CSCI-GA.3033-012
Graphics Processing Units (GPUs): Architecture and Programming

Lecture 11: OpenCL

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Open Computing Language

OpenCL Working Group

- Diverse industry participation
  - Processor vendors, system OEMs, middleware vendors, application developers
- Many industry-leading experts involved in OpenCL’s design
  - A healthy diversity of industry perspectives
- Apple initially proposed and is very active in the working group
  - Serving as specification editor
- Here are some of the other companies in the OpenCL working group
Processor Parallelism

- CPUs
  - Multiple cores driving performance increases
  - Multi-processor programming – e.g. OpenMP

- GPUs
  - Increasingly general purpose data-parallel computing
  - Improving numerical precision
  - Graphics APIs and Shading Languages

Emerging Intersection

OpenCL Heterogenous Computing
Design Goals

- Use all computation resources in the system (GPUs and CPUs as peers)
- Data parallel model (SIMD) and task parallel model
  - Efficient programming
  - Extension to C
- Abstract underlying parallelism
- Drive future hardware requirements
Implementation

• Each OpenCL implementation (OpenCL library from AMD, NVIDIA, etc.) defines *platforms* which enable the host system to interact with OpenCL-capable devices.
OpenCL Platform Model

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

Each processing element maintains its own program counter.

is whatever the OpenCL library runs on processors that the library can talk to (CPUs, GPUs, other accelerators, ...)

© Copyright Khronos Group, 2011
A Platform Is:

• "The host plus a collection of devices managed by the OpenCL framework that allow an application to share resources and execute kernels on devices in the platform."

• Platforms represented by a \texttt{cl\_platform} object, initialized with \texttt{clGetPlatformID()}
Simple code for identifying platform

```c
//Platform
cl_platform_id platform;
clGetPlatformIDs (1, &platform, NULL);
```

- **Number of platform entries**: Returns number of OpenCL platforms available. If NULL, ignored.
- **List of OpenCL platforms found. (Platform IDs)**: In our case just one platform, identified by `&platform`
A Bit of Vocabulary

• **Kernel**: Smallest unit of execution, like a C function
• **Host program**: A collection of kernels
• **Work group**: a collection of work items
  – Has a unique work-group ID
• **Work item**: an instance of kernel at run time
  – Has a unique ID within the work-group
OpenCL Memory Model

- Relaxed consistency model
- Implementations map this hierarchy to available memories
OpenCL Memory Model

• Memory management is explicit
  – Must move data from host memory to device global memory, from global memory to local memory, and back

• Work-groups are assigned to execute on compute-units
  – No guaranteed coherency between different work-groups
NDRange

- N-Dimensional Range
- $N = 1D, 2D, or 3D$
- An index space in which kernels are executed
- A work-item is a single kernel instance at a point in the index space
- Does this remind you of something? (GRID?)
Kernel Execution

- Total number of work-items = $G_x \times G_y$
- Size of each work-group = $S_x \times S_y$
- Global ID can be computed from work-group ID and local ID
Programming Model

• Data Parallel
  • Work-groups can be defined explicitly (like CUDA) or implicitly (specify the number of work-items and OpenCL creates the work-groups)

• Task Parallel
  • Kernel is executed independent of an index space
  • Can be written in OpenCL C or native compiled from C/C++
Once a platform is selected, we can then query for the devices that it knows how to interact with

\[
\text{clGetDeviceIDs}^4 (\text{cl}\_\text{platform}\_\text{id } platform, \\
\text{cl}\_\text{device}\_\text{type } device\_\text{type}, \\
\text{cl}\_\text{uint } num\_\text{entries}, \\
\text{cl}\_\text{device}\_\text{id } *devices, \\
\text{cl}\_\text{uint } *num\_\text{devices})
\]

• We can specify which types of devices we are interested in (e.g. all devices, CPUs only, GPUs only)
A Context

- A context refers to 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
Command Queues

- A command queue is the mechanism for the host to request that an action be performed by the device
  - Perform a memory transfer, begin executing, etc.
- A separate command queue is required for each device
- Commands within the queue can be synchronous or asynchronous
- Commands can execute in-order or out-of-order
Setup

1. Get the device(s)
2. Create a context
3. Create command queue(s)

```c
cl_uint num_devices Returned;
cl_device_id devices[2];
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1,
                      &devices[0], num_devices Returned);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1,
                      &devices[1], &num_devices Returned);

cl_context context;
context = clCreateContext(0, 2, devices, NULL, NULL, &err);

cl_command_queue queue_gpu, queue_cpu;
queue_gpu = clCreateCommandQueue(context, devices[0], 0, &err);
queue_cpu = clCreateCommandQueue(context, devices[1], 0, &err);
```
Memory Objects

