KITE
Khronos Initiative for Training & Education

Introduction to OpenCL

(utilisation des transparents de Cliff Woolley, NVIDIA)
intégré depuis à l'initiative Kite

riveill@unice.fr
http://www.i3s.unice.fr/~riveill

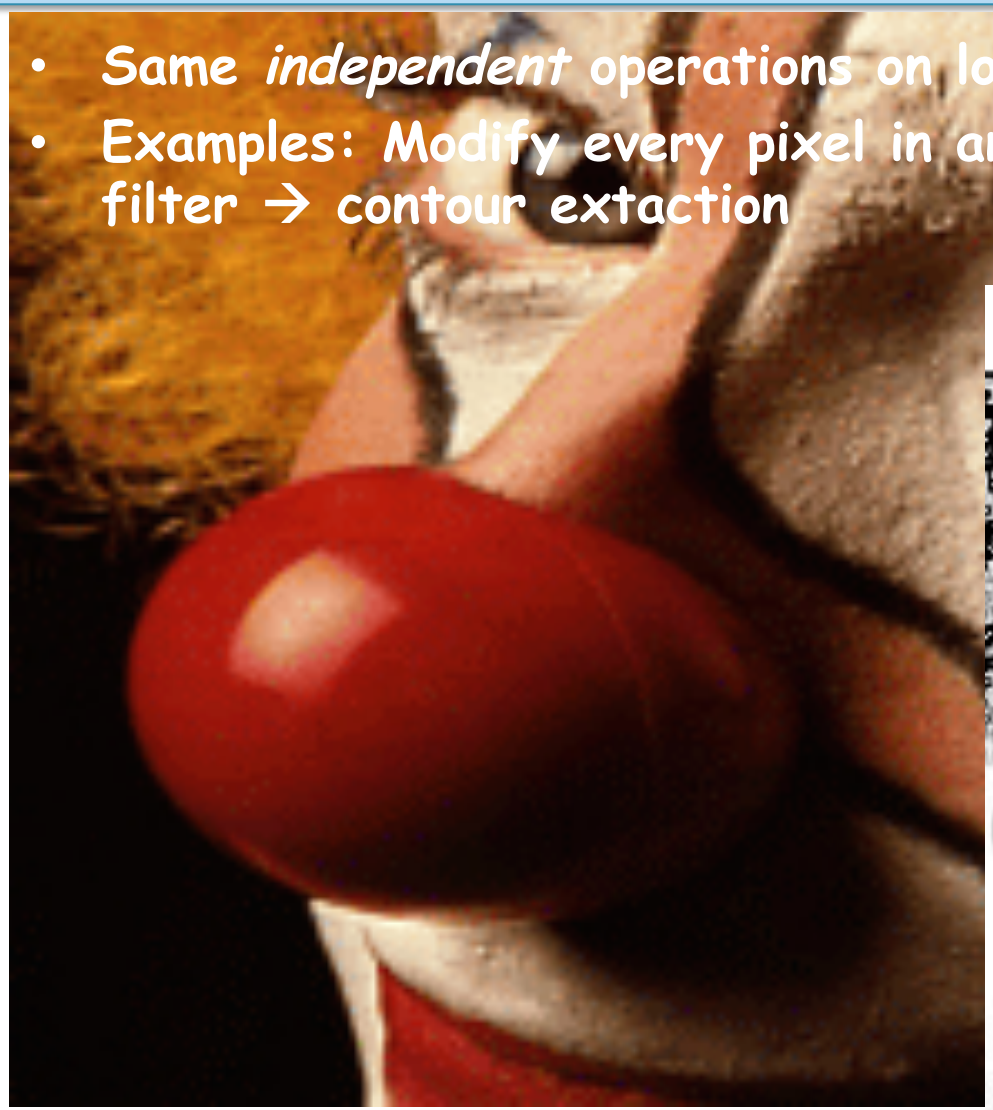
UNIVERSITÉ CÔTE D'AZUR

What is OpenCL good for

- Anything that is:
  - Computationally intensive
  - Data-parallel
  - Single-precision (subject to change in future)
- Portable application on different platform
  - CPU / GPU / FPGA / …
- Math operation >> memory operation
  - Remember: memory is slow
  - Linear algebra – matrix operation
  - Image processing

Data-parallelism

- Same *independent* operations on lots of data
- Examples: Modify every pixel in an image with *the same* filter → contour extaction



Data-parallelism

Application of the Solbel filter

```
// intensity
B[i,j] = 0.2989*rouge[i,j] + 0.5870*vert[i,j] + 0.1140*bleu[i,j]
// horizontal gradient
H[i,j] = -B[i-1,j-1] - 2*B[i-1,j] - B[i-1,j+1] + B[i+1,j-1] + 2*B[i+1,j] + B[i+1,j+1];
// vertical gradient
V[i,j] = - B[i-1,j-1] + B[i-1,j+1] - 2*B[i,j-1] + 2*B[i,j+1] - B[i+1,j-1] + B[i+1,j+1];
// Vector's norm
GN[i,j] = min(sqrt(H[i,j]*H[i,j] + V[i,j]*V[i,j]), 255);
```

General compute model

- Parallelism is defined by the 1D, 2D, or 3D global dimensions for each kernel execution (the task to be executed)
- A work-item / thread[1] is executed for every point in the global dimensions
- Examples
  - 1k audio: 1024 1024 work-items
  - HD video: 1920x1080 2M work-items
  - 3D MRI[2]: 256*256*256 16M work-items
  - HD per line: 1080 1080 work-items
  - HD per 8x8 block: 240*135 32k work-items

[1]work-item → OpenCL / thread → CUDA

[2]Magnetic Resonance Imaging

Today agenda

Lectures	Examples
Setting up OpenCL Platforms	Set up OpenCL
An overview of OpenCL	**Run the platform info command**
Important OpenCL concepts	**Running the Vadd kernel**
Overview of OpenCL APIs	**Chaining Vadd kernels**
A hosts view of working with kernels	The D = A+B+C problem
Introduction to OpenCL kernel programming	Matrix Multiplication
Understanding the OpenCL memory hierarchy	Optimize matrix multiplication
Synchronization in OpenCL	**The Pi program**
Heterogeneous computing with OpenCL	Run kernels on multiple devices
Optimizing OpenCL performance	Profile a program
Enabling portable performance via OpenCL	Optimize matrix multiplication for cross-platform
Debugging OpenCL	
Porting CUDA to OpenCL	Port CUDA code to OpenCL

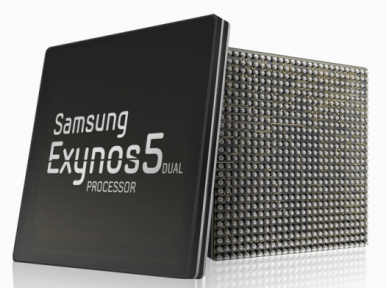
Lecture 2

AN OVERVIEW OF OPENCL

It's a Heterogeneous world

A modern computing platform includes:

- One or more CPUs
- One of more GPUs
- DSP processors
- Accelerators
- … other?



E.g. Samsung® Exynos 5:

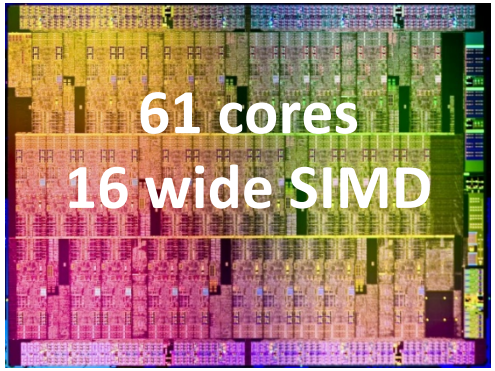
- Dual core ARM A15 1.7GHz, Mali T604 GPU

E.g. Intel XXX with IRIS

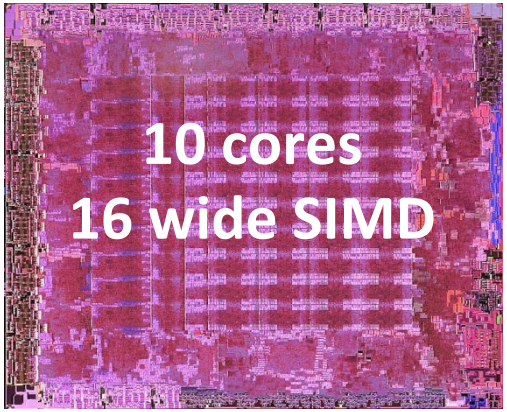
OpenCL lets Programmers write a single <u>portable</u> program that uses <u>ALL</u> resources in the heterogeneous platform

Microprocessor trends

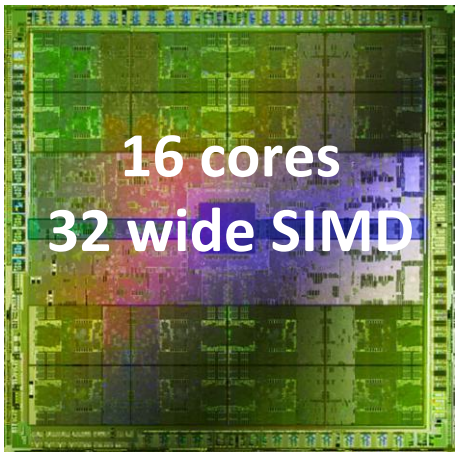
Individual processors have many (possibly heterogeneous) cores.



