EVENTS, PROFILING, AND DEBUGGING

OPENCL 2.0 UNIVERSITY TOOLKIT

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

Events

AMD

- Events are used to synchronize between individual commands
  - i.e., create a dependency graph of commands
- Explicit synchronization is required for
  - Out-of-order command queues
  - Multiple command queues
- Events are also used for storing timing information returned by the device

Events

AMD

- In addition to specifying dependencies, events are used for basic profiling of commands
- Profiling using events has to be enabled explicitly when creating a command queue
  - CL_QUEUE_PROFILING_ENABLE flag must be set
  - Requiring the runtime to generate timestamps for events may slow down execution
- A handle to store event information can be passed for all **clEnqueue*** commands
  - When commands such as **clEnqueueNDRangeKernel** and **clEnqueueReadBuffer** are invoked timing information is recorded in the provided event

Uses of Events

AMD

- Using events we can:
  - Time execution of **clEnqueue*** calls like kernel execution or explicit data transfers
  - Use the events from schedule asynchronous data transfers between host and device
  - Profile an application to understand an execution flow
  - Observe overhead and time consumed by a kernel in the command queue versus actually executing
- Event timestamps are consistent for both CPUs and GPUs

Profiling with Events

AMD

```
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)
```

- **clGetEventProfilingInfo** allows us to query cl_event to get desired counter values
- Timing information returned as cl_ulong data types
  - Returns timestamp on a nanosecond granularity

Profiling with Events

AMD

- Table shows event types described using cl_profiling_info enumerated type

Event Type	Description
CL_PROFILING_COMMAND_QUEUED	Command is enqueued in a command queue by the host.
CL_PROFILING_COMMAND_SUBMIT	Command is submitted by the host to the device associated with the command queue.
CL_PROFILING_COMMAND_START	Command starts execution on device.
CL_PROFILING_COMMAND_END	Command has finished execution on device.
CL_PROFILING_COMMAND_COMPLETE	Command and all of its child commands have finished execution on the device

Profiling with Events

AMD

- OpenCL events can easily be used for timing durations of kernels.

```
clGetEventProfilingInfo( event_time, CL_PROFILING_COMMAND_START,
sizeof(cl_ulong), &starttime, NULL);
```

- This method is reliable for performance optimizations since it uses counters from the device

```
clGetEventProfilingInfo( event_time, CL_PROFILING_COMMAND_END,
sizeof(cl_ulong), &starttime, NULL);
```

- By taking differences of the start and end timestamps we are discounting overheads like time spent in the command queue

```
unsigned long elapsed =  (unsigned long)(endtime - starttime);
```

Profiling with Events

AMD

- Before getting timing information, we must make sure that the events we are interested in have completed
- There are different ways of waiting for events:
  - clWaitForEvents(numEvents, eventList)
  - clFinish(commandQueue)
- Timer resolution can be obtained from the flag CL_DEVICE_PROFILING_TIMER_RESOLUTION when calling **clGetDeviceInfo**

Capturing Event Information

AMD

```
cl_int      clGetEventInfo (cl_event event,
                            cl_event_info param_name,
                            size_t param_value_size,
                            void *param_value,
                            size_t *param_value_size_ret)
```

- **clGetEventInfo** can be used to return information about the event object
- It can return details about the command queue, context, type of command associated with events, execution status
- This command can be used by along with timing provided by **clGetEventProfilingInfo** as part of a high level profiling framework to keep track of commands

User Events

AMD

- OpenCL 1.1 and above defines a user event object. Unlike **clEnqueue*** commands, user events can be set by the user

```
cl_event        clCreateUserEvent (cl_context context, cl_int *errcode_ret)
```

- When we create a user event, status is set to CL_SUBMITTED
- **clSetUserEventStatus** is used to set the execution status of a user event object.
- A user event can only be set to CL_COMPLETE once

```
cl_int          clSetUserEventStatus (cl_event event, cl_int execution_status)
```

