SIGGRAPHASIA2009

革新の波動

the pulse of innovation
Advanced OpenCL Event Model Usage

Derek Gerstmann
University of Western Australia
http://local.wasp.uwa.edu.au/~derek
OpenCL Event Model Usage

• Outline
  – Execution Model
  – Usage Patterns
  – Synchronisation
  – Event Model
  – Examples
Execution Model
Execution Model

- Command queues are used to submit work to a device
  - Commands are queued in-order
  - Commands execute in-order or out-of-order
Execution Model

- Explicit synchronisation is required for ...

- Out-of-order command queues

- Multiple command queues
Single Device In-Order Usage Model

- **1x In-Order Queue, 1x Context, 1x Device**
  - Simple and straightforward in-order queue
  - All commands execute on single device
  - All memory operations occur in single memory pool
Single Device In-Order Usage Model

```c
cl_uint num_devices;
cl_device_id devices[1];

err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[0], &num_devices);
context = clCreateContext(0, 1, devices, NULL, NULL, &err);

cl_command_queue queue_cpu;
queue_cpu = clCreateCommandQueue(context, devices[0], 0 /* IN-ORDER */, &err);

/* ... enqueue work ... */
```
Single Device In-Order Usage Model

- Commands
- In-order queue
- Context
  - CPU
  - MEM

Queue
- Execution
  - Memory
  - WRITE
  - READ
- Device executes commands after the previous one finishes
- Memory transactions have consistent view
Other Usage Patterns Which Require Synchronisation
1x Out-of-Order Queue, 1x Context, 1x Device

- Same as before but with an out-of-order queue
- All commands execute on single device
- All memory operations occur in single memory pool
- Execution order has no guarantees
`cl_uint num_devices;
cl_device_id devices[1];`

```
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[0], &num_devices);
context = clCreateContext(0, 1, devices, NULL, NULL, &err);
```

```
cl_command_queue queue_cpu;
queue_cpu = clCreateCommandQueue(context, devices[0],
                               CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);
```

`/* ... enqueue work ... */`
Single Device Out-of-Order Usage Model

- commands
- out-of-order queue
- CPU
- MEM
- queue
- execution
- memory
- WRITE
- READ
- time
• Device starts executing commands as soon as it can
• Memory transactions overlap and clobber data!
Separate Multi-Device Usage Model

- **2x In-Order Queues, 2x Separate Contexts, 2x Devices**
  - Commands execute on the device associated with the queue
  - Memory operations occur in two separate memory pools
  - Completely separate queues in different contexts
Separate Multi-Device Usage Model

cl_uint num_devices;
cl_device_id devices[2];

err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[0], &num_devices);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[1], &num_devices);
context_cpu = clCreateContext(0, 1, &devices[0], NULL, NULL, &err);
context_gpu = clCreateContext(0, 1, &devices[1], NULL, NULL, &err);

cl_command_queue queue_cpu, queue_gpu;
queue_cpu = clCreateCommandQueue(context_cpu, devices[0], 0 /* IN-ORDER */, &err);
queue_gpu = clCreateCommandQueue(context_gpu, devices[1], 0 /* IN-ORDER */, &err);
Separate Multi-Device Usage Model

- In-order CPU queue
- In-order GPU queue

Commands flow through separate queues for CPU and GPU memory.

Time progression shows execution and memory usage across devices.
Separate Multi-Device Usage Model

- Command queues can’t synchronise across contexts!
- Neither device sees the memory pool of the other!
- Won’t work unless you use `clFinish()`, and copy across contexts!
- In short, this is not what you want! Don’t do this!
Cooperative Multi-Device Usage Model

- **2x In-Order Queues, 1x Combined Context, 2x Devices**
  - Commands execute on the device associated with the queue
  - Memory operations occur in one combined memory pool
  - Combined memory pool being modified by multiple devices
cl_uint num_devices;
cl_device_id devices[2];

err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[0], &num_devices);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[1], &num_devices);
context = clCreateContext(0, 2, devices, NULL, NULL, &err);

cl_command_queue queue_cpu, queue_gpu;
queue_cpu = clCreateCommandQueue(context, devices[0], 0 /* IN-ORDER */, &err);
queue_gpu = clCreateCommandQueue(context, devices[1], 0 /* IN-ORDER */, &err);
Cooperative Multi-Device Usage Model

