Course materials

In addition to these slides, C++ API header files, a set of exercises, and solutions, the following are useful:

OpenCL API 1.1 Quick Reference Card - Page 1

OpenCL (Open Computing Language) is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs and other processors. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high-performance compute servers, desktop computer systems and handheld devices.

[n.n.n] refers to the section in the API Specification available at www.khronos.org/opencl.

The OpenCL Platform Layer

Contexts [4.4]

Querying Platform Info and Devices [4.1, 4.2]

The OpenCL Runtime

Command Queues [5.1]

Buffer Objects

Create Buffer Objects [5.2.1]

Read, Write, Copy Buffer Objects [5.2.2]

Map Buffer Objects [5.2.2]

Query Buffer Object [5.4.4]

Program Objects

Create Program Objects [5.6.1]

Build Program Executable [5.6.2]

Build Options [5.6.4]

Query Program Objects [5.6.5]

(Program Objects Continue >)

©2010 Khronos Group - Rev. 0610

www.khronos.org/opencl

OpenCL C   1.2 Reference Card
OpenCL C++ 1.2 Reference Card

These cards will help you keep track of the API as you do the exercises:

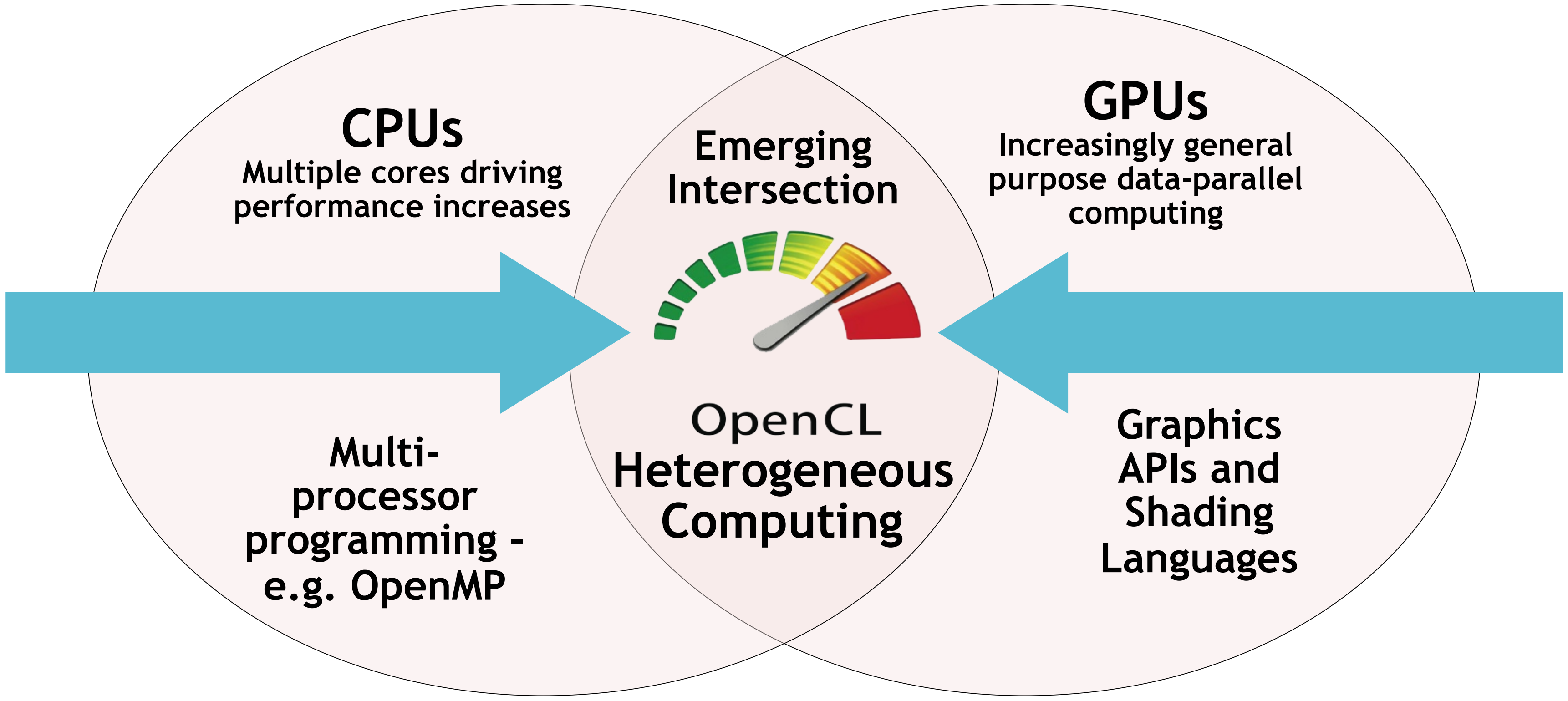
https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf

