In case of Fire Alarm
• Intros – who we are
• GPU computing
• GPUs in the CSF
  – Exercise 0, Exercise 1 (parts 1 & 2)
• OpenCL basics
• Writing an OpenCL Program (Host Code)
  – Exercise 1 (part3) & Exercise 2
• OpenCL C and Kernels
• Memory and Kernels
• Synchronization
  – Exercise 3
• Synchronization continued
  – Exercise 4
• Miscellaneous
• Calling OpenCL from Fortran
Many sources and previous presentations on OpenCL have inspired this course, have been read, digested, borrowed. You are encouraged to look at these.

– Intro to OpenCL, ISC2011, Tim Mattson, Intel, Udeepa Bordoloi, AMD
– Intro to OpenCL, HECToR Training Course, Simon McIntosh-Smith, Bristol
– Intro to OpenCL, Training Guide, AMD
– Intro To OpenCL, NAG Training Course, Jason Wood
– The Nvidia OpenCL documents (see GPU Computing SDK)
Research Computing

IT Services
University of Manchester

http://www.manchester.ac.uk/researchcomputing
Supporting Research Computing

– Directorate of IT Services
– Part of a virtual team with cross-faculty research computing support

Help with Computational and Data Resources

– Contribution-based access to managed compute resources (CSF) including GPUs
– Support & access to high throughput computing facilities (EPS Condor, Amazon EC2)
– Support & access to national and international HPC and e-Infrastructure Services (HECToR, DEISA, TeraGrid)
– Data Analytics including Research Data Management
– Support with use of emerging technologies and techniques
What is Research Computing?

- **Computing as part of research**
  - Running complex simulations
  - Performing vast parameter searches
  - Use of integrated systems to manage computational research

- **High Performance Computing (HPC)**
  - 100s-1000s of CPUs to run complex models quicker
  - 100s-1000s of GB of memory to handle very big jobs

- **High Throughput Computing (HTC)**
  - Computational experiment run many (1000s) of times with varying input

- **Data Intensive Computing & Data Analytics**
  - Management of, and extracting pertinent information from, vast quantities of data
Helping researchers

- **Training and Support**
  - Friendly staff on hand to advise on research computing issues
  - Comprehensive training course programme (free to attend)
  - [http://www.rcs.manchester.ac.uk/courses](http://www.rcs.manchester.ac.uk/courses)

- **In-depth Support**
  - Initial analysis and scoping, leading to programme of work

- **Collaborative Support**
  - Effort on extracting maximum performance/tackling larger research questions.
    Named resource/researchers on RCUK/EU grants.
  - Help with input to research proposals.

its-research@manchester.ac.uk
GPU Computing
Modern systems include
- Multi-core CPUs: ~16 cores
  - *Intel Knights Corner (MIC architecture)* +50 cores
- GPUs: 100s cores (more later)
  - *CSF: M2050/M2070s 448 core*
- APUs: CPU+GPU
  - *AMD Fusion*
  - *Nvidia + ARM64 (Project Denver)*
- Other devices? (IBM Cell, DSP, FPGAs?)

Why more cores?
- Split over N cores to get results N times quicker (strong scaling)
- Split over M cores to for M times larger problems (weak scaling)
- High Throughput Computing – repeat with different params
- Wish to run floating-point intensive programs
  - CPUs: <25% logic devoted to FP.
    - SSE, AVX? Compiler flags? OpenMP?
    - Caches for reducing latency
  - GPUs: Games, graphics apps? (FP intensive)
  - Other devices (some low-level language?)
- Exploit these heterogeneous resources

**OpenCL – Open Computing Language**

Open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. Gives portable and efficient access to heterogeneous resources.

Add two vectors of \( n \) floats: \[ z = x + y \quad \text{ (large \( n \))} \]

- Exploit the fact that iteration \( i \) independent of \( i+1 \)

```c
void arrayAddCPU( int n, const float *x, const float *y, float *z )
{
    for ( int i=0; i<n; i++ )
        z[i] = x[i] + y[i];
}
```
Vector Processing in CPU

- Extensions added to CPUs: SSE\textit{n}, AVX
  - H/W vector units: do more FP ops in parallel
- Must have \textit{data parallelism} / independence
- Single Instruction Multiple Data (SIMD)
  - OPs performed in \textit{lock-step}

```c
#include <xmmintrin.h>
void arrayAddSSE( int n, const float *x, const float *y, float *z )
{
    __m128 sse_x, sse_y, sse_z;
    for (int i=0; i<n/4; i++) {
        sse_x = _mm_load_ps(x);
        sse_y = _mm_load_ps(y);
        sse_z = _mm_add_ps(sse_x, sse_y);
        _mm_store_ps(z, sse_z);
        x+=4; y+=4; z+=4;
    }
}
```

Based on SSE code by Jacques du Triot, NAG.