• Memory objects are OpenCL data that can be moved on and off devices
  – Objects are classified as either buffers or images

• 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 read_image() and write_image()
  – Can either be read or written in a kernel, but not both
Allocating Memory on Device

Use clCreateBuffer:

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

- **OpenCL context**
- **Bit field to specify type of allocation/usage** (CL_MEM_READ_WRITE, …)
- **Ptr to buffer data** (May be previously allocated.)
- **Returns error code if an error**
- **Returns memory object**

Returns error code if an error
Example: Allocating Two Vectors on Device

// source data on host, two vectors

int *A, *B;
A = new int[N];
B = new int[N];
for(int i = 0; i < N; i++) {
    A[i] = rand()%1000;
    B[i] = rand()%1000;
}
...

// Allocate GPU memory for source vectors

cl_mem GPUVector1 = clCreateBuffer(GPUContext,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,sizeof(int)*N, A, NULL);

cl_mem GPUVector2 = clCreateBuffer(GPUContext,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,sizeof(int)*N, B, NULL);
Example: Allocating A Vector on Device for Results

// Allocate GPU memory for output vector

cl_mem GPUOutputVector =
clCreateBuffer(GPUContext,CL_MEM_WRITE_ONLY,sizeof(int)*N,
NULL,NULL);
Transferring Data

- OpenCL provides commands to transfer data to and from devices
  - `clEnqueue{Read|Write}{Buffer|Image}`
  - Copying from the host to a device is considered *writing*
  - Copying from a device to the host is *reading*
Transferring Data

cl_int  clEnqueueWriteBuffer (cl_command_queue command_queue,
                              cl_mem buffer,
                              cl_bool blocking_write,
                              size_t offset,
                              size_t cb,
                              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 and writes
data to the device associated with the command queue
  • The command will write data from a host pointer (ptr) to the device
• The blocking_write parameter specifies whether or not the
  command should return before the data transfer is complete
• Events can specify which commands should be completed before
  this one runs
Compilation Model

• More complicated than CUDA
• uses Dynamic/Runtime compilation model
  1. The code is compiled to an Intermediate Representation (IR)
     • Usually an assembler or a virtual machine
     • Known as offline compilation
  2. The IR is compiled to a machine code for execution.
     • This step is much shorter.
     • It is known as online compilation.
• Starting a kernel can be expensive, so try to make individual kernels do a large amount of work.
Typical OpenCL Program Flow

- Select the desired devices (ex: all GPUs)
- Create a context
- Create command queues (per device)
- Compile programs
- Create kernels
- Allocate memory on devices
- Transfer data to devices
- Execute
- Transfer results back
- Free memory on devices
OpenCL vs CUDA
Memory Model Comparison

OpenCL

CUDA
# CUDA vs OpenCL

## CUDA
- Global mem
- Shared (per-block) mem
- Local mem
- Kernel
- Block
- Thread
- Easier compilation
- A bit restricted

## OpenCL
- Global mem
- Local memory
- Private memory
- Program
- Work-group
- Work-item
- Complicated compilation
- More versatile (GPU, CPU, Cell, DSP, ..)
Figure from: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming, Pend Du et. al. Elsevier Parallel Computing Journal, Volume 38, Issue 8, August 2012, Pages 391–407
Software Stack for CUDA and OpenCL

Figure from: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming, Pend Du et. al. Elsevier Parallel Computing Journal, Volume 38, Issue 8, August 2012, Pages 391–407
OpenCL on NVIDIA Hardware

- Vector types in OpenCL
  - Use it for convenience not performance
  - NVIDIA is a scalar architecture
  - Better have more work items than large vector per work item

- Many concurrent work-items is a good way to overlap computation and memory access for high-intensity arithmetic programs
OpenCL on NVIDIA Hardware

• Take advantage of __local memory
• Work-items can cooperate via this __local memory using barrier() which has low overhead
• Use __local memory to manage locality and reduce global memory access
For Fun!

http://www.simplyhired.com/a/jobtrends/trend/q-opencl%C2%A0cuda%C2%A0openmp
Conclusions

• CUDA has been around for longer -> more libraries and OpenCL is playing catch-up
• OpenCL is more versatile (CUDA on Radeon?)