OS3 Assessed Exercise: OpenCL Host Programming – Part I

Wim Vanderbauwhede

1 Aim

The aim 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). 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 What to submit

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, the main requirements are that the diagram is correct, complete, clear and readable.

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*.

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* output_buffer
    );
```

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.

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:

```
void firmware(
    int* input_buffer_1,
    int* input_buffer_2,
    int* output_buffer,
    int* status,
    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;
```

It is populated by calling the subroutine `init_driver()`, which will be provided. 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 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 it is an argument of the `run_driver()` routine.

The way the firmware subroutine signature is configured on the device is through a series of registers that specify the arguments (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.

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.

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 Keppler 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 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

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, 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, 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.

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
```