Lec14: OpenCL

Modified for UH COSC4397
Spring 2025

Hands On OpenCL

Created by
Simon McIntosh-Smith
and Tom Deakin

University of BRISTOL

OpenCL

KITE
Khronos Initiative for Training & Education

Includes contributions from:
Timothy G. Mattson (Intel) and Benedict Gaster (Qualcomm)

V 1.2 – Nov 2014

Agenda

Lectures	Exercises
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
Appendices	

Course materials

In addition to these slides, C++ API header files, a set of exercises, and solutions, it is useful to have:

OpenCL 1.1 Reference Card

This card will help you keep track of the API as you do the exercises:

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

The v1.1 spec is also very readable and recommended to have on-hand:

https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

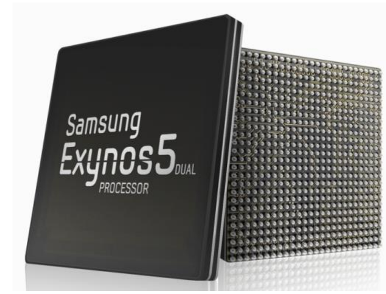
Lecture 2

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:

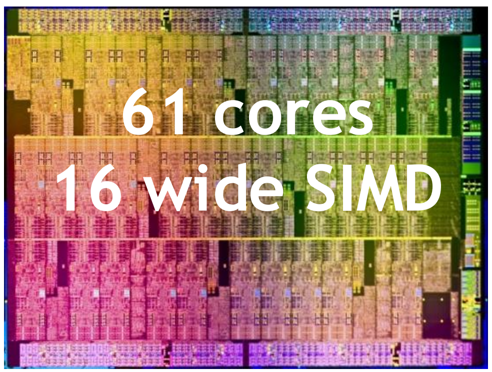
- Dual core ARM A15 1.7GHz, Mali T604 GPU

E.g. Intel XXX with IRIS

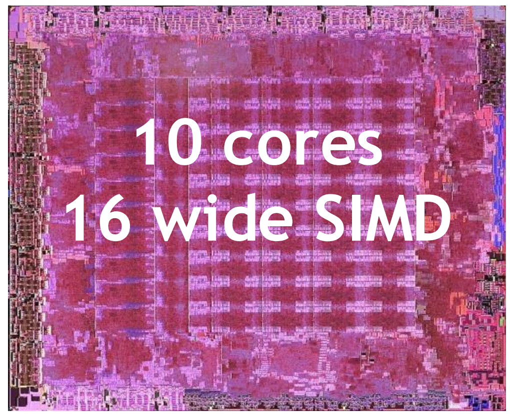
OpenCL lets Programmers write a single <u>portable</u> program that uses <u>ALL</u> resources in the heterogeneous platform

Microprocessor trends

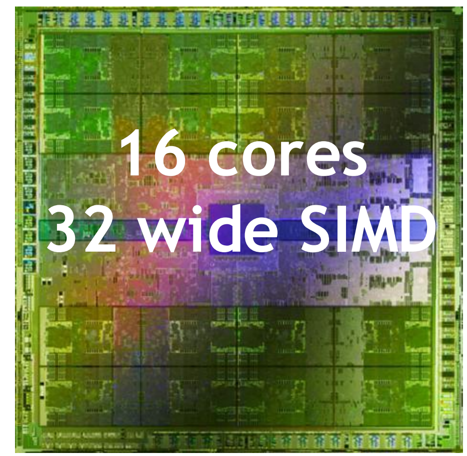
Individual processors have many (possibly heterogeneous) cores.



Intel® Xeon Phi™ coprocessor



ATI™ RV770

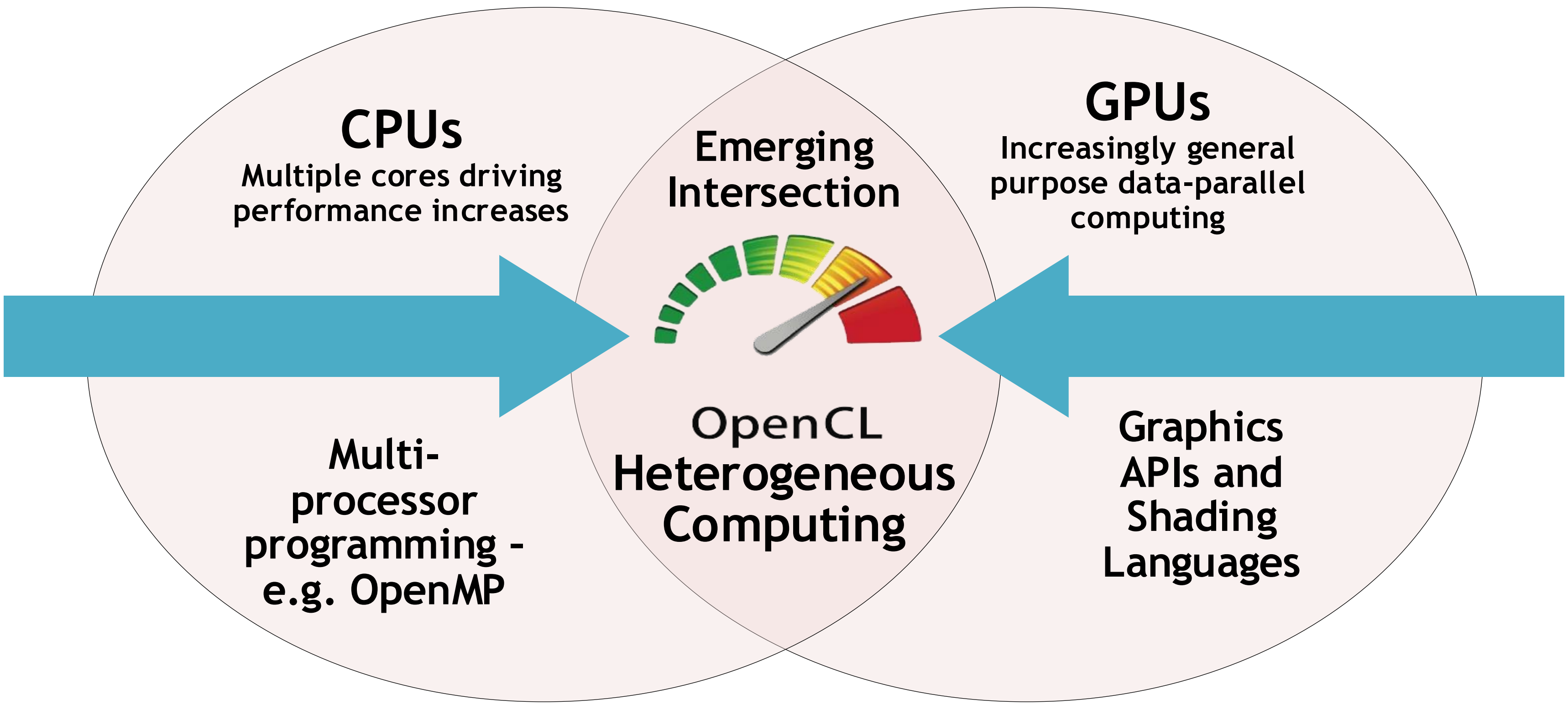


NVIDIA® Tesla® C2090

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

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
- Nokia
- IBM
- Sony
- Qualcomm
- Imagination
- TI
- + many more

OpenCL

Third party names are the property of their owners.

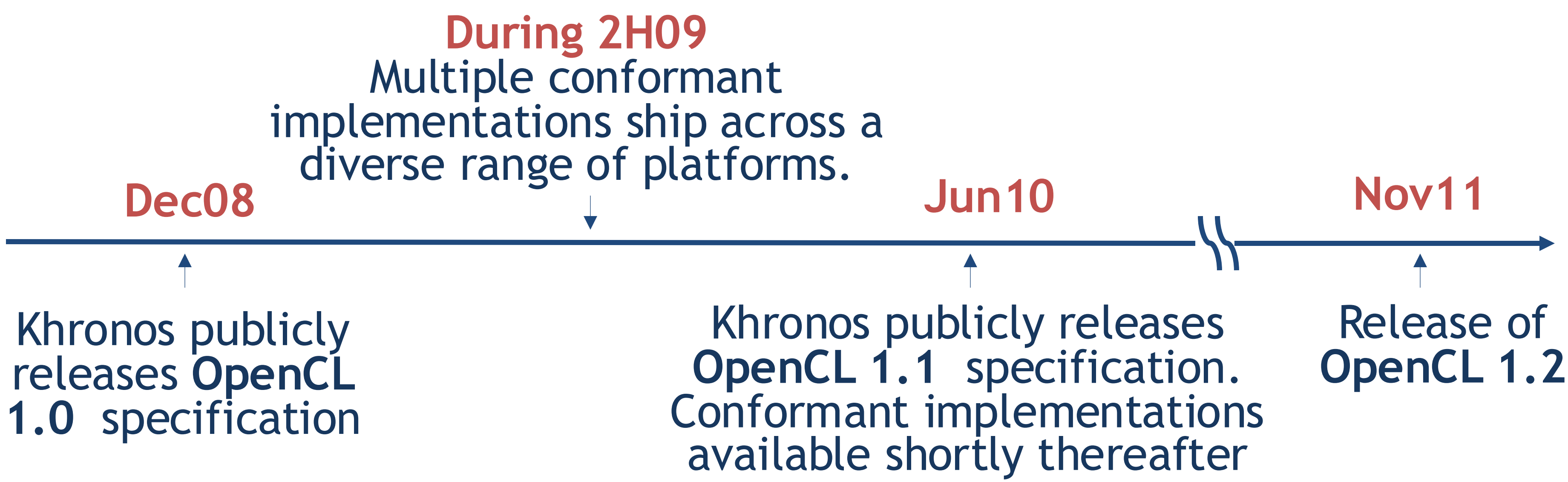
OpenCL Working Group within Khronos

- 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.

Third party names are the property of their owners.

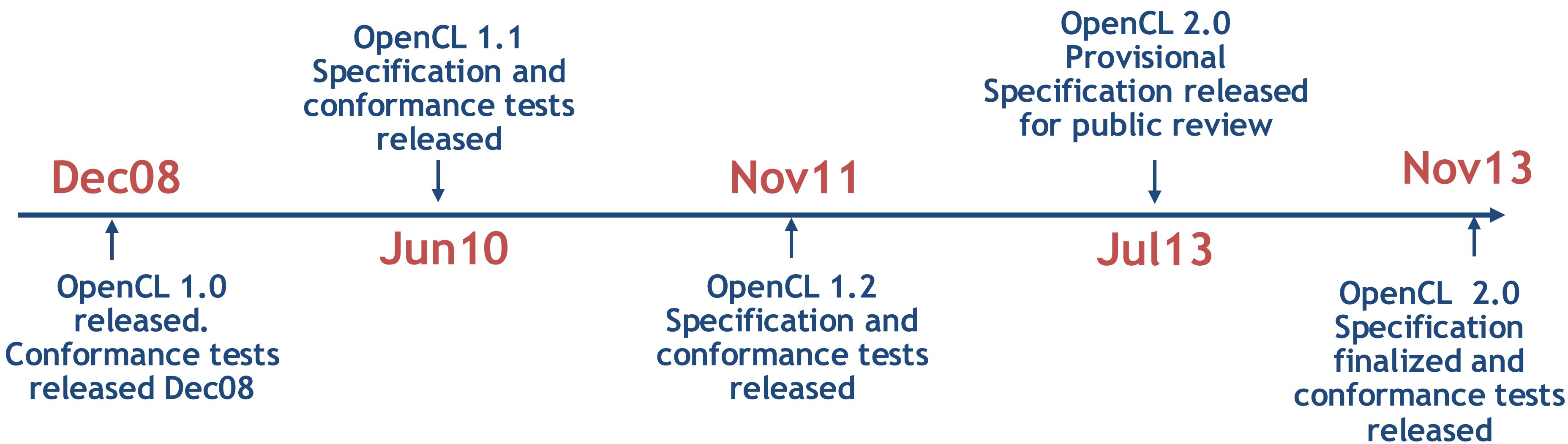
OpenCL Timeline

- 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



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

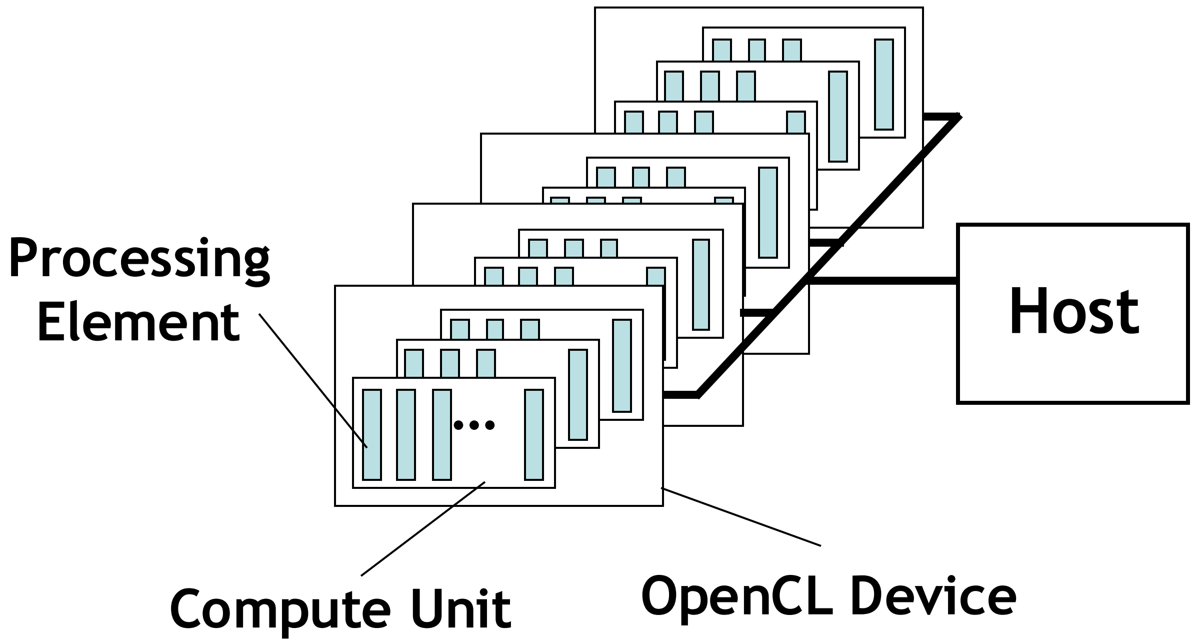


OpenCL: From cell phone to supercomputer

- 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***

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

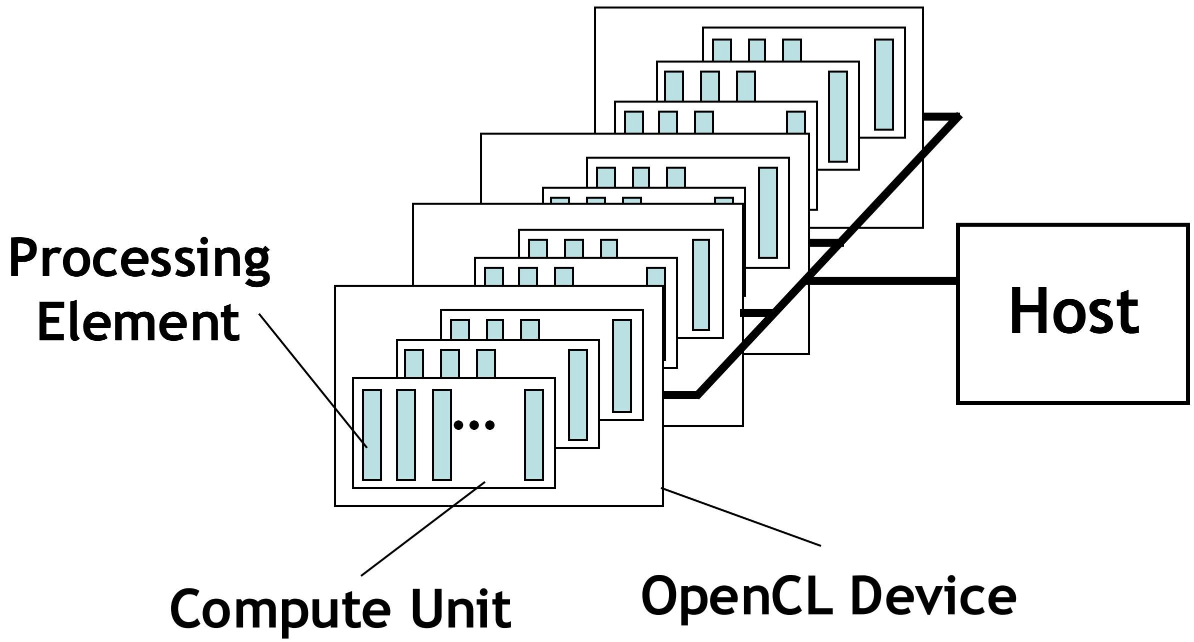
Exercise 1: Platform Information

- 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.

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

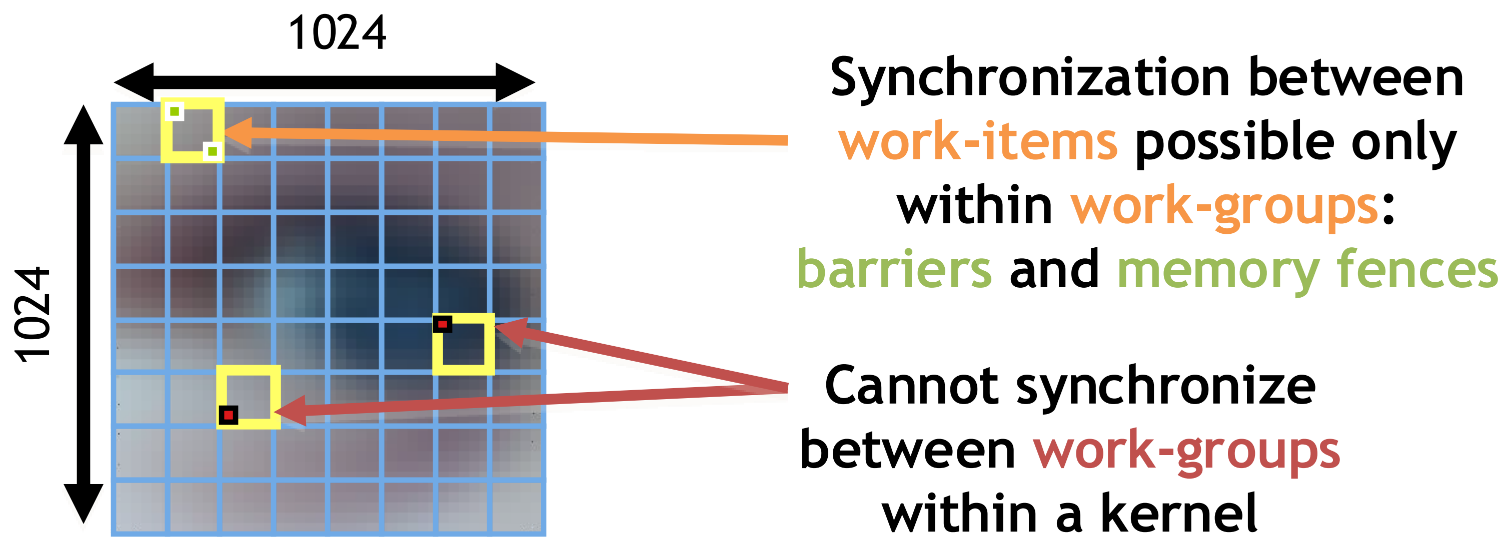
```
__kernel void
mul(__global const float *a,
    __global const float *b,
    __global       float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
// many instances of the kernel,
// called work-items, execute
// in parallel
```

Analogies to CUDA

OpenCL	**CUDA**	**Purpose**
get_global_id(0)	blockIdx.x * blockDim.x + threadIdx.x	Global thread ID in 1D grid.
get_local_id(0)	threadIdx.x	Local ID within a work-group (block).
get_group_id(0)	blockIdx.x	Work-group (block) ID.
get_global_size(0)	gridDim.x * blockDim.x	Total work-items in dimension 0.

