REPORT 1

Introduction to CUDA and openCL

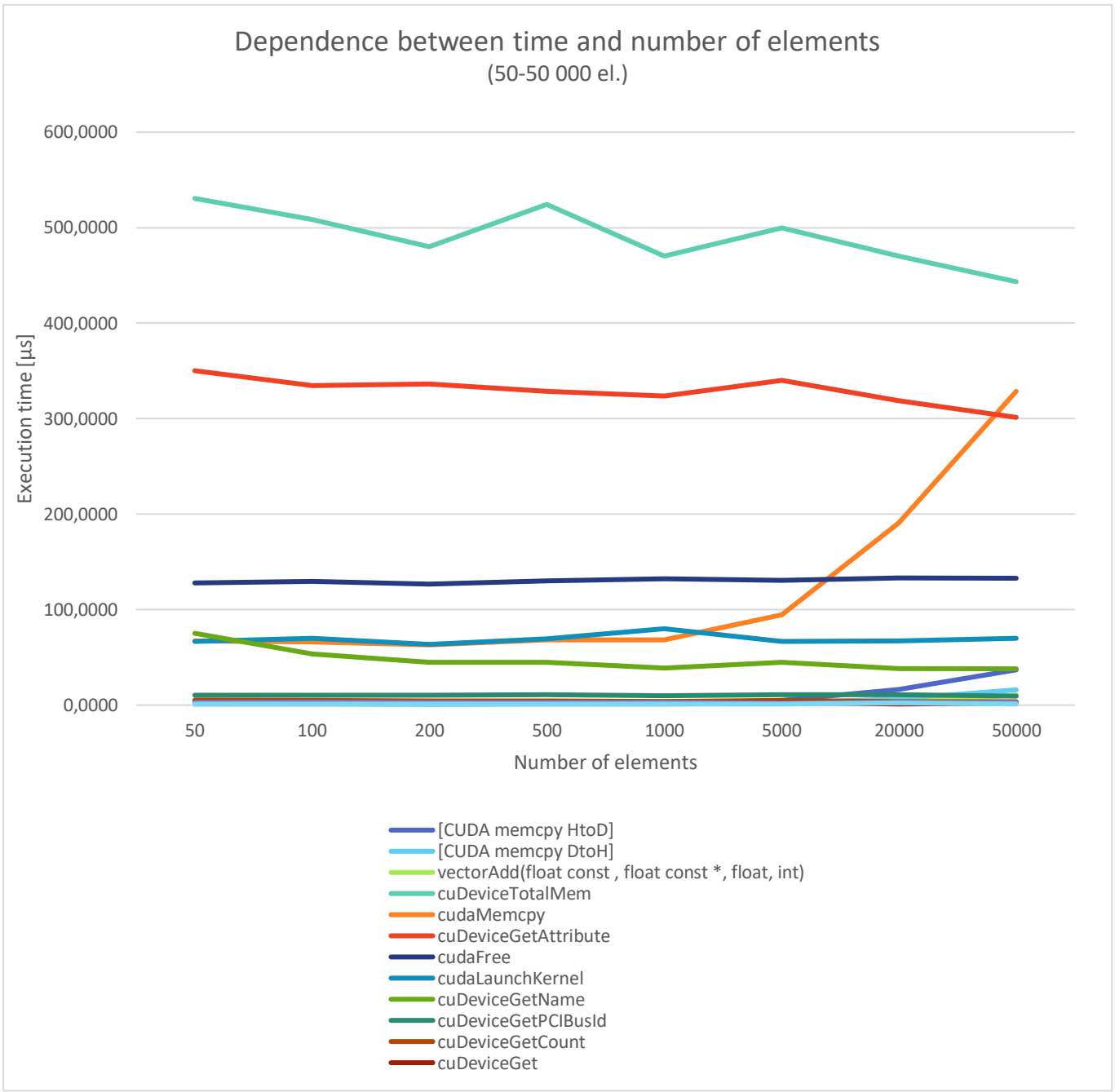
AGH-Faculty of Physics and Applied Computer Science