Introduction to OpenCL
Use OpenMP on multi-core CPU

- Each *thread* works on a chunk of the array
- Single Program Multiple Data (SPMD) – not in *lock-step*

```c
#include <omp.h>

void void arrayAddOMP( int n, const float *x, const float *y, float *z )
{
    int i;
    #pragma omp parallel for private(i) shared(n,x,y,z)
    for ( i=0; i<n; i++ )
    {
        z[i] = x[i] + y[i];
    }
}
```
Why all the talk of GPUs?

- Graphics rendering: highly data parallel
- Transform geometry: E.g., 4x4 matrix mult with n vertices
  - A vertex is [x, y, z, 1.0] – a vector of floats
- Manipulate fragments / pixels: E.g., 1280x1024 pixels
  - A pixel is [r, g, b, a] – a vector of floats
- GPU h/w: vector processors
Early GPU Programming

- Fixed function pipeline became programmable
  - Write shaders from point of view of processing a single vertex or pixel
  - Run in parallel on all vertices or pixels
    
    ```
    // Simple 'wavey' vertex shader
    void main(void) {
        vec4 v = vec4(gl_Vertex); // Get current vertex
        v.x = 0.5 * sin(2.0*v.y); // Makes the object wavey
        v.z = 0.5 * cos(2.0*v.y);
        // Set shader return value by doing fixed-function projection
        gl_Position = gl_ModelViewProjectionMatrix * v;
    }
    ```

- Re-use vertices, colours, fragments etc as general purpose data
  - Your "graphics" code now does general purpose compute
    - E.g., BrookGPU (2004, Stanford)
    - See http://www.gpgpu.org
  - Idea taken up by h/w manufacturers
    - ATI Stream SDK, Nvidia CUDA (~2006)
View GPU as vector processor or a multi-core processor?
- E.g.: Nvidia Fermi architecture (GeForce, Quadro and Tesla)

- 512 cores arranges as:
  - 16 x Streaming Multiprocessors
    - An SM is a SIMD vector processor
  - 32 cores per SM
  - 16 x 32 = 512 cores
  - Each SM has
    - scheduler / dispatch
    - Register file, cache: L1+shm (64KB), L2 (768KB)

Code runs on cores
- For now think of having $n$ cores
- Keep in mind the vector units
Summary: GPUs suited to highly data-parallel problems

- Many cores arranged as vector processing h/w
- Programmable
- Fast memory (>150GB/s)
The OpenCL way: replace loops with a kernel

- A *kernel* is run on every data item in parallel on the device (GPU)
  - *Similarities with the previous vector and shader examples!*

```c
void arrayAddCPU( const int n,
    const float *x,
    const float *y,
    float *z )
{
    int i;
    for ( i=0; i<n; i++ )
        z[i] = x[i] + y[i];
}

__kernel void arrayAddOCL(
    __global const float *x,
    __global const float *y,
    __global float *z )
{
    int i = get_global_id(0);
    z[i] = x[i] + y[i];
}
```
GPUs in the CSF

Some info prior to exercises
Computational Shared Facility

- Heterogeneous cluster
- Contribution model
- Around 1600 cores and growing
- Nvidia GPUs: 5xM2050 + 2xM2070 – One GPU per node
  - 3GB and 6GB memory
  - 448 CUDA cores (14 SMs, not 16)
  - Peak DP 512 GFLOPS
  - Mem bandwidth 148GB/s
- Nvidia drivers installed support OpenCL 1.0
- Nvidia CUDA SDK installed
  - Contains Nvidia OpenCL headers, samples, etc
On login node

```
module load libs/cuda/
```

# currently 4.0.17

Provides access to the following directories

- `CUDA_SDK/OpenCL/`  
  Nvidia OpenCL SDK (sample code etc)
- `CUDA_HOME/include/`  
  Nvidia OpenCL headers (CL/opencl.h)
- `libOpenCL.so`  
  Needed for compilation and execution

Only available on GPU nodes (usually)

OpenCL apps must run on backend GPU nodes

They have a GPU(!) and the `libOpenCL.so` file

- `qsub myjobfile`
- `qsub -b y -cwd -V -l nvidia -l course ./myoclprog`
- `qrsh -V -l nvidia -l inter -l course xterm`

(uppercase V, lowercase letter 'el' – not number 1)

```
#! /bin/bash
#$ -S bash
#$ -cwd
#$ -V
#$ -l nvidia -l course
./myoclprog
```
Not needed for this course's exercises, try it later

Take a local copy of the SDK, compile and modify samples

- The SDK isn't compiled in the central area
- Start an interactive shell on a backend GPU node using `qrsh`

```bash
qrsh -l course -l nvidia -l inter xterm
module load libs/cuda
mkdir ~/opencl
cd ~/opencl
cp -r $CUDA_SDK .
cd 4.0.17
export CUDA_INSTALL_PATH=$CUDA_HOME
make
```

