Synchronism and Events

AdC - 2025

Command Queues

- Command queues are used to submit work to a device
- Two main types of command queues
  - In Order Queue
  - Out of Order Queue
- Used to measure the performance of an application as a whole.
- This necessitates understanding of OpenCL synchronization techniques and events

Command Queues

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

creates a command-queue on a specific device.

context must be a valid OpenCL context.

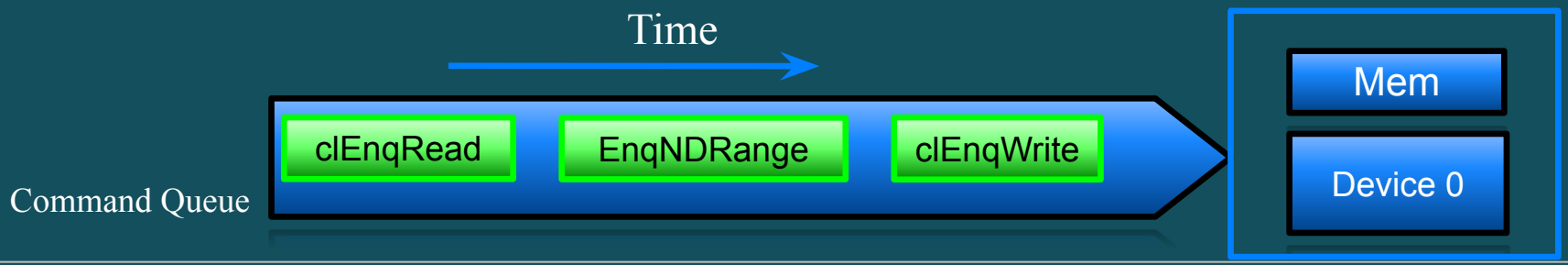
Command-Queue Properties	Description
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE	Determines whether the commands queued in the command-queue are executed in-order or out-of-order. If set, the commands in the command-queue are executed out-of-order. Otherwise, commands are executed in-order. For a detailed description about CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, refer to *section 5.11*.
CL_QUEUE_PROFILING_ENABLE	Enable or disable profiling of commands in the command-queue. If set, the profiling of commands is enabled. Otherwise profiling of commands is disabled.

Documentación PyOpenCL

In-Order Execution

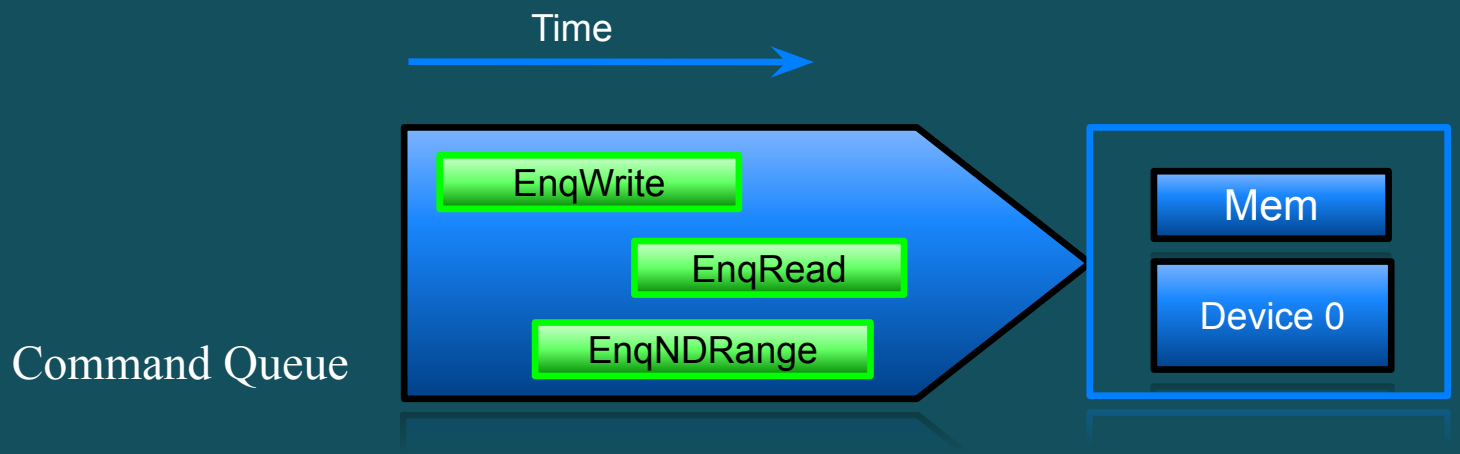
- In an in-order command queue, each command executes after the previous one has finished
  - For the set of commands shown, the read from the device would start after the kernel call has finished
- Memory transactions have consistent view

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);