authors:
Kinga Pyrek
Aleksandra Rolka

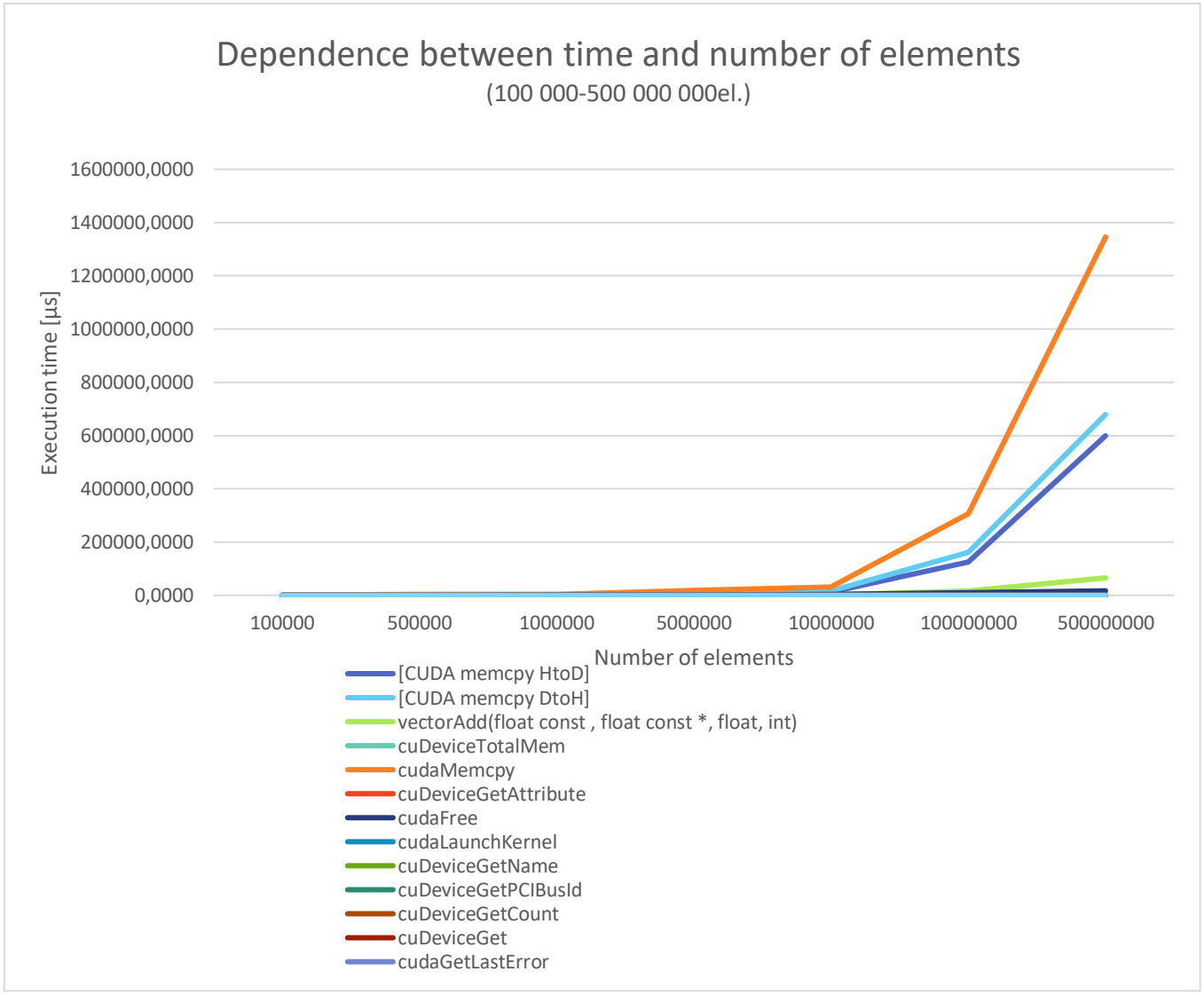
At the first laboratories we have been introduced to CUDA environment. We learned how to use Nsight and compile programs. Our first projects was addVector sample, which adds two vectors. We played with the code by changing number of elements in the vector, type of vector's variables and type of its size. On this basis, we have analyzed execution time of each functions:

- **CudaMemcpyKind-** Direction of data transfer (RAM/GPU).
Specifies what memory the source data is in and what memory area the target pointer points to.
  - **CUDA memcpy HtoD**-the source memory area belongs to the computer memory (RAM), while the destination memory area belongs to the graphics card.
  - **CUDA memcpy DtoH**- the source memory area belongs to the graphics card memory, while the destination memory area belongs to the computer (RAM)
- **vectorAdd**-adding 2 vectors
- **cudaMalloc**- Allocates memory on the graphics card.
- **CuDeviceTotalMem-** Returns the total amount of memory available on the device in bytes.
- **CudaMemcpy**- Copies data between the graphics card and RAM. The origin of the source and target memory data determines the argument of the function.
- **CuDeviceGetAttribute**- Returns the integer value of the attribute on device.
- **CudaFree**- Frees the memory space pointed to
- **CudaLaunchKernel**-launches Kernel
- **CuDeviceGetName**- Returns an ASCII string identifying the device
- **CuDeviceGetPCIBusId**- Returns in a device handle given a PCI bus ID string.
- **CuDeviceGetCount**- Returns the number of devices with compute capability greater than or equal to 2.0 that are available for execution
- **CuDeviceGet**- Returns in *device a device handle given an ordinal in the range [0, cuDeviceGetCount()-1].
- **CudaGetLastError** Returns the last error that has been produced by any of the runtime calls in the same host thread and resets it to cudaSuccess.
- **CuDeviceGetUuid**- Returns the last error that has been produced by any of the runtime calls in the same host thread and resets it to cudaSuccess

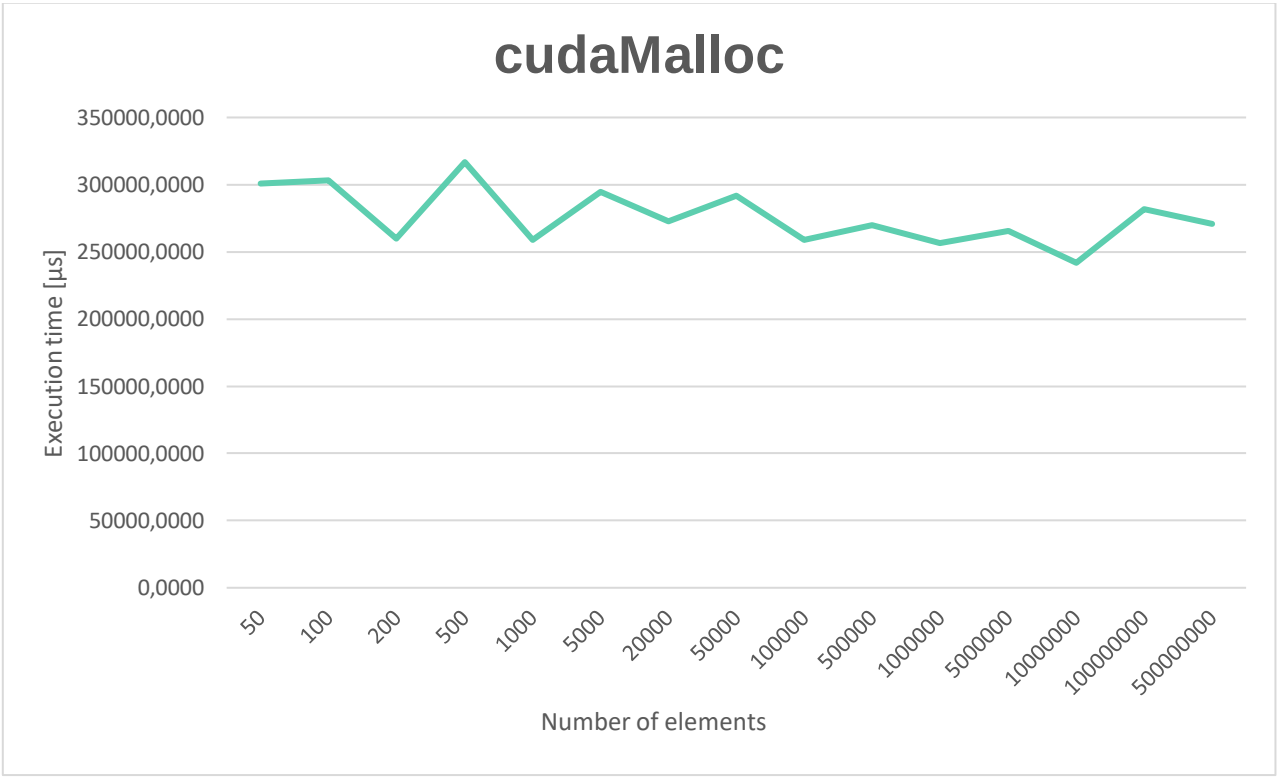
Here we have results showing the impact of number of (50,100,200,500,…) elements on the execution time[µs] :



… and for 100 000,500 000,…,5 00 000 000 elements of each vector:



For function cudaMalloc we've done separate plot because its execution time were much more demanding than other's functions.



