COMP 8551 Advanced Games Programming Techniques

Borna Noureddin, Ph.D.

British Columbia Institute of Technology

OpenCL

OpenCL Overview

- **Platform model**: a high-level description of the heterogeneous system

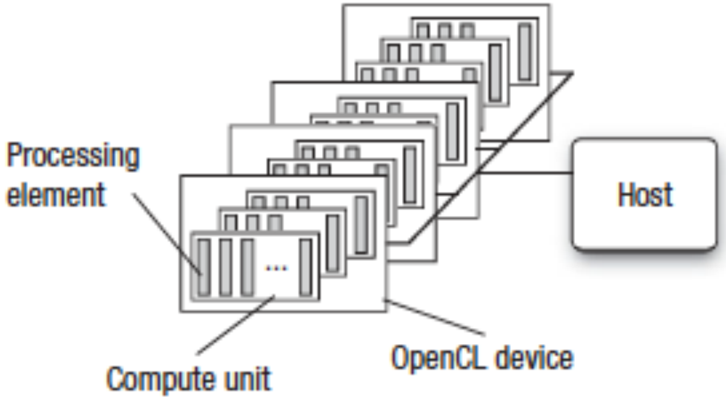
- **Execution model**: an abstract representation of how streams of instructions execute on the heterogeneous platform

- **Memory model**: the collection of memory regions within OpenCL and how they interact during an OpenCL computation

- **Programming models**: the high-level abstractions a programmer uses when designing algorithms to implement an application

Platform model

- Host interacts with environment external to OpenCL program (I/O, interaction user, etc)
- Host connected to 1+ OpenCL devices
- Device: where streams of instructions (or kernels) execute (aka "compute device")
    - can be CPU, GPU, DSP, or any other processor
    - further divided into compute units
    - compute units divided into one or more processing elements (PEs)
    - computations occur within PEs



Execution model

- OpenCL application consists of:
  - **host program**
  - collection of one or more **kernels**
- Host program runs on host
  - OpenCL does not define details of how host program works, only how it interacts with objects defined within OpenCL
- Kernels execute on OpenCL devices
  - Do real work of application
  - Typically simple functions that transform input memory objects into output memory objects
  - **OpenCL kernels**: functions written in OpenCL C
  - **Native kernels**: functions created outside OpenCL (function pointer) [OPTIONAL]

Execution model

- The OpenCL execution model defines how kernels execute

- How do individual kernels run on a device?

- How does the host define the context for kernel execution?

- How are the kernels enqueued for execution?

Kernel execution

- Host program issues **command** that submits kernel for execution on device
- Runtime system creates an integer index space
  - Instance of kernel executes for each point in this index space
- Each instance of an executing kernel: **work-item**
  - identified by coordinates in index space
  - coordinates are global ID for work-item
- Command creates collection of work-items, each of which uses same sequence of instructions defined by single kernel
- Sequence of instructions same, but behavior of each work-item can vary (branch statements or data selected through global ID)

Kernel execution

- Work-items organized into **work-groups**
- Provide coarse-grained decomposition of index space

- Exactly span global index space

- Work-groups same size in corresponding dimensions, and this size evenly divides global size in each dimension

- Work-groups assigned unique ID with same dimensionality as index space of work-items
- Work-items assigned unique local ID within work-group: can be uniquely identified by its global ID *or* by a combination of its local ID and work-group ID

Kernel execution

- Work-items in given work-group execute concurrently on PEs of single compute unit

- Implementation may serialize execution of kernels (may even serialize execution of workgroups in single kernel invocation)

- OpenCL only assures that workitems within a work-group execute concurrently

- You can never assume that work-groups or kernel invocations execute concurrently

Kernel execution

- Index space spans an N -dimensioned range of values (**NDRange**)

