1.Understanding the deviceQuerry

The laboratory classes began with executing the `deviceQuerry` program. It's main purpose is to detect any devices with CUDA installed in the system and display their specifications.

Firstly program checks the number of supported GPU's via `cudaGetDeviceCout(int n)`. If multiple are found, information about each of them will be displayed in sequence by using the *for* loop. Most of the data is included in `cudaDeviceProp` structure which consists of various components, such as: name of the device, the total amount of the global memory, clock frequency, CUDA cores/MP count, warp size, etc.

In order to fetch that data, first one of the GPU's must be chosen with `cudaSetDevice (int n)`, and then function `cudaGetDeviceProperties (cudaDeviceProp *prop ,int n)` returns in `*prop` the properties of the device indicated by the value of `n`. Since the shape of `cudaDeviceProp` structure was slightly different in older versions of CUDA, in order to display size of the L2 cache, memory clock rate and bus width we are using template with a function called `cudaDeviceGetAttribute (int* value, cudaDeviceAttr attr, int device)` while using appropriate attribute for each of them. Both, the version of the installed driver, as well as runtime is not stored in the mentioned structure and needs to be acquired via the `cudaDriverGetVersion(int *driverVersion)` and `cudaRuntimeGetVersion(int *runtimeVersion)`.

Afterwards all the data from `cudaDeviceProp` structure is displayed using `printf` function and, where it is needed, also converted into more understandable form. With 2 or more GPU's present, program also checks RDMA and P2P support. Finally, CUDA driver name and version are shown as well as runtime version and device count.

2. Time Performance Study - vectorAdd

2.1 Relationship between performance and the size of the vector.

In this section we are testing the performance of the GPU, depending on the size of the vector we want to calculate (from 10 to 10^8 elements), and the data type filling given vector (double, float, integer).

We conducted 3 tests on each variation (with grid of 4 blocks of 256 threads) to confirm the outcome, and get rid of major fluctuations. The results of the tests are placed in the table below.

Table 1:Execution time of each process in [µs]. Data type: double.

Number of elements	10^1	10^3	10^5	10^7	10^8
memcpy DtoH	0.896	1.248	61.857	34705	472720
memcpy HtoD	0.88	1.408	69.104	13004	161150
vectorAdd	3.232	3.008	20.32	1744	15496
cudaMalloc	110760	118380	112590	121890	113440
cudaMemcpy	35.083	38.412	576.03	21209	270750
cudaFree	72.658	68.375	120.48	1775.6	5808

Table 2:Execution time of each process in [µs]. Data type: float.

Number of elements	10^1	10^3	10^5	10^7	10^8
memcpy DtoH	0.896	1.12	31.232	25671	233540
memcpy HtoD	0.88	1.072	35.344	9990.7	81101
vectorAdd	3.104	2.656	18.592	1617.1	16148
cudaMalloc	126560	124790	123380	107040	130060
cudaMemcpy	34.269	35.828	307.81	16321	138100
cudaFree	73.496	81.016	80.783	1093	3857

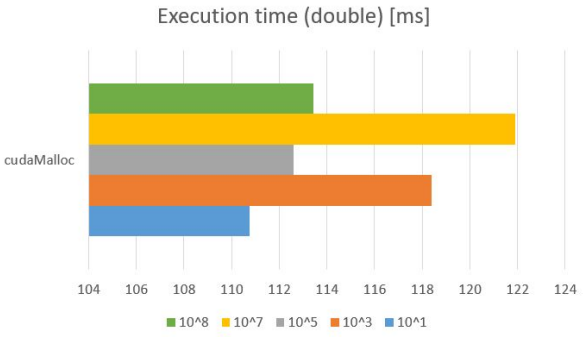
Table 3:Execution time of each process in [µs]. Data type: integer.

