OPENCL INTRODUCTION

OPENCL 2.0 UNIVERSITY TOOLKIT

Zhongliang Chen and Yash Ukidave,
Northeastern University Computer Architecture Research Lab
with
Perhaad Mistry and Dana Schaa, AMD

Instructor Notes

AMD

- These slides can serve as an introduction to OpenCL
- These slides cover the OpenCL models, the API and can serve as a reference for the API
- Some of the OpenCL examples can be shown in an IDE along with the slides

OpenCL Architecture

AMD

- OpenCL allows parallel computing on heterogeneous devices
  - CPUs, GPUs, FPGAs, etc.
  - Provides portable accelerated code
- Defined in four models
  - Platform model
  - Execution model
  - Memory model
  - Programming model

Platform Model

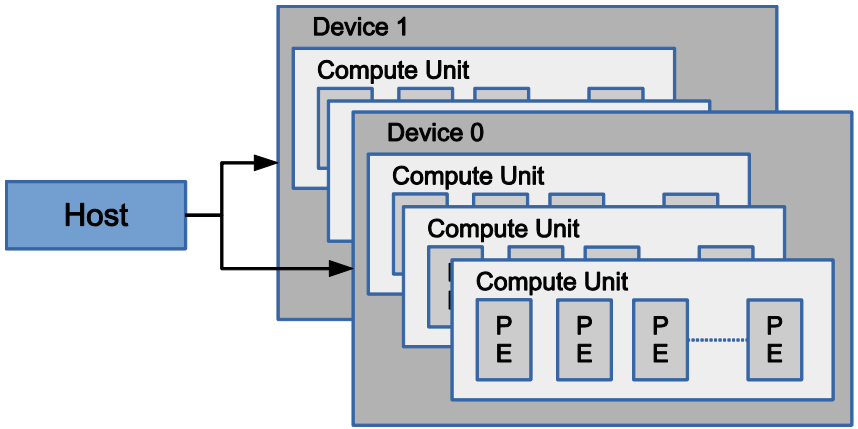
AMD

- The platform model describes the computational resources utilized by OpenCL and their relationship with one another
- Each OpenCL implementation (i.e. an OpenCL library) can create platforms consisting of resources in the system with which it is capable of interacting
  - For example, and AMD platform can consist of X86 CPUs and Radeon GPUs
- OpenCL uses an “Installable Client Driver” model
  - The goal is to allow platforms from different vendors to co-exist
  - Applications can choose a platform at runtime

Platform Model

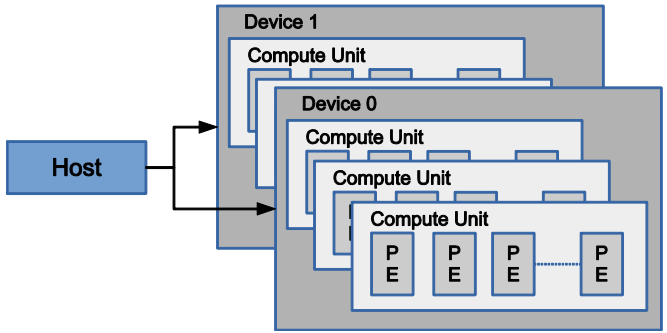
AMD

- The platform model defines a host connected to one or more compute devices
- A device is divided into one or more compute units
- Compute units are divided into one or more processing elements
  - Each processing element maintains its own program counter



Host/Devices

- The host is whatever processor the OpenCL library runs on
  - x86 CPUs generally
- Devices are processors that the library can talk to
  - CPUs, GPUs, FPGAs, and other accelerators
- For AMD
  - All CPUs are combined into a single device (each core is a compute unit and processing element)
  - Each GPU is a separate device



Selecting a Platform

AMD

```
cl_int  clGetPlatformIDs (cl_uint num_entries,
                          cl_platform_id *platforms,
                          cl_uint *num_platforms)
```

- The platform is selected using an API call
  - There is a pattern to many OpenCL calls which allows the API to be vendor neutral
    - A first call can be made to return storage requirements for data
    - A second call can be made to return the data
- Robust applications will therefore usually make this call twice
  - The first call is used to get the number of platforms available to the implementation
  - Memory space is then allocated for the platform objects
  - The second call is used to retrieve the platform objects