- commands
- in-order cpu queue
- in-order gpu queue

- CPU
- GPU
- MEM

queue
execution
queue
execution
memory

time

SIGGRAFAIA2009
the pulse of innovation
Still wrong!
Both devices start executing commands as soon as they can!
Memory transactions overlap and clobber combined memory!
Synchronisation Mechanisms
Synchronisation Mechanisms

- Command Queue Control Methods
  - `clFlush()`
    Send all commands in the queue to the compute device
  - `clFinish()`
    Flush, and then wait for all commands to finish
Synchronisation Mechanisms

- **Command Queue Control Methods**
  - `clFlush()`
    Send all commands in the queue to the compute device
  - `clFinish()`
    Flush, and then wait for all commands to finish

- Both are brute force, and lack fine-grained control
Synchronisation Mechanisms

- Command Execution Barriers

  - `clEnqueueBarrier()`

    Enqueue a fence which insures that all preceding commands in the queue are complete before any commands which get enqueued afterwards are processed.
Synchronisation Mechanisms

- **Event Based Synchronisation**

  - **Event objects**
    Unique objects which can be used to determine command status

  - **Event wait lists**
    Array of events used to indicate commands which must be complete before further commands are allowed to proceed
Event-Based Synchronisation

- Event objects are used to determine command status

- `clGetEventStatus()`
  Returns command status for an event

  - `CL_QUEUED` - Command is in a queue
  - `CL_SUBMITTED` - Command has been submitted to device
  - `CL_RUNNING` - Device is currently executing command
  - `CL_COMPLETE` - Command has finished execution
Event-Based Synchronisation

- All `clEnqueue()` methods can return event objects

- `clEnqueueNDRangeKernel`
- `clEnqueueTask`
- `clEnqueueNativeKernel`
- `clEnqueueCopyImageToBuffer`
- `clEnqueueCopyBufferToImage`
- `clEnqueueRead {Image|Buffer}`
- `clEnqueueWrite {Image|Buffer}`
- `clEnqueueMap {Image|Buffer}`
- `clEnqueueCopy {Image|Buffer}`
- `clEnqueueCopyImageToBuffer`
- `clEnqueueCopyBufferToImage`
Event-Based Synchronisation

- All `clEnqueue()` methods can return event objects

```c
cl_event returned_event;

err = clEnqueueReadBuffer(queue, buffer, 
                          CL_FALSE /* non-blocking */, 
                          0, 0, ptr, , 0, 
                          &returned_event);
```
Event-Based Synchronisation

- All clEnqueue() methods can return event objects

```c
cl_event returned_event;

err = clEnqueueReadBuffer(queue, buffer,
                          CL_FALSE /* non-blocking */, 
                          0, 0, ptr, , 0,
                          &returned_event);
```

- When blocking is false, call returns immediately!
Event-Based Synchronisation

- **Event objects** can be used as synchronisation points

  - `clWaitForEvents(num_events, event_list)`
    Waits and blocks until all commands identified by events in the given event list are complete
Event-based Synchronisation

- **Event objects** can be used as synchronisation points

```c
cl_event read_event;

err = clEnqueueReadBuffer(queue, buffer,
                           CL_FALSE /* blocking_read */,
                           0, 0, ptr, 0, 0,
                           &read_event);

err = clWaitOnEvents(1, &read_event);
```

- Above example is equivalent to blocking call
Event-Based Synchronisation

- All `clEnqueue()` methods also accept event wait lists

```c
cl_uint num_events_in_waitlist = 2;
cl_event event_waitlist[2] = { event_one, event_two };

err = clEnqueueReadBuffer(queue, buffer, CL_FALSE /* non-blocking */, 0, 0, num_events_in_waitlist, event_waitlist, NULL);
```
Event-Based Synchronisation

