מאיצים חישוביים ומערכות מואצות
046853

אביב תשע"ט
תרגיל בית 1

שם: יחזקאל עידו **תעודת זהות:** 204397368

שם: זוהר אוהד **תעודת זהות:** 305285694

1. Knowing the System:

1.1 CUDA Version: Cuda compilation tools, release 7.0, V7.0.27

1.2 GPU Name: Persistence-M

1.3 The number of SMs (Multiprocessors): 12

2. Implement device functions:

2.1 Implanted in the hw1.cu file.

2.2 Implanted in the hw1.cu file.

3. Implement a task serial version:

3.1 Implanted in the hw1.cu file.

3.2 *atomicAdd* has been used:

```
atomicAdd(hist_shared + pixelValue, 1);
```

3.3 atomicAdd is required for the correctness of the histogram - the histogram array allocated in shared memory, multiple threads running in the same thread block may trying to update the same pixel bucket in the same time. An atomic operation guarantees no two writers (in our case threads) can access in the same time to the same pixel bucket.

3.4 Coping only one picture per iteration, by changing the source start pointer for copying:

```
int imageStartIndex =  IMG_HEIGHT * IMG_WIDTH * i;
CUDA_CHECK(cudaMemcpy(image_in_device_serial, images_in
imageStartIndex, IMG_HEIGHT * IMG_WIDTH, cudaMemcpyHostToDevice));
```

3.5 The consideration in number of threads:

i. We would like to process the CDF in parallel, so we need at least 256 threads in order to implement the Kogge-Stone algorithm as we seen in class.
ii. We would like to find the minimum in parallel, based on the sum algorithm we seen in class, so we need at least 128 threads.
iii. Hardware limitations: 1024 threads/block.
iv. Executing only 1 block at the time no L2 cache pollution from other threads blocks.
v. We notice that the image size is 256X256 so we can't process 256 pixels at once because transaction to the global memory is 128 Bytes.
vi. We would like to process as much as possible pixels from the image in parallel.
vii. No need to consider load balancing because we invoke 1 kernel for each image (not true for bulk kernel).

Therefore, we decided to choose 1024 threads because we would like to process as much as possible pixels in parallel because we can't avoid 2 transaction per one image row. (for bulk processing we used 256 threads per thread block)

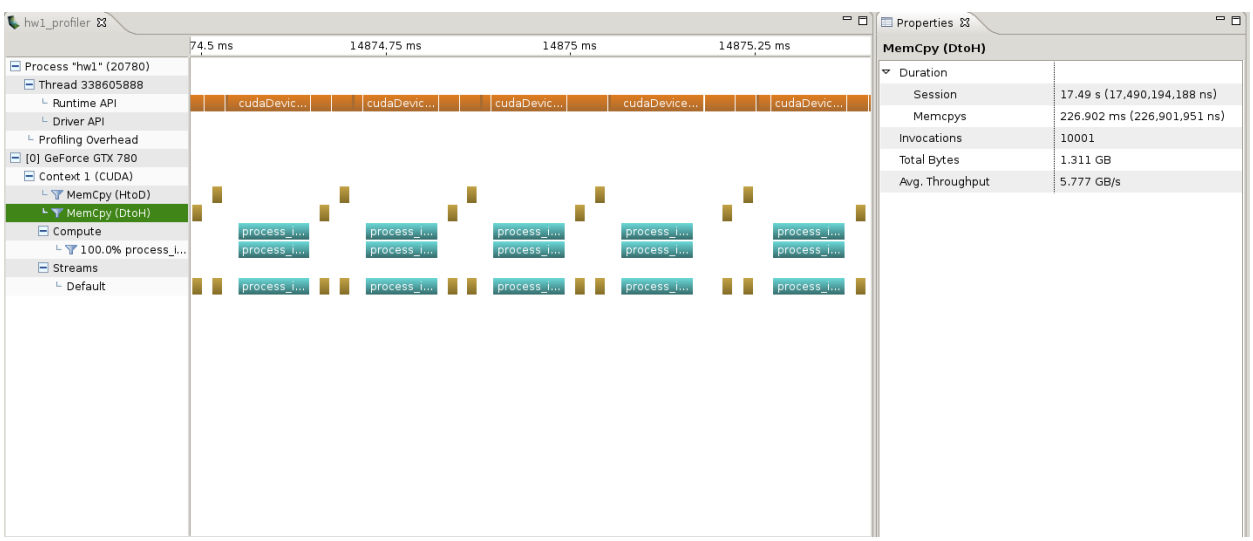
# Threads is 256	`=== GPU Task Serial ===` `total time 3282.309082 [msec]  distance from baseline 0 (should be zero)`
# Threads is 512	`=== GPU Task Serial ===` `total time 2043.091797 [msec]  distance from baseline 0 (should be zero)`
# Threads is 1024	`=== GPU Task Serial ===` `total time 1523.687012 [msec]  distance from baseline 0 (should be zero)`