Selecting Devices

AMD

- Once a platform is selected, we can then query for the devices that it knows how to interact with

```
cl_int      clGetDeviceIDs[4] (cl_platform_id platform,
                               cl_device_type device_type,
                               cl_uint num_entries,
                               cl_device_id *devices,
                               cl_uint *num_devices)
```

- We can specify which types of devices we are interested in (e.g. all devices, CPUs only, GPUs only)
- This call is performed twice as with **clGetPlatformIDs**
  - The first call is to determine the number of devices, the second retrieves the device objects

Contexts

AMD

- A context is the environment for managing OpenCL objects and resources
- To manage OpenCL programs, the following are associated with a context
  - Devices: the things doing the execution
  - Program objects: the program source that implements the kernels
  - Kernels: functions that run on OpenCL devices
  - Memory objects: data that are operated on by the device
  - Command queues: mechanisms for interaction with the devices
    - Commands include: data transfers, kernel execution, and synchronization
- When you create a context, you will provide a list of devices to associate with it
  - For the rest of the OpenCL resources, you will associate them with the context as they are created

Contexts

AMD

```
cl_context  clCreateContext (const cl_context_properties *properties,
                             cl_uint num_devices,
                             const cl_device_id *devices,
                             void (CL_CALLBACK *pfn_notify)(const char *errinfo,
                                                            const void *private_info, size_t cb,
                                                            void *user_data),
                             void *user_data,
                             cl_int *errcode_ret)
```

- This function creates a context given a list of devices
- The *properties* argument specifies which platform to use (if NULL, the default chosen by the vendor will be used)
- The function also provides a callback mechanism for reporting errors to the user

Command Queues

AMD

```
cl_command_queue  clCreateCommandQueueWithProperties (cl_context context,
                                                      cl_device_id device,
                                                      const cl_queue_properties *properties,
                                                      cl_int *errcode_ret)
```

- A command queue is the mechanism for the host to request that an action be performed by the device (i.e., the host sends *commands* to a device)
  - Commands include initiating a memory transfer, begin executing a kernel, etc.
- The command queue properties specify:
  - Whether out-of-order execution of commands is allowed
  - Whether profiling is enabled
  - Whether this queue should reside on a device

Command Queues

AMD

- Since a command queue targets a single device, a separate command queue is required for each device
- Some commands within the queue can be specified as synchronous or asynchronous
- Commands can execute in-order or out-of-order
- Command queues associate a context with a device

Events

AMD

- Events are OpenCL's mechanism of specifying dependencies between commands
- All OpenCL API calls to enqueue a command onto a command queue have the option of generating an event, and taking a list of events that must complete before this command can execute
  - The list of events that specify dependencies are referred to as a *wait list*
- Events are also used for profiling (discussed in a later series)
- With an in-order command queue (default), each command will complete before the next command begins, so manually specifying dependencies is not required

Queue Synchronization

AMD

cl_int **clFinish** (cl_command_queue *command_queue*)

- A call to **clFinish** blocks a host program until all enqueued commands have completed
  - In practice, this call has higher overhead than specifying dependencies using events, and should be used sparingly when high performance is required

Device-Side Enqueuing

AMD

- OpenCL 2.0 introduced device-side command queues
  - Allows a device to enqueue commands to itself
    - e.g. a kernel can enqueue another kernel execution onto the same device
  - Parent and child kernels execute asynchronously
  - A parent kernel is not registered as complete until all its child kernels have completed
- Device-side command queues are out-of-order
  - Events can be used to enforce dependencies

Memory Objects

AMD

- Memory objects are handles to data that can be accessed by a kernel
  - OpenCL memory object types are buffers, images, and pipes
- Buffers
  - Contiguous chunks of memory stored sequentially and can be accessed directly (arrays, pointers, structs)
  - Read/write capable
- Images
  - Opaque objects (2D or 3D)
  - Can only be accessed via intrinsic functions **read_image()** and **write_image()**
  - Can be read, written, or both in a kernel (new in OpenCL 2.0)
- Pipes (New in OpenCL 2.0)
  - Ordered sequence of data items called packets
  - Stored on the basis of a first in, first out method
  - Can only be accessed via intrinsics **read_pipe()** and **write_pipe()**

Creating Buffers

AMD

```
cl_mem    clCreateBuffer (cl_context context,
                          cl_mem_flags flags,
                          size_t size,
                          void *host_ptr,
                          cl_int *errcode_ret)
```

