Hands On OpenCL

Created by
Simon McIntosh-Smith
and Tom Deakin

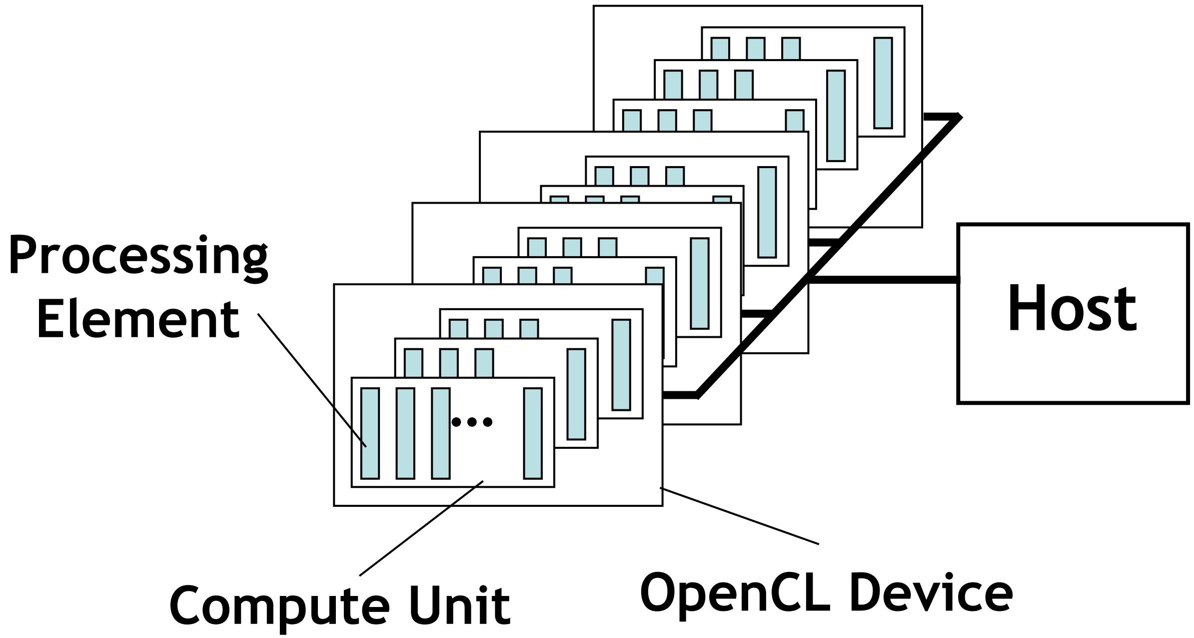
Includes contributions from:
Timothy G. Mattson (Intel) and Benedict Gaster (Qualcomm)