- N can be 1, 2, or 3

- Integer array of length N specifying size of index space in each dimension

- Work-item's global and local ID is an N–dimensional tuple

- Work-groups assigned IDs using a similar approach to that used for work-items

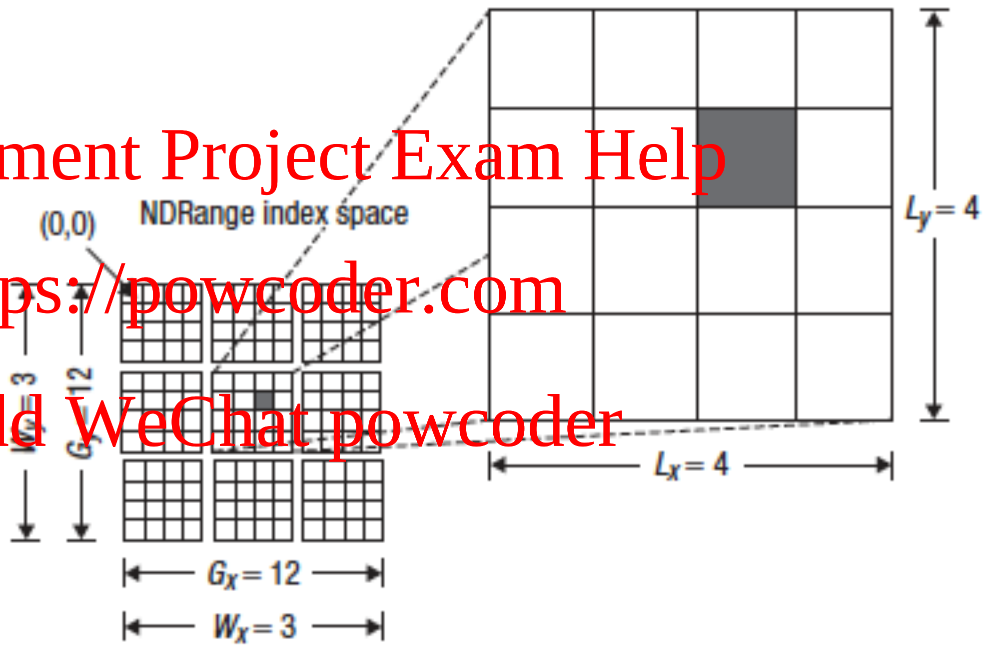
Kernel execution

$$L_x = G_x \,/\, W_x$$
$$L_y = G_y \,/\, W_y$$

$$g_x = w_x * L_x + l_x$$
$$g_y = w_y * L_y + l_y$$

$$w_x = g_x \,/\, L_x$$
$$w_y = g_y / L_y$$
$$l_x = g_x \,\%\, L_x$$
$$l_y = g_y \,\%\, L_y$$

$$g_x = w_x * L_x + l_x + o_x$$
$$g_y = w_y * L_y + l_y + o_y$$

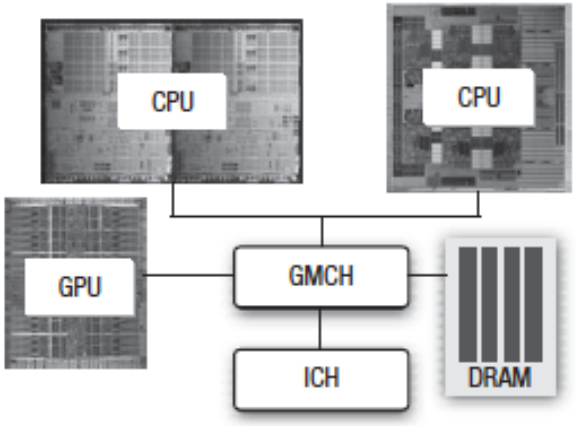


Context

- In OpenCL, computation takes place on device
- But host:
  - Defines and establishes context kernels
  - Defines NDRanges
  - Defines queues that control details of how/when kernels execute
- Context defines environment within which kernels are defined and execute:
  - **Devices**: collection of OpenCL devices to be used by host
  - **Kernels**: OpenCL functions that run on devices
  - **Program objects**: program source code and executables that implement kernels
  - **Memory objects**: set of objects in memory that are visible to OpenCL devices and contain values that can be operated on by instances of a kernel

Context

- Created and manipulated by host using the OpenCL API
- Context also contains one or more "program objects"
  - think of these as a dynamic library from which the functions used by the kernels are pulled
- Host program defines devices within context: only at that point is it possible to know how to compile the program source code to create the code for the kernels