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)

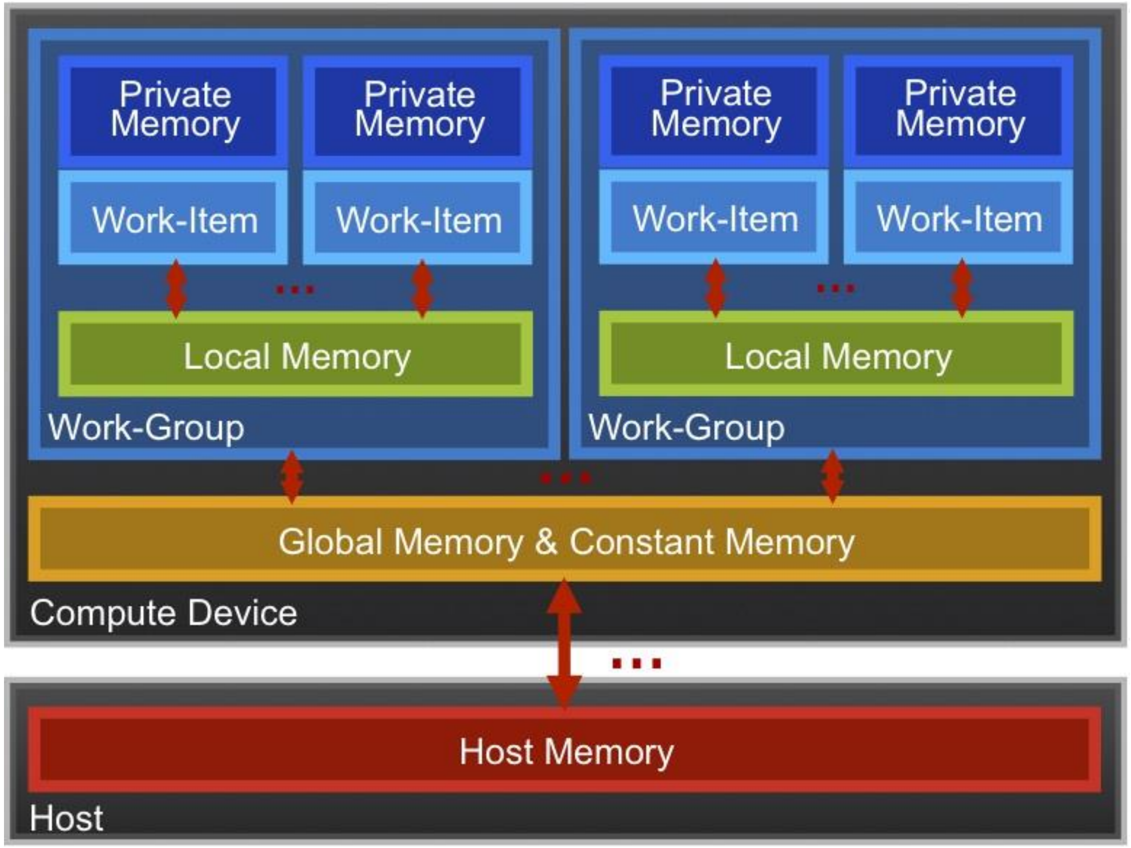
- The problem we want to compute should have some **dimensionality**;
  - For example, compute a kernel on all points in a cube
- When we execute the kernel we specify **up to 3 dimensions**
- We also **specify the total problem size** in each dimension – this is called the **global** size
- We associate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

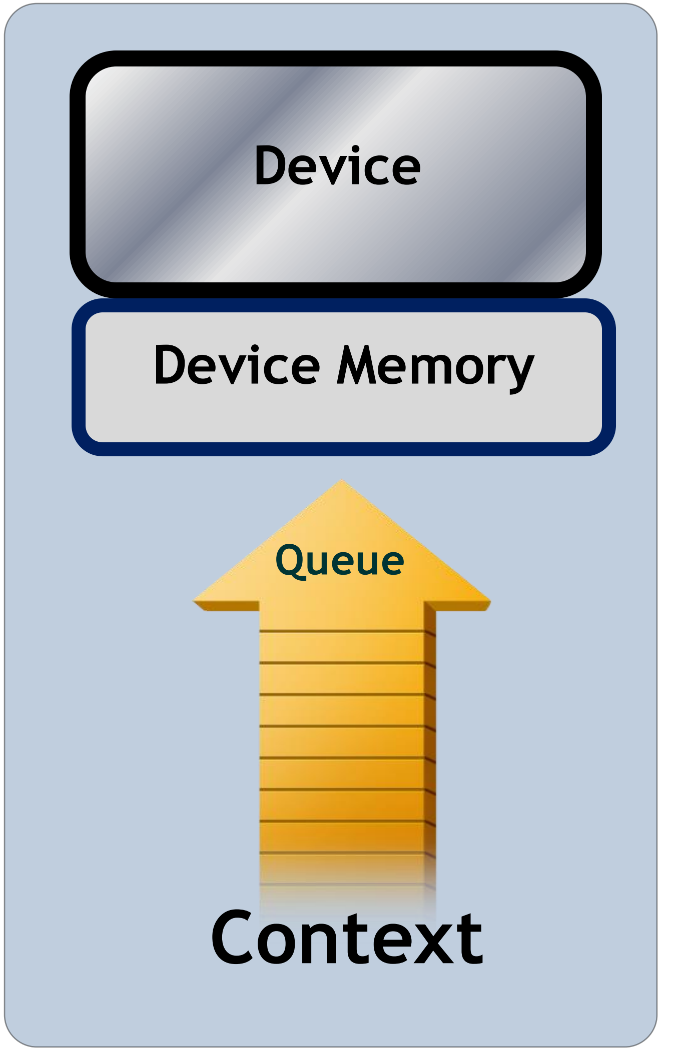
- *Private Memory*
  - Per work-item
- *Local Memory*
  - Shared within a work-group
- *Global Memory /Constant Memory*
  - Visible to all work-groups
- *Host memory*
  - On the CPU



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

Context and Command-Queues

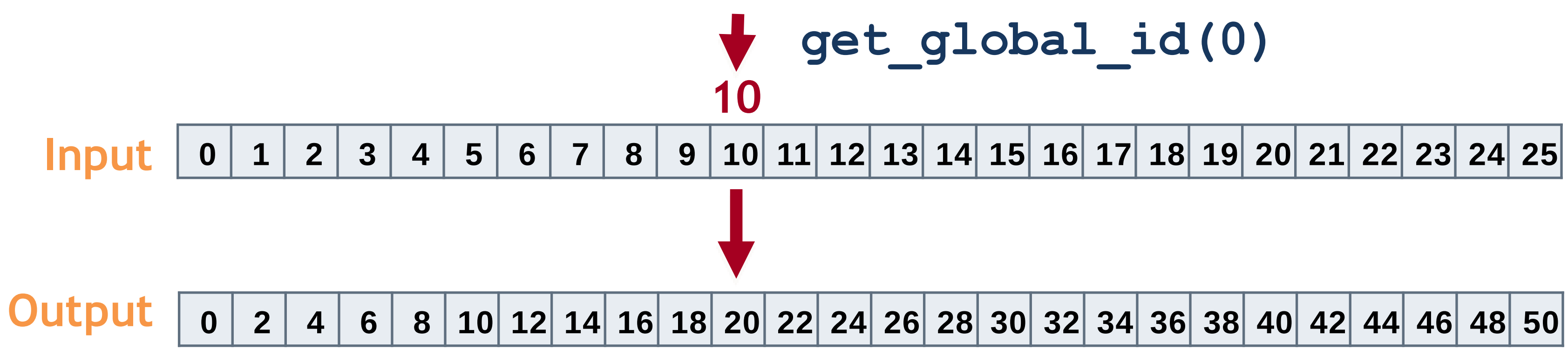
- ***Context***:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization,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

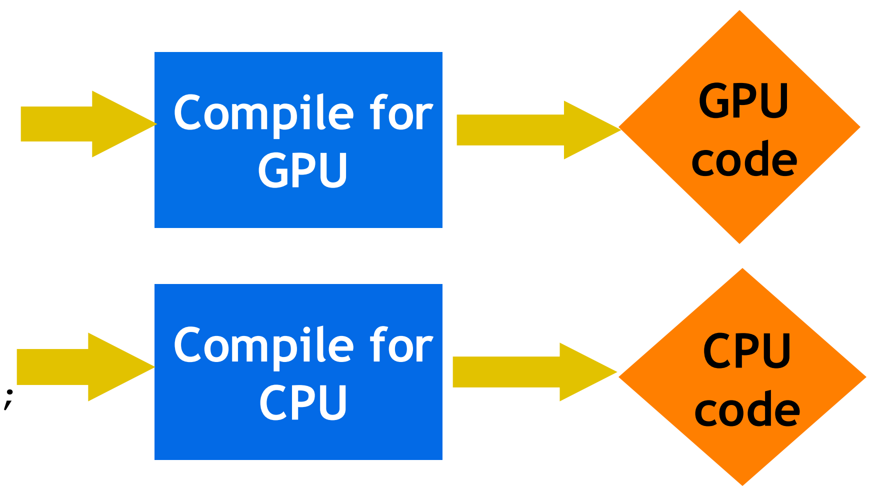


Building Program Objects

- 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

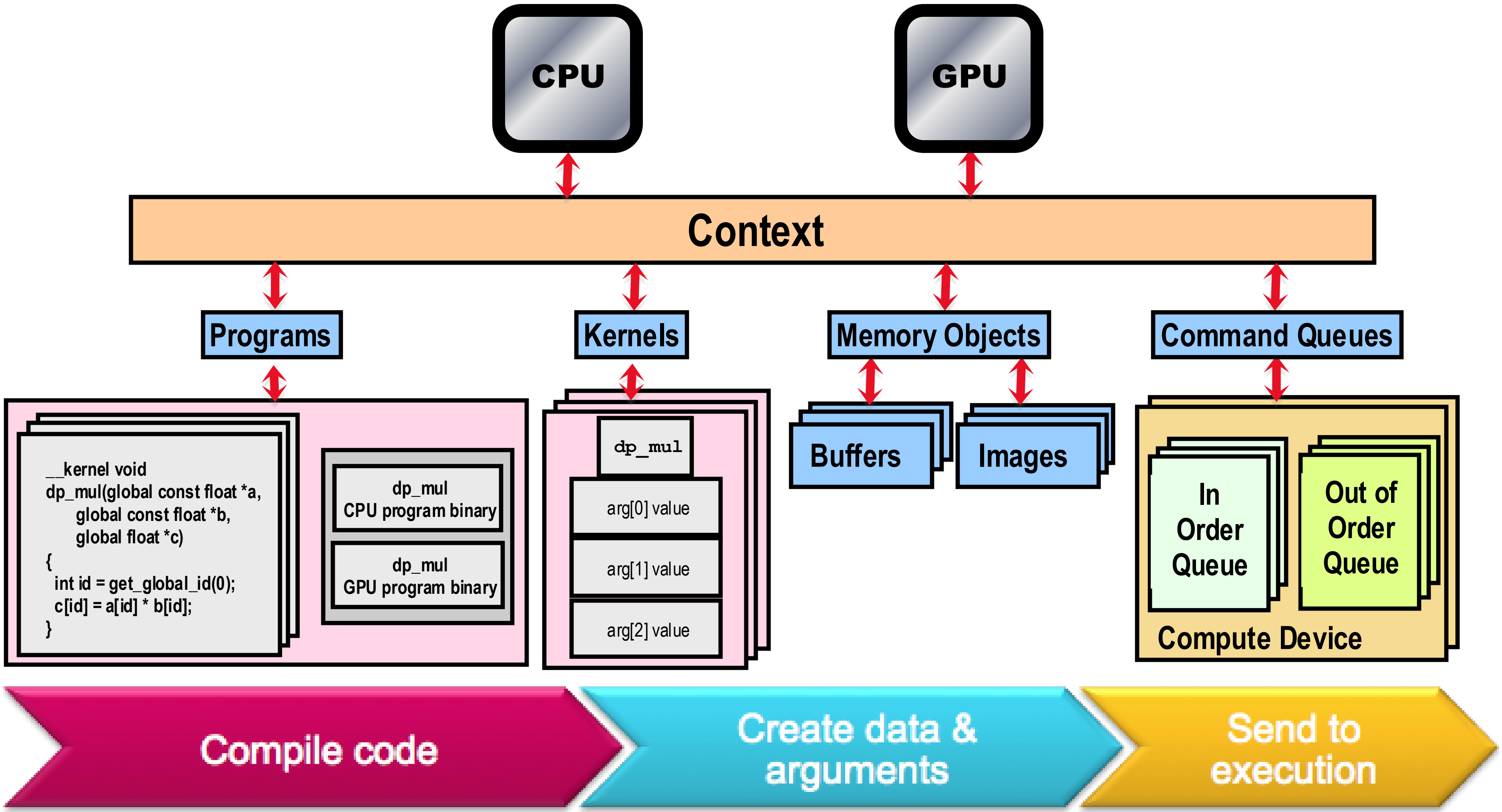
```
__kernel void vadd(__global const float *a,
                   __global const float *b,
                   __global       float *c)
{
    int gid = get_global_id(0);
    c[gid]  = a[gid] + b[gid];
}
```

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:

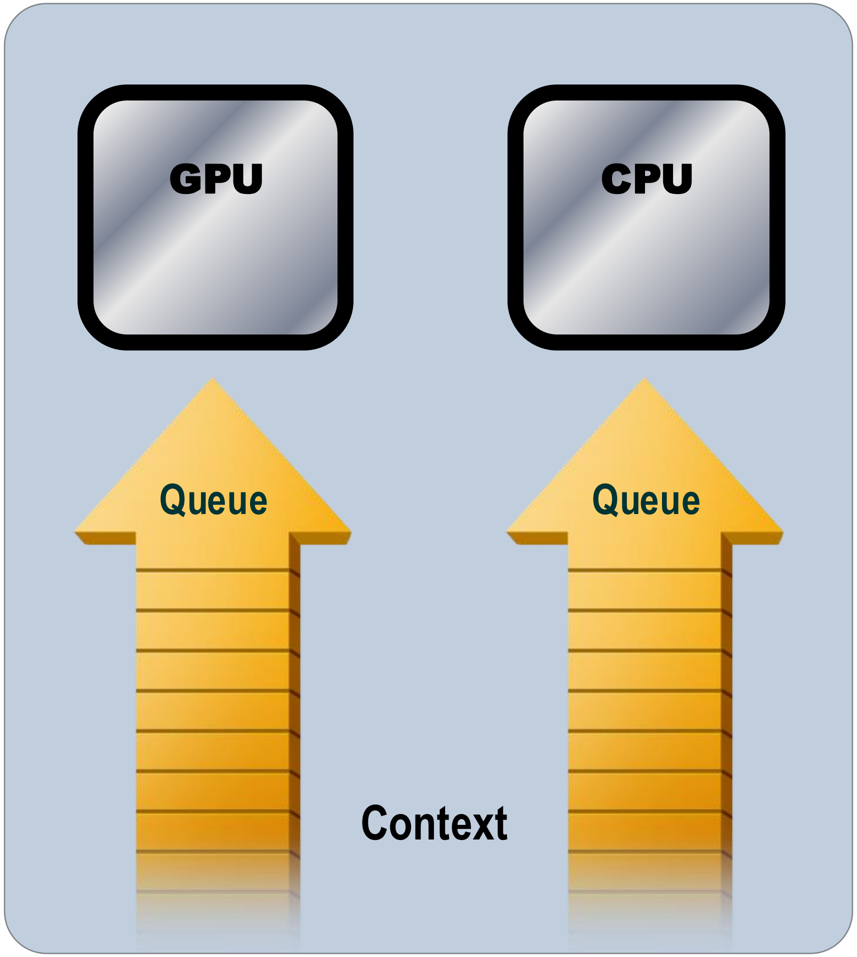
```
context = clCreateContext(firstPlatformId, 1,
                          &device_id, NULL, NULL, &err);
```

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

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

Command-Queues

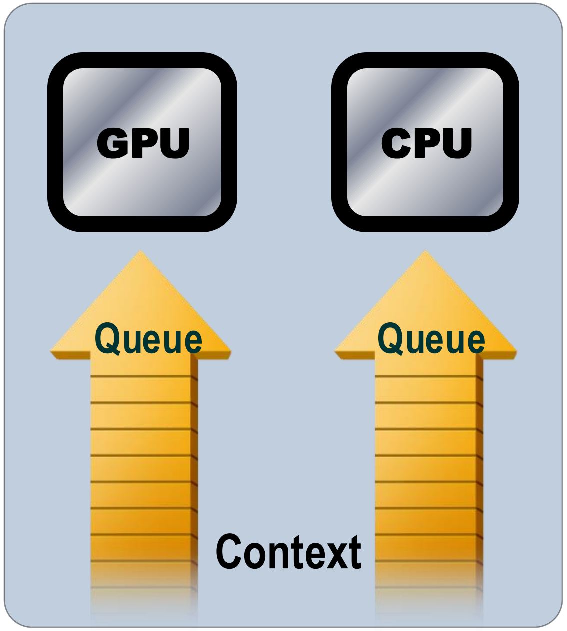
- Commands include:
  - Kernel executions
  - Memory object management
  - Synchronization
- The only way to submit commands to a device is through a command-queue.
- Each command-queue points to a single device within a context.
- Multiple command-queues can feed a single device.
  - Used to define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

Create the program

```
// build the
err = clBuil                         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                                    obj s[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],

                                    dst,
```

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

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 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>        // For C style
#include <iostream>      // For C++ style IO
#include <vector>        // For C++ vector types
```

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

C++ interface: The vadd host program

```
std::vector<float>
  h_a(N), h_b(N), h_c(N);
// initialize host vectors…

cl::Buffer d_a, d_b, d_c;

cl::Context  context(
   CL_DEVICE_TYPE_DEFAULT);

cl::CommandQueue
   queue(context);

cl::Program  program(
  context,
  loadprogram("vadd.cl"),
  true);

// Create the kernel functor
cl::make_kernel<cl::Buffer,
 cl::Buffer, cl::Buffer, int>
 vadd(program, "vadd");
```

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

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

d_b = cl::Buffer(context,
        h_b.begin(), h_b.end(), true);

d_c = cl::Buffer(context,
        CL_MEM_READ_WRITE,
        sizeof(float) * LENGTH);

// Enqueue the kernel
vadd(cl::EnqueueArgs(
                queue,
                cl::NDRange(count)),
        d_a, d_b, d_c, count);

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

The C++ Buffer Constructor

- This is the API definition:
  - Buffer(startIterator, endIterator, bool readOnly, bool useHostPtr)
- The readOnly boolean specifies whether the memory is CL_MEM_READ_ONLY (true) or CL_MEM_READ_WRITE (false)
  - You must specify a true or false here
- The useHostPtr boolean is default false
  - Therefore the array defined by the iterators is implicitly copied into device memory
  - If you specify true:
    - The memory specified by the iterators must be contiguous
    - The context uses the pointer to the host memory, which becomes device accessible - this is the same as CL_MEM_USE_HOST_PTR
    - The array is not copied to device memory
- We can also specify a context to use as the first argument in this API call

The C++ Buffer Constructor

- When using the buffer constructor which uses C++ vector iterators, remember:
  - This is a blocking call
  - The constructor will enqueue a copy to the first Device in the context (when useHostPtr == false)
  - The OpenCL runtime will automatically ensure the buffer is copied across to the actual device you enqueue a kernel on later if you enqueue the kernel on a different device within this context

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
```

```python
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)
```

Exercise 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

Exercise 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)

Review

A HOST VIEW OF WORKING WITH KERNELS

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 (toy codes)
- Or the kernel code can be loaded from a file (real codes)

- 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 function within the program – allows us to ‘call’ the kernel to enqueue it

```
cl::make_kernel
<cl::Buffer, cl::Buffer, cl::Buffer, int>
  vadd(program, "vadd");
```

Create a kernel (advanced)

- If you want to query information about a kernel, you will need to create a kernel object too:

If we set the local dimension ourselves or accept the OpenCL runtime's, we don't need this step

```
cl::Kernel ko_vadd(program, "vadd");
```

- Get the default size of local dimension (i.e. the size of a Work-Group)

```
::size_t local = ko_vadd.getWorkGroupInfo
  <CL_KERNEL_WORK_GROUP_SIZE>(cl::Device::getDefault());
```

We can use any work-group-info parameter from table 5.15 in the OpenCL 1.1 specification. The function will return the appropriate type.

Associate with args and enqueue kernel

- Enqueue the kernel for execution with buffer objects **d_a**, **d_b** and **d_c** and their length, **count**:

We can include any arguments from the clEnqueueNDRangeKernel function including Event wait lists (to be discussed later) and the command queue (optional)

```
vadd(cl::EnqueueArgs(
     queue, cl::NDRange(count), cl::NDRange(local)),
     d_a, d_b, d_c, count);
```

Working with Kernels (Python)

- Kernel source string can be defined with three quote marks – no need to escape new lines:

```
source = '''
   __kernel void func() {}
'''
```

- Or in a file and loaded at runtime:

```
source = open('file.cl').read()
```

- The program object is created and built:

```
prg =
pyopencl.Program(context,source).build()
```

Working with Kernels (Python)

- Kernels can be called as a method of the built program object; as in

```
program.kernel(q, t, l, a)
```

- The basic arguments to this call are:
  1. `q` is the Command Queue
  2. `t` is the Global size as a tuple: (x,), (x,y), or (x,y,z)
  3. `l` is the Local size as a tuple or None
  4. `a` is the list of arguments to pass to the kernel
     - Scalars must be type cast to numpy types; i.e. numpy.uint32(var), numpy.float32(var)

Working with Kernels (Python)

- Calling the kernel from within the program object calls clCreateKernel() from the C API
  - I.e. calling program.kernel() creates the kernel object every time, which is unnecessary

- Can pull out the kernel to stop this:

```
kernel = program.kernel
```

- Specify the scalar arguments on the kernel object to save casting in the kernel execution call:

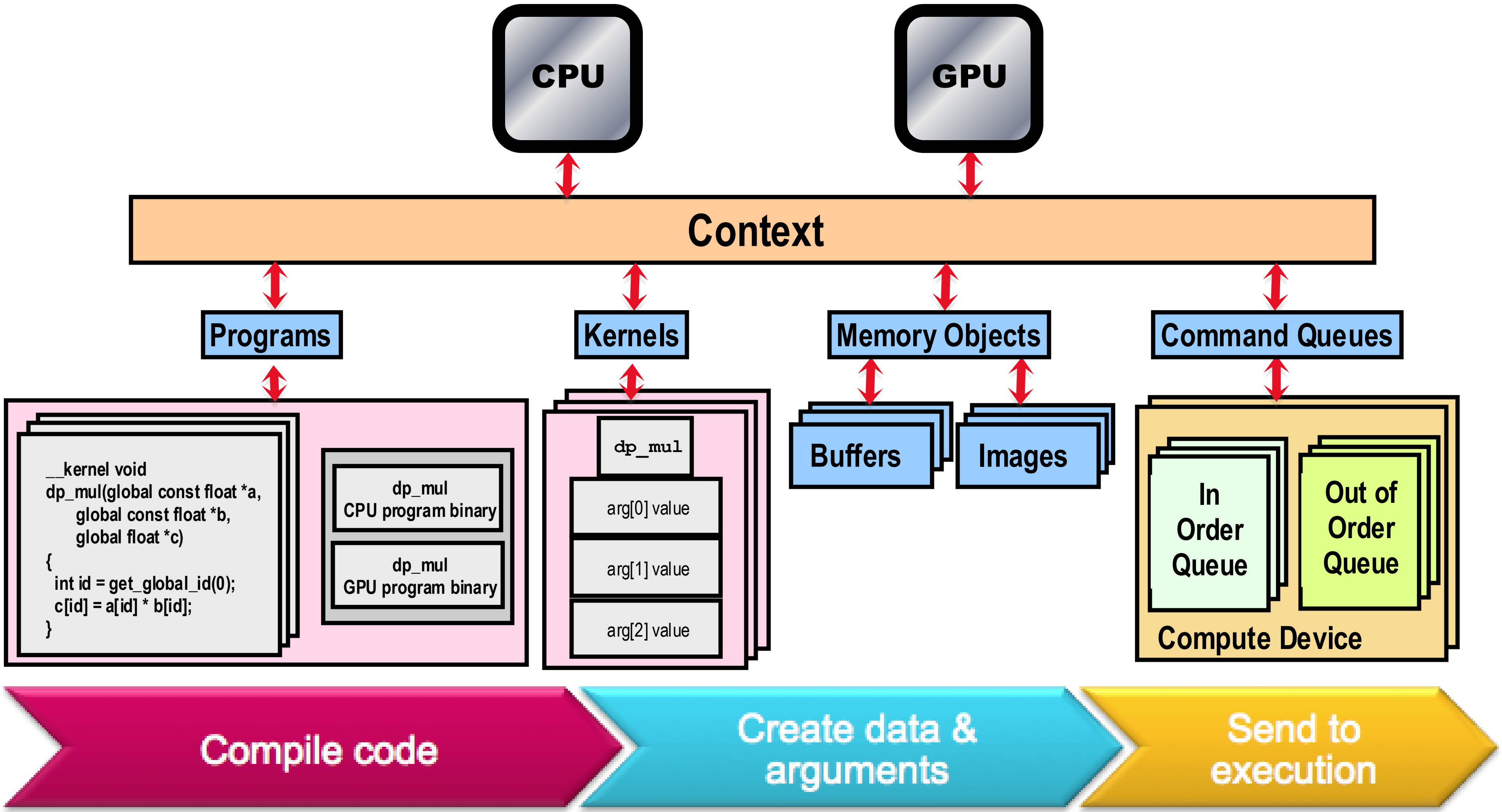
```
kernel.set_scalar_arg_dtypes([list, of, arg, types])
```

  - Buffer and local memory arguments should be set as None
  - Scalar arguments could be numpy.float32, numpy.uint32, etc.