V 1.2 – Nov 2014

Lecture 3

IMPORTANT OPENCL CONCEPTS

OpenCL Platform Model



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

The BIG idea behind OpenCL

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

Traditional loops

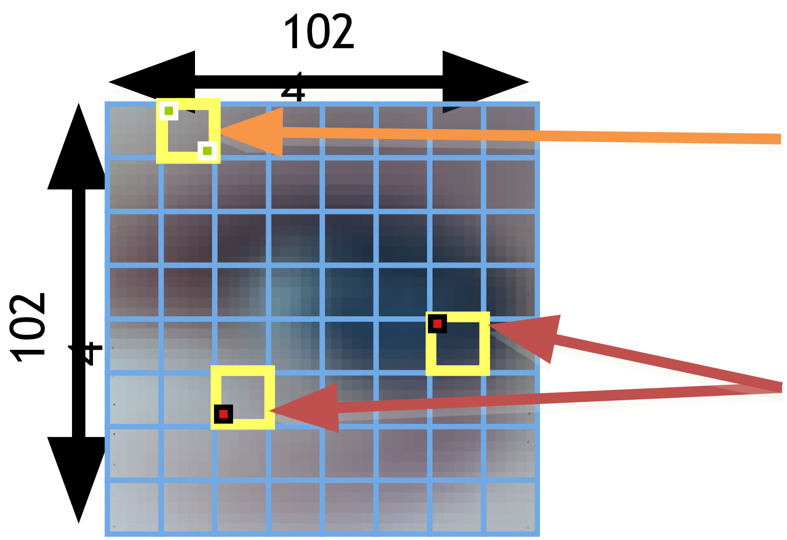
```
void
mul(const int n,
    const float *a,
    const float *b,
          float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

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

An N-dimensional domain of work-items

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



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

Cannot synchronize between work-groups within a kernel

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

OpenCL N Dimensional Range (NDRange)

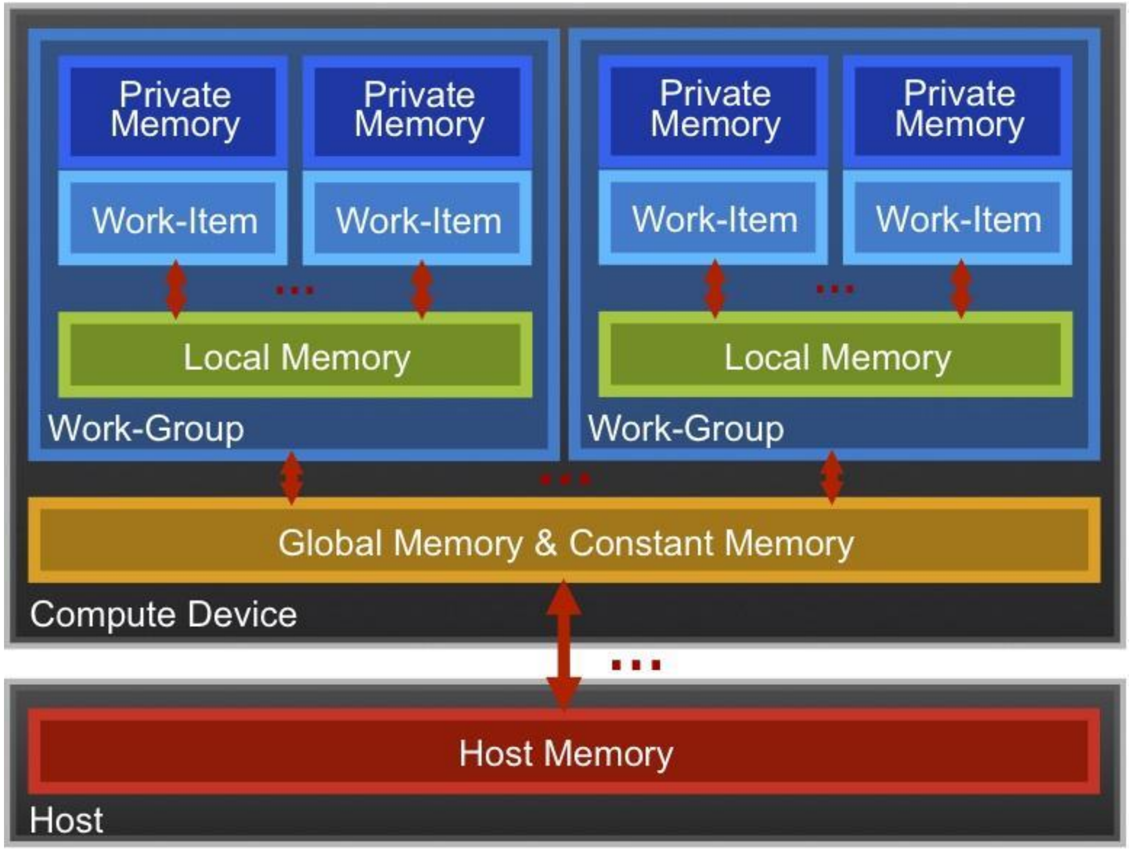
- The problem we want to compute should have some **dimensionality**;
  - For example, compute a kernel on all points in a cube
