HW1 Report

1. –
   1. Cuda compilation tools, release 9.1, V9.1.85
   2. NVIDIA GeForce GT 750M
   3. # SMs: 2



1. –

2.3. In the computation of the histograms, the threads are performing concurrent writes to the same array (same addresses). This may cause a situation where 2 threads (or more) are trying to increase the same element in the array in the same time. The threads read the same value from the array and then increment it by one. As a result, the value is increased only once. "atomicAdd" prevents it by making sure that the increasing is being done in a sequential order, this means that reading and writing to the element are done atomically. It also makes sure that the update propagates to the other threads.

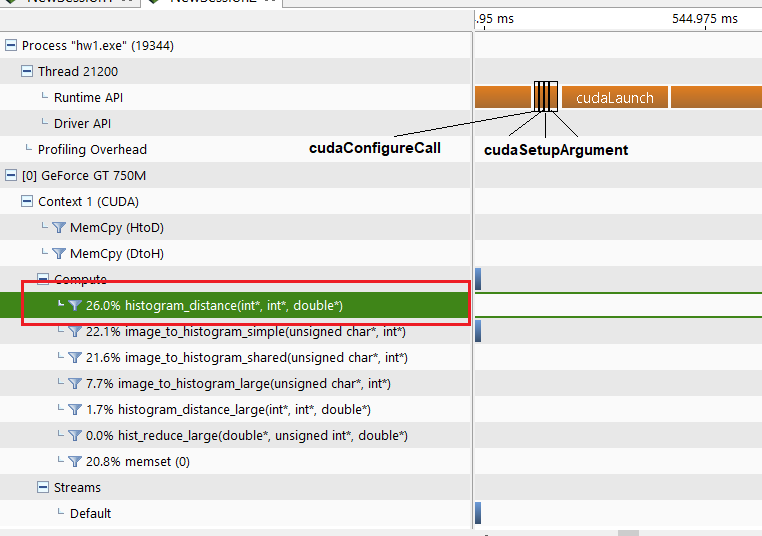
2.10. run time:

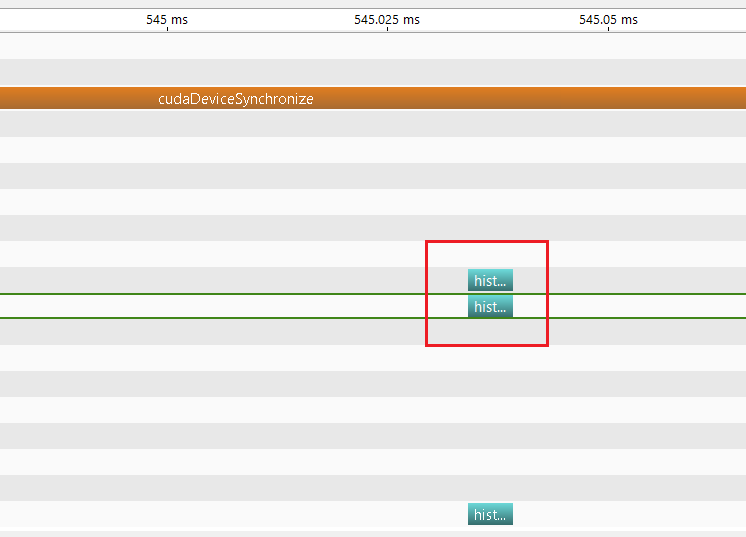


Throughput: 10,000 / 1.646 6075 [image comparisons / sec]

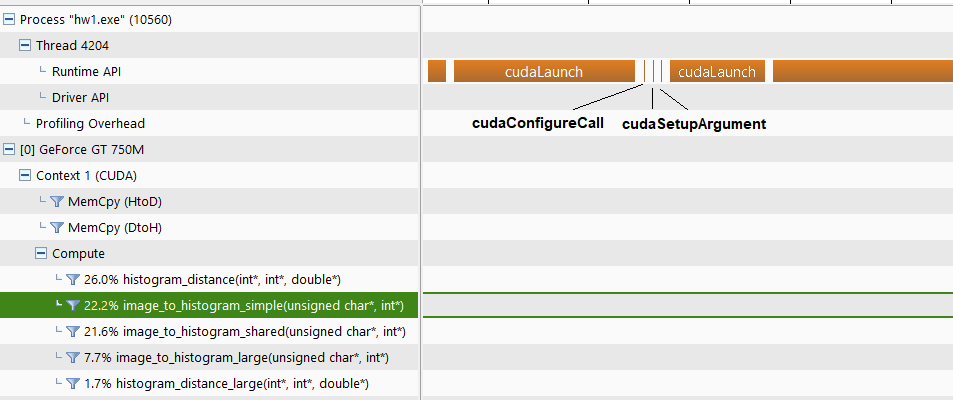
2.11. first, we'll concentrate on the kernel "histogram\_distance" which calculate the distance between two images based on their histograms. We can see that cuda first calls "cudaConfigureCall" which configures a device-launch and takes 427ns. Next cuda is setting up the kernel's arguments by calling "cudaSetupArgument" 3 times (for each of its arguments). This takes 428ns, 428ns and 1283ns (setting up an argument of type double takes ~3 times more time than an integer argument). Next "cudaLaunch" is called which launches the kernel function. Its total execution time (including the kernel runtime) is 11.975.