# Lowercase letter 'el'
# Current version 4.0.17
# Choose a directory name
# Go in to the new dir
# Notice the . at the end
# Build CUDA & OCL examples

# Now run your newly compiled OpenCL examples, eg: device query example
OpenCL/bin/linux/release/oclDeviceQuery
Exercise 0
Exercise 1 (parts 1 & 2)

See separate slides
OpenCL Basics
• Created by Khronos Compute Group + industry ~2008
  – Apple wanted many-core CPU, GPU standard
  – Specification to be implemented by vendors
    • OpenCL 1.0 (Dec 2008), 1.1 (Jun 2010), 1.2 (Nov 2011)
  – Currently most vendors support up to v1.1

Resources: OpenCL Registry: http://www.khronos.org/opencl/
  – Specification (READ THIS)
    • www.khronos.org/registry/cl/specs/opencl-1.1.pdf
    • www.khronos.org/registry/cl/specs/opencl-1.2.pdf
  – Reference card (REFER TO THIS)
  – Reference (man) pages:
    • http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/
    • http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/
Introduction to OpenCL

  - CUDA toolkit 4.2 (supports OpenCL 1.1)
  - Download CUDA driver, CUDA toolkit, GPU Computing SDK (example code)
  - OpenCL on Nvidia GPUs only

  - AMD APP SDK 2.7 (supports OpenCL 1.2)
  - Download the AMD driver if on GPU + AMD APP SDK
  - OpenCL on AMD GPUs & Intel/AMD CPUs (very useful)

  - Intel SDK for OpenCL Apps 2012 (supports OpenCL 1.1)
  - Download SDK
  - OpenCL on many Intel CPUs (see release notes)

  - OS X core technology – via Xcode

Vendor Support Q2 2012
- The OpenCL Programming Book
- OpenCL in Action: How to accelerate graphics and computation
  Scarpino. Manning Publications (Nov '11)
- Heterogeneous Computing with OpenCL
  Gaster, Howes, Kaeli, Mistry, Schaa. Morgan Kaufmann (Oct '11)
- OpenCL Programming Guide
  Munshi, Gaster, Mattson, Fung, Ginsburg. Addison Wesley (July '11)
Anatomy of OpenCL

- **Language Specification (use to write kernels)**
  - C-based subset of ISO C99 (+ language extensions)
  - IEEE 754 numerical accuracy
  - Online / offline compilation of compute kernels
- **Platform Layer API (use from the host)**
  - Hardware abstraction layer over your various devices
  - Query, select, init compute devices
  - Create compute contexts and work-queues
- **Runtime API (used to queue up and run your kernels)**
  - Execute compute kernels
  - Manage scheduling, compute and memory resources
The OpenCL view of your system

- E.g. A Core on Nvidia h/w
- E.g. A streaming multiprocessor (SM) on Nvidia h/w
- E.g. A CPU running your application
- E.g. PCI 8GB/s
- E.g. A GPU, could be your CPU!
OpenCL Execution Model

- Expression of parallelism for data-parallel execution
- A \textit{kernel} (function) executes on an OpenCL device.
  - Written in OpenCL C
  - Many instances of a kernel execute in parallel but on different data
- A \textit{work-item} is an instance of the kernel
  - Executed by a Processing Element (PE) in the device
  - Given a unique id to distinguish it from other work-items
- A \textit{work-group} is a collection of work-items
  - Grouped to support your algorithm, sharing local memory, barriers
  - GPU requires batches of work-items (num WI's >> num cores)
  - Its work-items execute concurrently on the same Compute Unit
Define our problem domain as an \( N \)-Dimensional domain

- \( N = 1, 2 \) or \( 3 \). Examples:
  - 1D: Add two vectors (already seen)
  - 2D: Process an image (e.g., blur) 2k x 2k pixels = 4M kernel execs
  - 3D: Process volumetric data acquired from scanner

Execute kernel at each point in the problem domain

- Sometime called a work-grid or index space
- No creating threads within kernel – write your code for a single thread (more like MPI than OpenMP?)

