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: В.А. Петросян
Группа: 8О-408Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2020

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: 3.0
2. Графическая память: 2147155968
3. Разделяемая память: 49152
4. Константная память: 65536
5. Количество регистров на блок: 65536
6. Максимальное количество блоков: (65535, 65535, 65535)
7. Максимальное количество нитей: (1024, 1024, 64)
8. Количество мультипроцессоров: 6

Сведения о системе:

1. Процессор: Intel Core i7-Q720 1.60GHz
2. Память: 8 ГБ
3. HDD: 465 ГБ

Программное обеспечение:

1. OS: Windows 7
2. IDE: Visual Studio 2019
3. Компилятор: nvcc

Метод решения

Есть несколько интересных идей, которые нужно отметить. В остальном эта лабораторная работа повторяет материал из курса Численных методов.

1) Матрицу хранил по столбцам, а не по строчкам как это принято делать. Такой метод хранения матрицы не распространен и может вызывать затруднение в обращение к конкретному элементу по индексу i, j. Приходиться писать matr[j * n + i] вместо matr[i * n + j].
2) Когда мы выбираем главный элемент на конкретном шаге алгоритма, нужно не забыть про то, что нам интересен максимум по модулю, а не по значению.
3) Если хочется использовать поиск максимума через библиотеку thrust, то придётся написать компаратор. Компаратор – это функция, цель которой объяснить алгоритму поиска (в данном случае max_element) что мы считаем за понятие меньше. Используя компаратор, алгоритм может сравнить элементы матрицы нужным нам образом.

Теперь комбинируя идеи из выше написанных пунктов можно спокойно использовать библиотеку thrust для поиска максимального элемента в столбце.

Описание программы

```
class Comparator{
public:
    __host__ __device__ bool operator()(const double a, const double b) const{
        return fabs(a) < fabs(b);
    }
};
```

Тот самый компаратор из третьего пункта. Сравнивает элементы по модулю.

```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
         cin >> matrix[j * n + i];
    }
}
```

Сохранение матрицы по столбцам

```
const thrust::device_ptr<double> ptr = thrust::device_pointer_cast(dev_matrix);

const Comparator comp;
dim3 block(32, 16);
dim3 thread(32, 16);

for (i = 0; i < n - 1; ++i) {
    const int max_idx = thrust::max_element(ptr + i*n + i, ptr + (i + 1)*n, comp)
                        - ptr - i * n;
    if (max_idx != i){
       swapLines<<<256, 256>>>(dev_matrix, dev_identity, n, i, max_idx);
    }
    makeDownNull<<<block, thread>>>(dev_matrix, dev_identity, n, i);
}
```

На каждой итерации производится поиск максимального элемента с помощью функции **thrust::max_element.** После этого мы вычитаем из итератора **ptr**, указывающий на начало матрицы, чтобы получить его полный номер в массиве. Теперь вычитая из этого полного номера, индекс текущего столбца умноженный на количество элементов в одном столбце получим индекс элемента внутри столбца(строки). Теперь, сравнивая его индекс с текущим шагом алгоритма, можем понять, нужно ли нам менять строчки местами или нет. После того как главный элемент выбран можно начинать процесс обнуления с помощью ядра **makeDownNull.**

```
for (i = n - 1; i > 0; i--) {
    makeUpNull<<<block, thread>>>(dev_matrix, dev_identity, n, i);
}
devideIdentity<<<block, thread>>>(dev_matrix, dev_identity, n);
```

После обнуляем значения выше главной диагонали, а потом вызываем ядро **devideIdentity**, чтобы закончить процесс получения единичной матрицы из нашей исходной.

```
__global__ void swapLines(double * matrix, double* identity,int n,int i, int j){

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int offset = gridDim.x * blockDim.x;

    int k;
    double tmp;
    for (k = idx; k < n; k += offset) {
        tmp = matrix[k * n + i];
        matrix[k * n + i] = matrix[k * n + j];
        matrix[k * n + j] = tmp;

        tmp = identity[k * n + i];
        identity[k * n + i] = identity[k * n + j];
        identity[k * n + j] = tmp;
    }
}
```

Ядро отвечает за обмен строчек(столбцов) с индексами i, j местами.

```
__global__ void devideIdentity(double* matrix, double* identity, int n) {
    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;
    int i, j;
    for (i = idx; i < n; i += offsetx) {
        for (j = idy; j < n; j += offsety) {
            identity[j * n + i] /= matrix[i * n + i];
        }
    }
}
```

Ядро делит значения правой матрицы(**identity**) на значение элементов на главной диагонали исходной матрицы(**matrix**), чтобы завершить процесс подсчета обратной матрицы.

```
__global__ void makeDownNull(double* matrix, double* identity, int n, int x) {
    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;

    int i, j;
    double particion;
    for (i = x + 1 + idx; i < n; i += offsetx) {
        particion = -matrix[x * n + i] / matrix[x * n + x];
        for (j = x + 1 + idy; j < n; j += offsety) {
            matrix[j*n + i] = particion * matrix[j*n + x] + matrix[j*n + i];
        }
        for (j = idy; j < n; j += offsety) {
            identity[j*n + i] = particion*identity[j*n+x] + identity[j*n + i];
        }
    }

}
```

Важная особенность данного ядра, что оно фактически не обнуляет значения под главным элементов, так как это могло бы вызвать состояние гонки и недетерминированность результата.

