

OS (H) Assessed Exercise: OpenCL Host Programming – Part I

Wim Vanderbauwhede

1 Brief

The purpose of this coursework is to create a simple host-side driver routine `run_driver()` to interact with an OpenCL-compliant device (e.g. a GPU or multicore CPU). This `run_driver()` routine is called from a multithreaded testbench which will take care of initialisation and shutdown of the device, creation of the data to be sent to the device and validation of the returned result. Source code for such a testbench will be provided for the second part of the assignment.

2 Aims

The aims of the coursework are to demonstrate your:

- ability to write multithreaded programs using the POSIX threads API;
- understanding of the concepts of a device driver, device hardware and device firmware code;
- ability to learn, understand and apply a standardised programming API, in this case OpenCL.

The aim of this first part of the assignment is to make you think carefully about your solution, before writing any code.

3 Overview of the OpenCL host/device interaction

More detail on the OpenCL API as well as source code skeletons and source code for the `init_driver()` and `shutdown_driver()` routines and the testbench will be provided for the second part of the assignment, when you will write the actual code.

The driver routine `run_driver()` has the following interface:

```
int run_driver(  
    CLObject* ocl,  
    unsigned int buffer_size,
```

```

int* input_buffer_1,
int* input_buffer_2,
int w1, int w2,
int* output_buffer
);

```

The routine takes as arguments three pointers to data buffers of the same size, the size of the buffers, two integer weights and a pointer to a `CLObject` struct, which encapsulates the required OpenCL-specific objects.

3.1 OpenCL as a generic device driver API

OpenCL is an open standard for heterogeneous programming. In the OpenCL model, the system consists of a host and a number of devices. Typically, the host is a platform running an conventional operating system, and the device is typically a GPU or similar hardware accelerator. However, OpenCL also works on the host CPU, and this is the recommended device to use in this coursework.

The OpenCL model can be regarded as a generic form of device driver programming: the device has configurable firmware (called the “kernel”) and a configurable data transfer interface (defined by the signature of the kernel subroutine).

For this coursework, we are not concerned with the firmware running on the device (the “kernel”) except that its source code consists of a single subroutine with following signature (you can ignore the `_kernel` qualifier:

```

__kernel void firmware(
    int* input_buffer_1,
    int* input_buffer_2,
    int* output_buffer,
    int* status,
    const unsigned int w1,
    const unsigned int w2,
    const unsigned int buffer_size
);

```

OpenCL provides API calls to load program source code from a file (or string), compile it and transfer it to the device. For the coursework, all these actions are performed by the routine `init_driver()`, the code for which will be provided. This routine returns a pointer to a `CLObject` struct:

```

CLObject* init_driver()

```

The `CLObject` is a convenience data structure (not part of the OpenCL API) containing the OpenCL objects required to interact with the device:

```
typedef struct ocl_objects {
    cl_device_id device_id;
    cl_context context;
    cl_command_queue command_queue;
    cl_program program;
    cl_kernel kernel;
    int status;
} CLObject;
```

For your code you only need the fields `context`, `command_queue` and `kernel`. The context is used by the OpenCL runtime for managing various objects such as command-queues and kernel objects. Some of the API calls that you need for the `run_driver()` code require the `context`, `command_queue` or `kernel` as arguments. This is why the `ocl` object returned by `init_driver()` is an argument of the `run_driver()` routine.

3.2 Interaction between driver and device in OpenCL

3.2.1 Device argument configuration

The way the arguments of the `firmware` subroutine are configured on the device is through a series of registers (the number is fixed but depends on the actual hardware device). Each register can contain either a constant or a pointer to an area of the device memory. For example for the `firmware()` subroutine signature above, registers 0 to 3 would contain pointers to memory, register 4 would contain an integer constant. OpenCL provides the API call `clSetKernelArg()` to configure the device.

3.2.2 Data transfers between host and device

Once the driver firmware has been loaded and the arguments configured, the data can be written to the device memory. To do so, OpenCL provides a datastructure to manage the relationship between the data in the host memory and the data in the device memory, called a Buffer Object. This object is created using `clCreateBuffer()`.