Choose dimension to match your algorithm, thought process …

- can do a 2D or 3D problem using 1D indexing
Example 2-D grid of work-items (e.g., image processing)

- Global dimensions: 1024 x 1024 (the entire problem domain)
- Local dimensions: 128 x 128 (work-group size)
Work-items and Work-groups

- Each instance of a kernel execution is a *work-item*
  - CUDA calls this a thread and refers to SIMT
  - Each work-item given a unique global id
  - Same code executed for every work-item
    - *Path through code can vary (if / else etc)*

- Work-items (threads) are grouped in to work-groups
  - Allows scheduling of device resources, providing scalability
  - Each work-group given a unique group id
  - Each work-item within a work-group given a unique local id

- Built-in functions for kernel to get the various ids
  - A work-item's kernel instance can use these to figure out which data elements to work with…but it is *your* code that does this!
1-D grid (32x1) with 16x1 work-group size

- All functions called from a kernel

```c
int i = get_global_id(0);
output[i] = input[i]*input[i];
```

**Kernel**

```
get_work_dim() = 1
1st dim idx 0
2nd dim idx 1
3rd dim idx 2
```

### Input
```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
9 2 1 7 9 8 4 5 2 1 0 2 6 6 3 0
```

### Output
```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
81 4 1 49 81 64 16 25 4 1 0 4 36 36 9 0
```
2-D Example

- 2-D grid 16x12 with 4x4 work-group size

get\_global\_id(1) = 3
get\_local\_id(1) = 12
get\_global\_size(1) = 16
get\_local\_size(1) = 4
get\_num\_groups(1) = 4
1st dim idx 0
2nd dim idx 1
3rd dim idx 2

work-item
L: (1, 0)
G: (9, 4)
Work-items (threads) within a work-group execute concurrently, at least conceptually

- All work-groups have same number of work-items
  - \( H/w \) specifies maximum number allowed
  - Cannot change number of work-items once kernel is running

You specify number of work-items (global dimensions)

- And optionally number of work-items per work-group (i.e., local dimensions)

```c
int nDim = 2;
int globalSize[2] = {4096, 2048}; // {x,y} not {rows,cols}
int localSize[2] = {32, 32}; // 1024 threads in our work-groups
// Launch kernel specifying nDim, globalSize, localSize (see later)
```

- OpenCL calculates number of work-groups
- The local-size must divide evenly the global size in each dimension
  - Different to CUDA which requires num work-groups and work-group size
- $n$ work-groups execute concurrently
  - Depends on device h/w
  - Work-groups are run in **any** order
  - Work-group sent to a Compute Unit
    - More CUs = more work-groups running in parallel
    - All work-groups eventually processed
    - E.g. 4M work-items to process a large array executed by a ~500 core GPU
  - Provides scalability
    - Automatic speed-up
    - Linear, in theory

Based on diagram in Nvidia, OpenCL Programming Guide for CUDA Architecture, v4.1, 3/1/2012
You'll usually have $num$ work-items $\gg$ $num$ GPU cores
- and $num$ work-groups $\gg$ $num$ Compute Units

GPU swaps work-groups (and their work-items) between being active and inactive
- Usually when work-items start waiting for data from memory
- Get other work-items working while some are waiting

You are hiding the delay (latency) in memory access with more work

Lots of work-items is good!
Compute Unit is a SIMD / vector unit

- Work-items in a work-group grouped in to
  - Warps (Nvidia: 32)
  - Wavefronts (AMD: 64)
- A warp contains consecutive work-item IDs
  - Warps: Work-items 0-31, 32-63, …
- Same instruction from your kernel executed at a time in SIMD unit
  - Each work-item executed by one element of the vector unit
  - Lock-step execution
  - Entire warp processed at once by a vector unit
- If work-group contains more work-items (often does!)
  - Warps of work-items swapped in and out on the vector unit
  - E.g.: when a warp stalls waiting for memory transfer
  - Very little cost associated with swapping (hardware to do it)
Writing an OpenCL Program

Host Code
What code are we going to develop?

Host program
- Set up your device (GPU etc)
- Transfer data to/from the device
- "Manage" kernels to be run on the device (see later)
- Don't try to turn all your code in to a kernel!

Device (GPUs) run your Kernel(s)
- Execute on each "data point" in the problem domain in parallel
- Performs your computation
  - Usually a small function. Several different kernels used for large problems.
- Read / write device memory
  - Think about which memory space a variable exists in (see later)
OpenCL skeleton (host code)

// 0. CPU application setup code (read files etc)

// OpenCL Setup
// 1. Set up OpenCL platform (AMD, Nvidia, Intel's OpenCL)
// 2. Set up OpenCL device(s) supported by platform (GPU,CPU,...)
// 3. Create OpenCL context to manage device(s), memory objects...
// 4. Create OpenCL command queues for device(s) in context
// 5. Create OpenCL program from kernel sources in context
// 6. Use OpenCL to build/compile source for the device(s) in context
// 7. Extract required kernel from compiled program (may be >1)

// Parts of your application that has been OpenCL-ised
// 8. Allocate memory objects on the device
// 9. Transfer data from host to device memory objects
// 10. Specify kernel arguments (input args)
// 11. Launch extracted kernel via device's command queue
// 12. Copy data back from device via device's command queue
// Repeat 8-12 usually