- When we execute the kernel we specify **up to 3 dimensions**
- We also **specify the total problem size** in each dimension – this is called the **global** size
- We associate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

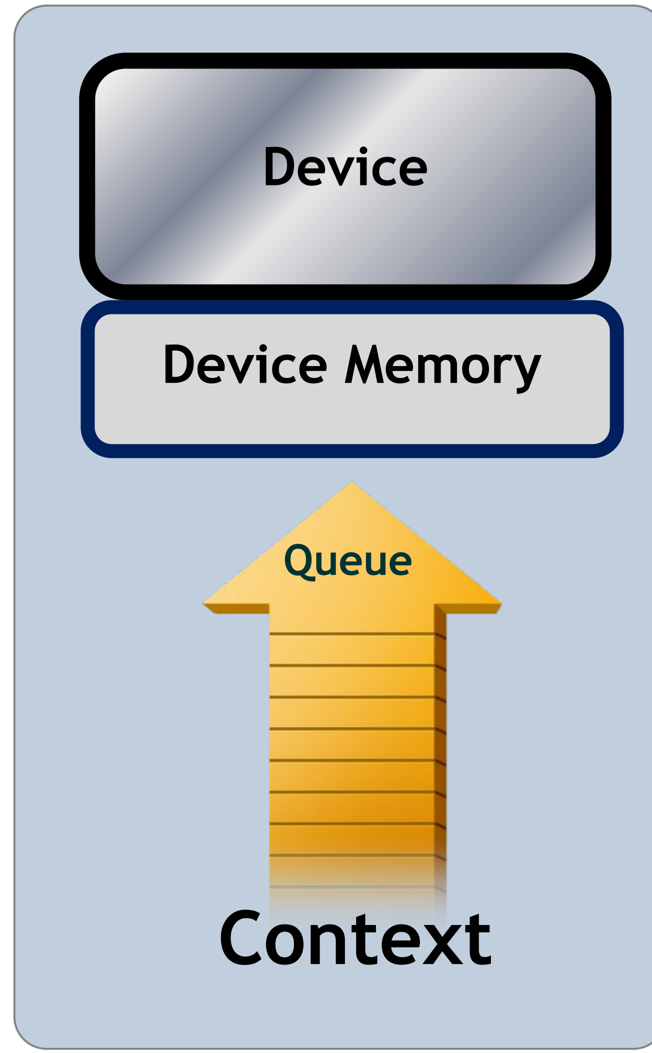
- *Private Memory*
  - Per work-item
- *Local Memory*
  - Shared within a work-group
- *Global Memory /Constant Memory*
  - Visible to all work-groups
- *Host memory*
  - On the CPU



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

Context and Command-Queues

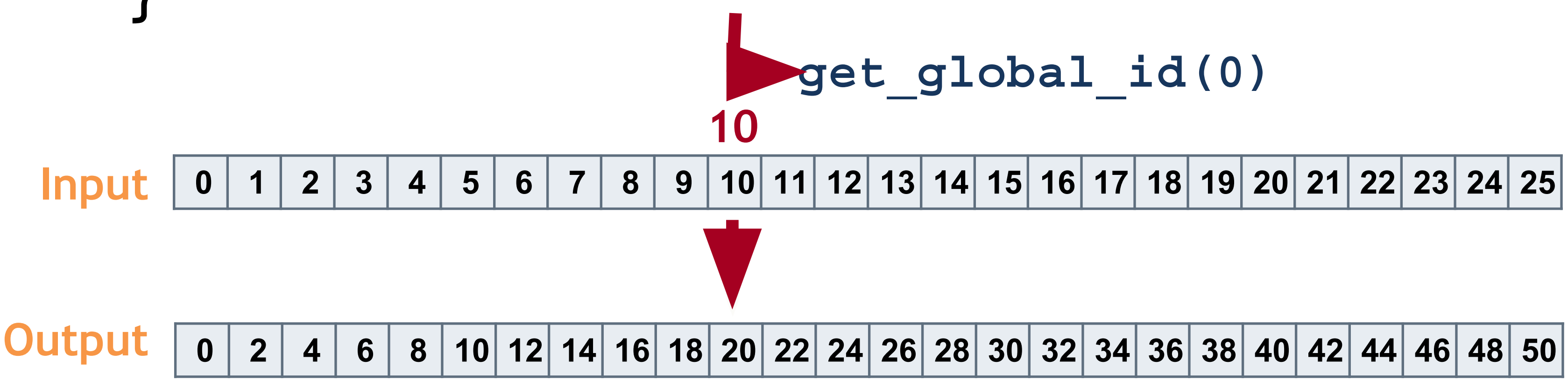
- ***Context***:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization,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

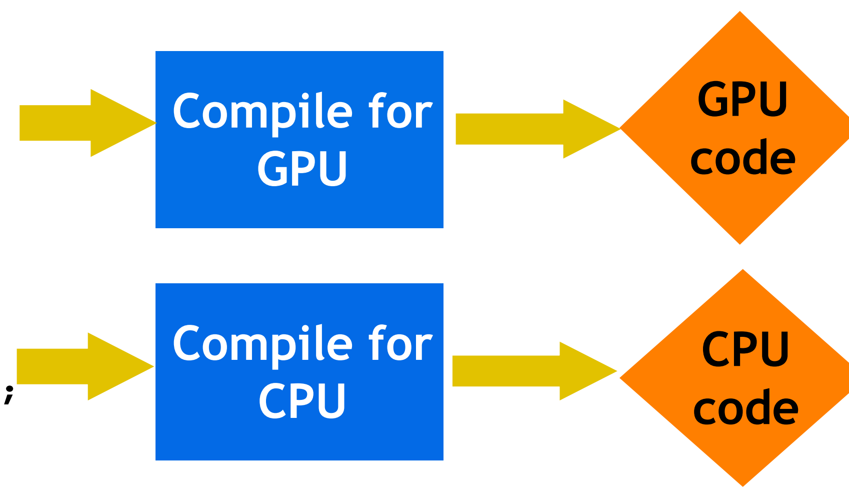


Building Program Objects

- 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

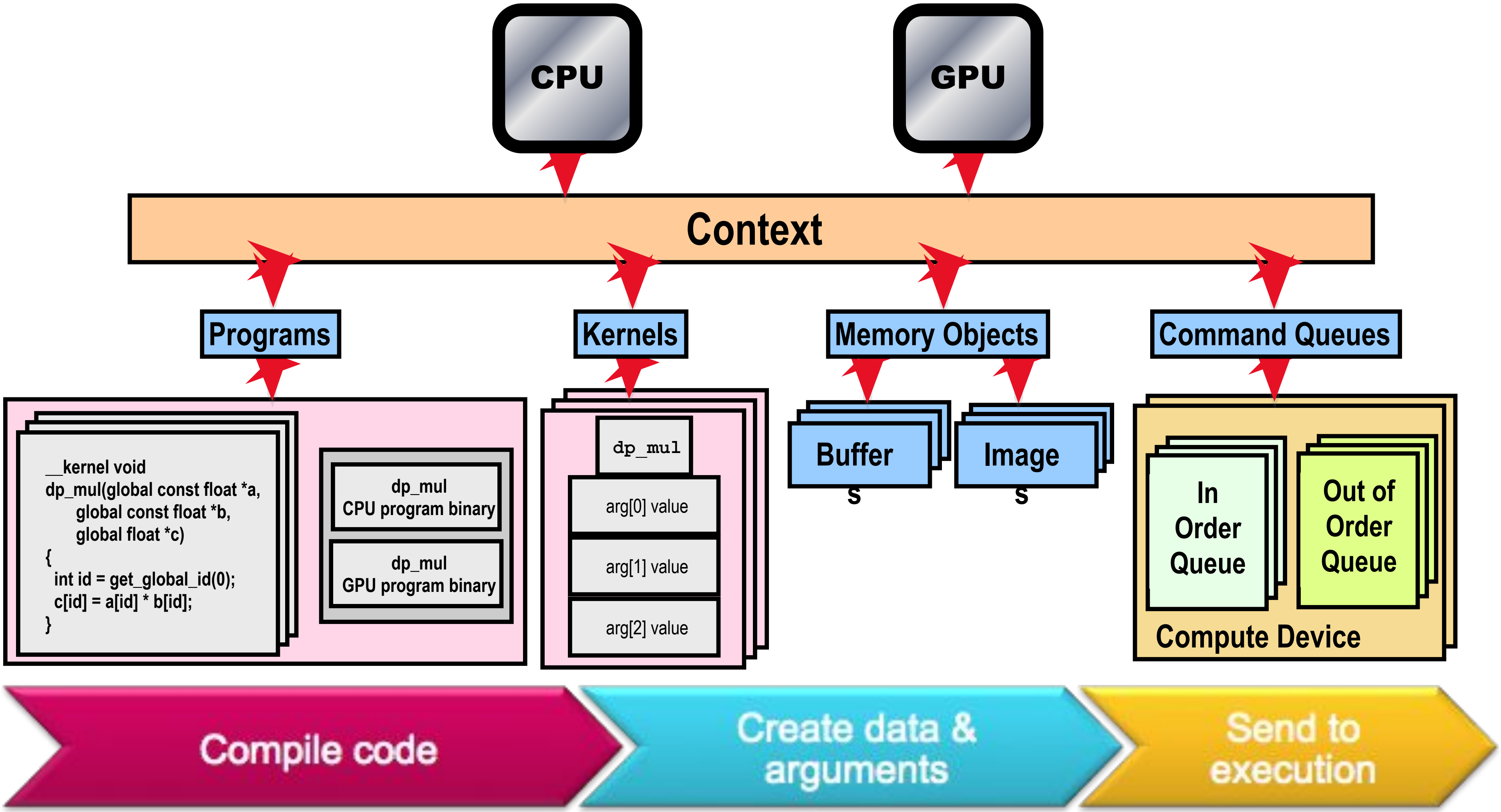
Vector Addition - Kernel

