Programming Multiple Devices

A Collaboration Between
David Kaeli, Northeastern University
Benedict R. Gaster, AMD

Instructor Notes

- This lecture describes the different ways to work with multiple devices in OpenCL (i.e., within a single context and using multiple contexts), and the tradeoffs associated with each approach
- The lecture concludes with a quick discussion of heterogeneous load-balancing issues when working with multiple devices

Approaches to Multiple Devices

- Single context, multiple devices
  - Standard way to work with multiple devices in OpenCL
- Multiple contexts, multiple devices
  - Computing on a cluster, multiple systems, etc.
- Considerations for CPU-GPU heterogeneous computing

Single Context, Multiple Devices

- Nomenclature:
  - “clEnqueue*” is used to describe any of the clEnqueue commands (i.e., those that interact with a device)
    - E.g. clEnqueueNDRangeKernel(), clEnqueueReadImage()
  - “clEnqueueRead*” and “clEnqueueWrite*” are used to describe reading/writing to either buffers or images
    - E.g. clEnqueueReadBuffer(), clEnqueueWriteImage()

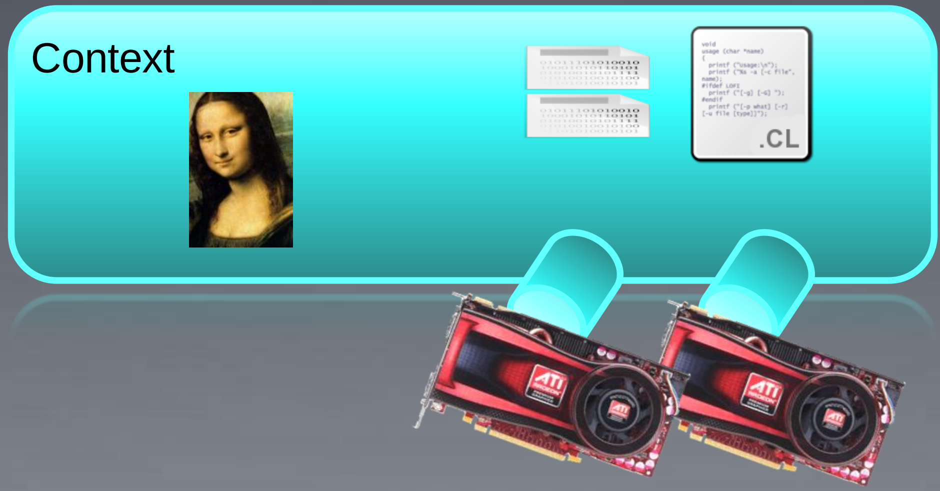
Single Context, Multiple Devices

- Associating specific devices with a context is done by passing a list of the desired devices to `clCreateContext()`
- The call `clCreateContextFromType()` takes a device type (or combination of types) as a parameter and creates a context with all devices of that type:

cl_device_type	Description
CL_DEVICE_TYPE_CPU	An OpenCL device that is the host processor. The host processor runs the OpenCL implementations and is a single or multi-core CPU.
CL_DEVICE_TYPE_GPU	An OpenCL device that is a GPU. By this we mean that the device can also be used to accelerate a 3D API such as OpenGL or DirectX.
CL_DEVICE_TYPE_ACCELERATOR	Dedicated OpenCL accelerators (for example the IBM CELL Blade). These devices communicate with the host processor using a peripheral interconnect such as PCIe.
CL_DEVICE_TYPE_DEFAULT	The default OpenCL device in the system.
CL_DEVICE_TYPE_ALL	All OpenCL devices available in the system.

