OpenCL

Events, Synchronization, and Profiling

Matt Sellitto

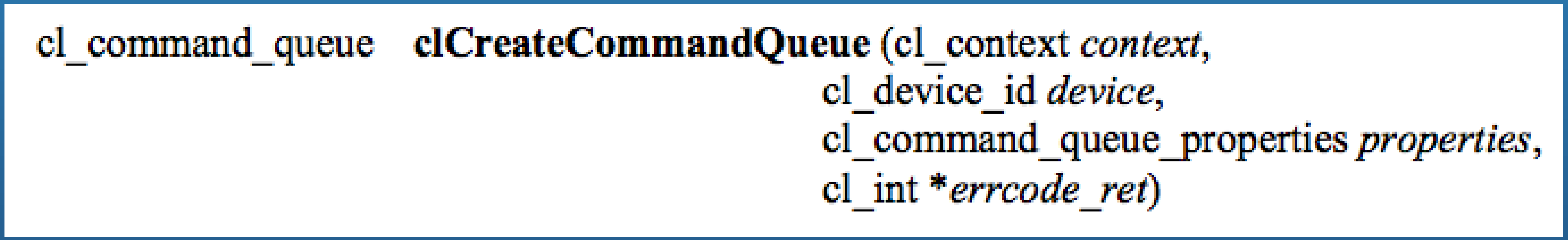
Dana Schaa

Northeastern University

NUCAR

Command Queues

- OpenCL command queues are used to submit work to an OpenCL device.
- Two main types of command queues
  - In Order Queue
  - Out of Order Queue



```
cl_command_queue   clCreateCommandQueue (cl_context context,
                                         cl_device_id device,
                                         cl_command_queue_properties properties,
                                         cl_int *errcode_ret)
```

- In our previous programs we passed 0 in for the "properties" argument when creating a new command queue, all queues are in-order by default.
- Passing in CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE for the properties makes the command queue an out-of-order queue.

Command Queue - Behavior

- All OpenCL API functions that begin with clEnque are ones that put something on a command queue.
- When a commands queue is "in-order" it will always finish commands one after another in the order they are added to the queue.
- If a command queue is "out-of-order" it may execute commands in the queue by over-lapping their execution or re-ordering them.

Command Queue
– In-order Execution

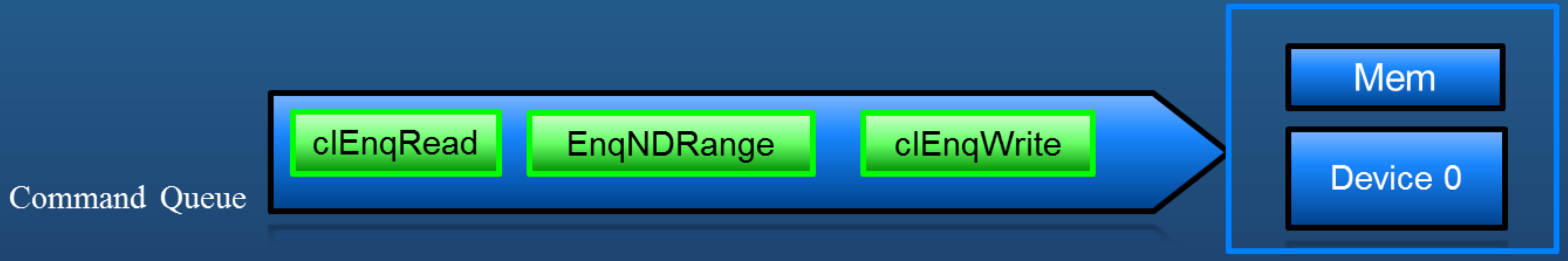
- In an in-order command queue, each command executes after the previous one has finished.
  - For the set of commands shown, the read from the device would start after the kernel call has finished

- Memory transactions have consistent view

- In the vectorAddition program, we first enqueued some writes, then we executed the kernel, and then we read the result back to the host:

Commands To Submit

```
clEnqueueWriteBuffer (queue , d_ip, CL_TRUE, 0, mem_size, (void *)ip, 0, NULL,  NULL)
clEnqueueNDRangeKernel (queue, kernel0, 2,0, global_work_size,   local_work_size, 0,NULL,NULL)
clEnqueueReadBuffer( context, d_op,  CL_TRUE, 0, mem_size,  (void *) op, NULL, NULL,NULL);
```

