Objective

- To Understand the OpenCL programming model
  - basic concepts and data types
  - OpenCL application programming interface - basic
  - Simple examples to illustrate basic concepts and functionalities
OpenCL Programs

- An OpenCL “program” contains one or more “kernels” and any supporting routines that run on a target device.
- An OpenCL kernel is the basic unit of parallel code that can be executed on a target device.

OpenCL Program

- Misc support functions
- Kernel A
- Kernel B
- Kernel C
OpenCL Execution Model

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

```
Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);
```

```
Serial Code (host)
```

```
Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);
```

```
Serial Code (host)
```
OpenCL Kernels

• Code that actually executes on target devices

• Kernel body is instantiated once for each work item
  – An OpenCL work item is equivalent to a CUDA thread

• Each OpenCL work item gets a unique index

```c
__kernel void vadd(__global const float *a,
                   __global const float *b,
                   __global float *result)
{
    int id = get_global_id(0);
    result[id] = a[id] + b[id];
}
```
Array of Parallel Work Items

- An OpenCL kernel is executed by an array of work items
  - All work items run the same code (SPMD)
  - Each work item has an index that it uses to compute memory addresses and make control decisions

```c
int id = get_global_id(0);
result[id] = a[id] + b [id];
```

threads

0 1 2 3 4 5 6 7
Work Groups: Scalable Cooperation

- Divide monolithic work item array into work groups
  - Work items within a work group cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Work items in different work groups cannot cooperate

```
int id = get_global_id(0);
result[id] = a[id] + b[id];
```
OpenCL Data Parallel Model Summary

• Parallel work is submitted to devices by launching kernels

• Kernels run over global dimension index ranges (NDRange), broken up into “work groups”, and “work items”

• Work items executing within the same work group can synchronize with each other with barriers or memory fences

• Work items in different work groups can’t sync with each other, except by launching a new kernel
OpenCL Host Code

• Prepare and trigger device code execution
  – Create and manage device context(s) and associate work queue(s), etc…
  – Memory allocations, memory copies, etc
  – Kernel launch

• OpenCL programs are normally compiled entirely at runtime, which must be managed by host code
OpenCL Hardware Abstraction

- OpenCL exposes CPUs, GPUs, and other Accelerators as “devices”
- Each “device” contains one or more “compute units”, i.e. cores, SMs, etc...
- Each “compute unit” contains one or more SIMD “processing elements”
An Example of Physical Reality Behind OpenCL Abstraction

---

**GPU w/ local DRAM (device)**

**CPU (host)**
OpenCL Context

- Contains one or more devices
- OpenCL memory objects are associated with a context, not a specific device
- `clCreateBuffer()` is the main data object allocation function
  - error if an allocation is too large for any device in the context
- Each device needs its own work queue(s)
- Memory transfers are associated with a command queue (thus a specific device)
OpenCL Context Setup Code (simple)

cl_int clerr = CL_SUCCESS;
cl_context clctx = clCreateContextFromType(0, CL_DEVICE_TYPE_ALL, NULL, NULL, &clerr);

size_t parmsz;
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, 0, NULL, &parmsz);

cl_device_id* cldevs = (cl_device_id *) malloc(parmsz);
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, parmsz, cldevs, NULL);

cl_command_queue clcmdq = clCreateCommandQueue(clctx, cldevs[0], 0, &clerr);
OpenCL Memory Model Overview

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- We will focus on global memory for now
OpenCL Device Memory Allocation

\textbf{clCreateBuffer();}
- Allocates object in the device \textbf{Global Memory}
- Returns a pointer to the object
- Requires five parameters
  - OpenCL context pointer
  - Flags for access type by device
  - Size of allocated object
  - Host memory pointer, if used in copy-from-host mode
  - Error code

\textbf{clReleaseMemObject()}
- Frees object
  - Pointer to freed object
OpenCL Device Memory Allocation (cont.)