Single Context, Multiple Devices

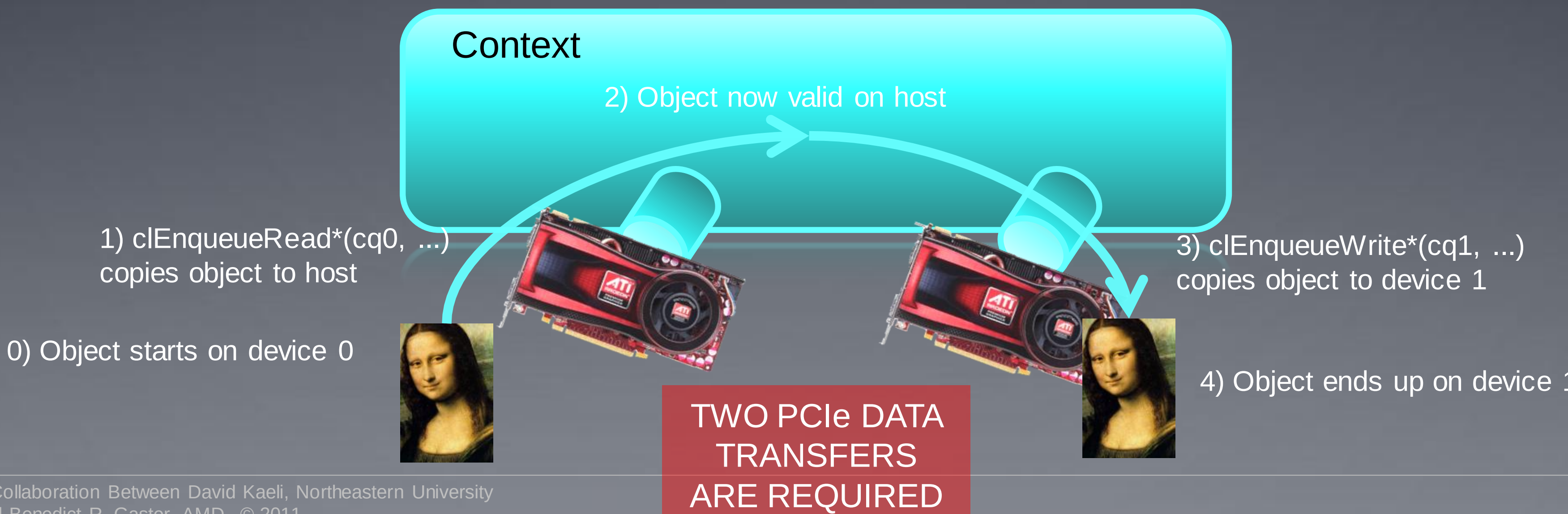
- When multiple devices are part of the same context, most OpenCL objects are shared
  - Memory objects, programs, kernels, etc.
- One command queue must exist per device and is supplied in OpenCL when the target GPU needs to be specified
  - Any clEnqueue* function takes a command queue as an argument



Single Context, Multiple Devices

- While memory objects are common to a context, they must be explicitly written to a device before being used
  - Whether or not the same object can be valid on multiple devices is vendor specific

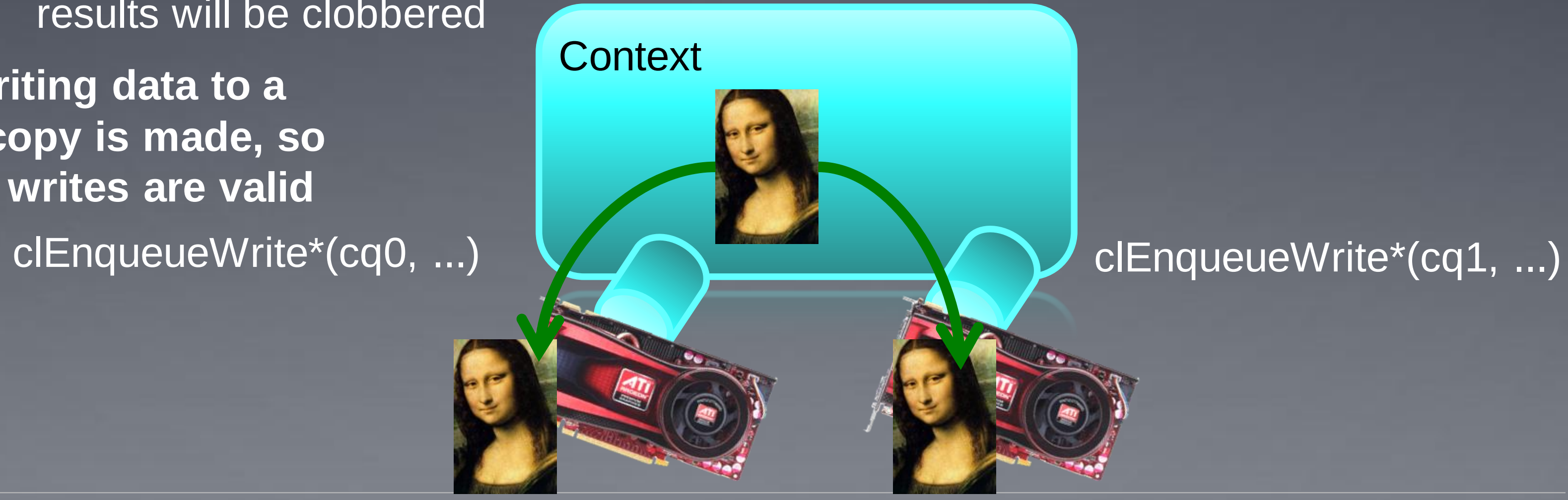
- OpenCL does not assume that data can be transferred directly between devices, so commands only exists to move from a host to device, or device to host
  - Copying from one device to another requires an intermediate transfer to the host