Exercise 5: The D = A + B + C problem

- Goal:
  - To verify that you understand how to control the argument definitions for a kernel
  - To verify that you understand the host/kernel interface
- Procedure:
  - Start with a VADD program.
  - Modify the kernel so it adds three vectors together
  - Modify the host code to define three vectors and associate them with relevant kernel arguments
  - Read back the final result and verify that it is correct
- Expected output:
  - Test your result and verify that it is correct. Print a message to that effect on the screen

We have now covered the basic platform runtime APIs in OpenCL



Lecture 5

INTRODUCTION TO OPENCL KERNEL PROGRAMMING

OpenCL C for Compute Kernels

- Derived from ISO C99
  - A few ***restrictions***: no recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported (#include etc.)
- Built-in data types
  - Scalar and vector data types, pointers
  - Data-type conversion functions:
    - convert_type<_sat><_roundingmode>
  - Image types:
    - image2d_t, image3d_t and sampler_t

OpenCL C for Compute Kernels

- Built-in functions — ***mandatory***
  - Work-Item functions, math.h, read and write image
  - Relational, geometric functions, synchronization functions
  - printf (v1.2 only, so not currently for NVIDIA GPUs)
- Built-in functions — ***optional*** (called "extensions")
  - Double precision, atomics to global and local memory
  - Selection of rounding mode, writes to image3d_t surface

OpenCL C Language Highlights

- Function qualifiers
  - __kernel qualifier declares a function as a kernel
    - I.e. makes it visible to host code so it can be enqueued
  - Kernels can call other kernel-side functions
- Address space qualifiers
  - __global, __local, __constant, __private
  - Pointer kernel arguments must be declared with an address space qualifier
- Work-item functions
  - get_work_dim(), get_global_id(), get_local_id(), get_group_id()
- Synchronization functions
  - Barriers - all work-items within a work-group must execute the barrier function before any work-item can continue
  - Memory fences - provides ordering between memory operations

OpenCL C Language Restrictions

- Pointers to functions are ***not*** allowed
- Pointers to pointers allowed ***within*** a kernel, but not as an argument to a kernel invocation
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported (yet!)
- Double types are optional in OpenCL v1.1, but the key word is reserved

  (note: most implementations support double)

Worked example: Linear Algebra

- Definition:
  - The branch of mathematics concerned with the study of vectors, vector spaces, linear transformations and systems of linear equations.
- Example: Consider the following system of linear equations

$$x + 2y + z = 1$$
$$x + 3y + 3z = 2$$
$$x + y + 4z = 6$$

  - This system can be represented in terms of vectors and a matrix as the classic "Ax = b" problem.

$$\begin{pmatrix} 1 & 2 & 1 \\ 1 & 3 & 3 \\ 1 & 1 & 4 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 6 \end{pmatrix}$$

Solving Ax=b

- LU Decomposition:
  - transform a matrix into the product of a lower triangular and upper triangular matrix. It is used to solve a linear system of equations.

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 1 \\ 1 & 3 & 3 \\ 1 & 2 & 4 \end{pmatrix}$$

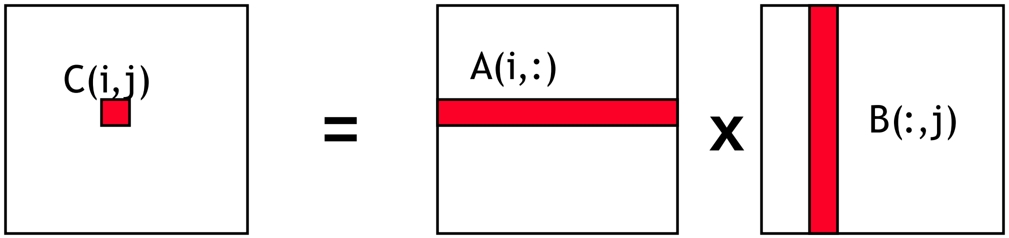
- We solve for x, given a problem Ax=b
  - Ax=b LUx=b
  - $Ux=(L^{-1})b$ $x = (U^{-1})(L^{-1})b$

So we need to be able to do matrix multiplication

Matrix multiplication: sequential code

We calculate C=AB, where all three matrices are NxN

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
                // C(i, j) = sum(over k) A(i,k) * B(k,j)
                C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```



Dot product of a row of A and a column of B for each element of C

Matrix multiplication performance

- Serial C code on CPU (single core).

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A

Device is Intel® Xeon® CPU, E5649 @ 2.53GHz using the gcc compiler.

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
              // C(i, j) = sum(over k) A(i,k) * B(k,j)
              C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

We turn this into an OpenCL kernel!

Matrix multiplication: OpenCL kernel (1/2)

```
__kernel void mat_mul(
 const int N,
__global float *A, __global float *B, __global float *C)
 {
       int i, j, k;
       for (i = 0; i < N; i++) {
         for (j = 0; j < N; j++) {
           // C(i, j) = sum(over k) A(i,k) * B(k,j)
           for (k = 0; k < N; k++) {
             C[i*N+j] += A[i*N+k] * B[k*N+j];
               }
           }
       }
 }
```

Mark as a kernel function and specify memory qualifiers

Matrix multiplication: OpenCL kernel (2/2)

```
__kernel void mat_mul(
  const int N,
  __global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
        for (k = 0; k < N; k++) {
            // C(i, j) = sum(over k) A(i,k) * B(k,j)
            C[i*N+j] += A[i*N+k] * B[k*N+j];
        }

}
```

Remove outer loops and set work-item co-ordinates

Matrix multiplication: OpenCL kernel

```
__kernel void mat_mul(
 const int N,
 __global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
    // C(i, j) = sum(over k) A(i,k) * B(k,j)
    for (k = 0; k < N; k++) {
      C[i*N+j] += A[i*N+k] * B[k*N+j];
    }
}
```

Matrix multiplication: OpenCL kernel improved

Rearrange and use a local scalar for intermediate C element values (a common optimization in Matrix Multiplication functions)

```
__kernel void mmul(
   const int N,
   __global float *A,
   __global float *B,
   __global float *C)
{
  int k;
  int i = get_global_id(0);
  int j = get_global_id(1);
  float tmp = 0.0f;
  for (k = 0; k < N; k++)
   tmp += A[i*N+k]*B[k*N+j];
  }
  C[i*N+j] += tmp;
}
```

Matrix multiplication host program (C++ API)

```
int main(int argc, char *argv[])
{
  std::vector<float> h_A, h_B, h_C; // matrices
  int Mdim, Ndim, Pdim; // A[N][P],B[P][M],C[N][M]
  int i, err;
  int szA, szB                    s in each matrix
  double start_                   timing data
  cl::Program p

  Ndim = Pdim =
  szA = Ndim*P
  szB = Pdim*Mdim;
  szC = Ndim*Mdim;
  h_A   = std::vector<float>(szA);
  h_B   = std::vector<float>(szB);
  h_C   = std::vector<float>(szC);

  initmat(Mdim, Ndim, Pdim, h_A, h_B, h_C);
```

Declare and initialize data

```
  // Compile for first kernel to setup program
  program =                     lSource, true);
  Context co                    AULT);
  cl::Comman
  std::vecto
      contex                    ICES>();
  cl::Device
  std::string s =
      device.getInfo<CL_DEVICE_NAME>();
  std::cout << "\nUsing OpenCL Device "
            << s << "\n";
```

Setup the platform and build program

```
  // Set                                    s,
  // and
  initma                                    ;
  cl::Bu                                    end(), true);
  cl::Bu                                    end(), true);
  cl::Bu
                                  CL_MEM_WRITE_ONLY,
                                  sizeof(float) * szC);
```

Setup buffers and write A and B matrices to the device memory

```
  cl::make_kernel<int, int, int,
                  cl::Buffer, cl::Buffer, cl::Buffer>

  zero_mat(Ndim, Mdim, h_C);
  start_time = wtime();
```

Create the kernel functor

```
  naive(cl::EnqueueArgs(queue,
                        cl::NDRange(Ndim, Mdim)),
        Ndi                          );

  cl::copy                           .end());

  run_time  = wtime() - start_time;
  results(Mdim, Ndim, Pdim, h_C, run_time);
}
```

Run the kernel and collect results

Note: To use the default context/queue/device, skip this section and remove the references to context, queue and device.

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Exercise 6: Matrix Multiplication

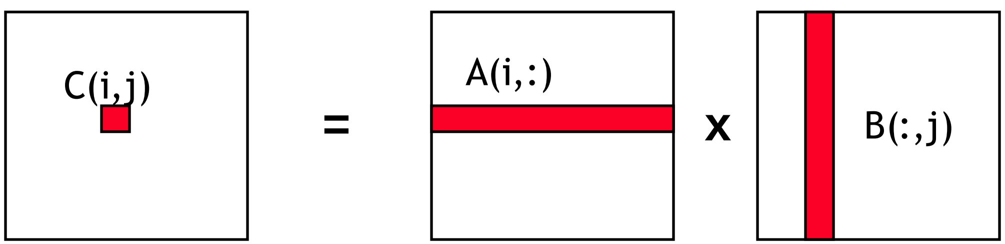
- Goal:
  - To write your first complete OpenCL kernel “from scratch”
  - To multiply a pair of matrices
- Procedure:
  - Start with the provided matrix multiplication OpenCL host program including the function to generate matrices and test results
  - Create a kernel to do the multiplication
  - Modify the provided OpenCL host program to use your kernel
  - Verify the results
- Expected output:
  - A message to standard output verifying that the chain of vector additions produced the correct result
  - Report the runtime and the MFLOPS

Lecture 6

UNDERSTANDING THE OPENCL MEMORY HIERARCHY

Optimizing matrix multiplication

- MM cost determined by FLOPS and memory movement:
  - $2*n^3 = O(n^3)$ FLOPS
  - Operates on $3*n^2 = O(n^2)$ numbers
- To optimize matrix multiplication, we must ensure that for every memory access we execute as many FLOPS as possible.
- Outer product algorithms are faster, but for pedagogical reasons, let's stick to the simple dot-product algorithm.

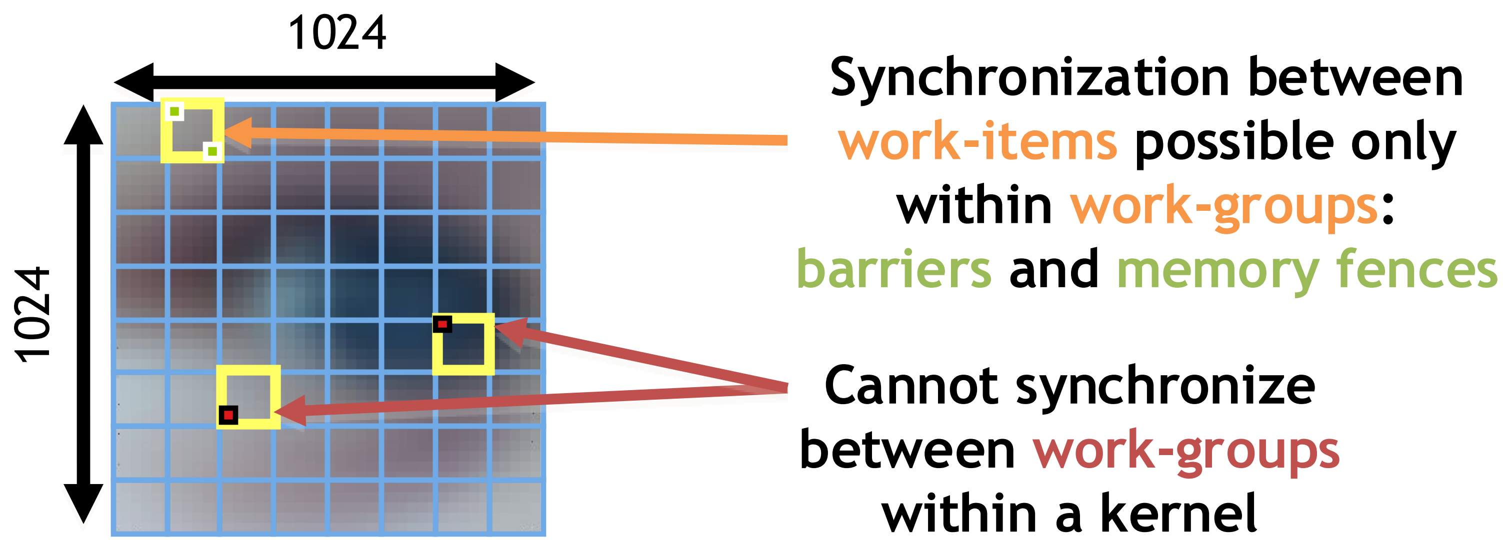


Dot product of a row of A and a column of B for each element of C

- We will work with work-item/work-group sizes and the memory model to optimize matrix multiplication

An N-dimensional domain of work-items

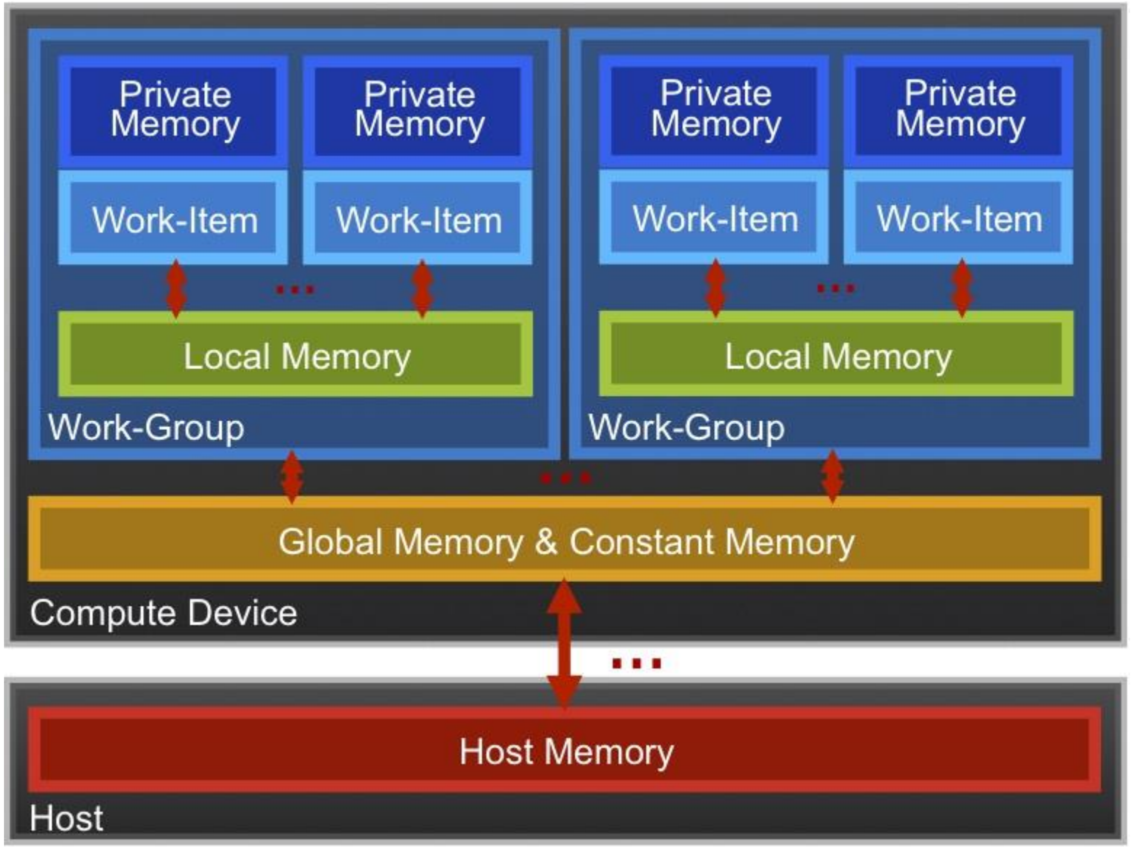
- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 128x128 (work-group, executes together)



- Choose the dimensions that are “best” for your algorithm

OpenCL Memory model

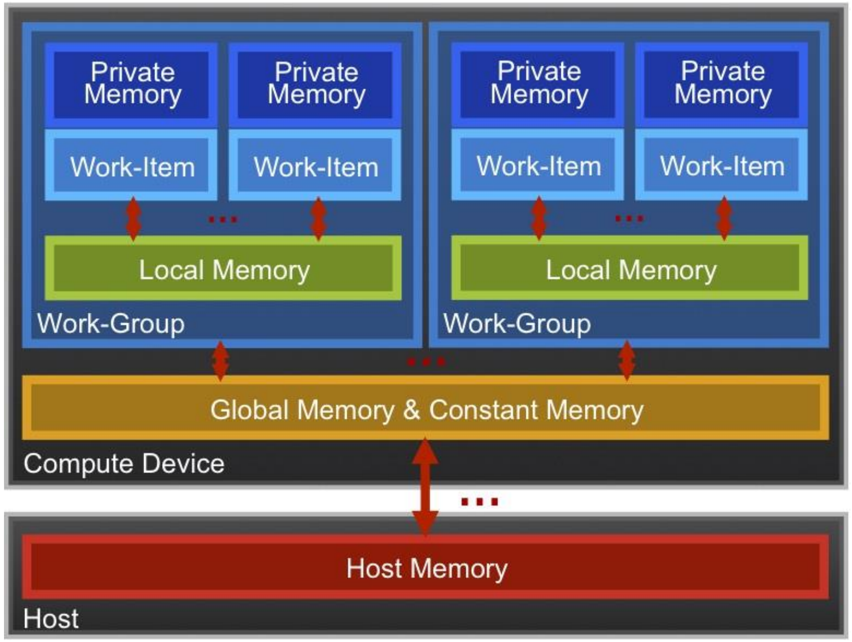
- Private Memory
  - Per work-item
- Local Memory
  - Shared within a work-group
- Global/Constant Memory
  - Visible to all work-groups
- Host memory
  - On the CPU



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

OpenCL Memory model

- Private Memory
  - Fastest & smallest: O(10) words/WI
- Local Memory
  - Shared by all WI's in a work-group
  - But not shared between work-groups!
  - O(1-10) Kbytes per work-group
- Global/Constant Memory
  - O(1-10) Gbytes of Global memory
  - O(10-100) Kbytes of Constant memory
- Host memory
  - On the CPU - GBytes



Memory management is **explicit**:
O(1-10) Gbytes/s bandwidth to discrete GPUs for
Host <-> Global transfers

Private Memory

- Managing the memory hierarchy is one of ***the*** most important things to get right to achieve good performance
- Private Memory:
  - A **very scarce** resource, only a few tens of 32-bit words per Work-Item at most
  - If you use too much it spills to global memory or reduces the number of Work-Items that can be run at the same time, potentially harming performance*
  - Think of these like registers on the CPU

* Occupancy on a GPU

Local Memory*

- Tens of KBytes per Compute Unit
  - As multiple Work-Groups will be running on each CU, this means only a fraction of the total Local Memory size is available to each Work-Group
- Assume O(1-10) KBytes of Local Memory per Work-Group
  - Your kernels are responsible for transferring data between Local and Global/Constant memories … there are optimized library functions to help
  - E.g. async_work_group_copy(), async_workgroup_strided_copy(), …
- Use Local Memory to hold data that can be reused by all the work-items in a work-group
- Access patterns to Local Memory affect performance in a similar way to accessing Global Memory
  - Have to think about things like coalescence & bank conflicts

* Typical figures for a 2013 GPU

Local Memory

- Local Memory doesn't always help…
  - CPUs don't have special hardware for it
  - This can mean excessive use of Local Memory might slow down kernels on CPUs
  - GPUs now have effective on-chip caches which can provide much of the benefit of Local Memory but without programmer intervention
  - So, your mileage may vary!

