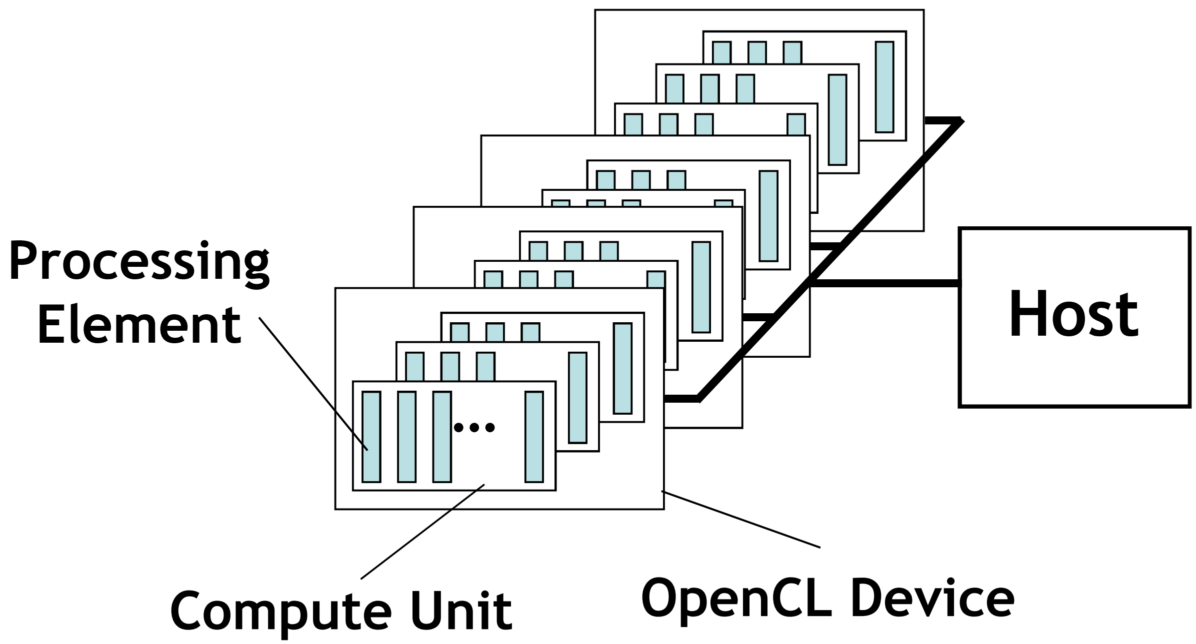
Lecture 3

IMPORTANT OPENCL CONCEPTS

OpenCL Platform Model



- One ***Host*** and one or more ***OpenCL Devices***
  - Each OpenCL Device is composed of one or more ***Compute Units***
    - Each Compute Unit is divided into one or more *Processing Elements*
- Memory divided into ***host memory*** and ***device memory***

The BIG idea behind OpenCL

- Replace loops with functions (a kernel) executing at each point in a problem domain
  - E.g., process a 1024x1024 image with one kernel invocation per pixel or 1024x1024=1,048,576 kernel executions

