Introduction to OpenCL

Cliff Woolley, NVIDIA
Developer Technology Group
Welcome to the OpenCL Tutorial!

- OpenCL Platform Model
- OpenCL Execution Model
- Mapping the Execution Model onto the Platform Model
- Introduction to OpenCL Programming
- Additional Information and Resources
Design Goals of OpenCL

- Use all computational resources in the system
  - CPUs, GPUs and other processors as peers

- Efficient parallel programming model
  - Based on C99
  - Data- and task- parallel computational model
  - Abstract the specifics of underlying hardware
  - Specify accuracy of floating-point computations

- Desktop and Handheld Profiles
OPENCL PLATFORM MODEL
It’s a Heterogeneous World

- A modern platform includes:
  - One or more CPUs
  - One or more GPUs
  - Optional accelerators (e.g., DSPs)

GMCH = graphics memory control hub
ICH = Input/output control hub
OpenCL Platform Model

- Host
- Compute Unit
- Compute Device
- Processing Element
- Computational Resources
OpenCL Platform Model

- **Host**
- **Processing Element**
- **Compute Device**
- **Compute Unit**

**Computational Resources**
OpenCL Platform Model on CUDA Compute Architecture

- CPU
- Host
- Compute Device
- Compute Unit
- CUDA-Enabled GPU
- CUDA Streaming Multiprocessor
- Processing Element
- CUDA Streaming Processor
Anatomy of an OpenCL Application

OpenCL Application

**Host Code**
- Written in C/C++
- Executes on the host

**Device Code**
- Written in OpenCL C
- Executes on the device

Host code sends commands to the Devices:
- to transfer data between host memory and device memories
- to execute device code
Anatomy of an OpenCL Application

- **Serial** code executes in a **Host** (CPU) thread
- **Parallel** code executes in many **Device** (GPU) threads across multiple processing elements
OPENCL EXECUTION MODEL
Decompose task into **work-items**

- Define N-dimensional computation domain
- Execute a *kernel* at each point in computation domain

**Traditional loop as a function in C**

```c
void trad_mul(int n,
        const float *a,
        const float *b,
        float *c)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] * b[i];
}
```

**OpenCL C kernel**

```c
__kernel void dp_mul(__global const float *a,
                    __global const float *b,
                    __global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over n “work items”
```
An N-dimension domain of work-items

Define the “best” N-dimensioned index space for your algorithm

• Kernels are executed across a global domain of work-items

• Work-items are grouped into local work-groups
  – Global Dimensions: 1024 x 1024 (whole problem space)
  – Local Dimensions: 32 x 32 (work-group ... executes together)
OpenCL Execution Model

The application runs on a **Host** which submits work to the **Devices**

- **Work-item**: the basic unit of work on an OpenCL device
- **Kernel**: the code for a work-item (basically a C function)
- **Program**: Collection of kernels and other functions (analogous to a dynamic library)
OpenCL Execution Model

The application runs on a Host which submits work to the Devices

- **Context**: The environment within which work-items execute; includes devices and their memories and command queues
- **Command Queue**: A queue used by the Host application to submit work to a Device (e.g., kernel execution instances)
  - Work is queued in-order, one queue per device
  - Work can be executed in-order or out-of-order
MAPPING THE EXECUTION MODEL ONTO THE PLATFORM MODEL
Kernel Execution on Platform Model

- Each work-item is executed by a compute element
- Each work-group is executed on a compute unit
- Several concurrent work-groups can reside on one compute unit depending on work-group’s memory requirements and compute unit’s memory resources
- Each kernel is executed on a compute device
OpenCL Memory Model

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup
- **Global/Constant Memory**
  - Visible to all workgroups
- **Host Memory**
  - On the CPU

Memory management is Explicit
You must move data from host -> global -> local ... and back
INTRODUCTION TO OPENCL PROGRAMMING
OpenCL Framework

- Platform layer
  - Platform query and context creation
- Compiler for OpenCL C
- Runtime
  - Memory management and command execution within a context
OpenCL Framework

Programs

Kernels

Memory Objects

Command Queues

Context

CPU

GPU

__kernel void dp_mul (global const float *a, global const float *b, global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
}"
OpenCL Framework: Platform Layer