The v1.2 spec is also very readable and recommended to have on-hand:

https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

AN INTRODUCTION TO OPENCL

Industry Standards for Programming Heterogeneous Platforms


CPUs
Multiple cores driving performance increases
Multi-processor programming - e.g. OpenMP
Emerging Intersection
OpenCL
Heterogeneous Computing
GPUs
Increasingly general purpose data-parallel computing
Graphics APIs and Shading Languages

OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

The origins of OpenCL

AMD

ATI

Merged, needed commonality across products

NVIDIA

GPU vendor – wants to steal market share from CPU

Intel

CPU vendor – wants to steal market share from GPU

Apple

Was tired of recoding for many core, GPUs. Pushed vendors to standardize.

Wrote a rough draft straw man API

Khronos Compute group formed

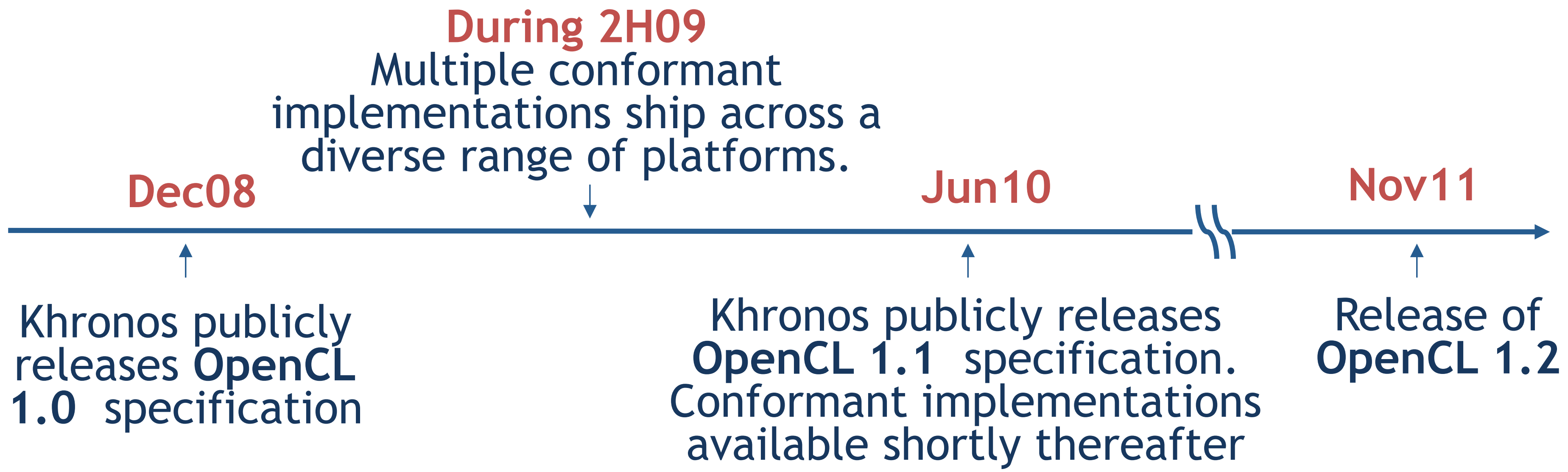
ARM
Altera
Xilinx
Sony
Qualcomm
Imagination
TI
+ many more

OpenCL

Third party names are the property of their owners.

OpenCL Timeline

- Launched Jun'08 … 6 months from "strawman" to OpenCL 1.0
- Rapid innovation to match pace of hardware innovation
  - 18 months from 1.0 to 1.1 and from 1.1 to 1.2
  - Goal: a new OpenCL every 18-24 months
  - Committed to backwards compatibility to protect software investments



During 2H09
Multiple conformant implementations ship across a diverse range of platforms.