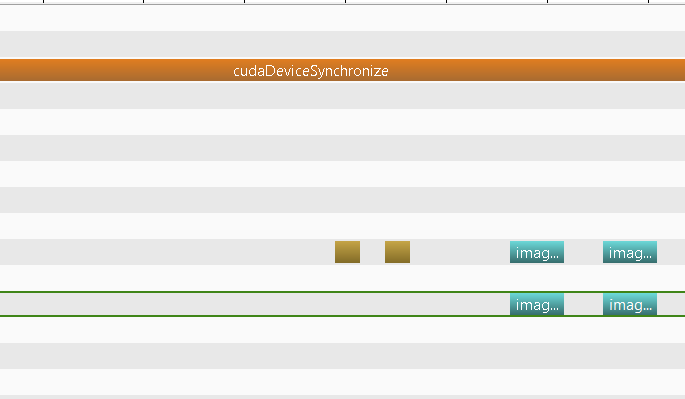
The image was cut in order to be readable.



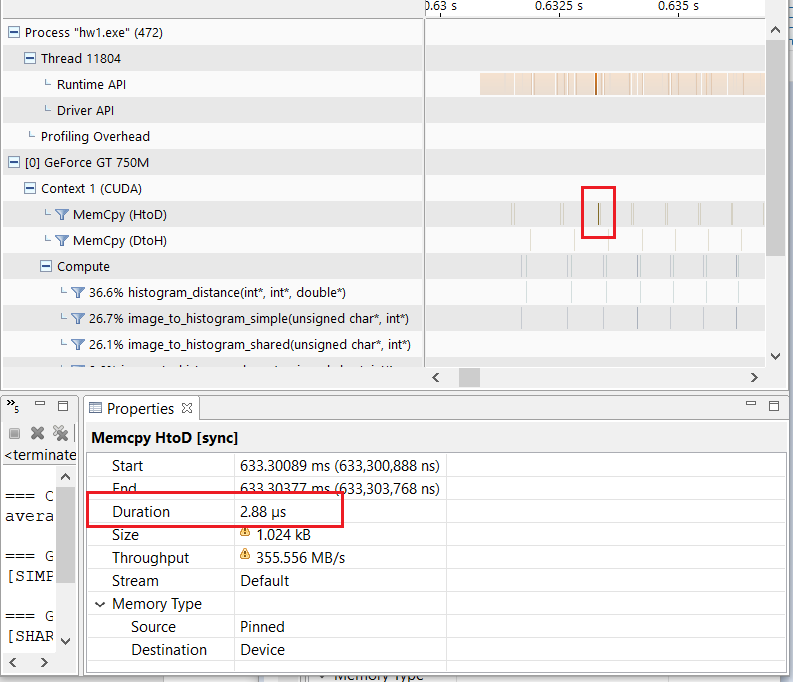


Second call is to "image\_to\_histogram\_simple" which creates the histogram of an image.





2.12.



1. –

3.3. every thread is responsible for computing the binary pattern of a single element in the image array. For this task it has to read 9 elements in the array – the element itself and its 8 neighbors, which means 9 global reads. When using shared memory, each thread performs 1 single global read for the copying its adjusted element to the shared memory and then another 8 shared memory reads for the neighbors. Since accessing the shared memory requires less cycles than accessing the global memory, it improves the total execution time.

3.4. In the previous answer, we described a scenario where shared memory improved latency due to the fact that the same thread had to perform multiple reads from memory. Here, on the contrast, the benefit is gained from having a group of threads accessing the same piece in memory sequentially (due to the use of "atomicAdd"), which is faster using the shared memory.

3.5. surprisingly (or not), the global memory implementation of (2) performs better than the shared memory implementation. As we conclude, global atomic operations perform better than shared in most cases. In particular, when we are talking about images with random pixel value distribution the shared version suffers more from serialization issues.