- All `clEnqueue()` methods also accept event wait lists

```c
cl_uint num_events_in_waitlist = 2;
cl_event event_waitlist[2] = { event_one, event_two };

err = clEnqueueReadBuffer(queue, buffer,
               CL_FALSE /* non-blocking */,
               0, 0,
               num_events_in_waitlist,
               event_waitlist, NULL);

- Read won’t occur until events in wait list are complete!
```
Event-Based Synchronisation

• All clEnqueue() methods also accept event wait lists

```c
cl_uint num_events_in_waitlist = 2;
cl_event event_waitlist[2];

err = clEnqueueReadBuffer(queue, buffer0, CL_FALSE /* non-blocking */, 0, 0, 0, NULL, &event_waitlist[0]);

err = clEnqueueReadBuffer(queue, buffer1, CL_FALSE /* non-blocking */, 0, 0, 0, NULL, &event_waitlist[1]);

/* last read buffer waits on previous two read buffer events */
err = clEnqueueReadBuffer(queue, buffer2, CL_FALSE /* non-blocking */, 0, 0, num_events_in_waitlist, event_waitlist, NULL);
```
Event-Based Synchronisation

- All `clEnqueue()` methods also accept **event wait lists**

```c
cl_uint num_events_in_waitlist = 2;
cl_event event_waitlist[2];

err = clEnqueueReadBuffer(queue, buffer0, CL_FALSE /* non-blocking */, 0, 0, 0, NULL, &event_waitlist[0]);

e = clEnqueueReadBuffer(queue, buffer1, CL_FALSE /* non-blocking */, 0, 0, 0, NULL, &event_waitlist[1]);

/* last read buffer waits on previous two read buffer events */
e = clEnqueueReadBuffer(queue, buffer2, CL_FALSE /* non-blocking */, 0, 0, num_events_in_waitlist, event_waitlist, NULL);
```

- Last read buffer waits on previous other two to complete!
Event-Based Synchronisation

- **Events** can also provide profiling information

```c
cl_event read_event;
cl_ulong start, end;

queue = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLED, ...)
err = clEnqueueReadBuffer(queue, buffer, 0, 0, 0, 0, NULL, &read_event);
err = clWaitOnEvents(1, read_event);

err = clGetEventProfilingInfo(read_event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);
err = clGetEventProfilingInfo(read_event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);

float milliseconds = (end - start) * 1.0e-6f;
```
Event-Based Synchronisation

- **Event**-Based Barrier Methods

  - `clEnqueueWaitList()`
    
    Enqueue a list of events to wait for such that all events need to complete before this particular command can be executed.

  - `clEnqueueMarker()`
    
    Enqueue a command which marks this location in the queue with a unique event object that can be used for synchronisation.
Cooperative Multi-Device Usage Model

commands

in-order cpu queue

in-order gpu queue

CPU

GPU

MEM

cl_event event0_cpu, event1_gpu;

err = clEnqueueNDRangeKernel(queue_cpu, kernel0_cpu, 2, NULL, global, local, 0, NULL, &event0_cpu);

err = clEnqueueNDRangeKernel(queue_gpu, kernel1_gpu, 2, NULL, global, local, 1, &event0_cpu, &event1_gpu);

err = clEnqueueNDRangeKernel(queue_cpu, kernel2_cpu, 2, NULL, global, local, 1, &event1_gpu, NULL);

clFlush(queue_cpu); clFlush(queue_gpu);
Cooperative Multi-Device Usage Model

```c
cl_event event0_cpu, event1_gpu;

err = clEnqueueNDRangeKernel(queue_cpu, kernel0_cpu, 2, NULL, global, local, 0, NULL, &event0_cpu);

err = clEnqueueNDRangeKernel(queue_gpu, kernel1_gpu, 2, NULL, global, local, 1, &event0_cpu, &event1_gpu);

err = clEnqueueNDRangeKernel(queue_cpu, kernel2_cpu, 2, NULL, global, local, 1, &event1_gpu, NULL);

clFlush(queue_cpu); clFlush(queue_gpu);
```
Cooperative Multi-Device Usage Model

- Event wait lists provide synchronised execution
- Execution is dependant on prior command execution status
- View of shared memory pool is consistent during execution
Cooperative Multi-Device Usage Model