Out-of-Order Execution

- In an out-of-order command queue, commands are executed as soon as possible, without any ordering guarantees
- All memory operations occur in single memory pool
- Out-of-order queues result in memory transactions that will overlap and clobber data without some form of synchronization
- The commands discussed in the previous slide could execute in any order on device



Host Synchronization

- Synchronization is required if we use an out-of-order command queue or multiple command queues
- Synchronization is restricted to within a context
- This is similar to the fact that it is not possible to share data between multiple contexts without explicit copying
- The proceeding discussion of synchronization is applicable to any OpenCL device (CPU, GPU, etc)

Command Queue Control

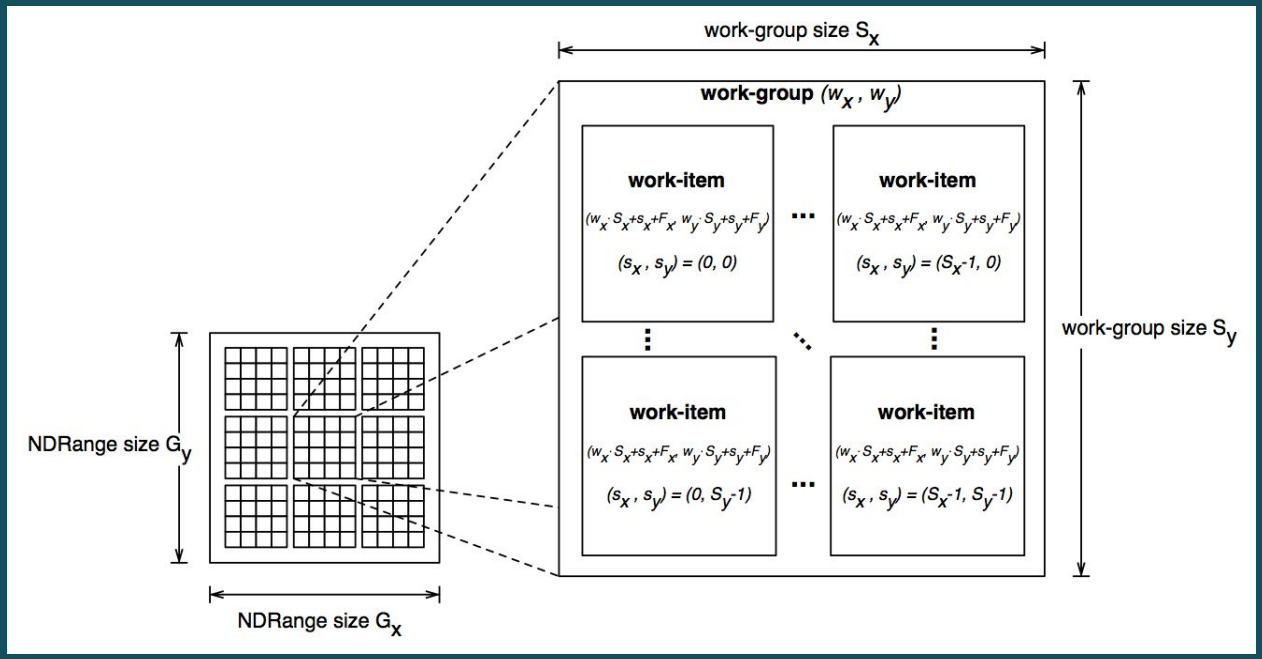
- Command queue synchronization methods work on a per-queue basis
- **Finish:** clFinish(`cl_commandqueue`)
  - Waits for all commands in the command queue to complete before proceeding (**host blocks on this call**)
- **Barrier:** clEnqueueBarrier(`cl_commandqueue`)
  - Enqueue a synchronization point that ensures all prior commands in a queue have completed before any further commands execute

Synchronization for clEnqueue Functions

- Functions like clEnqueueReadBuffer and clEnqueueWriteBuffer have a boolean parameter to determine if the function is blocking
  - This provides a blocking construct that can be invoked to block the host
- If blocking is **TRUE**, OpenCL enqueues the operation using the host pointer in the command-queue
  - Host pointer **can** be reused by the application after the enqueue call returns
- If blocking is **FALSE**, OpenCL will use the host pointer parameter to perform a non-blocking read/write **and returns immediately**
  - Host pointer **cannot** be reused safely by the application after the call returns
  - Event handle returned by clEnqueue* operations can be used to check if the non-blocking operation has completed

