OpenCL Events

Tim Matson, Intel

(OpenCL tutorial, HiPEAC'11)

Events

- **An event is an object that communicates the status of commands in OpenCL … legal values for an event:**
  - **CL_QUEUED**: command has been enqueued.
  - **CL_SUBMITED**: command has been submitted to the compute device
  - **CL_RUNNING**: compute device is executing the command
  - **CL_COMPLETE**: command has completed
  - **ERROR_CODE**: a negative value, indicates an error condition occurred.
- **Can query the value of an event from the host … for example to track the progress of a command.**

```
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)
```

- **Examples:**
  - **CL_EVENT_CONTEXT**
  - **CL_EVENT_COMMAND_EXECUTION_STATUS**
  - **CL_EVENT_COMMAND_TYPE**

Generating and consuming events

- Consider the command to enqueue a kernel. The last three arguments optionally expose events (NULL otherwise).

```
cl_int clEnqueueNDRangeKernel (
    cl_command_queue command_queue,
    cl_kernel kernel,     cl_uint work_dim,
    const size_t *global_work_offset,
    const size_t *global_work_size,
    const size_t *local_work_size,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event)
```

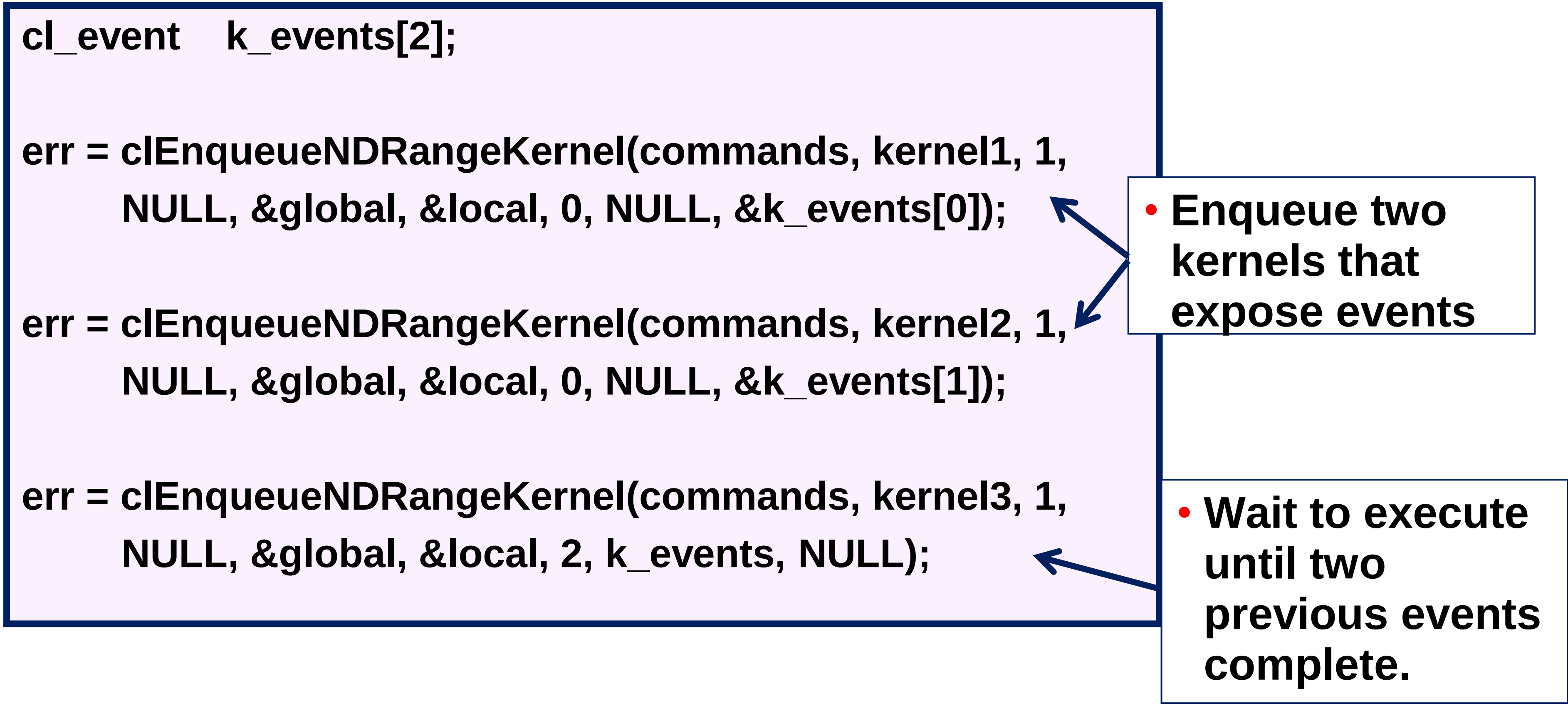
- Number of events this command is waiting to complete before executing