Observations:

The highest values of execution time are reached by the **cuDeviceTotalMem** function - stable, but slightly decreases.

Cuda Memcpy: for 0-1000 elements time is constant. For more than 1000 elements execution time function of **Cuda Memcpy** increases (approximately x^2).

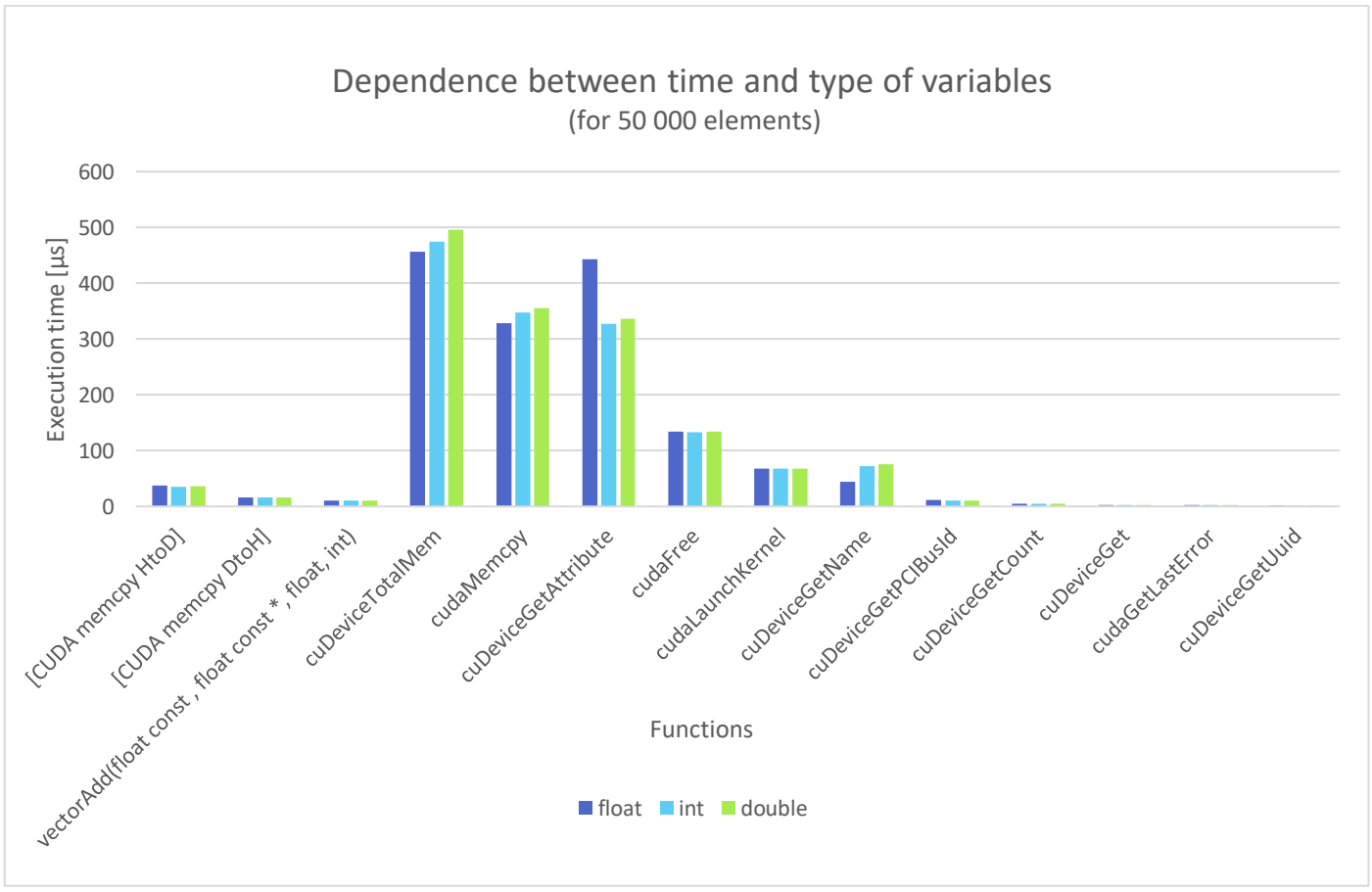
Other functions have constant and small execution time values. Only **cuDeviceGetAttribute** needs consistently large time values.