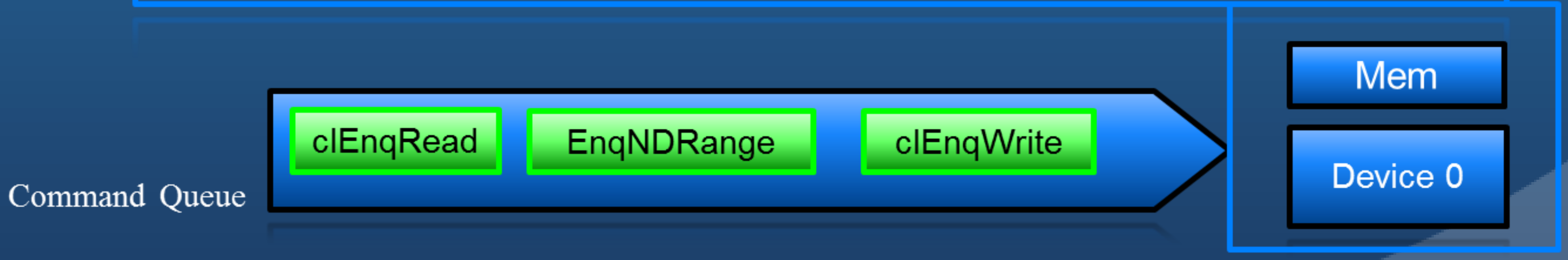


Command Queue
– In-order Execution - Synchronization

- When using a single in-order queue you do not have to worry about dependencies between commands in the queue.
- <u>**However**</u>, you must remember that enqueueing a command on a queue with clEnqueXXXX does not mean that the command is finished executing when the API function returns on the host side.
    - The command is just put into the queue, not executed!
- Exception: The cl{Read|Write}{Buffer|Image} functions take a boolean argument to determine weather they are blocking or non-blocking.
    - That is how we knew it was OK to check the output vector on the host after the clEnqueReadBuffer API function returned.

Code:

```
clEnqueueWriteBuffer (queue , d_ip, CL_TRUE, 0, mem_size, (void *)ip, 0, NULL,  NULL)
clEnqueueNDRangeKernel (queue, kernel0, 2,0, global_work_size,   local_work_size, 0,NULL,NULL)
clEnqueueReadBuffer( context, d_op,  CL_TRUE, 0, mem_size,  (void *) op, NULL, NULL,NULL);
// Use host data in pointer *op here
```



Command Queue
– In-order Execution - Synchronization

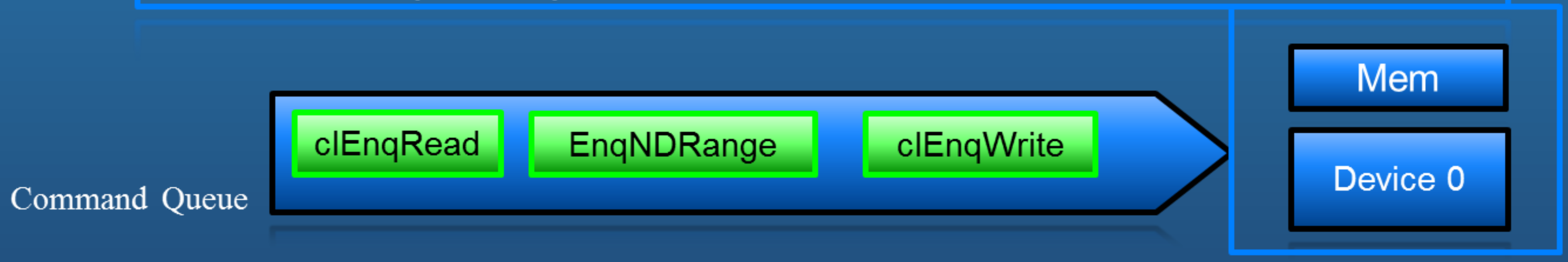
- OpenCL provides API functions for synchronization. One such function is clFinish:

```
cl_int clFinish(cl_command_queue command_queue)
```

- clFinish blocks until all commands on a given command queue are finished.
- We could use this API function to determine when we could start to read host side data after a clEnqueueRead if the read was non-blocking.