Intel® Xeon Phi™ coprocessor



ATI™ RV770

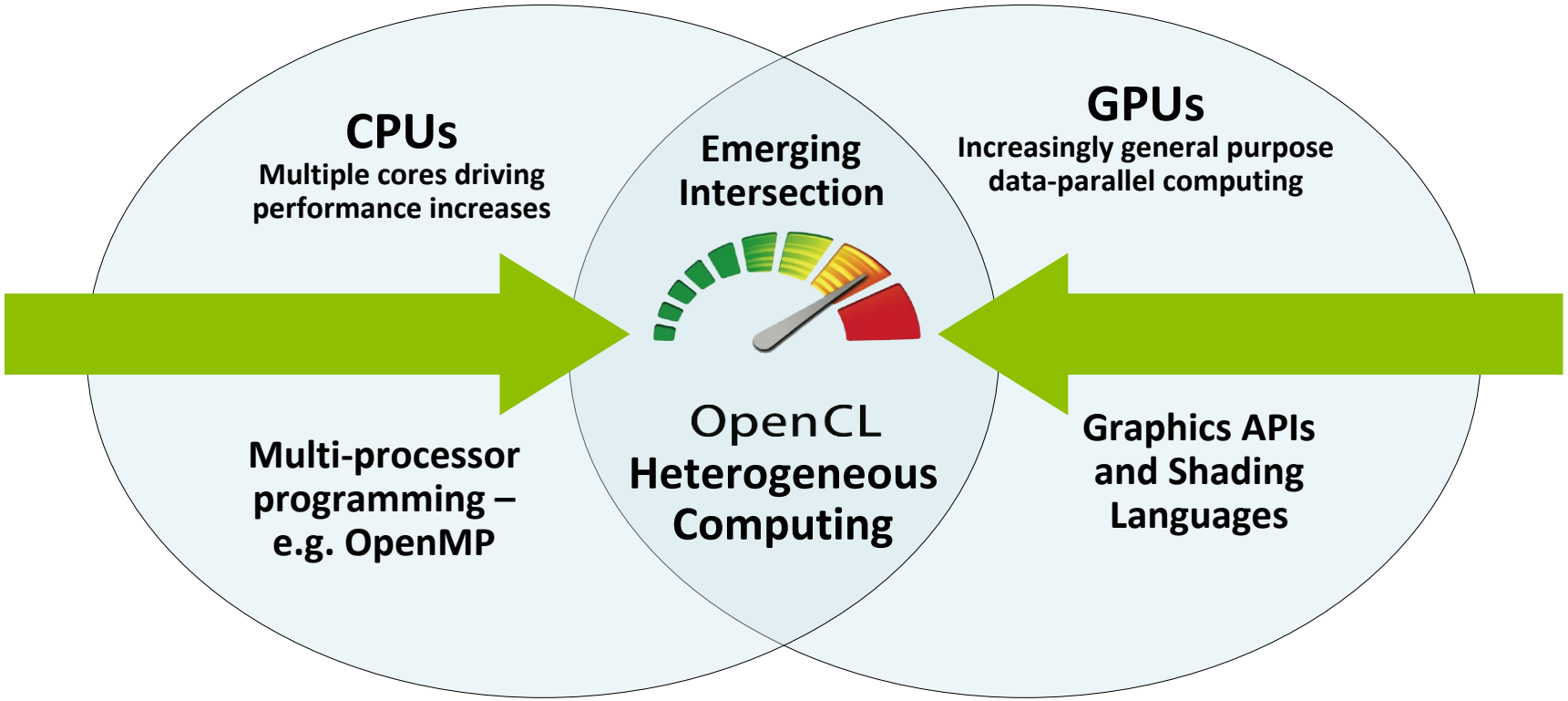


NVIDIA® Tesla® C2090

The Heterogeneous many-core challenge:

How are we to build a software ecosystem for the Heterogeneous many core platform?

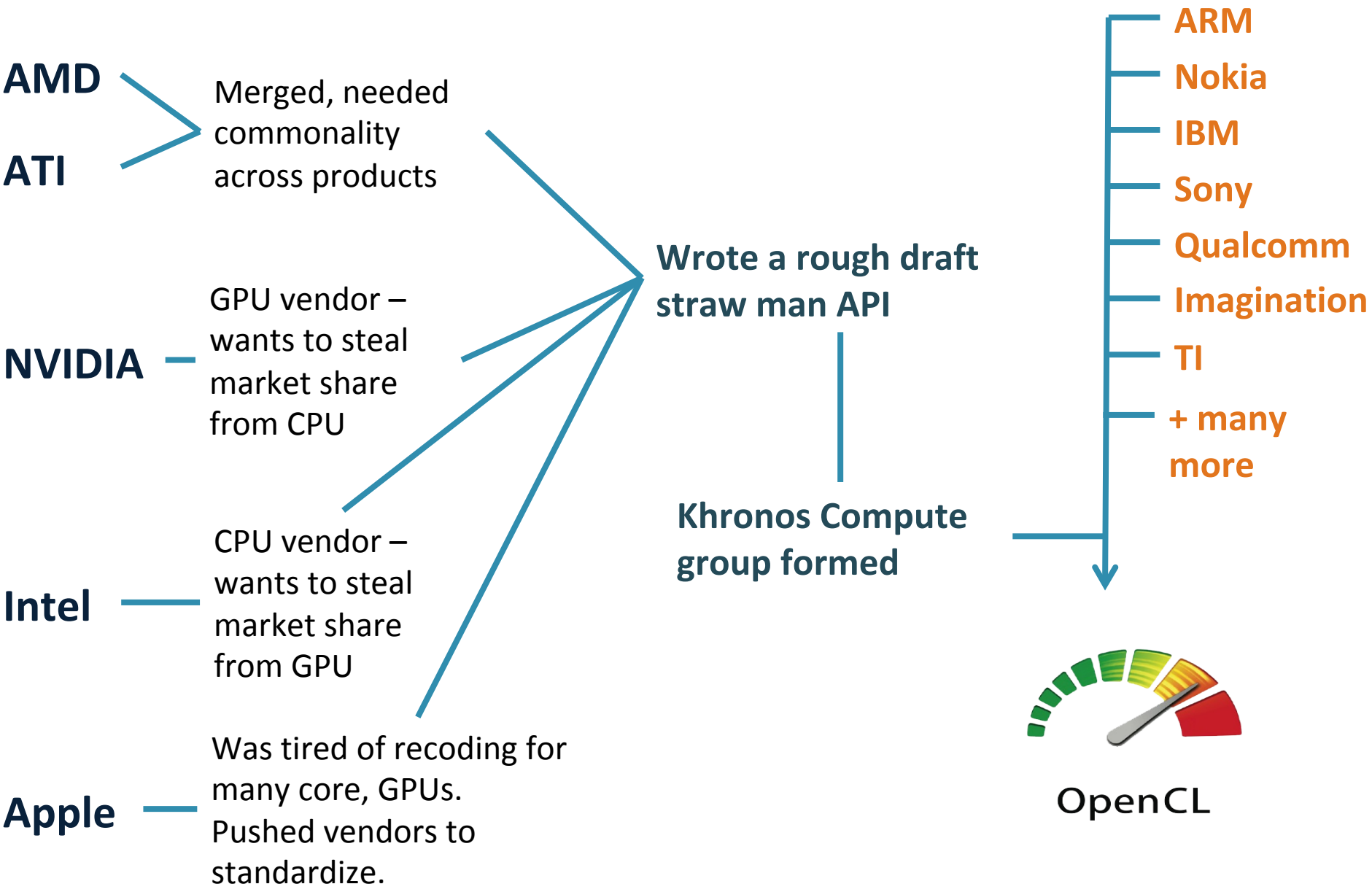
Industry Standards for Programming Heterogeneous Platforms



