

# Writing Your First OpenCL Application

*Zhongliang Chen*

*3-6-2015*

# Agenda

- OpenCL models and objects (65 min.)
  - Lecture (35 min.)
  - Exercises (30 min.)
- Profiling and debugging (25 min.)
  - Lecture (15 min.)
  - Exercise (10 min.)

# OpenCL Architecture

- Portable parallel computing programming model
  - CPUs, GPUs, FPGAs, DSPs, etc.
- 4 models
  - Platform model
  - Execution model
  - Memory model
  - Programming model

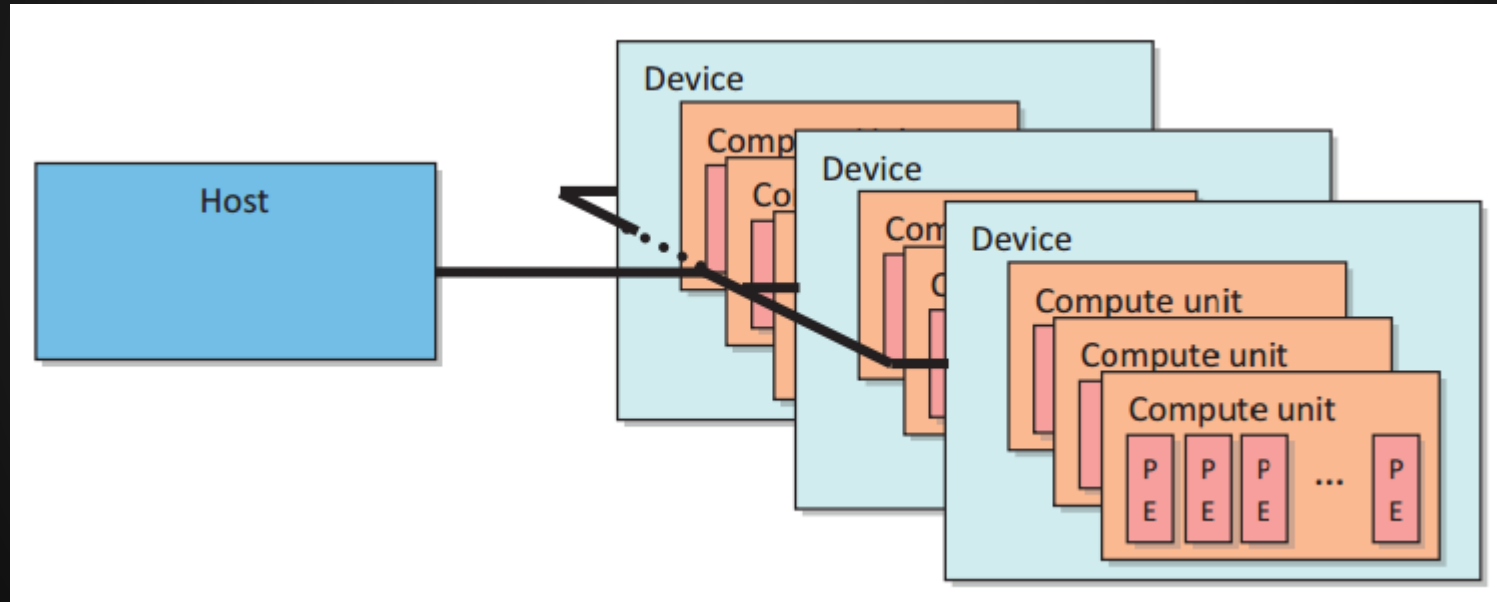
# OpenCL Architecture

- 7 common objects
  - Platform
  - Context
  - Device
  - Command queue
  - Program
  - Kernel
  - Memory object

# Platform Model

- Defined in OpenCL implementation, i.e., an OpenCL library
- Enables the host to interact with devices
- Currently each vendor provides only a single platform per implementation.

# Platform Model



# Platform Model

- Host
  - Processors an OpenCL library runs on
- Device
  - Processors an OpenCL library talks to
  - CPU: all cores are combined into a single device.
  - GPU: a GPU is a single device.

