Report 7

Introduction to CUDA and OpenCL

Histogram Pattern.

Dariusz Biela

Karol Sawicki

1. Introduction

Topic of this laboratory classes was Histogram Pattern. We had to prepare GPU accelerated code that calculates number of values from given numerical interval. When single threaded algorithm for this task is really trivial – the paralleled one requires more attention in writing.

2. Calculations and measurements

To do the task we needed data that could been used in our algorithms. We decided to try two different statistic distributions – uniform, where every value has the same probability of occurrence and normal. The normal distribution histogram should be the shape of the gaussian curve, in that case our results should be easy to verify.

At the start we tried to generate values with the C++ standard random library, but generating 10^9 values was long and inefficient. In that case we decided to implement CUDA specific CURAND functions that could use GPU to fast data generation.

In histogram we wanted 2048 bins for different values. To make task as easy ass possible we generated floating point numbers from 0 to 2048, then – in histogram – we determined values by the *floor* function of each generated number. Normal distribution have has the peek value in the 1024 and standard deviation value of 256.

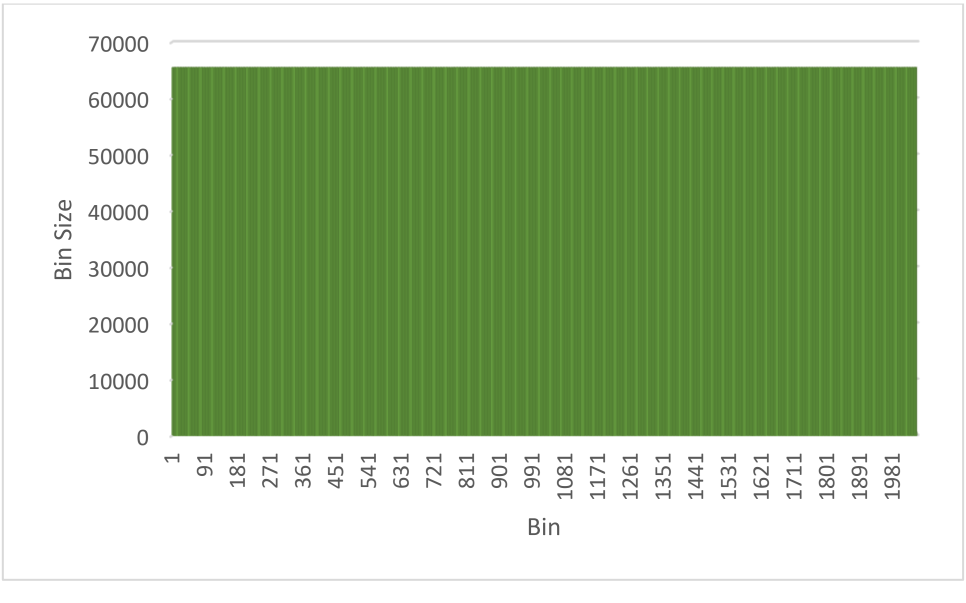

Figure 1 Histogram for uniform distribution generated on CPU.