// Other optional CPU work

// 13. Tidy up OpenCL objects
**Summary of the steps**

---

**OpenCL summary**

- **Compile code**
- **Create data & arguments**
- **Send to execution**

Diagram by Tim Mattson (Intel), Udepta Bordoloi (AMD), OpenCL An Introduction for HPC programmers, ISC11 presentation
The following slides are very verbose
- Use as reference material – copy the code etc

During the course we will point out a few key concepts
- Style of OpenCL host coding
- Flexibility

Don't be put off by all this setup code
- Write once then re-use
- All OpenCL coding is fun anyway :-)

Don't Panic
1. Setting up on the Host – Get a Platform

- Select a Platform (often just the one in your system)
  - Nvidia, AMD, Intel etc. Like selecting the correct libOpenCL.so/.dll

```c
cl_int clGetPlatformIDs( cl_uint num_entries, cl_platform_id *platforms,
                        cl_uint *num_platforms )

#include <CL/opencl.h>
int main( void )
{
    cl_int err; // OpenCL funcs give an error flag
    cl_uint num_platforms;
    cl_platform_id platform_id;

    // Assuming just one platform (not portable)
    err = clGetPlatformIDs( 1, &platform_id, &num_platforms );

    if ( err != CL_SUCCESS || num_platforms == 0 ) {
        // No platforms. Exit!
        printf( "No OpenCL platforms!\n" );
        return 1;
    }
    ...
}
```

Introduction to OpenCL
Platform API functions return an error code

- Function return value or output arg
- cl_int, several error values available
  - Errors if you supply bad function args
  - Errors if out of resources on host or device
- At least check for CL_SUCCESS then narrow it down

Example: \( \text{err} = \text{clGetPlatformIDs}(0, \ \&\text{platform\_id}, \ \&\text{num\_platforms}); \)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_SUCCESS</td>
<td>Function executed successfully</td>
</tr>
<tr>
<td>CL_INVALID_VALUE</td>
<td>If ( \text{num_entries} ) is zero and ( \text{platforms} ) is not NULL or if both ( \text{num_platforms} ) and ( \text{platforms} ) are NULL</td>
</tr>
<tr>
<td>CL_OUT_OF_HOST_MEMORY</td>
<td>If there is a failure to allocated resources required by the OpenCL implementation on the host.</td>
</tr>
</tbody>
</table>
More than one Platform?

- A rare case? Might get NV/AMD + Intel on CSF (in future)
  - Ask OpenCL for number of platforms, get them, query them

```c
cl_int clGetPlatformInfo( cl_platform_id platform, cl_platform_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret )
```

```c
int i;
cl_uint num_platforms;
cl_platform_id *platform_ids; // Will be an array of ids
char pinfo[256]; // We're going to ask for a string property

// Get number of ids then the actual ids in to an array
err = clGetPlatformIDs( 0, NULL, &num_platforms );
platform_ids = (cl_platform_id *)malloc(num_platforms*sizeof(cl_platform_id));
err = clGetPlatformIDs( num_platforms, platform_ids, NULL );

// Get info about each platform
for ( i=0; i<num_platforms; i++ ) {
    err = clGetPlatformInfo( platform_ids[i], CL_PLATFORM_VENDOR, sizeof(pinfo),
                                pinfo, NULL ); // Ignoring the return size
    if (!strcmp(pinfo, "Advanced Micro Devices, Inc."))
        break; // Stop searching. Use platform_ids[i] from here on.
}
```
2. Get a device

- Require the devices supported by the platform
  - GPUs, CPU, generic accelerator – depends on vendor

```c
cl_int clGetDeviceIDs( cl_platform_id platform, cl_device_type device_type, cl_uint num_entries, cl_device_id *devices, cl_uint *num_devices )
```

...  

```c
cl_device_id device_id;

// Assuming we just want the first GPU device
err = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL );
if ( err != CL_SUCCESS ) ...  

// Alternatively, get all devices (then choose one or more)
cl_uint num_devices;
cl_device_id *device_ids; // Will be an array of ids

clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 0, NULL, &num_devices );
device_ids = (cl_device_id *)malloc( num_devices*sizeof(cl_device_id) );
clGetDeviceIDs( platform_id, CLDEVICE_TYPE_GPU, num_devices, device_ids, NULL );
```
Choose a Device

- Can query a device for h/w details
  - See the OpenCL spec for param names

```c
cl_int clGetDeviceInfo( cl_device_id device,
                        cl_device_info param_name,
                        size_t param_value_size,
                        void *param_value,
                        size_t *param_value_size_ret )
```

```c
int j;
cl_uint dinfo;  // We're going to ask for a numeric property
for ( j=0; j<num_devices; j++ ) {
  err = clGetDeviceInfo( device_ids[j], CL_DEVICE_MAX_COMPUTE_UNITS,
                         sizeof(dinfo), &dinfo, NULL );  // Ignoring return size

  // Choose device based on properties (somehow - up to you)
  if ( dinfo > MY_MINIMUM_PROPERTY_VALUE )
    break;  // Stop searching. Use this device_ids[j] from here on.
}
```
The Context