# Get a Platform

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

- Usually called twice
  - First call to get the number of platforms
  - Then space is allocated for the platform objects.
  - Second call to get a platform



# Exercise

## Implement GetPlatform()

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

# Context

- Environment for managing OpenCL objects and resources
- All OpenCL objects except platforms are managed by a context.
  - Devices
  - Programs
  - Kernels
  - Memory objects
  - Command queues

# Create A Context Before Getting Devices

- Created with a type of devices

```
cl_context clCreateContextFromType (const cl_context_properties *properties,  
                                   cl_device_type device_type,  
                                   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)
```

# Get Devices After Creating A Context

```
cl_int clGetContextInfo (cl_context context ,  
                        cl_context_info param_name ,  
                        size_t param_value_size ,  
                        void *param_value ,  
                        size_t *param_value_size_ret )
```

- Usually called twice
  - First call to get the number of devices
  - Then space is allocated for the device objects.
  - Second call to get one or more devices

# Get Devices Before Creating A Context

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

- Usually called twice
  - First call to get the number of devices
  - Then space is allocated for the device objects.
  - Second call to get one or more devices

# Create A Context After Getting Devices

- Created with one or more devices

```
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)
```

# Exercise

## Implement CreateContext()

```
cl_context clCreateContextFromType (const cl_context_properties *properties,  
                                   cl_device_type device_type,  
                                   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)
```

# Exercise

## Implement GetDevice()

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



# Command Queues

- Connecting host and device
- Each device has its own command queue.
- Commands
  - Synchronous
  - **Asynchronous**
  - **In-order execution**
  - Out-of-order execution

# Command Queues

- In-order execution
  - Each command is executed after the previous one has finished.
- Out-of-order execution
  - Commands are executed as soon as they are ready with no guarantees of their ordering
  - Events are commonly used for synchronization.

# Command Queues (v1.2)

```
cl_command_queue clCreateCommandQueue(cl_context context,  
                                       cl_device_id device,  
                                       cl_command_queue_properties properties,  
                                       cl_int *errcode_ret)
```

Command-Queue Properties	Description
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE	Determines whether the commands queued in the command-queue are executed in-order or out-of-order. If set, the commands in the command-queue are executed out-of-order. Otherwise, commands are executed in-order. See note below for more information.
CL_QUEUE_PROFILING_ENABLE	Enable or disable profiling of commands in the command-queue. If set, the profiling of commands is enabled. Otherwise profiling of commands is disabled. See <a href="#">clGetEventProfilingInfo</a> for more information.

# Exercise

## Implement CreateCommandQueue()

```
cl_command_queue clCreateCommandQueue(cl_context context,  
                                       cl_device_id device,  
                                       cl_command_queue_properties properties,  
                                       cl_int *errcode_ret)
```

# Programs

- A collection of OpenCL kernels
  - Source code or pre-compiled binary
  - Can contain constant data and auxiliary functions
- To create a program object
  - Read in a source code string or a binary

# Programs

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

```
cl_program clCreateProgramWithBinary ( cl_context context,  
                                       cl_uint num_devices,  
                                       const cl_device_id *device_list,  
                                       const size_t *lengths,  
                                       const unsigned char **binaries,  
                                       cl_int *binary_status,  
                                       cl_int *errcode_ret)
```

# Programs

- To build a program
  - Specify target devices.
  - Pass in compiler flags.
  - Check for compilation errors.

```
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)
```

# Programs

- If a program fails to compile, OpenCL requires programmers to explicitly ask for compiler output

```
cl_int clGetProgramBuildInfo (cl_program program,  
                             cl_device_id device,  
                             cl_program_build_info param_name,  
                             size_t param_value_size,  
                             void *param_value,  
                             size_t *param_value_size_ret)
```



# Exercise

## Implement CreateProgram()

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

```
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)
```

# Kernels

- Functions declared in a program and running on a device
- Created from a compiled program

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

# Kernels

- Arguments are explicitly associated with a kernel.
- Arguments can be memory objects or individual values.