The actual command to transfer the data to the device is `clEnqueueWriteBuffer()`. The name contains “Enqueue” because of OpenCL’s model for sending commands to the device: rather than having API calls that perform the command immediately, the calls put the commands in a Command Queue, a special OpenCL data structure that allows the OpenCL runtime system to manage the commands in a flexible, event-driven way. In this way it is possible to create asynchronous operation with overlapping actions. For this coursework this is not important because we will use fully synchronous blocking operations.

3.2.3 Running the device firmware code

Once the data has been written to the device, we instruct the device to run the kernel (i.e. the firmware code). This is done using the command `clEnqueueNDRangeKernel()`. The “NDRange” in the name refers to the configuration of the number of parallel threads the device will use. This is the key mechanism that OpenCL uses to control parallelism. For example, an NVidia Kepler GPU has typically about 16 cores each with 64 hardware threads, and the NDRange allows to configure how many of these threads you want to use. For the coursework our focus is not on the amount of parallelism so the values for the NDRange will be provided in the code skeleton.

Once the kernel has been started on the device the host needs to be notified when it finishes, for that purpose OpenCL provides the `clFinish()` call. This call blocks until the action on the device has finished, so after that the status can be read back from the device using `clEnqueueReadBuffer()`. If the status is 0, the data can be read back from the device using `clEnqueueReadBuffer()`.

When that is done, you can release the Buffer objects using `clReleaseMemObject()`.

4 Specification of the run_driver subroutine behaviour

NOTE: In this section, the use of the word *must* means you will lose marks if you don’t do this.

In the testbench code, the subroutine `run_driver()` will be called from several threads after a call to `init_driver()` and before a call to `shutdown_driver()` (the code for both these routines will be provided for the second part of the assignment):

```
// ...
start_thread (...) {
// ...
    status = run_driver(ocl, buffer_size,
        input_buffer_1, input_buffer_2, w1, w2, output_buffer);
// ...
}

int main() {
    CLObject* ocl = init_driver();
// ...
    for (...) {
// run a number of threads calling the device using the driver
    }
    shutdown_driver(ocl);
}
```

- The subroutine `run_driver()` *must* have the signature as specified above;
- The subroutine `run_driver()` *must* be thread-safe
- The subroutine `run_driver()` *must* perform the following high-level actions when called:
 - transfer `input_buffer_1` and `input_buffer_2`, which are buffers of size `buffer_size` containing signed integers, and the weights `w1` and `w2`, to the device,
 - start the computation on the device,
 - when the device returns a `status` of 0 (i.e. success), transfer the result from the device to the host and return it into `output_buffer`, which must also be a buffer of size `buffer_size` and type `int`.
 - otherwise only return the status.
- The return value of the `run_driver()` subroutine *must* be the status of the device.
- It is possible (and in fact common) for a call of a driver to a device to fail and return a non-zero status. Following common practice, in such a case the driver *must* try again for a fixed number of attempts.

5 List of required OpenCL API calls

These are the OpenCL API calls required to program the functionality describe above.

```
clSetKernelArg
clCreateBuffer
clEnqueueWriteBuffer
clEnqueueNDRangeKernel
clFinish
clEnqueueReadBuffer
clReleaseMemObject
```

6 What to submit

NOTE: In this section, the use of the word *must* means you will lose marks if you don't do this.

For the first part of the assignment, you have to draw an interaction diagram that will illustrate the *complete* communication between the host and the device required to successfully perform the actions described below. There is no particular syntax required, but it is expected that the diagram consists of blocks and arrows. The main criteria are that the diagram is correct, complete, clear and readable. A hand-drawn diagram is perfectly acceptable if it meets these criteria.

You *must* submit this diagram in JPG or PNG format through the Moodle submission system, and the filename *must* be *< your matric + 1st char of your name in lowercase >. < jpg or png >*, so for example if your matric number is *1107023m* then your file *must* be named *1107023m.jpg*.

6.1 Criteria and weighting

The diagram does contribute to your final mark but not as a separate mark: you *must* submit it and it *must* be of sufficient quality, otherwise you will be penalised 10 marks (out of 60) on the total for the assignment.

The quality criteria for the diagram are:

- 4/10 correctness,
- 3/10 completeness,
- 2/10 clarity,
- 1/10 readability