OpenCL execution model

Introduction to Vitis

Goal

- Understand
  - What OpenCL is and how it can be used in Vitis
  - The OpenCL execution model for Xilinx accelerators

What is OpenCL?

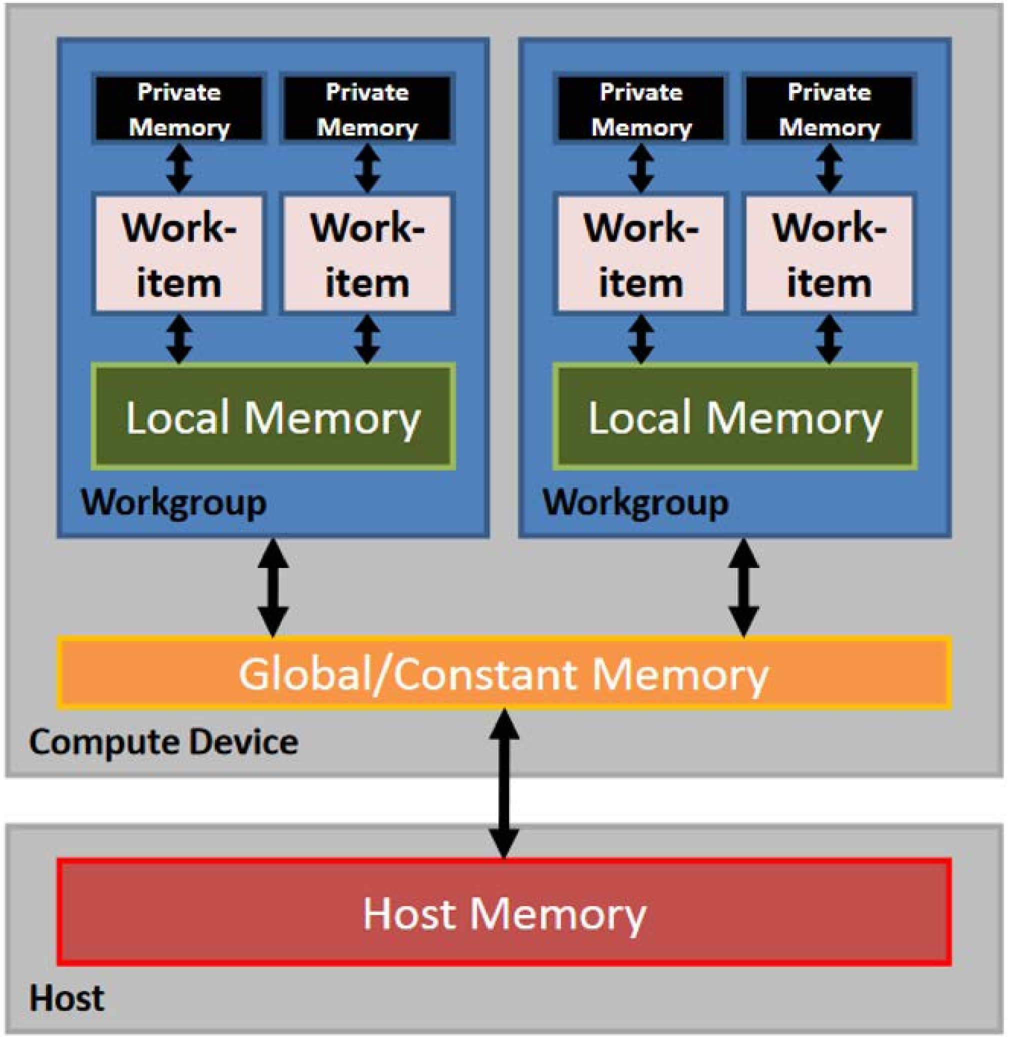
- OpenCL is open standard for cross-platform, parallel programming for diverse range of devices.
- Advantages
  - C/C++ based language for writing acceleration functions
  - Portable code across CPUs, GPUs and FPGAs
- Caveats
  - Performance is not portable
  - Separate optimization effort is required for each target platform

Basic definitions

- Host code:
  - Runs on host computer. Controls devices, kernels and data transfers with the kernel
- Kernel code:
  - The code that is executed on hardware
- Platform (OpenCL):
  - In OpenCL context – a data structure that identifies vendor's OpenCL implementation. Used to access a specific device.
- Platform (Xilinx):
  - Implementation of static part on Xilinx device. Contains all necessary IPs and logic for FPGA to enable communication between host and kernel(s).