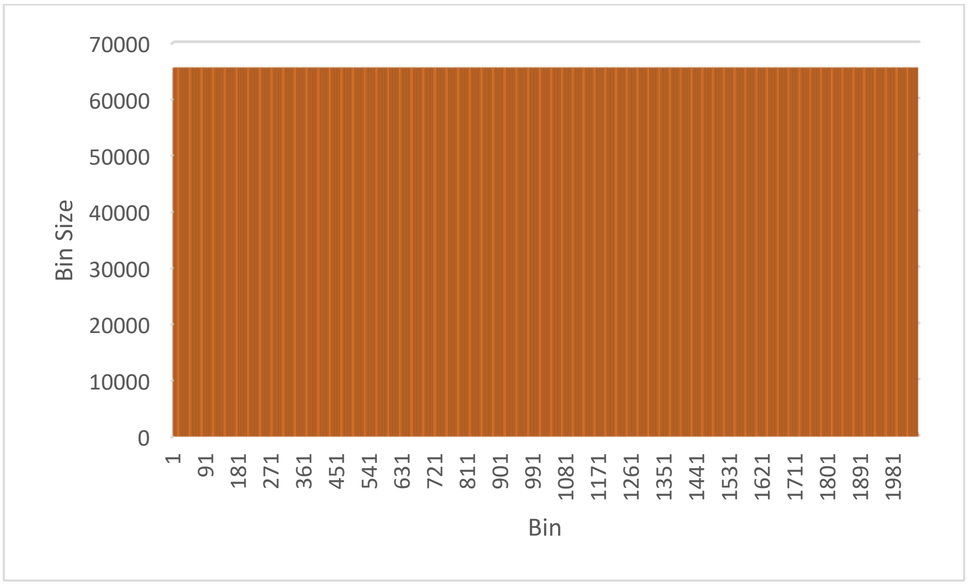

Figure 2 Histogram for uniform distribution generated by GPU.

In the *Figure 1* and *Figure 2* we could see that values are cut. This is the result of our assumptions – our single bin had to have resolution of 2^{16}. The single threaded implementation was simple *if* instruction in the loop, that cut values bigger than bin resolution. The saturation kernel was simple too – it was implemented separate just to avoid divergence in the main histogram kernel, it was solved similar to single threaded implementation, by the simple *if*.

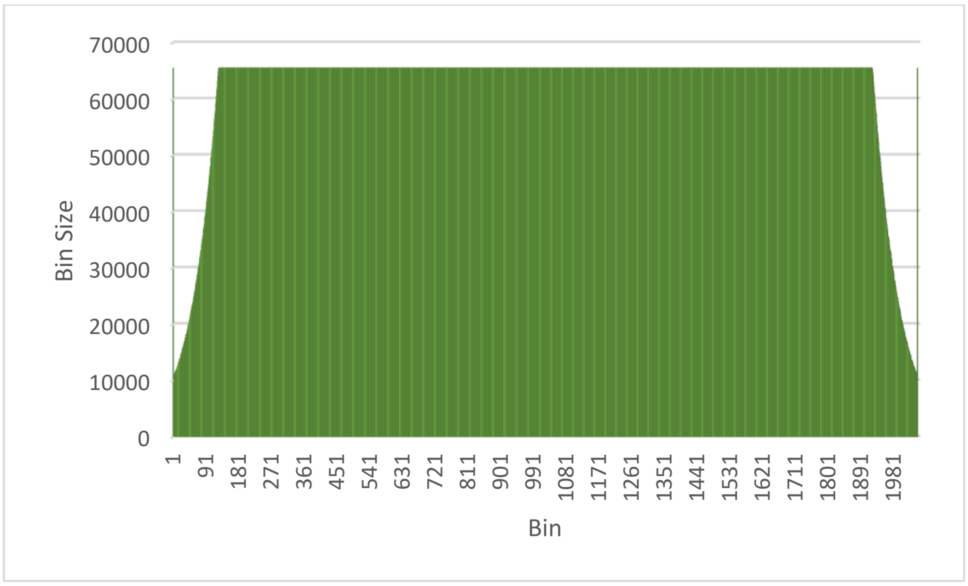

Figure 3 Histogram for normal distribution generated on CPU.