- Array of pointers to the events being waited upon … Command queue and events must share a context.

- Pointer to an event object generated by this command.

Event: basic event usage

- Events can be used to impose order constraints on kernel execution.
- Very useful with out of order queues.



```
cl_event    k_events[2];

err = clEnqueueNDRangeKernel(commands, kernel1, 1,
        NULL, &global, &local, 0, NULL, &k_events[0]);

err = clEnqueueNDRangeKernel(commands, kernel2, 1,
        NULL, &global, &local, 0, NULL, &k_events[1]);

err = clEnqueueNDRangeKernel(commands, kernel3, 1,
        NULL, &global, &local, 2, k_events, NULL);
```

- Enqueue two kernels that expose events

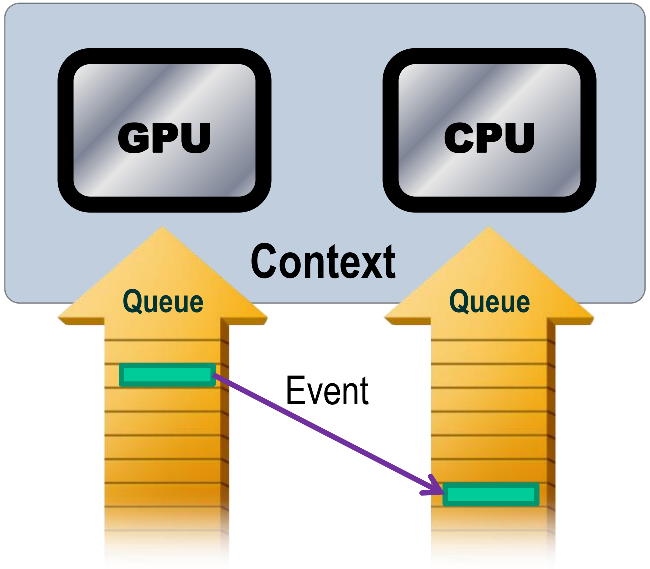
- Wait to execute until two previous events complete.

Why Events? Won't a barrier do?

- A barrier defines a synchronization point … commands following a barrier wait to execute until all prior enqueued commands complete

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

- Events provide fine grained control … this can really matter with an out of order queue.



- Events work between commands in different queues … as long as they share a context!

- Events convey more information than a barrier ... Provide info on state of a command, not just weather its complete or not.

Host code influencing commands: User events

- “user code” running on a host thread can generate event objects

```
cl_event clCreateUserEvent (
          cl_context context,
          cl_int *errcode_ret)
```