OpenCL Memory Objects

- Host Memory
  - OpenCL only defines how host memory interacts with OpenCL objects
- Global Memory
  - Visible to host and device
  - Accessible to all work items in all workgroups
- Constant Memory
  - Read-only region of a global memory
- Local Memory
  - Local to a workgroup (shared by all work-items in a group)
- Private Memory
  - Accessible by a work-item



Five structures of OpenCL application

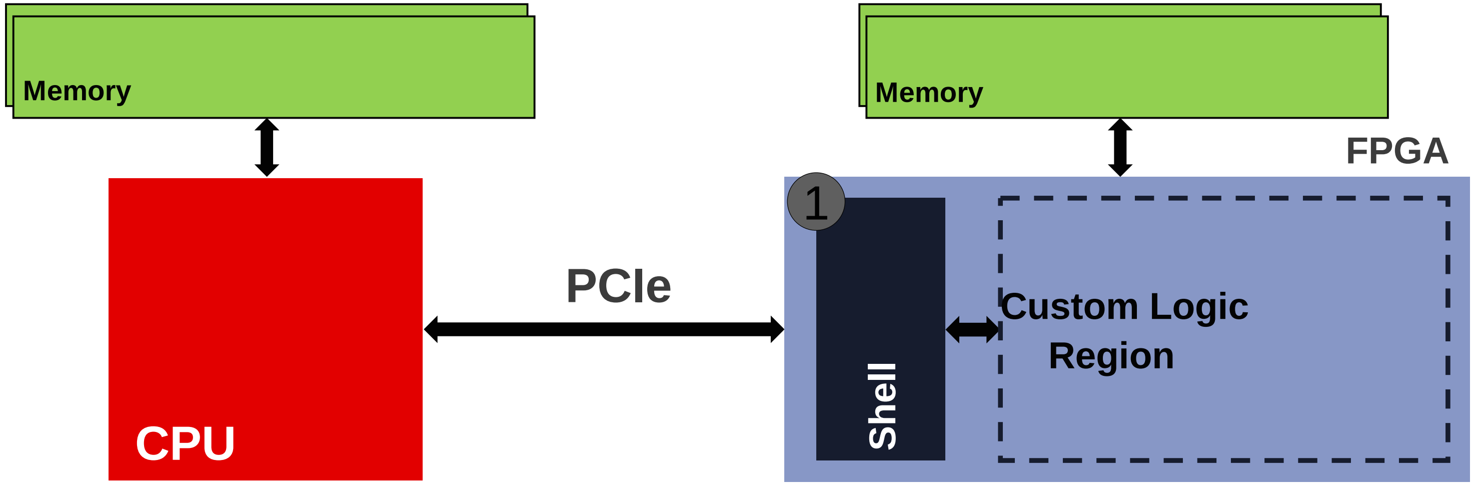
- First step of any OpenCL application – develop host code
  - Dispatches kernels to connected devices
- Host code needs to have following structures:
  - cl_device_id – represents the device
  - cl_kernel – represents specific kernel
  - cl_program – represents the source of where kernels come from
  - cl_command_queue – represents the queue which passes kernels to devices
  - cl_context – enables devices to receive kernels and transfer data

Host Code Execution on a Heterogeneous System

- Seven steps for application execution on a heterogeneous system
  - 1. Powering up
  - 2. Initializing runtime components
  - 3. Configuring the device
  - 4. Allocating buffers
  - 5. Writing the buffers to FPGA memory
  - 6. Running the accelerators
  - 7. Reading the buffers from FPGA memory