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

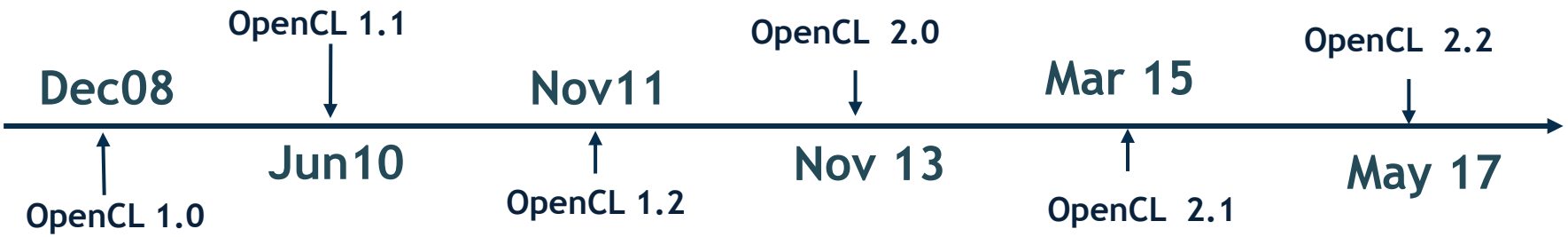


OpenCL Working Group within Khronos

- Diverse industry participation
  - Processor vendors, system OEMs, middleware vendors, application developers.
- OpenCL became an important standard upon release by virtue of the market coverage of the companies behind it.

OpenCL Timeline

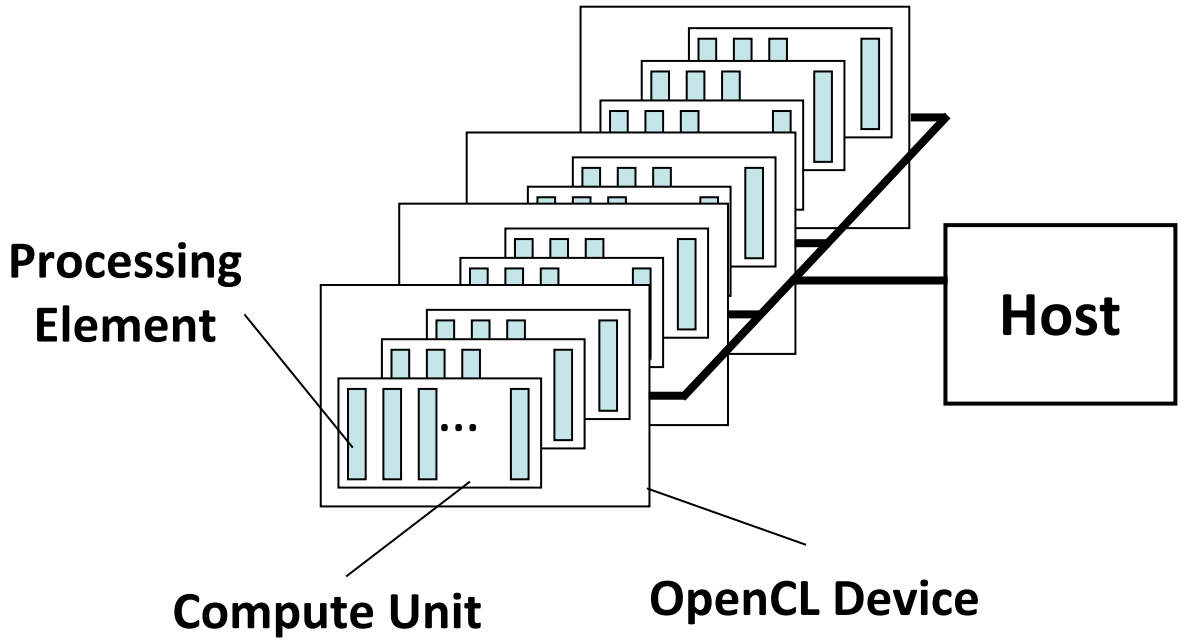
- Launched Jun'08 … 6 months from "strawman" to OpenCL 1.0
- Rapid innovation to match pace of hardware innovation
  - 18 months from 1.0 to 1.1 and from 1.1 to 1.2
  - Goal: a new OpenCL every 18-24 months
  - Committed to backwards compatibility to protect software investments



https://en.wikipedia.org/wiki/OpenCL

OpenCL + OpenGL − will be merge into − Vulkan (future)

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*

OpenCL Platform Example (One node, two CPU sockets, two GPUs)

CPUs:

- Treated as one OpenCL device
  - One CU per core
  - 1 PE per CU, or if PEs mapped to SIMD lanes, *n* PEs per CU, where *n* matches the SIMD width
- Remember:
  - the CPU will also have to be its own host!

GPUs:

- Each GPU is a separate OpenCL device
- Can use CPU and all GPU devices concurrently through OpenCL

CU = Compute Unit; PE = Processing Element

Example 1: Platform Information

- Goal:
  - Verify that you can run a simple OpenCL program.
- Procedure:
  - Take the provided **`DeviceInfo`** program, inspect it in the editor of your choice, build the program and run it.
- Expected output:
  - Information about the installed OpenCL platforms and the devices visible to them.
- Extension:
  - Run the command **`clinfo`** which comes as part of the AMD SDK but should run on all OpenCL platforms. This outputs all the information the OpenCL runtime can find out about devices and platforms.

En python

```
import pyopencl as cl
platforms = cl.get_platforms()
print ("\nNumber of OpenCL platforms:", len(platforms))
for p in platforms:
    print ("Platform:", p.name)              # Some information about the platforms
    print ("Vendor:", p.vendor)
    print ("Version:", p.version)
    devices = p.get_devices()               # Discover all devices
    print ("Number of devices:", len(devices))
    for d in devices:                       # Investigate each device
        print ("\t\tName:", d.name)          # some information about the devices
        print ("\t\tVersion:", d.opencl_c_version)
        print ("\t\tMax. Compute Units:", d.max_compute_units)
        print ("\t\tLocal Memory Size:", d.local_mem_size/1024, "KB")
        print ("\t\tGlobal Memory Size:", d.global_mem_size/(1024*1024), "MB")
        print ("\t\tMax Alloc Size:", d.max_mem_alloc_size/(1024*1024), "MB")
        print ("\t\tMax Work-group Total Size:", d.max_work_group_size)
        dim = d.max_work_item_sizes       # maximum dimensions of the work-groups
        print ("\t\tMax Work-group Dims:(", dim[0], " ".join(map(str, dim[1:])), ")")
```

En C/C++ (le code est beaucoup plus long ☹

```
int main(void)
{
   cl_int err;
   // Find of OpenCL platforms
   cl_uint num_platforms;
   err = clGetPlatformIDs(0, NULL,
&num_platforms);
   checkError(err, "Finding platforms");

   // Create a list of platform IDs
   cl_platform_id
platform[num_platforms];
   err = clGetPlatformIDs(num_platforms,
platform, NULL);

   printf("\nNumber of OpenCL platforms:
%d\n", num_platforms);
```

```
// Investigate each platform
for (int i = 0; i < num_platforms; i++) {
      cl_char string[10240] = {0};
      // Print out the platform name
      err = clGetPlatformInfo(platform[i],
CL_PLATFORM_NAME, sizeof(string),
&string, NULL);
     printf("Platform: %s\n", string);

      // Print out the platform vendor
      err = clGetPlatformInfo(platform[i],
CL_PLATFORM_VENDOR, sizeof(string),
&string, NULL);
      printf("Vendor: %s\n", string);

      // Print out the platform OpenCL
version
      err = clGetPlatformInfo(platform[i],
CL_PLATFORM_VERSION, sizeof(string),
&string, NULL);
       printf("Version: %s\n", string);
```

Résultat

MacBook Pro

Platform: Apple
Vendor: Apple
Version: OpenCL 1.2 (Sep 6 2017 16:05:06)
Number of devices: 2

Name: Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz
Version: OpenCL C 1.2
Max. Compute Units: 4
Local Memory Size: 32.0 KB
Global Memory Size: 16384.0 MB
Max Alloc Size: 4096.0 MB
Max Work-group Total Size: 1024
Max Work-group Dims:(1024 1 1)

Name: Intel(R) Iris(TM) Graphics 6100
Version: OpenCL C 1.2
Max. Compute Units: 48
Local Memory Size: 64.0 KB
Global Memory Size: 1536.0 MB
Max Alloc Size: 384.0 MB
Max Work-group Total Size: 256
Max Work-group Dims:(256 256 256)

Imac

Platform: Apple
Vendor: Apple
Version: OpenCL 1.2 (Sep 12 2017 16:28:17)
Number of devices: 2