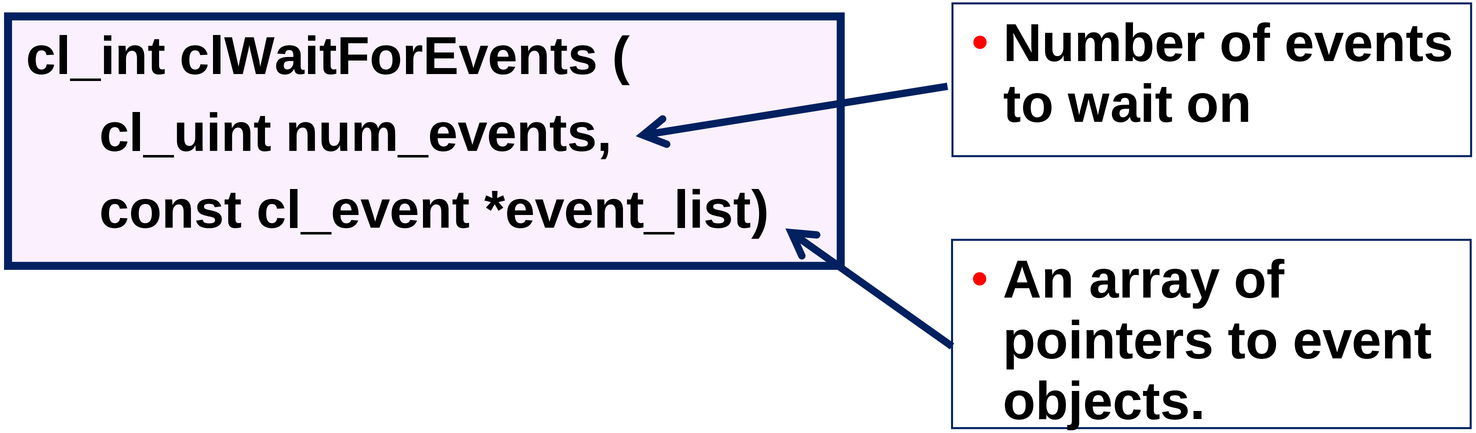
- Created with value CL_SUBMITTED.
- It’s just another event to enqueued commands.
- Can set the event to one of the legal event values

```
cl_int clSetUserEventStatus (
        cl_event event,
        cl_int execution_status)
```

- Example use case: Queue up block of commands that wait on user input to finalize state of memory objects before proceeding.

Commands Influencing host code

- A thread running on the host can pause waiting on a list of events to complete. This is done with the function:



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

- Number of events to wait on

- An array of pointers to event objects.

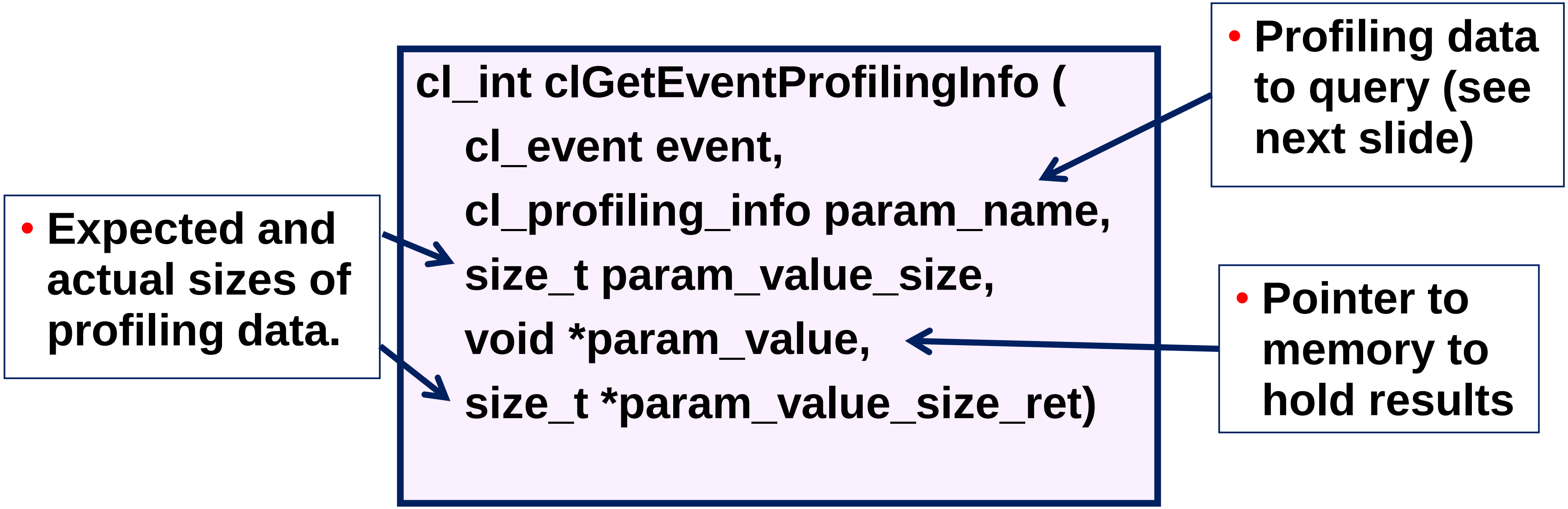
- Example use case: Host code waiting for an event to complete before extracting information from the event.