The Memory Hierarchy

Bandwidths

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

Local memory
O(10) words/cycle/WG

Global memory
O(100-200) GBytes/s

Host memory
O(1-100) GBytes/s

Sizes

Private memory
O(10) words/WI

Local memory
O(1-10) KBytes/WG

Global memory
O(1-10) GBytes

Host memory
O(1-100) GBytes

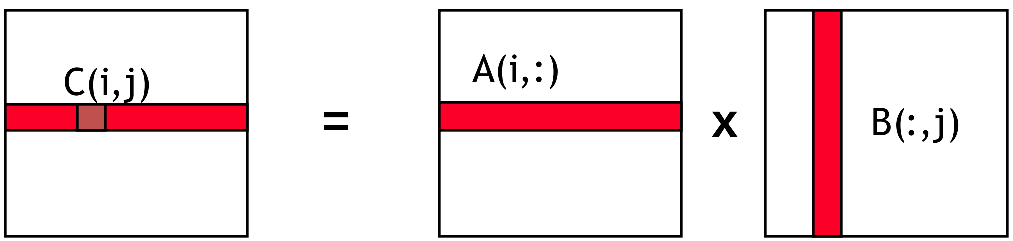
Speeds and feeds approx. for a high-end discrete GPU, circa 2011

Memory Consistency

- OpenCL uses a **relaxed consistency** memory model; i.e.
  - The state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.
- Within a work-item:
  - Memory has load/store consistency to the work-item's private view of memory, i.e. it sees its own reads and writes correctly
- Within a work-group:
  - Local memory is consistent between work-items at a barrier.
- Global memory is consistent within a work-group at a barrier, **but *not* guaranteed across different work-groups!!**
  - This is a common source of bugs!
- Consistency of memory shared between commands (e.g. kernel invocations) is enforced by synchronization (barriers, events, in-order queue)

Optimizing matrix multiplication

- There may be significant overhead to manage work-items and work-groups.
- So let's have each work-item compute a full row of C

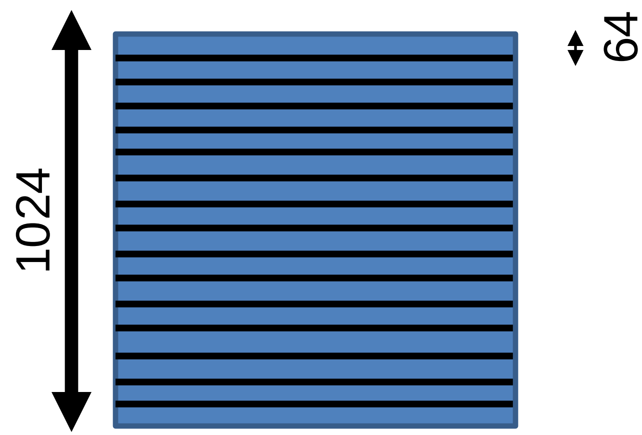


Dot product of a row of A and a column of B for each element of C

- And with an eye towards future optimizations, let's collect work-items into work-groups with 64 work-items per work-group

An N-dimension domain of work-items

- Global Dimensions: 1024 (1D)
  Whole problem space (index space)
- Local Dimensions: 64 (work-items per work-group)
  Only 1024/64 = 16 work-groups in total



- Important implication: we will have a lot fewer work-items per work-group (64) and work-groups (16). Why might this matter?

Matrix multiplication: One work item per row of C

```
__kernel void mmul(
   const int N,
   __global float *A,
   __global float *B,
   __global float *C)
{
  int j, k;
  int i = get_global_id(0);
  float tmp;
  for (j = 0; j < N; j++) {
   tmp = 0.0f;
   for (k = 0; k < N; k++)
     tmp += A[i*N+k]*B[k*N+j];
   C[i*N+j] = tmp;
  }
}
```

Matrix multiplication host program (C++ API)

Changes to host program:

1. 1D ND Range set to number of rows in the C matrix
2. Local Dimension set to 64 so number of work-groups match number of compute units (16 in this case) for our order 1024 matrices

```
int main(int
{
  std::vector
  int Mdim, N                                    true);
  int i, err;                                    true);
  int szA, sz
  double star
  cl::Program                                    C);

  Ndim = Pdim = Mdim = ORDER;
  szA = Ndim*Pdim;
  szB = Pdim*Mdim;
  szC = Ndim*Mdim;
  h_A   = std::vector<float>(szA);
  h_B   = std::vector<float>(szB);
  h_C   = std::vector<float>(szC);

  initmat(Mdim, Ndim, Pdim, h_A, h_B, h_C);

  // Compile for first kernel to setup program
  program = cl::Program(C_elem_KernelSource, true);
  Context context(CL_DEVICE_TYPE_DEFAULT);
  cl::CommandQueue queue(context);
  std::vector<Device> devices =
      context.getInfo<CL_CONTEXT_DEVICES>();
  cl::Device device = devices[0];
  std::string s =
      device.getInfo<CL_DEVICE_NAME>();
  std::cout << "\nUsing OpenCL Device "
            << s << "\n";

  cl::make_kernel<int, int, int,
                  cl::Buffer, cl::Buffer, cl::Buffer>
                  krow(program, "mmul");

  zero_mat(Ndim, Mdim, h_C);
  start_time = wtime();

  krow(cl::EnqueueArgs(queue
                       cl::NDRange(Ndim),
                       cl::NDRange(ORDER/16)),
       Ndim, Mdim, Pdim, a_in, b_in, c_out);

  cl::copy(queue, d_c, h_C.begin(), h_C.end());

  run_time  = wtime() - start_time;
  results(Mdim, Ndim, Pdim, h_C, run_time);
}
```

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8

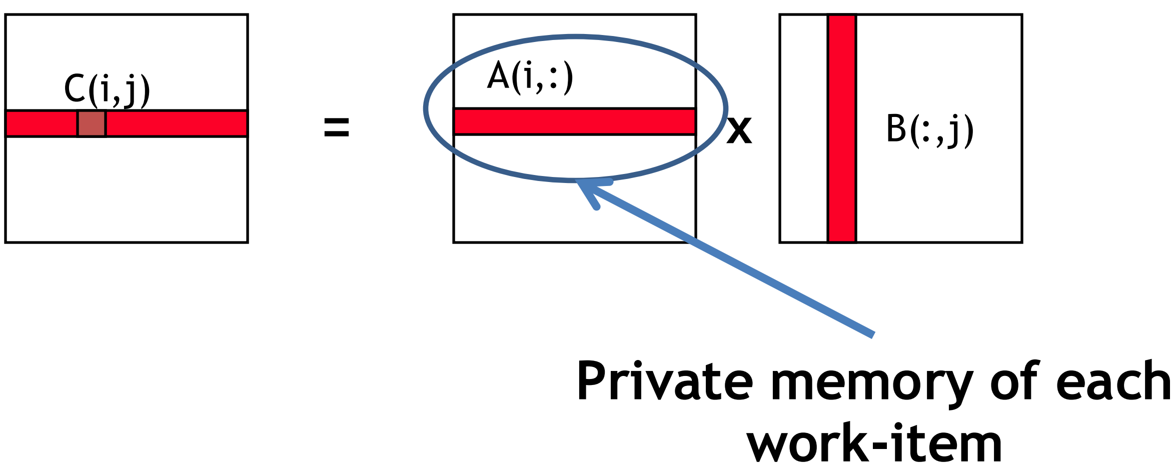
This has started to help.

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Optimizing matrix multiplication

- Notice that, in one row of C, each element reuses the same row of A.
- Let's copy that row of A into private memory of the work-item that's (exclusively) using it to avoid the overhead of loading it from global memory for each C(i,j) computation.



Matrix multiplication: (Row of A in private memory)

Copy a row of A into private memory from global memory before we start with the matrix multiplications.

```
__kernel void mmul(
   const int N,
   __global float *A,
   __global float *B,
   __global float *C)
{
  int j, k;
  int i =
    get_global_id(0);
  float tmp;
  float Awrk[1024];
```

Setup a work array for A in private memory*

```
  for (k = 0; k < N; k++)
    Awrk[k] = A[i*N+k];

  for (j = 0; j < N; j++) {
    tmp = 0.0f;
    for (k = 0; k < N; k++)
      tmp += Awrk[k]*B[k*N+j];

     C[i*N+j] += tmp;
  }
}
```

(*Actually, this is using *far* more private memory than we'll have and so Awrk[] will be spilled to global memory)

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8
C row per work-item, A row private	3,385.8	8,584.3

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

Big impact!

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Why using too much private memory can be a good thing

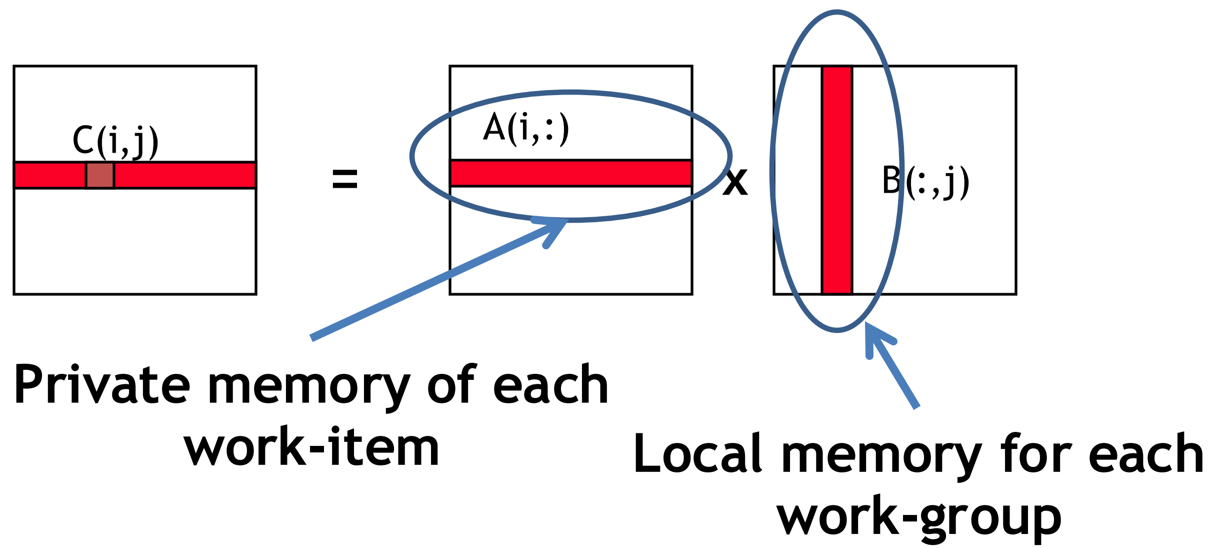
- In reality private memory is just hardware registers, so only dozens of these are available per work-item
- Many kernels will allocate too many variables to private memory
- So the compiler already has to be able to deal with this
- It does so by *spilling* excess private variables to (global) memory
- You still told the compiler something useful – that the data will only be accessed by a single work-item
- This lets the compiler allocate the data in such as way as to enable more efficient memory access

Exercise 7: using private memory

- Goal:
  - Use private memory to minimize memory movement costs and optimize performance of your matrix multiplication program
- Procedure:
  - Start with your matrix multiplication solution
  - Modify the kernel so that each work-item copies its own row of A into private memory
  - Optimize step by step, saving the intermediate versions and tracking performance improvements
- Expected output:
  - A message to standard output verifying that the matrix multiplication program is generating the correct results
  - Report the runtime and the MFLOPS

Optimizing matrix multiplication

- We already noticed that, in one row of C, each element uses the same row of A
- Each work-item in a work-group also uses the same columns of B
- So let's store the B columns in **local** memory (which is shared by the work-items in the work-group)



Matrix multiplication: B column shared between work-items

```
__kernel void mmul(
      const int N,
   __global float *A,
   __global float *B,
   __global float *C,
   __local  float *Bwrk)
{
 int j, k;
 int i =
    get_global_id(0);

 int iloc  =
      get_local_id(0);

 int nloc =
      get_local_size(0);

 float tmp;
 float Awrk[1024];
```

```
 for  (k = 0; k < N; k++)
   Awrk[k] = A[i*N+k];

 for (j = 0; j < N; j++) {

   for (k=iloc; k<N; k+=nloc)
     Bwrk[k] = B[k* N+j];

   barrier(CLK_LOCAL_MEM_FENCE);

   tmp = 0.0f;
   for (k = 0; k < N; k++)
     tmp += Awrk[k]*Bwrk[k];

   C[i*N+j] = tmp;

   barrier(CLK_LOCAL_MEM_FENCE);
 }
}
```

Pass a work array in local memory to hold a column of B. All the work-items do the copy "in parallel" using a cyclic loop distribution (hence why we need iloc and nloc)

Matrix multiplication host program (C++ API)

Changes to host program:

1. Pass local memory to kernels.
   1. This requires a change to the kernel argument lists … an arg of type LocalSpaceArg is needed.
   2. Allocate the size of local memory
   3. Update argument list in kernel functor

```
Ndim = Pdim = Mdim = ORDER;
szA = Ndim*Pdim;
szB = Pdim*Mdim;
szC = Ndim*Mdim;
h_A   = std::vector<float>(szA);
h_B   = std::vector<float>(szB);
h_C   = std::vector<float>(szC);

initmat(Mdim, Ndim, Pdim, h_A, h_B, h_C);

// Compile for first kernel to setup program
program = cl::Program(C_elem_KernelSource, true);
Context context(CL_DEVICE_TYPE_DEFAULT);
cl::CommandQueue queue(context);
std::vector<Device> devices =
    context.getInfo<CL_CONTEXT_DEVICES>();
cl::Device device = devices[0];
std::string s =
    device.getInfo<CL_DEVICE_NAME>();
std::cout << "\nUsing OpenCL Device "
          << s << "\n";
```

```
cl::LocalSpaceArg localmem =
                    cl::Local(sizeof(float) * Pdim);
```

```
cl::make_kernel<int, int, int,
           cl::Buffer, cl::Buffer, cl:::Buffer,
           cl::LocalSpaceArg>
              rowcol(program, "mmul");
```

```
zero_mat(Ndim, Mdim, h_C);
start_time = wtime();
```

```
rowcol(cl::EnqueueArgs(queue,
                       cl::NDRange(Ndim),
                       cl::NDRange(ORDER/16)),
      Ndim, Mdim, Pdim, d_a, d_b, d_c, localmem);
```

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

run_time  = wtime() - start_time;
results(Mdim, Ndim, Pdim, h_C, run_time);
}
```

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8
C row per work-item, A row private	3,385.8	8,584.3
C row per work-item, A private, B local	10,047.5	8,181.9

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Making matrix multiplication *really* fast

- Our goal has been to describe how to work with private, local and global memory. We've ignored many well-known techniques for making matrix multiplication fast
  - The number of work items must be a multiple of the fundamental machine "vector width". This is the wavefront on AMD, warp on NVIDIA, and the number of SIMD lanes exposed by vector units on a CPU
  - To optimize reuse of data, you need to use blocking techniques
    - Decompose matrices into tiles such that three tiles just fit in the fastest (private) memory
    - Copy tiles into local memory
    - Do the multiplication over the tiles
  - We modified the matrix multiplication program provided with the NVIDIA OpenCL SDK to work with our test suite to produce the blocked results on the following slide. This used register blocking with block sizes mapped onto the GPU's warp size

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8
C row per work-item, A row private	3,385.8	8,584.3
C row per work-item, A private, B local	10,047.5	8,181.9
Block oriented approach using local	1,534.0	230,416.7

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

Biggest impact so far!

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Exercise 8: using local memory

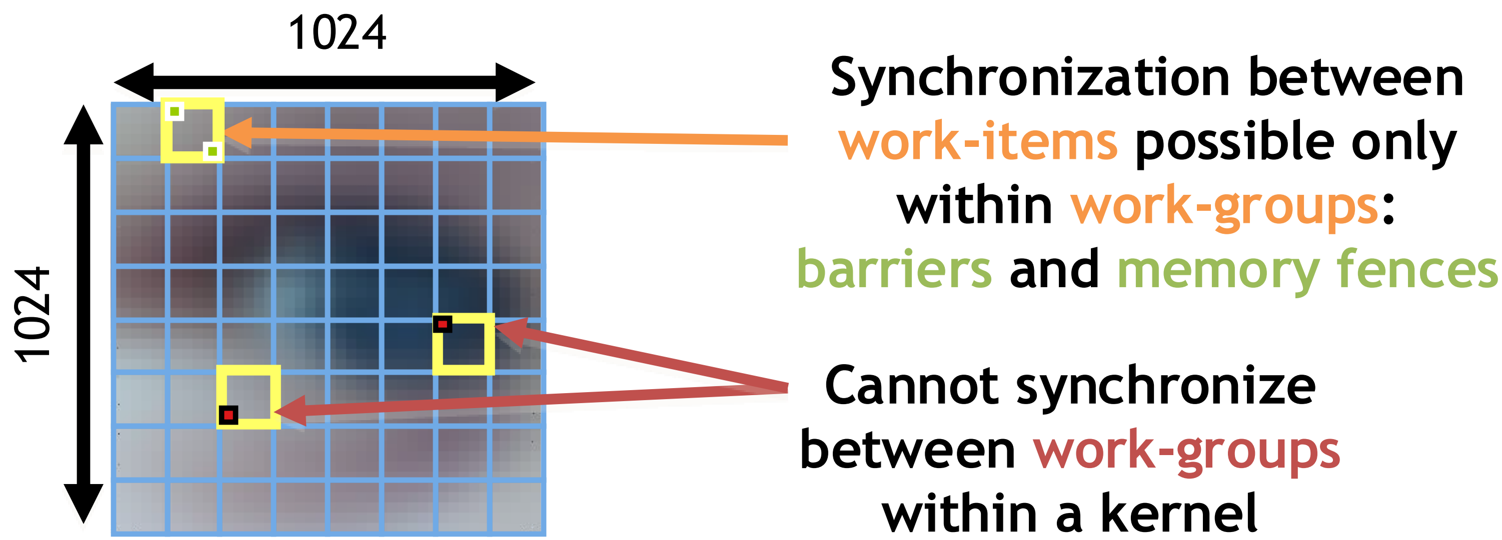
- Goal:
  - Use local memory to minimize memory movement costs and optimize performance of your matrix multiplication program
- Procedure:
  - Start with your matrix multiplication solution that already uses private memory from Exercise 7
  - Modify the kernel so that each work-group collaboratively copies its own column of B into local memory
  - Optimize step by step, saving the intermediate versions and tracking performance improvements
- Expected output:
  - A message to standard output verifying that the matrix multiplication program is generating the correct results
  - Report the runtime and the MFLOPS
- Extra:
  - Look at the fast, blocked implementation from the NVIDIA OpenCL SDK example. Try running it and compare to yours

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: 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;
}
```

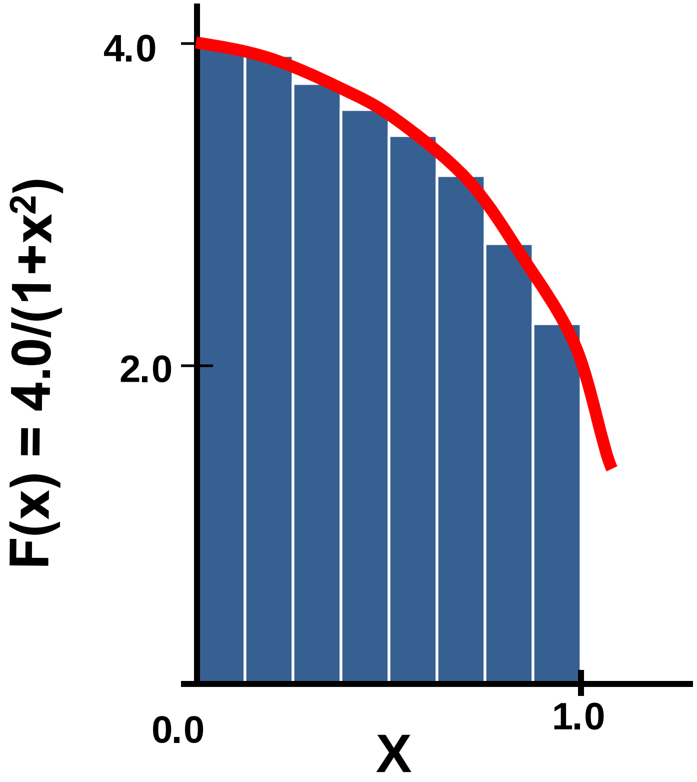
Simple parallel reduction

- 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

```c
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;
}
```

Exercise 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

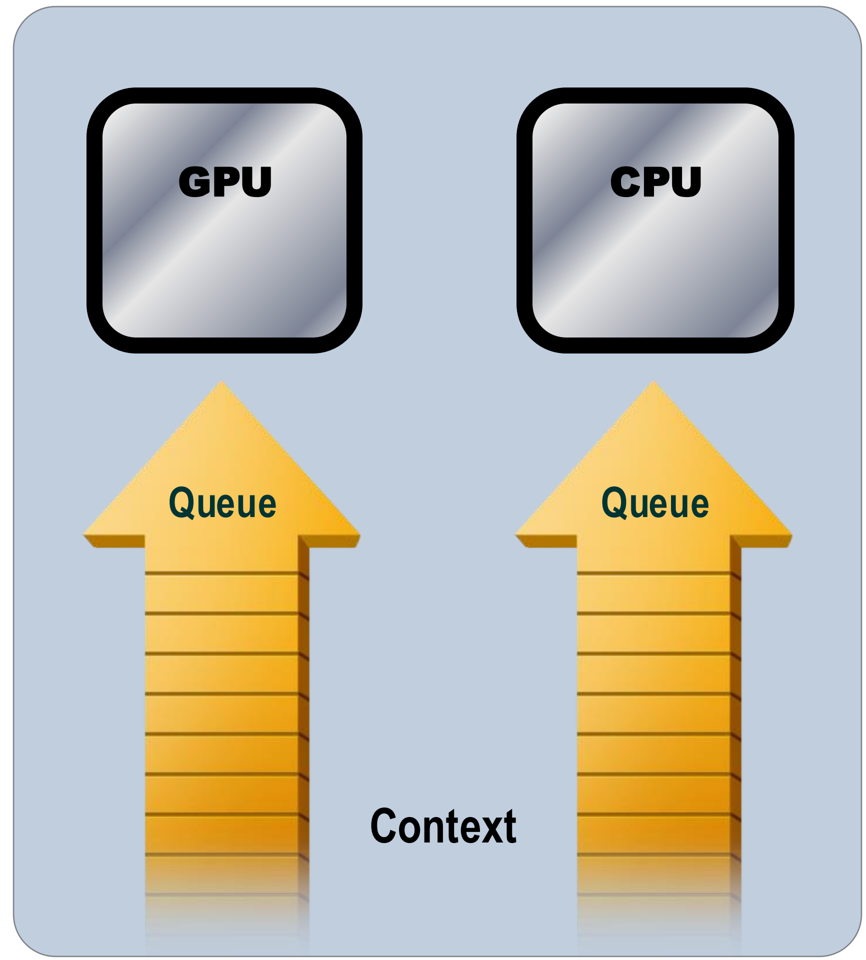
Hint: you will want each work-item to do many iterations of the loop, i.e. don't create one work-item per loop iteration. To do so would make the reduction so costly that performance would be terrible.

