OpenCL

Lecture 2

Exploring the spec through examples

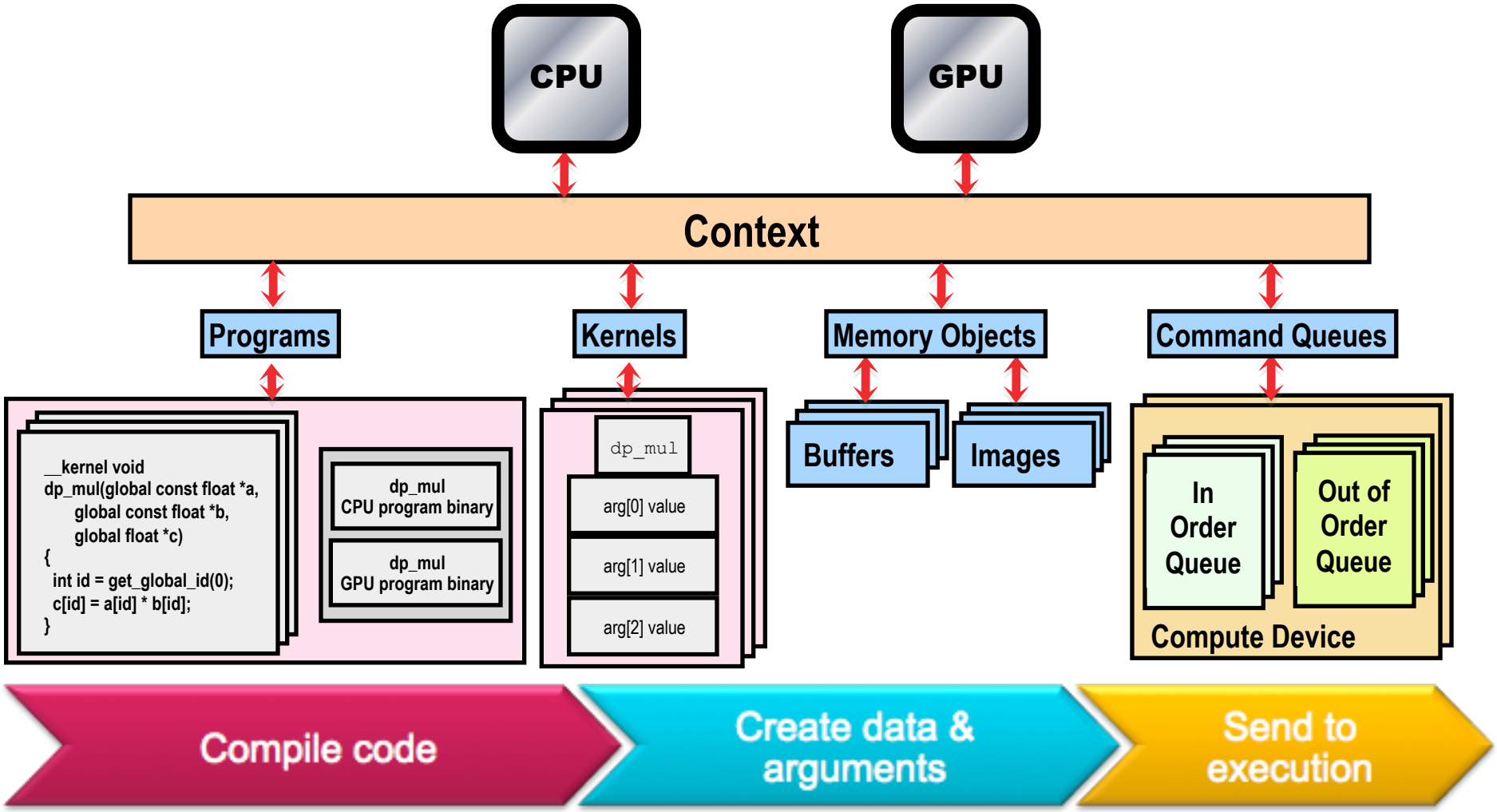
Based on material by Benedict Gaster and Lee Howes (AMD), Tim Mattson (Intel) and several others.

