Accelerators and Accelerated systems

046278

Authors:

Chris Shakkour, 208157826, Christian.s@campus.technion.ac.il

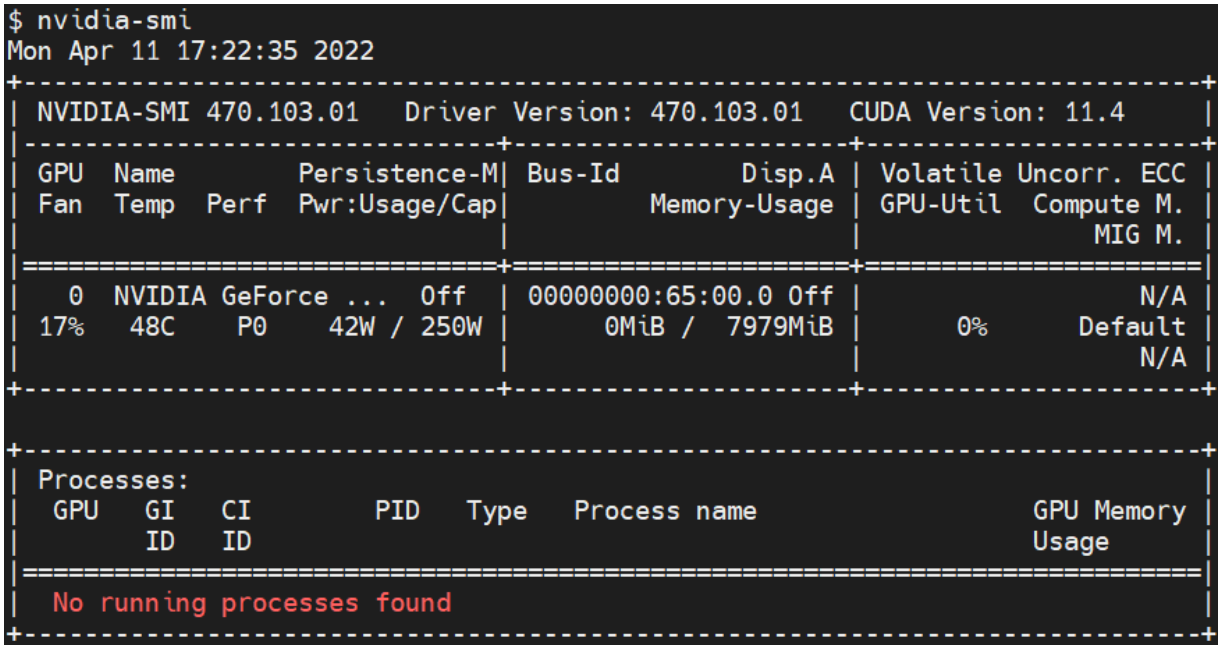
Itay Snir, 208726992, itaysnir@campus.technion.ac.il

Tasks

1. Knowing the system:
    a. Cuda Version:

```
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Wed_Oct_23_19:24:38_PDT_2019
Cuda compilation tools, release 10.2, V10.2.89
```

b. GPU Name:


$ nvidia-smi
Mon Apr 11 17:22:35 2022
NVIDIA-SMI 470.103.01 Driver Version: 470.103.01 CUDA Version: 11.4
GPU Name Persistence-M Bus-Id Disp.A Volatile Uncorr. ECC
Fan Temp Perf Pwr:Usage/Cap Memory-Usage GPU-Util Compute M.
MIG M.
0 NVIDIA GeForce ... Off 00000000:65:00.0 Off N/A
17% 48C P0 42W / 250W 0MiB / 7979MiB 0% Default
N/A
Processes:
GPU GI CI PID Type Process name GPU Memory
ID ID Usage
No running processes found

c. GPU:

```
$ ./deviceQuery
./deviceQuery Starting...

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

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce RTX 2080 SUPER"
  CUDA Driver Version / Runtime Version          11.4 / 10.2
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 7979 MBytes (8366784512 bytes)
  (48) Multiprocessors, ( 64) CUDA Cores/MP:     3072 CUDA Cores
  GPU Max Clock rate:                            1815 MHz (1.81 GHz)
  Memory Clock rate:                             7751 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 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:  1024
  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 3 copy engine(s)
  Run time limit on kernels:                     No
  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 / 101 / 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 = 10.2, NumDevs = 1
```

d. Number of SMs: **48** SMs (multiprocessors), each containing **64** CUDA Cores.