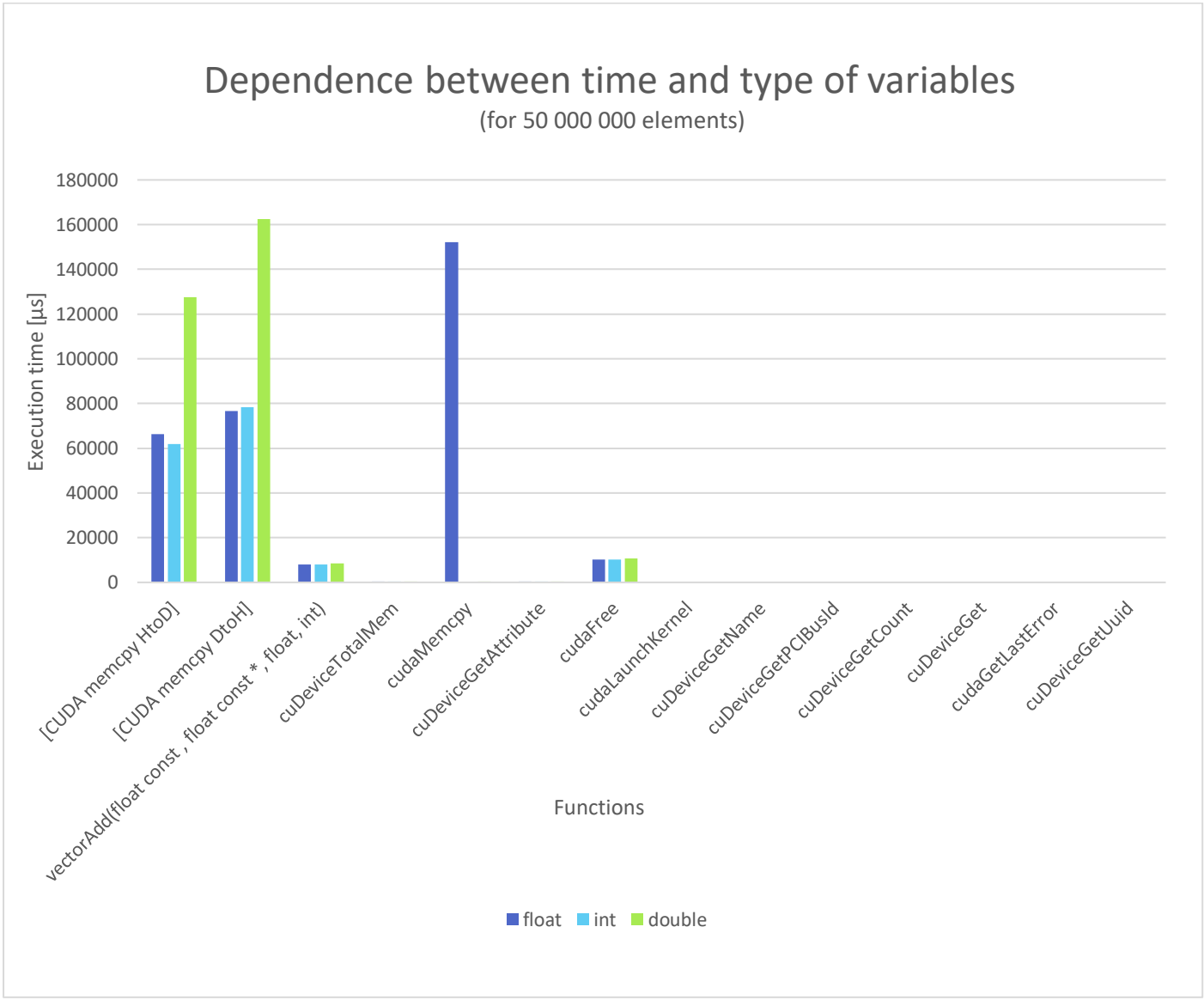
In the case of **other functions**, the number of elements has negligible significance on the time of their execution. We also noticed that up to 10^7 elements, functions need minimal execution time (order of magnitude: 10^0 - 10^3).

