AGH	Module: **Introduction to CUDA and OpenCL**	Report title: **Report 1**		
	Instructor: **dr hab. inż. Szumlak Tomasz**			
	Name and surname: **Marcin Filipek** **Anca Dărăbuț**	Group: **11**	Date: **23.10.2019**	Grade:

1. deviceQuery

During our first lab, we have been introduced in basic CUDA examples. In the beginning, we loaded an example which queries the device and shows CUDA-capable devices and their specifications.
For example, we found lines which tell us about the maximum grid dimension size, maximum number of threads per block and resources that the GPU has.

We put selected lines from output file below:
Total amount of constant memory: *65536 bytes*
Total amount of shared memory per block: *49152 bytes*
Maximum Texture Dimension Size (x,y,z) *1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)*
Maximum number of threads per block: *1024*

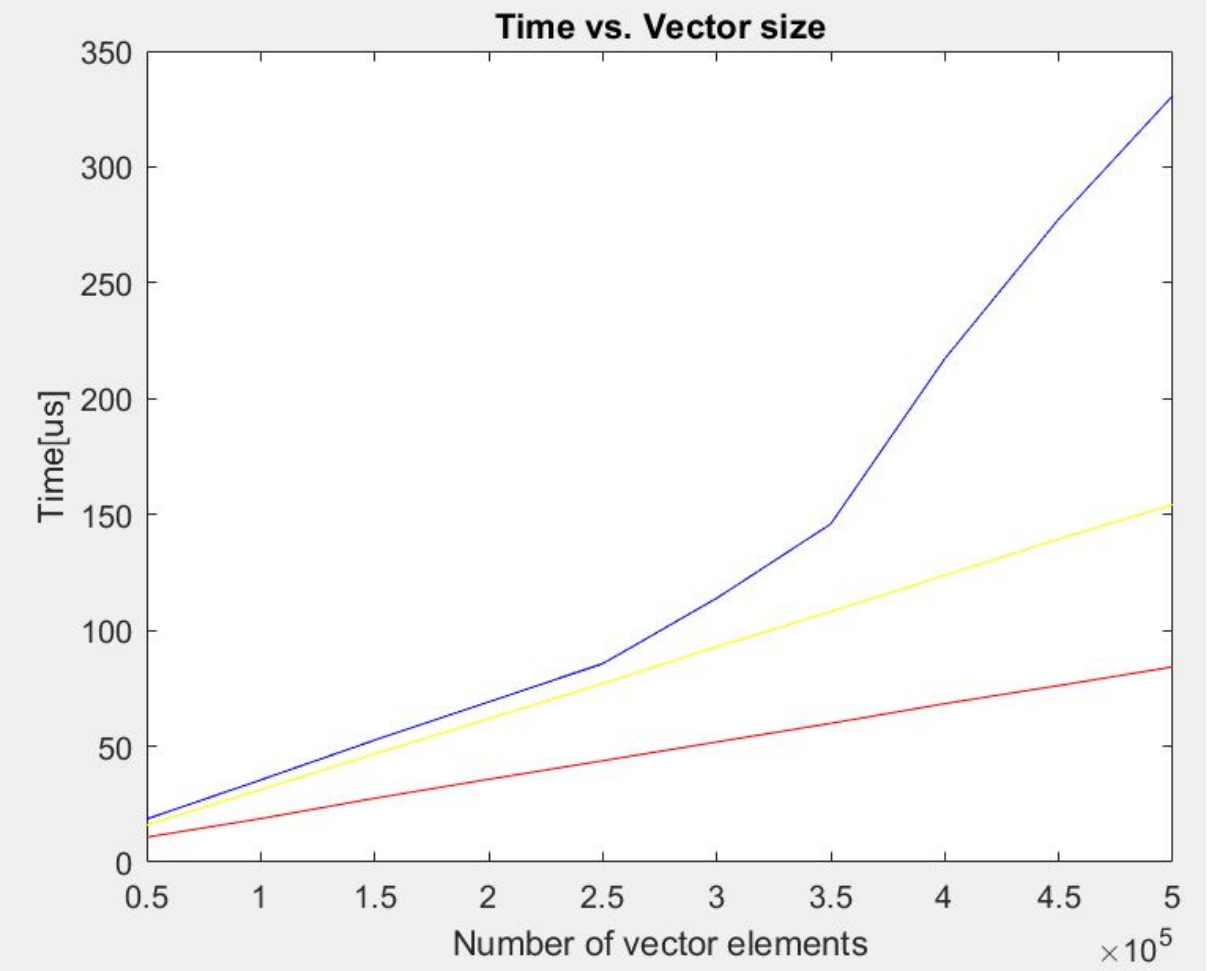
After a short while, we came to conclusion that it gives essential information about the possibilities of our device (we have only one) and using them allows us to optimise the code. Namely, if we write CUDA program on another device, it is necessary to query the device we have and fit the code to its parameters, in order to get correctly working and optimised application.