2. In **ex1.cu**

3. **Task Serial Version**

a. Implementation of the kernel (can be found within **ex1.cu**):

```
__global__ void process_image_kernel(uchar *all_in, uchar *all_out, uchar *maps)
{
    __shared__ int hist[COLOR_COUNT * sizeof(int)];
    const int image_offset = IMG_HEIGHT * IMG_WIDTH * blockIdx.x;
    const int maps_offset = COLOR_COUNT * TILE_COUNT * TILE_COUNT * blockIdx.x;

    for (int tile_row = 0 ; tile_row < TILE_COUNT ; tile_row++)
    {
        for (int tile_col = 0 ; tile_col < TILE_COUNT ; tile_col++)
        {
            memset(hist, 0, COLOR_COUNT * sizeof(int));
            __syncthreads();

            get_histogram(hist, all_in + image_offset, tile_row, tile_col);
            __syncthreads();

            prefix_sum(hist, COLOR_COUNT);
            __syncthreads();

            get_maps(hist, maps + maps_offset, tile_row, tile_col);
            __syncthreads();

        }
    }

    interpolate_device(maps + maps_offset, all_in + image_offset, all_out + image_offset);
    __syncthreads();

    return;
}
```

b. The histogram calculation method contains an atomic operation.

atomicAdd is required due to the race that may occur between different threads:

The “increment by one” operation (eg, histogram[color]++) is actually made of three operations: **read** of the previous value from memory, **arithmetic incrementation,** and finally a **write** of the updated value into memory. For example, two threads may race - thread #1 reads an old value but hasn't written the updated value yet. In the meantime, thread #2 does all three operations (to the same memory location). Once thread #1 completes the write operation, the memory location contains an invalid value, that doesn’t takes into account the addition that was made by thread #2. Atomic operations ensure all of the “sub operations” are made all at once, without any interfering thread in the meanwhile.

c. The global memory accesses are coalesced mainly because of the important need to reduce memory access count. Since global memory accesses are very costly, an efficient access

mechanism is mandatory for decent performance. Instead of fetching a single byte per access, 32 Bytes fetching is much more efficient, assuming all the fetched data will be processed soon (that's why the continuous access by the various threads is required).

d. In **ex1.cu**

e. In **ex1.cu**

f. In **ex1.cu**

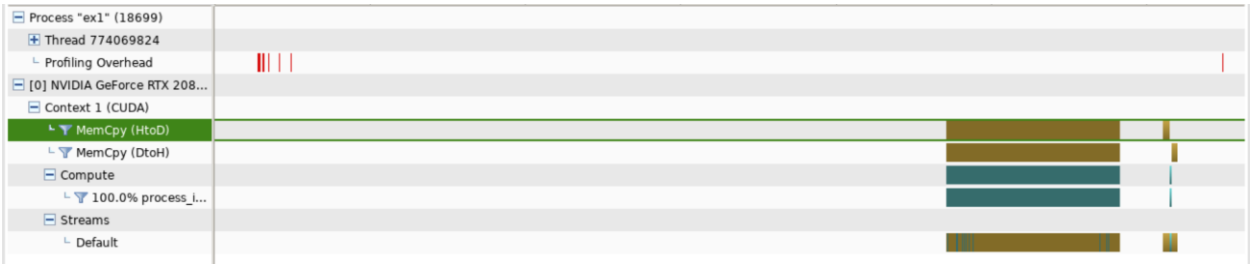
g. For the serial version, we were asked to invoke the kernel with a single thread block.
This single thread block runs **256 threads**. The main reasoning is simplicity, as the maps calculation function (**get_maps()**) requires exactly 256 threads.
We can further increase the amount of threads (up to 1024), and that will indeed improve the performance of the histogram and the prefix sum calculations.

h. We got the following results:

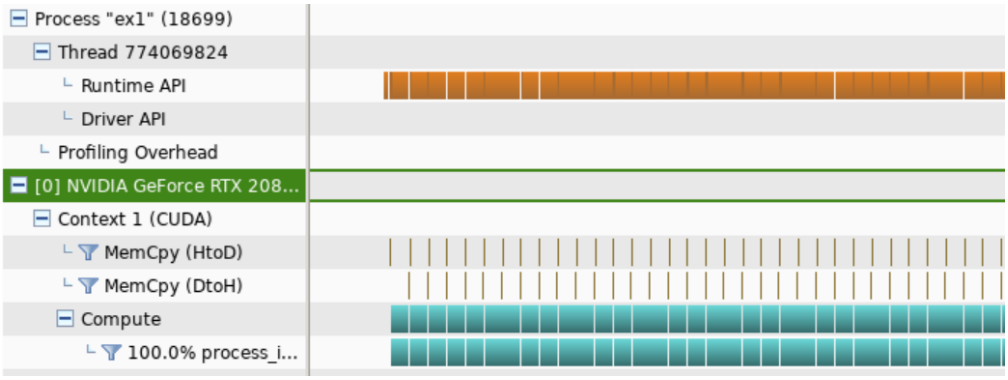
```
$ ./ex1
Number of devices: 1

=== Randomizing images ===
total time 484.036527 [msec]

=== CPU ===
total time 1428.061487 [msec]

=== GPU Task Serial ===
total time 603.628639 [msec]  distance from baseline 0 (should be zero)

=== GPU Bulk ===
total time 89.941910 [msec]  distance from baseline 0 (should be zero)
```

