<table>
<tr><td rowspan="3">  </td><td>Module:
Introduction to CUDA and OpenCL</td><td colspan="3" rowspan="2">Report title:
Report 2</td></tr>
<tr><td>Instructor:
dr hab. inż. Szumlak Tomasz</td></tr>
<tr><td>Name and surname:
Marcin Filipek
Anca Dărăbuț</td><td>Group:
11</td><td>Date:
30.10.2019</td><td>Grade:</td></tr>
</table>

1. Handling large data structures

Using massive parallelism is designed to process huge data structures faster. But even such a big calculator has limited memory. Programmers must be aware of this limit. During the labs we were checking what happens when we increase data size. We were continuously running the program with bigger and bigger vectors. Below 2^30 of floats we were able to get the result properly. When we reach the threshold the kernel did not launch and program had crashed.

2. Memory Management Utility

Memory Management Utility is a tool which brings an automation in managing memory device. It allocates the memory, copies it to and from device and frees the memory for us. Everything is done “behind the scene”. This effortless tool is very useful while writing CUDA algorithms. It replaces every memory allocs and frees instructions both on the host and the device, also every copy function with one line of code, which is

cudaMallocManaged(&variable,size), where:
variable is a name of our variable, **&** means that we pass its address,
size is a size of our data.

This simple tool clarifies the code and allows programmers to focus on the purpose of their application, instead of managing memory. It creates the unified memory, i.e. a single memory address space which is available from any processor unit. In this manner, applications from either CPUs or GPUs can process the data.

During our laboratories we messed up with the code and replaced every manual memory operation with automatic memory management. Then we did the profiling to check if it affects to the time of execution. What was different, that it printed only one line in laber, which was the sum of all memory operations. The comparison proved that it is

slightly longer, but it is caused by the fact that our program is a simple addition of two vectors. In a real example, where calculations are much more complex and take much more time it relatively does not affect.

3. Protection against copying too large data structures on GPU

While using memory managed utility, there may be cases when a great amount of data has to be copied to GPU, but the memory of the device is insufficient. In order to avoid such events, we created a snippet of code, following these steps:

- querying the device
- getting information about the amount of free memory
- informing the user about the error
- exit the code execution

```
// initialize the variables where we store the total and free memory size
size_t mem_free = 0 , mem_tot = 0;
// set the device 0 for GPU executions
cudaSetDevice(0);
// ask for the amount of total and free memory available for allocation by the device
cudaMemGetInfo(&mem_free, &mem_tot);
printf("Free memory: %zu\nTotal memory: %zu\n",mem_free,mem_tot);

//calculate total data size
size_t size = numElements * sizeof(float);
printf("Data size: %zu\n",size);
//check if the data is not too big
if (size > mem_free) {
    printf("Too big data!\n");
    exit(EXIT_FAILURE);
}
```

Consequently, if the size of the randomly generated data requires an amount of memory greater than the available one, the message “Too big data!” shall be printed and the data is not sent to the GPU. Execution stops.