```
__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

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:

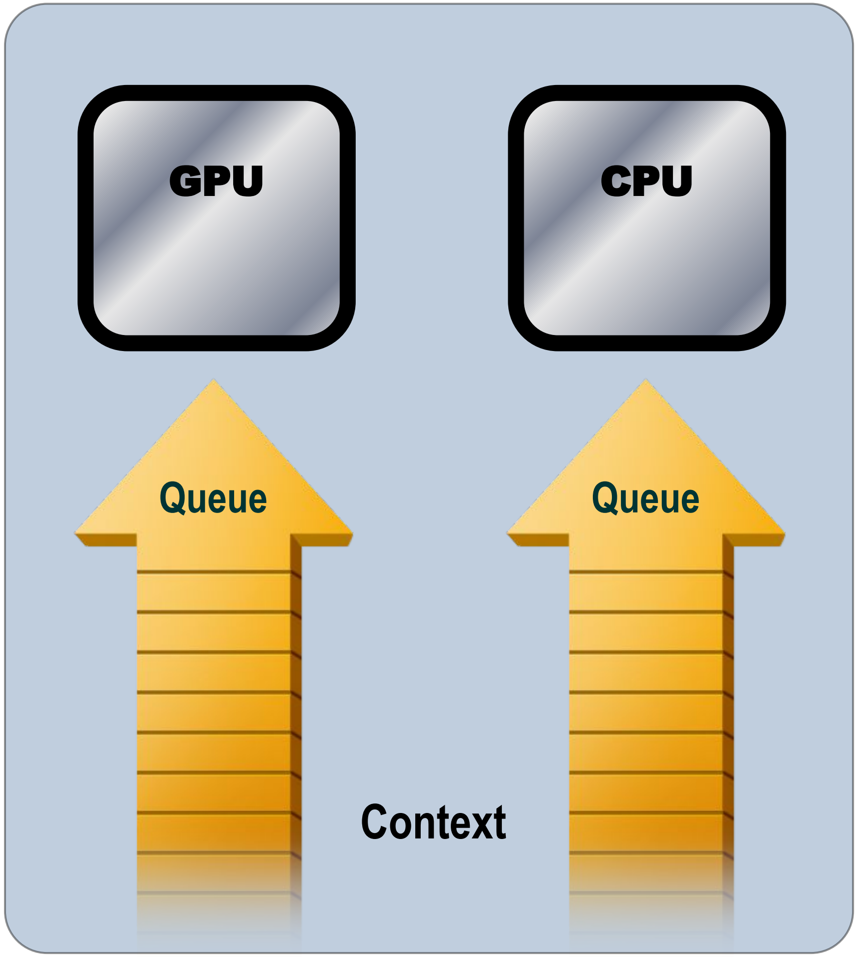
```
context = clCreateContext(firstPlatformId, 1,
                      &device_id, NULL, NULL, &err);
```