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

# Exercise

Implement `CreateKernel()` and `SetKernelArg()`

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

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

# Runtime Compilation

- OpenCL compiles programs and creates kernels at run time, which may incur high overhead. So each operation has to be performed only once.

# What Are Left?

- Execute a kernel
  - Execution model
- Create memory objects and transfer data between host and device
  - Memory model

# Execution Model

- Thread structure in SIMD
  - Each work-item works on one part of a problem.

Thread structure

Vector addition

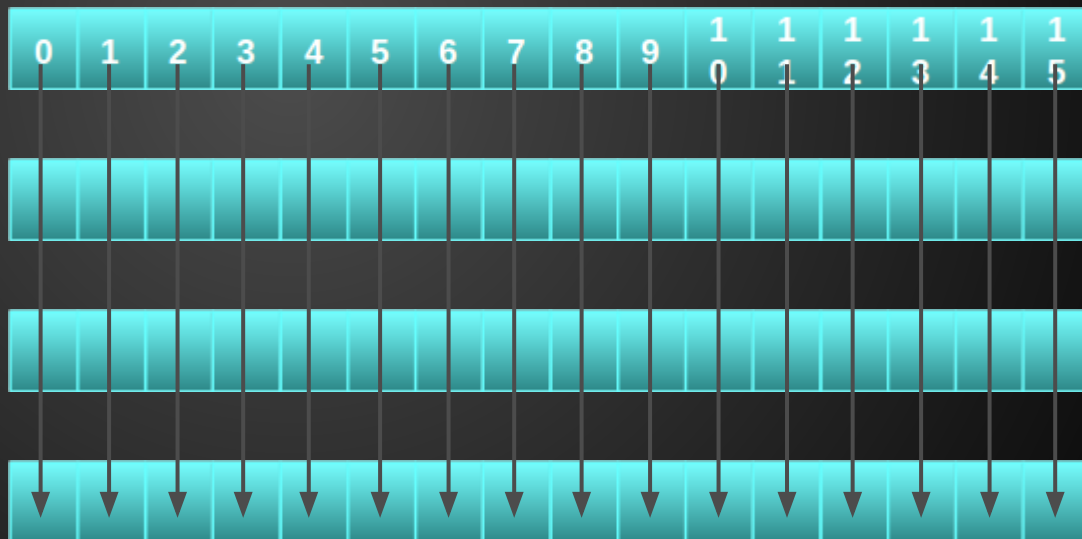
A

+

B

=

C



# Execution Model

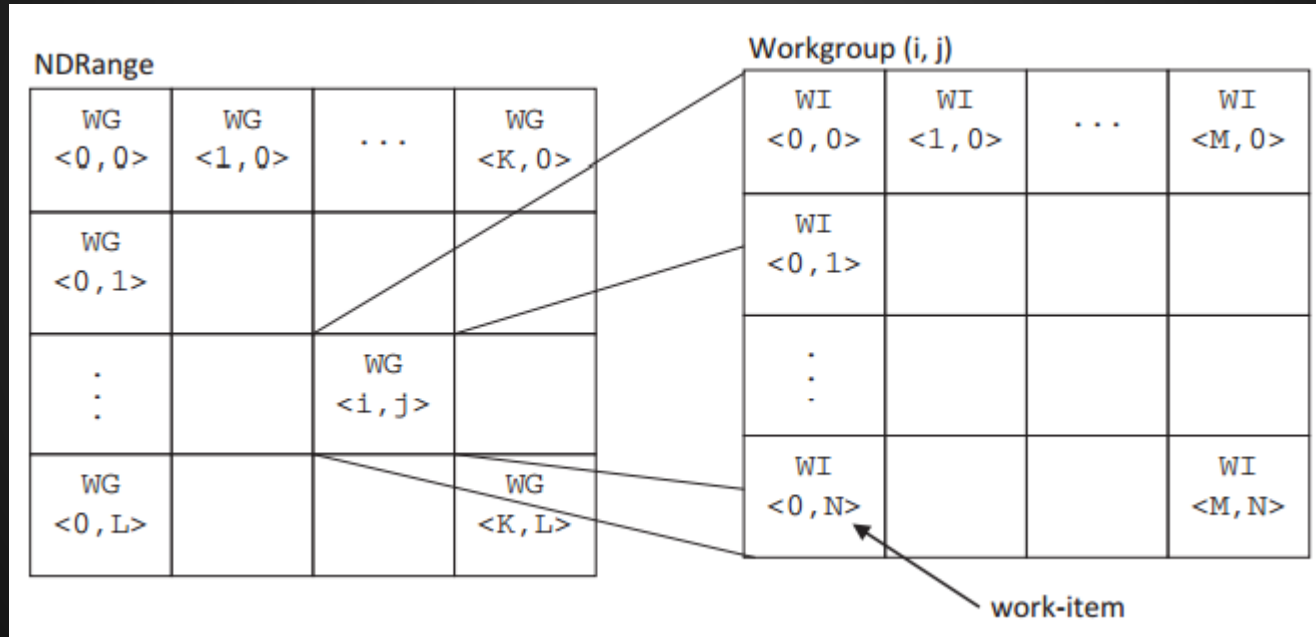
- Work-item: an instance of a kernel
- Work-group
  - Work-groups are independent of each other to guarantee scalability.
  - Scheduled to compute units and no guaranteed execution order
- Index space
  - A hierarchy of work-groups and work-items