Profiling with Events

- **OpenCL is a performance oriented language … Hence performance analysis is an essential part of OpenCL programming.**
- **The OpenCL specification defines a portable way to collect profiling data.**
- **Can be used with most commands placed on the command queue … includes:**
  - Commands to read, write, map or copy memory objects
  - Commands to enqueue kernels, tasks, and native kernels
  - Commands to Acquire or Release OpenGL objects
- **Profiling works by turning an event into an opaque object to hold timing data.**

Using the Profiling interface

- Profiling is enabled when a queue is created with the CL_QUEUE_PROFILING_ENABLE flag is set.
- When profiling is enabled, the following function is used to extract the timing data



```
cl_int clGetEventProfilingInfo (
    cl_event event,
    cl_profiling_info param_name,
    size_t param_value_size,
    void *param_value,
    size_t *param_value_size_ret)
```

- Profiling data to query (see next slide)

- Expected and actual sizes of profiling data.

- Pointer to memory to hold results

cl_profiling_info values

- **CL_PROFILING_COMMAND_QUEUED**
  - the device time in nanoseconds when the command is enqueued in a command-queue by the host. (cl_ulong)
- **CL_PROFILING_COMMAND_SUBMIT**
  - the device time in nanoseconds when the command is submited to compute device. (cl_ulong)
- **CL_PROFILING_COMMAND_START**
  - the device time in nanoseconds when the command starts execution on the device. (cl_ulong)
- **CL_PROFILING_COMMAND_END**
  - the device time in nanoseconds when the command has finished execution on the device. (cl_ulong)

Profiling Example

```
cl_event prof_event;
cl_command_queue comm;

comm = clCreateCommandQueue(
       context, device_id,
      CL_QUEUE_PROFILING_ENABLE,
       &err);

err = clEnqueueNDRangeKernel(
         comm, kernel,
         nd, NULL, global, NULL,
         0, NULL, prof_event);

clFinish(comm);
err = clWaitForEvents(1, &prof_event );
```

```
cl_ulong start_time, end_time;
size_t return_bytes;

err = clGetEventProfilingInfo(
   prof_event,
    CL_PROFILING_COMMAND_QUEUED,
    sizeof(cl_ulong),
    &start_time,
    &return_bytes);

err = clGetEventProfilingInfo(
    prof_event,
    CL_PROFILING_COMMAND_END,
    sizeof(cl_ulong),
    &end_time,
    &return_bytes);

run_time =(double)(end_time - start_time);
```

Events inside Kernels … Async. copy

```
// A, B, C kernel args … global  buffers.
// Bwrk is a local buffer

for(k=0;k<Pdim;k++)
        Awrk[k] = A[i*Ndim+k];

for(j=0;j<Mdim;j++){
   event_t ev_cp  = async_work_group_copy(
      (__local float*) Bwrk, (__global float*) B,
      (size_t) Pdim, (event_t) 0);

   wait_group_events(1, &ev_cp);

   for(k=0, tmp= 0.0;k<Pdim;k++)
         tmp  += Awrk[k] *  Bwrk[k];
   C[i*Ndim+j] = tmp;
}
```

- Compute a row of C = A * B:
  - 1 A col. per work-item
  - Work group shares rows of B

- Start an async. copy for row of B returning an event to track progress.

- Wait for async. copy to complete before proceeding.

- Compute element of C using A from private memory and B from local memory.