Total run time:



Speedup: 1.6 / 1.8 0.8

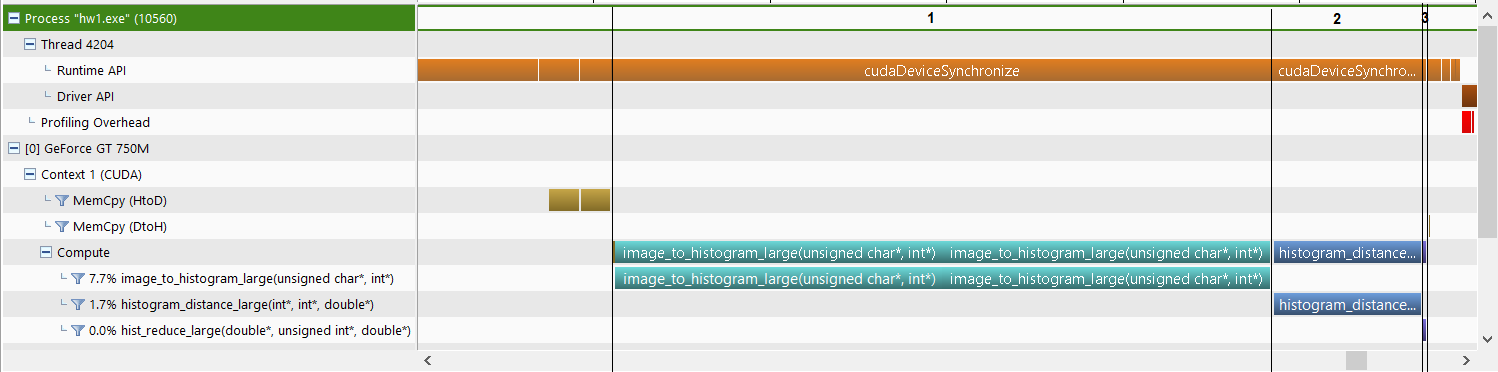
1. –

4.5. execution time:

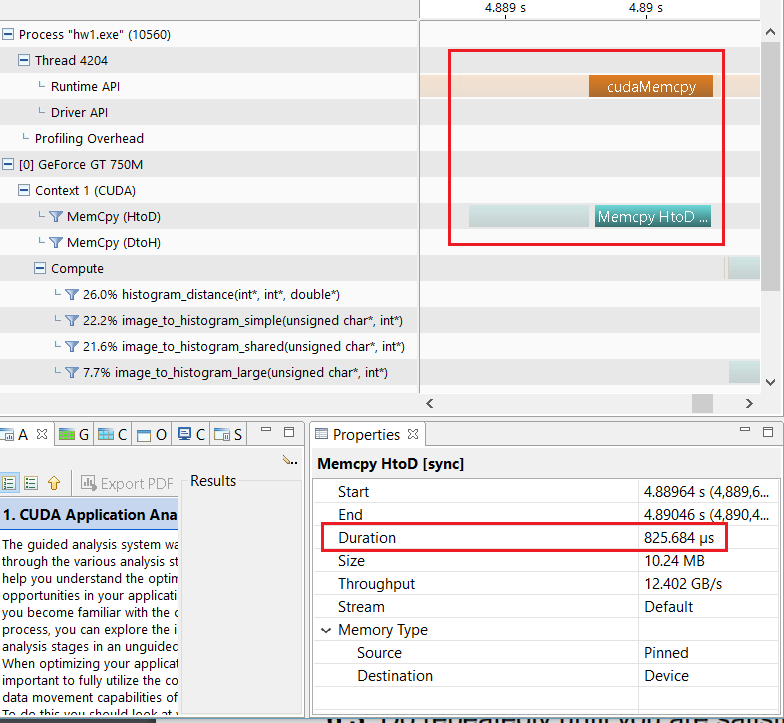


Speedup: 1.8 / 0.006 = 300

4.6.



4.7. 825.685



In (2) we got a time of for copying a single image. This means that for 10,000 images we've expected to get a total time of 10,000\*2.8 = 28,000. The conclusion from this result is that the time has not grown linearly with size.

1. Instead of performing the bounds checks, we can add a "frame" to the image matrix. This means that we are padding the matrix around with zeros. Now when a thread is calculating the binary pattern on an "edge" element, it will sometimes access a padding element but since the edge element is always greater or equal than zero, the final result will be the same.
2. –

6.4. As we inspected earlier, performing atomic operations on shared memory is expensive. First thing we have to do is to rollback the computation of the histogram to the global memory (note that the image is still copied to shared memory). After doing so we improved the latency:



Which is a speedup of 60 / 15 = 4 compared to section (4) version.