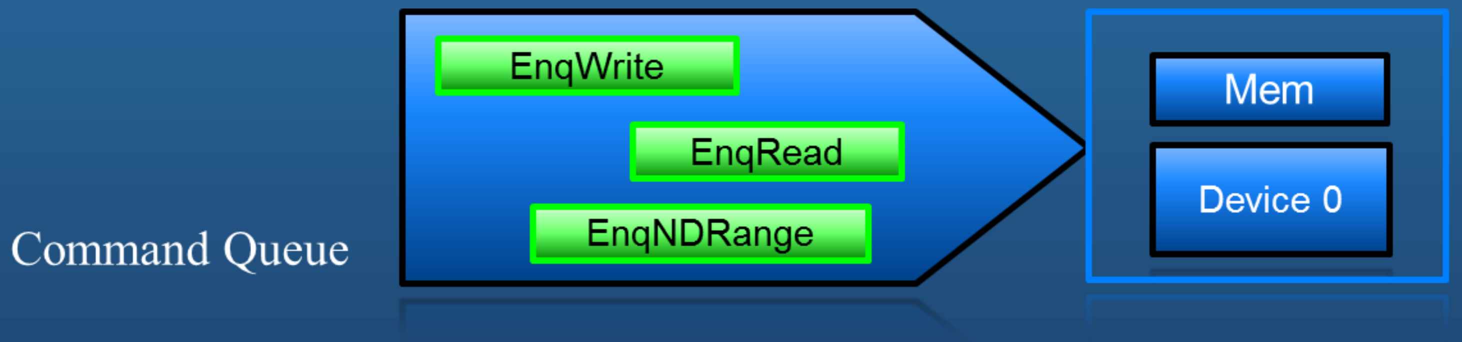
Code:

```
clEnqueueWriteBuffer (queue , d_ip, CL_FALSE, 0, mem_size, (void *)ip, 0, NULL,  NULL)
clEnqueueNDRangeKernel (queue, kernel0, 2,0, global_work_size,   local_work_size, 0,NULL,NULL)
clEnqueueReadBuffer( context, d_op,  CL_FALSE, 0, mem_size,  (void *) op, NULL, NULL,NULL);
clFinish(queue)
// Use host data in pointer *op here
```



Command Queue
- Out of Order Execution

- If using an out-of-order command queue you must take any dependencies into account between commands.
- You do not know what order they may execute and some may even overlap in execution!
- We need more control over when commands are executed by the OpenCL device…



Command Queue
- Out of Order Execution - Events

- OpenCL Event objects can be used to determine a command's status and to establish dependencies between commands.
- Events are only valid inside a particular context and not across multiple contexts.
- Each clEnqueueXXXX() function returns an event object for that command:

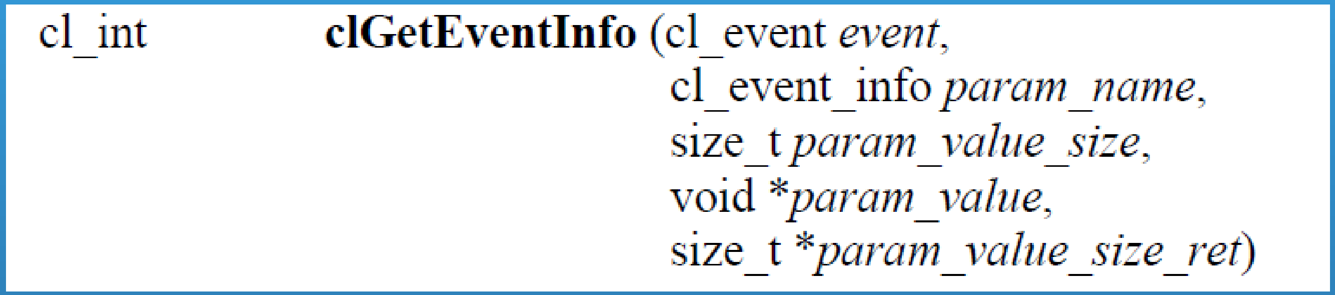
```
clEnqueueWriteBuffer(…, cl_event *event)

clEnqueueNDRangeKernel(…, cl_event *event)
```

- These are the objects that are used to coordinate the execution of commands and to query their status.