Context

Programs

Kernels

Memory Objects

Command Queues

CPU

GPU

Compile code

Create data & arguments

Send to execution

_kernel void
  dp_mul(global const float *a,
        global const float *b,
        global float *c)
  {
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
  }

_context

Images

Buffers

In Order Queue

Out of Order Queue

Compute Device

dp_mul

CPU program binary

arg[0] value

arg[1] value

arg[2] value

GPU program binary

dp_mul

arg[0] value

arg[1] value

arg[2] value

Third party names are the property of their owners.
OpenCL Framework: Platform Layer

- Query platform information
  - `clGetPlatformInfo()`: profile, version, vendor, extensions
  - `clGetDeviceIDs()`: list of devices
  - `clGetDeviceInfo()`: type, capabilities

- Create an OpenCL context for one or more devices

```
Context = cl_context
One or more devices
  cl_device_id
Memory and device code shared by these devices
  cl_mem  cl_program
Command queues to send commands to these devices
  cl_command_queue
```
Platform Layer:
Context Creation (simplified)

// Get the platform ID
cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);

// Get the first GPU device associated with the platform
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);

// Create an OpenCL context for the GPU device
cl_context context;
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
Platform Layer: Error Handling, Resource Deallocation

- **Error handling:**
  - All host functions return an error code
  - Context error callback

- **Resource deallocation**
  - Reference counting API: `clRetain*()`, `clRelease*()`

- **Both are removed from code samples for clarity**
  - Please see SDK samples for complete code
OpenCL Framework: OpenCL C

Kernel:

```c
__kernel void dp_mul(global const float *a, global const float *b, global float *c) {
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
}
```

Programs:

- `dp_mul` (CPU program binary)
- `dp_mul` (GPU program binary)

Kernels:

- `dp_mul`

Memory Objects:

- Images
- Buffers

Command Queues:

- In Order Queue
- Out of Order Queue

Compute Device:

- arg[0]: value
- arg[1]: value
- arg[2]: value

Compile code

Create data & arguments

Send to execution

© Copyright Khronos Group, 2010
OpenCL C

- Derived from ISO C99 (with some restrictions)
- Language Features Added
  - Work-items and work-groups
  - Vector types
  - Synchronization
  - Address space qualifiers
- Also includes a large set of built-in functions
  - Image manipulation
  - Work-item manipulation
  - Math functions
OpenCL C Language Restrictions

- Pointers to functions are not allowed
- Pointers to pointers allowed within a kernel, but not as an argument
- Bit-fields are not supported
- Variable-length arrays and structures are not supported
- Recursion is not supported
- Writes to a pointer to a type less than 32 bits are not supported*
- Double types are not supported, but reserved
- 3D Image writes are not supported

Some restrictions are addressed through extensions
OpenCL C Optional Extensions

- Extensions are optional features exposed through OpenCL
- The OpenCL working group has already approved many extensions to the OpenCL specification:
  - Double precision floating-point types (Section 9.3)
  - Built-in functions to support doubles
  - Atomic functions (Section 9.5, 9.6, 9.7)
  - Byte-addressable stores (write to pointers to types < 32-bits) (Section 9.9)
  - 3D Image writes (Section 9.8)
  - Built-in functions to support half types (Section 9.10)