2. VectorAdd: Time profiling

Next, we got acquainted with CUDA environment. We launched Nsight IDE for the first time and created our first project using sample provided by Nvidia. It was the same example as on the lecture. We compiled and launched it to check if it works.

Then we analysed the code and found the basic structure of CUDA programming model.

Next we made the profiling. It measures execution time and prints results. By manipulating number of threads, blocks and repeating the profiling we were trying to optimize the code. We also were changing vectors size and checking how it affect on time. We put our results below:



The graph reflects the behaviour of the executions time when the size of the vectors is changed, starting from 50k to 500k elements, with a pace of 50k. The average values of the following execution times were represented using the following colours:

CUDA memcpy HtoD - blue
CUDA memcpy DtoH - yellow
vectorAdd - red

It can be seen that the behaviour is linear for copying the data from device to host and performing the vector addition, whereas the profiling of copying data from host to device is rather like a square polynomial.

However, if it were to compare the times, copying the memory from the host to the device takes more time than the other two. The least time consuming action is the vector addition.

```
GPU activities:    Time(%)      Time     Calls       Avg       Min       Max  Name
-------------------------------------------------------------------------------------------
float              57.12%  70.336us         2  35.168us  35.168us  35.168us  [CUDA memcpy HtoD]
                   25.42%  31.296us         1  31.296us  31.296us  31.296us  [CUDA memcpy DtoH]
                   17.46%  21.504us         1  21.504us  21.504us  21.504us  vectorAdd
-------------------------------------------------------------------------------------------
double             62.67%  138.05us         2  69.024us  68.993us  69.056us  [CUDA memcpy HtoD]
                   27.80%  61.249us         1  61.249us  61.249us  61.249us  [CUDA memcpy DtoH]
                    9.53%  20.992us         1  20.992us  20.992us  20.992us  vectorAdd
-------------------------------------------------------------------------------------------
long long          62.28%  138.59us         2  69.296us  69.024us  69.568us  [CUDA memcpy HtoD]
                   28.26%  62.881us         1  62.881us  62.881us  62.881us  [CUDA memcpy DtoH]
                    9.46%  21.056us         1  21.056us  21.056us  21.056us  vectorAdd
-------------------------------------------------------------------------------------------
int                57.55%  71.492us         2  35.746us  35.138us  36.354us  [CUDA memcpy HtoD]
                   25.17%  31.266us         1  31.266us  31.266us  31.266us  [CUDA memcpy DtoH]
                   17.29%  21.474us         1  21.474us  21.474us  21.474us  vectorAdd
-------------------------------------------------------------------------------------------
long               62.27%  137.25us         2  68.624us  68.480us  68.769us  [CUDA memcpy HtoD]
                   27.26%  60.096us         1  60.096us  60.096us  60.096us  [CUDA memcpy DtoH]
                   10.47%  23.072us         1  23.072us  23.072us  23.072us  vectorAdd
-------------------------------------------------------------------------------------------
short(error)       49.03%  37.088us         2  18.544us  18.496us  18.592us  [CUDA memcpy HtoD]
                   21.57%  16.320us         1  16.320us  16.320us  16.320us  [CUDA memcpy DtoH]
                   29.40%  22.240us         1  22.240us  22.240us  22.240us  vectorAdd
with short result verification failed
```