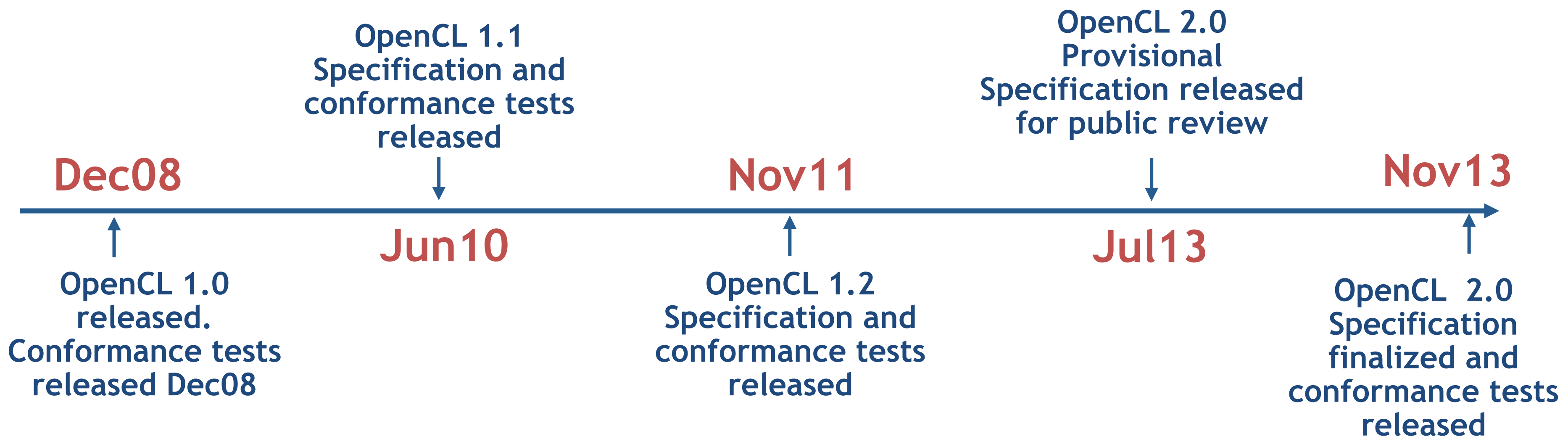
Dec08
Khronos publicly releases **OpenCL 1.0** specification

Jun10
Khronos publicly releases **OpenCL 1.1** specification. Conformant implementations available shortly thereafter

Nov11
Release of **OpenCL 1.2**

OpenCL Timeline

- Launched Jun'08 … 6 months from "strawman" to OpenCL 1.0
- Rapid innovation to match pace of hardware innovation
  - 18 months from 1.0 to 1.1 and from 1.1 to 1.2
  - Goal: a new OpenCL every 18-24 months
  - Committed to backwards compatibility to protect software investments

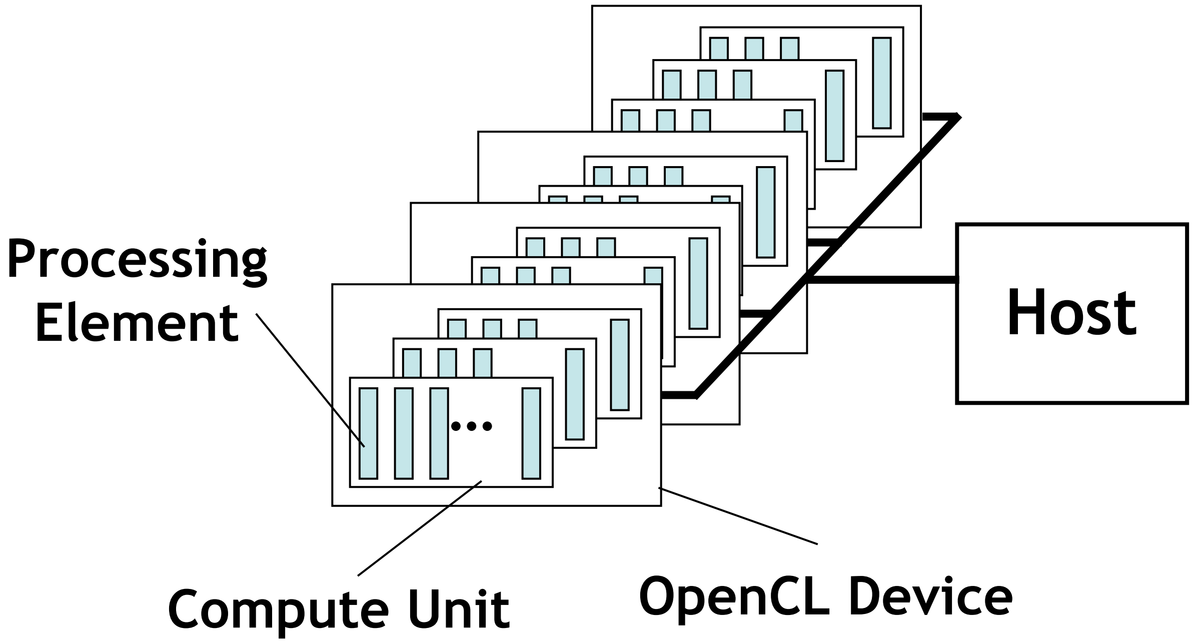

OpenCL 1.1 Specification and conformance tests released
OpenCL 2.0 Provisional Specification released for public review
Dec08
Nov11
Nov13
Jun10
Jul13
OpenCL 1.0 released. Conformance tests released Dec08
OpenCL 1.2 Specification and conformance tests released
OpenCL 2.0 Specification finalized and conformance tests released