Now core features in OpenCL 1.1
A kernel is a function executed for each work-item

```c
__kernel void square(__global float* input, __global float* output)
{
    int i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 1 1 0 9 2 4 1</td>
<td>36 1 1 0 81 4 16 1</td>
</tr>
<tr>
<td></td>
<td>6 8 2 2 5</td>
</tr>
</tbody>
</table>

get_global_id(0) = 7
Work-items and work-groups

input

| 6 | 1 | 1 | 0 | 9 | 2 | 4 | 1 | 1 | 9 | 7 | 6 | 8 | 2 | 2 | 5 |

- `get_work_dim()` = 1
- `get_global_size(0)` = 16
- `get_num_groups(0)` = 2

work-group

- `get_group_id(0)` = 0
- `get_local_size(0)` = 8
- `get_group_id(0)` = 1
- `get_local_size(0)` = 8

work-item

- `get_local_id(0)` = 3
- `get_global_id(0)` = 11
OpenCL C Data Types

- **Scalar data types**
  - char, uchar, short, ushort, int, uint, long, ulong, float
  - bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void, half (storage)

- **Image types**
  - image2d_t, image3d_t, sampler_t

- **Vector data types**
  - Vector lengths 2, 3, 4, 8, 16 (char2, ushort4, int8, float16, double2, ...)
  - Endian safe
  - Aligned at vector length
  - Vector operations

3-vectors new in OpenCL 1.1

double is an optional type in OpenCL
OpenCL C Synchronization Primitives

- Built-in functions to order memory operations and synchronize execution:
  - `mem_fence(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE)`
    - waits until all reads/writes to local and/or global memory made by the calling work-item prior to `mem_fence()` are visible to all threads in the work-group
  - `barrier(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE)`
    - waits until all work-items in the work-group have reached this point and calls `mem_fence(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE)`

- Used to coordinate accesses to local or global memory shared among work-items
OpenCL C Kernel Example

```c
__kernel void dp_mul(__global const float *a,
                     __global const float *b,
                     __global float *c,
                     int N)
{
    int id = get_global_id(0);
    if (id < N)
        c[id] = a[id] * b[id];
}
```
OpenCL Framework: Runtime

```
__kernel void dp_mul(global const float *a, global const float *b, global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

Images

Buffers

In Order Queue

Out of Order Queue

Compute Device

Command Queues

Memory Objects

Kernels

Programs

Context

CPU

GPU

Compile code

Create data & arguments

Send to execution

Third party names are the property of their owners.
OpenCL Framework: Runtime

- Command queues creation and management
- Device memory allocation and management
- Device code compilation and execution
- Event creation and management (synchronization, profiling)
OpenCL Runtime: Kernel Compilation

Programs

Kernels

Memory Objects

Command Queues

Context

CPU

GPU

In Order Queue

Out of Order Queue

GPU

Compute Device

In Order Queue

Out of Order Queue

Compile code

Create data & arguments

Send to execution

__kernel void dp_mul(global const float *a, global const float *b, global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
}
Kernel Compilation

- A `cl_program` object encapsulates some source code (with potentially several kernel functions) and its last successful build
  - `clCreateProgramWithSource()` // Create program from source
  - `clBuildProgram()` // Compile program

- A `cl_kernel` object encapsulates the values of the kernel’s arguments used when the kernel is executed
  - `clCreateKernel()` // Create kernel from successfully compiled program
  - `clSetKernelArg()` // Set values of kernel’s arguments
Kernel Compilation

// Build program object and set up kernel arguments
const char* source = "__kernel void dp_mul(__global const float *a, \n"  "  __global const float *b, \n"  "  __global float *c, \n"  "  int N) \n"  "{ \n"  "    int id = get_global_id (0); \n"  "    if (id < N) \n"  "      c[id] = a[id] * b[id]; \n"  "  } \n";
cl_program program = clCreateProgramWithSource(context, 1, &source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "dp_mul", NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&d_buffer);
clSetKernelArg(kernel, 1, sizeof(int), (void*)&N);
OpenCL Runtime: Memory Objects

Context

Programs

Kernels

Memory Objects

Command Queues

CPU

GPU

Kernel void
dp_mul(global const float *a,
global const float *b,
global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}

dp_mul
CPU program binary

dp_mul
GPU program binary

Images

Buffers

In Order Queue

Out of Order Queue

Compute Device

Compile code

Create data & arguments

Send to execution

© Copyright Khronos Group, 2010

Third party names are the property of their owners.
Memory Objects

- Two types of memory objects (*cl_mem*):
  - **Buffer** objects
  - **Image** objects

- Memory objects can be copied to host memory, from host memory, or to other memory objects

- Regions of a memory object can be accessed from host by mapping them into the host address space
Buffer Object

- One-dimensional array
- Elements are scalars, vectors, or any user-defined structures
- Accessed within device code through pointers
Image Object