Context

- Program object built at runtime within host program (like shader program)
- Context also defines how kernels interact with memory
- On heterogeneous platform, often multiple address spaces to manage
- Devices may have range of different memory architectures
- OpenCL introduces idea of "memory objects"
  - explicitly defined on host
  - explicitly moved between host and OpenCL devices
  - extra burden on programmer, but allows support for much wider range of platforms

Command-Queues

- Interaction between host and devices occurs through commands posted by host to **command-queue**
- Created by host and attached to single device after context has been defined
- Host places commands into command-queue, and commands are then scheduled for execution on the associated device
- OpenCL supports three types of commands:
  - **Kernel execution commands** execute kernel on PEs of device
  - **Memory commands** transfer data between host and different memory objects, move data between memory objects, or map and unmap memory objects from host address space
  - **Synchronization commands** put constraints on order of execution

Command-Queues

Typical host program

- Define context and command-queues
- Define memory and program objects
- Builds data structures needed on host
- Use command-queue to move memory objects from the host to devices
- Attach kernel arguments to memory objects
- Submit kernels to command-queue for execution
- When kernel completed, memory objects copied back to host

Command-Queues

- When multiple kernels are submitted, may need to interact
  - E.g., one set of kernels may generate memory objects that a following set of kernels needs to manipulate
  - Synchronization commands can be used to force first set to complete before following set begins
- Many additional subtleties associated with how the commands work in OpenCL
- Commands always execute asynchronously to host
- Host submits commands to command-queue and continues without waiting for them to finish
- If necessary for host to wait on command, can be explicitly established with synchronization command

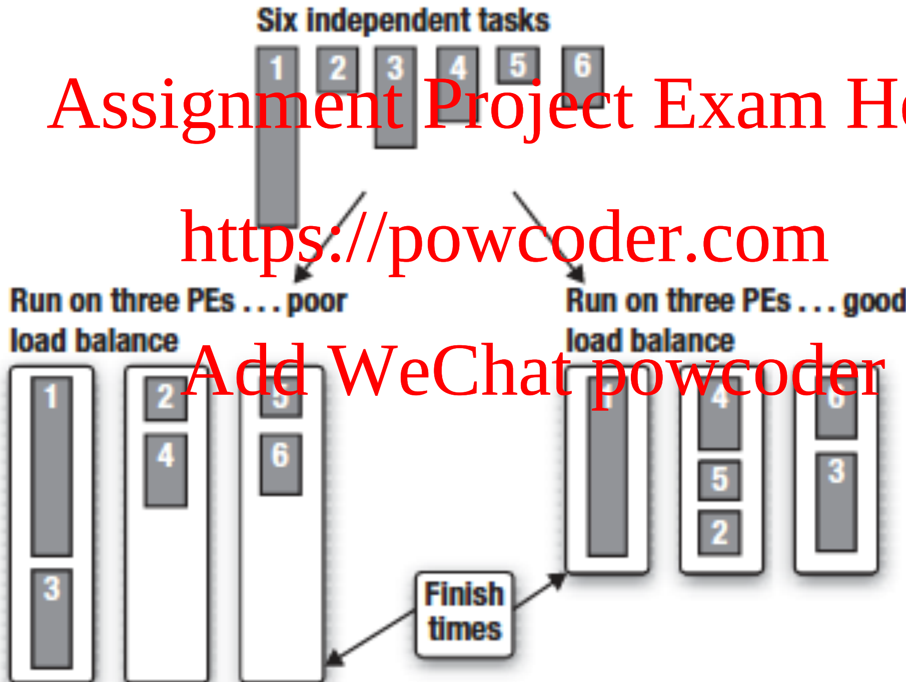
Command-Queues

Commands within single queue execute relative to each other in one of 2 modes:

- **In-order execution**: Commands launched in order in which they appear in command-queue and complete in order (serializes execution order of commands)

- **Out-of-order execution**: Commands issued in order but do not wait to complete before the following commands execute (order constraints enforced by programmer through explicit synchronization mechanisms) [OPTIONAL]