University of BRISTOL

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **OpenCL overview**
- **Exploring the spec through a series of examples**
  - **Vector addition:**
    - *the basic platform layer*
  - **Matrix multiplication:**
    - *writing simple kernels*
  - **Optimizing matrix multiplication:**
    - *work groups and the memory model*
  - **Radix Sort:**
    - *synchronization*
- **A survey of OpenCL 1.1**

OpenCL summary



Reminder of some OpenCL terminology

OpenCL term	Explanation
• Host	• Host CPU (e.g. x86)
• Compute device	• GPUs, CPUs
• Compute unit	• Sub unit of GPU / CPU
• Processor Element (PE)	• HW thread / core
• Global memory	• E.g. GPU memory
• Local memory	• Inside a compute unit
• Private memory	• Inside a PE

More important OpenCL terminology

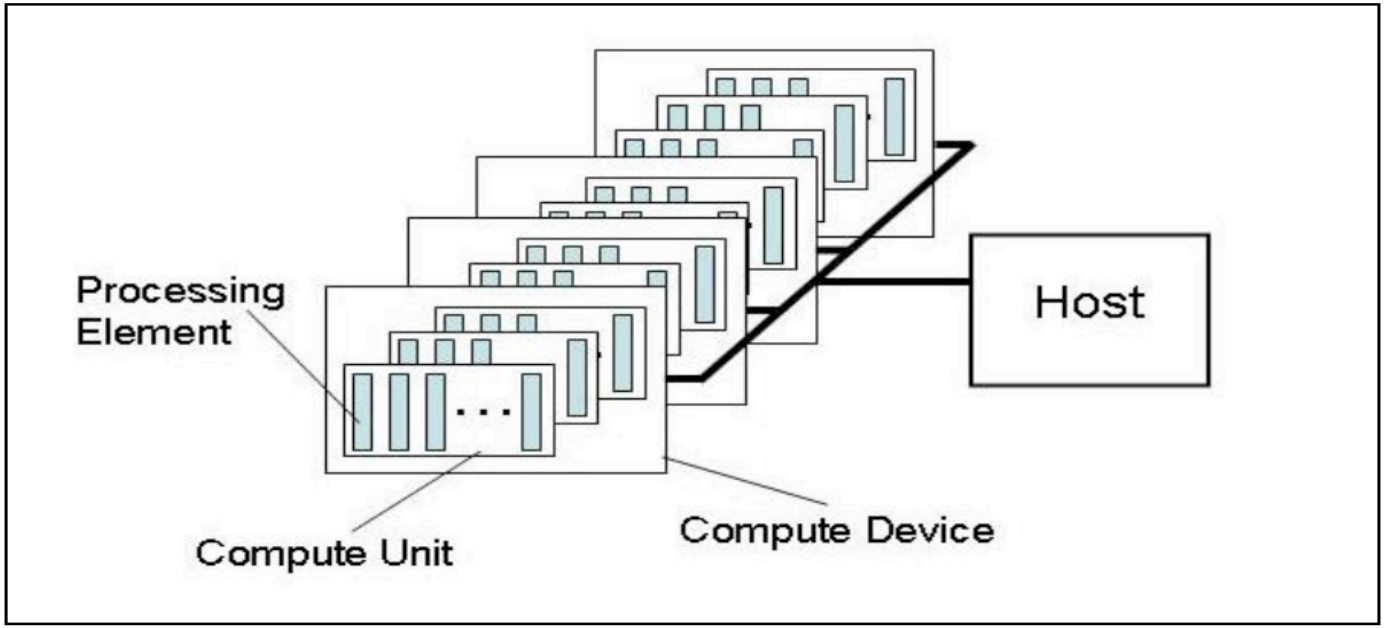
Kernel: A kernel is a function declared in a program and executed on an OpenCL device. A kernel is identified by the `__kernel` qualifier applied to any function defined in a program.

Work-item: One of a collection of parallel executions of a kernel invoked on a device by a command. A work-item is executed by one or more processing elements as part of a work-group executing on a compute unit. A work-item is distinguished from other executions within the collection by its global ID and local ID.

Work-group: A collection of related work-items that execute on a single compute unit. The work-items in the group execute the same kernel and share local memory and work-group barriers.

From OpenCL 1.1 spec.