OpenCL: From cell phone to supercomputer

- OpenCL Embedded profile for mobile and embedded silicon
  - Relaxes some data type and precision requirements
  - Avoids the need for a separate "ES" specification
- Khronos APIs provide computing support for imaging & graphics
  - Enabling advanced applications in, e.g., Augmented Reality
- OpenCL will enable parallel computing in new markets
  - Mobile phones, cars, avionics

A camera phone with GPS processes images to recognize buildings and landmarks and provides relevant data from internet

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

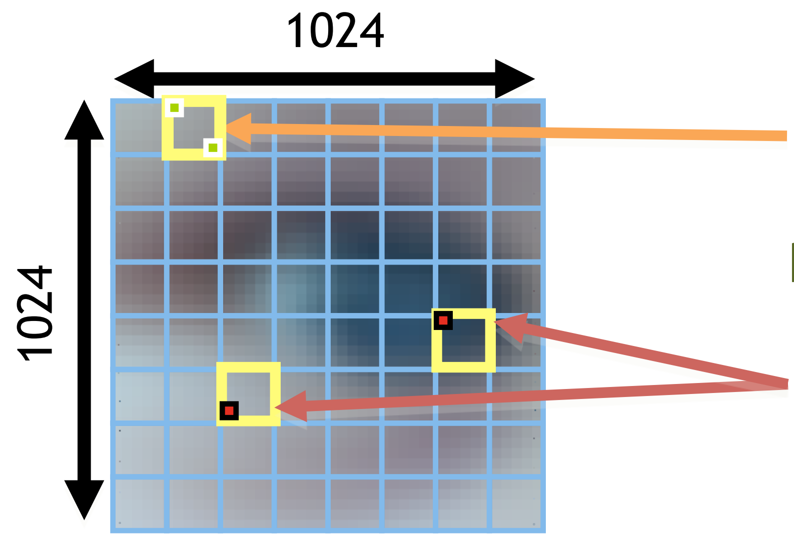
```
void
mul(const  int n,
    const  float *a,
    const  float *b,
           float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

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];
}
// execute over n work-items
```

An N-dimensional domain of work-items

- **Global** Dimensions:
  - 1024x1024 (whole problem space)
- **Local** Dimensions:
  - 128x128 (**work-group**, executes together)



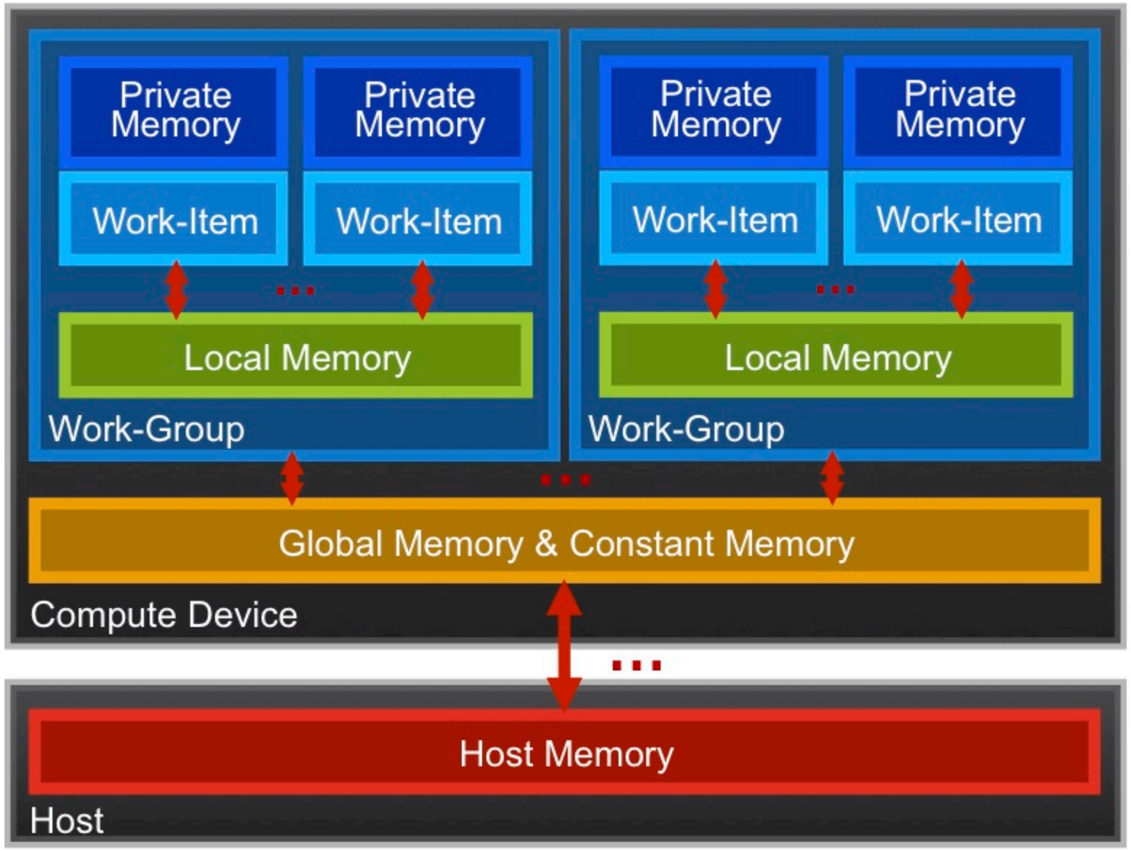
Synchronization between work-items possible only within work-groups: barriers and memory fences