Command-Queues

- Why out-of-order? Remember load balancing?
- Applicati


te
- Efficient compute units to b ately same amount
- You can ut order in which yo at the in-order exe l
- But what lifferent amounts an be very hard! Out-of-order queue can take care of this for you

Command-Queues

- **Automatic load balancing**: Commands can execute in any order, so if compute unit finishes its work early, it can immediately fetch a new command and start executing new kernel

- Commands generate event objects

  - Command can be told to wait until certain conditions on event objects exist

  - Events can also be used to coordinate execution between host and devices
- Also possible to associate multiple queues with single context for any devices within that context
  - Run concurrently and independently with no explicit mechanisms within OpenCL to synchronize between them

Memory Model

Two types of memory objects

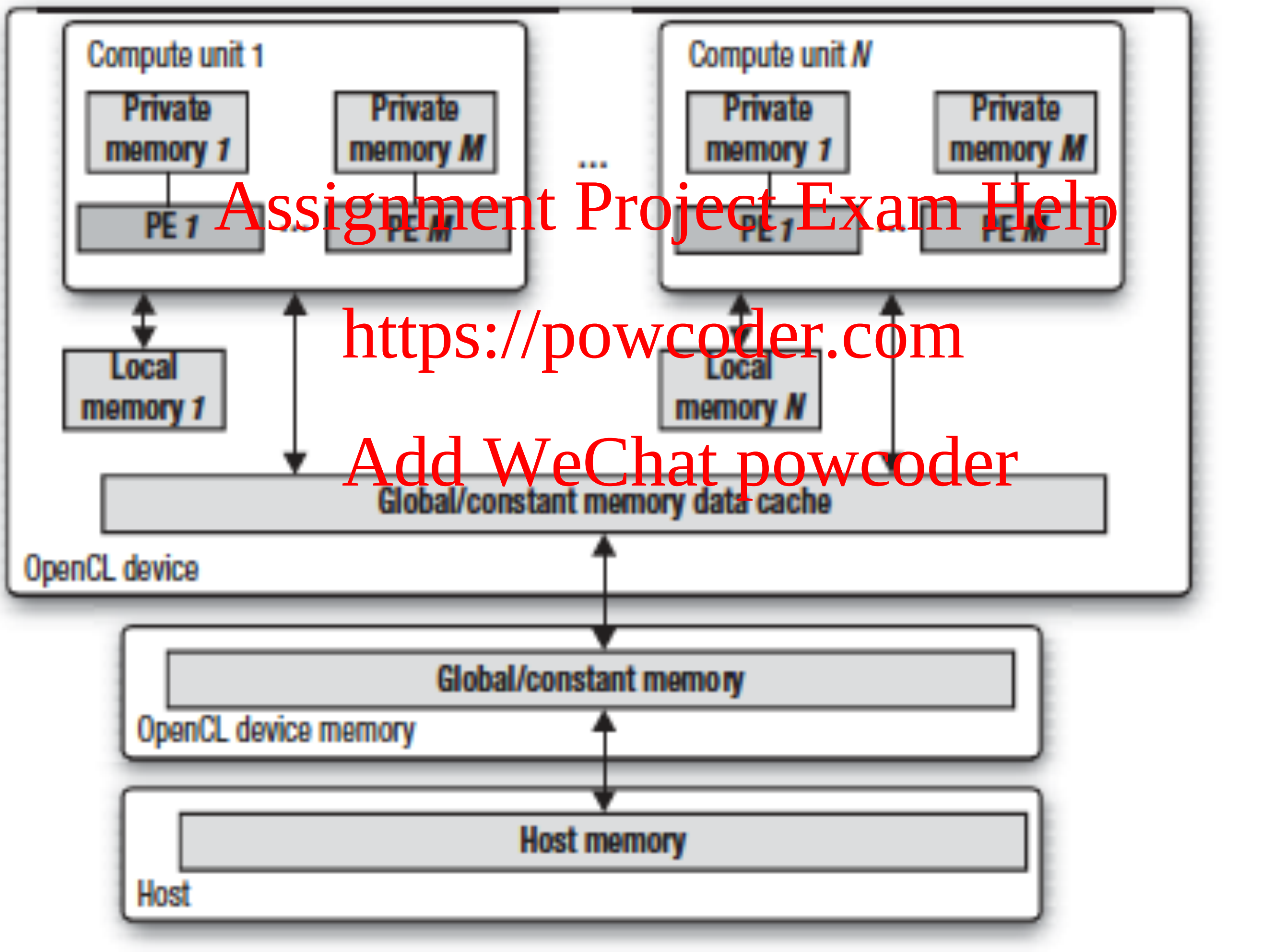
- **Buffer object**:
  - contiguous block of memory made available to kernels
  - programmer can map data structures onto this buffer and access buffer through pointers
  - flexibility to define just about any data structure
- **Image object**:
  - restricted to holding images
  - storage format may be optimized to needs of specific device
  - important to give an implementation freedom to customize image format
  - opaque object
  - OpenCL provides functions to manipulate images, but other than these specific functions, the contents of image object are hidden from kernel program