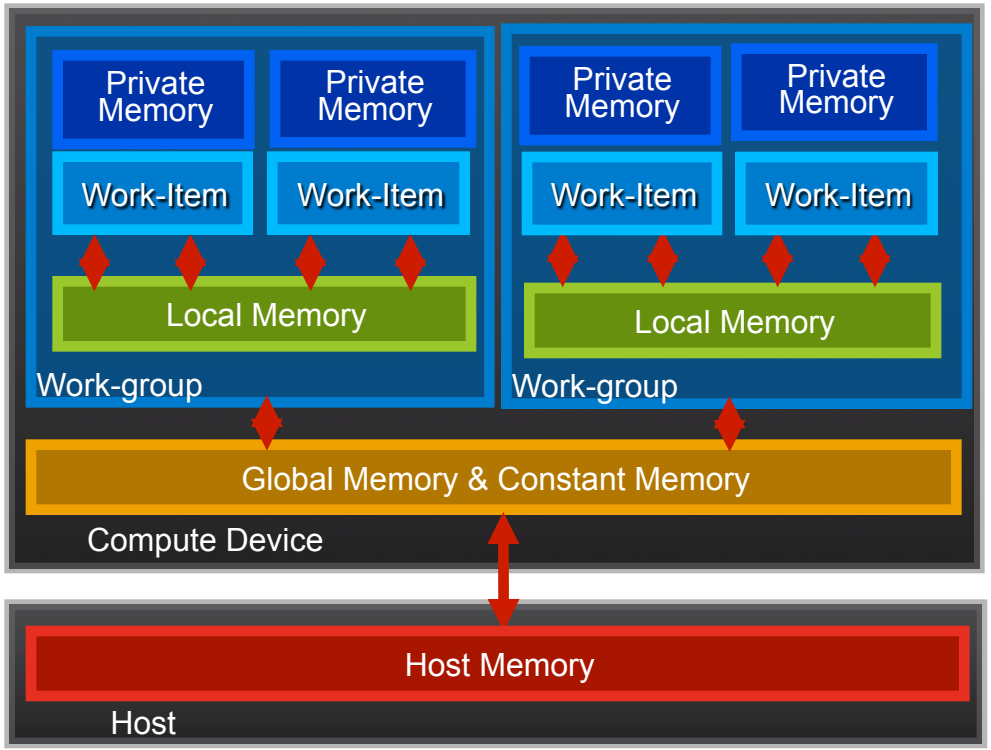
OpenCL Platform Model



- **One Host + one or more Compute Devices**
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements

OpenCL Memory Model

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



- **Memory management is explicit:**
  You must move data from host -> global -> local *and* back

Example: vector addition

- **The “hello world” program of data parallel programming is a program to add two vectors**

  C[i] = A[i] + B[i] for i=1 to N

- **For the OpenCl solution, there are two parts**
  - Kernel code
  - Host code

Vector Addition – Kernel

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global       float *c)
{
     int gid = get_global_id(0);
     c[gid] = a[gid] + b[gid];
}
```

Vec

```
// create
cl_contex                                                    NULL,
   CL_DEV                                                    ULL);

// get th
clGetCont                                                    ULL);

devices =
clGetCont                                                    mobjs[0],
   device                                                    m));
                                                             objs[1],
// create                                                    m));
cmd_queue                                                    objs[2],
   0, NUL                                                    em));

// alloca
memobjs[0
   CL_MEM

memobjs[1                                                    1, 1,
   CL_MEM                                                    L);

memobjs[2

                                                             2],
                                                             LL, NULL);
// create
program =
   &progr
```

Vector Addition - Host Program

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
```

Define platform and queues

```
// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                        NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
   devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
   0, NULL);
```

```
// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA,
   N
memo                                          ONLY |
   C                                          B,
   NULL);
memobjs[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                        sizeof(cl_float)*n, NULL,
   NULL);
```

Define Memory objects

```
// create the program
progra                                        1,
   &pr
```

Create the program

```
// build
err = clB                                     NULL,
   NULL);
```

Build the program

```
// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err                                           s[0],
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                  sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                   sizeof(cl_mem));
```

Create and setup kernel

```
// set work-item dimensions
global_w

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);
```

Execute the kernel

```
/
e                                                 ],
```

Read results back to the host

It's complicated, but most of this is "boilerplate" and not as bad as it looks.

Platform Layer: basic discovery

The '*platform layer*' allows applications to query a platform about the features it provides

- ***clGetDeviceIDs*()**
  - Find out ***what*** compute devices are in the system
  - Device types include CPUs, GPUs and Accelerators (Cell)

- ***clGetDeviceInfo*()**
  - Queries the ***capabilities*** of the discovered compute devices:
    - Number of processor elements
    - Maximum work-item and work-group size
    - Sizes of the different memory spaces
    - Maximum memory object size

The OpenCL platform on HECToR

HECToR GPGPU Testbed

The HECToR GPGPU testbed machine has been provided for researchers to test their scientific codes and problems on a modern GPGPU-accelerated system.