Examine our choice:

3.6 Total running time (5 runs average) is: 1527.160986 [mSec]

Throughput is: $\frac{10,000[images]}{1527.160986[m\,\mathrm{Sec}]} = 6548.09813\left[\frac{images}{\mathrm{Sec}}\right]$

3.7



3.8 Memory copy from CPU to GPU duration: $13.216[\mu\text{Sec}]$

Memcpy HtoD [sync]	
Start	14.875 s (14,874,860,241 ns)
End	14.875 s (14,874,873,457 ns)
Duration	13.216 µs
Size	65.536 kB
Throughput	4.959 GB/s
Memory Type	
Source	Pinned
Destination	Device

4 Implement a bulk synchronous version:

4.1 Implanted in the hw1.cu file.

4.2 Invoking the kernel with all input images at once:

```
process_image_kernel <<< N_IMAGES, THREADS_PER_BLOCK_BULK >>>
(image_in_device_bulk, image_out_device_bulk);
```

4.3 Memory copy of all input images from CPU to GPU:

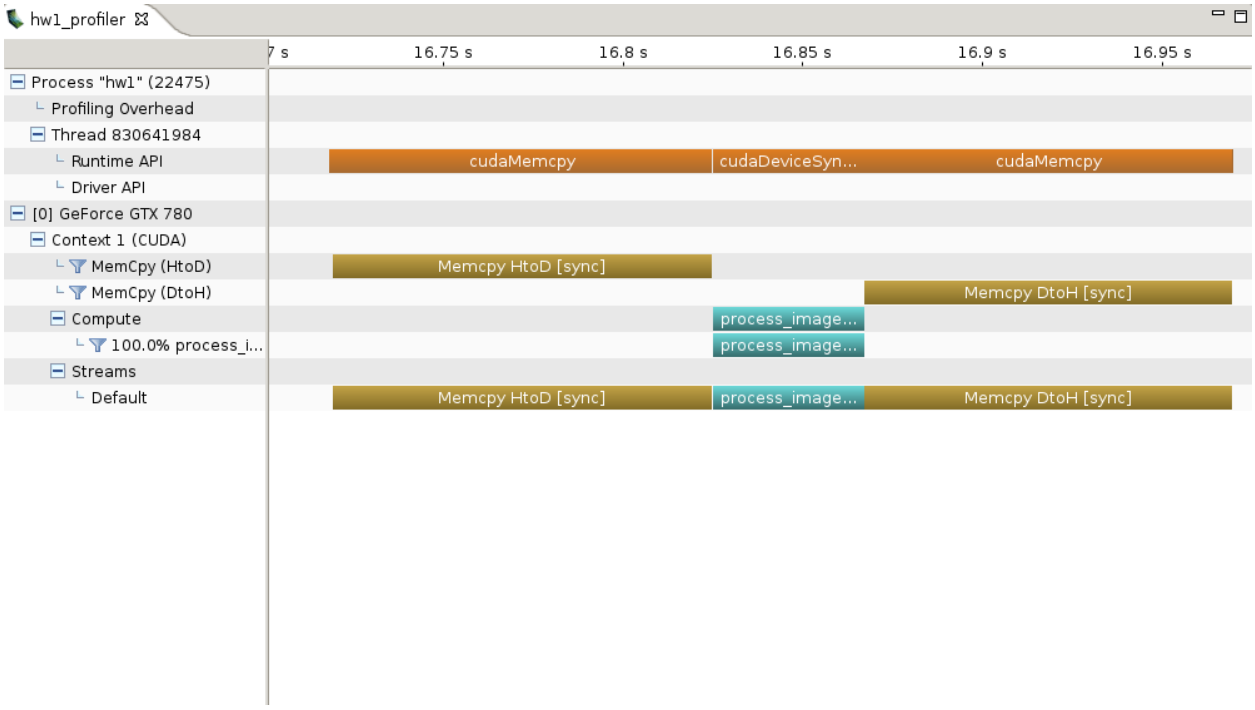
```
CUDA_CHECK(cudaMemcpy(image_in_device_bulk, images_in,
IMG_HEIGHT * IMG_WIDTH * N_IMAGES, cudaMemcpyHostToDevice));
```