Memory Model

OpenCL memory model defines five distinct memory regions:

- **Host memory**: visible only to host
- **Global memory**: permits read/write access to all work-items in all work-groups
- **Constant memory**: region of global memory that remains constant during kernel execution
  - host allocates and initializes
  - work-items have read-only access
- **Local memory**: local to work-group
  - can be used to allocate variables shared by all work-items
  - may be implemented as dedicated regions of memory on device or mapped onto sections of global memory
- **Private memory**: private to work-item
  - not visible to other work-items

Memory Model



Programming Model

- How to map parallel algorithms onto OpenCL
- Programming models intimately connected to how programmers reason about their algorithms
- OpenCL defined with two different programming models in mind: task parallelism and data parallelism
- Also possible to think in terms of a hybrid model: tasks that contain data parallelism

Programming Model

Data-Parallel Programming Model

- Problems well suited are organized around data structures, the elements of which can be updated concurrently
- Single logical sequence of instructions applied concurrently to elements of data structure
- Structure of algorithm is designed as sequence of concurrent updates to data structures within problem
- Natural fit with OpenCL's execution model
- Key is the NDRange defined when kernel is launched
- Algorithm designer aligns data structures with NDRange index space and maps them onto OpenCL memory objects
- Kernel defines sequence of instructions to be applied concurrently as work-items

Programming Model

Data-Parallel Programming Model

- Work-items in single work-group may need to share data (local memory region)

- Regardless of order in which work-items complete, same results should be produced

- Work-items in same work-group can participate in a **work-group barrier** (all must execute before any continuing)
- NB: no mechanism for synchronization between work-items from different work-groups

Programming Model

Data-Parallel Programming Model

- **Single Instruction Multiple Data** or SIMD: no branch statements in kernel, each work-item will execute identical operations but on subset of data items selected by its global ID
- **Single Program Multiple Data** or SPMD: branch statements within a kernel leading each work-item to possibly execute very different operations
- On platforms with restricted bandwidth to instruction memory or if PEs map onto vector unit, SIMD model can be dramatically more efficient

Programming Model

Data-Parallel Programming Model

- Vector instructions strictly SIMD
- E.g., numerical integration program (4.0/(1 + x2))


```
float8 x, psum_vec;
float8 ramp= (float8)(0.5, 1.5, 2.5, 3.5,
4.5, 5.5, 6.5, 7.5};
float8 four= (float8)(4.0); // fill with 8 4's
float8 one = (float8)(1.0); // fill with 8 1's
float step_number; // step number from loop index
float step_size; // Input integration step size
. . . and later inside a loop body . . .
x = ((float8)step_number +ramp)*step_size;
psum_vec+=four/(one + x*x);
```

Programming Model

Task-Parallel Programming Model

- Task = kernel that executes as a single work-item regardless of NDRange used by other kernels in application
- Concurrency is internal to the task (eg, vector operations on vector types)
- Kernels submitted as tasks that execute at the same time with an out-of-order queue
- Tasks connected into task graph using OpenCL's event model
  - Commands submitted to event queue may optionally generate events
  - Subsequent commands can wait for these events before executing