Lecture 8

HETEROGENEOUS COMPUTING WITH OPENCL

Running on the CPU and GPU

- Kernels can be run on multiple devices at the same time
- We can exploit many GPUs and the host CPU for computation
- Simply define a context with multiple platforms, devices and queues
- We can even synchronize between queues using Events (see appendix)
- Can have more than one context



Running on the CPU and GPU

1. Discover all your platforms and devices
   - Look at the API for finding out Platform and Device IDs

2. Set up the cl::Context with a vector of devices

```
cl::Context(const VECTOR_CLASS<Device> &devices,
          cl_context_properties *properties = NULL,
          void (CL_CALLBACK *pfn_notify)(
            const char *errorinfo,
            const void *private_info_size,
            ::size_t cb, void *user_data) = NULL,
          void *user_data = NULL, cl_int *err = NULL);
```

3. Create a Command Queue for each of these devices
   - C examples in the NVIDIA (oclSimpleMultiGPU) and AMD (SimpleMultiDevice) OpenCL SDKs

The steps are the same in C and Python, just the API calls differ as usual

Exercise 10: Heterogeneous Computing

- Goal:
  - To experiment with running kernels on multiple devices
- Procedure:
  - Take one of your OpenCL programs
  - Investigate the Context constructors to include more than one device
  - Modify the program to run a kernel on multiple devices, each with different input data
  - Split your problem across multiple devices if you have time
  - Use the examples from the SDKs to help you
- Expected output:
  - Output the results from both devices and see which runs faster

Lecture 9

ENABLING PORTABLE PERFORMANCE VIA OPENCL

Portable performance in OpenCL

- Portable performance is always a challenge, more so when OpenCL devices can be so varied (CPUs, GPUs, …)
- But OpenCL provides a powerful framework for writing performance portable code
- The following slides are general advice on writing code that should work well on most OpenCL devices

Optimization issues

- Efficient access to memory
  - Memory coalescing
    - Ideally get work-item i to access data[i] and work-item j to access data[j] at the same time etc.
  - Memory alignment
    - Padding arrays to keep everything aligned to multiples of 16, 32 or 64 bytes
- Number of work-items and work-group sizes
  - Ideally want at least 4 work-items per PE in a Compute Unit on GPUs
  - More is better, but diminishing returns, and there is an upper limit
    - Each work item consumes PE finite resources (registers etc)
- Work-item divergence
  - What happens when work-items branch?
  - Actually a SIMD data parallel model
  - Both paths (if-else) may need to be executed (*branch divergence*), avoid where possible (non-divergent branches are termed *uniform*)

Memory layout is critical to performance

- "Structure of Arrays vs. Array of Structures" problem:

```
struct { float x, y, z, a; } Point;
```

- Structure of Arrays (SoA) suits memory coalescence on GPUs

x	x	x	x	…	y	y	y	y	…	z	z	z	z	…	a	a	a	a	…

Adjacent work-items like to access adjacent memory

- Array of Structures (AoS) may suit cache hierarchies on CPUs

x	y	z	a	…	x	y	z	a	…	x	y	z	a	…	x	y	z	a	…

Individual work-items like to access adjacent memory

Other optimisation tips

- Use a profiler to see if you're getting good performance
  - Occupancy is a measure of how active you're keeping each PE
  - Occupancy measurements of >0.5 are good (>50% active)
- Other measurements to consider with the profiler:
  - Memory bandwidth – should aim for a good fraction of peak
    - E.g. 148 GBytes/s to Global Memory on an M2050 GPU
  - Work-Item (Thread) divergence – want this to be low
  - Registers per Work-Item (Thread) – ideally low and a nice divisor of the number of hardware registers per Compute Unit
    - E.g. 32,768 on M2050 GPUs
    - These are statically allocated and shared between all Work-Items and Work-Groups assigned to each Compute Unit
    - Four Work-Groups of 1,024 Work-Items each would result in just 8 registers per Work-Item! Typically aim for 16-32 registers per Work-Item

Portable performance in OpenCL

- Don't optimize too hard for any one platform, e.g.
  - Don't write specifically for certain warp/wavefront sizes etc
  - Be careful not to rely on specific sizes of local/global memory
  - OpenCL's vector data types have varying degrees of support – faster on some devices, slower on others
  - Some devices have caches in their memory hierarchies, some don't, and it can make a big difference to your performance without you realizing
  - Choosing the allocation of Work-Items to Work-Groups and dimensions on your kernel launches
  - Performance differences between unified vs. disjoint host/global memories
  - Double precision performance varies considerably from device to device
  - Some OpenCL SDKs give useful feedback about how well they can compile your code (but you have to turn on this feedback)

- It is a good idea to try your code on several different platforms to see what happens (profiling is good!)
  - At least two different GPUs (ideally different vendors) and at least one CPU

Advice for performance portability

- Discover what devices you have available at run-time, e.g.

C

```c
// Get available platforms
cl_uint nPlatforms;
cl_platform_id platforms[MAX_PLATFORMS];
int ret = clGetPlatformIDs(MAX_PLATFORMS, platforms, &nPlatforms);

// Loop over all platforms
for (int p = 0; p < nPlatforms; p++) {
  // Get available devices
  cl_uint nDevices = 0;
  cl_device_id devices[MAX_DEVICES];
  clGetDeviceIDs(platforms[p], deviceType, MAX_DEVICES, devices, &nDevices);

  // Loop over all devices in this platform
  for (int d = 0; d < nDevices; d++)
    getDeviceInformation(devices[d]);
}
```

Advice for performance portability

- Micro-benchmark all your OpenCL devices at run-time to gauge how to divide your total workload across all the devices
  - Ideally use some real work so you're not wasting resource
  - Keep the microbenchmark very short otherwise slower devices penalize faster ones
- Once you've got a work fraction per device calculated, it might be worth retesting from time to time
  - The behavior of the workload may change
  - The host or devices may become busy (or quiet)
- It is most important to keep the fastest devices busy
  - Less important if slower devices finish slightly earlier than faster ones (and thus become idle)
- Avoid overloading the CPU with both OpenCL host code and OpenCL device code at the same time

Timing microbenchmarks (C)

```
for (int i = 0; i < numDevices; i++) {
    // Wait for the kernel to finish
    ret = clFinish(oclDevices[i].queue);
    // Update timers
    cl_ulong start, end;
    ret = clGetEventProfilingInfo(oclDevices[i].kernelEvent,
              CL_PROFILING_COMMAND_START,
              sizeof(cl_ulong), &start, NULL);
    ret |= clGetEventProfilingInfo(oclDevices[i].kernelEvent,
              CL_PROFILING_COMMAND_END,
              sizeof(cl_ulong), &end, NULL);
    long timeTaken = (end - start);
    speeds[i] = timeTaken / oclDevices[i].load;
}
```

Advice for performance portability

- Optimal Work-Group sizes will differ between devices
  - E.g. CPUs tend to prefer 1 Work-Item per Work-Group, while GPUs prefer lots of Work-Items per Work-Group (usually a multiple of the number of PEs per Compute Unit, i.e. 32, 64 etc.)
- From OpenCL v1.1 you can discover the preferred Work-Group size multiple for a kernel once it's been built for a specific device
  - Important to pad the total number of Work-Items to an exact multiple of this
  - Again, will be different per device
- The OpenCL run-time will have a go at choosing good EnqueueNDRangeKernel dimensions for you
  - With very variable results
- **Your mileage will vary**, the best strategy is to write ***adaptive*** code that makes decisions at run-time

Tuning Knobs
some general issues to think about

- Tiling size (work-group sizes, dimensionality etc.)
  - For block-based algorithms (e.g. matrix multiplication)
  - Different devices might run faster on different block sizes
- Data layout
  - Array of Structures or Structure of Arrays (AoS vs. SoA)
  - Column or Row major
- Caching and prefetching
  - Use of local memory or not
  - Extra loads and stores assist hardware cache?
- Work-item / work-group data mapping
  - Related to data layout
  - Also how you parallelize the work
- Operation-specific tuning
  - Specific hardware differences
  - Built-in trig / special function hardware
  - Double vs. float (vs. half)

From Zhang, Sinclair II and Chien: Improving Performance Portability in OpenCL Programs – ISC13

Auto tuning

- Q: How do you know what the ***best*** parameter values for your program are?
  - What is the best work-group size, for example

- A: Try them all! (Or a well chosen subset)

- This is where auto tuning comes in
  - Run through different combinations of parameter values and optimize the runtime (or another measure) of your program.

Auto tuning example - Flamingo

- http://mistymountain.co.uk/flamingo/
- Python program which compiles your code with different parameter values, and calculates the “best” combination to use
- Write a simple config file, and Flamingo will run your program with different values, and returns the best combination
- Remember: scale down your problem so you don’t have to wait for “bad” values (less iterations, etc.)

Auto tuning - Example

- D2Q9 Lattice-Boltzmann
- What is the best work-group size for a specific problem size (3000x2000) on a specific device (NVIDIA Tesla M2050)?

Exercise 11: Optimize matrix multiplication

- Goal:
  - To understand portable performance in OpenCL
- Procedure:
  - Optimize a matrix multiply solution step by step, saving intermediate versions and tracking performance improvements
  - After you've tried to optimize the program on your own, study the blocked solution optimized for an NVIDIA GPU. Apply these techniques to your own code to further optimize performance
  - As a final step, go back and make a single program that is adaptive so it delivers good results on both a CPU and a GPU
- Expected output:
  - A message confirming that the matrix multiplication is correct
  - Report the runtime and the MFLOPS

Lecture 10

OPTIMIZING OPENCL PERFORMANCE

Extrae and Paraver

- From Barcelona Supercomputing Center
  - http://www.bsc.es/computer-sciences/performance-tools/trace-generation
  - http://www.bsc.es/computer-sciences/performance-tools/paraver
- Create and analyze traces of OpenCL programs
  - Also MPI, OpenMP
- Required versions:
  - Extrae v2.3.5rc
  - Paraver 4.4.5

Extrae and Paraver

1. Extrae *instruments* your application and produces "timestamped events of runtime calls, performance counters and source code references"
   - Allows you to measure the run times of your API and kernel calls

2. Paraver provides a way to view and analyze these traces in a graphical way

Important!

- At the moment NVIDIA® GPUs support up to OpenCL v1.1 and AMD® and Intel® support v1.2
- If you want to profile on NVIDIA® devices you **must** compile Extrae against the NVIDIA headers and runtime otherwise v1.2 code will be used by Extrae internally which will cause the trace step to segfault

Installing Extrae and Paraver

- Paraver is easy to install on Linux
  - Just download and unpack the binary
- Extrae has some dependencies, some of which you'll have to build from source
  - libxml2
  - binutils-dev
  - libunwind
  - PAPI
  - MPI (optional)
- Use something like the following command line to configure before "make && make install":

```
./configure --prefix=$HOME/extrae --with-
binutils=$HOME --with-papi=$HOME --with-mpi=$HOME
--without-dyninst --with-unwind=$HOME --with-
opencl=/usr/local/ --with-opencl-libs=/usr/lib64
```

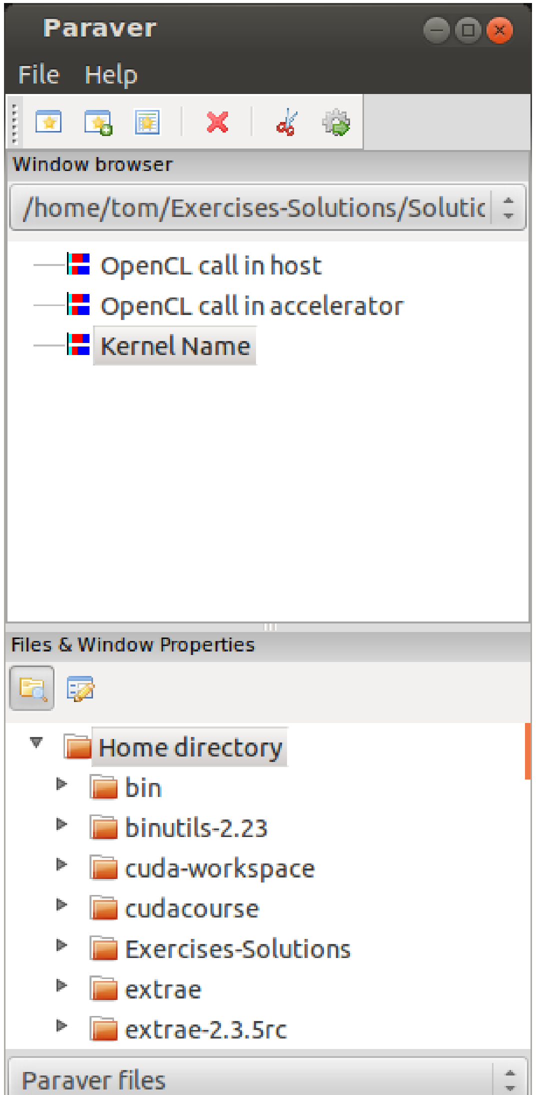
Step 1 – tracing your code

- Copy the trace.sh script from extrae/share/example/OPENCL to your project directory
  - This sets up a few environment variables and then runs your compiled binary
- Copy the extrae.xml file from the same location to your project directory
  - This gives some instructions to Extrae as to how to profile your code
  - Lots of options here – see their user guide
  - The default they provide is fine to use to begin with
- Trace!
  - **`./trace.sh ./a.out`**

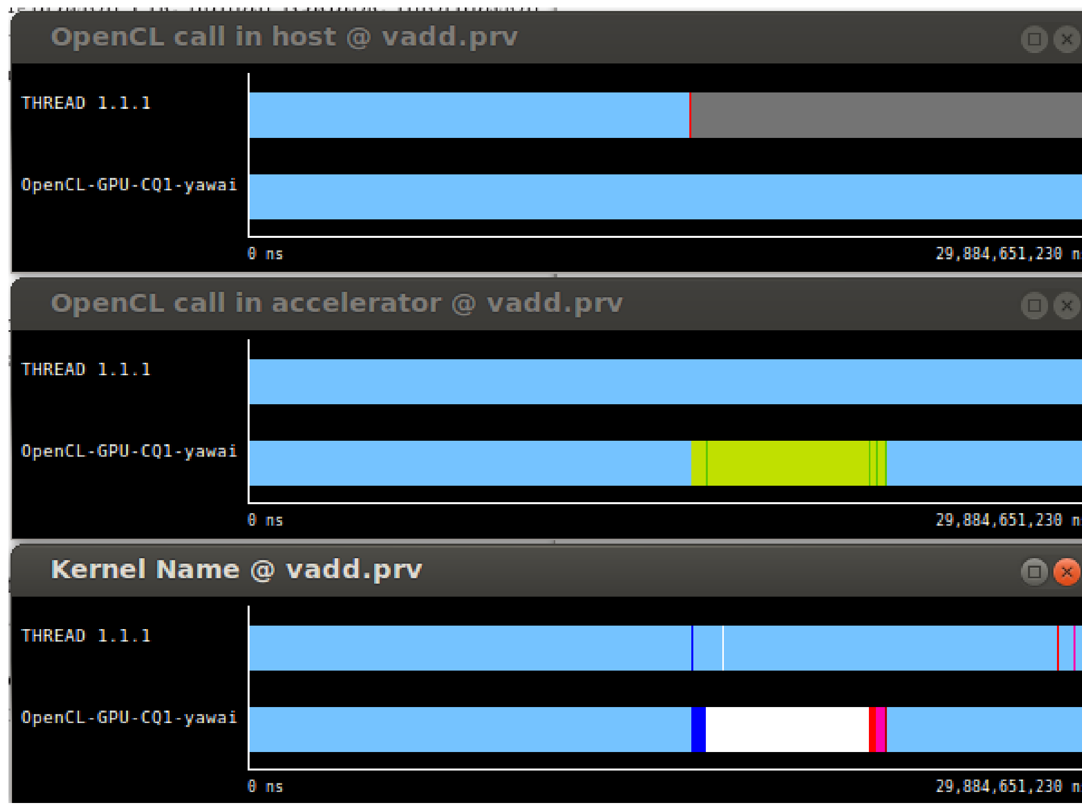
Step 2 – visualize the trace

- Extrae produces a number of files
  - .prv, .pcf, .row, etc…
- Run Paraver
  - **`./wxparaver-<version>/bin/wxparaver`**
- Load in the trace
  - File –> Load Trace -> Select the .prv file
- Load in the provided OpenCL view config file
  - File -> Load configuration -> wxparaver-<version>/cfgs/OpenCL/views/opencl_call.cfg
- The traces appear as three windows
  1. OpenCL call in host - timings of API calls
  2. Kernel Name – run times of kernel executions
  3. OpenCL call in accelerator – information about total compute vs memory transfer times

Paraver



Paraver
File
Help
Window browser
/home/tom/Exercises-Solutions/Solutic
OpenCL call in host
OpenCL call in accelerator
Kernel Name
Files & Window Properties
Home directory
bin
binutils-2.23
cuda-workspace
cudacourse
Exercises-Solutions
extrae
extrae-2.3.5rc
Paraver files



OpenCL call in host @ vadd.prv
THREAD 1.1.1
OpenCL-GPU-CQ1-yawai
0 ns
29,884,651,230 ns
OpenCL call in accelerator @ vadd.prv
THREAD 1.1.1
OpenCL-GPU-CQ1-yawai
0 ns
29,884,651,230 ns
Kernel Name @ vadd.prv
THREAD 1.1.1
OpenCL-GPU-CQ1-yawai
0 ns
29,884,651,230 ns

Usage Tips

- Show what the colours represent
  - Right click -> Info Panel
- Zoom in to examine specific areas of interest
  - Highlight a section of the trace to populate the timeline window
- Tabulate the data – numerical timings of API calls
  - Select a timeline in the Paraver main window, click on the 'New Histogram' icon and select OK
- Powerful software – can also pick up your MPI communications
- Perform calculations with the data – see the Paraver user guide

Platform specific profilers

- More information can be obtained about your OpenCL program by profiling it using the hardware vendors dedicated profilers

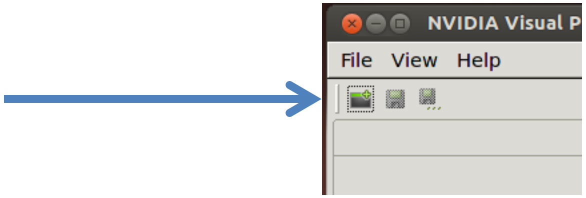
- OpenCL profiling can be done with Events in the API itself for specific profiling of queues and kernel calls

NVIDIA Visual Profiler®

This gives us information about:

- Device occupancy
- Memory bandwidth(between host and device)
- Number of registers uses
- Timeline of kernel executions and memory copies
- Etc…

- Start a new session:

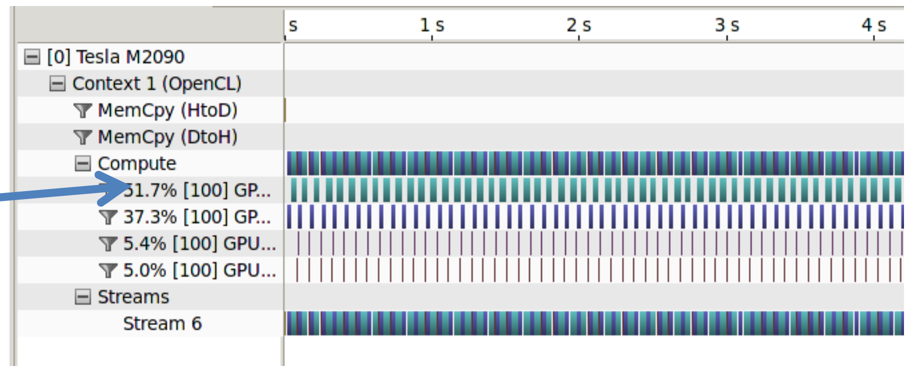


- Follow the wizard, selecting the compiled binary in the File box (you do not need to make any code or compiler modifications). You can leave the other options as the default.
- The binary is then run and profiled and the results displayed.