Cannot synchronize between work-groups within one kernel

- Kernels can be 1D, 2D or 3D

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

The Memory Hierarchy

Bandwidths

Private memory
O(2-3) words/cycle/WI

Local memory
O(10) words/cycle/WG

Global memory
O(200-300) GBytes/s

Host memory
O(10) GBytes/s

Sizes

Private memory
O(10) words/WI

Local memory
O(10) KBytes/WG

Global memory
O(10) GBytes

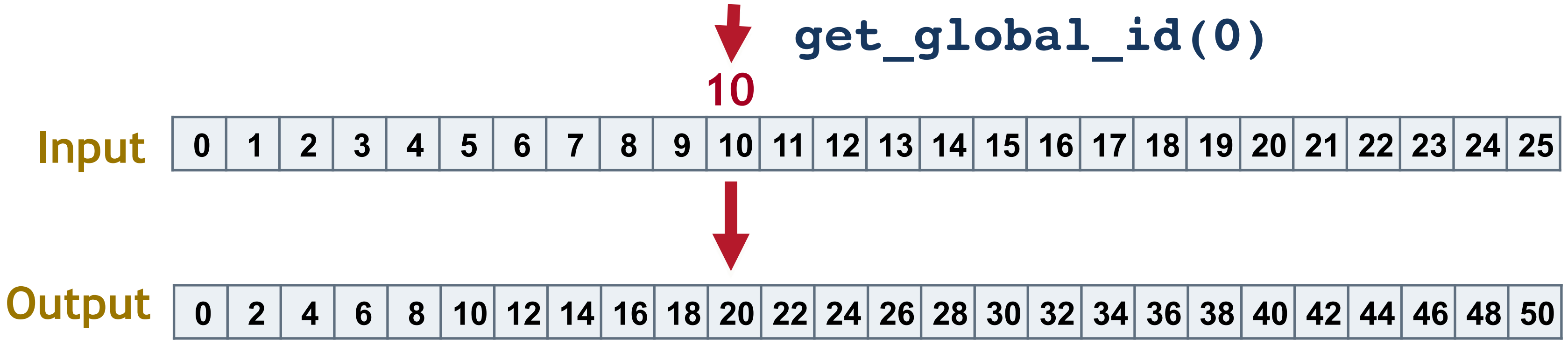
Host memory
O(1-100) GBytes

Speeds and feeds approx. for a high-end discrete GPU, circa 2014

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];
}
```



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];
 }
```

UNDERSTANDING THE HOST PROGRAM

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 function)
  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 your 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 C++ Interface

- Khronos has defined a common C++ header file containing a high level interface to OpenCL, **cl.hpp**
- This interface is dramatically easier to work with[1]
- Key features:
  - Uses common defaults for the platform and command-queue, saving the programmer from extra coding for the most common use cases
  - Simplifies the basic API by bundling key parameters with the objects rather than requiring verbose and repetitive argument lists
  - Ability to "call" a kernel from the host, like a regular function
  - Error checking can be performed with C++ exceptions

[1] especially for C++ programmers…

C++ Interface: setting up the host program

- Enable OpenCL API Exceptions. Do this before including the header file

```
#define __CL_ENABLE_EXCEPTIONS
```

- Include key header files … both standard and custom

```
#include <CL/cl.hpp>    // Khronos C++ Wrapper API
#include <cstdio>       // C style IO (e.g. printf)
#include <iostream>     // C++ style IO
#include <vector>       // C++ vector types
```

- Define key namespaces

```
using namespace cl;
using namespace std;
```

For information about C++, see the appendix: "C++ for C programmers".

