SCIT

School of Computing & Information Technology

CSCI376 – Multicore and GPU Programming

Events and Synchronisation

The aim of this tutorial is to introduce you to OpenCL events and synchronisation. Refer to the lecture notes on events and synchronisation for further details.

Host notification and event callback functions

For this section, examine the code Tutorial7a. The example illustrates how to associate events with an enqueuing command, and how to associate callback functions with the events.

In the code, two events are declared:

```
cl::Event kernelEvent, readEvent;
```

When the enqueuing commands are called, the respective events are associated as the last argument of the commands:

```
queue.enqueueTask(kernel, NULL, &kernelEvent);
```

The code in the kernel function simply stores a number of vectors with the value 5.0 into a buffer object in global memory:

```
__kernel void callback(__global float *buffer) {
   float4 five_vector = (float4)(5.0);

   for(int i=0; i<1024; i++) {
      vstore4(five_vector, i, buffer);
   }
}
```

The `readEvent` is associated with the read buffer command:

```
queue.enqueueReadBuffer(resultBuffer, CL_FALSE, 0, sizeof(cl_float) * 4096, &data[0],
       NULL, &readEvent);
```

Note that the `equeueReadBuffer()` command was set to be non-blocking (i.e. second argument is `CL_FALSE`), so it will return immediately after the call. The program will continue while the data transfer occurs. In other words, it will not wait for the transfer to complete before returning from the function call.

The callback functions for the respective events are set using:

```
const char* kernelMsg = "The kernel finished successfully.\n\0";
kernelEvent.setCallback(CL_COMPLETE, &kernel_complete, (void*)kernelMsg);
```

```
	readEvent.setCallback(CL_COMPLETE, &check_data, (void*)&data[0]);
```

The first argument is set to when the event has completed. The second argument is a reference to the respective callback function. The last argument is a reference to the data that will be passed to the callback function.

Note that these were set after the enqueuing commands. Attempting to set a callback function before the associated enqueuing command will result in an error.

The callback functions are defined near the top of the code. The kernel event callback function simply displays a string, whereas the read event callback function checks the data in the memory buffer. The callback functions are defined as:

```
void CL_CALLBACK kernel_complete(cl_event e, cl_int status, void* data)
{
	printf("%s", (char*)data);
}

void CL_CALLBACK check_data(cl_event e, cl_int status, void* data)
{
	cl_bool check;
	cl_float *bufferData;

	bufferData = (cl_float*)data;
	check = CL_TRUE;
	for (int i = 0; i < 4096; i++) {
		if (bufferData[i] != 5.0) {
			check = CL_FALSE;
			break;
		}
	}
	if (check)
		std::cout << "The data is accurate." << std::endl;
	else
		std::cout << "The data is not accurate." << std::endl;
}
```

Command synchronisation with wait lists and user events

The next example in Tutorial7b illustrates how to use wait lists to stall commands, as well as stalling command execution from the host using user events. It also shows an example of creating an out-of-order command queue. For this example, have a look at the code in Tutorial4b.cpp and user_event.cl. We will look at some of the notable lines of code.

Three events are declared (two events and one user event):

```
	cl::Event kernelEvent, readEvent;
	cl::UserEvent userEvent;
```

Followed by a two element array of event waitlist vectors:

```
	std::vector<cl::Event> waitList[2];
```

The code checks whether the device supports out-of-order execution of commands. If it does, the command queue is created by setting the out-of-order flag. If not, a default in-order execution command queue is created:

```
if ((device.getInfo<CL_DEVICE_QUEUE_PROPERTIES>() &
        CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE) != 0)
{
        queue = cl::CommandQueue(context, device,
                CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE);
}
else
{
        queue = cl::CommandQueue(context, device);
}
```

The user event is created and inserted into an event waitlist:

```
userEvent = cl::UserEvent(context);
waitList[0].push_back((cl::Event)userEvent);
```

The following command is called with the waitlist and an event. What this does is stalls the kernel execution until all events in the waitlist have completed. After the command completes, the `kernelEvent` status will be set to `CL_COMPLETE`.

```
queue.enqueueTask(kernel, &waitList[0], &kernelEvent);
```

If you look further down the code, the status of the user event in the waitlist will only change after the user presses a key:

```
getchar();
userEvent.setStatus(CL_COMPLETE);
```

Note that if you set the user event to a negative value, it will abort the commands that are waiting on it, e.g.: `userEvent.setStatus(-1);`

Next, `kernelEvent` is inserted into the second waitlist. The read buffer command is set to be non-blocking, it will wait for events in `waitList[1]` to complete before executing, and when the command completes, it will set the status of `readEvent` to `CL_COMPLETE`.

```
queue.enqueueReadBuffer(dataBuffer, CL_FALSE, 0, sizeof(cl_float) * 4, &data[0],
        &waitList[1], &readEvent);
```

A callback function for `readEvent` is set as follows (can only be set after the preceding command):

```
readEvent.setCallback(CL_COMPLETE, &read_complete, (void*)&data[0]);
```

Compile and run the program.