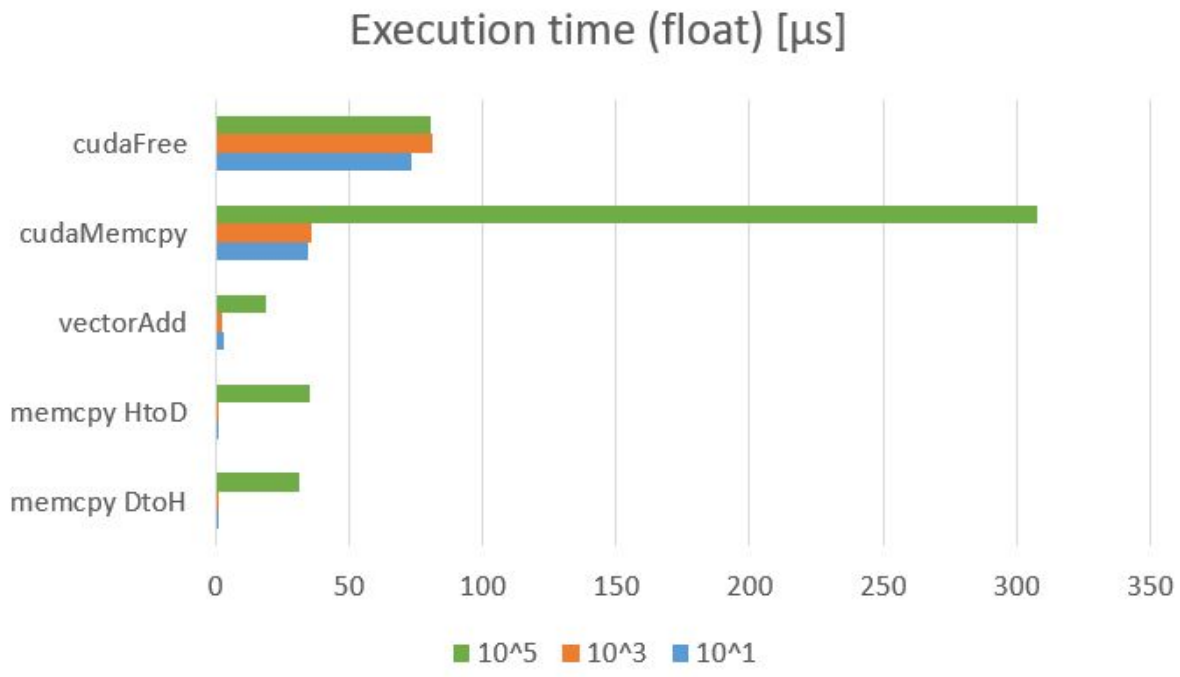
Number of elements	10^1	10^3	10^5	10^7	10^8
memcpy DtoH	0.928	0.992	30.465	22088	247710
memcpy HtoD	0.88	1.088	35.248	8957.2	88903
vectorAdd	3.168	3.008	18.912	1618.8	16157
cudaMalloc	125050	112530	121760	126210	127210
cudaMemcpy	32.895	46.887	291.77	14471	147820
cudaFree	79.503	74.94	75.615	1094.5	5089.9

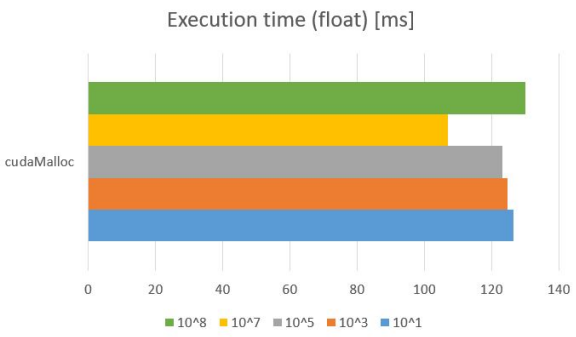
On the base of the tables above we created charts to visualize the data.