- This function creates a buffer (cl_mem object) for the given context
- The flags specify:
  - Reading and writing permissions on the data
  - If the data should be copied from the host pointer
  - If the data should be accessed directly from the host pointer itself

Transferring Data

AMD

- While the OpenCL runtime is responsible for ensuring data is accessible by a kernel, explicit memory transfer commands can be used for improved performance
- OpenCL provides commands to transfer data to and from devices
  - **clEnqueue{Write|Read}{Buffer|Image}**
  - Writing is copying from the host to a device
  - Reading is copying from a device to the host
- OpenCL API calls also exist to directly map all or part of a memory object to a host pointer

Transferring Data

AMD

```
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)
```

- This command initializes the OpenCL memory object
  - The command will write data from a host pointer (*ptr*) to the memory object
  - Often the data will be resident on the device, but this is not explicitly required by the OpenCL specification
- The *blocking_write* parameter specifies that *ptr* can be reused after the command completes

Programs

AMD

- A program object is a collection of OpenCL kernels, and functions and data used by kernels
  - Can be source code (text) or precompiled binary
- Creating a program object requires either reading in a string (source code) or a precompiled binary
- To compile the program
  - Specify which devices are targeted
    - Program is compiled for each device
  - Pass in compiler flags (optional)
  - Check for compilation errors (optional, output to screen)

Creating Programs

AMD

```
cl_program  clCreateProgramWithSource (cl_context context,
                                       cl_uint count,
                                       const char **strings,
                                       const size_t *lengths,
                                       cl_int *errcode_ret)
```

- This function creates a program object from strings of source code
  - *count* specifies the number of strings
  - The user must create a function to read in the source code to a string
- If *strings* are not NULL-terminated, *lengths* is used to specify the strings' lengths

Compiling Programs

AMD

```
cl_int      clBuildProgram (cl_program program,
                            cl_uint num_devices,
                            const cl_device_id *device_list,
                            const char *options,
                            void (CL_CALLBACK *pfn_notify)(cl_program program,
                                                           void *user_data),
                            void *user_data)
```

- This function compiles and links an executable from the program object for each device in the context
  - If *device_list* is supplied, then only those devices are targeted
- Optional preprocessor, optimization, and other options can be supplied by *options*

Kernels

AMD

- A kernel is a function declared in a program that is executed on an OpenCL device
  - A kernel object is a kernel function along with its associated arguments
- A kernel object is created from a compiled program
- The user must explicitly associate arguments (memory objects, primitives, etc.) with the kernel object
- Kernel objects are created from a program object by specifying the name of the kernel function

Creating Kernels

AMD

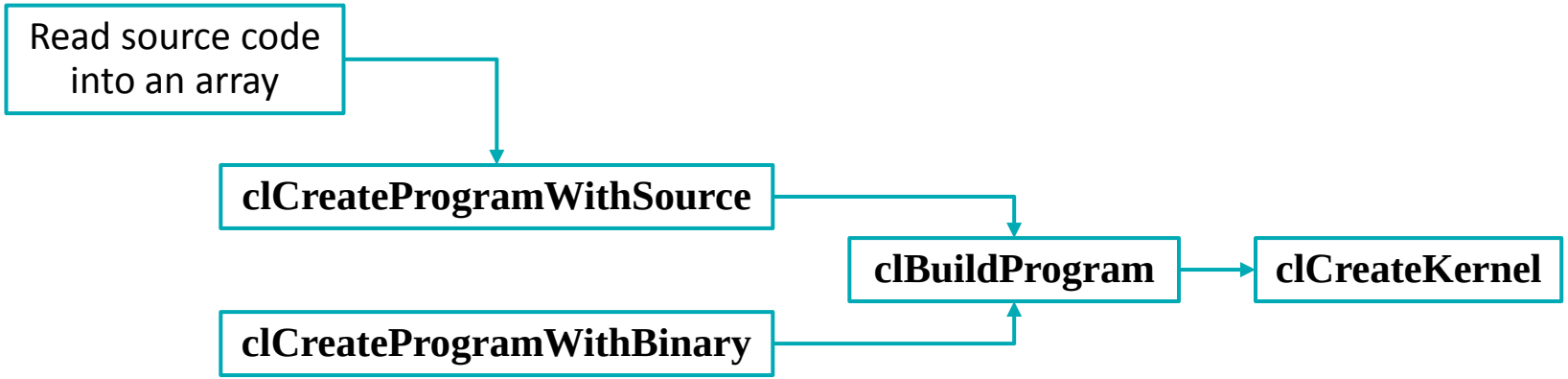
```
cl_kernel    clCreateKernel (cl_program program,
                             const char *kernel_name,
                             cl_int *errcode_ret)
```