1. Powering Up

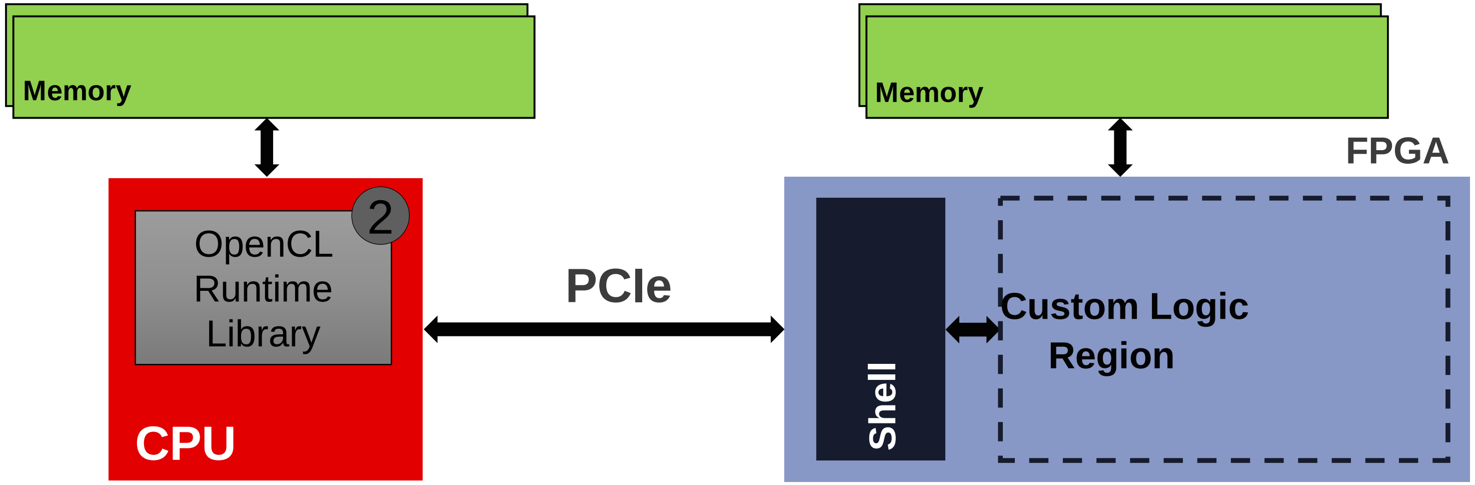
- On power up, only the *shell* is initialized in the FPGA
  - This enables the PCIe interface
  - The custom logic region is now available for programming by host CPU
- The shell will manage communication with the host



2. Initialize Runtime Components

- Creation of OpenCL context and device
  - Device → FPGA
  - Context → Platform
- Creation of OpenCL command queues used to send commands to the FPGA

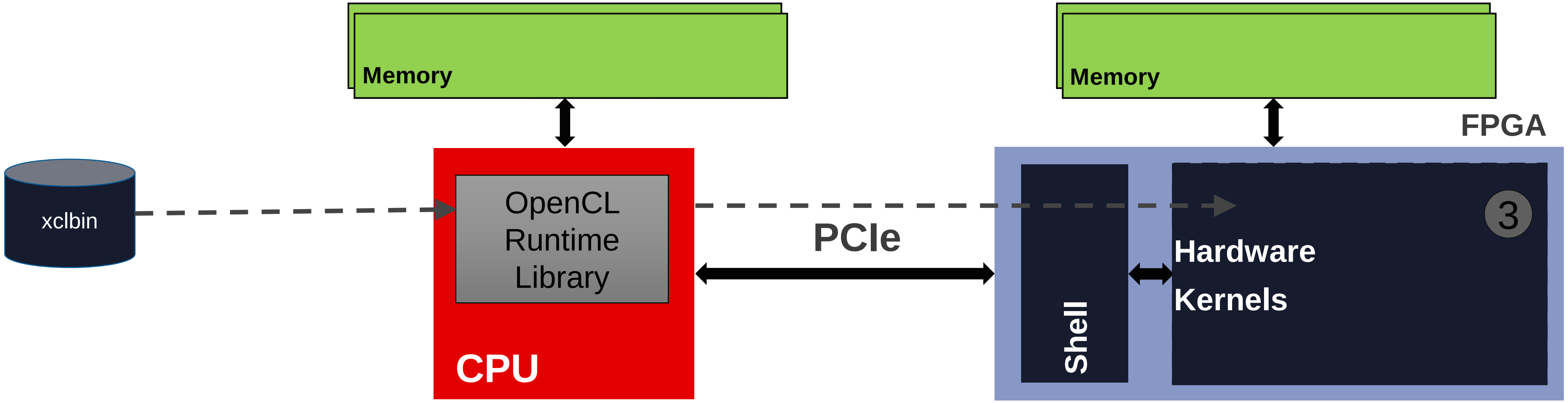
```
devices = get_devices("Xilinx");
cl::Device device = devices[0];
cl::Context context(…);

cl::CommandQueue q(context, device,…);
```