Accessing the GPGPU Machine

Once you have successfully applied for an account on the testbed via SAFE, you can access the frontend node via a SSH connection to:

```
<username>@gpu.hector.ac.uk
```

Note : Your GPU password is not synchronised with your HECToR password

Hardware Details

Currently the testbed machine has four compute nodes connected by Quad-band Infiniband interconnects. All of the compute nodes have a single quad-core Intel Xeon 2.4GHz CPU and 32 GB of main memory. Three of the compute nodes (gpu1, gpu2, gpu3) have 4 NVidia Fermi GPGPU cards installed and the remaining compute node (gpu4) has 1 NVidia Fermi and 1 AMD FireStream GPGPU card installed. The layout is summarised in the table below.

Compute Node	CPU	Main Memory	GPGPU Cards
gpu1	Quad-core Intel Xeon 2.4GHz	32GB	4x NVidia Fermi C2050 (3GB Memory)
gpu2	Quad-core Intel Xeon 2.4GHz	32GB	4x NVidia Fermi C2050 (3GB Memory)
gpu3	Quad-core Intel Xeon 2.4GHz	32GB	2x NVidia Fermi C2050 (3GB Memory) 2x NVidia Fermi C2070 (6GB Memory)
gpu4	Quad-core Intel Xeon 2.4GHz	32GB	1x NVidia Fermi C2050 (3GB Memory) 1x AMD FireStream 9270

Screenshot from http://www.hector.ac.uk/support/documentation/guides/gpgpu/

The OpenCL platform on HECToR

- **To set up your GPU environment:**

1. Log in to the head node, e.g. ssh username@gpu.hector.ac.uk
2. `cp -r ~crsadmin/NVIDIA_GPU_Computing_SDK .`
3. `cp -r ~crsadmin/opencl_course .`
4. `cd opencl_course/prac1`
5. `make`
6. Submit jobs to the GPUs via the queue manager using ‘qsub’, e.g.
7. `qsub jobSub1`
8. Keep track of where your jobs are in the queue with “qstat”

oclDeviceQuery example

```
[u04n033]$ oclDeviceQuery
oclDeviceQuery.exe Starting...

OpenCL SW Info:

  CL_PLATFORM_NAME:      NVIDIA CUDA
  CL_PLATFORM_VERSION:   OpenCL 1.0 CUDA 3.2.1
  OpenCL SDK Revision:   7027912
```

OpenCL Device Info:

```
 2 devices found supporting OpenCL:
--------------------------------
Device Tesla M2050
--------------------------------
 CL_DEVICE_NAME:                          Tesla M2050
 CL_DEVICE_VENDOR:                        NVIDIA Corporation
 CL_DRIVER_VERSION:                       260.24
 CL_DEVICE_VERSION:                       OpenCL 1.0 CUDA
 CL_DEVICE_TYPE:                          CL_DEVICE_TYPE_GPU
 CL_DEVICE_MAX_COMPUTE_UNITS:             14      (OpenCL: #Compute Units)
 CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS:      3       (OpenCL: 3D index space)
 CL_DEVICE_MAX_WORK_ITEM_SIZES:           1024 / 1024 / 64
 CL_DEVICE_MAX_WORK_GROUP_SIZE:           1024
 CL_DEVICE_MAX_CLOCK_FREQUENCY:           1147 MHz
 CL_DEVICE_ADDRESS_BITS:                  32
 CL_DEVICE_MAX_MEM_ALLOC_SIZE:            767 MByte
 CL_DEVICE_GLOBAL_MEM_SIZE:               3071 MByte
```

```
CL_DEVICE_ERROR_CORRECTION_SUPPORT:      no      (we'd turned this off)
CL_DEVICE_LOCAL_MEM_TYPE:                local
CL_DEVICE_LOCAL_MEM_SIZE:                48 KByte
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE:      64 KByte
CL_DEVICE_QUEUE_PROPERTIES:
     CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
CL_DEVICE_QUEUE_PROPERTIES:              CL_QUEUE_PROFILING_ENABLE
CL_DEVICE_SINGLE_FP_CONFIG:              denorms
                                         INF-quietNaNs
                                         round-to-nearest
                                         round-to-zero
                                         round-to-inf
                                         fma
```

```
CL_DEVICE_EXTENSIONS:
        cl_khr_byte_addressable_store
        cl_khr_global_int32_base_atomics
        cl_khr_global_int32_extended_atomics
        cl_khr_local_int32_base_atomics
        cl_khr_local_int32_extended_atomics
        cl_khr_fp64

  CL_DEVICE_COMPUTE_CAPABILITY_NV:        2.0
  NUMBER OF MULTIPROCESSORS:              14      (OpenCL: Compute Units)
  NUMBER OF CUDA CORES:                   448     (OpenCL: total PEs)
  CL_DEVICE_REGISTERS_PER_BLOCK_NV:       32768
  CL_DEVICE_WARP_SIZE_NV:                 32      (OpenCL: PEs per CU)
  CL_DEVICE_GPU_OVERLAP_NV:               CL_TRUE
  CL_DEVICE_KERNEL_EXEC_TIMEOUT_NV:       CL_FALSE
  CL_DEVICE_INTEGRATED_MEMORY_NV:         CL_FALSE
  CL_DEVICE_PREFERRED_VECTOR_WIDTH_<t>    CHAR 1, SHORT 1, INT 1,
                                          LONG 1, FLOAT 1, DOUBLE 1
                                          (Nvidia doesn't need vectors)
```