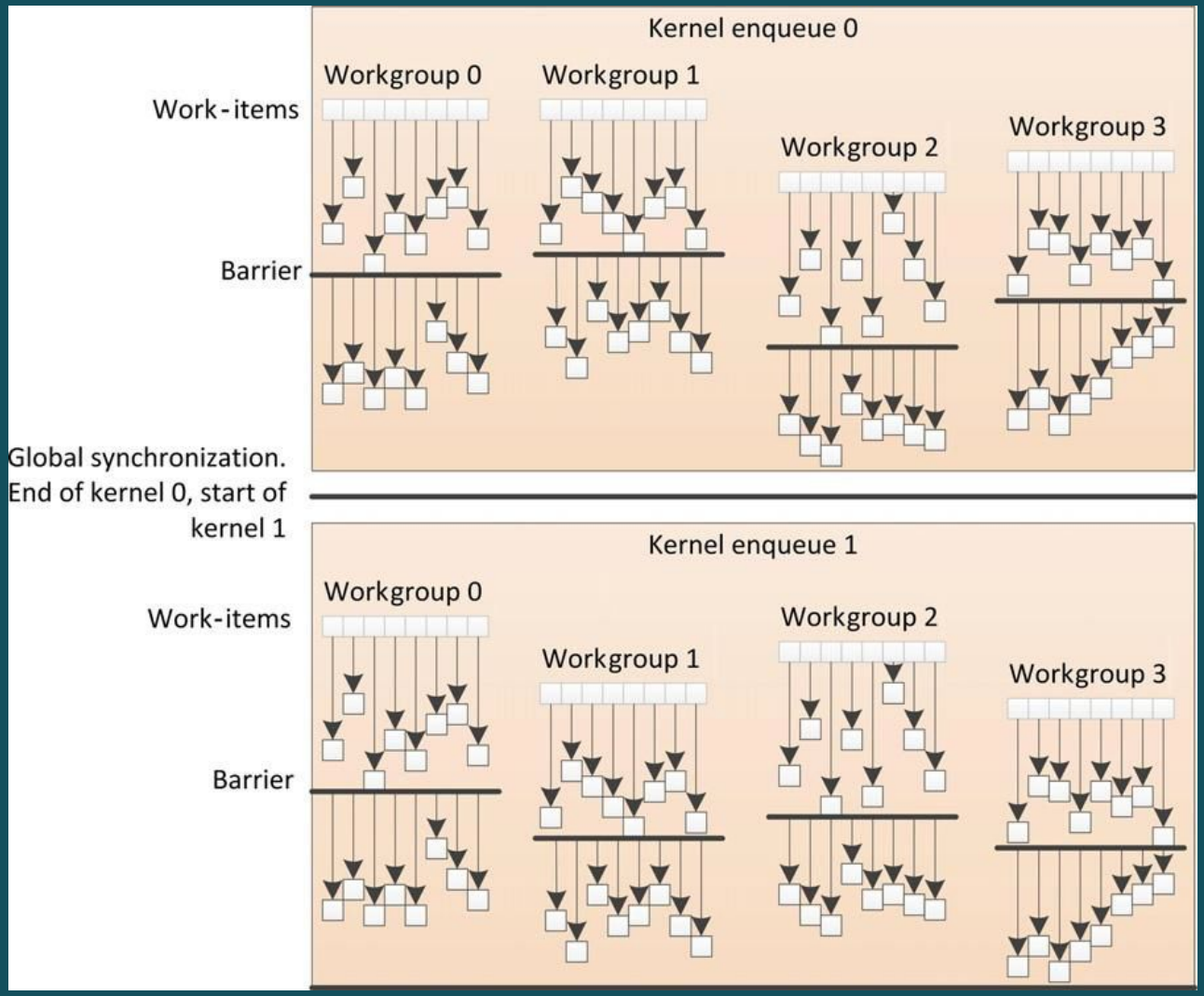
Kernel Space

- In Kernel Space: **"Device Synchronization model"**



Device Synchronism

Kernel Synchronism:
Barriers



Device Synchronism

```
// Kernel
__kernel void simpleKernel(
      __global float *a,
      __global float *b,
    __local float *l )
{
   l[get_local_id(0)] = a[get_global_id(0)];

   barrier(CLK_LOCAL_MEM_FENCE);

   unsigned int otherAddress =
       (get_local_id(0) + 1) % get_local_size(0);

   b[get_local_id(0)] = l[get_local_id(0)] + l[otherAddress];
}
```

OpenCL Events

- **Explicit** synchronization is required for
  - Out-of-order command queues
  - Multiple command queues
- OpenCL events are data-types defined by the specification for storing timing information returned by the device

OpenCL Events

- Profiling of OpenCL programs using events has to be enabled explicitly when creating a command queue
  - `CL_QUEUE_PROFILING_ENABLE` flag must be set
  - `command_queue = clCreateCommandQueue(context, devices[deviceUsed], CL_QUEUE_PROFILING_ENABLE, &err);`
  - Keeping track of events may slow down execution
- A handle to store event information can be passed for all clEnqueue* commands
  - When commands such as `clEnqueueNDRangeKernel` and `clEnqueueReadBuffer` are invoked timing information is recorded at the passed address