Profiling using nvvp

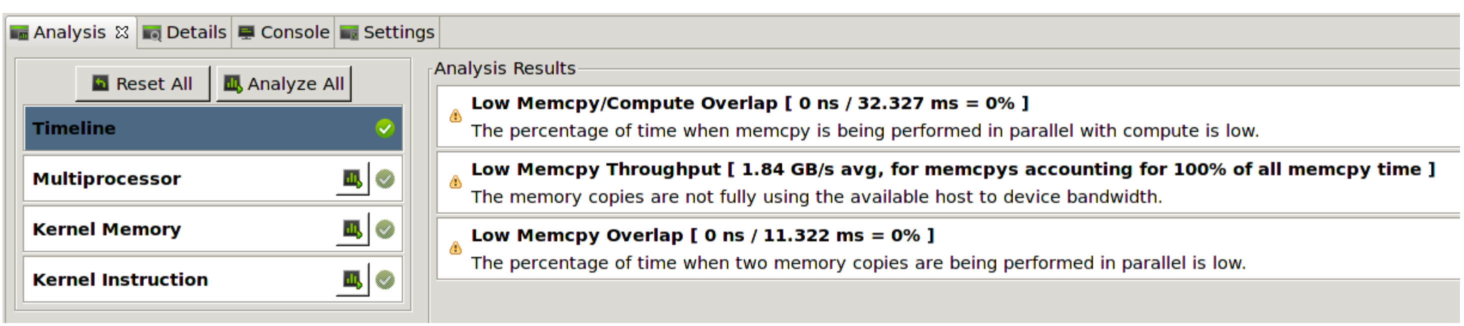
- The timeline says what happened during the program execution:

Kernels



Each invocation of the kernel is pictured as a box

- Some things to think about optimising are displayed in the Analysis tab:



Profiling using nvvp

- The Details tab shows information for each kernel invocation and memory copy
  - number of registers used
  - work group sizes
  - memory throughput
  - amount of memory transferred

- No information about which parts of the kernel are running slowly, but the figures here might give us a clue as to where to look

- Best way to learn: experiment with an application yourself

Profiling from the command line

- NVIDIA® also have nvprof and 'Command Line Profiler'
- nvprof available with CUDA™ 5.0 onwards, but currently lacks driver support for OpenCL profiling
- The legacy command-line profiler can be invoked using environment variables:

```
$ export COMPUTE_PROFILE=1
$ export COMPUTE_PROFILE_LOG=<output file>
$ export COMPUTE_PROFILE_CONFIG=<config file>
```

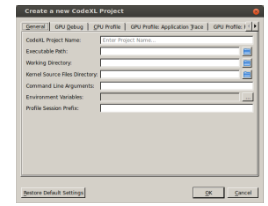
- Config file controls which events to collect (run **nvprof --query-events** for a comprehensive list)
- Run your application to collect event information and then inspect output file with text editor
- Can also output CSV information (**COMPUTE_PROFILE_CSV=1**) for inspection with a spreadsheet or import into nvvp (limited support)

AMD® CodeXL

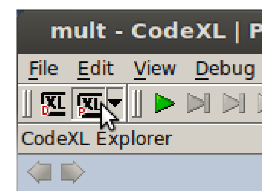
- AMD provide a graphical Profiler and Debugger for AMD Radeon™ GPUs
- Can give information on:
  - API and kernel timings
  - Memory transfer information
  - Register use
  - Local memory use
  - Wavefront usage
  - Hints at limiting performance factors

CodeXL

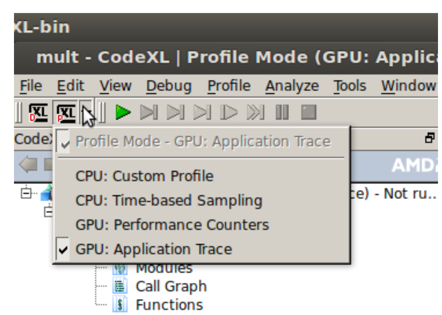
- Create a new project, inserting the binary location in the window



- Click on the Profiling button, and hit the green arrow to run your program



- Select the different traces to view associated information



CodeXL

- GPU: Performance Counters
  - Information on kernels including work group sizes, registers, etc.
  - View the kernel instruction code
    - Click on the kernel name in the left most column
  - View some graphs and hints about the kernel
    - Click on the Occupancy result

Performance Counters

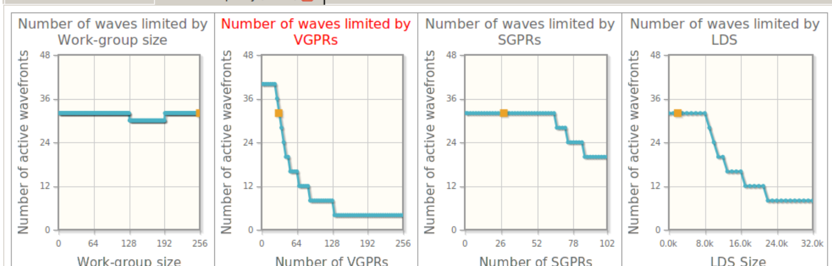
☑ Show Zero Columns

	Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	WorkGroupSize	Time	Lc
1	mmul_k1_Tahiti1	1	31203	164	{ 1024 1024 1}	NULL	411.37407	0
2	mmul_k2_Tahiti1	2	31203	275	{ 1024 1 1}	NULL	873.42963	0
3	mmul_k3_Tahiti1	3	31203	386	{ 1024 1 1}	{ 64 1 1}	536.93926	0
4	mmul_k4_Tahiti1	4	31203	498	{ 1024 1 1}	{ 64 1 1}	534.13407	409
5	mmul_k5_Tahiti1	5	31203	611	{ 1024 1024 1}	{ 16 16 1}	2.69600	204

elp

57 (CPU: Time-based Sampling) | Aug-28-2013_10-23-28 (GPU: Application Trace) | Aug-28-2013_10-23-52 (GPU: Performance Counters)

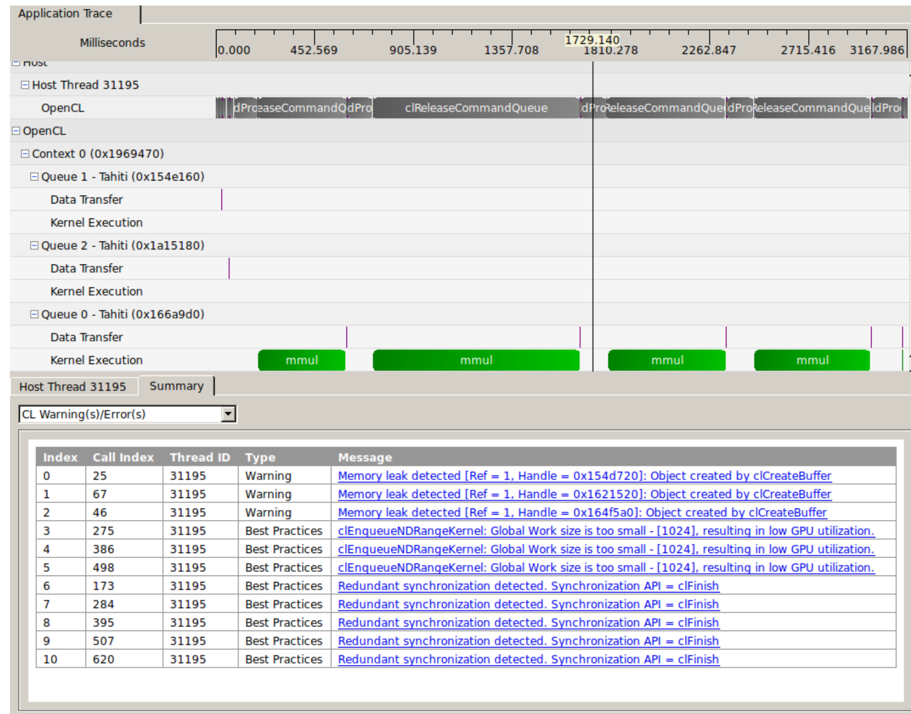
Performance Counters | Kernel Occupancy (mmul)



Variable	Value	Device Limit
Device Info		
Device name	Tahiti	
Number of compute units	32	
Max number of waves per compute unit	40	
Max number of work-groups per compute unit	16	
Wavefront size	64	
Kernel Info		
Kernel name	mmul	
Vector GPR usage per work-item	31	256
Scalar GPR usage per work-item	28	102
LDS usage per work-group	2048	65536
Flattened work-group size	256	256
Flattened global work size	1048576	16777216
Number of waves per work-group	4	4
Kernel Occupancy		
Number of waves limited by Vector GPR and Work-group size	32	40
Number of waves limited by Scalar GPR and Work-group size	40	40
Number of waves limited by LDS and Work-group size	40	40
Number of waves limited by Work-group size	40	40
Limiting factor(s)	**VGPR**	
Estimated occupancy	80%	

CodeXL

- GPU: Application Trace
  - See timing information about API calls
  - Timings of memory movements
  - Timings of kernel executions



Exercise 12: Profiling OpenCL programs

- Goal:
  - To experiment with profiling tools
- Procedure:
  - Take one of your OpenCL programs, such as matrix multiply
  - Run the program in the profiler and explore the results
  - Modify the program to change the performance in some way and observe the effect with the profiler
  - Repeat with other programs if you have time
- Expected output:
  - Timings reported by the host code and via the profiling interfaces should roughly match

Lecture 11

DEBUGGING OPENCL

Debugging OpenCL

- Parallel programs can be challenging to debug
- Luckily there are some tools to help
- Firstly, if your device can run OpenCL 1.2, you can printf straight from the kernel.

```
__kernel void func(void)
{
  int i = get_global_id(0);
  printf(" %d\n ", i);
}
```

- Here, each work-item will print to stdout
- Note: there is some buffering between the device and the output, but will be flushed by calling clFinish (or equivalent)

Debugging OpenCL 1.1

- Top tip:
  - Write data to a global buffer from within the kernel

```
result[ get_global_id(0) ] = … ;
```

  - Copy back to the host and print out from there or debug as a normal serial application
- Works with any OpenCL device and platform

Debugging OpenCL – more tips

- **Check your error messages!**
  - If you enable Exceptions in C++ as we have here, make sure you print out the errors.
- Don't forget, use the err_code.c from the tutorial to print out errors as strings (instead of numbers), or check in the cl.h file in the include directory of your OpenCL provider for error messages
- Check your work-group sizes and indexing

Debugging OpenCL - GDB

- Can also use GDB to debug your programs on the CPU
  - This will also leverage the memory system
    - Might catch illegal memory dereferences more accurately
  - But it does behave differently to accelerator devices so bugs may show up in different ways
- As with debugging, compile your C or C++ programs with the `–g` flag

Debugging OpenCL - GDB

- Require platform specific instructions depending on if you are using the AMD® or Intel® OpenCL platform
  - This is in part due to the ICD (Installable Client Driver) ensuring that the correct OpenCL runtime is loaded for the chosen platform
  - Also different kernel compile flags are accepted/required by different OpenCL implementations
- Remember: your CPU may be listed under each platform – ensure you choose the right debugging method for the *platform*

Using GDB with AMD®

- Ensure you select the CPU device from the AMD® platform
- Must use the –g flag and turn off all optimizations when building the kernels:

```
program.build(" -g -O0" )
```

- The symbolic name of a kernel function "**__kernel void foo(args)**" is "**__OpenCL_foo_kernel**"
  - To set a breakpoint on kernel entry enter at the GDB prompt:

```
break __OpenCL_foo_kernel
```

- Note: the debug symbol for the kernel will not show up until the kernel has been built by your host code
- AMD® recommend setting the environment variable `CPU_MAX_COMPUTE_UNITS=1` to ensure deterministic kernel behaviour

Using GDB with Intel®

- Ensure you select the CPU device from the Intel® platform
- Must use the -g flag and specify the kernel source file when building the kernels:

```
program.build(" -g -s
/full/path/to/kernel.cl" )
```

- The symbolic name of a kernel function "**__kernel void foo(args)**" is "**foo**"
  - To set a breakpoint on kernel entry enter at the GDB prompt:

```
break foo
```

- Note: the debug symbol for the kernel will not show up until the kernel has been built by your host code

Debugging OpenCL – Using GDB

- Use ***n*** to move to the next line of execution
- Use ***s*** to step into the function
- If you reach a segmentation fault, ***backtrace*** lists the previous few execution frames
  - Type ***frame 5*** to examine the 5th frame
- Use ***print varname*** to output the current value of a variable

Oclgrind

- A SPIR interpreter and OpenCL simulator
- Developed at the University of Bristol
- Runs OpenCL kernels in a simulated environment to catch various bugs:
  - `oclgrind ./application`
  - Invalid memory accesses
  - Data-races (`--data-races`)
  - Work-group divergence
  - Runtime API errors (`--check-api`)
- Also has a GDB-style interactive debugger
  - `oclgrind -i ./application`
- More information on the Oclgrind Website

GPUVerify

- A useful tool for detecting data-races in OpenCL programs
- Developed at Imperial College as part of the CARP project
- Uses static analysis to try to prove that kernels are free from races
- Can also detect issues with work-group divergence
- More information on the [GPUVerify Website]

```
gpuverify --local_size=64,64 --num_groups=256,256 kernel.cl
```

Other debugging tools

- AMD® CodeXL
  - For AMD® APUs, CPUs and GPUs
    - Graphical Profiler and Debugger
- NVIDIA® Nsight™ Development Platform
  - For NVIDIA® GPUs
    - IDE, including Profiler and Debugger
- GPUVerify
  - Formal analysis of kernels
  - http://multicore.doc.ic.ac.uk/tools/GPUVerify/

Note: Debugging OpenCL is still changing rapidly - your mileage may vary when using GDB and these tools

Lecture 12

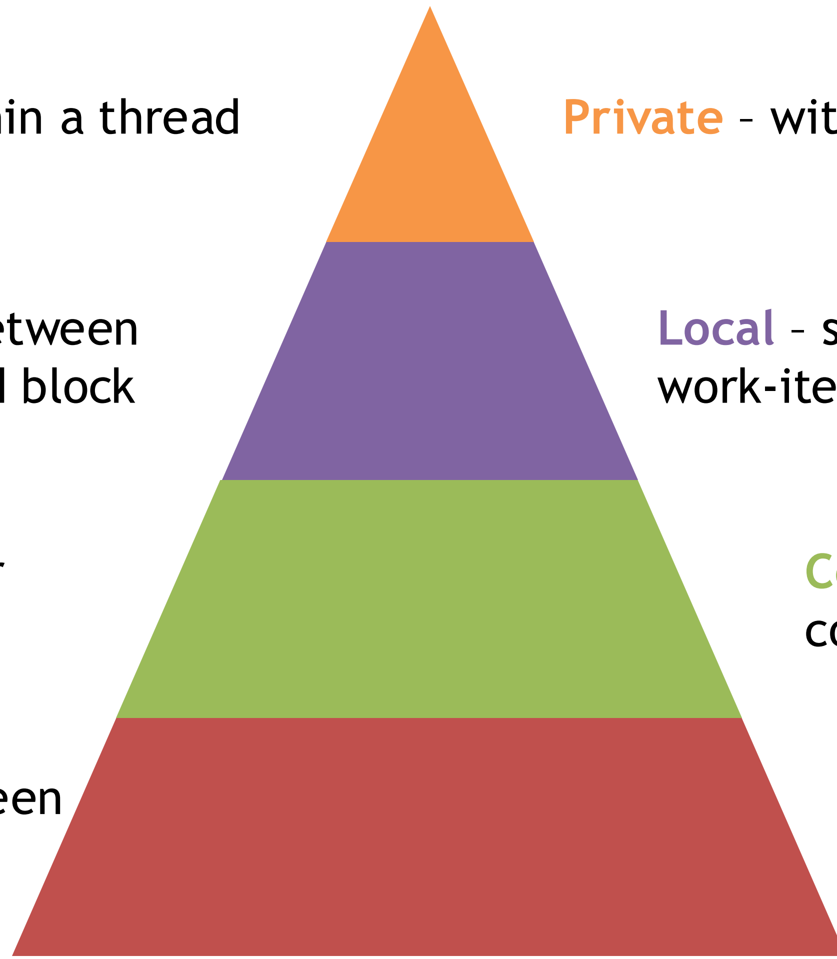
PORTING CUDA TO OPENCL

Introduction to OpenCL

- If you have CUDA code, you've already done the hard work!
  - I.e. working out how to split up the problem to run effectively on a many-core device

- Switching between CUDA and OpenCL is mainly changing the host code syntax
  - Apart from indexing and naming conventions in the kernel code (simple to change!)

Memory Hierarchy Terminology



CUDA	OpenCL
Local – within a thread	**Private** – within a work-item
Shared – shared between threads in a thread block	**Local** – shared between work-items in a work-group
Constant – a cache for constant memory	**Constant** – a cache for constant memory
Device – shared between all thread blocks	**Global** – shared between all work-groups

Allocating and copying memory

	CUDA C	OpenCL C

Allocate	`float* d_x;`
	`cudaMalloc(&d_x,`
	`sizeof(float)*size);`
		`cl_mem d_x =`
		`  clCreateBuffer(context,`
		`    CL_MEM_READ_WRITE,`
		`    sizeof(float)*size,`
		`    NULL, NULL);`

Host to Device	`cudaMemcpy(d_x, h_x,`
	`    sizeof(float)*size,`
	`    cudaMemcpyHostToDevice);`
		`clEnqueueWriteBuffer(queue, d_x,`
		`      CL_TRUE, 0,`
		`      sizeof(float)*size,`
		`      h_x, 0, NULL, NULL);`

Device to Host	`cudaMemcpy(h_x, d_x,`
	`    sizeof(float)*size,`
	`    cudaMemcpyDeviceToHost);`
		`clEnqueueReadBuffer(queue, d_x,`
		`      CL_TRUE, 0,`
		`      sizeof(float)*size,`
		`      h_x, 0, NULL, NULL);`

Allocating and copying memory

	CUDA C	OpenCL C++

Allocate	`float* d_x;`
	`cudaMalloc(&d_x,`
	`    sizeof(float)*size);`
		`cl::Buffer`
		`  d_x(begin(h_x), end(h_x), true);`

Host to Device	`cudaMemcpy(d_x, h_x,`
	`    sizeof(float)*size,`
	`    cudaMemcpyHostToDevice);`
		`cl::copy(begin(h_x), end(h_x),`
		`         d_x);`

Device to Host	`cudaMemcpy(h_x, d_x,`
	`    sizeof(float)*size,`
	`    cudaMemcpyDeviceToHost);`
		`cl::copy(d_x,`
		`         begin(h_x), end(h_x));`

Declaring dynamic local/shared memory

CUDA C

1. Define an array in the kernel source as extern

```
__shared__ int array[];
```

2. When executing the kernel, specify the third parameter as size in bytes of shared memory

```
func<<<num_blocks,
 num_threads_per_block,
 shared_mem_size>>>(args);
```

OpenCL C++

1. Have the kernel accept a local array as an argument

```
__kernel void func(
        __local int *array)
{}
```

2. Define a local memory kernel kernel argument of the right size

```
cl::LocalSpaceArg localmem =
    cl::Local(shared_mem_size);
```

3. Pass the argument to the kernel invocation

```
func(EnqueueArgs(…),localmem);
```

Declaring dynamic local/shared memory

CUDA C

1. Define an array in the kernel source as extern

```
__shared__ int array[];
```

2. When executing the kernel, specify the third parameter as size in bytes of shared memory

```
func<<<num_blocks,
 num_threads_per_block,
 shared_mem_size>>>(args);
```

OpenCL C

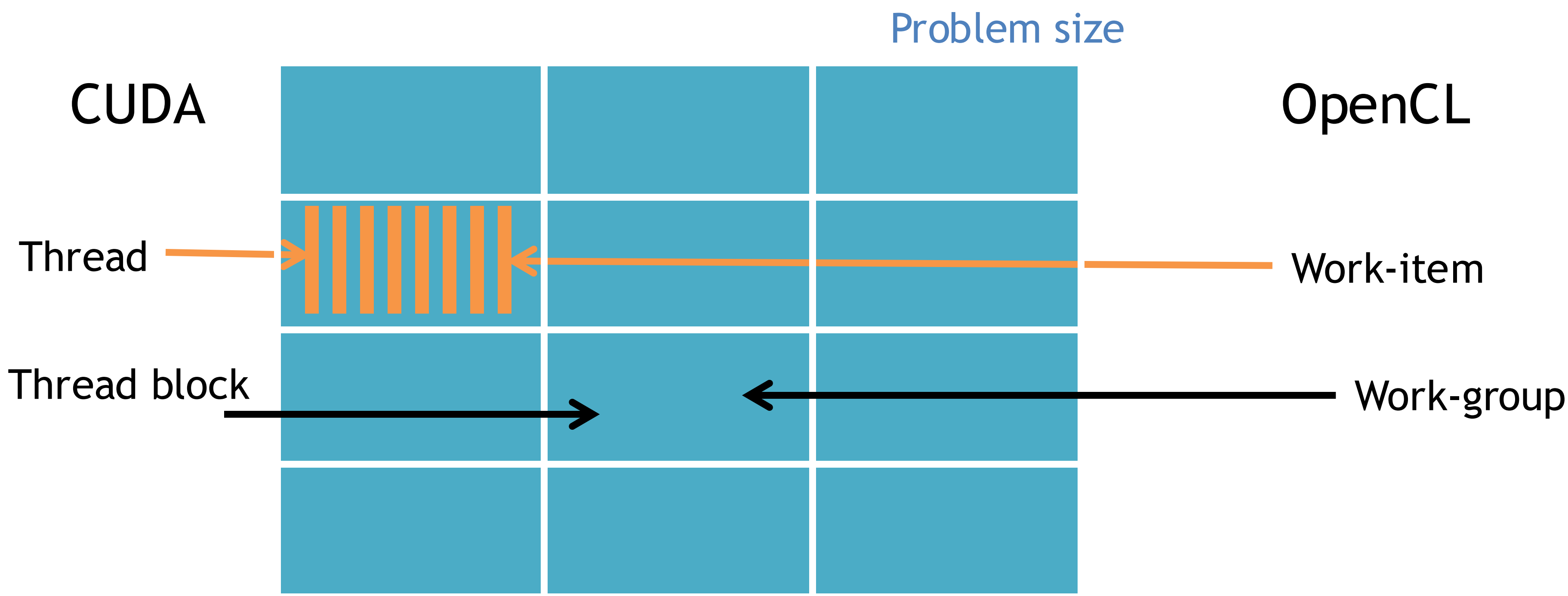
1. Have the kernel accept a local array as an argument

```
__kernel void func(
        __local int *array)
{}
```

2. Specify the size by setting the kernel argument

```
clSetKernelArg(kernel, 0,
   sizeof(int)*num_elements,
   NULL);
```

Dividing up the work



- To enqueue the kernel
  - CUDA – specify the number of thread blocks and threads per block
  - OpenCL – specify the problem size and (optionally) number of work-items per work-group

Enqueue a kernel (C)

CUDA C

```
dim3 threads_per_block(30,20);

dim3 num_blocks(10,10);

kernel<<<num_blocks,

threads_per_block>>>();
```

OpenCL C

```
const size_t global[2] =
                  {300, 200};

const size_t local[2] =
                  {30, 20};

clEnqueueNDRangeKernel(
       queue, &kernel,
       2, 0, &global, &local,
       0, NULL, NULL);
```

Enqueue a kernel (C++)

CUDA C

```
dim3
threads_per_block(30,20);


dim3 num_blocks(10,10);

kernel<<<num_blocks,
  threads_per_block>>>(…);
```

OpenCL C++

```
const cl::NDRange
       global(300, 200);

const cl::NDRange
       local(30, 20);

kernel(
  EnqueueArgs(global, local),
  …);
```

Indexing work

CUDA	OpenCL
gridDim	get_num_groups()
blockIdx	get_group_id()
blockDim	get_local_size()
gridDim * blockDim	get_global_size()
threadIdx	get_local_id()
blockIdx * blockdim + threadIdx	get_global_id()

Differences in kernels

- Where do you find the kernel?
  - OpenCL - either a string (const char *), or read from a file
  - CUDA – a function in the host code
- Denoting a kernel
  - OpenCL - __kernel
  - CUDA - __global__
- When are my kernels compiled?
  - OpenCL – at runtime
  - CUDA – with compilation of host code

Host code

- By default, CUDA initializes the GPU automatically
  - If you needed anything more complicated (multi-device etc.) you must do so manually
- OpenCL always requires explicit device initialization
  - It runs not just on NVIDIA® GPUs and so you must tell it which device(s) to use

Thread Synchronization

CUDA	OpenCL
__syncthreads()	barrier()
__threadfenceblock()	mem_fence(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE)
No equivalent	read_mem_fence()
No equivalent	write_mem_fence()
__threadfence()	Finish one kernel and start another

Translation from CUDA to OpenCL

CUDA	**OpenCL**
GPU	Device (CPU, GPU etc)
Multiprocessor	Compute Unit, or CU
Scalar or CUDA core	Processing Element, or PE
Global or Device Memory	Global Memory
Shared Memory (per block)	Local Memory (per workgroup)
Local Memory (registers)	Private Memory
Thread Block	Work-group
Thread	Work-item
Warp	No equivalent term (yet)
Grid	NDRange

More information

- http://developer.amd.com/Resources/hc/OpenCLZone/programming/pages/portingcudatoopencl.aspx

Exercise 13: Porting CUDA to OpenCL

- Goal:
  - To port the provided CUDA/serial C program to OpenCL
- Procedure:
  - Examine the CUDA kernel and identify which parts need changing
    - Change them to the OpenCL equivalents
  - Examine the Host code and port the commands to the OpenCL equivalents
- Expected output:
  - The OpenCL and CUDA programs should produce the same output – check this!

