AGH	Module: **Introduction to CUDA and OpenCL**	Report title: **Report 6**		
	Instructor: **dr hab. inż. Szumlak Tomasz**			
	Name and surname: **Marcin Filipek** **Anca Dărăbuț**	Group: **11**	Date: **11.12.2019**	Grade:

1. overview

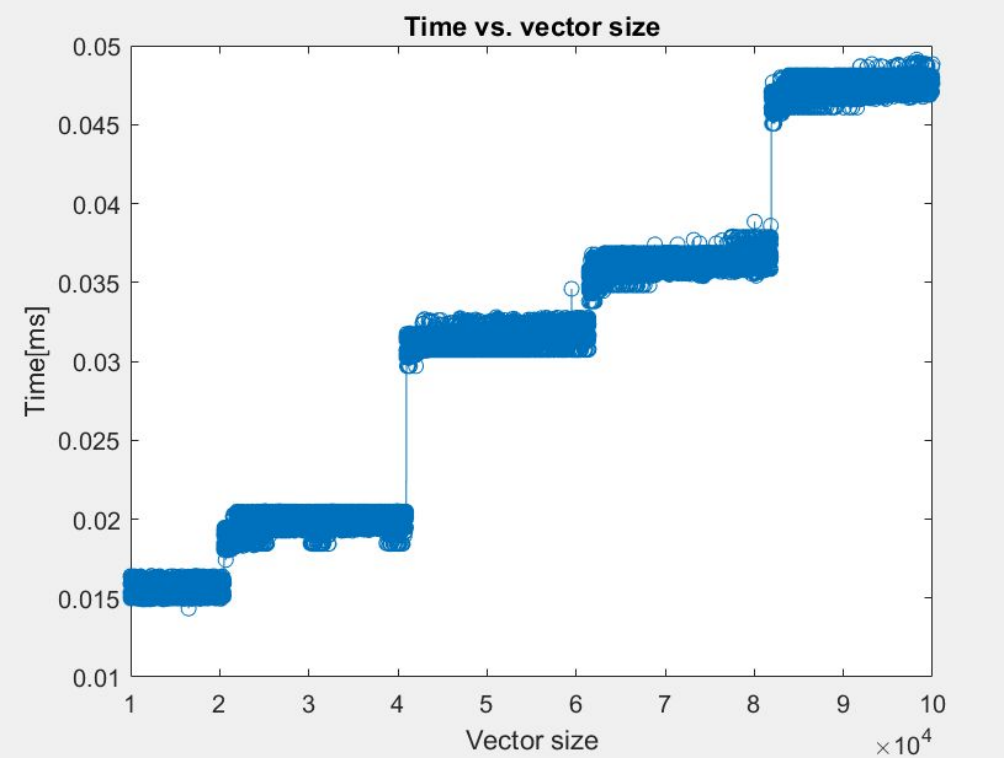
In contrast to the previously written applications, reduction algorithm changes the rules of the game. The basic idea is that the number of parallel threads is gradually decreased until there is only one output location.

The first of our applications using parallel reduction is addition problem. The algorithm is simple: add two values next to each other, then add two of these results and repeat until there is only one number.

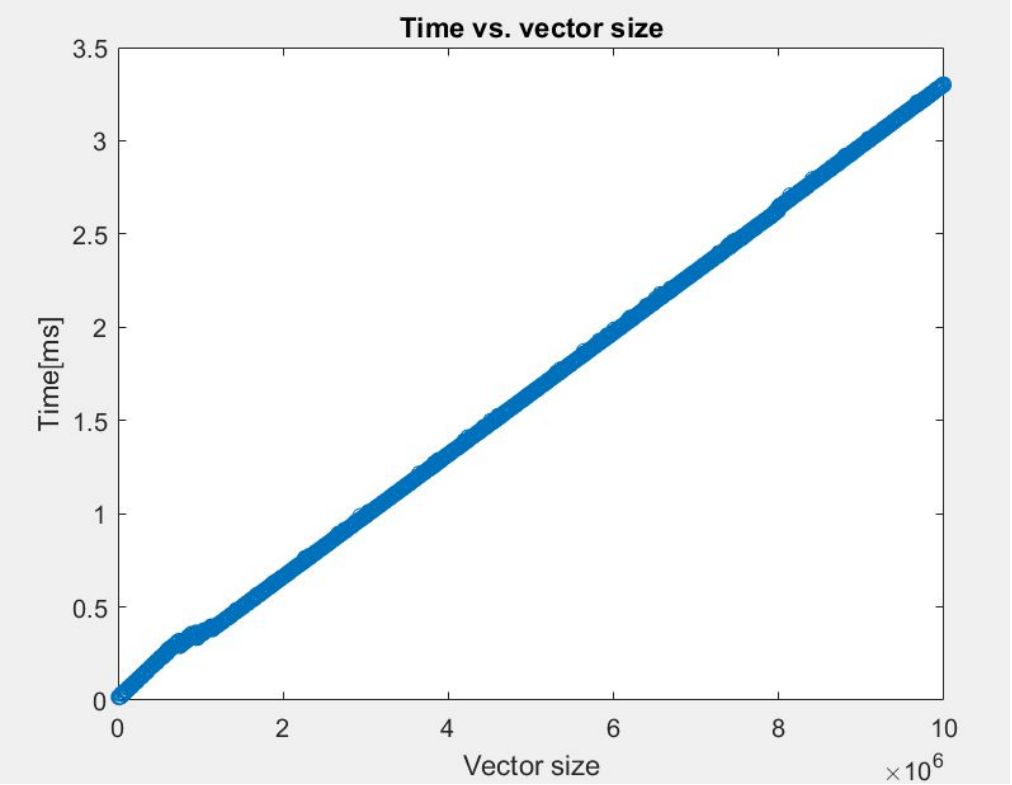
Throughout our trials to write the program, we witnessed some interesting issues. While trying to sum up elements from a vector that had a relatively greater size, the final results were not as expected. On the one hand, the blocks seemed to be competing with each other and, on the other hand, only half of the number of blocks we used performed computations. To be more specific, we set the number of elements to be greater than the number of threads in 2 blocks. By launching it multiple times, the result was either the sum of the partial sums of 2 blocks or even one block. Moreover, the partial sum of almost half of the blocks was equal to 0.
The solution we found was to replace the simple addition (**result[0] += partialSum[0]**) with the function **atomicAdd()**. The advantageous part of using this function is that it computes the sum without interfering with other threads, hence no thread can access the address where the parameters are stored until the operation is complete. In addition, for calling the kernel sum, the number of blocks per grid was halved.