Traditional loops

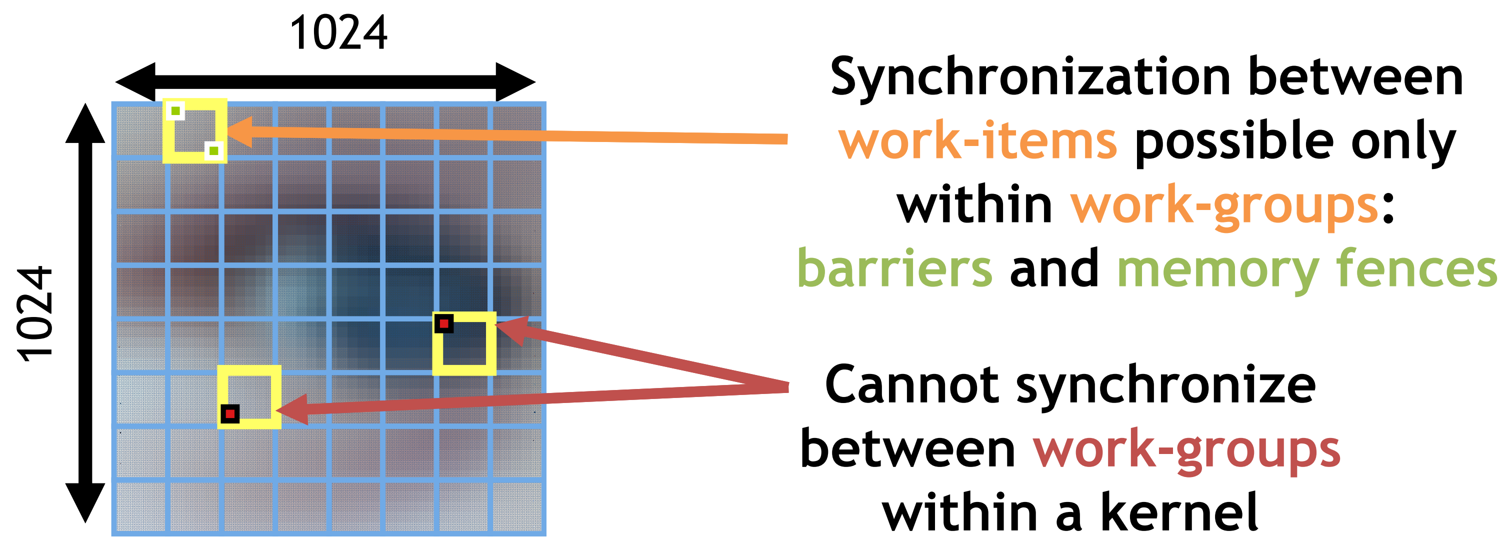
```
void
mul(const int n,
    const float *a,
    const float *b,
          float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
__kernel void
mul(__global const float *a,
    __global const float *b,
    __global       float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
// many instances of the kernel,
// called work-items, execute
// in parallel
```

An N-dimensional domain of work-items

- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 64x64 (work-group, executes together)



- Choose the dimensions that are "best" for your algorithm

OpenCL N Dimensional Range (NDRange)

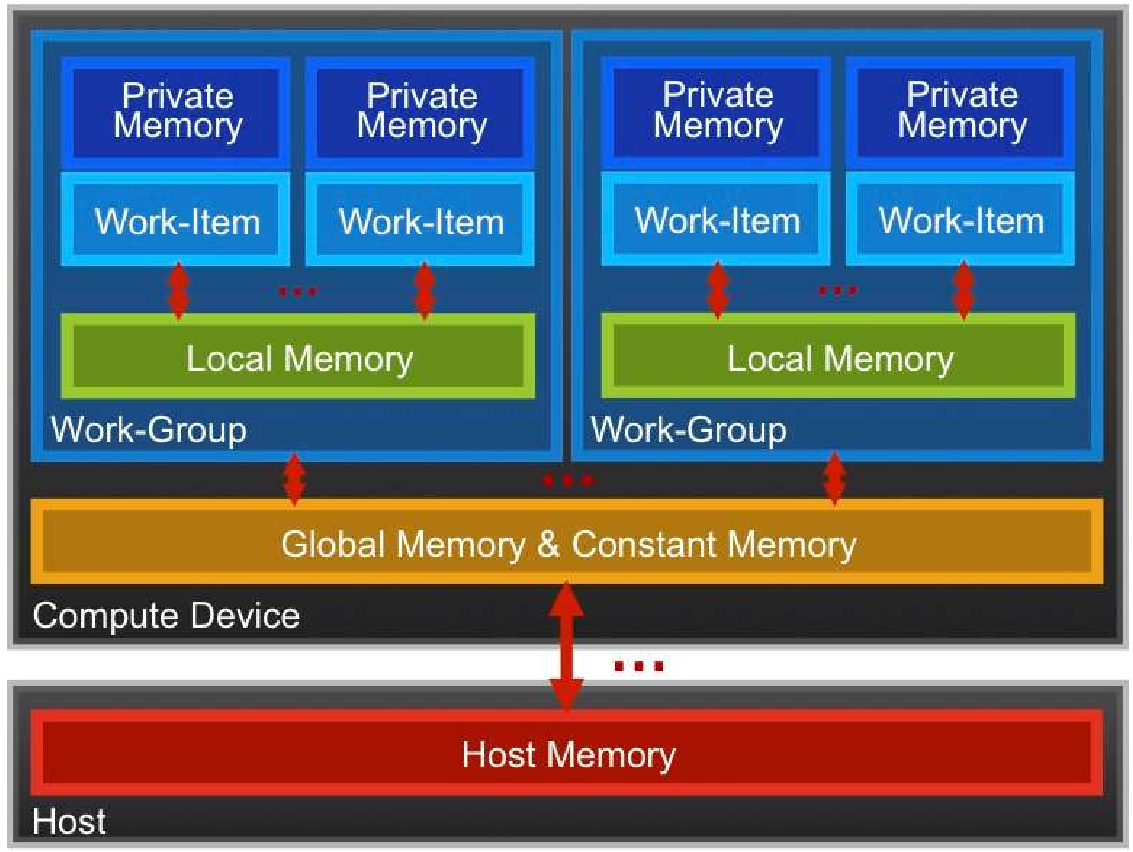
- The problem we want to compute should have some **dimensionality**;
  - For example, compute a kernel on all points in a cube
- When we execute the kernel we specify **up to 3 dimensions**
- We also **specify the total problem size** in each dimension – this is called the **global** size
- We associate each point in the iteration space with a **work-item**