- Creates a kernel from the given program
  - The kernel that is created is specified by a string that matches the name of the function within the program

Runtime Compilation of OpenCL kernels

AMD

- There is a high overhead for compiling programs and creating kernels
  - Each operation only has to be performed once (at the beginning of the program)
    - The kernel objects can be reused any number of times by setting different arguments



Reporting Compile Errors

AMD

- If a program fails to compile, OpenCL requires the programmer to explicitly ask for compiler output
  - A compilation failure is determined by an error value returned from **clBuildProgram**
  - Calling **clGetProgramBuildInfo** with the program object and the parameter **CL_PROGRAM_BUILD_STATUS** returns a string with the compiler output

Setting Kernel Arguments

AMD

- Memory objects and individual data values can be set as kernel arguments
- Kernel arguments are set by repeated calls to **clSetKernelArgs**

```
cl_int  clSetKernelArg (cl_kernel kernel,
                        cl_uint arg_index,
                        size_t arg_size,
                        const void *arg_value)
```

- Each call must specify
  - The index of the argument as it appears in the function signature, the size, and a pointer to the data
- Examples
  - clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&d_iImage);
  - clSetKernelArg(kernel, 1, sizeof(int), (void*)&a);

Execution Model

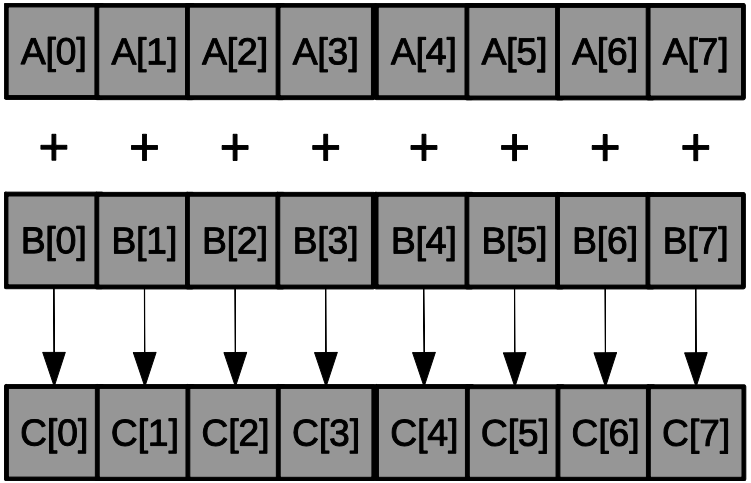
AMD

- Massively parallel programs are usually written so that each thread computes one part of a problem
  - For vector addition, we will add corresponding elements from two arrays, so each thread will perform one addition
  - If we think about the thread structure visually, the threads will usually be arranged in the same shape as the data

Execution Model

AMD

- Consider a simple vector addition of 8 elements
  - 2 input buffers (A, B) and 1 output buffer (C) are required
  - 1-dimensional problem in this case
  - Each thread is responsible for adding the indices corresponding to its ID