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

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

Command-Queues

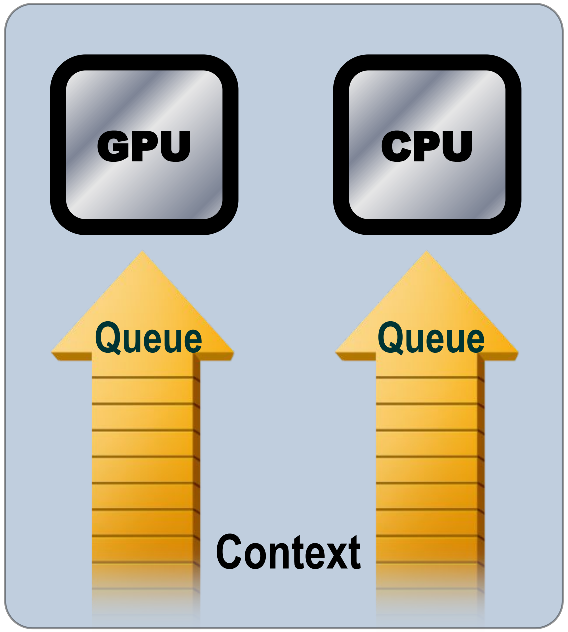
- Commands include:
  - Kernel executions
  - Memory object management
  - Synchronization
- The only way to submit commands to a device is through a command-queue.
- Each command-queue points to a single device within a context.
- Multiple command-queues can feed a single device.
  - Used to define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

Define platform and queues

```
cl_dev
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 |
        C                                        , NULL);
```

Define memory objects

```
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);
```

Create the program

```
// build the
err = clBuild                               ,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                                   bjs[1],

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 = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
                                            0,NULL,NULL);
```

Execute the kernel

```
// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
                          CL_TRUE, 0,
                                             st,
```

Read results on the host

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

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

or `cl2.hpp`

[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

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.

