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: К.О. Вахрамян
Группа: 8О-406Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

Кратко описывается задача:

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 №2.
   Вычисление обратной матрицы.

Программное и аппаратное обеспечение

GPU:

--- General Information for device ---
Name: NVIDIA GeForce GTX 1650
Compute capability: 7.5
Clock rate: 1560000
Device copy overlap: Enabled
Kernel execution timeout : Enabled
--- Memory Information for device ---
Total global mem: 4100521984
Total constant Mem: 65536
Max mem pitch: 2147483647
Texture Alignment: 512
--- MP Information for device ---
Multiprocessor count: 16
Shared mem per mp: 49152
Registers per mp: 65536
Threads in warp: 32
Max threads per block: 1024
Max thread dimensions: (1024, 1024, 64)
Max grid dimensions: (2147483647, 65535, 65535)

CPU:

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
Address sizes: 39 bits physical, 48 bits virtual
CPU(s): 8
On-line CPU(s) list: 0-7
Thread(s) per core: 2
Core(s) per socket: 4
Socket(s): 1
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 158

Model name: Intel(R) Core(TM) i5-9300HF CPU @ 2.40GHz
Stepping: 13
CPU MHz: 1274.759
CPU max MHz: 2400.0000
CPU min MHz: 800.0000
BogoMIPS: 4800.00
Virtualization: VT-x
L1d cache: 128 KiB
L1i cache: 128 KiB
L2 cache: 1 MiB
L3 cache: 8 MiB

OS:
Linux Mint 20

Compiler:
nvcc

Code Editor:
VS Code

Метод решения

Обратную матрицу будем искать методом Гаусса с поиском ведущего элемента.
Необходимо дописать к исходной матрице единичную. Тогда количество итераций по столбцам возрастет в 2 раза.
Прямой ход:
На каждом шаге выбирается элемент $a_{ii} \neq 0$ (если он равен 0, то первую строку переставляем с какой-либо нижележащей строкой)

$$a_{jk} = a_{jk} - a_{ik}\frac{a_{ji}}{a_{ii}}, i = 1..n, k = i+1..2n, j = i+1..n$$

Обратный ход:
Подсчет происходит снизу вверх.

$$a_{jk} = a_{jk} - a_{ik}\frac{a_{ji}}{a_{ii}}, i = n..1, k = i+1..2n, j = i-1..1$$

После прямого и обратного хода необходимо нормализовать матрицу. Нормализуем только обратную матрицу, т. к. исходную больше нигде не используем.

$$a_{ij} = \frac{a_{ij}}{a_{ii}}, i = 1..n, j = n..2n$$

Описание программы

Матрицу храним в линеаризованном виде. Значения хранятся по столбцам, чтобы легче искать максимальный элемент столбце.

```
double* matrix = (double*)malloc(sizeof(double) * n * n * 2);
for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
                scanf("%lf", &matrix[i + j * n]);
for (int i = 0; i < n; i++)
        for (int j = n; j < 2 * n; j++)
                matrix[i + j * n] = (i == j - n) ? 1. : 0.;
```

Подсчет обратной матрицы реализован в отдельной функции. При прямом ходе в каждом столбце ищется максимальный по модулю элемент, соответствующие строки меняются местами.

```
thrust::device_ptr<double> max = thrust::max_element(&thrust_matrix[i + i * n],
&thrust_matrix[n + i * n], comp);
int max_idx = max - (thrust_matrix + i * n);
if (max_idx != i)
        SwapRows<<<dim3(xBlocks), dim3(xThreads)>>>(dev_matrix, n, i,
        max_idx);
```

Затем происходит прямой ход для данного столбца по всем строкам

```
__global__ void ForwardGauss(double* dev_matrix, int n, int i) {
int idx = threadIdx.x + blockDim.x * blockIdx.x;
int idy = threadIdx.y + blockDim.y * blockIdx.y;
int offsetx = blockDim.x * gridDim.x;
int offsety = blockDim.y * gridDim.y;

for (int k = idy + i + 1; k < 2 * n; k += offsety)
        for (int j = idx + i + 1; j < n; j += offsetx)
                dev_matrix[k * n + j] -= (dev_matrix[k * n + i] * dev_matrix[i * n + j] /
                dev_matrix[i + i * n]);
}
```

В цикле по j идем с i+1 т. к.: 1)обнуляемый столбец нигде не используется, 2) если бы мы шли с i то то обнулили бы значение a[k * n +i], которое используется в других потоках.

Затем происходит обратный ход:

```
__global__ void BackwardGauss(double* dev_matrix, int n, int i) {
int idx = threadIdx.x + blockDim.x * blockIdx.x;
int idy = threadIdx.y + blockDim.y * blockIdx.y;
int offsetx = blockDim.x * gridDim.x;
int offsety = blockDim.y * gridDim.y;
for (int k = idy + i + 1; k < 2 * n; k += offsety)
        for (int j = i - 1 - idx; j >= 0; j -= offsetx)
```

```
                dev_matrix[k * n + j] -= (dev_matrix[k * n + i] * dev_matrix[i * n + j] /
                dev_matrix[i + i * n]);
}
```

Здесь логика та же, j != i иначе не получится распараллелить.

В конце нормируем значения.

```
__global__ void Normalize(double* dev_matrix, int n) {
int idx = threadIdx.x + blockDim.x * blockIdx.x;
int idy = threadIdx.y + blockDim.y * blockIdx.y;
int offsetx = blockDim.x * gridDim.x;
int offsety = blockDim.y * gridDim.y;

for (int i = idy; i < n; i += offsety)
for (int j = n + idx; j < 2 * n; j += offsetx)
dev_matrix[i + j * n] /= dev_matrix[i + i * n];
}
```

Результаты

Все время представлено в миллисекундах.

	10x10	100x100	1000x1000
<<<(1,1),(1,32)>>>	0.93	38.24	21117.38
<<<(1,1),(32,32)>>>	0.92	18.74	9483.52
<<<(1,32),(1,32)>>>	0.91	15.48	1925.72
<<<(1,128),(1,128)>>>	0.92	18.43	1968
<<<(1,32),(32,32)>>>	0.91	6.54	1080.16
<<<(1,128),(128,128)>>>	1.47	12.22	1062.74
<<<(32,32),(32,32)>>>	1.12	9.45	589.18
<<<(128,128),(128,128)>>>	0.85	5.71	242.85
<<<(128,1024),(128,1024)>>>	0.84	6.02	239.86
CPU	0.02	11.56	23158.98

Выводы

Методы линейной алгебры в общем и метод Гаусса в частности имеют огромное применение в разнообразных задачах. В целом, алгоритм был знаком мне, я реализовывал его в курсе Численных методов, однако с применением CUDA можно видеть, какой прирост производительности (почти в 100 раз!) достигается на тестах, где входные данные порядка 10^6 .