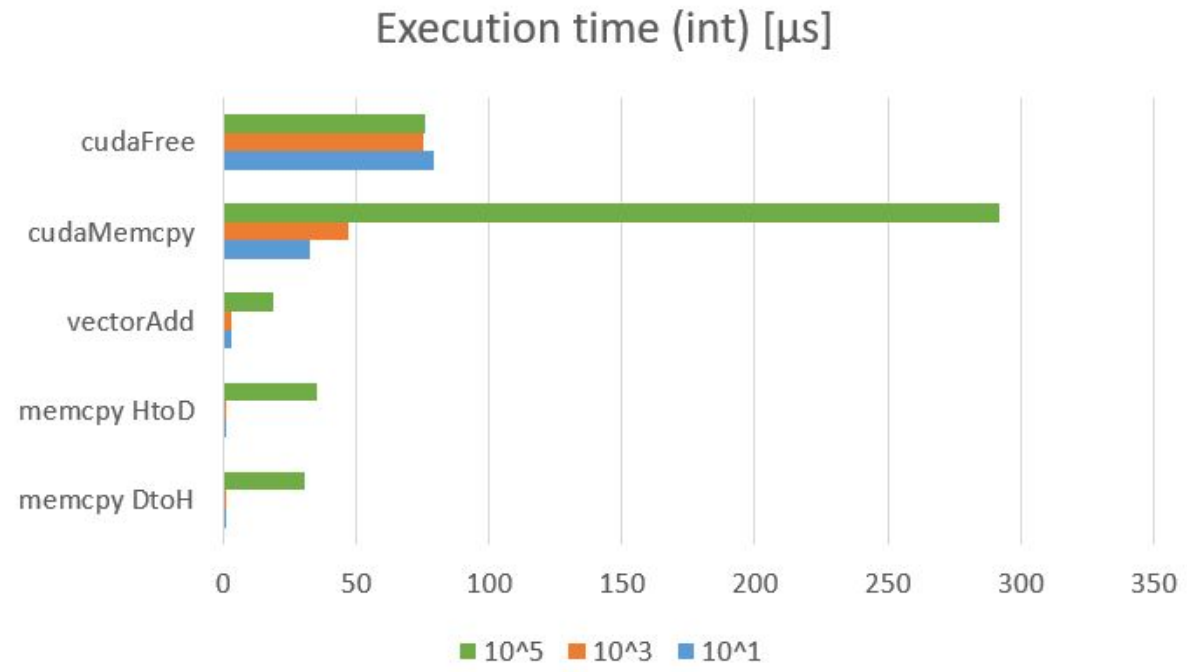

Execution time (double) [µs]
cudaFree
cudaMemcpy
vectorAdd
memcpy HtoD
memcpy DtoH
0
100
200
300
400
500
600
10^5
10^3
10^1

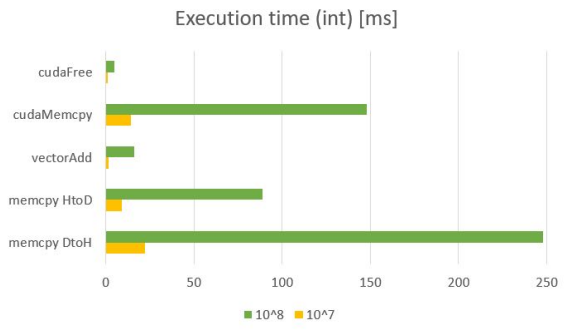

Execution time (double) [ms]
cudaFree
cudaMemcpy
vectorAdd
memcpy HtoD
memcpy DtoH
0
100
200
300
400
500
10^8
10^7

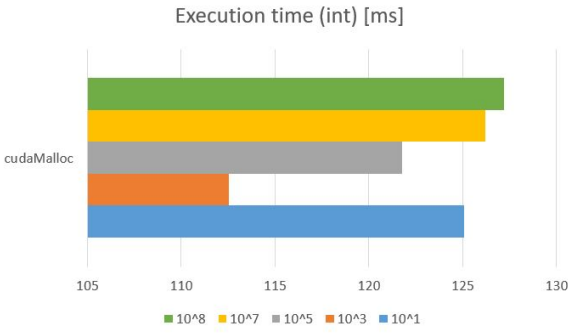

Execution time (double) [ms]
cudaMalloc
104
106
108
110
112
114
116
118
120
122
124
10^8
10^7
10^5
10^3
10^1


Execution time (float) [µs]
cudaFree
cudaMemcpy
vectorAdd
memcpy HtoD
memcpy DtoH
0
50
100
150
200
250
300
350
10^5
10^3
10^1







