**מאיצים חישוביים ומערכות מואצות**

**046853**

**אביב תשע"ט**

**תרגיל בית 1**

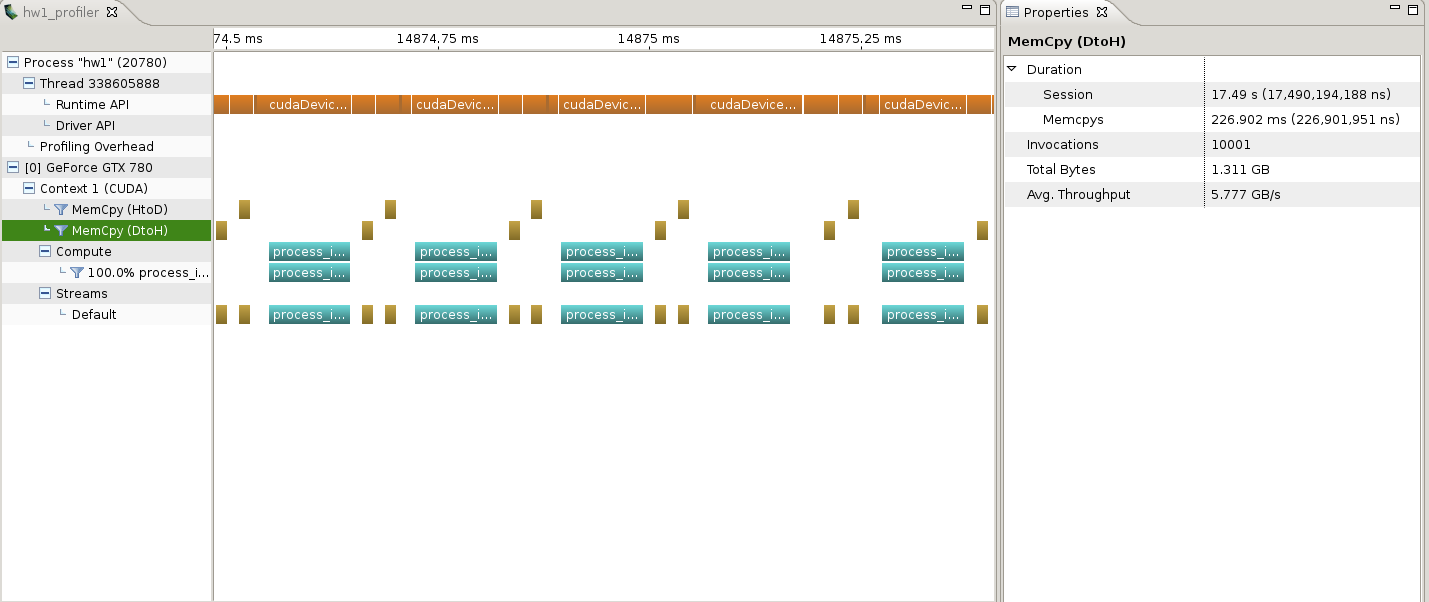
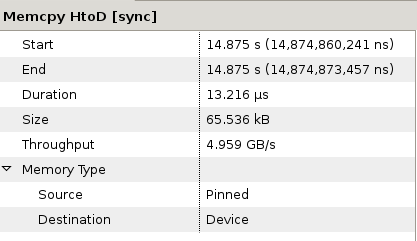
204397368 **:שם:** יחזקאל עידו **תעודת זהות**

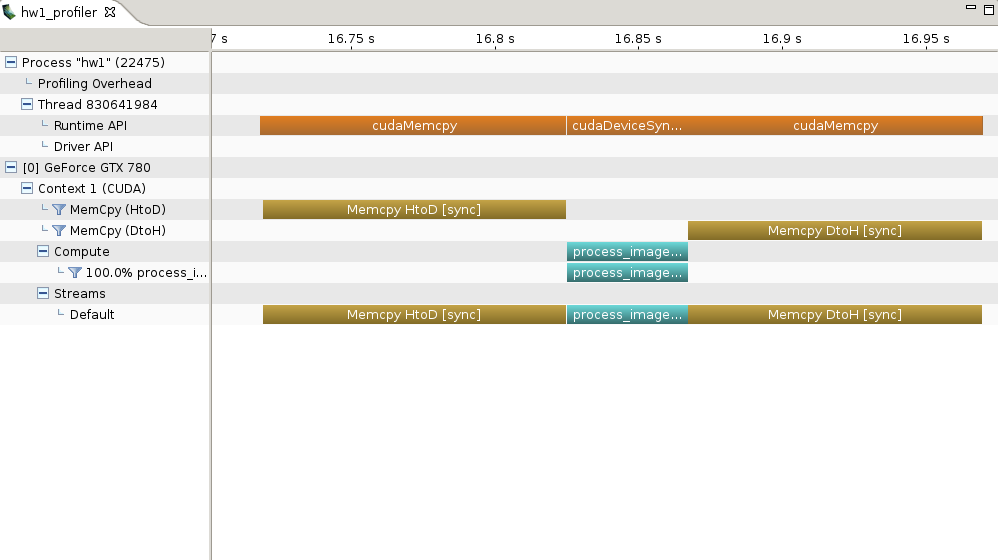
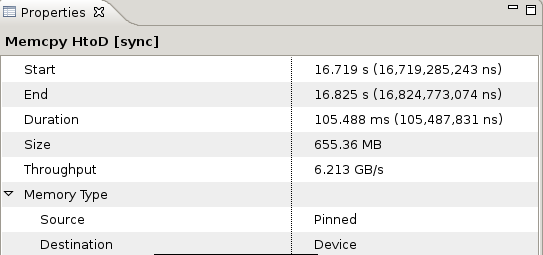
305285694 **:שם:** זוהר אוהד **תעודת זהות**

1. Knowing the System:
   1. CUDA Version: Cuda compilation tools, release 7.0, V7.0.27
   2. GPU Name: Persistence-M
   3. The number of SMs (Multiprocessors): 12
2. Implement device functions:
   1. Implanted in the hw1.cu file.
   2. Implanted in the hw1.cu file.
3. Implement a task serial version:
   1. Implanted in the hw1.cu file.
   2. *atomicAdd* has been used:  
      atomicAdd**(**hist\_shared **+** pixelValue**,** 1**);**
   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.
   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**));**
   5. The consideration in number of threads:
      1. 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.
      2. 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.
      3. Hardware limitations: 1024 threads/block.
      4. Executing only 1 block at the time no L2 cache pollution from other threads blocks.
      5. 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.
      6. We would like to process as much as possible pixels from the image in parallel.
      7. No need to consider load balancing because we invoke 1 kernel for each image (not true for bulk kernel).

|  |  |
| --- | --- |
| # Threads is 256 |  |
| # Threads is 512 |  |
| # Threads is 1024 |  |

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)  
Examine our choice:

* 1. Total running time (5 runs average) is: 1527.160986 [mSec]  
     Throughput is: 
  2. 
  3. Memory copy from CPU to GPU duration:   
     

1. Implement a bulk synchronous version:
   1. Implanted in the hw1.cu file.
   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**);**
   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. Total running time (5 runs average) is: 250.0182128 [mSec].  
      Throughput is:  nice😊.  
      Speedup Bulk Vs. Serial is: .
   5. 
   6. Memory copy from CPU to GPU duration:   
      

For 1 image it takes and for 10,000 it takes  if it is grow linearly it should take  so it isn’t grow linearly, because the delta is less. We assume the reason is because the overhead of copy initializing is only happening once when copy all images at once.