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: А.В. Ефимов
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

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

В качестве вещественного типа данных необходимо использовать тип данных double. Библиотеку **Thrust** использовать только для поиска максимального элемента на каждой итерации алгоритма. В качестве нулевого значения использовать 10^{-7}. Все результаты выводить с относительной точностью 10^{-10}.

Вариант 7. Решение матричного уравнения.

Программное и аппаратное обеспечение

```
##### CUDA Info

/opt/cuda/extras/demo_suite/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce MX150"
  CUDA Driver Version / Runtime Version          11.4 / 11.4
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 2003 MBytes (2099904512
bytes)
  ( 3) Multiprocessors, (128) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            1532 MHz (1.53 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072,
65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048
layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
```

```
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA Runtime
Version = 11.4, NumDevs = 1, Device0 = NVIDIA GeForce MX150
Result = PASS

##### CPU Info

Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Address sizes:                   39 bits physical, 48 bits virtual
Byte Order:                      Little Endian
CPU(s):                          8
On-line CPU(s) list:             0-7
Vendor ID:                       GenuineIntel
Model name:                      Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
CPU family:                      6
Model:                           142
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
Stepping:                        10
CPU max MHz:                     3400.0000
CPU min MHz:                     400.0000
BogoMIPS:                        3601.00
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts
rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64
monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2
x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm
abm 3dnowprefetch cpuid_fault epb invpcid_single pti ibrs ibpb stibp
tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2
smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt
xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window
hwp_epp
Virtualization:                  VT-x
L1d cache:                       128 KiB (4 instances)
L1i cache:                       128 KiB (4 instances)
L2 cache:                        1 MiB (4 instances)
L3 cache:                        6 MiB (1 instance)
NUMA node(s):                    1
NUMA node0 CPU(s):               0-7
Vulnerability Itlb multihit:     KVM: Mitigation: VMX disabled
Vulnerability L1tf:              Mitigation; PTE Inversion; VMX conditional
cache flushes, SMT vulnerable
Vulnerability Mds:               Vulnerable: Clear CPU buffers attempted,
no microcode; SMT vulnerable
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and
__user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Full generic retpoline, IBPB
conditional, IBRS_FW, STIBP conditional, RSB filling
Vulnerability Srbds:             Vulnerable: No microcode
Vulnerability Tsx async abort:   Not affected
```

```
##### RAM Info

               total        used        free      shared  buff/cache
available
Mem:           7.6Gi       1.7Gi       4.6Gi       633Mi       1.4Gi
5.1Gi
Swap:             0B          0B          0B
```

Метод решения

Решение матричного уравнения сводится к решению СЛАУ, где элементы вектора b – матричные строки. Соответственно для решения нужно составить алгоритм решения обычного СЛАУ – матрица приводится к единичному виду, где первые единицы каждой строки соответствуют строкам неизвестной матрицы. Все остальные строки равны нулю.

Описание программы

Так как подсчеты по большей части проходят по столбцам, то и матрицы хранятся в транспонированном виде, чтобы элементы столбцов были последовательными и вызовы к глобальной памяти объединялись.

Матрицы также хранятся в одной области, так как подсчеты в одной матрице напрямую влияет на другую.

Ядра, которые соответственно:

- Меняет строки транспонированной матрицы;
- Вычитает выбранную строку из других, ниже её (выбирается максимальная), причем первые элементы игнорируются (можно считать, что все они равны нулю, кроме элемента выбранной строки)
- Вычитает выбранную строку из других, выше её. Это нужно для нормализации верхней матрицы, полученной после предыдущего ядра.

```
__global__
void gpu_swap_rows(double* matrix_m,
                   long n, long m, long k, long row_a, long row_b) {
    long idx = blockDim.x * blockIdx.x + threadIdx.x;
    long stride = blockDim.x * gridDim.x;

    for (long i = idx; i < (m + k); i += stride) {
        long col = i * n;
        double t = matrix_m[col + row_a];
        matrix_m[col + row_a] = matrix_m[col + row_b];
        matrix_m[col + row_b] = t;
    }
}

__global__
void gpu_diff_rows(double* matrix_m,
                long n, long m, long k, long start_col, long max_row)
{
```

```
    long idx = blockDim.x * blockIdx.x + threadIdx.x;
    long idy = blockDim.y * blockIdx.y + threadIdx.y;
    long stridex = blockDim.x * gridDim.x;
    long stridey = blockDim.y * gridDim.y;

    for (long i = idy + start_col + 1; i < (m + k); i += stridey) {
        long col = i * n;
        double val = matrix_m[col + max_row] / matrix_m[start_col *
n + max_row];

        for (long j = idx + max_row + 1; j < n; j += stridex) {
            matrix_m[col + j] -= val * matrix_m[start_col * n + j];
        }
    }

}

__global__
void gpu_normalize_rows(double* matrix_m,
                    long n, long m, long k, long col, long row) {
    long idx = blockDim.x * blockIdx.x + threadIdx.x;
    long idy = blockDim.y * blockIdx.y + threadIdx.y;
    long stridex = blockDim.x * gridDim.x;
    long stridey = blockDim.y * gridDim.y;

    double div = matrix_m[col * n + row];

    for (long i = idy + col + 1; i < (m + k); i += stridey) {
        double temp = matrix_m[i * n + row];

        for (long j = idx; j < row; j += stridex) {
            matrix_m[i * n + j] -= temp * matrix_m[col * n + j] /
div;
        }
    }
}
```