3. Configuring the Device

- Create cl::Program::Binaries program
  - Object to store the xclbin binary file
- Host programs the FPGA by calling cl::Program program
  - Loads the *.xclbin* file (FPGA binary)

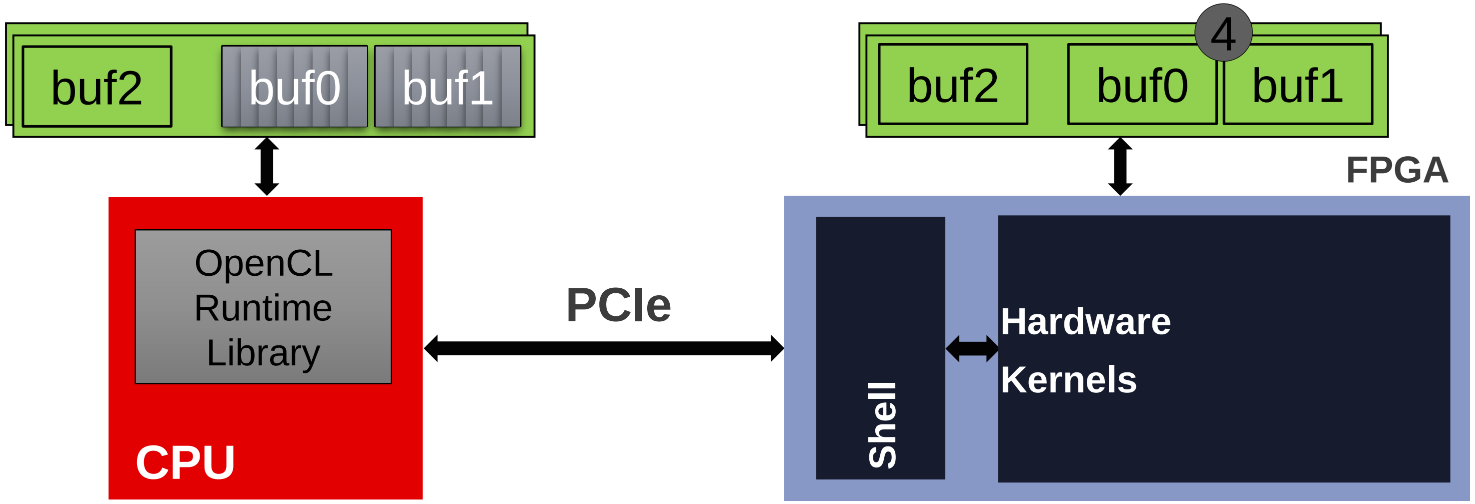
```
cl::Program::Binaries bins{{fileBuf,
        FileBufSize}};

cl::Program program(context, devices,
        bins,...);
```



4. Allocating Buffers

- Host allocates buffers in the device
- Buffers are used to transfer data from the CPU to the FPGA and back

