Introduction to CUDA and OpenCL
Lab 8 report

30.03.2020
Piotr Litwin
Paweł Skalny

Introduction

During lab we were fully focused on histogram pattern. Our task was to create a complete CUDA application. We started with the single-thread implementation. As our input vector was in the range of zero to the number of bins, we could easily count them by mapping the value to the index of bin. Then we launched a simple CUDA kernel basing on the lecture sample. We analyzed our code and we were trying improve it using shared memory inside blocks and understanding the ordering of read-modify-write operations in an atomic manner.

Results

For the input vector we used 1e9 elements provided by a built-in random number generator with a linear distribution.

The main challange was to improve simple cuda kernel. We observed that the biggest problem was atomic operations when we wanted to write and read many times from the same memory locations in parallel threads. The final results are in *table 1*.

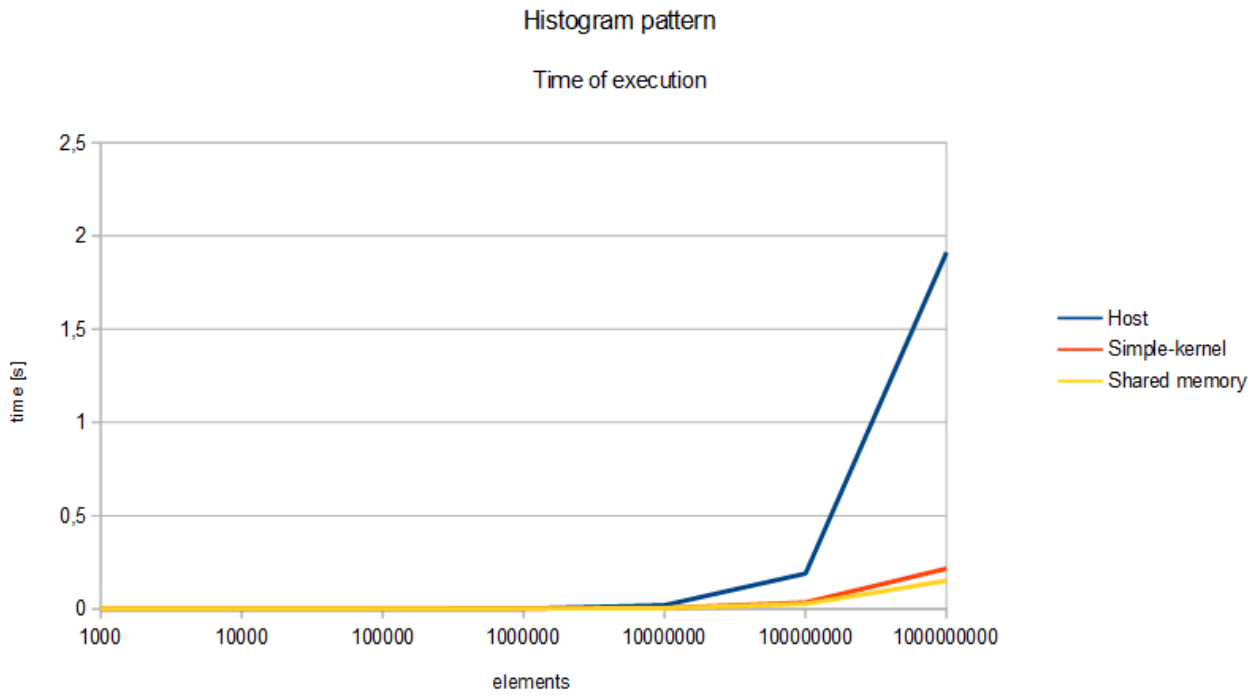
	Single-thread	**Simple-kernel**	**Shared memory**
Time	1,9125192 s	0,2163637 s	0,1513893 s

Table 1. Time of execution for 1e9 elements.