User Events

AMD

- A simple example of user events being triggered and used in a command queue

```
// Create user event which will start the write of buf1
user_event = clCreateUserEvent(ctx, NULL);
clEnqueueWriteBuffer(cq, buf1, CL_FALSE, ..., 1, &user_event , NULL);
// The write of buf1 is now enqueued and waiting on user_event

X = foo(); // Lots of complicated host processing code

clSetUserEventStatus(user_event, CL_COMPLETE);

// The clEnqueueWriteBuffer to buf1 can now proceed
```

Wait Lists

AMD

- Wait lists are arrays of cl_event type
- All **clEnqueue*** methods also accept event wait lists
- OpenCL defines waitlists to provide precedence rules

cl_int	**clWaitForEvents** (cl_uint *num_events*, const cl_event **event_list*)

Wait Lists

AMD

```
cl_int  clEnqueueBarrierWithWaitList (cl_command_queue command_queue,
                                      cl_uint num_events_in_wait_list,
                                      const cl_event *event_wait_list,
                                      cl_event *event)
```

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

```
cl_int  clEnqueueMarkerWithWaitList (cl_command_queue command_queue,
                                     cl_uint num_events_in_wait_list,
                                     const cl_event *event_wait_list,
                                     cl_event *event)
```

- Enqueue a command to mark this location in the queue with a unique event object that can be used for synchronization

Event Callbacks

AMD

```
cl_int        clSetEventCallback (cl_event event,
                                  cl_int command_exec_callback_type,
                                  void (CL_CALLBACK *pfn_event_notify)(cl_event event,
                                                                       cl_int event_command_exec_status,
                                                                       void *user_data),
                                  void *user_data)
```

- OpenCL 1.1 or above allows registration of a user callback function for a specific command execution status
  - Event callbacks can be used to enqueue new commands based on event state changes in a non-blocking manner
  - Using blocking versions of **clEnqueue*** OpenCL functions in callback leads to undefined behavior
- The callback takes an cl_event, status and a pointer to user data as its parameters

Command Queue Synchronization

AMD

- Command queue synchronization methods work on a per-queue basis
- Flush:

```
cl_int  clFlush (cl_command_queue command_queue)
```

  - Sends all commands in the queue to the compute device
  - No guarantee that they will be complete when **clFlush** returns
- Finish:

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

  - Blocks host by waiting for all commands in the command queue to complete
- Barrier:

```
cl_int  clEnqueueBarrierWithWaitList (cl_command_queue command_queue,
                                      cl_uint num_events_in_wait_list,
                                      const cl_event *event_wait_list,
                                      cl_event *event)
```

  - Enqueues a synchronization point: ensures all prior commands in a queue have completed before any further commands execute

Debugging using **printf**

AMD

- Starting in OpenCL 1.2, OpenCL C supports printing during execution using **printf**
- **printf** closely matches the definition found in the C99 standard
- **printf** can be used to print information about threads or help track down bugs
- **printf** works by buffering output until the end of execution and transferring the output back to the host
  - It is important that a kernel completes in order to retrieve printed information
  - Commenting out code following **printf** is a good technique if the kernel is crashing

Debugging using **printf**

AMD

- The following example prints information about threads trying to perform an improper memory access

```
int myIdxX = ... // column index for addressing a matrix
int myIdxY = ... // row index for addressing a matrix
if(myIdxX < 0 || myIdxX >= cols ||
   myIdxY < 0 || myIdx >= rows)
 {
   printf("Work item %d,%d: bad index (%d, %d)\n",
              get_global_id(1), get_global_id(0),
              myIdxX, myIdxY));
 }
```