Name: Intel(R) Core(TM) i5-2400S CPU @ 2.50GHz
Version: OpenCL C 1.2
Max. Compute Units: 4
Local Memory Size: 32.0 KB
Global Memory Size: 12288.0 MB
Max Alloc Size: 3072.0 MB
Max Work-group Total Size: 1024
Max Work-group Dims:(1024 1 1)

Name: ATI Radeon HD 6750M
Version: OpenCL C 1.2
Max. Compute Units: 6
Local Memory Size: 32.0 KB
Global Memory Size: 512.0 MB
Max Alloc Size: 128.0 MB
Max Work-group Total Size: 256
Max Work-group Dims:(256 256 256)

En détail : device[0]

MacBook Pro	iMac
Name: Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz Version: OpenCL C 1.2 **Max. Compute Units: 4** Local Memory Size: 32.0 KB Global Memory Size: 16384.0 MB **Max Alloc Size: 4096.0 MB** Max Work-group Total Size: 1024 Max Work-group Dims:(1024 1 1)	Name: Intel(R) Core(TM) i5-2400S CPU @ 2.50GHz Version: OpenCL C 1.2 **Max. Compute Units: 4** Local Memory Size: 32.0 KB Global Memory Size: 12288.0 MB **Max Alloc Size: 3072.0 MB** Max Work-group Total Size: 1024 Max Work-group Dims:(1024 1 1)

En détail : device[1] / MacBook Pro

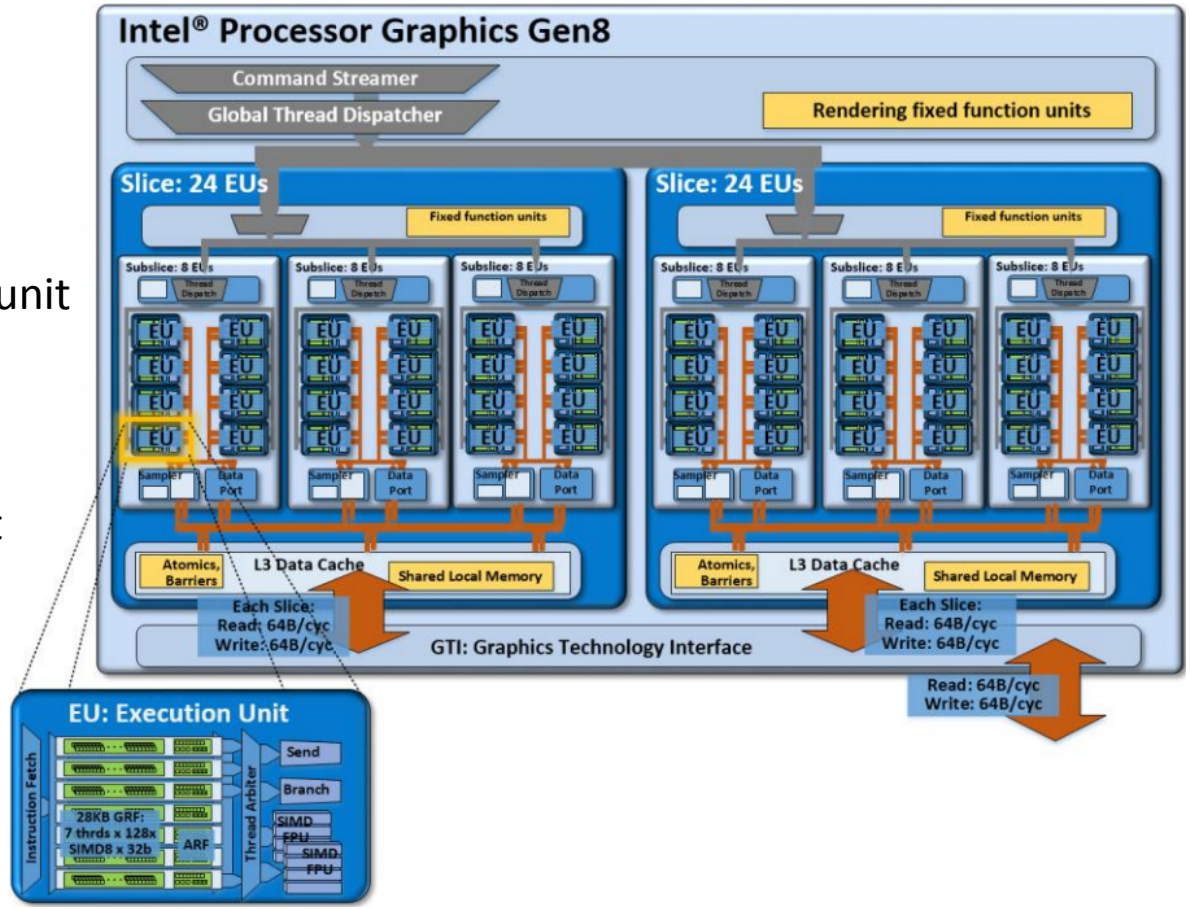
Name: Intel(R) Iris(TM) Graphics 6100
Version: OpenCL C 1.2

Max. Compute Units: 48
Local Memory Size: 64.0 KB
Global Memory Size: 1536.0 MB

Max Alloc Size: 384.0 MB
Max Work-group Total Size: 256
Max Work-group Dims:(256 256 256)

1 device = 2 warp
1 wrap = 3 group unit
1 group unit = 8 compute unit
1 compute unit = 7 thread