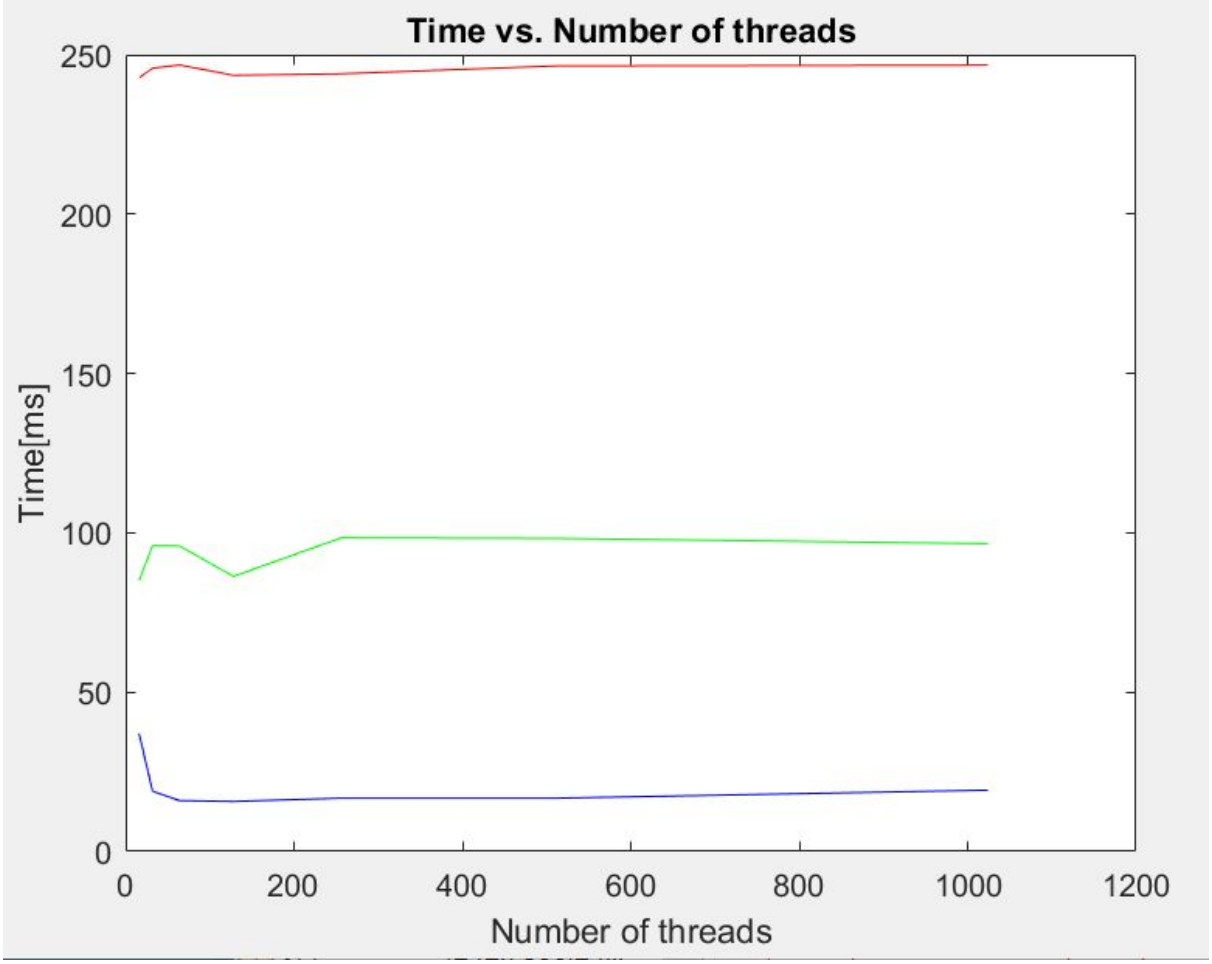
If the only change that intervenes in the program is related to the data type labels, the profiling is linear.