Speeds and Feeds

Quad-core Xeon 2.4GHz

- 38.4 GFLOPS 64-bit
- ~ 30 GBytes/s BW

Fermi C2050 GPU

- ~500 GFLOPS 64-bit
- ~150 GBytes/s BW

- Up to 4 GBytes/s full-duplex over PCI-Express to each GPU

Bandwidth between CPU and GPU

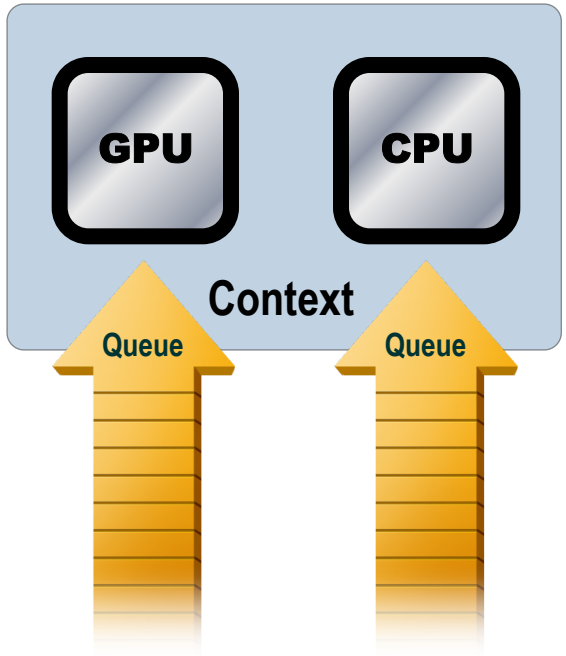
```
[u04n033]$ oclBandwidthTest

Host to Device Bandwidth, 1 Device(s), Paged memory, direct access
   Transfer Size (Bytes)    Bandwidth(MB/s)
   33,554,432               3,659.0

Device to Host Bandwidth, 1 Device(s), Paged memory, direct access
   Transfer Size (Bytes)    Bandwidth(MB/s)
   33,554,432               3,450.2

Device to Device Bandwidth, 1 Device(s)
   Transfer Size (Bytes)    Bandwidth(MB/s)
   33,554,432               91,965.9
```

Platform Layer: Contexts

- **Creating contexts**
  - Contexts are used by the OpenCL runtime to manage objects and execute kernels on one or more devices
  - Contexts are associated to one or more devices
    - Multiple contexts could be associated to the same device
  - *clCreateContext*() and *clCreateContextFromType*() return a *handle* to the created contexts

Platform layer: Command-Queues

- Command-queues store a set of operations to perform
- Command-queues are associated to a context
- Multiple command-queues can be created to handle independent commands that don’t require synchronization
- Execution of the command-queue is guaranteed to be completed at sync points



VecAdd: Context, Devices, Queues

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(
                        0,                    // platform ID
                        CL_DEVICE_TYPE_GPU,   // ask for a GPU
                        NULL,                 // error callback
                        NULL,                 // user data for callback
                        NULL);                // error code