What happens is:

- Data before kernel execution is displayed.
- The program then waits for the user to press a key.

- Once a key is pressed, the kernel is executed. This sets the kernelEvent object to complete, which will allow the read buffer command to execute.
- Once this command completes the readEvent object will be complete, which will invoke the callback that is associated with the event.
- This in turn, displays the content of the data after kernel execution.

Profiling

Profiling is important when testing performance. The example in Tutorial7c shows a method of obtaining timing information from events. For this example, refer to the code in Tutorial7c.cpp and profile_item.cl.

Note that when the command queue was created, the profiling flag was enabled:

```
queue = cl::CommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE);
```

To perform profiling, an event must be associated with the command to profile. The following data are declared for profiling:

```
cl::Event profileEvent;
cl_ulong timeStart, timeEnd, timeTotal;
```

When the kernel enqueuing command is called, the `profileEvent` is associated with it (the last argument):

```
queue.enqueueNDRangeKernel(kernel, offset, globalSize, cl::NullRange, NULL,
       &profileEvent);
```

The kernel performs some vector calculations on an array of integers. This task is distributed based on the number of work-items given by `NUM_ITEMS`.

The following obtains the start and end time stamps of when the command started and ended its execution. Subtracting the start from the end time will give how long the command took to execute. The `queue.finish()` command blocks the program from continuing until all commands in the command queue have completed:

```
queue.finish();

timeStart = profileEvent.getProfilingInfo<CL_PROFILING_COMMAND_START>();
timeEnd = profileEvent.getProfilingInfo<CL_PROFILING_COMMAND_END>();

timeTotal += timeEnd - timeStart;
```

Note that to obtain more reasonable timing results; the average time over a number of iterations should be taken, rather than the time for a single execution. This is illustrated in the example by running it a number of times based on:

```
#define NUM_ITERATIONS 2000
```

Atomic operations

The example in Tutorial7d shows an example of OpenCL atomic operations. The code is provided in Tutorial7d.cpp and atomic.cl. The following is the kernel function:

```
__kernel void atomic(__global int* x) {
```

What the kernel does is declares int a and b in local memory (i.e. shared between work-items within a work group):

```
   __local int a, b;
```

It gets the work-item's local ID:

```
   int lid = get_local_id(0);
```

Then checks whether the work-item is the first work-item in the work-group. If it is, the work-item sets a and b to 0:

```
   if(lid == 0)
   {
         a = 0;
         b = 0;
   }
```

In each work-item, two types of increments are run. A normal increment and an atomic increment:

```
   a++;
   atomic_inc(&b);
```

Finally in each work-item, the data is written into global memory:

```
   x[0] = a;
   x[1] = b;
}
```

The kernel was enqueued as follows:

```
       cl::NDRange offset(0);
       cl::NDRange globalSize(8);
       cl::NDRange localSize(4);

       queue.enqueueNDRangeKernel(kernel, offset, globalSize, localSize);
```

This will create 8 work-items in 2 work-groups. Each work-group will have 4 work-items.

Atomic operations are operations that cannot be interrupted. Hence, the same section of memory cannot be accessed at the same time. In the example given above, the `a++` command will be executed on 4 work-items in parallel, the result of this will be device and platform specific (i.e. results will be different depending on the platform and device). Whereas the `atomic_inc` atomic operation will force `b` to only be accessible by one of the work-items at any time. This means that the 4 work-items can only access `b` one at a time. This results in `b` being consistent on different platforms and devices.

Try this yourself

To test performance, try profiling the example in the profiling example by changing the size of:

```
#define NUM_ITEMS 512
```

Changing this changes the number of work-items. Plot a graph showing the performance in relation to the number of work-items.

References

Among others, the material in this tutorial was sourced from:

- M. Scarpino, “OpenCL in Action: How to Accelerate Graphics and Computation,” Manning