Introduction of OpenCL and GPU Computing
A kernel is a program that is duplicated on compute devices, and the many instances of the kernel run in parallel to process a data array.

In another words, OpenCL Kernel is a code that will be performed on GPU.

The compute shader in OpenGL is a type of kernel that is executed on the 3D graphics pipeline.

The compute kernel in OpenCL is not part of the 3D graphics pipeline.
NDRange and work items

- NDRange is to specify the number of work items on a compute device.
- Work items are divided into work groups.
- Work group divided into wavefronts.
- Wavefront is a group of work items.
Threads can determine their global ID in each dimension

- `get_global_id(dim)`
- `get_global_size(dim)`
- Example ➔ `get_global_id(0) = row, get_global_id(1) = column`

Or they can determine their work-group ID and ID within the workgroup

- `get_group_id(dim)`
- `get_num_groups(dim)`
- `get_local_id(dim)`
- `get_local_size(dim)`
- Example ➔ `get_global_size(0) == get_num_groups(0) * get_local_size(0)`
# Thread structure (2)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint get_work_dim()</code></td>
<td>The number of dimensions</td>
</tr>
<tr>
<td><code>get_global_id(uint dim)</code></td>
<td>The ID of the current work-item ([0, WI)) in dimension (\text{dim})</td>
</tr>
<tr>
<td><code>get_global_size(uint dim)</code></td>
<td>The total number of work-items ((WI)) in dimension (\text{dim})</td>
</tr>
<tr>
<td><code>get_global_offset(uint dim)</code></td>
<td>The offset as specified in the <code>enqueueNDRangeKernel</code> API in dimension (\text{dim})</td>
</tr>
<tr>
<td><code>get_group_id(uint dim)</code></td>
<td>The ID of the current work-group ([0, WG)) in dimension (\text{dim})</td>
</tr>
<tr>
<td><code>get_local_id(uint dim)</code></td>
<td>The ID of the work-item within the work-group ([0, WI/WG)) in dimension (\text{dim})</td>
</tr>
<tr>
<td><code>get_local_size(uint dim)</code></td>
<td>The number of work-items per work-group (= WI/WG) in dimension (\text{dim})</td>
</tr>
<tr>
<td><code>get_num_groups(uint dim)</code></td>
<td>The total number of work-groups ((WG)) in dimension (\text{dim})</td>
</tr>
</tbody>
</table>
**Memory model(1)**

- **Buffer object** is a block of memory allocated by an OpenCL program. Usually it stores the input/output data that will be processed/generated by work items.

- **Constant memory** is part of the compute device memory and is read-only.

- **Global memory** is part of the compute device memory. Any work item can read/write any location within the global memory.
  - To find out the amount of global memory, call `clGetDeviceInfo()`
Memory model(2)

- **Host memory** is the memory attached to the host device. Usually, it is the computer's main memory.
- **Local memory** accessible to all work-items (threads) within a workgroup
- **Private memory** can only be accessed by one work item.
  - A kernel program can allocate a variable private memory by using the qualifier `__private`. 
One of the most difficult tasks of parallel programming is how to manage the sequence of multiple threads or processes, some of them, all, which one is before others.

- In other words, synchronization often means serialization, a thread waiting for another thread to finish or a thread waiting for a procedure.

Two types of synchronization:

- Synchronization among **work items** within a **work group**.
- Synchronization among **command queues** within a **context**.
Synchronization among work items (1)

- **Atomics**
  - If one thread reads or writes an atomic variable, an atomic operation keeps the current thread from interfering with another.
    ```
    // Thread A
    x--;   // Thread B may change the value of x before the next line is executed.
    if (x > 0) {
        // To do
    }
    ```
  - `atomic_work_item_fence()`
  - It has two flags `CLK_GLOBAL_MEM_FENCE` and `CLK_LOCAL_MEM_FENCE`
Synchronization among work items (2)

- **Barrier**
  - Barrier operation lets work item waits until all work items have reached the same barrier, and then all work items continue to run.
  - `work_group_barrier();` ➔ Work-items in a work-group must execute this before any thread can continue.
  - `barrier(CLK_LOCAL_MEM_FENCE);`
  - It has two flags
    - `CLK_GLOBAL_MEM_FENCE` (only for global memory)
    - `CLK_LOCAL_MEM_FENCE` (only for local memory)
The GPU’s main function, kernel, is invoked from the CPU host code.

CPU host code can process multiple kernels codes.

Each Kernel code has same or different functionality.
We need to release memory from the objects, kernels, and program once we finish the processing.

To release a memory object, call the function

- `clReleaseMemObject(cl_mem bufC)`

To release a kernel, call the function

- `clReleaseKernel(cl_kernel kernel_1)`

To release a program. Call the function

- `clReleaseProgram(cl_program program)`
To release the command queue, call the function

- `clReleaseCommandQueue(cl_command_queue command_queue)`

To release the context, call the function

- `clReleaseContext(cl_context context)`
THANK YOU
References