OpenCL Events

```
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)
```

OpenCL Events

```
cl_int  clEnqueueReadBuffer (cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_read,
                             size_t offset,
                             size_t size,
                             void *ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)

cl_int  clEnqueueWriteBuffer (cl_command_queue command_queue,
                              cl_mem buffer,
                              cl_bool blocking_write,
                              size_t offset,
                              size_t size,
                              const void *ptr,
                              cl_uint num_events_in_wait_list,
                              const cl_event *event_wait_list,
                              cl_event *event)
```

Uses of OpenCL Events

- Using OpenCL Events we can:
  - time execution of clEnqueue* calls like kernel execution or explicit data transfers
  - use the events from OpenCL to schedule asynchronous data transfers between host and device
  - observe overhead and time consumed by a kernel in the command queue versus actually executing
- **Note:** OpenCL event handling can be done in a consistent manner on both CPU and GPU for AMD and NVIDIA's implementations

Capturing Event 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 `cl_event` to get required counter values
- Timing information returned as `cl_ulong` data types
  - Returns device time counter in nanoseconds

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

Event Type	Description
CL_PROFILING_COMMAND_QUEUED	Command is enqueued in a command-queue by the host.
CL_PROFILING_COMMAND_SUBMIT	Command is submitted by the host to the device associated with the command queue.
CL_PROFILING_COMMAND_START	Command starts execution on device.
CL_PROFILING_COMMAND_END	Command has finished execution on device.

Profiling with Event Information

- 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 can be obtained from the flag `CL_DEVICE_PROFILING_TIMER_RESOLUTION` when calling `clGetDeviceInfo()`

Event Profiling Information

```
command_queue = clCreateCommandQueue(context, devices[deviceUsed], CL_QUEUE_PROFILING_ENABLE, &err);

//Ensure to have executed all enqueued tasks
clFinish(command_queue);

//Launch Kernel linked to an event
err = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, workGroupSize, NULL, 0, NULL, &event);

//Ensure kernel execution is finished
clWaitForEvents(1 , &event);

//Get the Profiling data
cl_ulong time_start, time_end;
double total_time;

clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(time_start), &time_start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(time_end), &time_end, NULL);
total_time = time_end - time_start;
printf("\nExecution time in milliseconds = %0.3f ms\n", (total_time / 1000000.0) );
```

Using Events for Timing

- 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

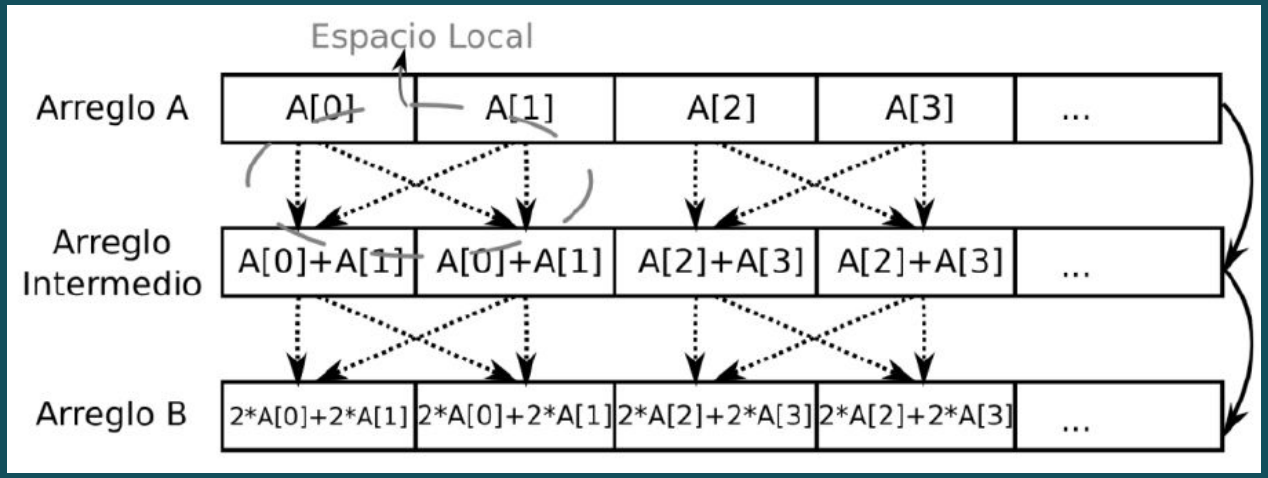
```
clGetEventProfilingInfo( event_time,
CL_PROFILING_COMMAND_START,
sizeof(cl_ulong), &starttime, NULL);
```

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

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

Ejercicio 3

Escribir un programa que realice dos veces la suma de los componentes vecinos de un arreglo ‘A’ como se muestra en la siguiente figura:



Utilizar los dos tipos de sincronismo: al nivel del host y al nivel del kernel.