Execution Model

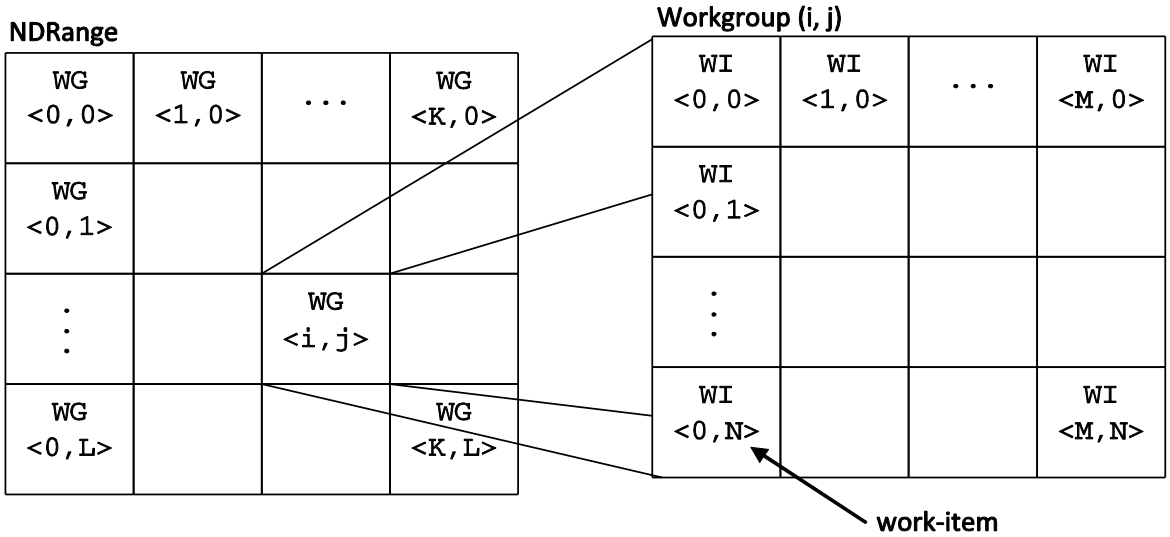
AMD

- OpenCL's execution model is designed to be scalable
- Each instance of a kernel is called a work-item (though "thread" is commonly used as well)
- Work-items are organized as work-groups
  - Work-groups are independent from one-another (this is where scalability comes from)
- An index space defines a hierarchy of work-groups and work-items

Execution Model

AMD

- Work-items can uniquely identify themselves based on:
  - A global ID (unique within the index space)
  - A work-group ID and a local ID within the work-group



Execution Model

AMD

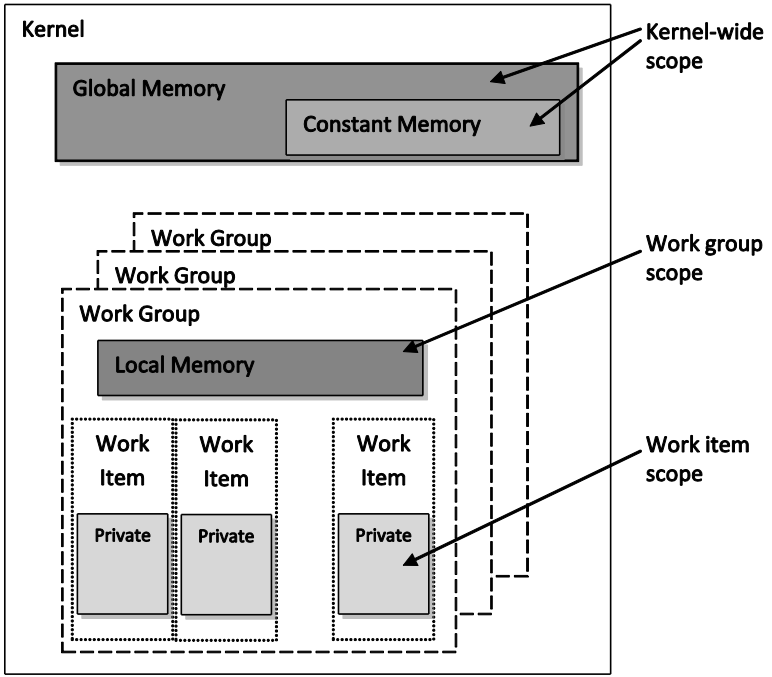
- API calls allow threads to identify themselves and their data
- Threads can determine their global ID in each dimension
  - get_global_id(dim)
  - get_global_size(dim)
- Threads can determine their work-group ID and local ID within the workgroup
  - get_group_id(dim)
  - get_num_groups(dim)
  - get_local_id(dim)
  - get_local_size(dim)
- Relationship between global and local sizes
  - get_global_size(0) == get_local_size(0) * get_num_groups(0)

Memory Model

AMD

- The OpenCL memory model defines the various types of memories (closely related to GPU memory hierarchy)

Memory	Description
Global	Accessible by all work-items
Constant	Read-only, global
Local	Local to a work-group
Private	Private to a work-item



Generic Address Space

AMD

- A single generic address space is added since OpenCL 2.0
- Supports conversion of pointers to and from private, local, and global address spaces

