# Introduction to CUDA and OpenCL

Łukasz Gut, Grzegorz Litarowicz

Lab 07, 23.04.2020

## Introduction

During seventh laboratories, we were introduced to the OpenCL framework. We were assigned four tasks: compare the device query output from the OpenCL with the CUDA one, perform performance studies on simple vector addition examples and implement cascade addition of multiple vectors.

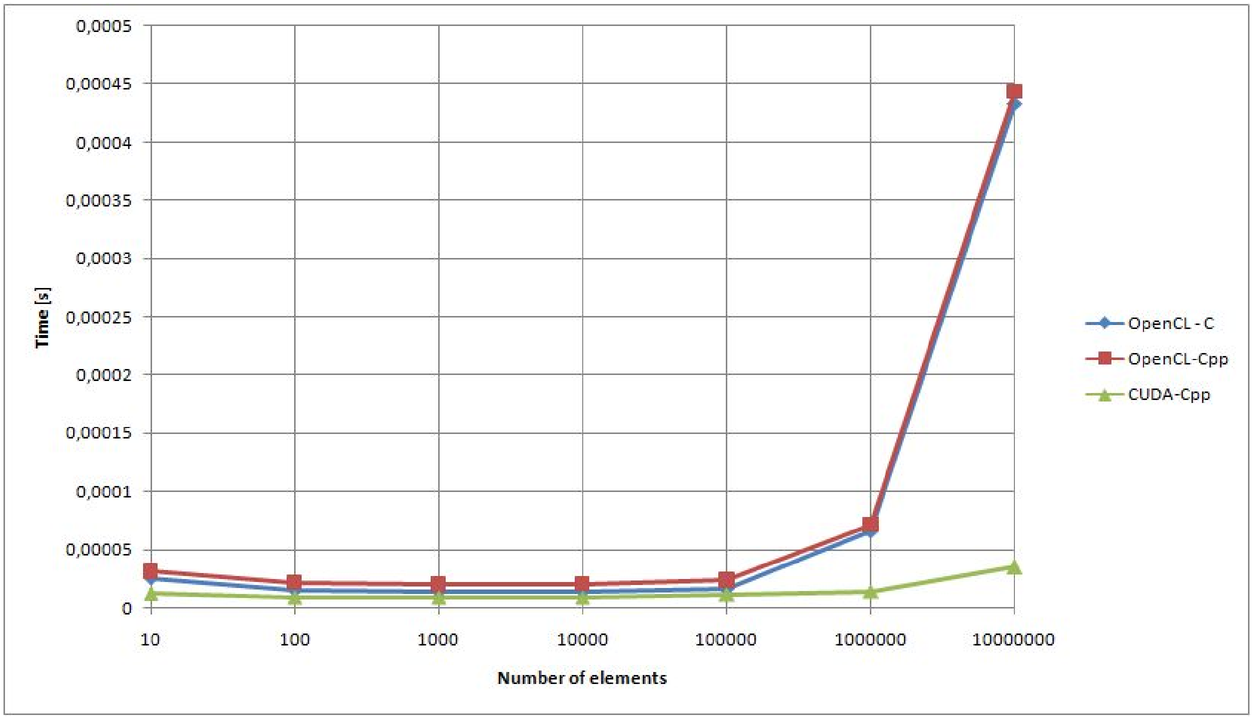
## Device Query

In the first task, we had to analyze, a program written in OpenCL which collected information about available compute devices. At first sight, we thought due to a higher level of abstraction we get much less information about GPU, but after reading the documentation it turns out we can get very similar information like in CUDA framework. So this feels very familiar if you already know OpenCL.

## Vector addition benchmark

The second task was to take a look at vector addition implementation in OpenCL and benchmark it against the CUDA example.

This time we only measured the kernel execution time, to see how these compare.



As we can see, the CUDA framework is much faster. The difference is huge when dealing with very large data. The OpenCL example could probably be optimized a little bit more. We could, for example, optimize it by defining our own group size and not leave it to the OpenCL runtime.

## Cascade Addition

With cascade addition, we decided to abstract OpenCL state into a global struct. This way we can easily create the whole OpenCL context  and then reuse it across the application. This is essential because we do not want to create and destroy the context all the time. Usually, this type of stuff is done during initialization and shutdown routines. What is more, this gives us an easy way to reuse the OpenCL. Similarly to the context, we allocate buffers once and then reuse them for our convenience.

Since we decided to implement cascade vector addition using the C language, the struct that holds the context looks as follows:

```
typedef struct DeviceContext {
    cl_device_id device_id;
    cl_context context;
    cl_command_queue commands;
    cl_program program;
    cl_kernel ko_vadd;
    cl_mem d_a;
    cl_mem d_b;
    cl_mem d_c;
} DeviceContext;
```

Then we just created a simple function that performs vector addition on data provided in h_a and h_b and stores the result back in h_c.

```
typedef struct HostData {
    float* h_a;
    float* h_b;
    float* h_c;
} HostData;

void add_vectors(HostData hData);
```

This allowed us to invoke the function as many times as we want and perform cascade vector addition.

## Conclusion

In conclusion, working with OpenCL is very similar to working with CUDA. Although this framework requires a little bit of more context preparation, it nicely abstracts all of our available computing devices and makes use of them.

We can also see, that using OpenCL with the C language is negligibly faster than using it with C++. However, for this tiny bit of performance, we pay a huge price - no abstractions. In our opinion, it is really not worth using C when you have C++ at your disposal.

What is more, we can observe that the higher-level abstractions of OpenCL are not zero-cost abstractions.