- The Context is an environment / container on the host
- Manages OpenCL objects and resources. It contains
  - Device(s) from a single platform (AMD or Nvidia or …)
  - Memory objects on the devices
    - Devices in same context can share memory objects
  - Programs (collection of kernels and functions to run on the device)
  - Command queues: host interacts with a device via a queue
    - Memory commands (transfers to/from the device)
    - Kernel launches
    - Events / synchronisation of commands
3. Create a context

- Context contains one or more devices
  - Nasty looking function but we ignore most of the args

```c
cl_context clCreateContext( const cl_context_properties *properties,
    cl_uint num_devices,
    const cl_device_id *devices
    void (CL_CALLBACK *pfn_notify)(...),
    void *user_data,
    cl_int *errcode_ret )
```

```c
cl_context ctx;
cl_context_properties ctx_props[3];  // Store required properties in array

ctx_props[0] = CL_CONTEXT_PLATFORM;  // Name of property
ctx_props[1] = platform_id;  // Value of property (from earlier)
ctx_props[2] = 0;  // Must end list with 0
                  // (Odd way of supplying a platform_id)

// Just the one device in our context
ctx = clCreateContext( ctx_props, 1, &device_id, NULL, NULL, &err );

// Alternatively, multiple devices (maybe GPUs + CPU etc) in our context
ctx = clCreateContext( ctx_props, num_devices, device_ids, NULL, NULL, &err );
```
Each device requires its own command queue

- Queue points to one device (in a context)
  - *Typically only one queue and one device associated*
- A device can be attached to multiple queues
  - *Allows processing of independent async commands*
- Host puts commands (operations) in to the queue
  - *This is how you ask the GPU to do something*
  - *All commands go via the queue. Like tasks.*
- Device executes the commands
  - *in-order : order in which you put commands in queue (default)*
  - *out-of-order : device schedules commands as it sees fit*
- Can synchronize within and across queues
  - *e.g., prevent one command running before another completes*
4. Create a Command Queue

- Can't talk to our device without one
  - Two properties: `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`
    `CL_QUEUE_PROFILING_ENABLE`

```c
cl_command_queue clCreateCommandQueue( cl_context context,
                                      cl_device_id device,
                                      cl_command_queue_properties properties,
                                      cl_int *errcode_ret )
```

```c
cl_command_queue cmd_queue;         // Stores our new command queue
cl_command_queue_properties q_props; // Bitfield: OR any flags

q_props = 0;                        // Will use queue defaults (in-order, no profiling)

// Repeat this function for each device in our context (ctx)
cmd_queue = clCreateCommandQueue( ctx, device_id, q_props, &err );

// E.g., if we have multiple devices (from earlier) and want a queue per device
*cmd_queue;
cmd_queues = (cl_command_queue *)malloc(num_devices*sizeof(cl_command_queue));
for ( q=0; q<num_devices; q++ ) {
    cmd_queues[q] = clCreateCommandQueue( ctx, device_ids[q], 0, &err );
}
```
OpenCL Programs

- OpenCL Program is a set of kernels (and functions)
  - Kernels, functions written in OpenCL C source
  - Compiled at runtime by the OpenCL driver
    - i.e., while your application is running, OpenCL will compile your kernel code
  - Require the kernel source code at runtime
    - As a string (or array of strings)
    - You could create this by reading from a text file
  - Could also supply a compiled binary
    - See clCreateProgramWithBinary() and clGetProgramInfo(..., CL_PROGRAM_BINARIES, ...) (not covered today)
  - This is all a bit messy
    - However, kernels are compiled for your specific device
    - You only need a C compiler, for host code (no additional compiler ala CUDA)
Supplying OpenCL C source in host code (nasty)

```c
cl_program clCreateProgramWithSource( cl_context context,
                                    cl_uint count,
                                    const char **strings,
                                    const size_t *lengths,
                                    cl_int *errcode_ret )
```

// An array of 5 strings in your host code (very nasty - see next slide)
const char *ksource1[] = {
    "__kernel void arraySqr(__global float *a, unsigned int n ) { ",
    "    unsigned int idx = get_global_id(0);
    ",
    "    if ( idx < n )
    ",
    "        a[idx] = a[idx]*a[idx];
    