# Execution Model

- Work-items' IDs
  - Global ID, unique in the index space
  - Local ID in the work-group
  - $GID = WG\_ID * WG\_SIZE + LID$

# Execution Model



# Execution Model

<code>get_work_dim</code>	Number of dimensions in use
<code>get_global_size</code>	Number of global work items
<code>get_global_id</code>	Global work item ID value
<code>get_local_size</code>	Number of local work items
<code>get_local_id</code>	Local work item ID
<code>get_num_groups</code>	Number of work groups
<code>get_group_id</code>	Work group ID
<code>get_global_offset</code>	Work offset

# Execution Model

```
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)
```

- Asynchronous execution
- A list of events can be used to specify prerequisite operations that must be completed before execution

```
cl_int clFinish ( cl_command_queue command_queue)
```

# Exercise

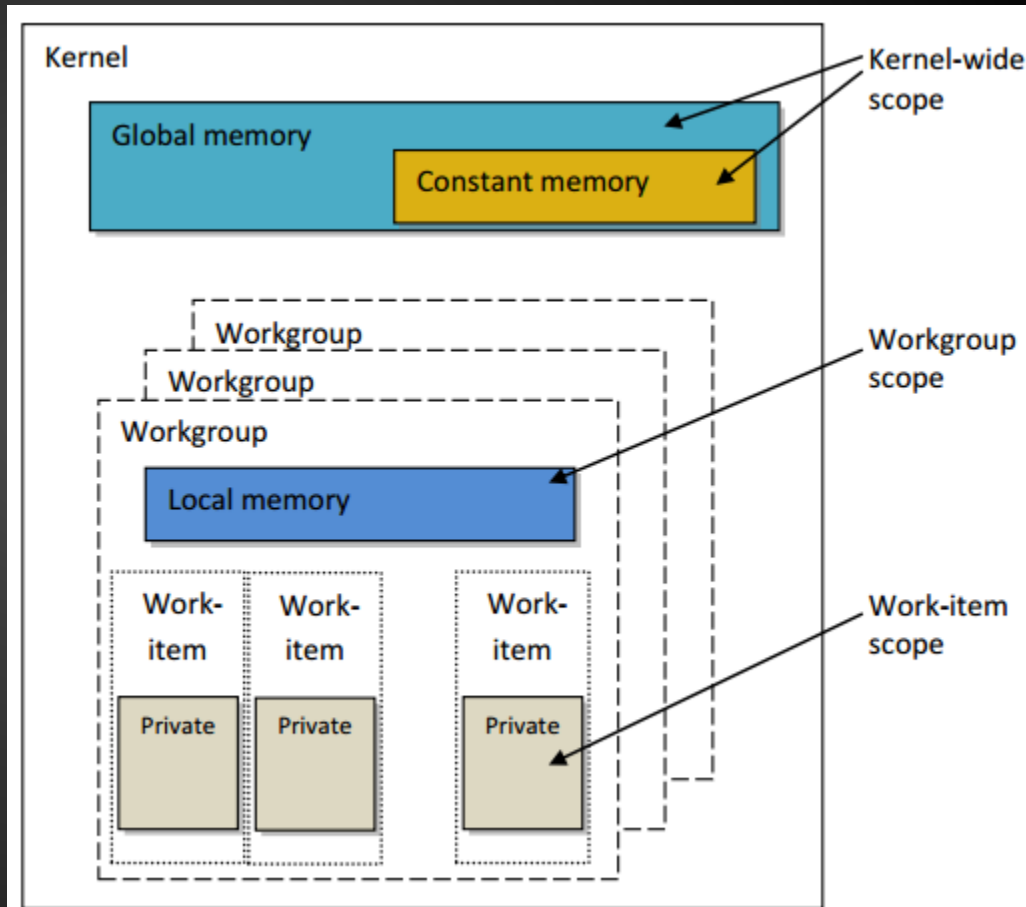
## Implement RunKernel()

```
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)
```