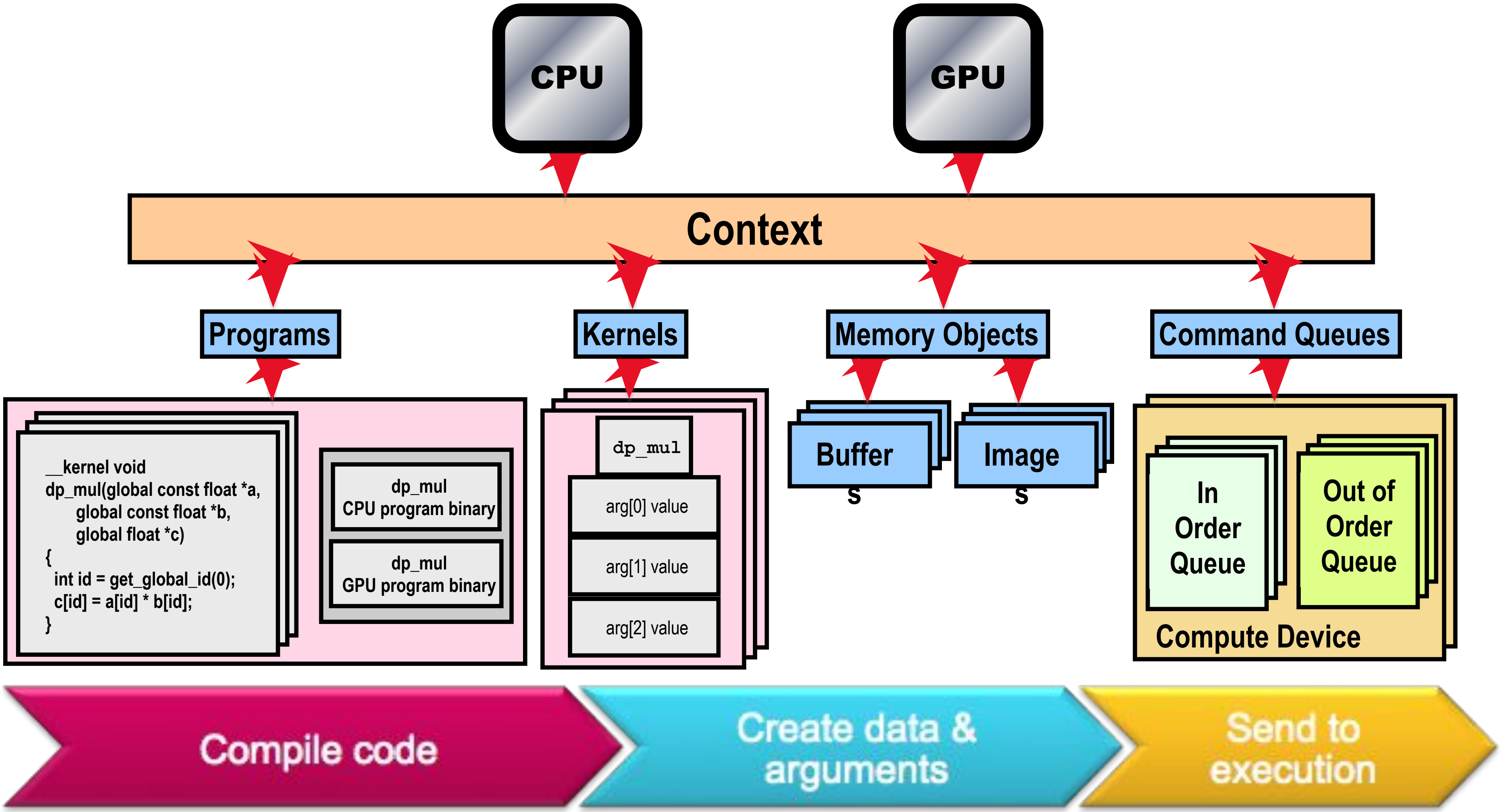
Associate with args and enqueue kernel

- 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);
```

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.

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

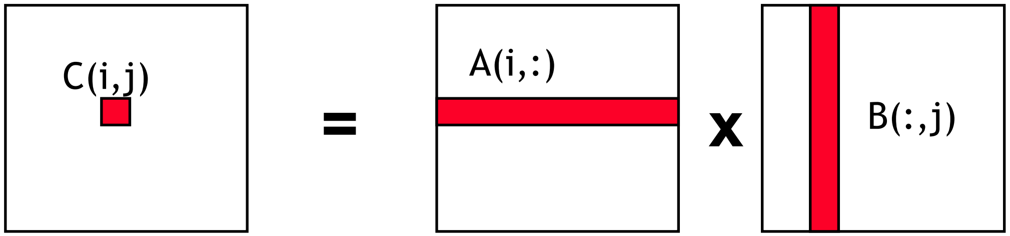
- We solve for x, given a problem Ax=b
  - $Ax=b$ $\quad$ $LUx=b$
  - $Ux=(L^{-1})b$ $\quad$ $x = (U^{-1})(L^{-1})b$

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_                 iming data
  cl::Program p

  Ndim = Pdim =
  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);
```