Writing Kernels

AMD

- One instance of the kernel is executed for each work-item
- Kernels:
  - Must begin with keyword `__kernel`
  - Must have return type `void`
  - Must declare the address space of each argument that is a memory object
  - Use API calls (such as `get_global_id()`) to determine which data a work-item will work on

Writing Kernels: Address Space Identifiers

AMD

- Inside a kernel, memory objects are specified using type qualifiers
  - __global: memory allocated from global address space
  - __constant: a special type of read-only memory
  - __local: memory shared by a work-group
  - __private: private per work-item memory
  - automatic variables are placed in the private address space by default
- Kernel arguments that are memory objects must be global, local, or constant

Writing Kernels: A vector addition kernel

AMD

- An example vector addition kernel that adds to arrays (A, B) and stores the result in a third array (C)

```
__kernel
void vectorAddition(__global int *A, __global int *B, __global int *C)
{
   // Get the work-item's unique ID
   int idx = get_global_id(0);

   // Add the corresponding locations of
   // 'A' and 'B', and store the result in 'C'.
  C[idx] = A[idx] + B[idx];
}
```

Writing Kernels: Executing Kernels on the device

AMD

- Need to set the dimensions of the index space, and (optionally) of the work-group sizes
- Kernels execute asynchronously from the host
  - **clEnqueueNDRangeKernel** just adds the kernel to the queue, but doesn't guarantee that it will start executing
- A thread structure defined by the index-space that is created
  - Each thread executes the same kernel on different data

Executing Kernels

AMD

```
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)
```

- Tells the device associated with a command queue to begin executing the specified kernel
- The global (index space) size must be specified and the local (work-group) sizes are optionally specified
  - In prior releases of OpenCL, the global size was required to be a multiple of the local size. OpenCL 2.0 has removed this restriction.

Copying Data Back

AMD

```
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)
```

- After kernel execution, we copy the data back from the device to the host
- Similar call as writing a buffer to a device, but data will be transferred back to the host
- A blocking flag assures that the data in *ptr* is valid before the call returns

Releasing Resources

AMD

- OpenCL objects should be freed after they are done being used
- There is a **clRelease{Resource}** command for most OpenCL types
  - Example: **clReleaseProgram**, **clReleaseMemObject**