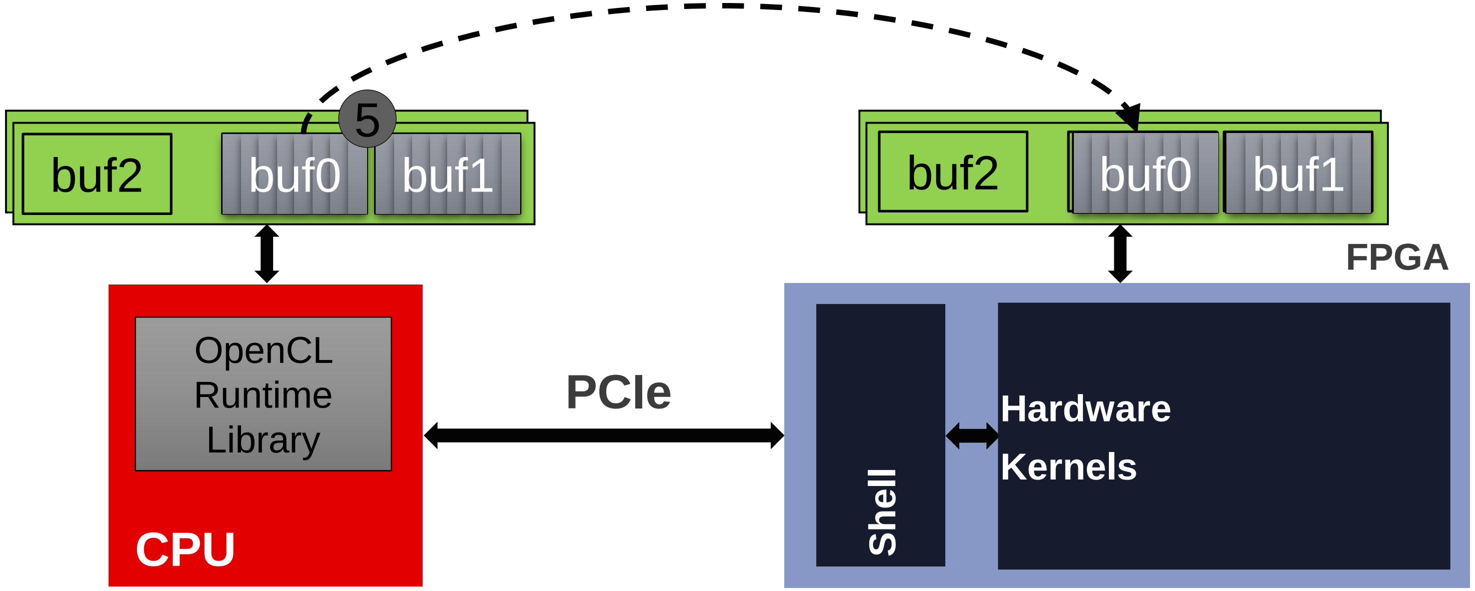
```
buf0 = cl::clCreateBuffer(context, CL_MEM_READ_ONLY,
         …);
buf1 = cl::clCreateBuffer(context, CL_MEM_READ_ONLY,
         …);
buf2 = cl::clCreateBuffer(context,
         CL_MEM_WRITE_ONLY, …);
```



5. Writing the Buffers to FPGA Memory

- Host copies data to be processed from host memory to the buffer in global memory