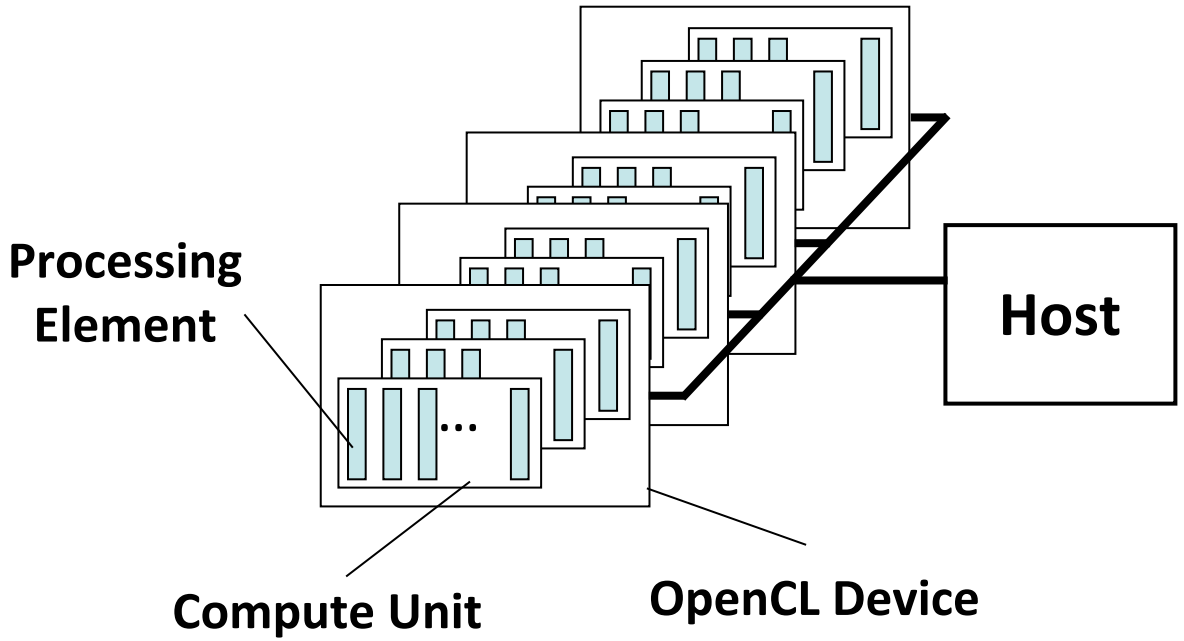
1 device = 2*3*8 =
48 compute unit



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

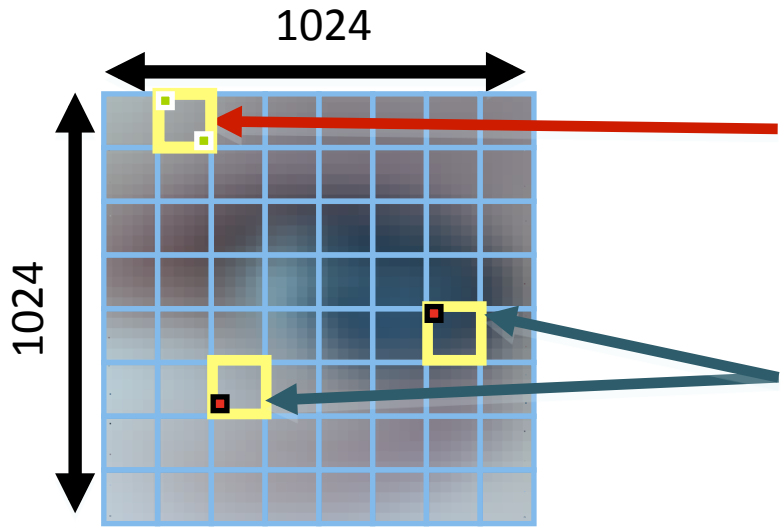
```
void
mul(const int n,
    const float *a,
    const float *b,
          float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

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)



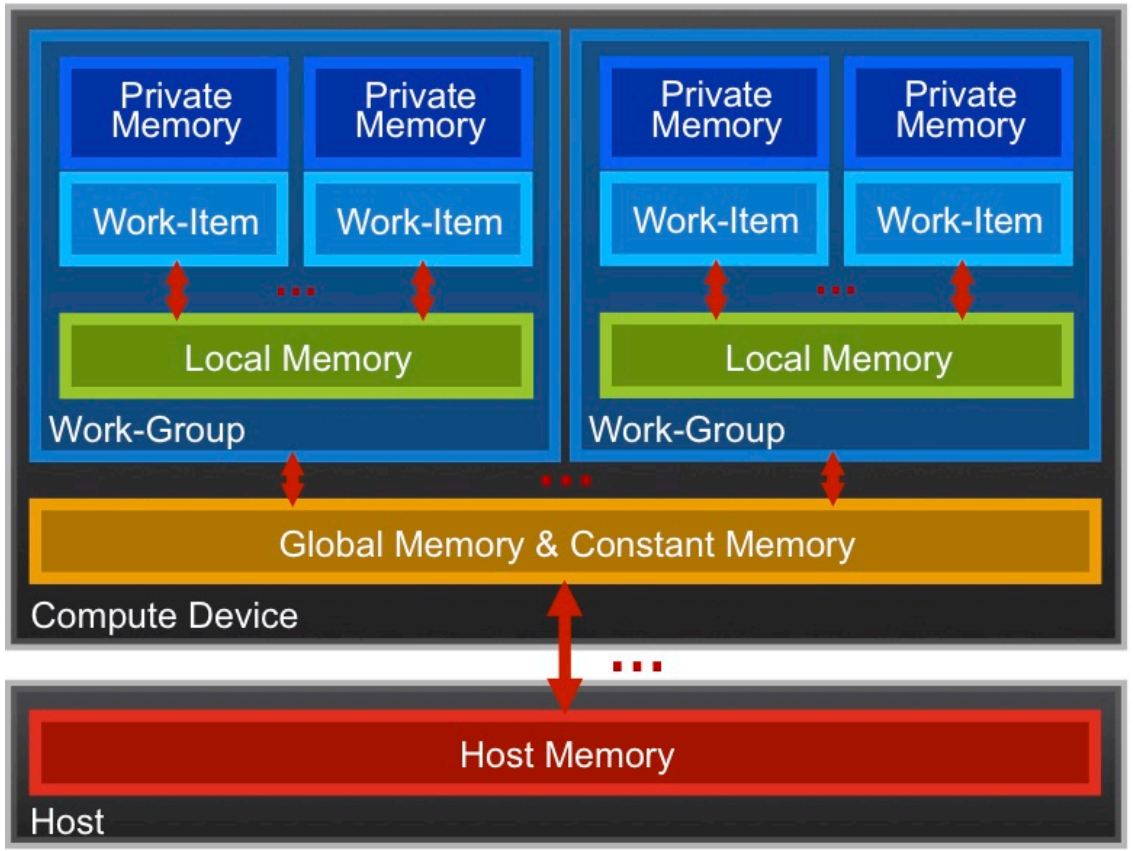
Synchronization between work-items possible only within work-groups: barriers and memory fences

Cannot synchronize between work-groups within a kernel

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

OpenCL Memory model

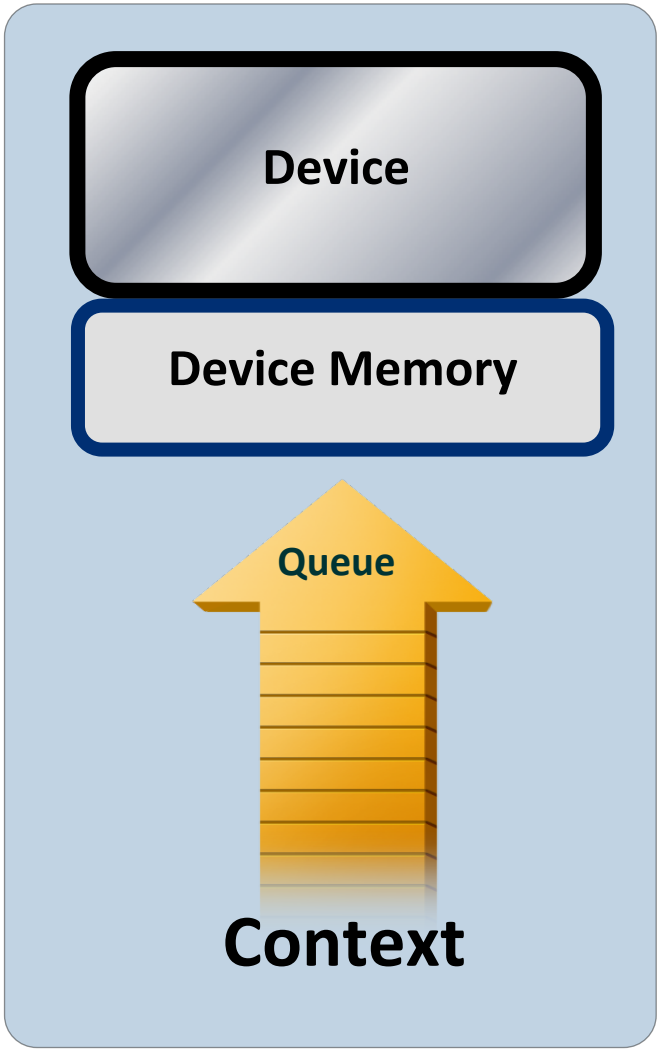
- ***Private Memory***
  - Per work-item
- ***Local Memory***
  - Shared within a work-group
- ***Global Memory / Constant Memory***
  - Visible to all work-groups
- ***Host memory***
  - On the CPU



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

Context and Command-Queues

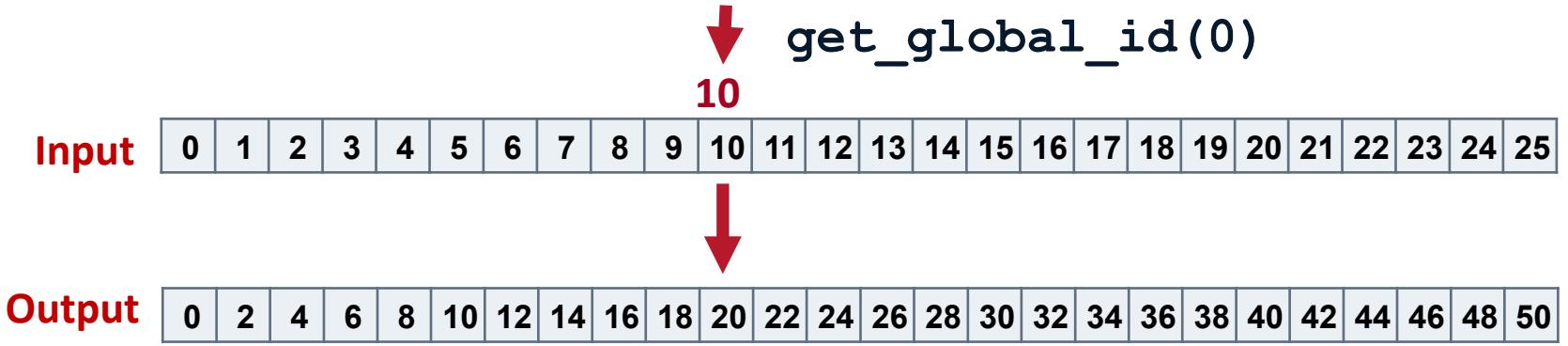
- ***Context***:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization, 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];
}
```