- Code example:
  - Allocate a 1024 single precision float array
  - Attach the allocated storage to d_a
  - “d_” is often used to indicate a device data structure

VECTOR_SIZE = 1024;
cl_mem d_a;
int size = VECTOR_SIZE* sizeof(float);

d_a = clCreateBuffe(clctx, CL_MEM_READ_ONLY, size, NULL, NULL);
cliReleaseMemObject(d_a);
OpenCL Device Command Execution

Application → Command → Cmd Queue → Cmd Queue → OpenCL Device

OpenCL Context

OpenCL Device
OpenCL Host-to-Device Data Transfer

- `clEnqueueWriteBuffer();`
  - memory data transfer to device
  - Requires nine parameters
    - OpenCL command queue pointer
    - Destination OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size (in bytes) of written data
    - Host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command

- Asynchronous transfer later
OpenCL Device-to-Host Data Transfer

- `clEnqueueReadBuffer();`
  - memory data transfer to host
  - requires nine parameters
    - OpenCL command queue pointer
    - Source OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size of bytes of read data
    - Destination host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command

- Asynchronous transfer later
OpenCL Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - a is in host memory and d_a is in device memory

```c
int mem_size = 64*64*sizeof(float);
clEnqueueWriteBuffer(clcmdq, d_a, CL_FALSE, 0, mem_size, (const void *)a, 0, 0, NULL);

cEnqueueReadBuffer(clcmdq, d_result, CL_FALSE, 0, mem_size, (void *) host_result, 0, 0, NULL);
```
OpenCL Host-Device Data Transfer (cont.)

- clCreateBuffer and clEnqueueWriteBuffer can be combined into a single command using special flags.

- Eg:

  ```c
  d_A = clCreateBuffer(clctxt, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, mem_size, h_A, NULL);
  ```
  
  - Combination of 2 flags here. CL_MEM_COPY_HOST_PTR to be used only if a valid host pointer is specified.
  
  - This creates a memory buffer on the device, and copies data from h_A into d_A.

  - Includes an implicit clEnqueueWriteBuffer operation, for all devices/command queues tied to the context clctxt.
OpenCL Memories

- **__global** – large, long latency
- **__private** – on-chip device registers
- **__local** – memory accessible from multiple PEs or work items. May be SRAM or DRAM, must query…
- **__constant** – read-only constant cache
- Device memory is managed explicitly by the programmer, as with CUDA
OpenCL Kernel Execution Launch

Application

Kernel

Cmd Queue

OpenCL Device

OpenCL Context
const char* vaddsrc =

    "__kernel void vadd(__global float *d_A, __global float *d_B, __global float *d_C, int N)
    { \n        […] etc and so forth […] \n    }"

cl_program clpgm;
clpgm = clCreateProgramWithSource(clctx, 1, &vaddsrc, NULL, &clerr);
char clcompileflags[4096];
sprintf(clcompileflags, "-cl-mad-enable");
clerr = clBuildProgram(clpgm, 0, NULL, clcompileflags, NULL, NULL);
cl_kernel clkern = clCreateKernel(clpgm, "vadd", &clerr);
Summary: Host code for $\text{vadd}$

```c
int main()
{
    // allocate and initialize host (CPU) memory
    float *h_A = ..., *h_B = ...
    // allocate device (GPU) memory
    cl_mem d_A, d_B, d_C;
    d_A = clCreateBuffer(clctx, CL_MEM_READ_ONLY |
                          CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_A, NULL);
    d_B = clCreateBuffer(clctx, CL_MEM_READ_ONLY |
                          CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_B, NULL);
    d_C = clCreateBuffer(clctx, CL_MEM_WRITE_ONLY, N *sizeof(float), NULL, NULL);
    clkern = clCreateKernel(clpgm, "vadd", NULL);
    clerr = clSetKernelArg(clkern, 0, sizeof(cl_mem), (void *)&d_A);
    clerr = clSetKernelArg(clkern, 1, sizeof(cl_mem), (void *)&d_B);
    clerr = clSetKernelArg(clkern, 2, sizeof(cl_mem), (void *)&d_C);
    clerr = clSetKernelArg(clkern, 3, sizeof(int), &N);
}```
Summary of Host Code (cont.)

cl_event event=NULL;
clerr= clEnqueueNDRangeKernel(clcmdq, clkern, 2, NULL, Gsz,Bsz, 0, NULL, 
&event);
clerr= clWaitForEvents(1, &event);
clEnqueueReadBuffer(clcmdq, d_C, CL_TRUE, 0, N*sizeof(float), h_C, 0, NULL, 
NULL);
clReleaseMemObject(d_A);
clReleaseMemObject(d_B);
clReleaseMemObject(d_C);
}
ANY MORE QUESTIONS?  
READ CHAPTER 14