As we can see, the total run time for the GPU serial processing is **603.63 msec**.
Since **N_IMAGES = 1000**, the throughput is about **1657 images / sec.**

i. Execution diagram:

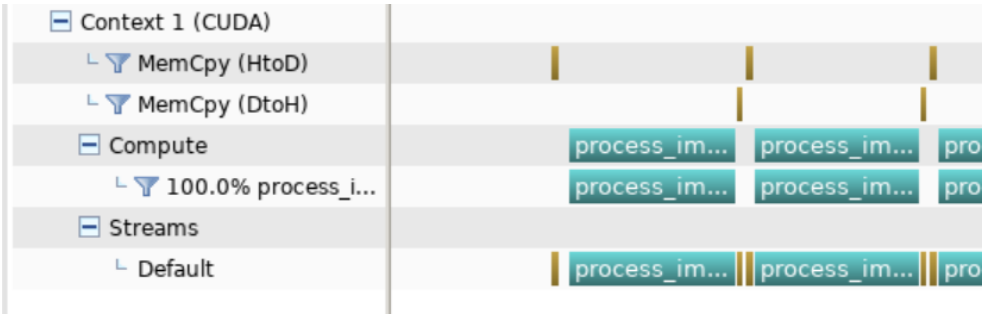


Zoom in …



j. Memcpy from CPU to GPU time(Host to Device):

We will choose the first memcpy from CPU to GPU, following task duration.



Memcpy HtoD [sync]

Start	2.35312 s (2,353,117,207 ns)
End	2.35314 s (2,353,141,112 ns)
Duration	23.905 µs
Size	262.144 kB
Throughput	10.966 GB/s
Stream	Default
Memory Type	

4. **Bulk Synchronous Version**

a. In **ex1.cu**

b. We've modified the kernel presented in (3), so that it can easily support bulk execution for multiple thread blocks. In **ex1.cu**

c. In **ex1.cu**

d. In **ex1.cu**

e. In **ex1.cu**

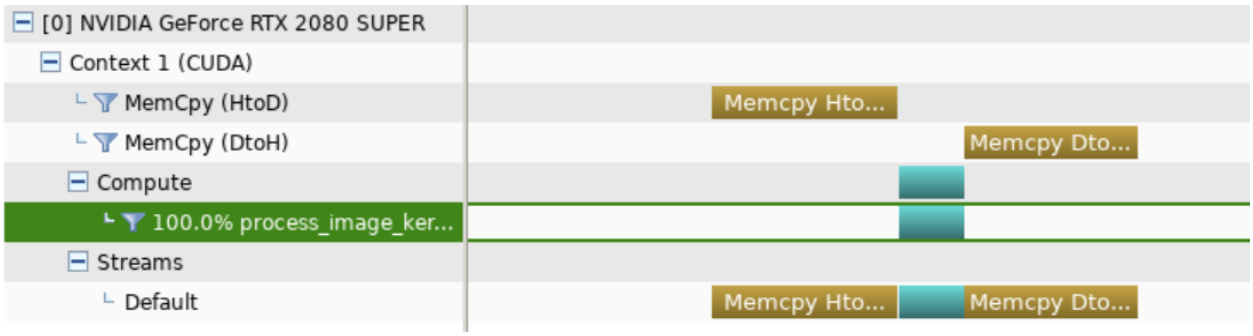
f. Another execution of the results:

```
$ ./ex1
Number of devices: 1

=== Randomizing images ===
total time 482.549735 [msec]

=== CPU ===
total time 1426.530920 [msec]

=== GPU Task Serial ===
total time 602.680784 [msec]  distance from baseline 0 (should be zero)

=== GPU Bulk ===
total time 89.933959 [msec]  distance from baseline 0 (should be zero)
```

As we can see, the execution time for the bulk version is **89.93 msec**.

The resulting speedup is **x6.7!** The bulk version is faster by 6.7 times.

g. Execution diagram:



h. Memcpy from CPU to GPU time(Host to Device):

Memcpy HtoD [sync]

Start	3.05155 s (3,051,545,142 ns)
End	3.0728 s (3,072,803,428 ns)
Duration	21.25829 ms (21,258,286 ns)
Size	262.144 MB
Throughput	12.331 GB/s
Stream	Default
▽ Memory Type	