- Two- or three-dimensional array
- Elements are 4-component vectors from a list of predefined formats
- Accessed within device code via built-in functions (storage format not exposed to application)
  - Sampler objects are used to configure how built-in functions sample images (addressing modes, filtering modes)
- Can be created from OpenGL texture or renderbuffer
OpenCL Runtime: Command Queues

Context

Programs

Kernels

Memory Objects

Command Queues

CPU

GPU

In Order Queue

Out of Order Queue

Compute Device

Compile code

Create data & arguments

Send to execution

_kernel void
dp_mul(global const float *a,
global const float *b,
global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
}
Commands

- Memory copy or mapping
- Device code execution
- Synchronization point
Command Queue

- Sequence of commands scheduled for execution on a specific device
  - Enqueuing functions: `clEnqueue*()`
  - Multiple queues can execute on the same device

- Two modes of execution:
  - In-order: Each command in the queue executes only when the preceding command has completed (including memory writes)
  - Out-of-order: No guaranteed order of completion for commands

```c
// Create a command-queue for a specific device
cl_command_queue cmd_queue = clCreateCommandQueue(context, device_id, 0, NULL);
```
Data Transfer between Host and Device

// Create buffers on host and device
size_t size = 100000 * sizeof(int);
int* h_buffer = (int*)malloc(size);
cl_mem d_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, size, NULL, NULL);
...

// Write to buffer object from host memory
clEnqueueWriteBuffer(cmd_queue, d_buffer, CL_FALSE, 0, size, h_buffer, 0, NULL, NULL);
...

// Read from buffer object to host memory
clEnqueueReadBuffer(cmd_queue, d_buffer, CL_TRUE, 0, size, h_buffer, 0, NULL, NULL);

Blocking?  Offset  Event synch
Kernel Execution: NDRange

- Host code invokes a kernel over an index space called an *NDRange*
  - NDRange = “N-Dimensional Range” of work-items
  - NDRange can be a 1-, 2-, or 3-dimensional space
  - Work-group dimensionality matches work-item dimensionality
// Set number of work-items in a work-group
size_t localWorkSize = 256;
int numWorkGroups = (N + localWorkSize - 1) / localWorkSize;  // round up
size_t globalWorkSize = numWorkGroups * localWorkSize;  // must be evenly divisible by localWorkSize
clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, &globalWorkSize, &localWorkSize, 0, NULL, NULL);
Command Synchronization

- **Queue barrier command**: `clEnqueueBarrier()`
  - Commands after the barrier start executing only after all commands before the barrier have completed.

- **Events**: a `cl_event` object can be associated with each command.
  - Commands return events and obey event waitlists.
    - `clEnqueue* (...)`, `num_events_in_waitlist`, `event_waitlist`, `*event`);
  - Any commands (or `clWaitForEvents()`) can wait on events before executing.
  - Event object can be queried to track execution status of associated command and get profiling information.

- **Some `clEnqueue*()` calls can be optionally blocking**
  - `clEnqueueReadBuffer(..., CL_TRUE, ...)`;
Synchronization: Queues & Events

- You must explicitly synchronize between queues
  - Multiple devices each have their own queue
  - Possibly multiple queues per device
  - Use events to synchronize
ADDITIONAL INFORMATION AND RESOURCES
Next Steps

- Begin hands-on development with our publicly available OpenCL driver and GPU Computing SDK
- Read the OpenCL Specification and the extensive documentation provided with the SDK
- Read and contribute to OpenCL forums at Khronos and NVIDIA
NVIDIA OpenCL Resources

- NVIDIA OpenCL Web Page:

- NVIDIA OpenCL Forum:

- NVIDIA driver, profiler, code samples for Windows and Linux:
Khronos OpenCL Resources

- OpenCL Specification

- OpenCL Registry
  - http://www.khronos.org/registry/cl/

- OpenCL Developer Forums
  - http://www.khronos.org/message_boards/

- OpenCL Quick Reference Card

- OpenCL Online Man pages
  - http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/
OpenCL Books

• The OpenCL Programming Book
  – Available now: search for OpenCL on Amazon

• OpenCL Programming Guide - The “Red Book” of OpenCL
  – Coming in July 2011; rough cut available on SafariBooks
  – http://my.safaribooksonline.com/9780132488006
Questions?