1.OpenCL Structure

During this lab our task was to understand the structure of openCL on the basis of a few examples starting from querying the device and ending with consecutive vector addition.

Although CUDA and OpenCL have many similarities, here are some of the differences worth talking about. The platform model consist of host and various OpenCL Devices (divided further into Compute Units and Processing Elements). The main advantage of OpenCL is that both CPU's and GPU's can act as a Device - large number of various processing units is supported. When it comes to memory - here we have the well known division - host and device. As in CUDA we have Kernel and host application. Instead of Grid we create NDRange - domain of work items (equivalents of threads) which can be a part of work group.

2.OpenCL Vector addition

The main task of these labs was to rewrite given sample of OpenCL code which adds two vectors. Final code was supposed to perform following additions: C=A+B,D = C+E,F=G+D and check if the result is correct using serial code.

First part of the OpenCL code is the defining of platforms and queues as well as creating the kernel functor. The context is a way of managing the resources and objects in OpenCL.

```
cl::Context context(DEVICE);
cl::Program program(context, util::loadProgram("vadd.cl"), true);
cl::CommandQueue queue(context);
cl::make_kernel<cl::Buffer, cl::Buffer, cl::Buffer, int>
vadd(program, "vadd");
```

Used kernel is shown below:

```
__kernel void vadd(
    __global float* a,
    __global float* b,
    __global float* c,
    const unsigned int count)
  {
    int i = get_global_id(0);
    if(i < count)  {
        c[i] = a[i] + b[i];
    }
  }
```

It is quite similar to the kernels used in CUDA equivalent of this code. Only noticeable difference is the get_global_id(0) which can be understood as the equivalent of mapping threads.

Defining memory objects and managing it in the code requires us to create the so called Global Buffers. They are the conduit through which data is communicated from the host application to OpenCL C kernels . Here we can see simple allocation of memory at host (h_a) and how the vector is copied into the device.

```
std::vector<float> h_a(LENGTH);
cl::Buffer d_a;
d_a = cl::Buffer(context, h_a.begin(), h_a.end(), true);
// …
```

In order to allocate memory on the device we have to create buffer with appropriate flag to specify allocation and usage information. Here flag CL_MEM_READ_WRITE allows us to use this vector in the next calculations without the need of copying it back to the host before the operations are complete.

```
d_c  = cl::Buffer(context, CL_MEM_READ_WRITE, sizeof(float) *
LENGTH);
// ...
cl::copy(queue, d_c, h_c.begin(), h_c.end())
```

The kernel execution is shown below. As we can see it requires all the variables specified in kernel as well as the queue and the domain of work-items.

```
int count = LENGTH;
vadd(cl::EnqueueArgs(queue,cl::NDRange(count)),d_a,
d_b,d_c,count);
queue.finish();
```

It is clear that that the way of programming becomes much more abstract in comparison to the CUDA.

3.CUDA equivalent

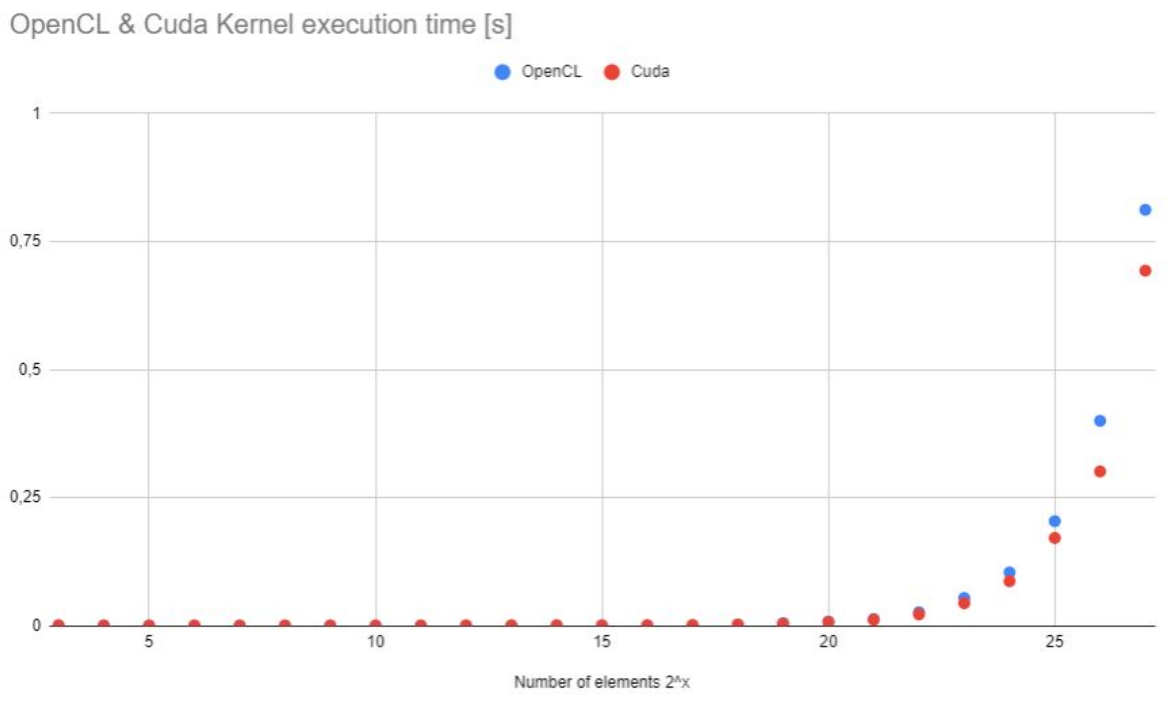
In the CUDA equivalent we have a simple kernel:

```
__global__ void
vectorAdd(const float *A, const float *B, float *C, int
numElements)
{
int i = blockDim.x * blockIdx.x + threadIdx.x;
if (i < numElements)
{
C[i] = A[i] + B[i];
}
}
```

In this case we modified the code from third laboratories to perform consecutive addition. Because the program was already a topic of one of the previous reports, the description of it will be ommited. Level of abstraction in programming is noticeably lower, which makes the operation of adding two vectors much more straightforward.

4.Performance comparison

Now it's time for the results:



On the first look we can see that despite the fact that OpenCL enables us to use not only GPU, there is a significant difference in performance for the CUDA site.

For the vectors up to around 2^10 elements, the difference is non-measurable.

When we are going to the higher numbers of elements we can notice an advantage for the OpenCL computing - for 2^18 - element vectors, OpenCL computing takes 90% of the CUDA computation time. This however may be caused by the not satisfying precision of the time measurement.

As we go higher in the numbers of vector elements we can clearly see the advantage for the CUDA computing - from the amount of 2^19 elements CUDA performance is significantly better.

The peak of the superiority CUDA has at exactly the same amounts of elements - 2^19 where OpenCL computation takes 136% of the CUDA computation.
However CUDA times are constantly better, where for the 2^26 -element vectors OpenCL still takes 133% of the CUDA computation time.

Basing on the results we can be a bit disappointed having on the mind that OpenCL utilizes the common computing power of CPU and GPU.
However we have to remember the level of complexity of our computational task - we were just adding vectors, here the level of abstraction which OpenCL gives, apparently showed it’s weakness.