OpenCL N Dimensional Range (NDRange)

- Work-items are grouped into **work-groups**; work-items within a work-group can share **local memory** and can **synchronize**
- We can specify the number of work-items in a work-group – this is called the **local** (work-group) size
- Or the OpenCL run-time can choose the work-group size for you (usually not optimally)

OpenCL Memory model

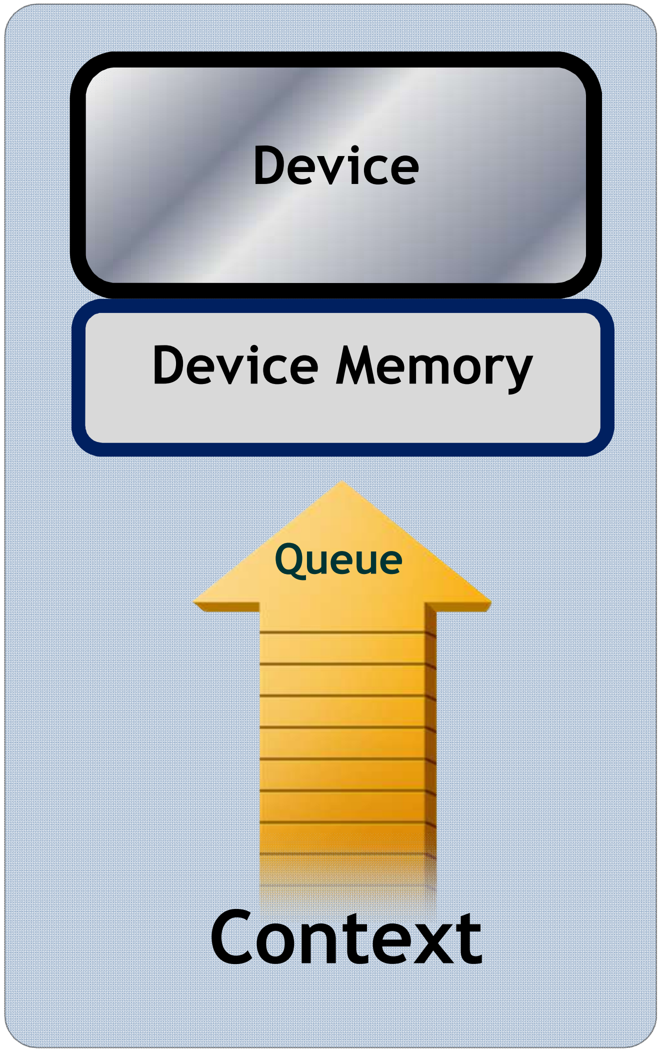
- *Private Memory*
  - Per work-item
- *Local Memory*
  - Shared within a work-group
- *Global Memory /Constant Memory*
  - Visible to all work-groups
- *Host memory*
  - On the CPU