Single Context, Multiple Devices

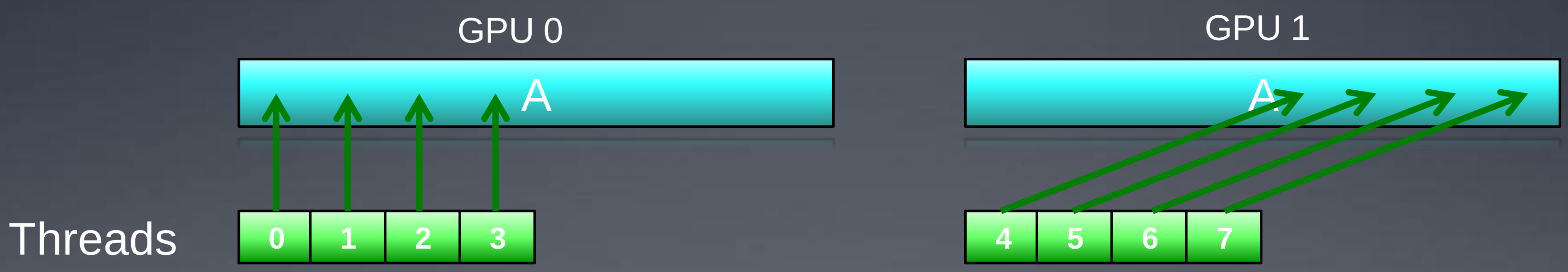
- The behavior of a memory object written to multiple devices is vendor-specific
  - OpenCL does not define if a copy of the object is made or whether the object remains valid once written to a device
  - We can imagine that a CPU would operate on a memory object in-place, while a GPU would make a copy (so the original would still be valid until it is explicitly written over)
    - Fusion GPUs from AMD could potentially operate on data in-place as well
  - Currently AMD/NVIDIA implementations allow an object to be copied to multiple devices (even if the object will be written to)
    - When data is read back, separate host pointers must be supplied or one set of results will be clobbered

When writing data to a GPU, a copy is made, so multiple writes are valid

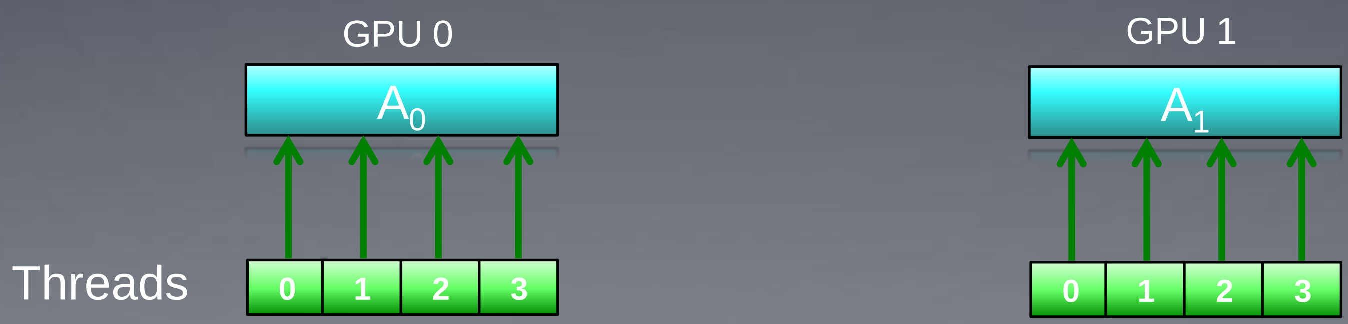


Single Context, Multiple Devices

- Just like writing a multi-threaded CPU program, we have two choices for designing multi-GPU programs
  1. Redundantly copy all data and index using global offsets