If thorough analysis was to be made on the tendency of execution times distribution, long long type is the most time consuming in terms of copying the data from host to device and vice versa, but long data type requires more time in the execution of the vector addition. The fastest times can be seen for short type, but, with this data type label, the verification failed.

```
* * * Test for 100000 blocks and 1 thread/block * * *
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   83.46%  515.04us         1  515.04us  515.04us  515.04us  vectorAdd
                   11.48%  70.817us         2  35.408us  35.264us  35.553us  [CUDA memcpy HtoD]
                    5.07%  31.264us         1  31.264us  31.264us  31.264us  [CUDA memcpy DtoH]

* * * Test for 10000 blocks and 10 threads/block * * *
 GPU activities:   45.10%  70.848us         2  35.424us  35.040us  35.808us  [CUDA memcpy HtoD]
                   35.00%  54.977us         1  54.977us  54.977us  54.977us  vectorAdd
                   19.90%  31.264us         1  31.264us  31.264us  31.264us  [CUDA memcpy DtoH]

* * * Test for 1000 blocks and 100 threads/block * * *
 GPU activities:   56.52%  71.328us         2  35.664us  35.168us  36.160us  [CUDA memcpy HtoD]
                   24.77%  31.264us         1  31.264us  31.264us  31.264us  [CUDA memcpy DtoH]
                   18.71%  23.616us         1  23.616us  23.616us  23.616us  vectorAdd

* * * Test for 100 blocks and 1000 threads/block * * *
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   57.39%  71.296us         2  35.648us  35.456us  35.840us  [CUDA memcpy HtoD]
                   25.19%  31.296us         1  31.296us  31.296us  31.296us  [CUDA memcpy DtoH]
                   17.41%  21.633us         1  21.633us  21.633us  21.633us  vectorAdd

* * * Test for 10 blocks and 10000 threads/block * * *
Failed to launch vectorAdd kernel (error code invalid configuration argument)!
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  70.337us         2  35.168us  35.072us  35.265us  [CUDA memcpy HtoD]

* * * Test for 1 block and 100000 threads/block * * *
the same result as above
Failed to launch vectorAdd kernel (error code invalid configuration argument)!
```

When changing the number of threads and blocks per threads, we started from a great number of blocks (100000) and just one thread per block, then decreasing the former one and increasing the latter one ten times each.

For the beginning, the vectorAdd function seems to take the most of the execution time. By making the changes, its time decreases, until it is not executed (for the case of 10 blocks and 10000 threads per block). When there is only one block and 100000 threads per block, the launching of vectorAdd kernel failed to launch. We know from deviceQuery that on this particular device we should not put more than 1024 threads in one block, that is why we had got errors in the last two tries.



In the fourth case, the number of threads are modified from 16 to 2048, by powers of 2. The following colour code was used to profile the execution times:

CUDA memcpy DtoH - red
CUDA memcpy HtoD - green
vectorAdd - blue

The execution times seem to be influenced only if a small number of threads is implied. Overall, the time is not affected by the total number of threads so much. If it were to compare the execution times for the three actions, the most time consuming action remains CUDA memcpy device to host, while the vector addition is the least.

3. grid_debug

At last, we took a closer look at processing grid structure.We printed how the grid looks like from the CPU point of view. We also implemented a new kernel in vectorAdd to check the same from GPU.

In the output we got these 2 lines:

grid.x 47 grid.y 1 grid.z 1
block.x 32 block.y 1 block.z 1

This is grid seen from the host side. It means that the CPU can distinguish only the number of blocks and threads if they are organised in 1D, 2D or 3D arrays. These particular lines tells us that the grid we created has 1D 1D build (it means that we have 1D blocks array and 1D threads array within a block) and consist of 47 blocks with 32 threads per block, which gives us 1504 threads in total.

Below are several lines printed by our kernel function:
threadIdx:(8, 0, 0) blockIdx:(24, 0, 0) blockDim:(32, 1, 1) gridDim:(47, 1, 1)
threadIdx:(9, 0, 0) blockIdx:(24, 0, 0) blockDim:(32, 1, 1) gridDim:(47, 1, 1)
threadIdx:(10, 0, 0) blockIdx:(24, 0, 0) blockDim:(32, 1, 1) gridDim:(47, 1, 1)

We got more detailed information about the grid. There were exactly 1504 similar lines, one for every thread. They provide information about thread ID in block (*threadIdx*), block ID in the grid (*blockIdx*), the size of a block, which is array of threads (*blockDim*) and finally blocks array in the grid (*gridDim*). It is easy to assume that GPU has wider view of the processing grid, while the CPU keeps only basic information.