1. Create a context and queue

- Grab a context using a device type:

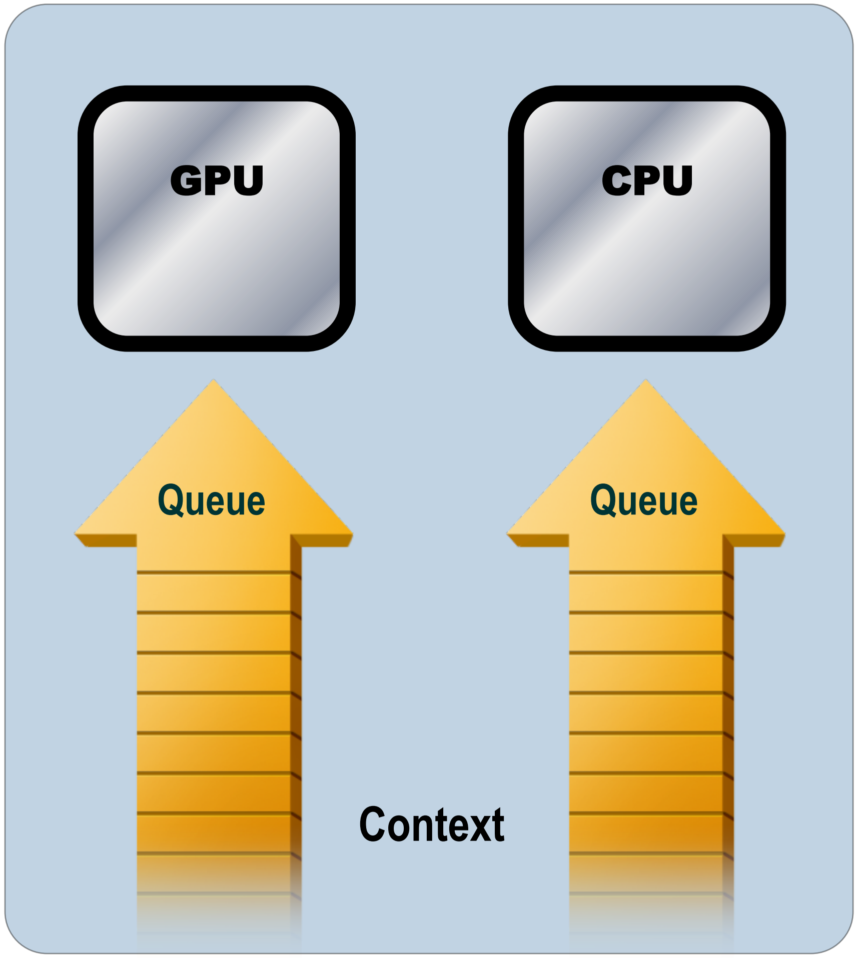
```
cl::Context
        context(CL_DEVICE_TYPE_DEFAULT);
```

- Create a command queue for the first device in the context:

```
cl::CommandQueue queue(context);
```

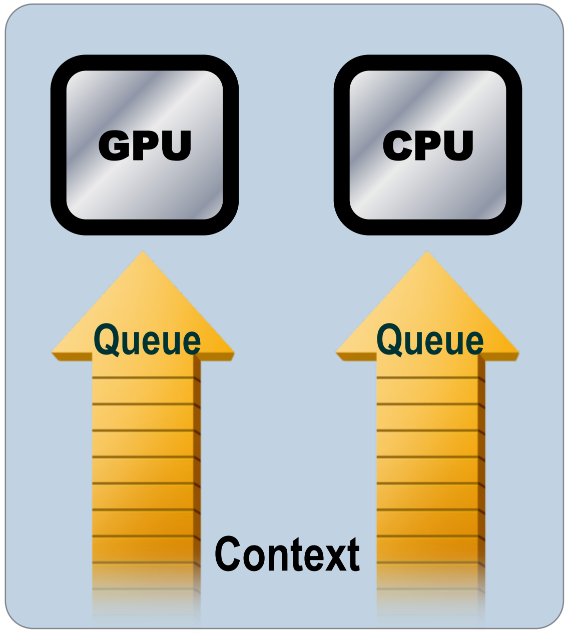
Commands and Command-Queues

- 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 either as a string literal (great for toy programs) or read it from a file (for real applications).

- Create the **program object and compile** to create a "dynamic library" from which specific kernels can be pulled:

"true" tells OpenCL to build (compile/link) the program object

```
cl::Program program(context, KernelSource, true);
```

KernelSource is a string … either statically set in the host program or returned from a function that loads the kernel code from a file.

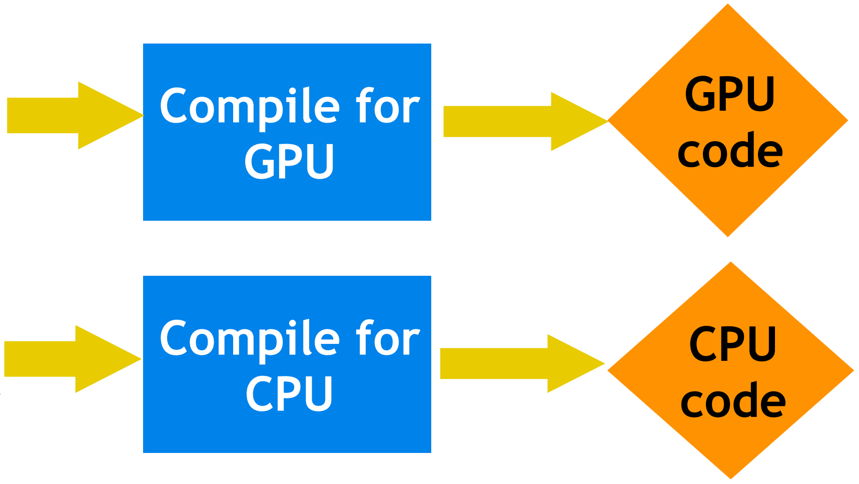
Building Program Objects

- The program object encapsulates:
  - A context
  - The program source or binary, and
  - List of target devices and build options
- The build process to create a program object:

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

```
cl::Program program(context, KernelSource, true);
```

```
__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);
}
```



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**:

```
std::vector<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 device buffers** and copy from **host buffers**:

```
cl::Buffer d_a(context, begin(h_a), end(h_a), true);
cl::Buffer d_b(context, begin(h_b), end(h_b), true);
cl::Buffer d_c(context, CL_MEM_WRITE_ONLY,
                                 sizeof(float)*LENGTH);
```

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.
    - 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.

Images used when interfacing with a graphics API such as OpenGL. We won't use these in our class.

Creating and manipulating buffers

- Buffers are declared on the host as object type: **`cl::Buffer`**

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

```
std::vector<float> h_a, h_b;
```

- Create the device-side **buffer** (d_a), assign read only memory to hold the host array (h_a) and copy it into device memory:

```
cl::Buffer  d_a(context, begin(h_a), end(h_a), true);
```

Start_iterator and end_iterator for the container holding host side object

Stipulates that this is a read-only buffer

Creating and manipulating buffers

- The last argument sets the device's read/write access to the Buffer. **true** means “read only” while **false** (the default) means “read/write”.

- Can use explicit copy commands to copy from the device buffer (in global memory) to host memory:

```
cl::copy(queue, d_c, begin(h_c), end(h_c));
```

- Can also copy from host memory to global memory:

```
cl::copy(queue, begin(h_c), end(h_c), d_c);
```

4. Define the kernel

- Create a ***kernel functor*** for the kernels you want to be able to "call" from the **program**:

```
auto vadd = cl::make_kernel
        <cl::Buffer, cl::Buffer, cl::Buffer>
        (program, “vadd”);
```

Must match the pattern of arguments to the kernel.

A previously created “program object” serving as a dynamic library of kernels

The name of the function used for the kernel

- This means you can ‘call’ the kernel as a ‘function’ in your host code to enqueue the kernel.

5. Enqueue commands

- For kernels, specify *global* and *local* dimensions
  - cl::NDRange global(1024) – sets a 1D global size of 1024
  - Cl::NDRange local(64) – sets a 1D local (workgroup) size of 64
  - If you don't specify a local dimension, it is assumed as cl::NullRange, and the runtime chooses a size for you

- Enqueue the kernel for execution (note: non-blocking on the host, i.e. host can get on with other things):

```
vadd(cl::EnqueueArgs(queue, global), d_a, d_b, d_c);
```

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

```
cl::copy(queue, d_c, begin(h_c), end(h_c));
```

C++ interface: The vadd host program

```
#define N 1024
using namespace cl;

int main(void) {
vector<float>
   h_a(N), h_b(N), h_c(N);
// initialize these host vectors…
Buffer d_a, d_b, d_c;

Context
  context(CL_DEVICE_TYPE_DEFAULT);
CommandQueue
  queue(context);

Program  program(
    context,
    loadprogram(“vadd.cl”), true);

// Create the kernel functor
auto vadd = make_kernel
        <Buffer, Buffer, Buffer>
        (program, “vadd”);
```

```
// Create buffers
// True indicates CL_MEM_READ_ONLY
// False indicates CL_MEM_READ_WRITE

d_a =
 Buffer(context,begin(h_a),end(h_a),true);
d_b =
 Buffer(context,begin(h_b),end(h_b),true);
d_c =
 Buffer(context,
 CL_MEM_WRITE_ONLY,sizeof(float)*N);

// Enqueue the kernel
vadd(EnqueueArgs(queue, NDRange(N)),
         d_a, d_b, d_c);

copy(queue, d_c, begin(h_c), end(h_c));
}
```

UNDERSTANDING THE KERNEL PROGRAM

Working with Kernels (C++)

- The kernels are where all the action is in an OpenCL program

- Steps to using kernels:
  1. Load kernel source code into a **program object** from a file
  2. Make a **kernel functor** from a function within the program
  3. Initialize **device memory**
  4. Call the **kernel functor**, specifying memory objects and global/local sizes
  5. Read **results** back from the device

- Note the kernel function argument list must match the kernel definition on the host

Create a kernel

- Kernel code:
  - Can be a string in the host code (for “toy codes”).
  - Or can be loaded from a file as a string or binary.

- Compile for the default devices within the default Context