Declare and initialize data

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

Setup the platform and build program

```
  // Set                                       ,
  // 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

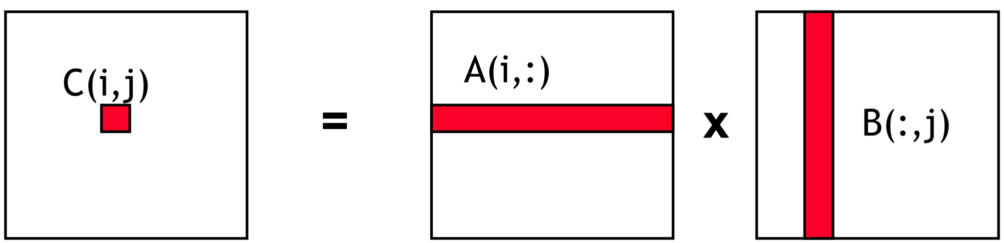
These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

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.

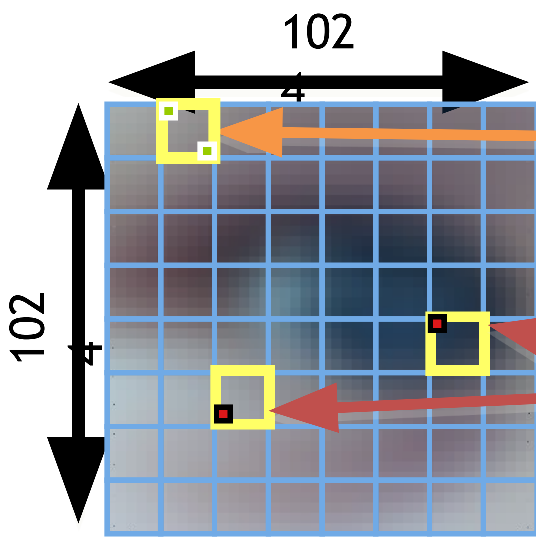


Dot product of a row of A and a column of B for each element of C

- 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)



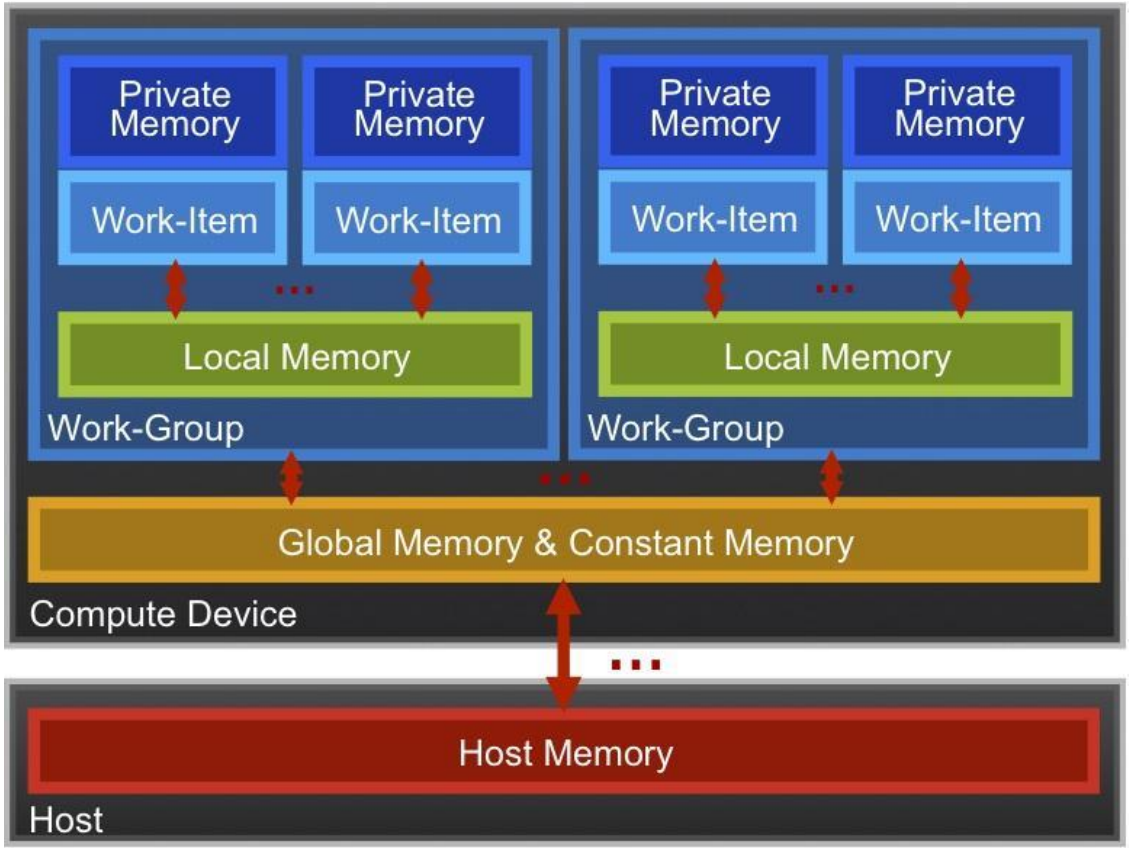
Synchronization between work-items possible only within work-groups: barriers and memory fences

Cannot synchronize between work-groups within a kernel

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

OpenCL Memory model

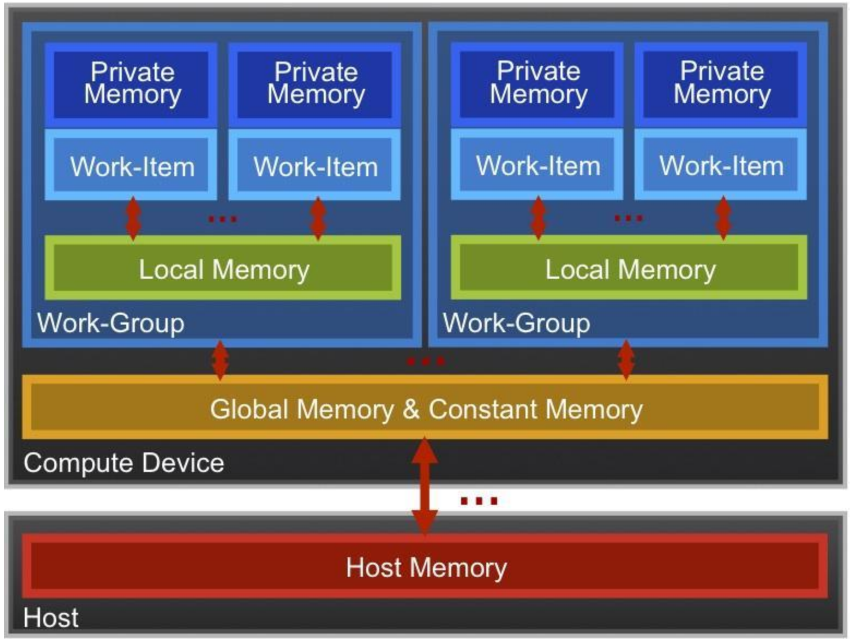
- Private Memory
  - Per work-item
- Local Memory
  - Shared within a work-group
- Global/Constant Memory
  - Visible to all work-groups
- Host memory
  - On the CPU



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



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	Sizes
Private memory O(2-3) words/cycle/WI	**Private memory** O(10) words/WI
Local memory O(10) words/cycle/WG	**Local memory** O(1-10) KBytes/WG
Global memory O(100-200) GBytes/s	**Global memory** O(1-10) GBytes
Host memory O(1-100) GBytes/s	**Host memory** O(1-100) GBytes

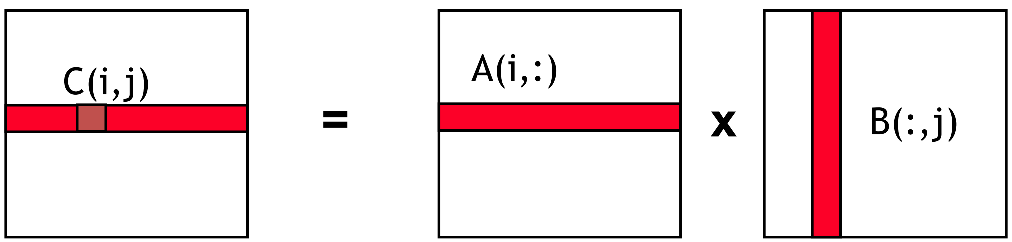
Speeds and feeds approx. for a high-end discrete GPU, circa 2011

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

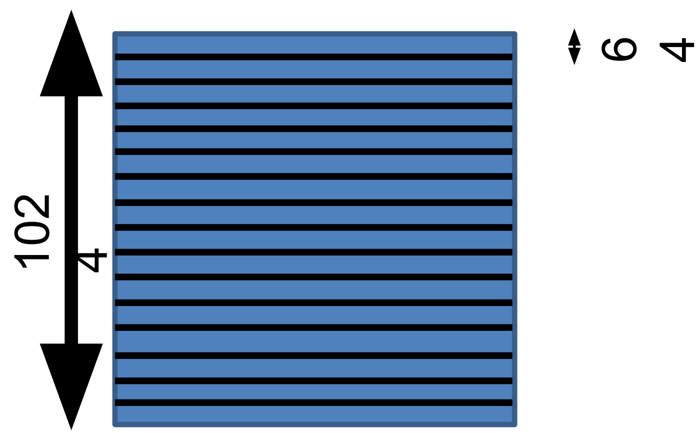


Dot product of a row of A and a column of B for each element of C

- 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.

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

Biggest impact so far!

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

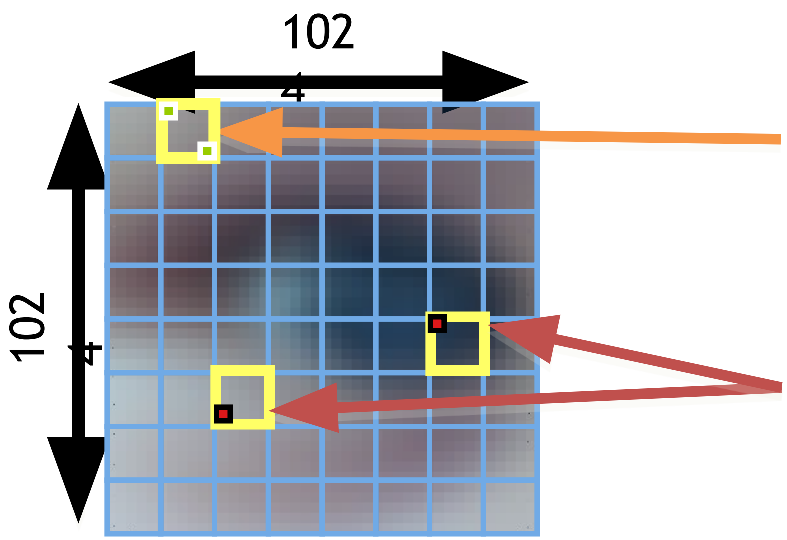
These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Lecture 7

SYNCHRONIZATION IN OPENCL

Consider N-dimensional domain of work-items

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



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