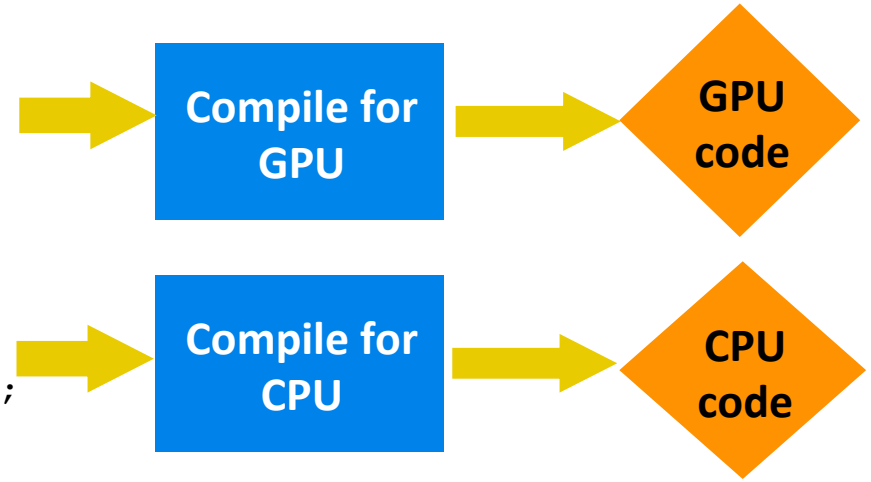


Building Program Objects

- 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 2: 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

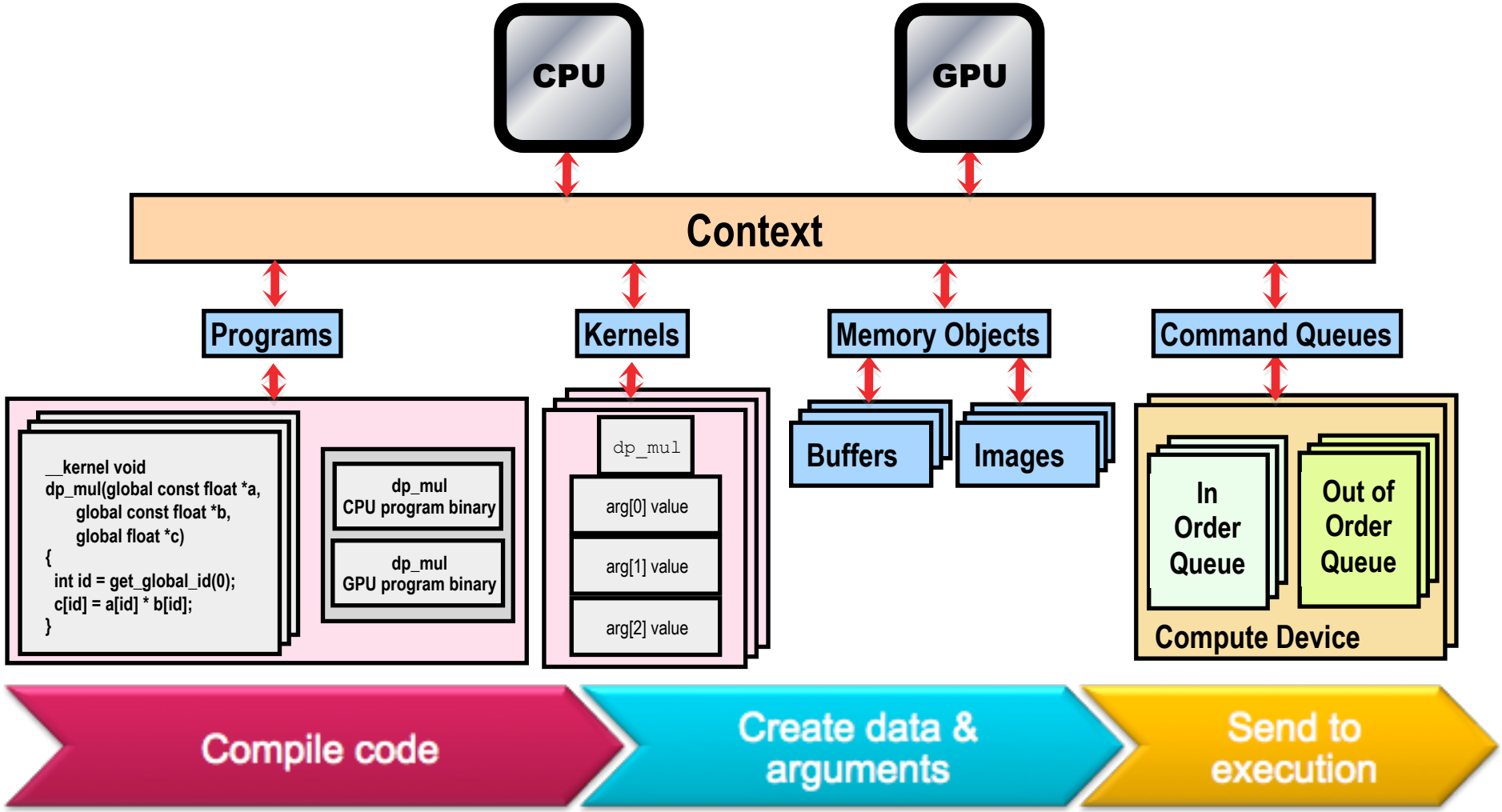
```
__kernel void vadd(__global const float *a,
                   __global const float *b,
                   __global       float *c)
{
    int gid = get_global_id(0);
    c[gid]  = a[gid] + b[gid];
}
```

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:

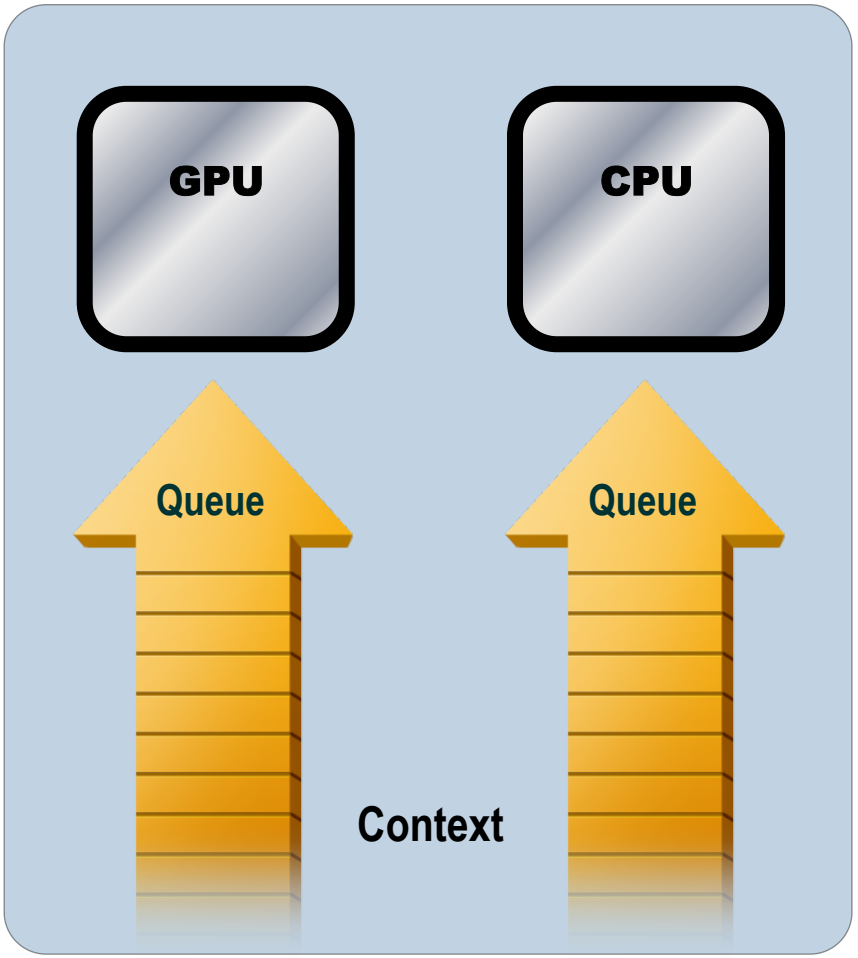
```
context = clCreateContext(firstPlatformId, 1,
                        &device_id, NULL, NULL, &err);
```

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

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

Command-Queues

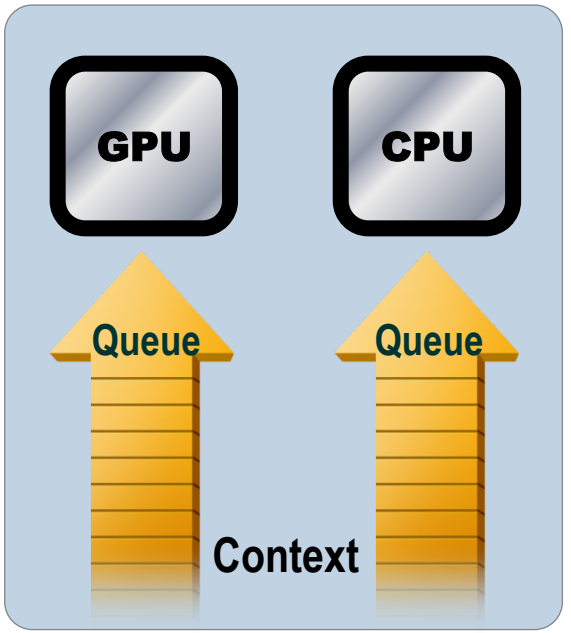
- Commands include:
  - Kernel executions
  - Memory object management
  - Synchronization
- The only way to submit commands to a device is through a command-queue.
- Each command-queue points to a single device within a context.
- Multiple command-queues can feed a single device.
  - Used to define 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.