Event Info

- OpenCL Event objects can be queried with clGetEventInfo() to get information about the event:



```
cl_int      clGetEventInfo (cl_event event,
                            cl_event_info param_name,
                            size_t param_value_size,
                            void *param_value,
                            size_t *param_value_size_ret)
```

- Using clGetEventInfo() you can query an events current execution status (submitted, running, complete), type of command, context that it was in, etc

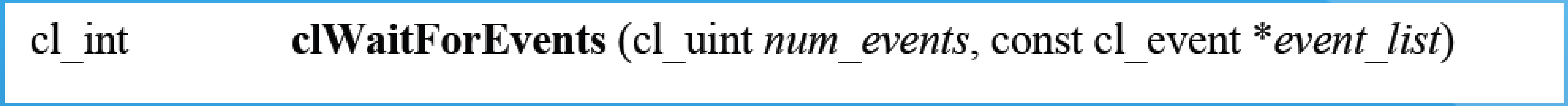
Waiting for an Event

- You can wait (block) for an event to complete with clWaitForEvents():

```
cl_int          clWaitForEvents (cl_uint num_events, const cl_event *event_list)
```

- This function takes an array of event objects and will block until all the given events reach the CL_COMPLETE status.

Waiting for an Event



cl_int **clWaitForEvents** (cl_uint *num_events*, const cl_event **event_list*)

- If we did a non-blocking read of the output data in vector addition, we could use this API function to know when the read was complete and that it was safe to use the host-side data pointer.
- To do:
  - Modify the vectorAddition program to use non-blocking reads and writes.
  - Use the clWaitForEvents() function to know when it is safe to use the host side data after the clEnqueueReadBuffer() Note: We are still using an in-order command queue.

Waiting for an Event

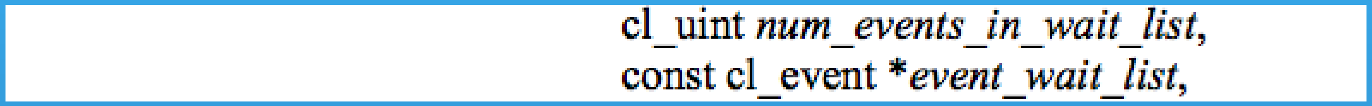
cl_int **clWaitForEvents** (cl_uint *num_events*, const cl_event **event_list*)

```
clEnqueueReadBuffer( context, d_C,  CL_FALSE, 0, datasize, C, 0,
NULL, &read_event);

clWaitForEvents(1, &read_event); // block until read cmd finishes

// Use host data in pointer *op here
```

Event Wait Lists

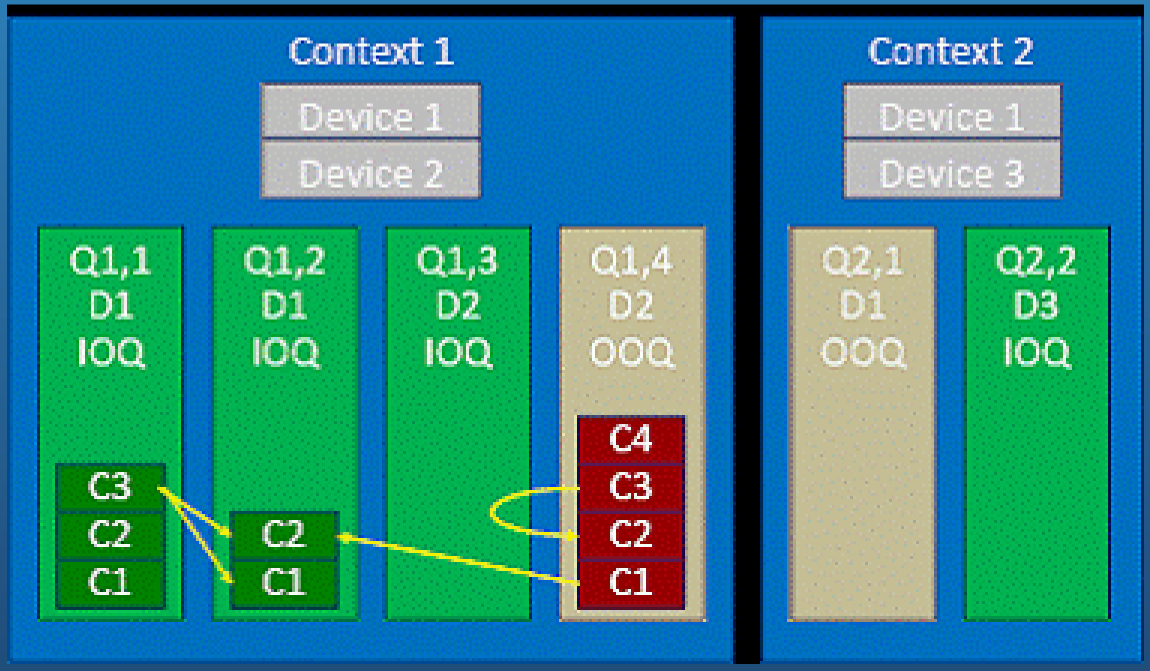


```
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
```