Memory management is explicit:
You are responsible for moving data from host → global → local *and* back

Context and Command-Queues

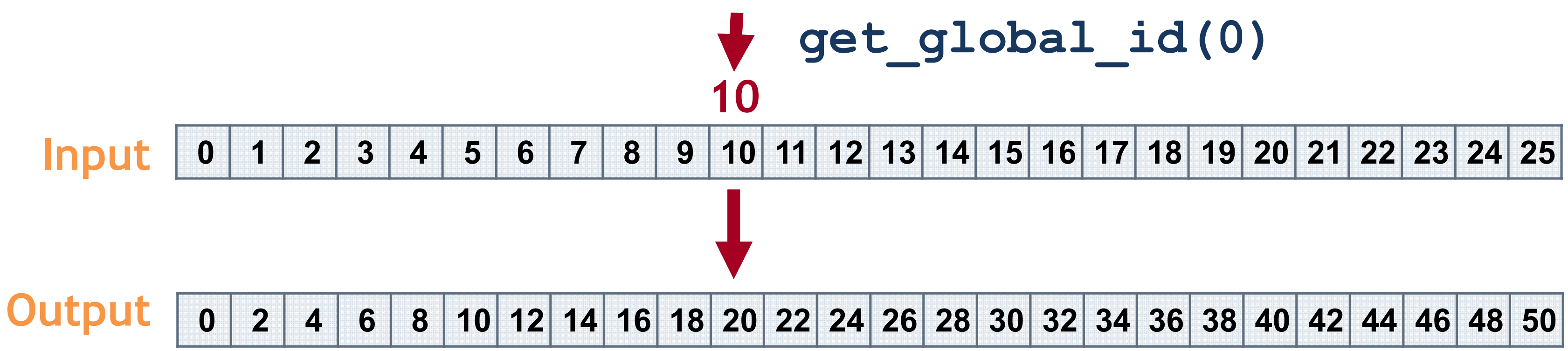
- ***Context***:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization, and memory transfer operations) are submitted through a ***command-queue***.
- Each ***command-queue*** points to a single device within a context.



Execution model (kernels)

- OpenCL execution model … define a problem domain and execute an instance of a **kernel** for each point in the domain

```
__kernel void times_two(
      __global float* input,
      __global float* output)
{
    int i = get_global_id(0);
    output[i] = 2.0f * input[i];
}
```

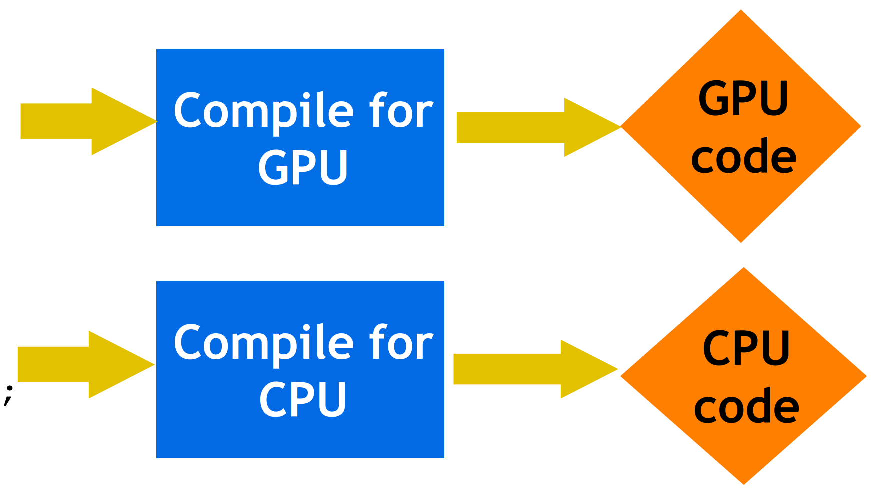


Building Program Objects

- The program object encapsulates:
  - A context
  - The program kernel source or binary
  - List of target devices and build options
- The C API build process to create a program object:
  - `clCreateProgramWithSource()`
  - `clCreateProgramWithBinary()`