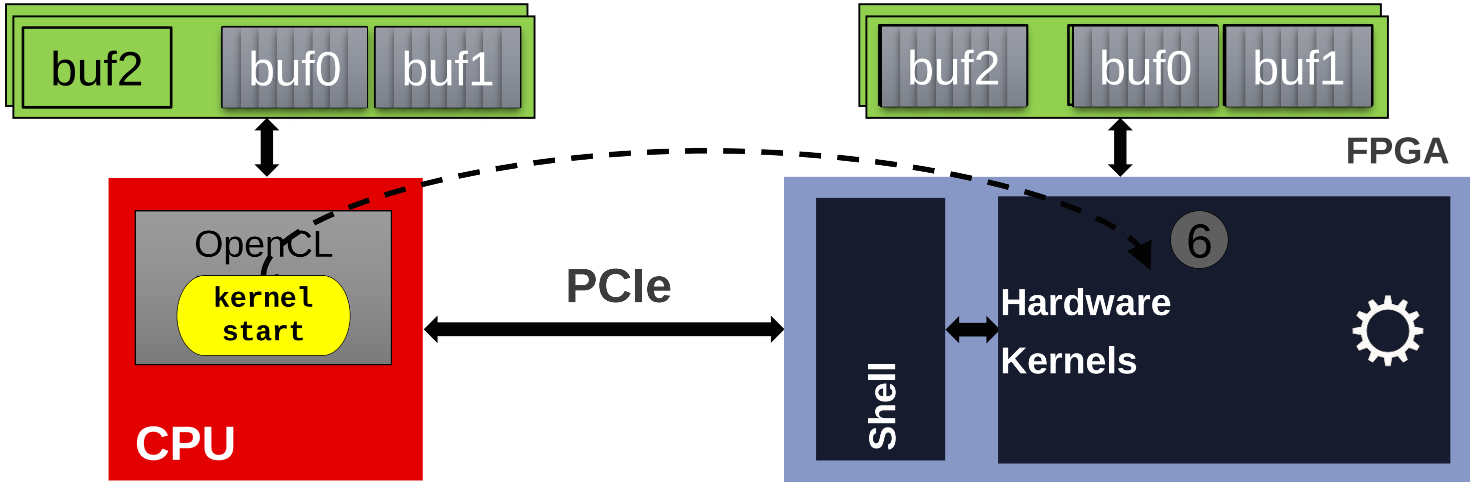
```
clEnqueueMigrateMemObjects(Command_Queue, 1,
        &GlobMem_BUF_DataIn_1, ..);
```



6. Running the Accelerators

- Host schedules execution of the desired kernel with EnqueueTask
- Runtime starts the kernel
- Kernel processes data previously copied to from host buffer to global memory

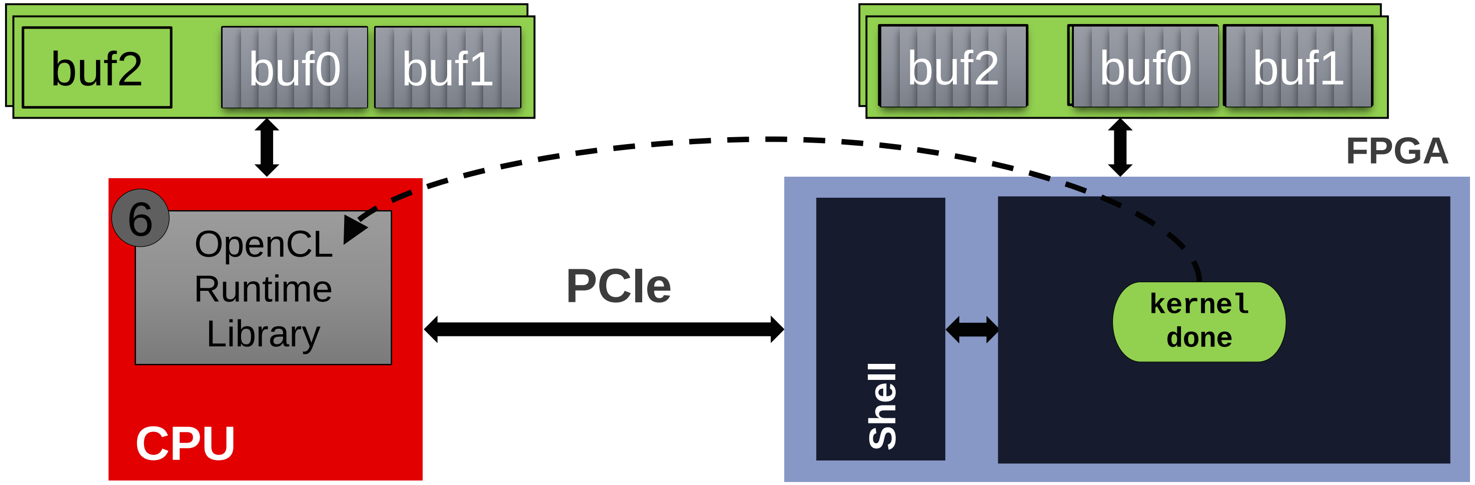
```
kernel = cl::clCreateKernel(program,
         “mykernel", …);


cl::clSetKernelArg(kernel, 0, sizeof(cl_mem), &buf0);
cl::clSetKernelArg(kernel, 1, sizeof(cl_mem), &buf1);
cl::clSetKernelArg(kernel, 2, sizeof(cl_mem), &buf2);

cl::clEnqueueTask(Command_Queue, Kernel, 0, NULL,
         NULL);
```