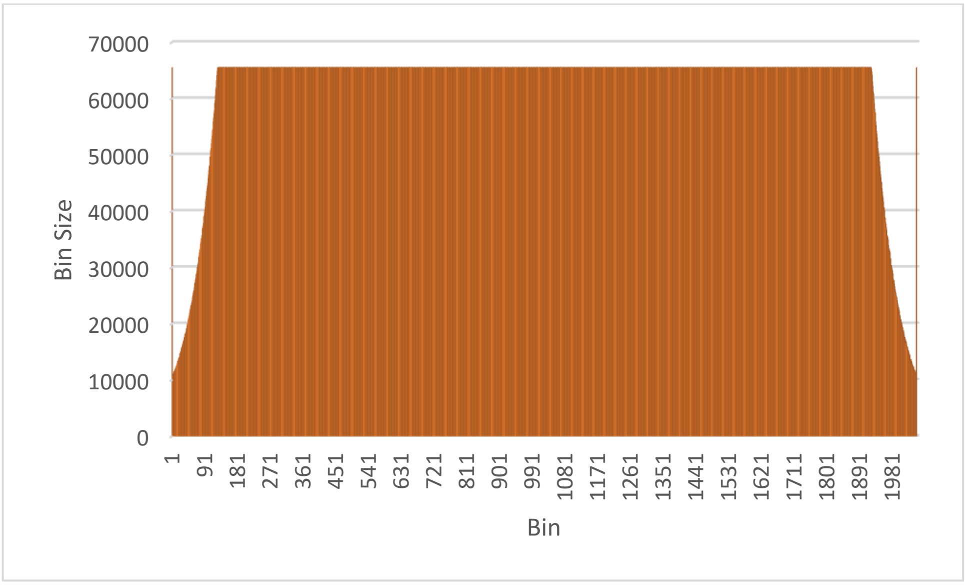

Figure 4 Histogram for normal distribution generated on GPU.

Figure 3 and *Figure 4* shapes are the same. The figures are not particularly gaussian curves, but we could assume that is just effect of small bins resolution, because both of those figures ends are almost identical as the gaussians ends.

At the end, we measured time of single threaded and GPU accelerated execution and prepared one more graph.

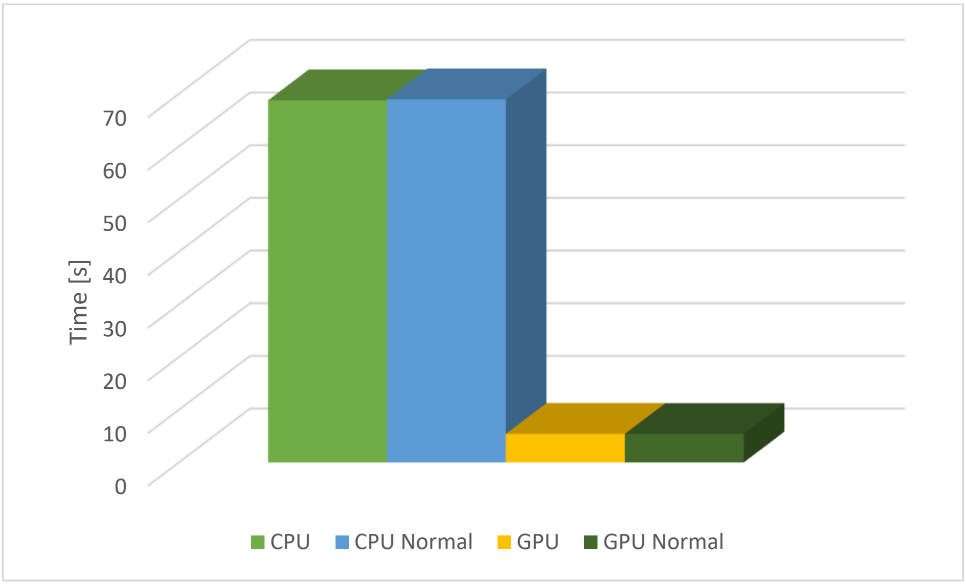

Figure 5 Execution time comparison.

Figure 5 shows that different data distributions does not affect performance, as the time results are pretty much the same.

We could see that GPU accelerated kernel code is more than 10 times faster, than the single threaded implementation, but it could be not that good in every case. Our kernel benefits from usage of GPU shared memory. Firstly – each block allocates temporary shared memory array, that is modified by that block. It is good, because access time for shared memory is so much

smaller, than access time of global GPU memory. In the case of histogram pattern algorithm it is really important, because each thread in block potentially could reference the same data so many times, especially when input data size is so big as 10^9 elements or even more. With shared memory each block references many times only fast memory blocks and the main output data only once, at the end of kernel execution.
Furthermore – we added asynchronous memory prefetching to GPU and used knowledge about multiprocessor count to invoke kernels.

3. Conclusions

Different distributions does not affect performance, it should be not a big surprise, because in every case the same operations are performed. The algorithm is good for any data size, if it only fits in GPU memory and in every case should gave us proper output histograms.

Histogram pattern is not trivial to implement on the GPU, it should use atomic operations, when it needs to modify data, because it is crucial for the output to be correct. If normal operations would be used, the output could be correct in some cases, but it would be determined only by the luck.

Shared memory is important, when many threads shall refer to the same memory and work with it a lot. It is so much faster, but we should remember that it is much smaller than GPU global memory and could not be used in every case.