. В следующих n строках, записано по m вещественных чисел -- элементы матрицы A. Далее записываются n строк, по k чисел -- элементы матрицы B. $n * m + m * k + n * k \leq 1.2 * 10^8$.

Выходные данные: Необходимо вывести на m строках, по k чисел -- элементы неизвестной матрицы X.

Программное и аппаратное обеспечение

Compute capability : 6.1
Name : GeForce GTX 1050
Total Global Memory : 2096103424
Shared memory per block : 49152
Registers per block : 65536
Max threads per block : (1024, 1024, 64)
Max block : (2147483647, 65535, 65535)
Total constant memory : 65536
Multiprocessors count : 5

OS:Linux
compiler:nvcc
IDE:VS Code
Editor:nano

Метод решения

Основная идея метода Гаусса очень проста и её можно разбить на два этапа:
1. На первом этапе осуществляется так называемый прямой ход, когда путём элементарных преобразований над строками систему приводят к ступенчатой или треугольной форме, либо устанавливают, что система несовместна. Для этого среди элементов первого столбца матрицы выбирают ненулевой, перемещают содержащую его строку в крайнее верхнее положение, делая эту строку первой. Далее ненулевые элементы первого столбца всех нижележащих строк обнуляются путём вычитания из каждой строки первой строки, домноженной на отношение первого элемента этих строк к первому элементу первой строки. После того, как указанные преобразования были совершены, первую строку и первый столбец мысленно вычёркивают и продолжают, пока не останется матрица нулевого размера. Если на какой-то из итераций среди элементов первого столбца не нашёлся ненулевой, то переходят к следующему столбцу и проделывают аналогичную операцию.

2. На втором этапе осуществляется так называемый обратный ход, суть которого заключается в том, чтобы выразить все получившиеся базисные переменные через небазисные и построить фундаментальную систему решений, либо, если все переменные являются базисными, то выразить в численном виде единственное решение системы линейных уравнений. Эта процедура начинается с последнего уравнения, из которого выражают соответствующую базисную переменную (а она там всего одна) и подставляют в предыдущие уравнения, и так далее, поднимаясь по «ступенькам» наверх. Каждой строчке соответствует ровно одна базисная переменная, поэтому на каждом шаге, кроме последнего (самого верхнего), ситуация в точности повторяет случай последней строки.

При этом в нашем случае мы используем алгоритм с выбором главного элемента, идея которого состоит в том, чтобы на очередном шаге исключать не следующее по номеру неизвестное, а то неизвестное, коэффициент при котором является наибольшим по модулю. Таким образом, в качестве ведущего элемента здесь выбирается главный, т.е. наибольший по модулю. Так мы не получим деления на 0.

Описание программы

Вся программа была реализована в одном файле **main.cu**.

Реализованные ядра:

```
__global__ void down(double* matrix, uint32_t n, uint32_t m, uint32_t row,
uint32_t clm) {
    uint32_t offsetx = blockDim.x * gridDim.x;
    uint32_t offsety = blockDim.y * gridDim.y;
    uint32_t idx = blockDim.x * blockIdx.x + threadIdx.x;
    uint32_t idy = blockDim.y * blockIdx.y + threadIdx.y;

    for (uint32_t j = clm + 1 + idy; j < m; j += offsety) {
        for (uint32_t i = row + 1 + idx; i < n; i += offsetx) {
            matrix[i + n * j] -= matrix[n * clm + i] / matrix[n * clm + row] *
matrix[n * j + row];
        }
    }
}

__global__ void up(double* matrix, uint32_t n, uint32_t m, uint32_t k,
uint32_t row, uint32_t clm) {
    uint32_t offsetx = blockDim.x * gridDim.x;
    uint32_t offsety = blockDim.y * gridDim.y;
    uint32_t idx = blockDim.x * blockIdx.x + threadIdx.x;
    uint32_t idy = blockDim.y * blockIdx.y + threadIdx.y;

    for (uint32_t j = m + idy; j < m + k; j += offsety) {
        for (uint32_t i = idx; i < row; i += offsetx) {
            matrix[i + n * j] -= matrix[n * clm + i] / matrix[n * clm + row] *
matrix[n * j + row];
        }
    }
}

__global__ void swap(double* matrix, uint32_t n, uint32_t m, uint32_t clm,
uint32_t left, uint32_t right) {
    uint32_t offsetx = blockDim.x * gridDim.x;
    uint32_t idx = blockDim.x * blockIdx.x + threadIdx.x;

    for (uint32_t i = idx + clm; i < m; i += offsetx) {
        double tmp = matrix[n * i + left];
        matrix[n * i + left] = matrix[n * i + right];
        matrix[n * i + right] = tmp;
    }
}
```

Результаты

block	threads	time
64	64	314.766ms
128	64	314.09ms
128	128	310.978ms
256	128	319.5495ms
256	256	315.6925ms
1024	256	312.192ms
CPU		4921.875ms

Из полученных данных можно сделать вывод, что при правильных настройках вычисления на **GPU** превосходят **CPU** примерно в **16 раз.**

Выводы

Данный алгоритм можно было бы использовать в задачах нахождения решения СЛАУ. Написание базовой версии программы заняло не очень много времени, после того как был осознано линейное взаимодействие с элементами матрицы. Как можно заметить из результатов, выполнение таких алгоритмов на GPU намного эффективнее чем на CPU, благодаря распараллеливанию вычислительных процессов, кроме того важно отметить удобство использования Thrust для параллельных рассчетов.