```
program.build();
```

The build step can be carried out by specifying *true* in the program constructor. If you need to specify build flags you must specify *false* in the constructor and use this method instead.

- Define the kernel functor from a kernel function within the OpenCL program – allows us to ‘call’ the kernel to enqueue it as if it were just another function

```
auto vadd = make_kernel<Buffer, Buffer, Buffer>
                        (program, “vadd”);
```

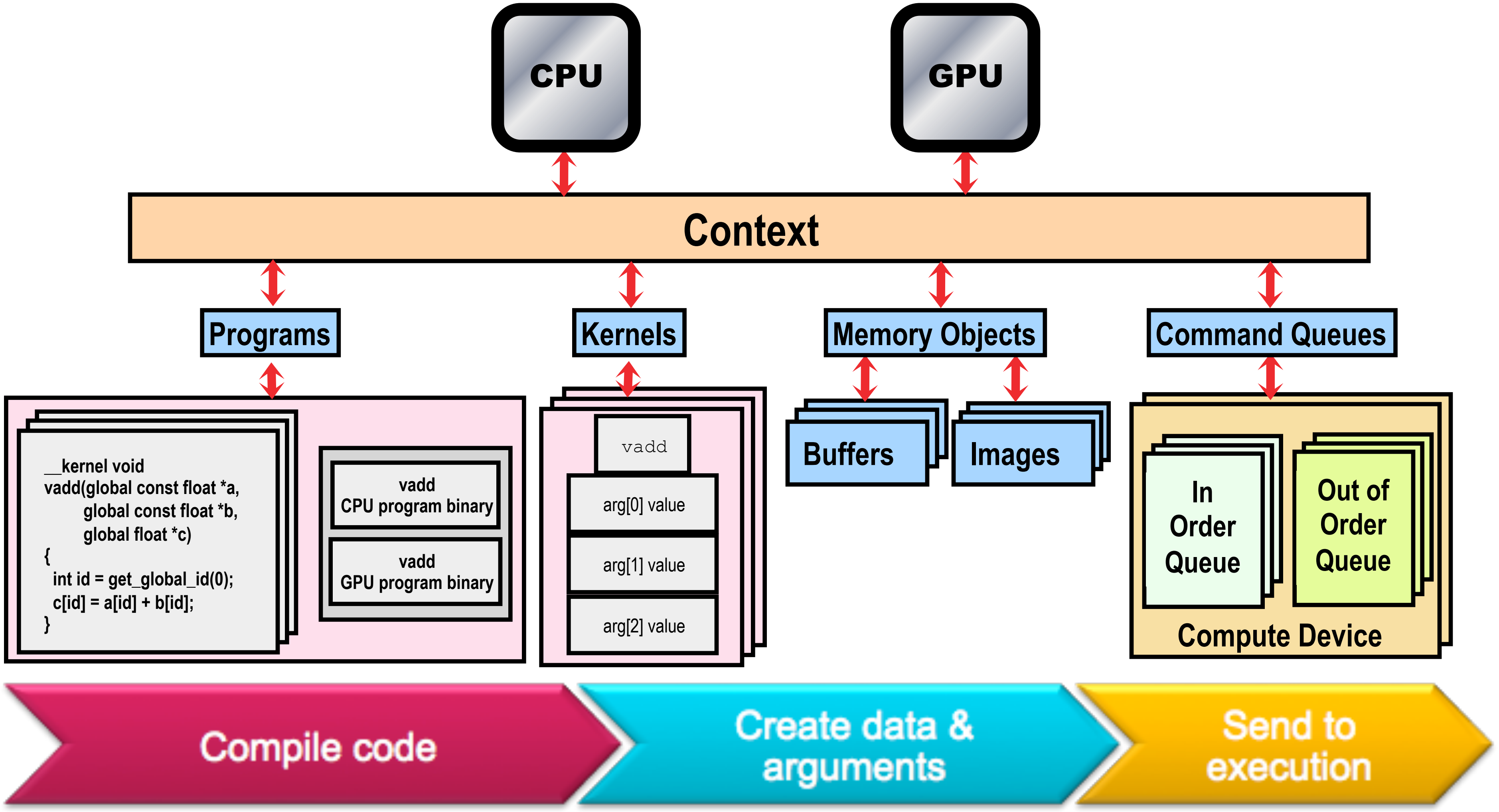
Call (enqueue) the kernel

- Enqueue the kernel for execution with buffer objects `d_a`, `d_b` and `d_c`:

We can include any arguments from the clEnqueueNDRangeKernel C API function (lots!)

```
vadd(
   EnqueueArgs(
    queue, NDRange(N), NDRange(local)),
   d_a, d_b, d_c );
```

We have now covered the basic platform runtime APIs in OpenCL



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