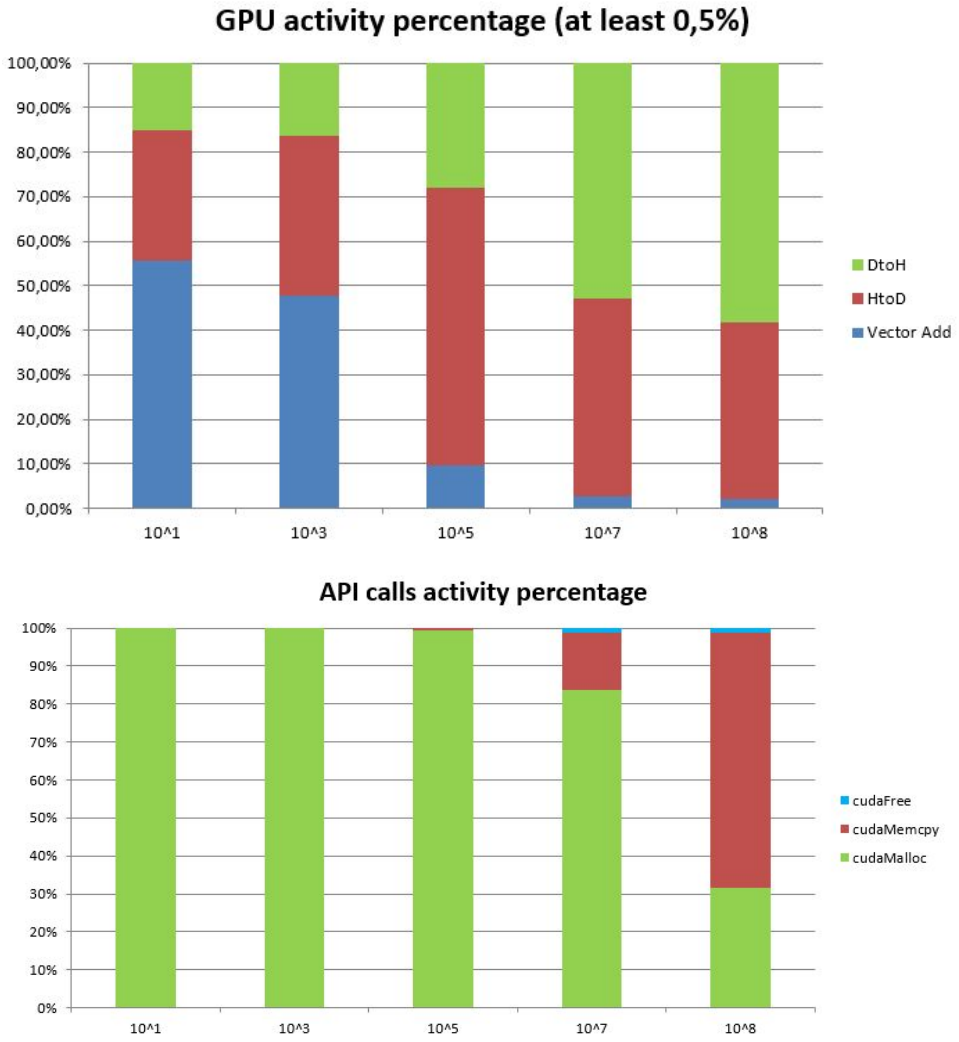


We can clearly see that the size of the vector has a clear influence on all the processes except the process of allocating the memory, which takes approximately 120 ms.

As far as the data types are concerned, there is a minor difference between integer and float. However, when adding vectors with double data type, the time consumed to move data between Host and Device is almost twice as long. This is not surprising due to the fact that the size of the double type is 8 bytes, compared to 4 bytes in integer and float. The

execution of vectorAdd function takes approximately the same amount of time, regardless of the data type.

Additionally, basing on the previous results we created the charts below, which clarify the percentage share of particular processes.



As expected with increasing data copying it becomes more and more time consuming in comparison to making actual operations by the GPU. This is also clearly visible on the second graph as cudaMemcpy is taking almost 70% of total activity in the last case.

2.2 Relationship between performance and number of threats.

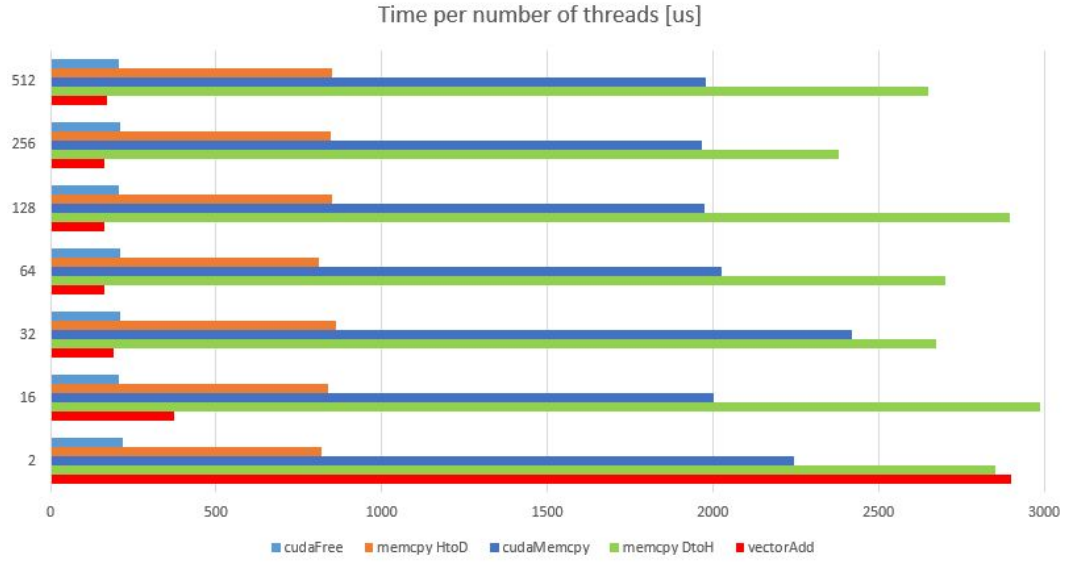
In order to find out, how a number of threads per block influences the performance of GPU in our process, we conducted three tests on each thread number (2-512), in order to get rid of potential, major fluctuations. The size of data in each test was the same: 10^6 elements.