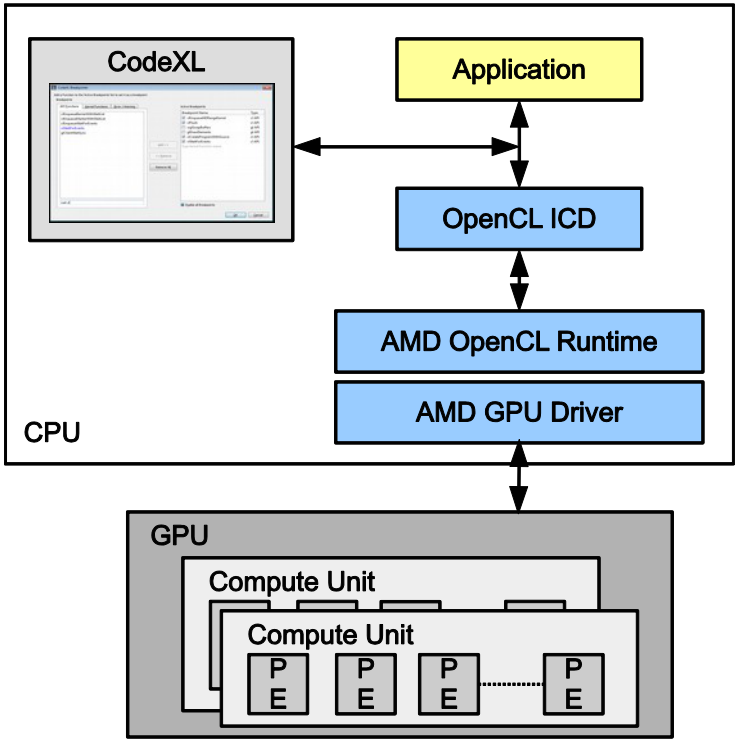
CodeXL

AMD

- Integrated profiler, kernel analyzer, and debugger tool developed by AMD
- Profile mode
  - Gathers performance data from the OpenCL runtime and AMD GPUs during execution
- Analysis mode
  - Statically compiles, analyzes, and disassembles OpenCL kernels for AMD GPUs
- Debug mode
  - Debugs an application by stepping through OpenCL API calls and kernel source code
  - Views function parameters and reduces memory consumption

Debugging Using CodeXL

- CodeXL intercepts the OpenCL API calls between the application and the OpenCL ICD
- CodeXL can debug at the API-level
  - Record the OpenCL API call history
  - Program and kernel information
  - Image and buffer data
  - Memory checking
  - API usage statistics
  - Kernel function breakpoints



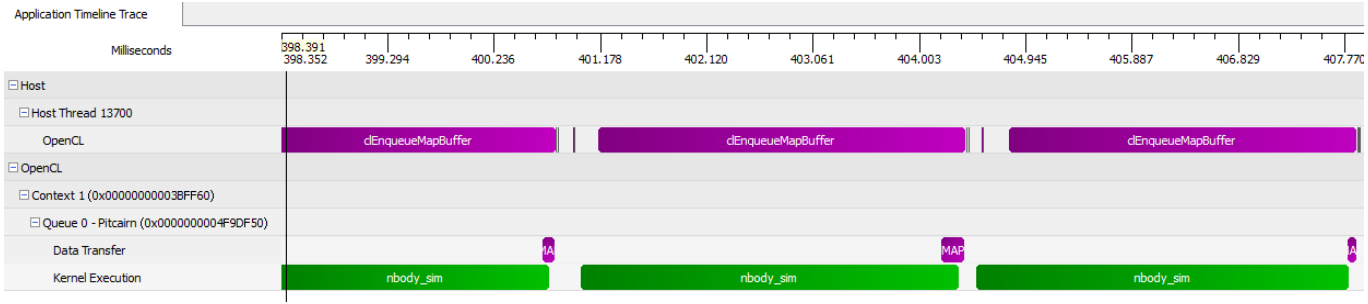
Profiling Using CodeXL

AMD

- Profiling modes
  - GPU application timeline traces
  - GPU performance counters during kernel execution
  - Collecting CPU performance information