Error Checking

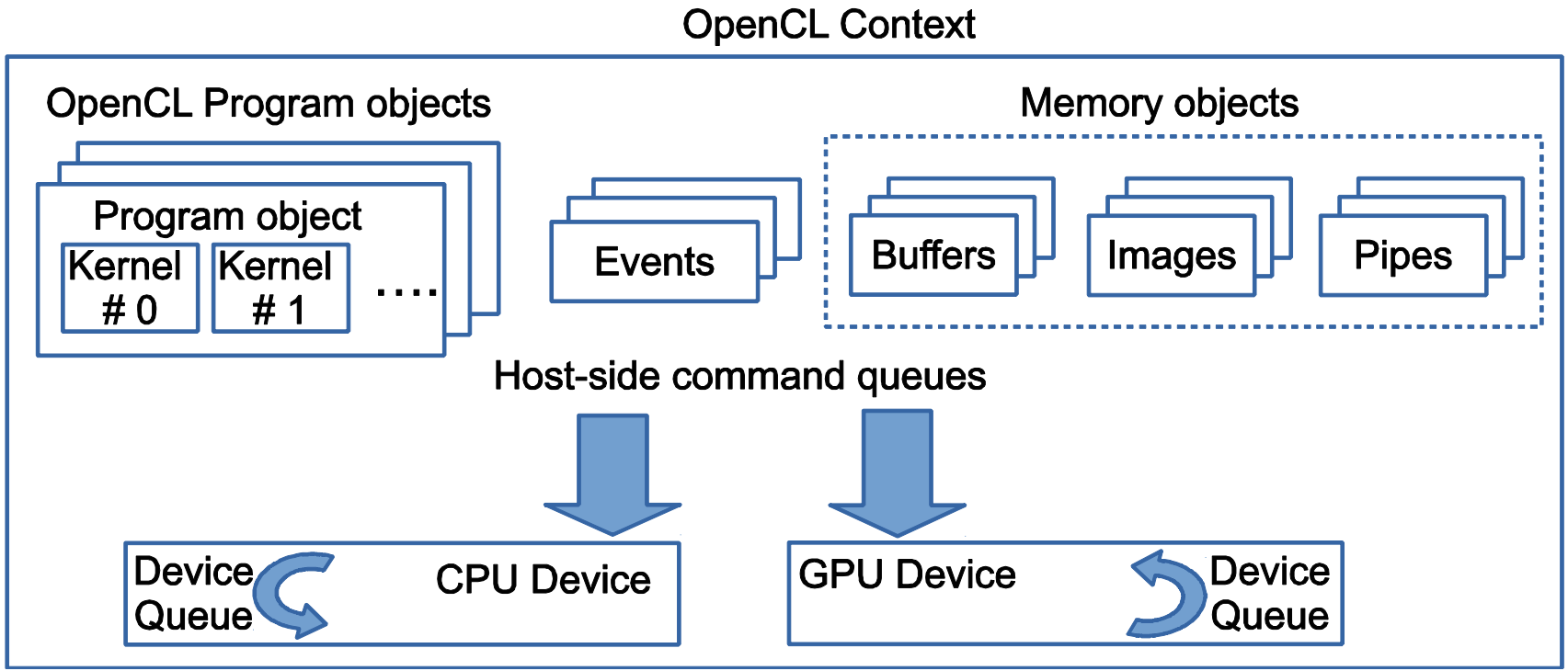
AMD

- OpenCL commands return error codes as negative integer values
  - Return value of 0 indicates **CL_SUCCESS**
  - Negative values indicates an error
  - cl.h defines meaning of each return value

CL_DEVICE_NOT_FOUND	-1
CL_DEVICE_NOT_AVAILABLE	-2
CL_COMPILER_NOT_AVAILABLE	-3
CL_MEM_OBJECT_ALLOCATION_FAILURE	-4
CL_OUT_OF_RESOURCES	-5

- Errors are sometimes reported asynchronously

Big Picture


OpenCL Context
OpenCL Program objects
Program object
Kernel # 0
Kernel # 1
....
Events
Memory objects
Buffers
Images
Pipes
Host-side command queues
Device Queue
CPU Device
GPU Device
Device Queue

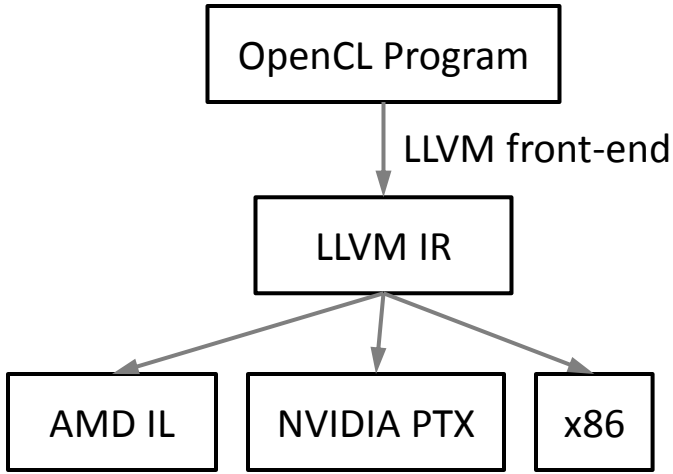
Programming Model

AMD

- Data parallel
  - One-to-one mapping between work-items and elements in a memory object
  - Work-groups can be defined explicitly or implicitly
- Task parallel
  - Kernel is executed independent of an index space
  - Other ways to express parallelism: enqueuing multiple tasks, using device-specific vector types, etc.

OpenCL Compilation System

- OpenCL kernels are usually converted into a binary representation of an intermediate language
- Common intermediate representations are the LLVM (Low Level Virtual Machine) IR or the Khronos SPIR
  - Kernels compiled to the IR
- LLVM is an open source compiler
  - Platform, OS independent
  - Multiple back ends



More information at http://llvm.org

Installable Client Driver

AMD

- ICD allows multiple implementations to co-exist
- Code only links to libOpenCL.so
- Application selects implementation at runtime
  - **clGetPlatformIDs** and **clGetPlatformInfo** examine the list of available implementations and select a suitable one
- Current GPU driver model does not easily allow devices from different vendors in same platform

Summary

AMD

- OpenCL provides an interface for the interaction of hosts with accelerator devices
- A context is created that contains all of the information and data required to execute an OpenCL program
  - Memory objects are created that can be moved on and off devices
  - Command queues allow the host to request operations to be performed by the device
  - Programs and kernels contain the code that devices need to execute