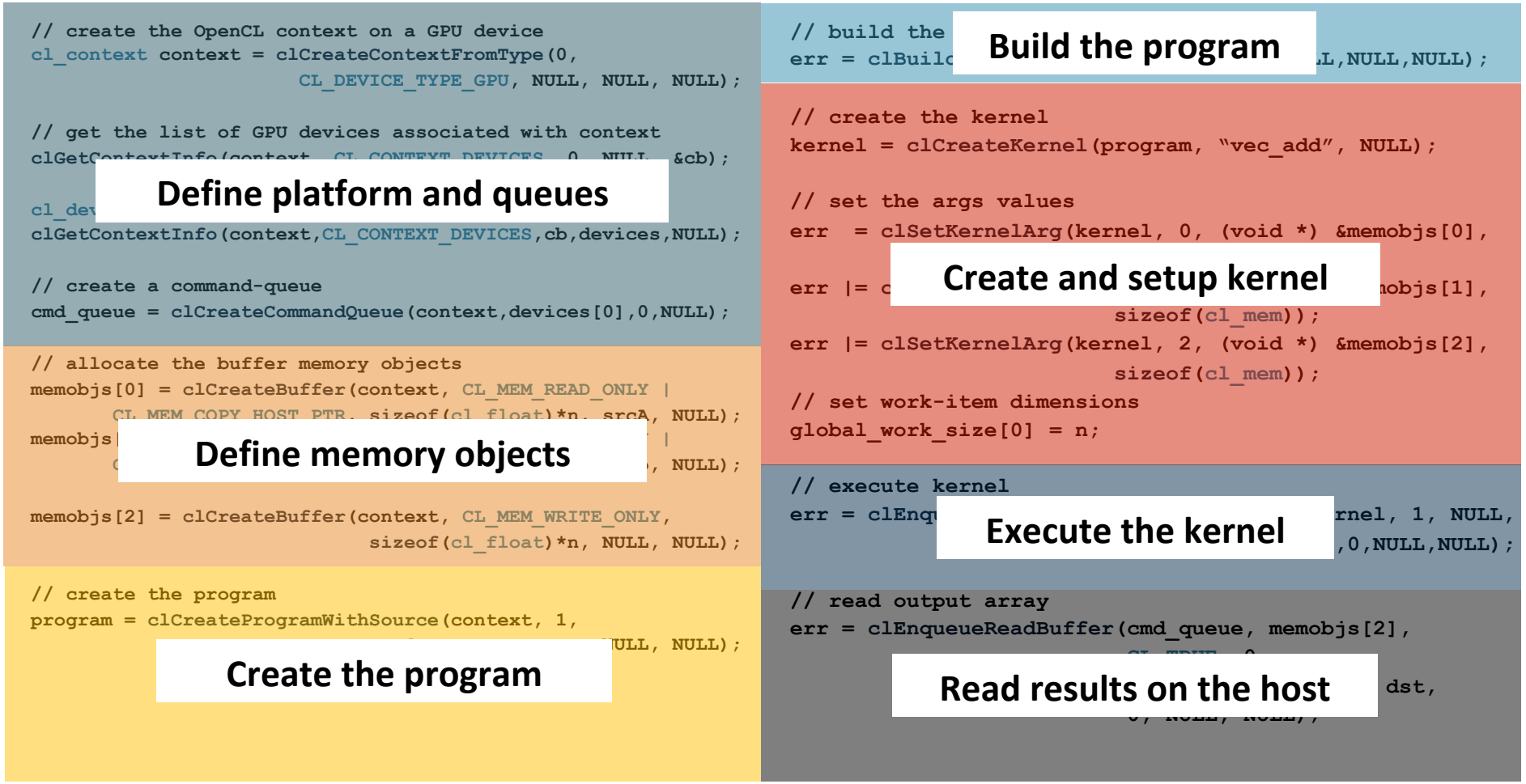
```
err = clEnqueueReadBuffer(commands, d_c, CL_TRUE,
                sizeof(float)*count, h_c, 0, NULL, NULL);
```

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



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

Example 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

Lecture 4

OVERVIEW OF OPENCL APIS

Host programs can be “ugly”

- OpenCL’s goal is extreme portability, so it exposes ***everything***
  - (i.e. it is quite verbose!).
- But most of the host code is the same from one application to the next – the re-use makes the verbosity a non-issue.
- You can package common API combinations into functions or even C++ or Python classes to make the reuse more convenient.

The Python Interface

- A python library by Andreas Klockner from University of Illinois at Urbana-Champaign
- This interface is dramatically easier to work with[1]
- Key features:
  - Helper functions to choose platform/device at runtime
  - getInfo() methods are class attributes – no need to call the method itself
  - Call a kernel as a method
  - Multi-line strings – no need to escape new lines!

[1] not just for python programmers…

Setting up the host program

- Import the pyopencl library

```
import pyopencl as cl
```

- Import numpy to use arrays etc.

```
import numpy
```

- Some of the examples use a helper library to print out some information

```
import deviceinfo
```

```
N = 1024
# create context, queue and program
context = cl.create_some_context()
queue = cl.CommandQueue(context)
kernelsource = open('vadd.cl').read()
program = cl.Program(context, kernelsource).build()

# create host arrays
h_a = numpy.random.rand(N).astype(float32)
h_b = numpy.random.rand(N).astype(float32)
h_c = numpy.empty(N).astype(float32)

# create device buffers
mf = cl.mem_flags
d_a = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_a)
d_b = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_b)
d_c = cl.Buffer(context, mf.WRITE_ONLY, h_c.nbytes)

# run kernel
vadd = program.vadd
vadd.set_scalar_arg_dtypes([None, None, None, numpy.uint32])
vadd(queue, h_a.shape, None, d_a, d_b, d_c, N)

# return results
cl.enqueue_copy(queue, h_c, d_c)
```

Example 3: Running the Vadd kernel (C++ / Python)

- Goal:
  - To learn the C++and/or Python interface to OpenCL's API
- Procedure:
  - Examine the provided program. They will run a simple kernel to add two vectors together
  - Look at the host code and identify the API calls in the host code. Note how some of the API calls in OpenCL map onto C++/Python constructs
  - Compare the original C with the C++/Python versions
  - Look at the simplicity of the common API calls
- Expected output:
  - A message verifying that the vector addition completed successfully

Example 4: Chaining vector add kernels (C++ / Python)

- Goal:
  - To verify that you understand manipulating kernel invocations and buffers in OpenCL
- Procedure:
  - Start with a VADD program in C++ or Python
  - Add additional buffer objects and assign them to vectors defined on the host (see the provided vadd programs for examples of how to do this)
  - Chain vadds … e.g. C=A+B; D=C+E; F=D+G.
  - Read back the final result and verify that it is correct
  - Compare the complexity of your host code to C
- Expected output:
  - A message to standard output verifying that the chain of vector additions produced the correct result

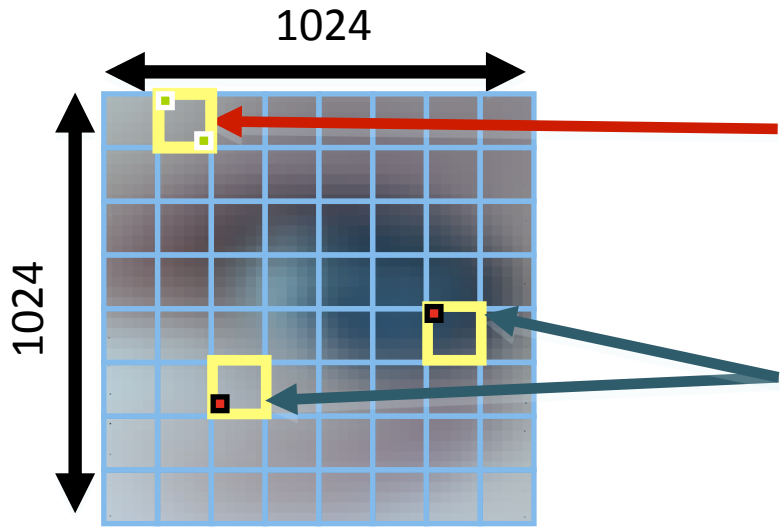
(Sample solution is for C = A + B; D = C + E; F = D + G; return F)

Lecture 7

SYNCHRONIZATION IN OPENCL

Consider N-dimensional domain of work-items

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



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

Cannot synchronize between work-groups within a kernel

Synchronization: when multiple units of execution (e.g. work-items) are brought to a known point in their execution. Most common example is a barrier … i.e. all units of execution "in scope" arrive at the barrier before any proceed.