// get the list of GPU devices associated with the context
size_t cb;
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
cl_device_id *devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cl_cmd_queue cmd_queue = clCreateCommandQueue(context,
                           devices[0], // use the first GPU device
                           0,          // default options
                           NULL);      // error code
```

Memory Objects

- **Buffers**
  - Simple contiguous chunks of memory
  - Kernels can access buffers however they like (arrays, pointers, structs)
  - Kernels can directly read and write buffers

- **Images**
  - Opaque 2D or 3D formatted data structures
  - Kernels access images only via `read_image()` and `write_image()`
  - Each image can be read or written in a kernel, but not both
    - Use multiple kernels to each read/write

Creating Memory Objects

- **Memory objects are created within an associated context**
  - *clCreateBuffer()*, *clCreateImage2D*(), and *clCreateImage3D*()
- **Memory objects can be created as read only, write only, or read-write**
- **You can control where objects are created in the platform's memory space**
  - Device memory
  - Device memory with data copied from a host pointer
  - Host memory
  - Host memory associated with a pointer
    - Memory at that pointer is guaranteed to be valid at synchronization points

VecAdd: Create Memory Objects

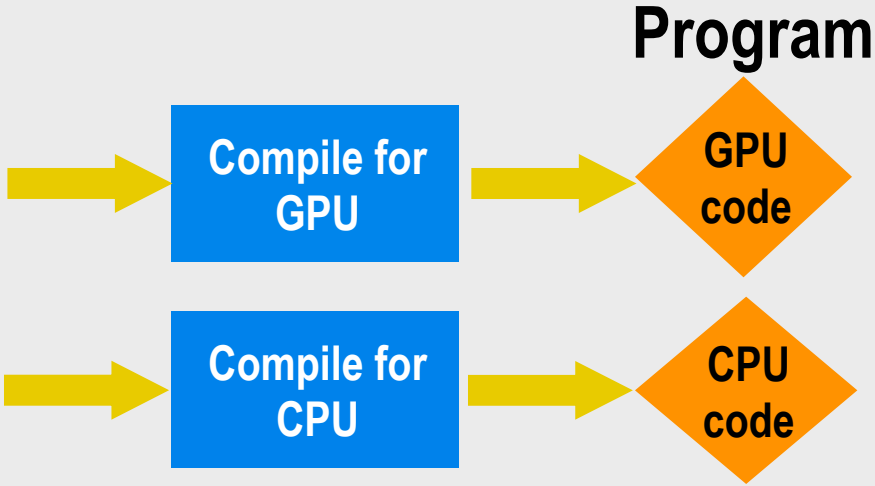
```
cl_mem memobjs[3];
// allocate input buffer memory objects
memobjs[0] = clCreateBuffer(context,
                    CL_MEM_READ_ONLY |    // bitwise flags ORd together
                    CL_MEM_COPY_HOST_PTR,
                    sizeof(cl_float)*n,   // size
                    srcA,                 // host pointer
                    NULL);                // error code
memobjs[1] = clCreateBuffer(context,
                    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                    sizeof(cl_float)*n, srcB, NULL);

// allocate output buffer memory object
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                    sizeof(cl_float)*n, NULL, NULL);
```

Build the Program object

- **The program object encapsulates:**
  - A context
  - The program source/binary
  - List of target devices and build options
- **The build process to create a program object**
  - clCreateProgramWithSource()
  - clCreateProgramWithBinary()

Kernel Code

```
__kernel void
horizontal_reflect(read_only image2d_t src,
                   write_only image2d_t dst)
{
  int x = get_global_id(0);  // x-coord
  int y = get_global_id(1);  // y-coord
  int width = get_image_width(src);
  float4 src_val = read_imagef(src, sampler,
                               (int2)(width-1-x, y));
  write_imagef(dst, (int2)(x, y), src_val);
}
```



VecAdd: Create and Build the Program