2. different vector sizes

We conducted a series of tests. The first one is time efficiency. We launched our program in a loop which was gradually increasing the number of input elements, measuring time of execution for each size. The results are presented below.



And for much larger numbers of elements:

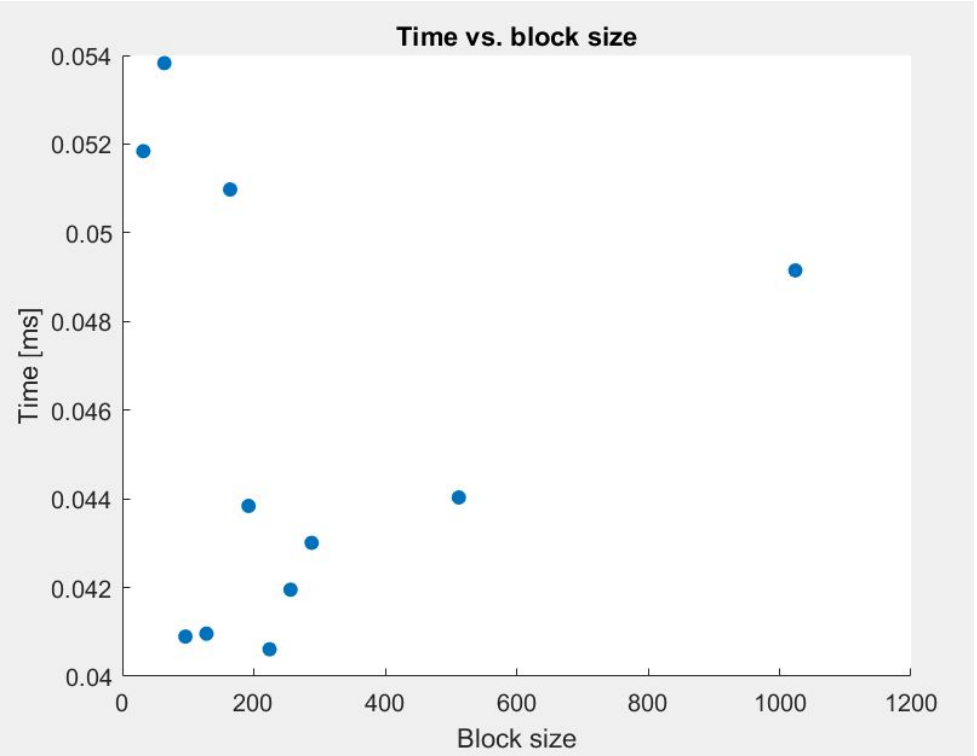


As it can be noticed in the second graph, the time increases in a linear way as the size of the vector increases. The former graph can be perceived as an image taking with the magnifier glas from the latter one, thus, even though the tendency is seen as linear, it is rather formed from so called “steps”.

3. different block sizes

The next test we did was finding the optimal number of threads per block, as we did it in every program we wrote. The number of elements was unchanged and set to 100000. There is a table presenting our results.

Number of threads	Time [ms]
32	0.051840
64	0.053824
96	0.040896
128	0.040960
164	0.050976
192	0.043840
224	0.040608
256	0.041952
288	0.043008
512	0.044032
1024	0.049152



The conclusion is that the optimal number of threads per block is 224, but it is optimal only for a certain number of elements. If we need to change it, the results would be different and this test should be conducted again.