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

    int i, j;
    double particion;
    for (i = x - 1 - idx; i >= 0; i -= offsetx) {
        particion = -matrix[x * n + i] / matrix[x * n + x];
        for (j = idy; j < n; j += offsety) {
          identity[j*n + i] = particion*identity[j*n + x] + identity[j*n + i];
        }
    }
}
```

Важная особенность данного ядра, что оно не обнуляет значения в исходной матрице(**matrix**) в целях экономии времени. Нам не важно, какие будут лежать значения в исходной матрице по завершении алгоритма.

Результаты

CPU

N	10^2	10^3	$2 * 10^3$
Time (sec)	0.010015	15.2192	159.141

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

N	10^3	$2 * 10^3$	$4 * 10^3$
Time (sec)	1.11093	6.82851	56.2784

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

N	10^3	$2 * 10^3$	$4 * 10^3$
Time (sec)	2.36444	10.1052	59.5529

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

N	10^3	$2 * 10^3$	$4 * 10^3$
Time (sec)	1.07307	6.9942	57.6501

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

N	10^3	$2 * 10^3$	$4 * 10^3$
Time (sec)	1.005	6.88209	65.5802

GPU <<< (64, 16), (32, 16)>>>

N	10^3	$2 * 10^3$	$4 * 10^3$
Time (sec)	1.1955	6.79831	54.7871

Чтобы тестировать программу на матрицах большого размера пришлось написать программный комплекс для генерации данных. Идея очень простая. Сначала в одной программе посчитаем большое количество простых чисел через решето Эратосфена и запишем их в файл.

```
vector<bool> prime (n+1, true);
    prime[0] = prime[1] = false;
    for (long long i = 2; i <= n; ++i){
        if (prime[i] == true){
            if (i * 1ll * i <= n){
                for (long long j = i * 1ll * i; j <= n; j += i){
                    prime[j] = false;
                }
            }
        }
    }
```

Теперь напишем другую программу, которая будет считывать эти простые числа и заполнять ими диагонали матрицы.

```
for(int i = n - 1; i > 0; --i){
    cin >> value1 >> value2;
    int idx = i;
    for(int j = 0; idx < n; ++j){
        matrix[idx * n + j] = value1;
        matrix[j * n + idx] = value2;
        ++idx;
    }
}
```

Если матрицу заполнять таким образом, то не будет линейно зависимых столбцов(строк).

Результат работы для N = 10

```
user19@server-i72:~/lab4/bench$ g++ generateMatrix.cpp
user19@server-i72:~/lab4/bench$ ./a.out
user19@server-i72:~/lab4/bench$ cat matr.txt
10
67 61 53 43 37 29 19 13 7 3
59 67 61 53 43 37 29 19 13 7
47 59 67 61 53 43 37 29 19 13
41 47 59 67 61 53 43 37 29 19
31 41 47 59 67 61 53 43 37 29
23 31 41 47 59 67 61 53 43 37
17 23 31 41 47 59 67 61 53 43
11 17 23 31 41 47 59 67 61 53
5 11 17 23 31 41 47 59 67 61
2 5 11 17 23 31 41 47 59 67
user19@server-i72:~/lab4/bench$
```

Конечно, была идея заполнять случайными числами, взяв большой диапазон генерации рандома. Вероятность линейно зависимых строк, в таком случае, будет очень мала, но я не стал рисковать.

Выводы

В результате выполнения лабораторной работы изучил и реализовал алгоритм Гаусса оптимизированный под GPGPU. Результаты показывают, что мой алгоритм на GPU работает в 2-3 раза быстрее, чем версия на CPU. Я уверен, что можно разогнать ещё сильнее.
Например, мне очень не нравится хранить матрицу по столбцам. Единственный профит от этого, что можно через библиотеку thrust искать максимум, но зато в остальных ядрах, как я понял, это только мешает. Возможно, если хранить матрицу в привычном для нас образе, можно будет получить прирост производительности. Так же стоит отметить, что в данной лабораторной работе использую только глобальную память. Она очень медленная по сравнению с другими типами памяти на GPU. Для более внушительных результатов нужно будет использовать все возможности GPU.