```
// create the program
cl_program program = clCreateProgramWithSource(
                context,
                1,                  // number of source strings
                &program_source, // program strings
                NULL,               // string lengths if not NULL term'td
                NULL);              // error code
// build the program
cl_int err = clBuildProgram(program,
                0,      // device number within the device list
                NULL,   // device list
                NULL,   // options
                NULL,   // notifier callback function ptr
                NULL); // user data for callback function
```

Kernel Objects

- **Kernel objects encapsulate**
  - Specific kernel functions declared in a program
  - Argument values used for kernel execution
- **Creating kernel objects**
  - ***clCreateKernel()*** - creates a kernel object for a single function in a program
- **Setting arguments**
  - ***clSetKernelArg(<kernel>, <argument index>)***
  - Each argument's data must be set for each kernel function
  - Argument values are copied and stored in the kernel object
- **Kernel objects vs. program objects**
  - Kernels are related to *program execution*
  - Programs are related to *program source*

VecAdd: Create the Kernel and Set the Arguments

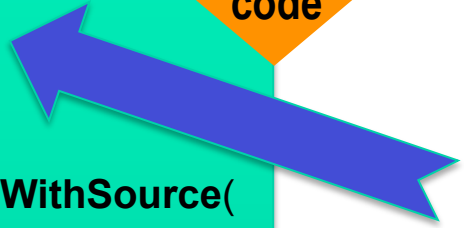
```
// create the kernel
cl_kernel kernel = clCreateKernel(program, "vec_add", NULL);

// set the "a" argument
err  = clSetKernelArg(kernel,
                      0,                          // argument index
                      (void *)&memobjs[0], // argument data
                      sizeof(cl_mem));     // argument data size

// set the "b" argument
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                       sizeof(cl_mem));

// set the "c" argument
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                       sizeof(cl_mem));
```

File structure of OpenCL programs

oclDotProduct.cpp

CPU code

```
szKernelLength = fread(
   &cSourceCL, 1,
   MAX_FILE_SIZE, clSrcFile);
…
cpProgram = clCreateProgramWithSource(
      cxGPUContext, 1,
      (const char **)&cSourceCL,
      &szKernelLength,
      &ciErrNum);

ciErrNum = clBuildProgram(
   cpProgram, 0, NULL, NULL, NULL, NULL);

ckKernel = clCreateKernel(
   cpProgram, "DotProduct", &ciErrNum);

ciErrNum = clSetKernelArg(ckKernel, 0,
   sizeof(cl_mem), (void*)&cmDevSrcA);
…
```



DotProduct.cl

GPU code

```
__kernel void DotProduct(
     __global float* a,
     __global float* b,
     __global float* c,
     int iNumElements)
{
  …
}
```

See oclLoadProgSource() in ~/NVIDIA_GPU_Computing_SDK/OpenCL/common/src

Kernel Execution

- **A command to execute a kernel must be enqueued to the command-queue**
  - Command-queues could be explicitly flushed to the device
  - Command-queues execute in-order or out-of-order
    - **In-order**: commands complete in the order queued and memory is consistent
    - **Out-of-order**: no guarantee of (1) when commands are executed or (2) if memory is consistent … unless specific synchronization is used.

Enqueue command types

- ***clEnqueueNDRangeKernel*()**
  - N Dimensional Range (N=1..3)
  - Data-parallel execution model
  - Describes the ***index space*** for kernel execution
  - Requires information on NDRange dimensions and work-group size
- ***clEnqueueTask*()**
  - Task-parallel execution model (multiple queued tasks)
  - Kernel is executed on a single work-item
- ***clEnqueueNativeKernel*()**
  - Task-parallel execution model
  - Executes a native C/C++ function not compiled using the OpenCL compiler
  - This mode does not use a kernel object so arguments must be passed in

VecAdd: Invoke Kernel

```
size_t global_work_size[1] = n; // set work-item dimensions
// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel,
                      1,          // Work dimensions
                      NULL,       // must be NULL (work offset)
                      global_work_size,
                      NULL,       // automatic local work size
                      0,          // no events to wait on
                      NULL,       // list of events to wait for
                      NULL);      // event created for this kernel
```

Synchronization

- **Synchronization**
  - Signals when commands are completed to the host or to other commands still in the queue
  - *Blocking calls*
    - Commands that do not return until complete
    - clEnqueueReadBuffer() can be called as blocking and will block until complete
  - *Event objects*
    - Tracks execution status of a command
    - Some commands can be blocked until event objects signal a completion of previous command
      - clEnqueueNDRangeKernel() can take an event object as an argument and wait until a previous command (e.g., clEnqueueWriteBuffer) is complete
  - *Queue barriers*
    - Queued commands that can block command execution

VecAdd: Read Output

```
// read output array
err = clEnqueueReadBuffer( context, memobjs[2],
                      CL_TRUE,              // blocking
                      0,                    // offset (must be 0)
                      n*sizeof(cl_float), // size in bytes
                      dst,                  // host memory pointer
                      0, NULL, NULL);       // events
```