    }
};

cl_program prog;                // Holds ours new program object

// No string lengths supplied. Ensure strings are NULL terminated (true above)
prog = clCreateProgramWithSource( ctx, 5, (const char **)ksource1, NULL, &err);
- Write kernels etc in text file (often named .cl)

```c
cl_program prog; // Holds our new program

const char *fname = "mykernel.cl";
FILE *kfile;
size_t kfilesize;
char *ksource;

// Read a text file in to a string
kfile = fopen( fname, "r" ); // Open .cl file for reading
fseek( kfile, 0, SEEK_END ); // Go to end of file
kfilesize = (size_t)ftell(kfile); // Get position (i.e., length)
 rewind( kfile ); // Go back to start
ksource = (char *)malloc(kfilesize*sizeof(char)); // Alloc one long string
fread(ksource, 1, kfilesize, kfile ); // Read file in to string
fclose( kfile ); // Finished with source file

// One string containing all the source code read in, length passed in.
prog = clCreateProgramWithSource( ctx, 1, (const char **)ksource, &kfilesize, &err);
```
6. Building your Kernel source

- Ask OpenCL to build (compile & link) the source
  - Compiled for specific devices. Looks nasty (but ignore most args)
  - Supply compiler flags in a string

```c
cl_int clBuildProgram(  
    cl_program program,  
    cl_uint num_devices,  
    const cl_device_id *devices,  
    const char *options,  
    void (CL_CALLBACK *pfn_notify)(...),  
    void *user_data )
```

// Build source for all devices associated with the program
// or supply a list of specific (one or more) devices
err = clBuildProgram( program, 0, NULL, NULL, NULL, NULL );
if ( err != CL_SUCCESS ) ... // CHECK THE ERROR - INDICATES COMPILATION FAILED

// Usual compiler options (-I for include dirs, -D for #ifdef, #define macros)
// There are vendor-specific options (e.g., Nvidia: -cl-nv-verbose)
char options[] = 
                   
err = clBuildProgram( program, 0, NULL, options, NULL, NULL );
// Or supply a simple const string
err = clBuildProgram( program, 0, NULL, 
                      
```
When it doesn't Compile

- Very easy to make mistakes in kernel source
  - Not easy to get compiler output and helpful messages
  - Might need several attempts at compiling simpler code

```c
cl_int clGetProgramBuildInfo( cl_program program,
                              cl_device_id device,
                              cl_program_build_info param_name,
                              size_t param_value_size,
                              void *param_value,
                              size_t *param_value_size_ret )
```

```c
size_t log_size; // Length of build log
char *build_log; // String containing the build log

// Use CL_PROGRAM_BUILD_LOG for the param name. Get the log size first.
err = clGetProgramBuildInfo( program, device_id, CL_PROGRAM_BUILD_LOG,
                              0, NULL, &log_size );
build_log = (char *)malloc(log_size*sizeof(char));
err = clGetProgramBuildInfo( program, device_id, CL_PROGRAM_BUILD_LOG,
                              log_size, build_log, NULL );
printf( "Build log\n%s\n", build_log );
```
7. Extract required Kernel for Launching

- Program now contains one or more compiled kernels
  - Get a kernel object from the program so we can run it
  - Use same kernel name from the source code

```c
cl_kernel clCreateKernel( cl_program program,
                         const char *kname,
                         cl_int *errcode_ret )
```

```c
cl_kernel kernel; // Holds our new kernel
const char *kname = "arrayAdd"; // Name of the kernel

// Get the named kernel
kernel = clCreateKernel( program, kname, &err );

// May have compiled several kernels
cl_kernel kernel1, kernel2;
kernell1 = clCreateKernel( program, "arrayAdd", &err );
kernell2 = clCreateKernel( program, "arraySum", &err );
```
Summary of Setup

- Got a platform (Nvidia, AMD, etc)
- Got device(s) supported by that platform (GPUs …)
- Created a context to manage devices and other objects
- Created a command queue, in that context, to feed device
- Created a program, in that context, from OpenCL C source code
- Built (compiled and linked) the program for device(s) in the context
- Extracted a specific kernel so we can launch it on the device
Now some more interesting work

- Allocate memory objects on the device
- Transfer data from host to device
- Specify the arguments for a kernel launch
- Launch the kernel
- Read data back from the device (results of our kernel)

Repeat as required

- May have other kernels to launch

Cleanup objects at end
Memory Objects / Buffers

- Data from host must be copied to the device
  - GPU device has physical DRAM distinct from host
  - Buffer objects used as containers
    - Managed by the context and visible to the devices in that context
    - If using CPU devices, still require buffer objects

- Create the buffer then populate it
  - Can do this in one step or separately

- Transferring data to devices is costly
  - Transfer over PCIe bus (e.g., current GPUs)
  - This can kill any performance gain

- Kernels take mem objs as inputs and outputs
8. Create a Memory Object

- Specify type of access and size
  - Buffers are 1D hence calc your size correctly (number of bytes)
  - Can contain scalars (int, float,...), vectors (float4, ...), user structs