Work-Item Synchronization

Ensure correct order of memory operations to local or global memory (with flushes or queuing a memory fence)

- Within a work-group

```
void barrier()
```

  - Takes optional flags
    CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE
  - A work-item that encounters a barrier() will wait until ALL work-items in its work-group reach the barrier()
  - Corollary: If a barrier() is inside a branch, then the branch **must** be taken by either:
    - ALL work-items in the work-group, OR
    - NO work-item in the work-group

- Across work-groups
  - No guarantees as to where and when a particular work-group will be executed relative to another work-group
  - Cannot exchange data, or have barrier-like synchronization between two different work-groups! (Critical issue!)
  - Only solution: finish the kernel and start another

Where might we need synchronization?

- Consider a reduction … reduce a set of numbers to a single value
  - E.g. find sum of all elements in an array
- Sequential code

```
int reduce(int Ndim, int *A)
{
  int sum = 0;
  for (int i = 0; i < Ndim; i++)
    sum += A[i];
  return sum;
}
```

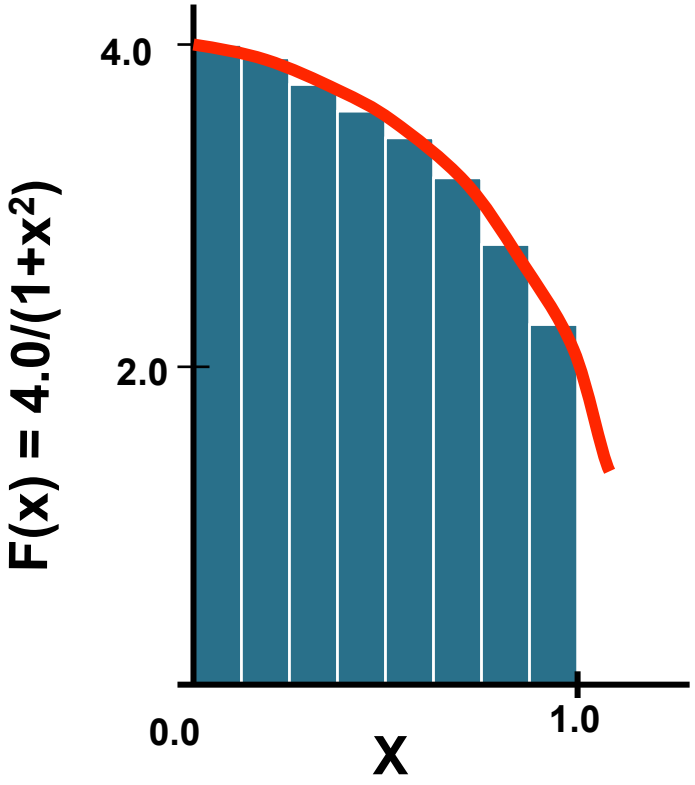
Simple parallel reduction

- A reduction can be carried out in three steps:
  1. Each work-item sums its private values into a local array indexed by the work-item's local id
  2. When all the work-items have finished, one work-item sums the local array into an element of a global array (indexed by work-group id).
  3. When all work-groups have finished the kernel execution, the global array is summed on the host.

- Note: this is a simple reduction that is straightforward to implement. More efficient reductions do the work-group sums in parallel on the device rather than on the host. These more scalable reductions are considerably more complicated to implement.

A simple program that uses a reduction

Numerical Integration



Mathematically, we know that we can approximate the integral as a sum of rectangles.

Each rectangle has width and height at the middle of interval.

Numerical integration source code

The serial Pi program

```
static long num_steps = 100000;
double step;
void main()
{
  int i; double x, pi, sum = 0.0;

  step = 1.0/(double) num_steps;

  for (i = 0; i < num_steps; i++) {
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

OpenCL kernel for calculating π

```
int num_wrk_items
                    = get_local_size(0);
int local_id        = get_local_id(0);
int group_id        = get_group_id(0);

float x, accum      = 0.0f;
int i, istart, iend;

istart = (group_id * num_wrk_items
                    + local_id) * niters;
iend   = istart+niters;
for(i= istart; i<iend; i++) {
      x = (i+0.5f)*step_size;
      accum += 4.0f/(1.0f+x*x);
}
```

```
local_sums[local_id] = accum;

barrier(CLK_LOCAL_MEM_FENCE);

float sum;

if (local_id == 0) {
      // executed by one work_item
      sum = 0.0f;

      for (i=0; i<num_wrk_items; i++)
            sum += local_sums[i];

      partial_sums[group_id] = sum;
}
```

Example 9: The Pi program

- Goal:
  - To understand synchronization between work-items in the OpenCL C kernel programming language
- Procedure:
  - Start with the provided serial program to estimate Pi through numerical integration
  - Write a kernel and host program to compute the numerical integral using OpenCL
  - Note: You will need to implement a reduction
- Expected output:
  - Output result plus an estimate of the error in the result
  - Report the runtime

Hint: you will want each work-item to do many iterations of the loop, i.e. don't create one work-item per loop iteration. To do so would make the reduction so costly that performance would be terrible.

Appendix E

PYTHON FOR C PROGRAMMERS

Python 101

- Python is an interpreted language, and so doesn't need to be compiled
- Python is often used as a language to glue other parts of your application together – with OpenCL this is great as the host code is fast to write and the heavy computation is done on your accelerator
- Run your code as:
  - python file.py
- No curly braces – indent consistently to define blocks of code
- Print to stdout with print – it will try it's best to format variables:

```
print 'a =', a, 'and b =', b
```

Comments, variables and includes

- A comment is prefixed with the hash

```
# this is a comment
```

- Initilize variables as you go – no need for a type

```
N = 1024
x = 5.23
my_string = 'hello world'
```

- Use single or double quotes for strings

```
'this is the same'
"as this"
"no need to escape 'opposite' quotes!"
```

- Also use three quotes ''' or """ for multiline strings without escaping anything!
- Include additional modules and libraries with

```
import sys
```

Conditionals

```
if n == 1:
  print 'n was 1'
elif n == 2 or n == 3:
  print 'n was 2 or 3'
else:
  print 'n was', n
```

Loops

```
# loop from 0 to 1023
for i in range(1024):
        print i

# iterate through an array
for x in my_array:
        x += 1

# same as the first one
while i < 1024:
        print i
        i += 1
```

Functions and classes

- Define a function with the def keyword

```
def func(arg):
```

- You don't specify the types or return arguments
  - you just return what you like
- Define a class with the class keyword

```
class name:
```

- Classes contain function definitions and variables
  - These are both called attributes

More about classes

- There is a lot more about classes e.g. inheritance
- Python is an object-oriented language
- A small example from the python tutorial:

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
```

- Initilize an instance of the class with:

```
x = Complex(3.0, -4.5)
```

Python has functional programming elements

- Filter

  `filter(function, sequence)`

  – Returns a list from sequence which function returns true
- Map

  `map(function, sequence)`

  – Applies the function to each element in the sequence
- Reduce

  `reduce(function, sequence)`

  – Applies binary function with first two in sequence, then with the result with third, etc.

Python has functional programming elements

- List comprehensions

```
squares = [x*x for x in range(10)]
# squares = [0, 1, 4, 9, 16, etc]
```

- Zip

```
zip(list1, list2)
```

  - Creates a list of tuples, where the ith tuple consists of the ith elements of each list

- Generators
  - Lazy generation of lists
  - Either:
    - Replace [] with () in list comprehensions to use as expression, i.e. to pass to another function
    - Use the yield keyword instead of return in a function which builds and returns a list

Further information:

- There is lots more to python, this is just a flavor of the language to help you understand the syntax in this course
- The official python tutorial is much more complete:
  - http://docs.python.org/2/tutorial/index.html
- The python docs are really good too
  - http://docs.python.org/2/library/index.html