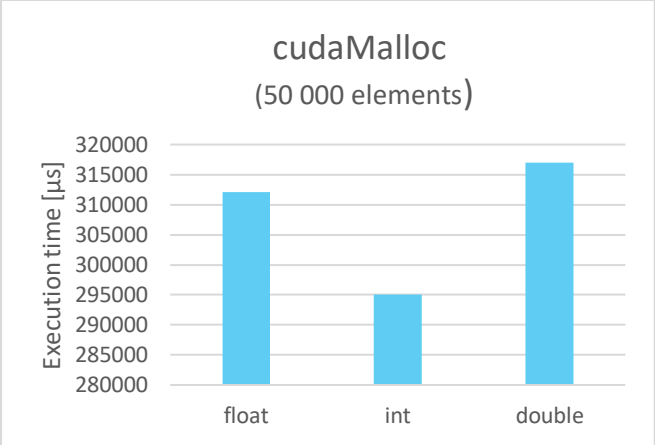
Then from 10,000,000 the time needed to perform the CudaMemcpy HtoD and DtoH function begins to increase dramatically, and the rest remains unchanged.

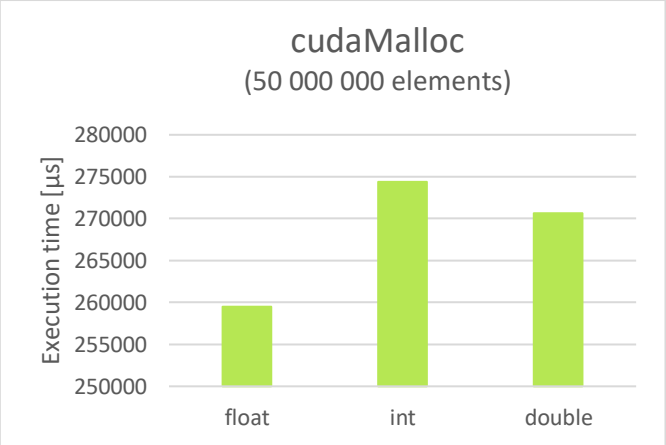
CudaMalloc: On the analyzed range of the number of elements, the time oscillates between 24830 µs - 316700 µs.

The next part of the analysis was based on changing the type of variables with constant number of elements for 50 000 and 500 000 000. We also separated the cudaMalloc function's plot:




Dependence between time and type of variables
(for 50 000 000 elements)
Execution time [µs]
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
[CUDA memcpy HtoD]
[CUDA memcpy DtoH]
vectorAdd(float const , float const * , float, int)
cuDeviceTotalMem
cudaMemcpy
cuDeviceGetAttribute
cudaFree
cudaLaunchKernel
cuDeviceGetName
cuDeviceGetPCIBusId
cuDeviceGetCount
cuDeviceGet
cudaGetLastError
cuDeviceGetUuid
Functions
float
int
double





When we changed types, the only major differences are noticeable for the functions **cuDeviceTotalMem, CudaMemcpy, cuDeviceGetName**. The time value increases with the float, int, double types respectively.
For types int and double as for previous functions. Execution time for double is greater than for int, but the largest execution time is needed for float (much larger than for int and double).

When we increased the number of elements, the type of variables began to matter for the following functions:

- **CUDA memcpyDtoH and CUDA memcpyHtoD** - the execution time for double is twice as long as for int and float, between which there are no major differences.
- **Cuda memcpy** – for float the time it takes to perform this function is huge compared to execution time for int and double, which is 1000 times shorter.

For the **malloc** function we see that for 50 000 elements execution time increases in order: int, float, double, whereas the double has two times shorter execution time. It is different when we have 1000 times more elements. Then int and double have similar values, and float has 2 times less execution time as we can see in the chart.

Conclusions:

When constructing a program whose main task is to add elements, we must be aware that with a large amount of data, the functions: CUDA Memcpy HtoD, CUDA Memcpy DtoH and cudaMemcpy have a greater demand for execution time and this time increases very quickly as the number of elements increases (from 10,000,000 up). As you can see, these are memory functions, so the execution time strongly depends on the number of elements.

When building a program, we should also consider what types of data to use. With more elements, we have to choose whether we care more about better precision or a shorter time of performance of the CUDA Memcpy HtoD and CUDA Memcpy DtoH functions, because for the double type this time is over 2 times greater than for int and float.