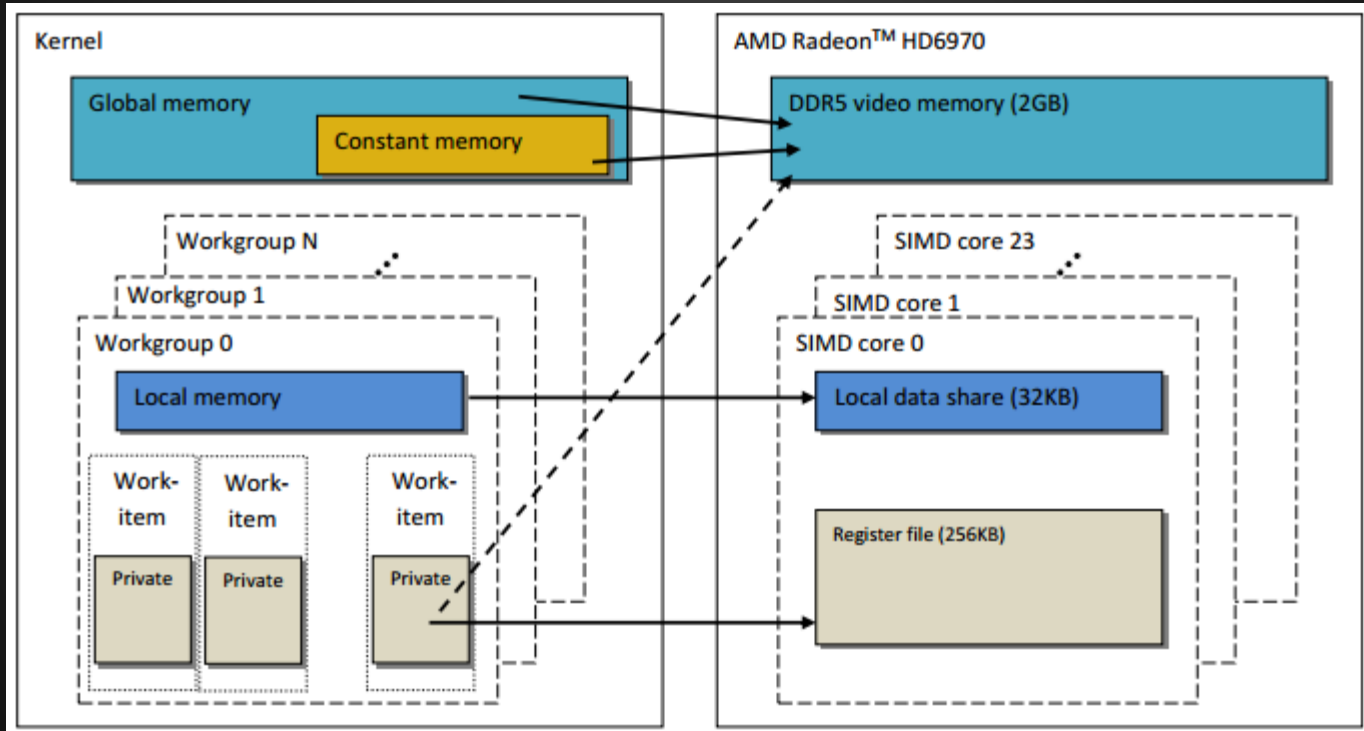
```
cl_int clFinish ( cl_command_queue command_queue)
```