SOME CONCLUDING REMARKS

Conclusion

- OpenCL has widespread industrial support
- OpenCL defines a platform-API/framework for heterogeneous computing, not just GPGPU or CPU-offload programming
- OpenCL has the potential to deliver portably performant code; but it has to be used correctly
- The latest C++ and Python APIs make developing OpenCL programs much simpler than before
- The future is clear:
  - OpenCL is the only parallel programming standard that enables mixing task parallel and data parallel code in a single program while load balancing across ALL of the platform's available resources.

Other important related trends

- OpenCL's Standard Portable Intermediate Representation (SPIR)
  - Based on LLVM's IR
  - Makes interchangeable front- and back-ends straightforward
  - Now libraries of OpenCL kernels can be distributed in "binary" form, protecting software developer IP

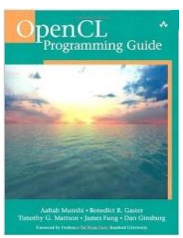
- OpenCL 2.0 adds support for:
  - Shared virtual memory to share addresses between the host and the devices
  - Dynamic (nested) parallelism, enabling kernels to directly enqueue other kernels on the same device without host intervention
  - A formal memory model based on C11
  - A generic address space to enable easier mixing and matching between host/global/local/private
  - Pipes as memory objects
  - Sub-groups to expose warp/wavefront-like hardware features
  - Lots of other improvements!

- For the latest news on SPIR and new OpenCL versions see:
  - http://www.khronos.org/opencl/

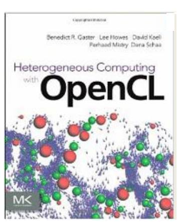
Resources: https://www.khronos.org/opencl/

The OpenCL specification
Surprisingly approachable for a spec!
https://www.khronos.org/registry/cl/

OpenCL reference card
Useful to have on your desk(top)
Available on the same page as the spec.



OpenCL Programming Guide:
Aaftab Munshi, Benedict Gaster, Timothy G. Mattson and James Fung, 2011



Heterogeneous Computing with OpenCL
Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry and Dana Schaa, 2011

Other OpenCL resources

- New annual OpenCL conference
  - http://www.iwocl.org/
  - Held in May each year
  - CFP to be announced at SC

- OpenCL Forums:
  - Khronos' OpenCL forums are the central place to be:
  - http://www.khronos.org/message_boards/forumdisplay.php?f=61

Other OpenCL resources

- CLU: a library of useful C-level OpenCL utilities, such as program initialization, CL kernel code compilation and calling kernels with their arguments (bit like GLUT!):https://github.com/Computing-Language-Utility/CLU

- clMath: an open source BLAS / FFT library originally developed by AMDhttps://github.com/clMathLibraries/clBLAS and https://github.com/clMathLibraries/clFFT

VERSIONS OF OPENCL

OpenCL 1.0

- First public release, December 2008

OpenCL 1.1

- Released June 2010
- Major new features:
  - Sub buffers
  - User events
  - More built-in functions
  - 32-bit atomics become core features

OpenCL 1.2

- Released November 2011
- Major new features:
  - Custom devices and built-in kernels
  - Device partitioning
  - Support separate compilation and linking of programs
  - Greater support for OpenCL libraries

OpenCL 2.0

- Released in November 2013
- Major new features:
  - Shared virtual memory (SVM)
  - Dynamic parallelism
  - Pipes
  - Built-in reductions/broadcasts
  - Sub-groups
  - "generic" address space
  - C11 atomics
  - More image support

Appendix A

VECTOR OPERATIONS WITHIN KERNELS

Before we continue...

- The OpenCL device compilers are good at auto-vectorising your code
  - Adjacent work-items may be packed to produce vectorized code
- By using vector operations the compiler may not optimize as sucessfully
- So ***think twice*** before you explicitly vectorize your OpenCL kernels, you might end up hurting performance!

Vector operations

- Modern microprocessors include vector units:
  Functional units that carry out operations on blocks of numbers
- For example, x86 CPUs have over the years introduced MMX, SSE, and AVX instruction sets …
  characterized in part by their widths (e.g. SSE operates on 128 bits at a time, AVX 256 bits etc)
- To gain full performance from these processors it is important to exploit these vector units
- Compilers can sometimes automatically exploit vector units.
  Experience over the years has shown, however, that you all too often have to code vector operations by hand.
- Example using 128 bit wide SSE:

```
#include "xmmintrin.h "     // vector intrinsics from gcc for SSE (128 bit wide)

__m128 ramp = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);   // pack 4 floats into vector register
__m128 vstep = _mm_load1_ps(&step);              // pack step into a vector register
__m128 xvec; = _mm_mul_ps(ramp,vstep);           // multiple corresponding 32 bit
                                                  //  floats and assign to xvec
```

Vector intrinsics challenges

- Requires an assembly code style of programming:
  - Load into registers
  - Operate with register operands to produce values in another vector register
- Non portable
  - Change vector instruction set (even from the same vendor) and code must be re-written. Compilers might treat them differently too
- Consequences:
  - Very few programmers are willing to code with intrinsics
  - Most programs only exploit vector instructions that the compiler can automatically generate – which can be hit or miss
  - Most programs grossly under exploit available performance.

Solution: a high level portable vector instruction set …
which is precisely what OpenCL provides.

Vector Types

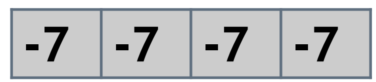
- The OpenCL C kernel programming language provides a set of vector instructions:
  - These are portable between different vector instruction sets
- These instructions support vector lengths of 2, 4, 8, and 16 … for example:
  - **`char2, ushort4, int8, float16, double2`**, …
- Properties of these types include:
  - Endian safe
  - Aligned at vector length
  - Vector operations (elementwise) and built-in functions

Remember, double (and hence vectors of double) are optional in OpenCL v1.1

Vector Operations

- Vector literal

```
int4 vi0 = (int4) -7;
```

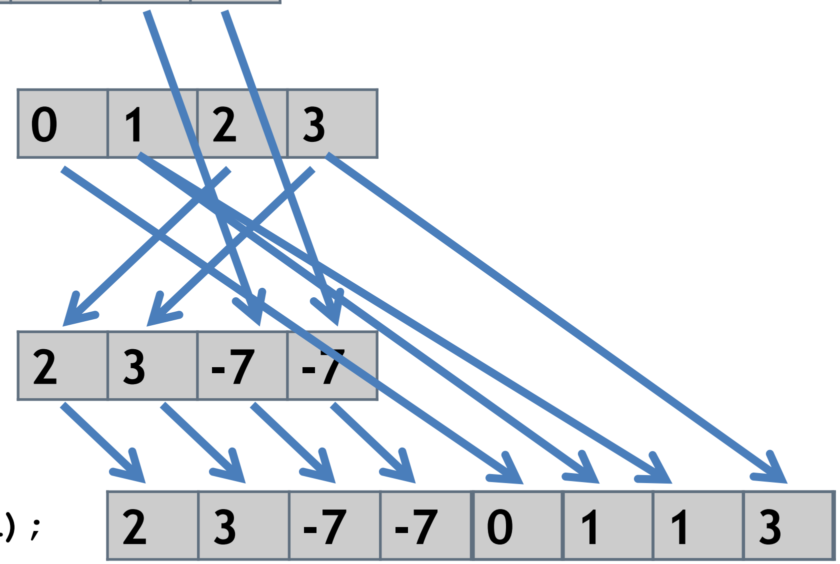



```
int4 vi1 = (int4) (0, 1, 2, 3);
```

- Vector components

```
vi0.lo = vi1.hi;
```

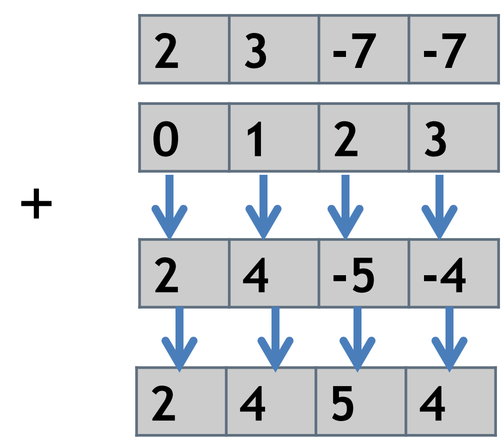
```
int8 v8 = (int8) (vi0, vi1.s01, vi1.odd);
```



- Vector ops

```
vi0 += vi1;
```

```
vi0 = abs(vi0);
```



Using vector operations

- You can convert a scalar loop into a vector loop using the following steps:
  - Based on the width of your vector instruction set and your problem, choose the number of values you can pack into a vector register (the width):
    - E.g. for a 128 bit wide SSE instruction set and float data (32 bit), you can pack four values (128 bits =4*32 bits) into a vector register
  - Unroll the loop to match your width (in our example, 4)
  - Set up the loop preamble and postscript. For example, if the number of loop iterations doesn't evenly divide the width, you'll need to cover the extra iterations in a loop postscript or pad your vectors in a preamble
  - Replace instructions in the body of the loop with their vector instruction counter parts

Vector instructions example

- Scalar loop:

```
for (i = 0; i < 34; i++) x[i] = y[i] * y[i];
```

- Width for a 128-bit SSE is 128/32=4
- Unroll the loop, then add postscript and premable as needed:

```
NLP = 34+2; x[34]=x[35]=y[34]=y[35]=0.0f // preamble to zero pad
for (i = 0; i < NLP; i = i + 4) {
  x[i] = y[i] * y[i];  x[i+1] = y[i+1] * y[i*1];
  x[i+2] = y[i+2] * y[i*2];  x[i+3] = y[i+3] * y[i*3];
}
```

- Replace unrolled loop with associated vector instructions:

```
float4 x4[DIM], y4[DIM];
// DIM set to hold 34 values extended to multiple of 4 (36)
float4 zero = {0.0f, 0.0f, 0.0f, 0.0f};
NLP = 34 % 4 + 1; // 9 values (as 34 isn't a multiple of 4)
x4[NLP-1] = 0.0f; y4[NLP-1] = 0.0f; // zero pad arrays

for (i = 0; i < NLP; i++)
 x4[i] = y4[i] * y4[i]; // actual vector operations
```

Exercise A: The vectorized Pi program

- Goal:
  - To understand the vector instructions in the kernel programming language
- Procedure:
  - Start with your best Pi program
  - Unroll the loops 4 times. Verify that the program still works
  - Use vector instructions in the body of the loop
- Expected output:
  - Output result plus an estimate of the error in the result
  - Report the runtime and compare vectorized and scalar versions of the program
  - You could try running this on the CPU as well as the GPU…

Appendix B

THE OPENCL EVENT MODEL

OpenCL Events

- An event is an object that communicates the status of commands in OpenCL … legal values for an event:
  - CL_QUEUED: command has been enqueued.
  - CL_SUBMITTED: command has been submitted to the compute device
  - CL_RUNNING: compute device is executing the command
  - CL_COMPLETE: command has completed
  - ERROR_CODE: a negative value indicates an error condition occurred.
- Can query the value of an event from the host … for example to track the progress of a command.

Examples:
- CL_EVENT_CONTEXT
- CL_EVENT_COMMAND_EXECUTION_STATUS
- CL_EVENT_COMMAND_TYPE

```
cl_int clGetEventInfo (
          cl_event event,    cl_event_info param_name,
          size_t param_value_size, void *param_value,
          size_t *param_value_size_ret)
```

Generating and consuming events

- Consider the command to enqueue a kernel. The last three arguments optionally expose events (NULL otherwise).

```
cl_int clEnqueueNDRangeKernel (
      cl_command_queue command_queue,
      cl_kernel kernel,
      cl_uint work_dim,
      const size_t *global_work_offset,
      const size_t *global_work_size,
      const size_t *local_work_size,
      cl_uint num_events_in_wait_list,
      const cl_event *event_wait_list,
      cl_event *event)
```

Number of events this command is waiting to complete before executing

Array of pointers to the events being waited upon … Command queue and events must share a context.

Pointer to an event object generated by this command

Event: basic event usage

- Events can be used to impose order constraints on kernel execution.
- Very useful with out-of-order queues.

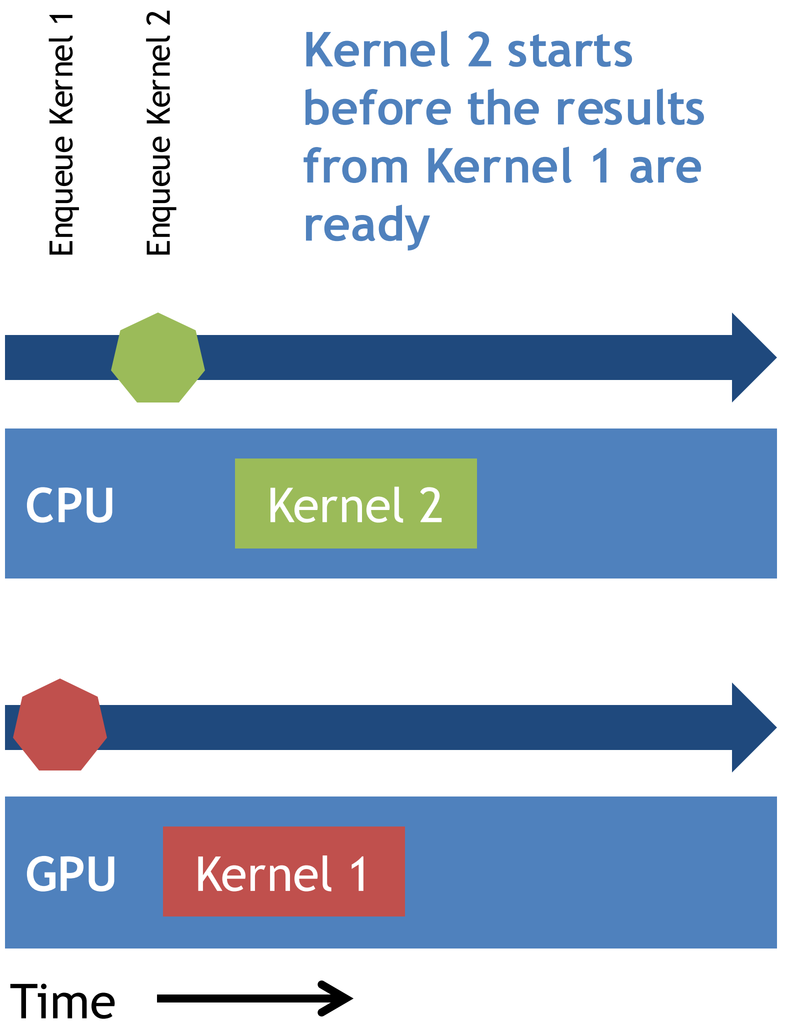
```
cl_event    k_events[2];

err = clEnqueueNDRangeKernel(commands, kernel1, 1,
 NULL, &global, &local, 0, NULL, &k_events[0]);

err = clEnqueueNDRangeKernel(commands, kernel2, 1,
 NULL, &global, &local, 0, NULL, &k_events[1]);

err = clEnqueueNDRangeKernel(commands, kernel3, 1,
 NULL, &global, &local, 2, k_events, NULL);
```

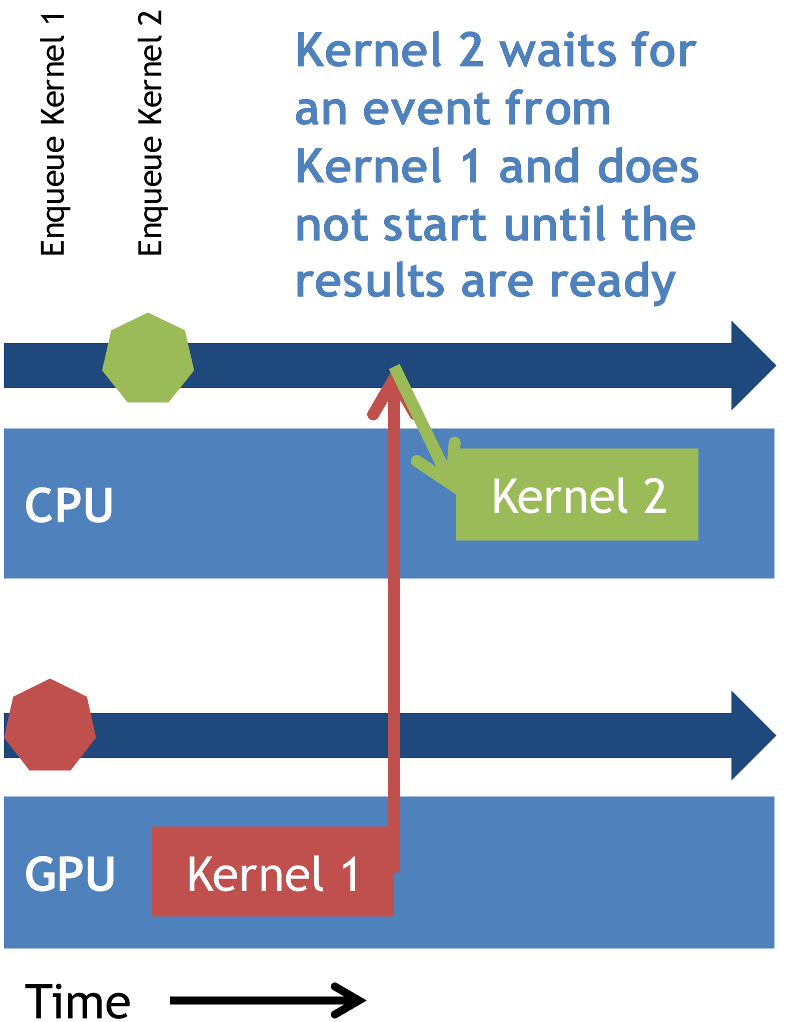
Enqueue two kernels that expose events

Wait to execute until two previous events complete

OpenCL synchronization: queues & events

- Events connect command invocations. Can be used to synchronize executions inside out-of-order queues or between queues
- Example: 2 queues with 2 devices



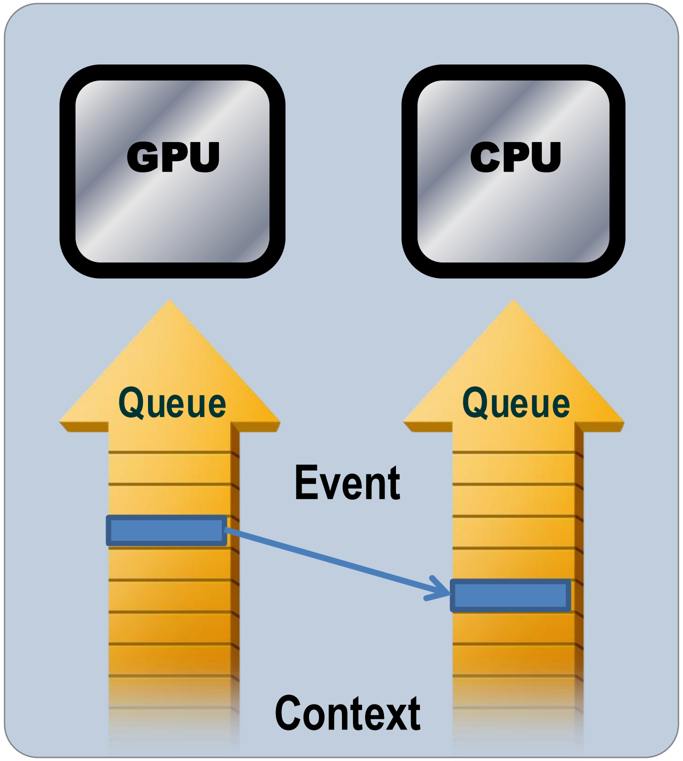


Why Events? Won't a barrier do?