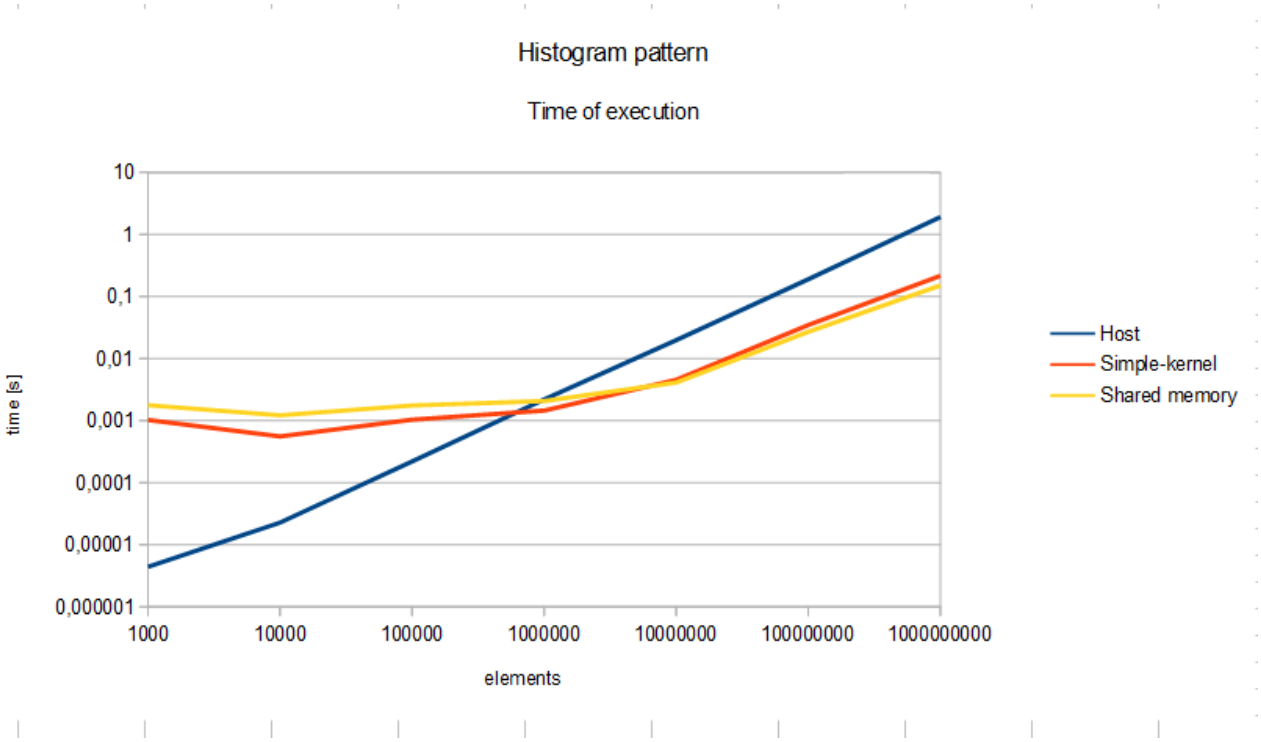
We also checked the program for different input data size (*Table 2*) and we made a plot, that we can see below on *Pic. 1*. All measurements were made 10 times and we obtained the avengare of them.

Elements	Single-thread	Simple-kernel	Shared memory
1000	0,0000040	0,0010240	0,0017741
10000	0,0000228	0,0005576	0,0012193
100000	0,0002202	0,0010367	0,0017549
1000000	0,0021988	0,0014514	0,0020671
10000000	0,0198108	0,0045533	0,0041194
100000000	0,1899415	0,0347751	0,0270357
1000000000	1,9125192	0,2163637	0,1513893

Table 2. Average time of execution for different data size.



Pic. 1. Graph from Table 1.



Pic. 2. Graph from Table 1. in logarithmic scale.

Elements	Speed increase
1000	-42,28%
10000	-54,27%
100000	-40,93%
1000000	-29,79%
10000000	10,53%
100000000	28,63%
1000000000	42,92%

Table 3. Comparison of the speed increase using shared memory instead of the simple kernel

Conclusions

In *Table* 2 we can see that the simple kernel with usage of built-in atomic operations (addition) gives a lot of performance. However, there is another way to get even more performance. The crucial was to avoid atomic operations. We can limit them by using them only within the scope of a single block. To do this, we create a separate histogram for each of the blocks. These histograms are stored in shared memory. We also have to remember that there are three phases: memory initialization, processing using atomic operations within block and moving a local histogram to the global memory. Threads must be synchronized between them and we have to ensure that the local histogram is well mapped in the block. Then each block has its own registered location in the global memory. It causes that we need to allocate nBins * nBlocks space, where nBins means the number of bins in the histogram and nBlocks – the number of blocks in the grid. It is not the end, then we have to join all local histograms spread in the global memory to the final histogram. We have implemented it as a separated kernel (measurements include its) in which we count the content for each bin. One thread for each bin, so each thread performs a number of reads/writes equal to the number of blocks (960 in our case). It is worth noting, that the presented method provides profit above a cartain size of data. We observed it for 1e7 elements.