# Memory Model

- Global mem
- Constant mem
- Local mem
- Private mem



# Memory Model



# Memory Model

- Memory objects are explicitly managed.
  - Between host memory and device memory
  - Between global memory and local memory

```
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)
```

```
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)
```



# Exercise

## Implement WriteToGPU() and ReadFromGPU()

```
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)
```

```
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)
```

# Writing a Kernel

- One instance of a kernel is created for each work-item.
- Kernels
  - Must begin with keyword `__kernel`
  - Must have return type `void`
  - Must declare the address space of each memory object argument
  - Use built-in work-item functions to get IDs, sizes, etc.

# Exercise

## Write VectorAddKernel

- Arguments: c, a, b, n
  - c: buffer, \_\_global
  - a: buffer, \_\_global
  - b: buffer, \_\_global
  - n: unsigned int
- Functionality: vector addition  $c = a + b$

# Programming Model

- Data parallel
  - One-to-one mapping between work-items and elements in a memory object
  - Work-group defined explicitly or implicitly
- Task parallel
  - Kernel executed independent of an index space
- Synchronization
  - Between work-items in a work-group
  - Between command queues in a context

# Wrap-Up

- 4 models
  - Platform, Execution, Memory, Programming
- 7 common objects
  - Platform, Context, Device
  - Command Queue, Program, Kernel, Memory Objects

# Exercise

Make VectorAdd example work.

# Agenda

- OpenCL models and objects (65 min.)
  - Lecture (35 min.)
  - Exercises (30 min.)
- Profiling and debugging (25 min.)
  - Lecture (15 min.)
  - Exercise (10 min.)

# Command Queues With Profiling Enabled

```
cl_command_queue clCreateCommandQueue(cl_context context,  
                                       cl_device_id device,  
                                       cl_command_queue_properties properties,  
                                       cl_int *errcode_ret)
```

Command-Queue Properties	Description
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE	Determines whether the commands queued in the command-queue are executed in-order or out-of-order. If set, the commands in the command-queue are executed out-of-order. Otherwise, commands are executed in order. See note below for more information.
CL_QUEUE_PROFILING_ENABLE	Enable or disable profiling of commands in the command-queue. If set, the profiling of commands is enabled. Otherwise profiling of commands is disabled. See <a href="#">clGetEventProfilingInfo</a> for more information.



# Events For Profiling

```
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)
```

# Profiling API

```
cl_int clGetEventProfilingInfo (cl_event event,  
                                cl_profiling_info param_name,  
                                size_t param_value_size,  
                                void *param_value,  
                                size_t *param_value_size_ret)
```

cl_profiling_info	Return Type	Info. returned in <i>param_value</i>
CL_PROFILING_COMMAND_QUEUED	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by <i>event</i> is enqueued in a command-queue by the host.
CL_PROFILING_COMMAND_SUBMIT	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by <i>event</i> that has been enqueued is submitted by the host to the device associated with the command-queue.
CL_PROFILING_COMMAND_START	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by <i>event</i> starts execution on the device.
CL_PROFILING_COMMAND_END	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by <i>event</i> has finished execution on the device.

# Exercise

## Profile the kernel execution time

Step 1: Enable profiling for the command queue.

Step 2: Create an event and associate it with  
`clEnqueueNDRangeKernel`

Step 3: Use profiling APIs to get time

# Demo

See how much a GPU is faster than a CPU.