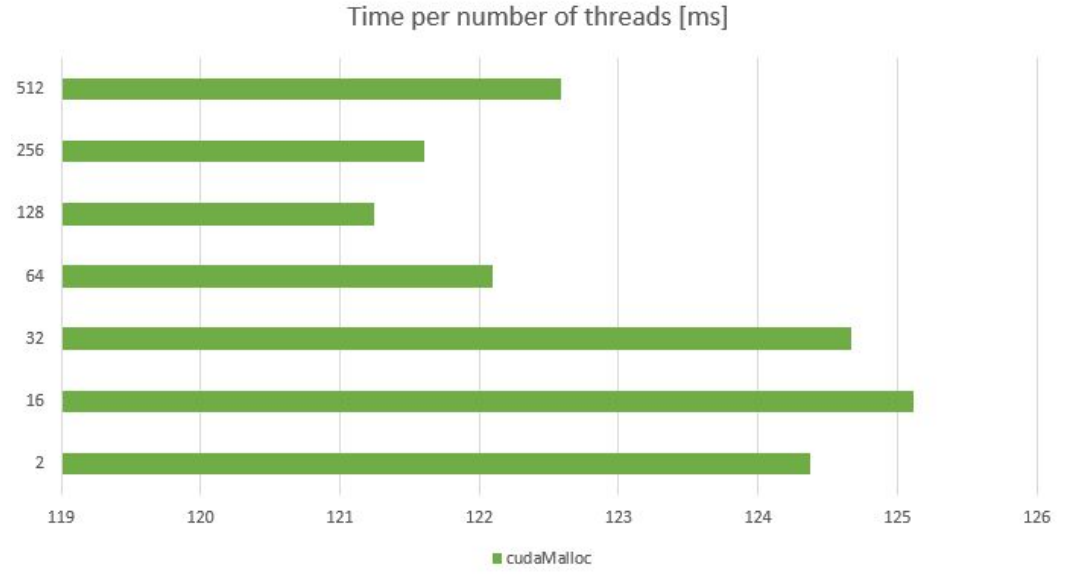
A table, containing the results of our analyzed test outcomes is placed below.
The cells contain the time [in μs] consumed on particular process depending on the number of threads in the blocks.

Table 4: Execution time of each process in [µs].

Threads per block:	2	16	32	64	128	256	512
memcpy DtoH	2852.6	2985.9	2672.3	2702.2	2895.7	2380.8	2649.4
memcpy HtoD	819.25	837.08	861.57	810.9	851.17	847.01	850.77
vectorAdd	2901	372.61	191.27	163.1	162.88	164.29	170.24
cudaMalloc	124370	125110	124670	122100	121250	121610	122590
cudaMemcpy	2242.7	2001.9	2419.8	2024.1	1974.3	1966.2	1976.7
cudaFree	219.44	205.96	210.97	212.34	205.94	210.15	207.2

We also created charts which visualize data in the table above.





According to the charts above, we can conclude that the main thing that is changing depending on the number of threads per block is the actual time of adding two vectors, due to the `vectorAdd` function. As we could have expected, actions connected to the data transfer from Host are virtually the same, because the size of vector is the same in each test, so the time consumed on processes connected with copying memory should stay on the same, or a very similar level. With the increase of threads per block there is a slight time decrease in copying memory from the Device, where 256 threads appear to be the most efficient with 2,38 ms. In the default code there is implemented simple algorithm which calculates how many blocks of threads need to be created to make sure that the program will finish it's task successfully. Changing it to constant value leads to interesting finds. When the total number of threads is lower than the number of elements in each of the vectors, the program will return failed verification result, as some elements of them were not added. On the contrary when the thread count exceeds 10^6 by a significant amount, we can see time increase in `vectorAdd` and `memcpyDtoH` functions. Results are shown in table below.

Table 5:Time per function in [µs]

Total number of threads	**1000192**	**2560000**
memcpy DtoH	1553,2	2460,4
memcpy HtoD	835,65	786,34
vectorAdd	165,79	325,15
cudaMalloc	123020	120080
cudaMemcpy	1898,4	1996,7
cudaFree	212,32	207,48