OpenCL exercise 1: Basics

Kaicong Sun

Task 1

- Calculate cosine function for a number of values on the GPU
- CPU code:

```
for (std::size_t i = 0; i < h_output.size (); i++)
    h_output[i] = std::cos (h_input[i]);
```

- Use one work item per value on the GPU

Steps

- Allocate memory on the device (`cl::Buffer` constructor)
- Initialize the memory on the device (`cl::CommandQueue::enqueueWriteBuffer()`)
- Copy the input data to the device (`cl::CommandQueue::enqueueWriteBuffer()`)
- Launch the kernel (`cl::Kernel::setArg()`, `cl::CommandQueue::enqueueNDRangeKernel()`)
- Copy the output data to the host (`cl::CommandQueue::enqueueReadBuffer()`)

Task 2

Add code for measuring

- Time needed for calculation on host
- Time needed for calculation on device
- Time needed for memory transactions

Speedup:

- Time on the CPU / Time on the GPU
- Time on the CPU / (Time on the GPU + Time for Memory transactions)

Compare the times using a Debug build and a Release build.

Task 3

Use `native_cos()` instead of `cos()` in the kernel and compare the performance.

Syntax: Memory allocation

Allocate memory:

```
cl::Buffer::Buffer(cl::Context context,
    cl_mem_flags flags, std::size_t size);
```

`context` = The context to use
`flags` = Normally `CL_MEM_READ_WRITE`
`size` = Number of bytes (not elements) to allocate

Syntax: Copying data

Copy data from CPU to GPU (global) memory:

```
cl::CommandQueue::enqueueWriteBuffer(cl::Buffer buffer,
    bool blocking, std::size_t offset, std::size_t size,
    const void* ptr, eventsToWaitFor = NULL,
    cl::Event* resultEvent = NULL) const;
```

`buffer` = The buffer to copy to
`blocking` = Wait until the copy operation has finished (normally `true`)
`offset` = Offset into the buffer (in bytes)
`size` = Number of bytes (not elements) to copy
`ptr` = Pointer to source data in CPU memory
`eventsToWaitFor` = Events which have to occur before the copy operation is started, normally `NULL`
`resultEvent` = Pointer to a variable where an event is stored (can be used for profiling)

Syntax: Copying data

Copy data from GPU (global) to CPU memory:

```
cl::CommandQueue::enqueueReadBuffer(cl::Buffer buffer,
    bool blocking, std::size_t offset, std::size_t size,
    void* ptr, eventsToWaitFor = NULL,
    cl::Event* resultEvent = NULL) const;
```

`buffer` = The buffer to copy from
`blocking` = Wait until the copy operation has finished (normally `true`)
`offset` = Offset in the buffer object (in bytes)
`size` = Number of bytes (not elements) to copy
`ptr` = Pointer to destination in CPU memory
`eventsToWaitFor` = Events which have to occur before the copy operation is started, normally `NULL`
`resultEvent` = Pointer to a variable where an event is stored (can be used for profiling)

Syntax: Launching a kernel

Set parameters for a kernel launch:

```
cl::Kernel::setArg<T>(cl_uint index, T variable);
```

`T` = The type of the parameter (e.g. `cl_int` or `cl::Buffer`)
`index` = 0-based index of the parameter
`variable` = The variable you use as argument of kernel function

Launch the kernel:

```
cl::CommandQueue::enqueueNDRangeKernel(Kernel kernel,
    NDRange offset, NDRange global, NDRange local,
    eventsToWaitFor = NULL, cl::Event* event=NULL) const;
```

`kernel` = The kernel to launch
`offset` = Normally `0` or `cl::NullRange`
`global` = The overall number of work items
`local` = The number of work items per work group

Syntax: Kernel code

Address space qualifier specifies the region of memory used to allocate object:

```
__global, __local, __constant, __private
```

Kernel arguments declared as pointer of a type can point to only following address space

```
__global, __local, __constant
```

for instance:

```
__global int* foo; //Declare foo as a pointer to global mem.
```

All variables inside a function (including kernel functions), or passed into the function as arguments are in the private address space if without explicit address space qualifier except for arguments declared to be of type (considered in global address space)

```
image2d_t, image3d_t
```

Syntax: Work-Item Functions

Get global index of the current work item in the x-direction:

```
size_t i = get_global_id(0);
0 means x-direction, 1 means y-direct., 2 means z-direct.
```

Get global size of work group in the x-direction:

```
size_t i = get_global_size(0);
```

Some other functions:

```
size_t i = get_local_size(0);
size_t i = get_local_id(0);
size_t i = get_num_groups(0);
size_t i = get_group_id(0);
```

Profiling

On the CPU:

```
Core::TimeSpan time1 = Core::getCurrentTime();
// Execute some code ...
Core::TimeSpan time2 = Core::getCurrentTime();

Core::TimeSpan time = time2 - time1;
std::cout << time << std::endl;
```

On the GPU:

```
cl::Event event;
queue.enqueue...(..., &event);
queue.finish();//or queue.enqueueWriteBuffer(blocking=true)

Core::TimeSpan time = OpenCL::getElapsedTime(event);
std::cout << time << std::endl;
```