6. Running the Accelerators (Cont’d)

- After the kernel finishes processing the data, it notifies the host

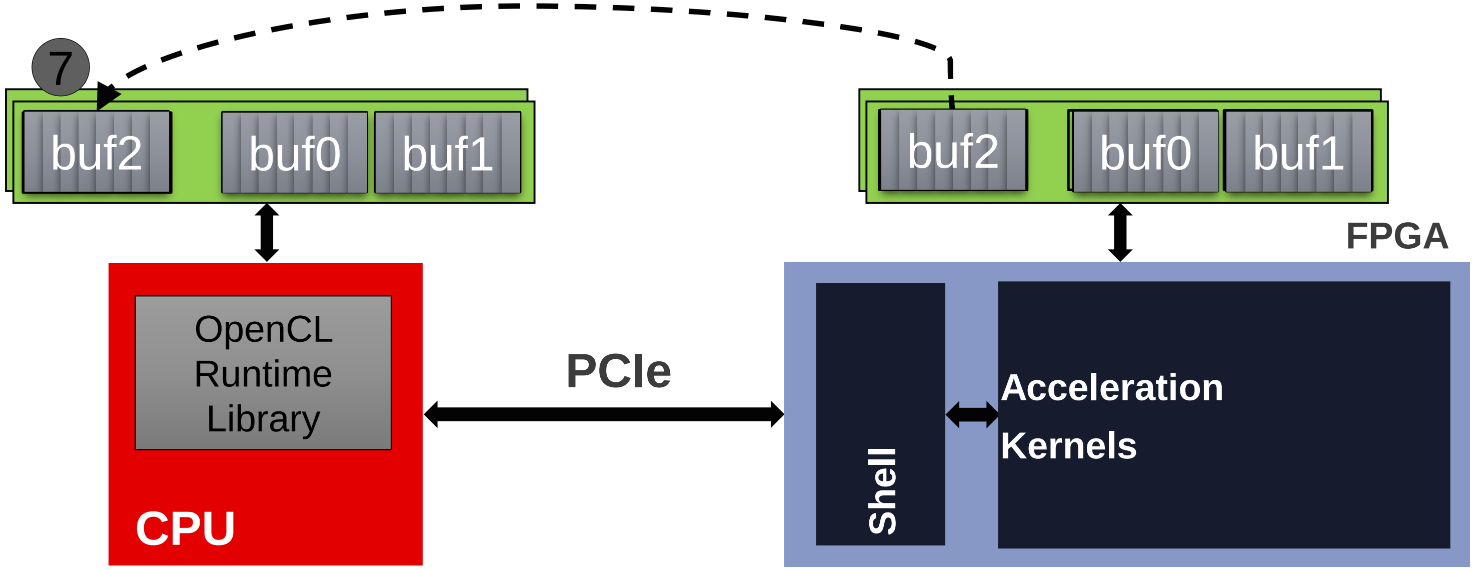
```
cl::clFinish(Command_Queue);
```



7. Reading the Buffers from FPGA Memory

- Host retrieves the results by scheduling a copy of global memory content back to host memory

```
cl::clEnqueueMigrateMemObjects(Command_Queue, 1, &GlobMem_BUF_RES, CL_MIGRATE_MEM_OBJECT_HOST, ..);
```



Thank You