  2. Split the data into subsets and index into the subset



Single Context, Multiple Devices

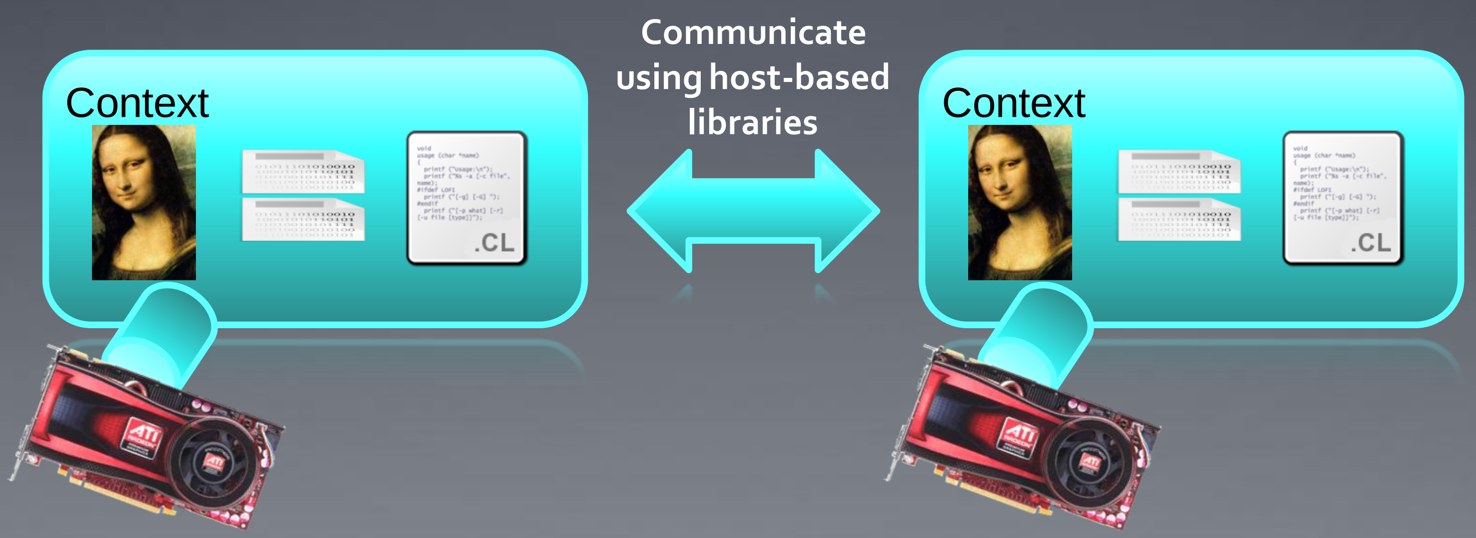
- OpenCL provides mechanisms to help with both multi-device techniques
  - `clEnqueueNDRangeKernel()` optionally takes offsets that are used when computing the global ID of a thread
    - Note that for this technique to work, any objects that are written to will have to be synchronized manually
  - *SubBuffers* were introduced in OpenCL 1.1 to allow a buffer to be split into multiple objects
    - This allows reading/writing to offsets within a buffer to avoid manually splitting and recombining data

Single Context, Multiple Devices

- OpenCL *events* are used to synchronize execution on different devices within a context
- Each `clEnqueue*` function generates an event that identifies the operation
- Each `clEnqueue*` function also takes an optional list of events that must complete before that operation should occur
- `clEnqueueWaitForEvents()` is the specific call to wait for a list of events to complete
- Events are also used for profiling and were covered in more detail in Lecture 11

Multiple Contexts, Multiple Devices

- An alternative approach is to create a redundant OpenCL context (with associated objects) per device
- Perhaps is an easier way to split data (based on the algorithm)
  - Would not have to worry about coding for a variable number of devices
  - Could use CPU-based synchronization primitives (such as locks, barriers, etc.)



Multiple Contexts, Multiple Devices

- Follows SPMD model more closely
  - CUDA/C's runtime-API approach to multi-device code
- No code required to consider explicitly moving data between a variable number of devices
  - Using functions such as scatter/gather, broadcast, etc. may be easier than creating subbuffers, etc. for a variable number of devices
- Supports distributed programming
  - If a distributed framework such as MPI is used for communication, programs can be ran on multi-device machines or in distributed environments

Multiple Contexts, Multiple Devices

- In addition to PCI-Express transfers required to move data between host and device, extra memory and network communication may be required
- Host libraries (e.g., pthreads, MPI) must be used for synchronization and communication

Heterogeneous Computing

- Targeting heterogeneous devices (e.g., CPUs and GPUs at the same time) requires awareness of their different performance characteristics for an application

- To generalize:

	CPUs	GPUs
Overhead	Low	High (depending on data)
Performance	Variable	High*

*otherwise application wouldn't use OpenCL

Heterogeneous Computing

- Factors to consider
  - Scheduling overhead
    - What is the startup time of each device?
  - Location of data
    - Which device is the data currently resident on?
      - Data must be transferred across the PCI-Express bus
  - Granularity of workloads
    - How should the problem be divided?
    - What is the ratio of startup time to actual work
  - Execution performance relative to other devices
    - How should the work be distributed?

Heterogeneous Computing

- Granularity of scheduling units must be weighed
  - Workload sizes that are too large may execute slowly on a device, stalling overall completion
  - Workload sizes that are too small may be dominated by startup overhead
- Approach to load-balancing #1:
  - Begin scheduling small workload sizes
  - Profile execution times on each device
  - Extrapolate execution profiles for larger workload sizes
  - Schedule with larger workload sizes to avoid unnecessary overhead
- Approach to load-balancing #2:
  - If one device is much faster than anything else in the system, just run on that device

Summary

- There are different approaches to multi-device programming
  - Single context, multiple devices
    - Can only communicate with devices recognized by one vendor
    - Code must be written for a general number of devices
  - Multiple contexts, multiple devices
    - More like distributed programming
    - Code can be written for a single device (or multiple devices), with explicit movement of data between contexts