Application Timeline View

- Provides a visual representation of the execution of the application



Host API Trace View

AMD

- Lists all the OpenCL API calls made by each host thread in the application

Host Thread 12328 | Summary

Call Index	Interface	Parameters	Result	Device Block	Kernel Occupancy	CPU Time	Device Time
102	clSetKernelArg	0x000000000458F620;6;8;[0x468AD00]	CL_SUCCESS			0.0003	
103	clEnqueueNDRangeKernel	0x0000000000514170;0x000000000458F620;1;NULL;[32768];[128];0;NULL;NULL	CL_SUCCESS	nbody_sim	50%	0.1530	59.6324
104	clFlush	0x0000000000514170	CL_SUCCESS			0.0031	
105	clEnqueueMapBuffer	0x0000000000514170;0x000000000468AB90;CL_TRUE;CL_MAP_READ;0;524288;0;NULL;NULL...	0x0000000007A77...	512.0 KB MAP_BU...		64.2462	1.2806
106	clSetKernelArg	0x000000000458F620;0;8;[0x468AB90]	CL_SUCCESS			0.0017	
107	clSetKernelArg	0x000000000458F620;1;8;[0x468AD00]	CL_SUCCESS			0.0003	
108	clSetKernelArg	0x000000000458F620;5;8;[0x468A8B0]	CL_SUCCESS				
109	clSetKernelArg	0x000000000458F620;6;8;[0x468AA20]	CL_SUCCESS			0.0003	
110	clEnqueueNDRangeKernel	0x0000000000514170;0x000000000458F620;1;NULL;[32768];[128];0;NULL;NULL	CL_SUCCESS	nbody_sim	50%	0.0626	54.8519
111	clFlush	0x0000000000514170	CL_SUCCESS			0.0048	
112	clEnqueueUnmapMemObject	0x0000000000514170;0x000000000468AB90;0x0000000007A77000;0;NULL;NULL	CL_SUCCESS			0.0205	
113	clFlush	0x0000000000514170	CL_SUCCESS			0.0068	
114	clEnqueueMapBuffer	0x0000000000514170;0x000000000468A8B0;CL_TRUE;CL_MAP_READ;0;524288;0;NULL;NULL...	0x0000000078B2...	512.0 KB MAP_BU...		2.9424	1.6532
115	clSetKernelArg	0x000000000458F620;0;8;[0x468A8B0]	CL_SUCCESS			0.0017	
116	clSetKernelArg	0x000000000458F620;1;8;[0x468AA20]	CL_SUCCESS			0.0003	
117	clSetKernelArg	0x000000000458F620;5;8;[0x468AB90]	CL_SUCCESS				
118	clSetKernelArg	0x000000000458F620;6;8;[0x468AD00]	CL_SUCCESS				
119	clEnqueueNDRangeKernel	0x0000000000514170;0x000000000458F620;1;NULL;[32768];[128];0;NULL;NULL	CL_SUCCESS	nbody_sim	50%	0.0623	46.4723
120	clFlush	0x0000000000514170	CL_SUCCESS			0.0034	

Collecting GPU Kernel Performance Counters

AMD

- The GPU kernel performance counters can be used to find possible bottlenecks in the kernel execution

	Method	tion	ThreadID	Time	VGPRs	SGPRs	FCStacks	ernelOccupanc	Wavefronts	VALUInsts	SALUInsts	VFetchInsts	SFetchInsts	VWriteInsts	VALUUtilization (%)	VALUBusy (%)	SALUBusy (%)
1	nbody_sim__k1_...	1	12616	61.77452	46	48	NA	50	512	626723	53279	2	32781	2	100	59.79	6.55

Summary

AMD

- Debugging
  - **printf** can be a light-weight debugging method
  - CodeXL debugging mode is a comprehensive debugger
- Profiling: OpenCL events allow us to use the execution model and synchronization to benefit application performance
  - Command queue synchronization constructs for coarse grained control
  - Use events for fine grained control over an application
  - OpenCL 1.1 or above allows more complicated event handling and adds callbacks and also provides for events that can be triggered by the user
  - CodeXL profiling mode can generate timing information without changing source code