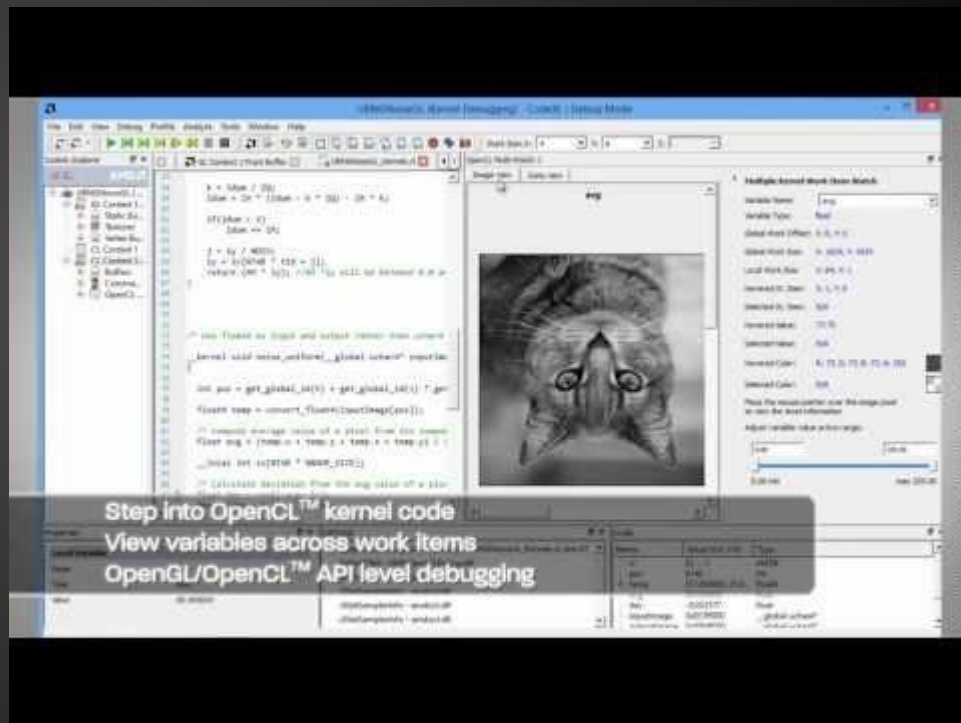
# Debugging Kernels With `printf()`

- AMD GPUs support printing in kernels.
  - `cl_amd_printf` extension has to be enabled.
  - Remember that a kernel has many instances.

Generally one wants to check out only a few work-items.
- Information to be printed is buffered until the kernel completes and then transferred back to the host.
  - The kernel must be finished.

# AMD CodeXL

- Integrated tool for debugging and profiling



# Programming Multiple Devices

- Single context, multiple devices
  - Simple
  - Memory objects are common in a context
- Multiple context, multiple devices
  - Computing on a cluster
- Considerations for CPU-GPU heterogeneous computing
  - Scheduling, load balancing

# Summary

- OpenCL models
  - Platform, Execution, Memory, Programming
- OpenCL common objects
  - Platform, Context, Device
  - Command Queue, Program, Kernel, Memory Object
- Profiling and debugging



# Next Session (Leiming Yu)

- Global memory is off chip and so slow, which is a very bad idea for data reuse. Local memory is on chip and much faster. How to use it?
- Vector addition is just a toy. How to write and optimize a real application?

# Reference

- OpenCL university kit
  - By Perhaad Mistry and Dana Schaa
- AMD OpenCL Programming User Guide
  - rev 1.0 Beta
- Heterogeneous Computing with OpenCL
  - By Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry & Dana Schaa

# Thank you!

*Questions?*

# OpenCL 2.0 Main Features

- **SVM: Shared Virtual Memory**
  - Pointer-containing data structures can be easily shared between host and device.
- **DP: Dynamic Parallelism**
  - Kernels can enqueue kernels without host interactions.
- **Pipes**
  - Pipes enable data transfers between kernels without host interactions.