OpenCL uses runtime compilation … because in general you don't know the details of the target device when you ship the program

```
__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);
}
```



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=0 to N-1
```

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

Vector Addition - Kernel

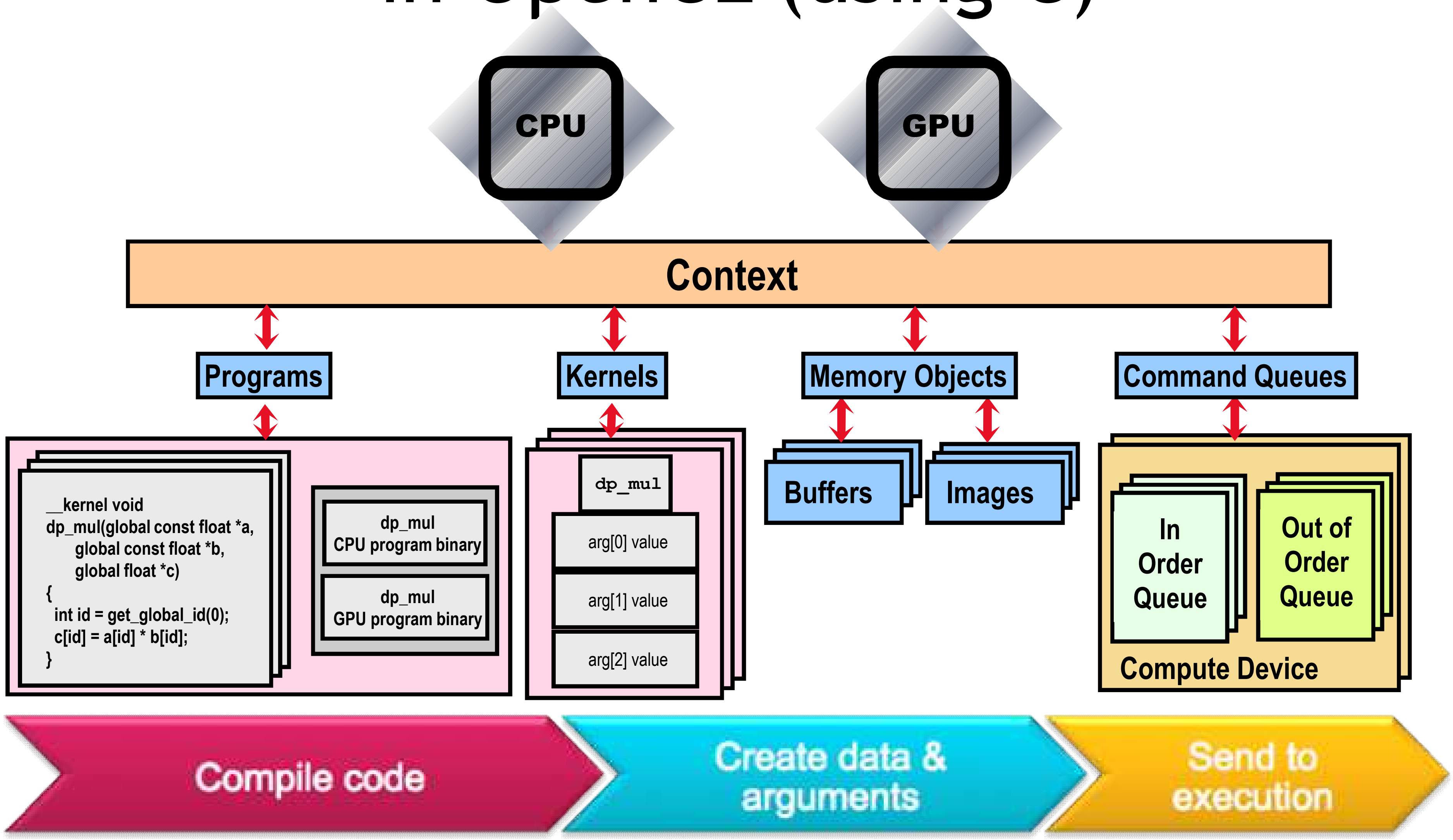
```
__kernel void vadd(__global const float *a,
                   __global const float *b,
                   __global       float *c)
{
    int gid = get_global_id(0);
    c[gid]  = a[gid] + b[gid];
}
```

Vector Addition – Host

- The host program is the code that runs on the host to:
  - Setup the environment for the OpenCL program
  - Create and manage kernels
- 5 simple steps in a basic host program:
  1. Define the ***platform*** … platform = devices+context+queues
  2. Create and Build the ***program*** (dynamic library for kernels)
  3. Setup ***memory*** objects
  4. Define the ***kernel*** (attach arguments to kernel functions)
  5. Submit ***commands*** … transfer memory objects and execute kernels

As we go over the next set of slides, cross reference content on the slides to the reference card. This will help you get used to the reference card and how to pull information from the card and express it in code.

The basic platform and runtime APIs in OpenCL (using C)



1. Define the platform

- Grab the first available platform:

```
err = clGetPlatformIDs(1, &firstPlatformId,
                          &numPlatforms);
```

- Use the first CPU device the platform provides:

```
err = clGetDeviceIDs(firstPlatformId,
            CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);
```

- Create a simple context with a single device:

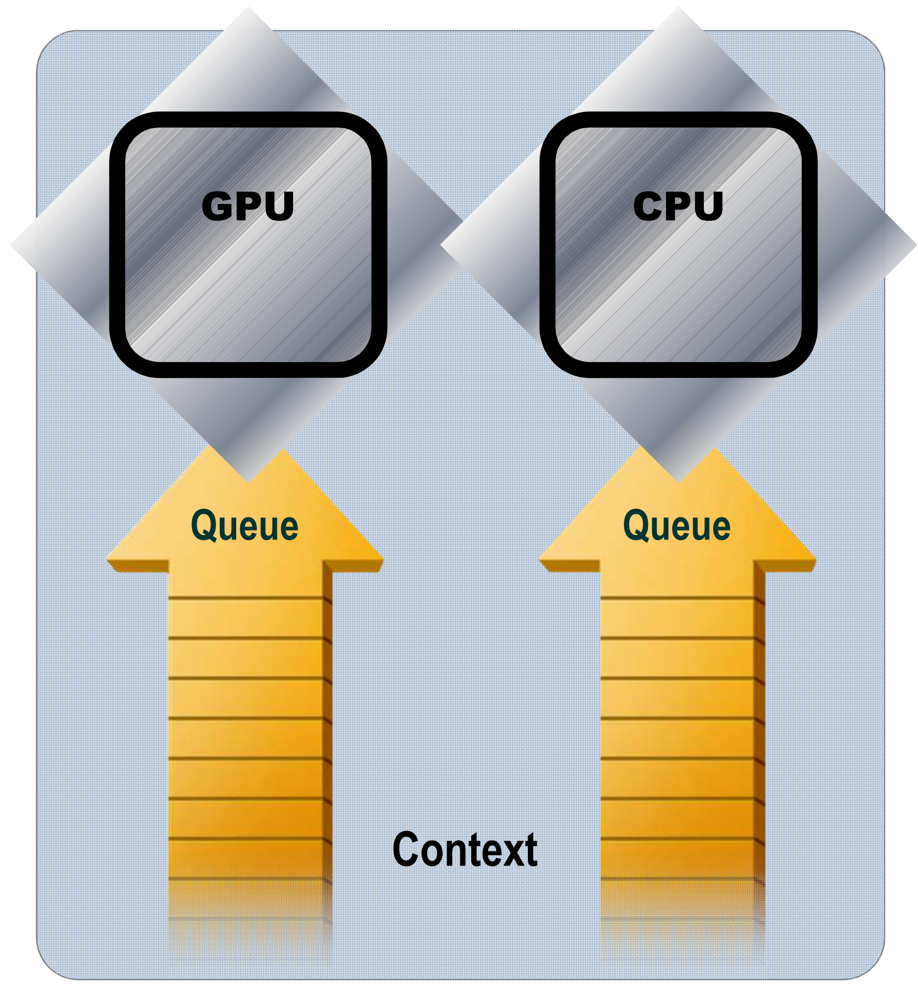
```
context = clCreateContext(firstPlatformId, 1,
                    &device_id, NULL, NULL, &err);
```

- Create a simple command-queue to feed our device:

```
commands = clCreateCommandQueue(context, device_id,
                                          0, &err);
```

Command-Queues

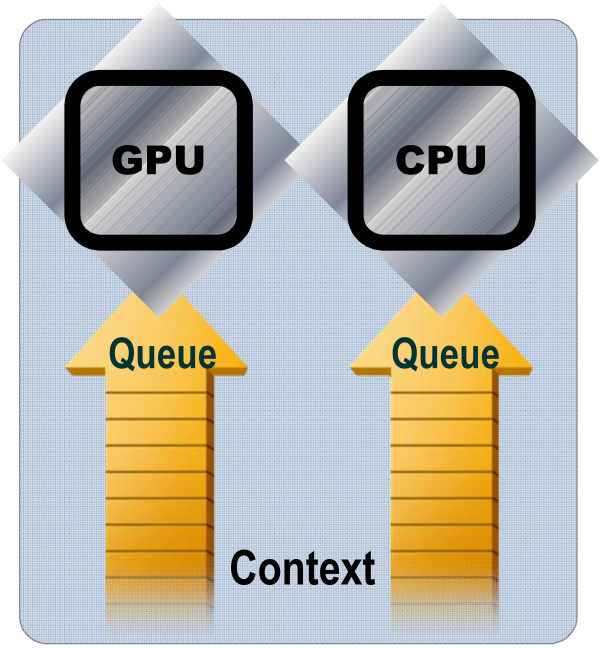
- Commands include:
  - Kernel executions
  - Memory object management
  - Synchronization
- The only way to submit commands to a device is through a command-queue.
- Each command-queue points to a single device within a context.
- Multiple command-queues can feed a single device.
  - Used to define independent streams of commands that don't require synchronization



Command-Queue execution details

Command queues can be configured in different ways to control how commands execute

- *In-order queues*:
  - Commands are enqueued and complete in the order they appear in the program (program-order)
- *Out-of-order queues*:
  - Commands are enqueued in program-order but can execute (and hence complete) in any order.
- Execution of commands in the command-queue are guaranteed to be completed at synchronization points
  - Discussed later



2. Create and Build the program

- Define source code for the kernel-program as a string literal (great for toy programs) or read from a file (for real applications).

- Build the program object:

```
program = clCreateProgramWithSource(context, 1
               (const char**) &KernelSource, NULL, &err);
```

- Compile the program to create a "dynamic library" from which specific kernels can be pulled:

```
err = clBuildProgram(program, 0, NULL,NULL,NULL,NULL);
```

Error messages

- Fetch and print error messages:

```
if (err != CL_SUCCESS) {
 size_t len;
 char buffer[2048];
 clGetProgramBuildInfo(program, device_id,
  CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
 printf("%s\n", buffer);
}
```

- Important to do check all your OpenCL API error messages!
- Easier in C++ with try/catch (see later)

3. Setup Memory Objects

- For vector addition we need 3 memory objects, one each for input vectors A and B, and one for the output vector C.
- Create input vectors and assign values on the host:

```
float h_a[LENGTH], h_b[LENGTH], h_c[LENGTH];
for (i = 0; i < length; i++) {
    h_a[i] = rand() / (float)RAND_MAX;
    h_b[i] = rand() / (float)RAND_MAX;
}
```

- Define OpenCL memory objects:

```
d_a = clCreateBuffer(context, CL_MEM_READ_ONLY,
                          sizeof(float)*count, NULL, NULL);
d_b = clCreateBuffer(context, CL_MEM_READ_ONLY,
                          sizeof(float)*count, NULL, NULL);
d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                          sizeof(float)*count, NULL, NULL);
```

What do we put in device memory?

Memory Objects:

- A handle to a reference-counted region of global memory.

There are two kinds of memory object

- *Buffer* object:
  - Defines a linear collection of bytes ("*just a C array*").
  - The contents of buffer objects are fully exposed within kernels and can be accessed using pointers
- *Image* object:
  - Defines a two- or three-dimensional region of memory.
  - Image data can only be accessed with read and write functions, i.e. these are opaque data structures. The read functions use a sampler.

Used when interfacing with a graphics API such as OpenGL. We won't use image objects in this tutorial.

Creating and manipulating buffers

- Buffers are declared on the host as type: `cl_mem`

- Arrays in host memory hold your original host-side data:

```
float h_a[LENGTH], h_b[LENGTH];
```

- Create the buffer (d_a), assign sizeof(float)*count bytes from "h_a" to the buffer and copy it into device memory:

```
cl_mem d_a = clCreateBuffer(context,
     CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
      sizeof(float)*count, h_a, NULL);
```

Conventions for naming buffers

- It can get confusing about whether a host variable is just a regular C array or an OpenCL buffer

- A useful convention is to prefix the names of your regular **h**ost C arrays with "**h_**" and your OpenCL buffers which will live on the **d**evice with "**d_**"

Creating and manipulating buffers

- Other common memory flags include:

  `CL_MEM_WRITE_ONLY, CL_MEM_READ_WRITE`

- These are from the point of view of the **device**

- Submit command to copy the buffer back to host memory at “h_c”:
  - CL_TRUE = blocking, CL_FALSE = non-blocking

```
clEnqueueReadBuffer(queue, d_c, CL_TRUE,
          sizeof(float)*count, h_c,
          NULL, NULL, NULL);
```

4. Define the kernel

- Create kernel object from the kernel function "vadd":

```
kernel = clCreateKernel(program, "vadd", &err);
```

- Attach arguments of the kernel function "vadd" to memory objects:

```
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int),
                                &count);
```

5. Enqueue commands

- Write Buffers from host into global memory (as non-blocking operations):

```
err = clEnqueueWriteBuffer(commands, d_a, CL_FALSE,
          0, sizeof(float)*count, h_a, 0, NULL, NULL);
err = clEnqueueWriteBuffer(commands, d_b, CL_FALSE,
          0, sizeof(float)*count, h_b, 0, NULL, NULL
```

- Enqueue the kernel for execution (note: in-order so OK):

```
err = clEnqueueNDRangeKernel(commands, kernel, 1,
                NULL, &global, &local, 0, NULL, NULL);
```

5. Enqueue commands

- Read back result (as a blocking operation). We have an in-order queue which assures the previous commands are completed before the read can begin.

```
err = clEnqueueReadBuffer(commands, d_c, CL_TRUE,
            sizeof(float)*count, h_c, 0, NULL, NULL);
```

Vector Addition – Host Program

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

// get the list of GPU devices associated with context
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
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, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcb, NULL);

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL, NULL);

// create the program
program = clCreateProgramWithSource(context, 1,
                               &program_source, NULL, NULL);
```

```
// build the program
err = clBuildProgram(program, 0, NULL,NULL,NULL,NULL);

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

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                         sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *) &memobjs[1],
                         sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
                         sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

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

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
                          CL_TRUE, 0,
                          n*sizeof(cl_float), dst,
                          0, NULL, NULL);
```

Vector Addition – Host Program

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

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

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

Define platform and queues

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

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                      sizeof(cl_float)*n, NULL, NULL);
```

Define memory objects

```
// create the program
program = clCreateProgramWithSource(context, 1,
                                         NULL, NULL);
```

Create the program

```
// build the
err = clBuild                        L,NULL,NULL);
```

Build the program

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

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],

err |= c                                   nobjs[1],
                      sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
                      sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;
```

Create and setup kernel

```
// execute kernel
err = clEnq                          rnel, 1, NULL,
                                     ,0,NULL,NULL);
```

Execute the kernel

```
// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
                          CL_TRUE, 0,
                                           dst,
                          0, NULL, NULL);
```

Read results on the host

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

Exercise 2: Running the Vadd kernel

- Goal:
  - To inspect and verify that you can run an OpenCL kernel
- Procedure:
  - Take the provided C Vadd program. It will run a simple kernel to add two vectors together.
  - Look at the host code and identify the API calls in the host code. Compare them against the API descriptions on the OpenCL reference card.
  - There are some helper files which time the execution, output device information neatly and check errors.
- Expected output:
  - A message verifying that the vector addition completed successfully