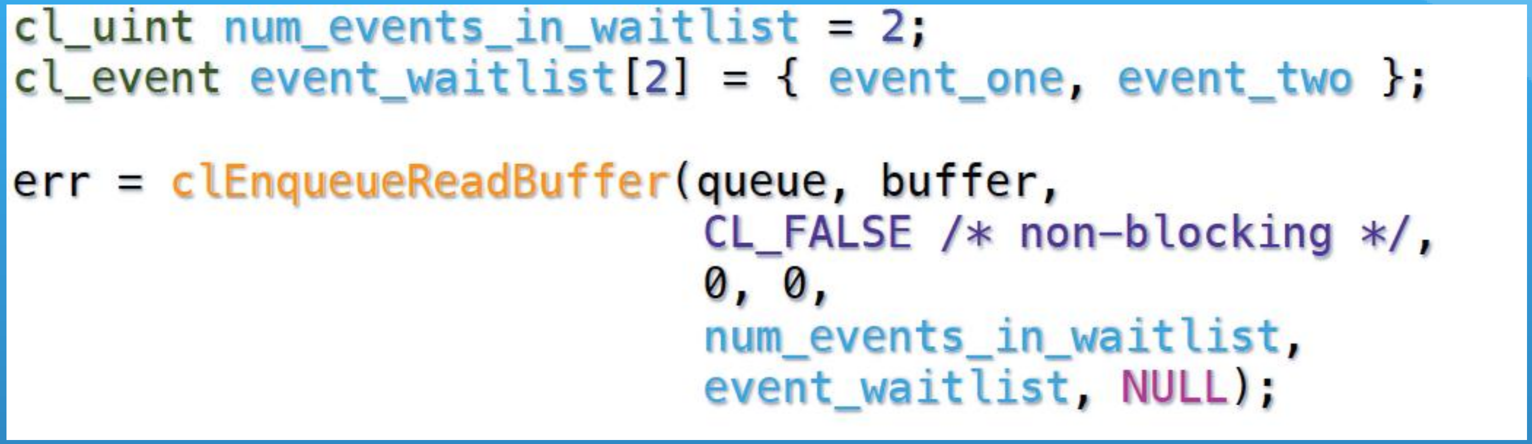
- All clEnqueueXXXX commands take an event_wait_list argument.
- This argument tells the command queue to not let this command execute until all commands that correspond to the events in the event_wait_list have been completed.
- These event wait lists can be used to create a sort of dependency graph between OpenCL commands when using an out-of-order queue (or even multiple queues).

Event Wait Lists

- Dependency graph of different commands using:
  - 2 contexts
  - 3 devices
  - 6 command queues



Event Wait Lists - Example



```
cl_uint num_events_in_waitlist = 2;
cl_event event_waitlist[2] = { event_one, event_two };

err = clEnqueueReadBuffer(queue, buffer,
                          CL_FALSE /* non-blocking */,
                          0, 0,
                          num_events_in_waitlist,
                          event_waitlist, NULL);
```

- This read will be enqueued on to the command queue and is still non-blocking, but the read will not actually happen until event_one and event_two have been completed.
- To do:
  - Modify the vectorAddition program to use an out-of-order queue.
  - Use event_wait_lists to synchronize between different commands.

Other Synchronization Functions

- clEnqueueBarrier() :

```
cl_int          clEnqueueBarrier (cl_command_queue command_queue)
```

- Places a barrier into a command queue. All commands that were enqueued before the barrier must complete before any commands enqueued after the barrier start executing.

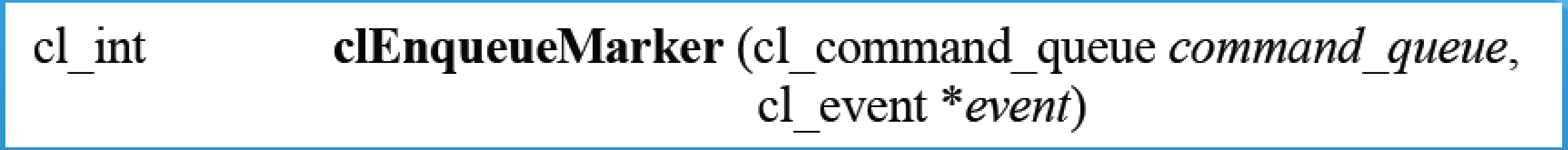
- clEnqueueWaitForEvents():

```
cl_int          clEnqueueWaitForEvents (cl_command_queue command_queue,
                                        cl_uint num_events,
                                        const cl_event *event_list)
```

- All events in the event_list must complete before any commands enqueued after the barrier begin executing.