Максимальные элементы в столбцах ищутся с помощью функции библиотеки **Thrust**:

```
…
thrust::device_ptr<double> col_ptr, max_ptr;

for (; col < m; ++col) {
    col_ptr = thrust::device_pointer_cast(dev_matrix_m + col * n);
    max_ptr = thrust::max_element(col_ptr + row, col_ptr + n, comp);
…
```

Результаты

Во всех тестах $k = 100$, меняется значения n и m.

CPU		
n	m	Время
100	100	65.495ms
100	500	221.043ms
100	1000	411.823ms
100	2500	997.724ms
100	5000	1984.322ms
500	100	287.108ms
1000	100	572.179ms
2500	100	1428.275ms
5000	100	2852.527ms

Подбирается количество потоков для ядра, меняющего строки местами (другие два ядра работают с 512 потоками):

<<<64, 64>>>		
n	m	Время
100	100	12.5485ms
100	500	17.0599ms
100	1000	24.795ms
100	2500	46.5558ms
100	5000	87.6311ms

<<<64, 128>>>		
n	m	Время
100	100	12.6103ms
100	500	17.0836ms
100	1000	25.0245ms
100	2500	47.1847ms
100	5000	89.108ms

<<<64, 256>>>		
n	m	Время
100	100	13.3421ms
100	500	17.2595ms
100	1000	25.128ms
100	2500	47.8217ms
100	5000	89.8748ms

<<<64, 512>>>		
n	m	Время
100	100	12.5474ms
100	500	17.2347ms
100	1000	25.426ms
100	2500	48.1885ms
100	5000	89.8811ms

<<<64, 1024>>>		
n	m	Время
100	100	12.715ms
100	500	17.624ms
100	1000	25.4235ms
100	2500	46.8207ms
100	5000	88.7327ms

Подбирается количество поток для ядра, выполняющего вычитание (ядро, меняющее местами строки, имеет 64 потока)

<<<dim3(8, 64), dim3(8, 8)>>>		
n	m	Время
100	100	10.7864ms
500	100	14.0623ms
1000	100	20.5583ms
2500	100	48.3497ms
5000	100	78.4428ms

<<<dim3(8, 64), dim3(8, 16)>>>		
n	m	Время
100	100	11.1922ms
500	100	14.6156ms
1000	100	20.2644ms
2500	100	49.0452ms
5000	100	80.5138ms

<<<dim3(8, 64), dim3(8, 32)>>>		
n	m	Время
100	100	11.4792ms
500	100	21.0605ms
1000	100	15.1123ms
2500	100	51.1478ms
5000	100	82.4619ms

<<<dim3(8, 64), dim3(8, 64)>>>		
n	m	Время
100	100	13.002ms
500	100	16.4205ms
1000	100	23.0082ms
2500	100	52.6012ms
5000	100	89.1288ms

<<<dim3(8, 64), dim3(8, 128)>>>		
n	m	Время
100	100	16.6889ms
500	100	19.406ms
1000	100	26.336ms
2500	100	59.0715ms
5000	100	104.051ms

Выводы

- Данные следует хранить с учетом обращения GPU данных – если потоки обращаются к ним последовательно, то такие запросы будут объединятся;
- Один *if* может оказывать больше влияние на работу ядра на больших данных.