МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №2
по курсу «Программирование графических процессоров»

Обработка изображений на GPU. Фильтры.

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

Москва, 2021

Условие

Научиться использовать GPU для обработки изображений. *Использование текстурной памяти*.

Вариант 6: Выделение контуров. Метод Превитта.

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

После получения названий вводных и выводных файлов, из первого необходимо считать сами данные: ширина w и высота h матрицы и саму матрицу пикселей (структур *RGBA*, хотя в данном задании компоненту *Alpha* можно сбросить) количеством $w \cdot h$.

После загрузки матрицы, она помещается в текстурную память с параметрами Clamp, FilterPoint и численной индексацией.

Потоки запущенного ядра начинают проходиться фильтром по текстуре и считать градиент в каждом пикселе. Для подсчета градиента необходимо обращаться к пикселям, окружающим текущий. Чтобы достичь этого при индексации, нужно, чтобы индекс текущего пикселя был равен $(0,0)$. Для этого достаточно вычесть единицу из индексов на каждой оси. Тогда:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \rightarrow \begin{pmatrix} a_{-1,-1} & a_{-1,0} & a_{-1,1} \\ a_{0,-1} & a_{0,0} & a_{0,1} \\ a_{1,-1} & a_{1,0} & a_{1,1} \end{pmatrix}$$

Выходы за массивы обрабатываются самостоятельно текстурной памятью.

После получения пикселя, он конвертируется в монотонный серый цвет и умножается на значение фильтра для этого пикселя для каждой оси (индексация происходит первым методом, без смещения), результат прибавляется в суммы G_x и G_y для соответствующих ось.

Полученные суммы описывают градиент, величина которого высчитывается по формуле $G = \sqrt{G_x^2 + G_y^2}$.

Описание программы

Ядро принимает на вход массив в глобальной памяти, в который будут записаны результаты подсчетов градиентов (плоский массив, достаточно большой чтобы поместить всю обработанную матрицу), а также размеры матрицы/текстуры.

```
__global__
void kernel(uchar4 *image, int32_t w, int32_t h) {
    // Индекс потока по каждой оси и сдвиг блока
    int idx     = blockDim.x * blockIdx.x + threadIdx.x;
    int idy     = blockDim.y * blockIdx.y + threadIdx.y;
    int offsetx = blockDim.x * gridDim.x;
    int offsety = blockDim.y * gridDim.y;

    // Значения фильтров по каждой оси
    int x_filter[3][3] = {
        {-1,  0,  1},
```

```
        {-1,  0,  1},
        {-1,  0,  1}
    };
    int y_filter[3][3] = {
        {-1, -1, -1},
        { 0,  0,  0},
        { 1,  1,  1}
    };

    for(int y = idy; y < h; y += offsety) {
        for(int x = idx; x < w; x += offsetx) {
            double gx = 0;
            double gy = 0;

            // Подсчет градиента вокруг текущего пикселя
            for (int i = 0; i < 3; ++i) {
                for (int j = 0; j < 3; ++j) {
                    double grey_pixel = convert_greyscale(
tex2D(tex, x + i - 1, y + j - 1) );
                    gx += x_filter[i][j] * grey_pixel;
                    gy += y_filter[i][j] * grey_pixel;
                }
            }

            // Must be 0 to 255 inclusive, else overflow
            unsigned char g = min( sqrt(gx * gx + gy * gy), 255.0 );

            // Запись в выходной массив результата
            int offset = y * w + x;
            image[offset].x = g;
            image[offset].y = g;
            image[offset].z = g;
            image[offset].w = 0;
        }
    }

}
```

Результаты

CPU	
100	2ms
500	43ms
1000	173ms
2500	1143ms
5000	4607ms

<<< dim3(32, 32), dim3(32, 32) >>>	
100	254.30us
500	1.4327ms
1000	5.1389ms
2500	30.157ms
5000	115.79ms

<<< dim3(64, 64), dim3(32, 32) >>>	
100	830.34us
500	2.0188ms
1000	5.7369ms
2500	31.050ms
5000	115.99ms

<<< dim3(128, 128), dim3(32, 32) >>>	
100	3.0908ms
500	4.2895ms
1000	8.0227ms
2500	33.654ms
5000	123.38ms

<<< dim3(256, 256), dim3(32, 32) >>>	
100	12.117ms
500	13.322ms
1000	17.117ms
2500	42.874ms
5000	131.58ms

Изображения:

Выводы

- В отличие от задачи выполнения операции на двух векторах, эта задача отлично показывает преимущество над вычислением на процессорах, достигая почти *40x* большей скорости на GPU, чем на CPU.
- Задача вычисления сверток является основной в глубоком обучении. Применение GPU значительно ускорит процесс обучения (особенно потому, что сверточный слой обычно не один), а соответственно и процесс создания сети и тестирования на разных гиперпараметрах.