4.4 Total running time (5 runs average) is: 250.0182128 [mSec].

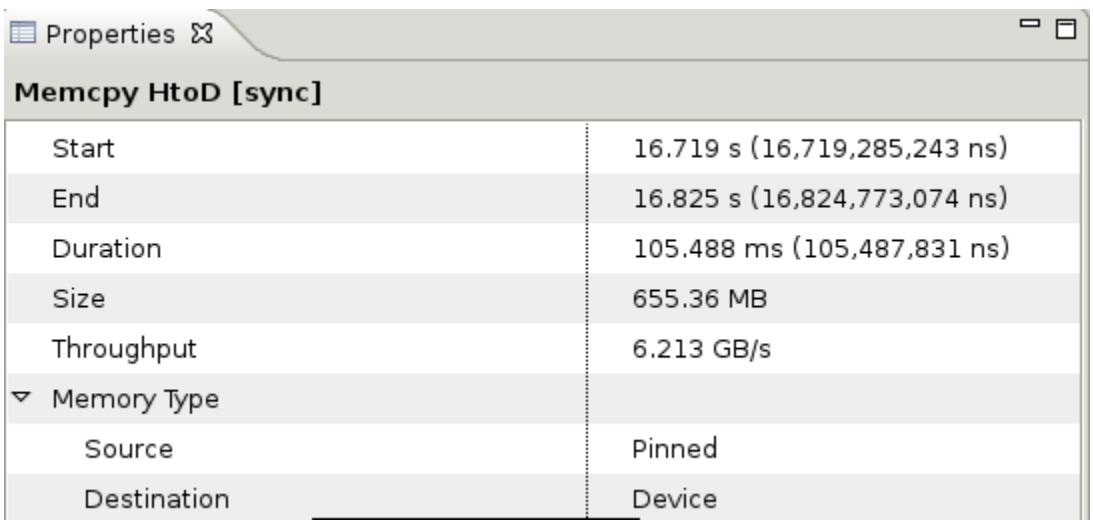
Throughput is: $\frac{10,000[images]}{250.0182128[m\text{Sec}]} = 39,997.08616\left[\frac{images}{\text{Sec}}\right]$ nice 😊.

Speedup Bulk Vs. Serial is: $\frac{t\_old}{t\_new} = \frac{1527.160986}{250.0182128} = 6.11.$

4.5



4.6 Memory copy from CPU to GPU duration: $105.488[mSec]$



Properties

Memcpy HtoD [sync]

Start	16.719 s (16,719,285,243 ns)
End	16.825 s (16,824,773,074 ns)
Duration	105.488 ms (105,487,831 ns)
Size	655.36 MB
Throughput	6.213 GB/s
Memory Type	
Source	Pinned
Destination	Device

For 1 image it takes $13.216[\mu Sec]$ and for 10,000 it takes $105.488[mSec]$ if it is grow linearly it should take $13.216[\mu Sec]\cdot 10,000 = 132.16[mSec]$ so it isn't grow linearly, because the delta is $\sim 27[mSec]$ less. We assume the reason is because the overhead of copy initializing is only happening once when copy all images at once.