Cannot synchronize between work-groups within a kernel

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

Work-Item Synchronization

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

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

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

Where might we need synchronization?

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

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

Simple parallel reduction

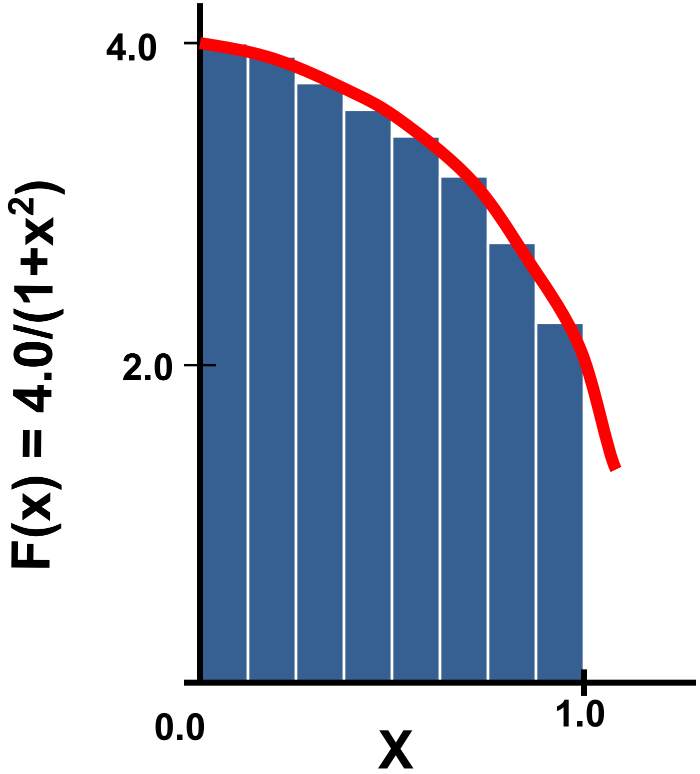
A reduction can be carried out in three steps:

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

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

A simple program that uses a reduction

Numerical Integration



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

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

Numerical integration source code

The serial Pi program

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

  step = 1.0/(double) num_steps;

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

The Pi kernels

```
__kernel void pi(
   const int          niters,
   const float        step_size,
   __local  float*    local_sums,
   __global float*    partial_sums)
{
   int num_wrk_items  = get_local_size(0);
   int local_id       = get_local_id(0);
   int group_id       = get_group_id(0);

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

   istart = (group_id * num_wrk_items + local_id) * niters;
   iend   = istart+niters;

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

   local_sums[local_id] = accum;
   barrier(CLK_LOCAL_MEM_FENCE);

   reduce(local_sums, partial_sums);
}
```

```
void reduce(
   __local  float*    local_sums,
   __global float*    partial_sums)
{
   int num_wrk_items  = get_local_size(0);
   int local_id       = get_local_id(0);
   int group_id       = get_group_id(0);

   float sum;
   int i;

   if (local_id == 0) {
      sum = 0.0f;

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

      partial_sums[group_id] = sum;
   }
}
```

There are smarter ways to do this using more than 1 thread.

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