- Event wait lists provide synchronised execution
- Execution is dependant on prior command execution status
- View of shared memory pool is consistent during execution
- Could submit other work to fill in the time gaps!
Event-Based Synchronisation

- Event objects identify unique commands in a queue
- Event objects can be used to determine command status
- Event objects also provide profiling information
- Event objects can be used as synchronisation points
Event-Based Synchronisation

Event Model Summary

- Event objects identify unique commands in a queue
- Event objects can be used to determine command status
- Event objects also provide profiling information
- Event objects can be used as synchronisation points
- Event objects provide command-level control!
- Event wait lists allow execution graphs!
Example Usage
• Some OpenCL kernels may be better off using buffers
• Some OpenGL commands may be better off using textures
• Use acquire and release interop methods to hint API usage
• Use copy format methods to convert memory
size_t origin[] = { 0, 0, 0 };
size_t region[] = { width, height, 1 };
cl_event convert_event;

err = clEnqueueAcquireGLObjects(queue, 1, &image, 0, 0, 0);

err = clEnqueueCopyBufferToImage(queue, buffer, image, 0,
origin, region, 0, NULL, &convert_event);

err = clEnqueueReleaseGLObjects(queue, 1, &image, 0, 0, 0);
• Overlap streaming data operations with compute
• Use device to do all computation with **kernels**
• Use host to do all streaming operations w/**native** functions
• Use **event wait lists** for execution order
Streaming Data

- On the device:
  - **filter kernel** reads from `cl_mem A` writes to `cl_mem B`
Streaming Data

• On the device:
  filter kernel reads from cl_mem A writes to cl_mem B

• Meanwhile, on the host:
  stream_write method outputs previous cl_mem C
Streaming Data

• On the device:
  filter kernel reads from cl_mem A writes to cl_mem B

• Meanwhile, on the host:
  stream_write method outputs previous cl_mem C
  stream_read method prefetches and fills cl_mem C
Streaming Data

• Continue and alternative memory locations....

  - filter kernel:  ( A – B )
  - stream_write:   ( C )
  - stream_read:    ( C )
Streaming Data

- Continue and alternative memory locations....

  filter kernel: \((A - B)(C - A)\)
  stream_write: \((C)(B)\)
  stream_read: \((C)(B)\)
Streaming Data

- Continue and alternative memory locations....

  filter kernel: \((A - B)(C - A)(B - C)\)
  stream_write: \((C)(B)(A)\)
  stream_read: \((C)(B)(A)\)
Streaming Data

- Some caveats....
  - Size of data stream must be large to offset transfer costs
  - Complexity of filter must be large to offset launch costs
  - Both `stream_write` and `stream_read` methods must be timely
Streaming Data

- If data size is too large....
  - `stream_read` can’t fill full data size in memory
  - `stream_write` can’t output full data size
Streaming Data

- Don’t want to delay device *kernel*
  - Not willing to sacrifice efficiency
  - Might have strict time constraints for device results
Streaming Data

- Use event profiling to identify time window for stream operations

Diagram showing stream_read and stream_write operations with filters.
• **Allocate large fixed size memory buffers**
  - Use best estimate based on kernel execution time
  - Pass in data size as kernel argument
  - Adjust data size in stream operation
• Dynamically adjust streaming data size
  • Estimate reasonable time window for first stream operation
  • Use measured time window to adjust data size in stream operation
OpenCL Event Model Usage

- Use the execution model to your advantage
- Use synchronisation to your benefit
- Use events for fine grained control
- Plenty of flexibility already (even with immutable queues)

• Summary
• Know Your Platform

  – Understand the cost of API calls
  – Profile the overhead of your platform and commands
  – Even compliant vendor implementations differ
  – Give feedback to platform vendors
  – Work with Khronos to improve OpenCL
Acknowledgements

• Course Organisers
  • AMD / NVIDIA / Khronos / Apple

• University of Western Australia
  • CMCA / WASP / iVEC

Derek Gerstmann
University of Western Australia
http://local.wasp.uwa.edu.au/~derek