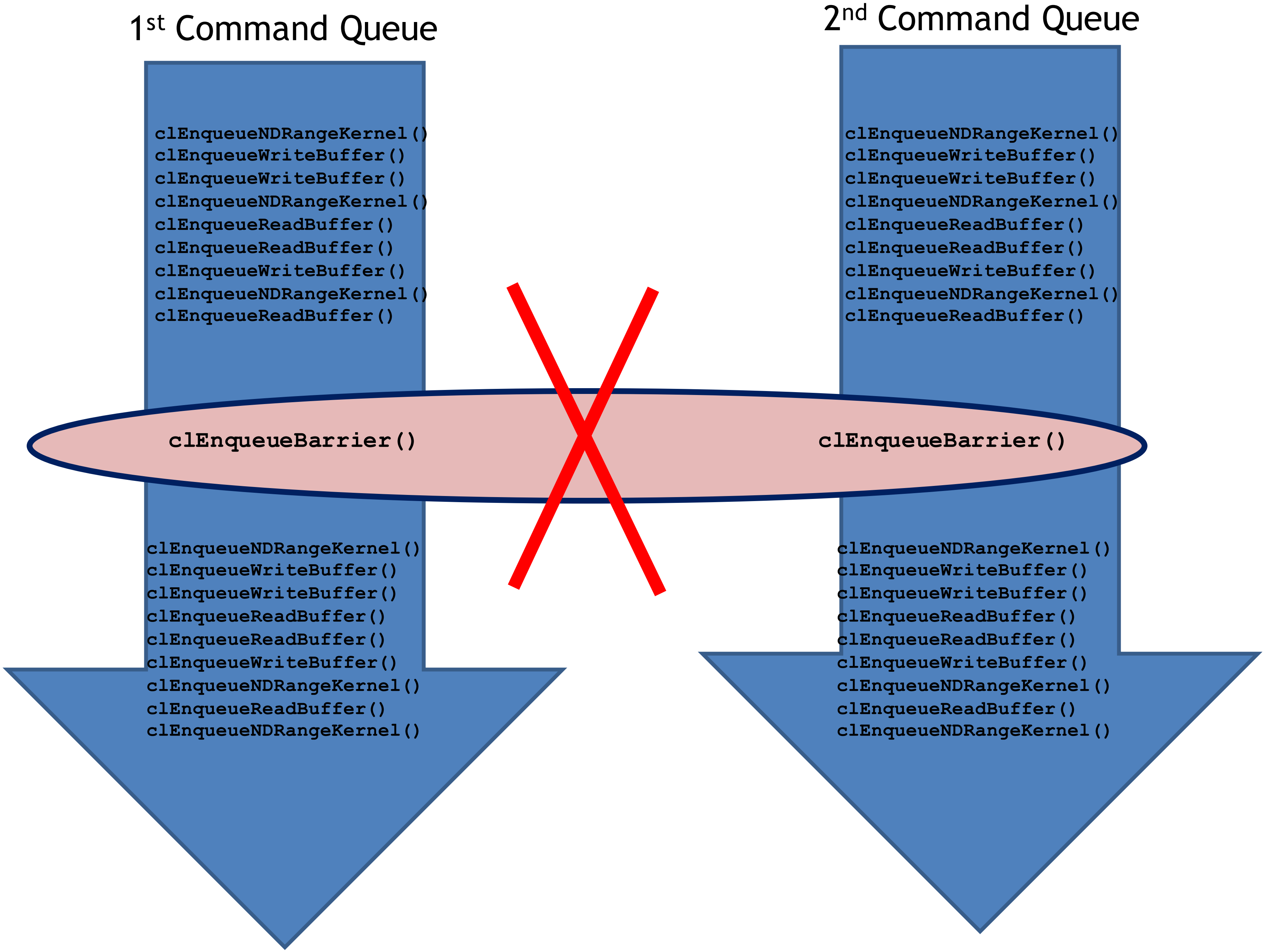
- A barrier defines a synchronization point … commands following a barrier wait to execute until all prior enqueued commands complete

```
cl_int
clEnqueueBarrier(cl_command_queue
queue)
```

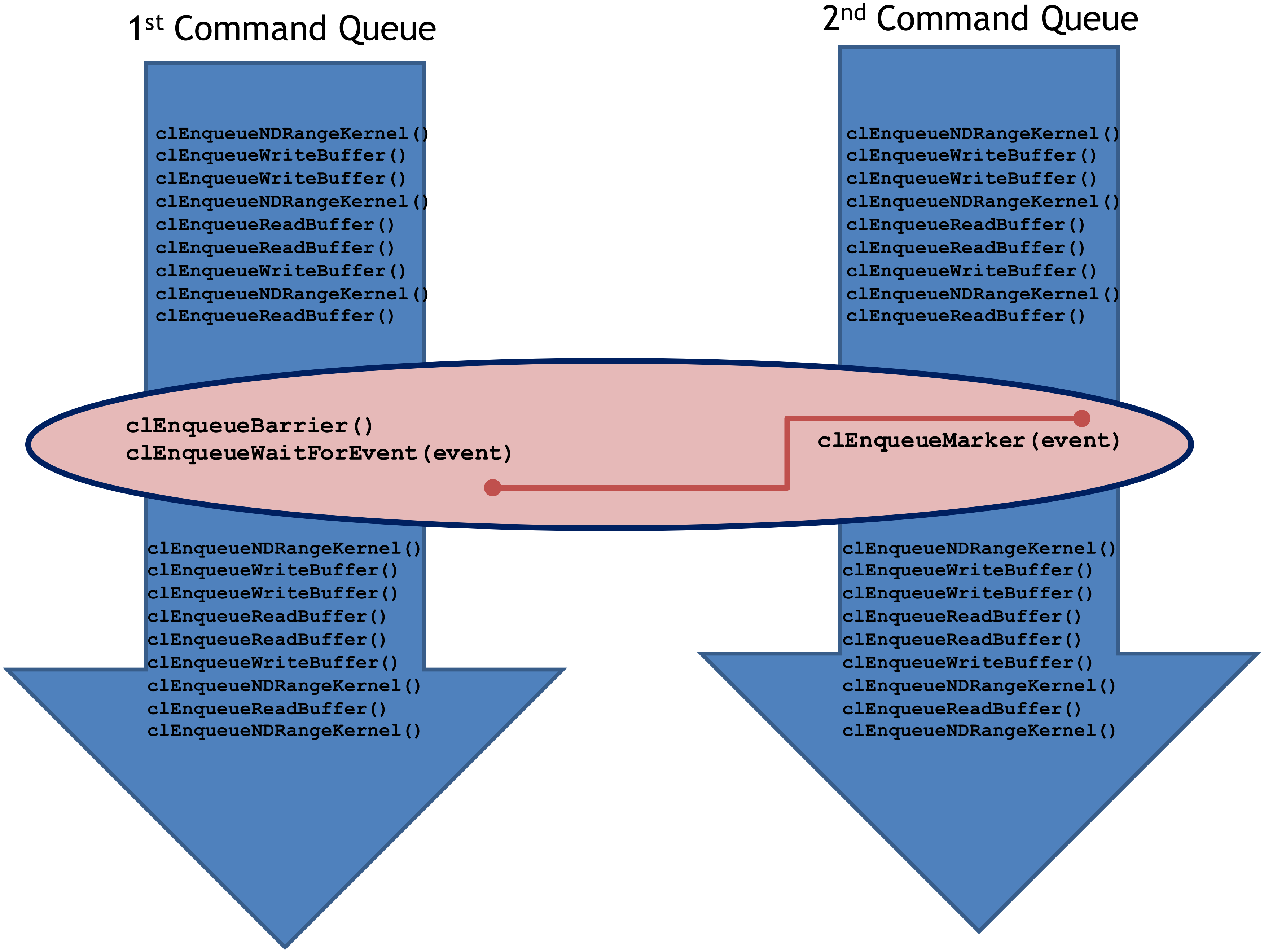
- Events provide fine grained control … this can really matter with an out-of-order queue.
- Events work between commands in the different queues … as long as they share a context
- Events convey more information than a barrier … provide info on state of a command, not just whether it's complete or not.



Barriers between queues: clEnqueueBarrier doesn't work



Barriers between queues: this works!



Host generated events influencing execution of commands: User events

- "user code" running on a host thread can generate event objects

```
cl_event clCreateUserEvent(cl_context context, cl_int *errcode_ret)
```

- Created with value CL_SUBMITTED.
- It's just another event to enqueued commands.
- Can set the event to one of the legal event values

```
cl_int clSetUserEventStatus(cl_event event, cl_int execution_status)
```

- Example use case: Queue up block of commands that wait on user input to finalize state of memory objects before proceeding.

Command generated events influencing execution of host code

- A thread running on the host can pause waiting on a list of events to complete. This can be done with the function:

```
cl_int clWaitForEvents(
    cl_uint num_events,
    const cl_event *event_list)
```

Number of events to wait on

An array of pointers to event object

- Example use case: Host code waiting for an event to complete before extracting information from the event.

Profiling with Events

- OpenCL is a performance oriented language … Hence performance analysis is an essential part of OpenCL programming.
- The OpenCL specification defines a portable way to collect profiling data.
- Can be used with most commands placed on the command queue … includes:
  - Commands to read, write, map or copy memory objects
  - Commands to enqueue kernels, tasks, and native kernels
  - Commands to Acquire or Release OpenGL objects
- Profiling works by turning an event into an opaque object to hold timing data.

Using the Profiling interface

- Profiling is enabled when a queue is created with the CL_QUEUE_PROFILING_ENABLE flag set.
- When profiling is enabled, the following function is used to extract the timing data

```
cl_int clGetEventProfilingInfo(
      cl_event event,
      cl_profiling_info param_name,
      size_t param_value_size,
      void *param_value,
      size_t *param_value_size_ret)
```

Profiling data to query (see next slide) → param_name

Expected and actual size of profiling data. → param_value_size, param_value_size_ret

Pointer to memory to hold results → param_value

cl_profiling_info values

- CL_PROFILING_COMMAND_QUEUED
  - the device time in nanoseconds when the command is enqueued in a command-queue by the host. (cl_ulong)
- CL_PROFILING_COMMAND_SUBMIT
  - the device time in nanoseconds when the command is submitted to compute device. (cl_ulong)
- CL_PROFILING_COMMAND_START
  - the device time in nanoseconds when the command starts execution on the device. (cl_ulong)
- CL_PROFILING_COMMAND_END
  - the device time in nanoseconds when the command has finished execution on the device. (cl_ulong)

Profiling Examples

```
cl_event prof_event;
cl_command_queue comm;

comm = clCreateCommandQueue(
       context, device_id,
      CL_QUEUE_PROFILING_ENABLE,
       &err);

err = clEnqueueNDRangeKernel(
          comm, kernel,
          nd, NULL, global, NULL,
          0, NULL, prof_event);

clFinish(comm);
err = clWaitForEvents(1,
&prof_event );
```

```
cl_ulong start_time, end_time;
size_t return_bytes;

err = clGetEventProfilingInfo(
        prof_event,

CL_PROFILING_COMMAND_QUEUED,
         sizeof(cl_ulong),
         &start_time,
         &return_bytes);

err = clGetEventProfilingInfo(
         prof_event,

CL_PROFILING_COMMAND_END,
         sizeof(cl_ulong),
         &end_time,
         &return_bytes);

run_time =(double)(end_time -
start_time);
```

Events inside Kernels … Async. copy

```
// A, B, C kernel args … global  buffers.
// Bwrk is a local buffer

for(k=0;k<Pdim;k++)
          Awrk[k] = A[i*Ndim+k];

for(j=0;j<Mdim;j++){
    event_t ev_cp  = async_work_group_copy(
        (__local float*) Bwrk, (__global float*) B,
        (size_t) Pdim, (event_t) 0);

    wait_group_events(1, &ev_cp);

    for(k=0, tmp= 0.0;k<Pdim;k++)
           tmp  += Awrk[k] *  Bwrk[k];
    C[i*Ndim+j] = tmp;
}
```

- Compute a row of C = A * B
  - 1 A col.per work-item
  - Work group shares rows of B

Start an async. copy for row of B returning an event to track progress.

Wait for async. copy to complete before proceeding.

Compute element of C using A from private memory and B from local memory.

Events and the C++ interface (for profiling)

- Enqueue the kernel with a returned event

```
Event event =
  vadd(
    EnqueueArgs(commands,NDRange(count), NDRange(local)),
    a_in, b_in, c_out, count);
```

- What for the command attached to the event to complete

  event.wait();

- Extract timing data from the event:

```
  cl_ulong ev_start_time =
    event.getProfilingInfo<CL_PROFILING_COMMAND_START>();

 cl_ulong ev_end_time =
    event.getProfilingInfo<CL_PROFILING_COMMAND_END>();
```

Appendix C

PINNED MEMORY

Pinned Memory

- In general, the fewer transfers you can do between host and device, the better
- But some are unavoidable
- It is possible to speed up these transfers, by using ***pinned memory*** (also called **page-locked** memory)
- If supported, can enable much faster host <-> device communications

Pinned Memory

- A regular enqueueRead/enqueueWrite command might manage ~6GB/s
- But PCI-E Gen 3.0 can sustain transfer rates of up to 16GB/s
- So, where has our bandwidth gone?
- The operating system
- Why? Let's consider when memory is actually allocated…

Malloc Recap

- Consider a laptop which has 16GB of RAM.
- What is the output of the code on the right if run on this laptop?
- Bonus Question: if compiled with **-m32**, what will the output be?

```
#include <stdlib.h>
#include <stdio.h>

int
main
(int argc, char **argv)
{
  //64 billion floats
  size_t len    = 64 * 1024*1024*1024;

  //256GB allocation
  float *buffer =
            malloc(len*sizeof(float));

  if (NULL == buffer)
  {
    fprintf(stderr, "malloc failed\n");
    return 1;
  }

  printf("got ptr %p\n", buffer);
  return 0;
}
```

```
% gcc test.c -o test

% ./test
got ptr 0x7f84b0c03350
```

Malloc Recap

- A non-NULL pointer was returned
- Both OS X and Linux will *oversubscribe* memory
- When will this memory actually get allocated?
- Checking the return value of malloc/calloc is useless – malloc *never** returns NULL!

```
#include <stdlib.h>
#include <stdio.h>

int
main
(int argc, char **argv)
{
  //64 billion floats
  size_t len    = 64 * 1024*1024*1024;

  //256GB allocation
  float *buffer =
            malloc(len*sizeof(float));

  if (NULL == buffer)
  {
    fprintf(stderr, "malloc failed\n");
    return 1;
  }

  printf("got ptr %p\n", buffer);
  return 0;
}
```

* This might not be true for an embedded system

Malloc Recap

- This program does not actually allocate any memory
- We call malloc, but we never use it!

```
#include <stdlib.h>
#include <stdio.h>

int
main
(int argc, char **argv)
{
  size_t len    = 16 * 1024*1024;

  float *buffer =
             malloc(len*sizeof(float));

  return 0;
}
```

Malloc Recap

- So what happens here?
- The pointer we got back, when accessed, will trigger a page fault in the kernel.
- The kernel will then allocate us some memory, and allow us to write to it.
- But how much was allocated in this code? Only 4096 bytes (one page)

```
#include <stdlib.h>
#include <stdio.h>

int
main
(int argc, char **argv)
{
  size_t len    = 16 * 1024*1024;

  float *buffer =
             malloc(len*sizeof(float));

  buffer[0] = 10.0f;

  return 0;
}
```

Malloc Recap

- 4KB pages will be allocated at a time, and can also be swapped to disk dynamically
- In fact, an allocation may not even be contiguous
- So, enqueueRead/enqueueWrite ***must*** incur an additional host memory to host memory copy, wasting bandwidth and costing performance

- **EnqueueWrite**:
  - Allocate contiguous portion of DRAM
  - Copy host data into this contiguous memory
  - Signal the DMA engines to start the transfer
- **EnqueueRead**:
  - Allocate contiguous portion of DRAM
  - Signal DMA engine to start transfer
  - Wait for interrupt to signal that the transfer has finished
  - Copy transferred data from the contiguous memory into memory in the host code's address space

- Pinned memory side-steps this issue by giving the host process *direct* access to the portions of host memory that the DMA engines read and write to.
- This results in much less time spent waiting for transfers!

- Disclaimer: Not all drivers support it, and it makes allocations much more expensive (so it would be slow to continually allocate and free pinned memory!)

Using Pinned Memory

- OpenCL has no official support for pinned memory
- But e.g. NVIDIA supports pinned memory allocations (**CL_MEM_ALLOC_HOST_PTR** flag)
- When you allocate a cl_mem object, you also allocate page-locked host memory of the same size
- But this does not return the host pointer
- Reading and writing data is handled by **enqueueMapBuffer**, which *does* return the host pointer
- Eventually call clEnqueueUnmapMemObject when you're done

```
//create device buffer
cl_mem devPtrA = clCreateBuffer(
  context,
  CL_MEM_ALLOC_HOST_PTR, //pinned memory flag
  len,
  NULL, //host pointer must be NULL
  NULL
);

float *hostPtrA =
(float *) clEnqueueMapBuffer(
  queue,
  devPtrA,
  CL_TRUE, //blocking map
  CL_MAP_WRITE_INVALIDATE_REGION, //write data
  0,         //offset of region
  len,       //amount of data to be mapped
  0, NULL, NULL, //event information
  NULL       //error code pointer
);
```

`CL_MAP_WRITE_INVALIDATE_REGION` is a v1.2 feature; if using v1.1 or earlier, would have to use `CL_MAP_WRITE` instead.

Caveats

- Again, allocating pinned memory is much more expensive (about 100x slower) than regular memory, so frequent allocations will be bad for performance.
- However, frequent reads and writes will be much faster!
- Not all platforms support pinned memory. But, the above method will still work, and at least will not be any slower than regular use

Appendix D

C++ FOR C PROGRAMMERS

C++ for C programmers

- This Appendix shows and highlights some of the basic features and principles of C++.

- It is intended for the working C programmer.

- The C++ standards:
  – ISO/ANSI Standard 1998 (revision 2003)
  – ISO/ANSI Standard 2011 (aka C++0x or C++11)

Comments, includes, and variable definitions

- Single line comments:

```
// this is a C++ comment
```

- C includes are prefixed with “c”:

```
#include <cstdio>
```

- IO from keyboard and to console

```
#include <iosteam>
int a;
std::cin >> a;  // input integer to ‘a’
std::cout << a; // outputs ‘a’ to console
```

Namespaces

- Definitions and variables can be scoped with namespaces.
  :: is used to dereference.
- Using namespace opens names space into current scope.
- Default namespace is std.

```
#include <iostream> // definitions in std namespace
        namespace foo {
             int id(int x) { return x; }
         };
         int x = foo::id(10);
         using namespace std;
         cout << x; // no need to prefix with std::
```

References in C++ …
a safer way to do pointers

- References are non-null pointers. Since they can't be NULL, you don't have to check for NULL value all the time (as you do with C)
- For example, in C we need to write:

```
int foo(int * x)   {
     if (x != NULL) return *x;
     else return 0;
   }
```

- In C++ we could write:

```
 int foo(int & x)   {
          return x;
   }
```

- Note that in both cases the memory address of x is passed (i.e. by reference) and not the value!

New/Delete Memory allocation

- C++ provides safe(r) memory allocation
- new and delete operator are defined for each type, including user defined types. No need to multiple by sizeof(type) as in C.

```
int * x = new int;
    delete x;
```

- For multi element allocation (i.e. arrays) we must use delete[].

```
int * array = new int[100];
delete[] array;
```

Overloading

- C++ allows functions to have the same name but with different argument types.

```
int add(int x, int y)
    {
        return x+y;
    }
    float add(float x, float y)
    {
        return x+y;
    }
    // call the float version of add
    float  f = add(10.4f, 5.0f);
    // call the int version of add
    int i = add(100,20);
```

Classes (and structs)

- **C++ classes are an extension of C structs (and unions) that can functions (called member functions) as well as data.**

```
class Vector {
  private:
     int x_, y_, z_ ;
  public:
      Vector (int x, int y, int z) : x_(x), y_(y), z_(z) {} // constructor

     ~Vector // destructor
       {
            cout << “vector destructor”;
       }
      int getX() const { return x_; } // access member function
      …
 };
```

The keyword “const” can be applied to member functions such as getX() to state that the particular member function will not modify the internal state of the object, i.e it will not cause any visual effects to someone owning a pointer to the said object. This allows for the compiler to report errors if this is not the case, better static analysis, and to optimize uses of the object , i.e. promote it to a register or set of registers.

More information about constructors

- Consider the constructor from the previous slide …

```
Vector (int x, int y, int z): x_(x), y_(y), z_(z) {}
```

- C++ member data local to a class (or struct) can be initialized using the noation

```
: data_name(initializer_name), ...
```

- Consider the following two semantically equivalent structs in which the constructor sets the data member x_ to the input value x:

A

```
struct Foo
{
  int x_;
  Foo(int x) : x_(x) {}
}
```

B

```
struct Foo
{
  int x_;
  Foo(int x) { x_ = x; }
}
```

- Case B must use a temporary to read the value of x, while this is not so for Case A. This is due to C's definition of local stack allocation.
- This turns out to be very import in C++11 with its memory model which states that an object is said to exist once inside the body of the constructor and hence thread safety becomes an issue, this is not the case for the constructor initalization list (case A). This means that safe double locking and similar idioms can be implemented using this approach.

Classes (and structs) continued

- Consider the following block where we construct an object (the vector “v”), use it and then reach the end of the block

```
{
   Vector v(10,20,30);
   // vector {x_ = 10, y_ = 20 , z_ = 30}
   // use v
} // at this point v’s destructor would be called!
```

- Note that at the end of the block, v is no longer accessible and hence can be destroyed. At this point, the destructor for v is called.

Classes (and structs) continued

- There is a lot more to classes, e.g. inheritance but it is all based on this basic notion.

- The previous examples adds no additional data or overhead to a traditional C struct, it has just improved software composibility.

Function objects

- Function application operator can be overloaded to define functor classes

```
struct Functor
  {
      int operator() (int x) { return x*x; }
  };
  // create an object of type Functor
  Functor f();
  int value = f(10); // call the operator()
```

Template functions

- Don’t want to write the same function many times for different types?
- Templates allow functions to be parameterized with a type(s).

```
template<typename T>
    T add(T x, T y) { return x+y; }

    float  f = add<float>(10.4f, 5.0f); // float version
    int i = add<int>(100,20);           // int version
```

- You can use the templatized type, T, inside the template function

Template classes

- Don’t want to write the same class many times for different types?
- Templates allow class to be parameterized with a type(s) too.

```
template <typename T>
    class Square
    {
        T operator() (T x) { return x*x; }
    };
    Square<int> f_int();
    int value = f_int(10);
```

C++11 defines a function template

- C++ function objects can be stored in the templated class std::function. The following header defines the class std::function

```
#include <functional>
```

- We can define a C++ function object (e.g. functor) and then store it in the tempated class std::function

```
struct Functor
   {
      int operator() (int x) { return x*x; }
   };
   std::function<int (int)> square(Functor());
```

C++ function template: example 1

The header <functional> just defines the template std::function. This can be used to warp standard functions or function objects, e.g.:

```
int foo(int x)  { return x; }  // standard function
std::function<int (int)> foo_wrapper(foo);

struct Foo // function object
{
  void operator()(int x)  {return x;}
};
 std::function<int (int)> foo_functor(Foo());
```

foo_functor and foo_wrapper are basically the same but one is using a standard C like function, while the other is using a function object

C++ function template: example 2

What is the point of function objects? Well they can of course contain local state, which functions cannot, they can also contain member functions and so on. A silly example might be:

```
struct Foo // function object
{
  int y_;
  Foo() : y_(100) {}

  void operator()(int x) { return x+100; }
};

std::function<int (int)> add100(Foo());
// function that adds 100 to its argument
```

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