Other Synchronization Functions

- clEnqueueMarker()



```
cl_int    clEnqueueMarker (cl_command_queue command_queue,
                           cl_event *event)
```

- Enqueues a marker command on to the command_queue. The marker command is not completed until all commands enqueued before it have completed.
  - The marker command returns an event which can be waited on, i.e. this event can be waited on to ensure that all commands which have been queued before the market command have been completed.

Profiling

- Events can also be used to profile and time your OpenCL kernels.
- Profiling of OpenCL programs using events has to be enabled explicitly when creating a command queue
    - CL_QUEUE_PROFILING_ENABLE flag must be set in the properties argument in clCreateCommandQueue()
    - May have a small performance penalty.

Profiling - Capabilities

- Using OpenCL Events we can:
  - time execution of clEnqueue* calls like kernel execution or explicit data transfers
  - observe overhead and time consumed by a kernel in the command queue versus actually executing
  - profile an application to understand an execution flow

Capturing Event Profiling Information

```
cl_int clGetEventProfilingInfo (
              cl_event  event,                        //event object
              cl_profiling_info  param_name,          //Type of data of event
              size_t  param_value_size,               //size of memory pointed to by param_value
              void *  param_value,                    //Pointer to returned timestamp
              size_t * param_value_size_ret)           //size of data copied to param_value
```

- clGetEventProfilingInfo() allows us to query a cl_event to get required counter values
- Timing information returned as cl_ulong data types
  - Returns device time counter in nanoseconds

Capturing Event Profiling Information

```
cl_int clGetEventProfilingInfo (
          cl_event  event,                    //event object
          cl_profiling_info  param_name,      //Type of data of event
          size_t  param_value_size,           //size of memory pointed to by param_value
          void *  param_value,                //Pointer to returned timestamp
          size_t * param_value_size_ret)      //size of data copied to param_value
```

- Table shows event types described using `cl_profiling_info` enumerated type

Param_name	Returned Information
CL_PROFILING_COMMAND_QUEUED	The time at which the command is enqueued in a command-queue by the host.
CL_PROFILING_COMMAND_SUBMIT	The time at which the command is submitted by the host to the device associated with the command queue.
CL_PROFILING_COMMAND_START	The time at which the command starts execution on device.
CL_PROFILING_COMMAND_END	The time at which the command has finished execution on device.

Profiling – How To

- OpenCL events can easily be used for timing durations of kernels.
- This method is reliable for performance optimizations since it uses counters from the device
- By taking differences of the start and end timestamps we are discounting overheads like time spent in the command queue

```
clGetEventProfilingInfo( event_time,
CL_PROFILING_COMMAND_START,
sizeof(cl_ulong), &starttime, NULL);
```

```
clGetEventProfilingInfo(event_time,
CL_PROFILING_COMMAND_END,
sizeof(cl_ulong), &endtime, NULL);
```

```
unsigned long elapsed =
(unsigned long)(endtime - starttime);
```

Profiling - How To

- Before getting timing information, we must make sure that the events we are interested in have completed.
- There are different ways of waiting for events:
  - `clWaitForEvents(numEvents, eventlist)`
  - `clFinish(commandQueue)`
- Timer resolution of the timeingcan be obtained from the flag: `CL_DEVICE_PROFILING_TIMER_RESOLUTION` when calling `clGetDeviceInfo`()

How Profiling Can Help

- A heterogeneous application can have multiple kernels and a large amount of host device IO
- Questions that can be answered by profiling using OpenCL events
  - We need to know which kernel to optimize when multiple kernels take similar time ?
    - Small kernels that may be called multiple times vs. large slow complicated kernel ?
  - Are the kernels spending too much time in queues ?
  - Understand proportion between execution time and setup time for an application
  - How much does host device IO matter ?
- By profiling an application with minimum overhead and no extra synchronization, most of the above questions can be answered

To Do

- In our matrixMultiply program we have two kernels that run, one is a basic naïve matrix multiply and one is a more optimized one using local memory. Modify the program to profile these two kernels and print the results to the screen.