```c
cl_mem clCreateBuffer(cl_context context, cl_mem_flags flags,
                      size_t size, void *host_ptr,
                      cl_int *errcode_ret)
```

```c
long count;                // Example number of elements in my array
float *vecA_h;            // Often use a _h suffix for host mem
cl_mem vecA_d;            // Often use a _d suffix for device mem

// Create a write-only buffer: kernel will only write it, not read it
vecA_d = clCreateBuffer(ctx, CL_MEM_WRITE_ONLY, sizeof(float)*count,
                        NULL, &err);

// Flags OR-ed together: kernel will only read buffer. We populate buffer too.
cl_mem vecB_d;
float *vecB_h = readFloatDataFromFile("myinput.dat", &count);
cl_mem_flags mflags = CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR;
vecB_d = clCreateBuffer(ctx, mflags, sizeof(float)*count, vecB_h, &err);
```
### cl_mem_flags bitfield

<table>
<thead>
<tr>
<th>Buffer Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CL_MEM_READ_WRITE</strong></td>
<td>Allocate a mem obj that can be read from and written to by a kernel.</td>
</tr>
<tr>
<td><strong>CL_MEM_WRITE_ONLY</strong></td>
<td>Mem obj not read by kernel. Reading is undefined. Cannot use with CL_MEM_READ_WRITE.</td>
</tr>
<tr>
<td><strong>CL_MEM_READ_ONLY</strong></td>
<td>Mem obj is read-only when used in a kernel. Writing to within kernel is undefined. Cannot use with other WRITE flags.</td>
</tr>
<tr>
<td><strong>CL_MEM_USE_HOST_PTR</strong></td>
<td>Use storage bits referenced by host_ptr arg (cannot be NULL) as mem obj. OpenCL may cache content in device memory for use by kernels.</td>
</tr>
<tr>
<td><strong>CL_MEM_ALLOC_HOST_PTR</strong></td>
<td>Asks OpenCL to alloc memory for mem obj from host memory. Cannot use with CL_MEM_USE_HOST_PTR.</td>
</tr>
<tr>
<td><strong>CL_MEM_COPY_HOST_PTR</strong></td>
<td>Ask OpenCL to alloc device memory and copy data in from host_ptr. Cannot be used with CL_MEM_USE_HOST_PTR. Can be used with CL_MEM_ALLOC_HOST_PTR.</td>
</tr>
</tbody>
</table>
9. Copying data via Command Queue

- Can **read** contiguous data from **device to host** or **write** from **host to device** (a)synchronously (non-blocking).

```
cl_int clEnqueueReadBuffer( cl_command_queue command_queue,
                            cl_mem buffer, cl_bool blocking_read,
                            size_t offset, size_t cb,
                            void *ptr, cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list, cl_event *event )
```

```
cl_int clEnqueueWriteBuffer( cl_command_queue command_queue,
                             cl_mem buffer, cl_bool blocking_write,
                             size_t offset, size_t cb,
                             const void *ptr, cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list, cl_event *event )
```

- Set blocking_[read|write] to CL_TRUE to wait until transfer completes. Use CL_FALSE for call to return immediately.
- Host can carry on working while transfer occurs (don't use mem)
- Wait on an event to be sure transfer has completed
Copying data via Command Queue

- Example copying to device via separate alloc and transfer

```c
long count;                     // Example number of elements in my array
float *vecA_h;                  // Often use a _h suffix for host mem
cl_mem vecA_d;                  // Often use a _d suffix for device mem
cl_event event;                 // Indicates when transfer has completed

// Alloc host array and populate it (somehow)
vecA_h = (float *)malloc(count*sizeof(float));
readMyDataFromSomewhere(vecA_h, count);

// Create a read-only buffer: kernel will only read it, not write it
vecA_d = clCreateBuffer( ctx, CL_MEM_READ_ONLY, count*sizeof(float), NULL, &err );

// Enqueue a non-blocking write from host to device
err = clEnqueueWriteBuffer( cmd_queue, vecA_d, CL_FALSE, 0, count*sizeof(float),
                             vecA_h, 0, NULL, &event );

// Host can carry on with other work (don't touch vecA_h though!)
// E.g., create and populate another host array
...

// Eventually wait until previous transfer has completed
clWaitForEvents( 1, &event );

// Can now safely do something to the host array
free(vecA_h);
```
10. Set up Kernel Args

- Use a function to set our kernel inputs/outputs
  - Sadly we don't just call my_kernel(arg1, arg2, ...);

```c
cl_int clSetKernelArg( cl_kernel kernel, cl_uint arg_index, size_t arg_size, const void *arg_value )
```

// Extract kernel from program earlier
// Allocated device memory objects earlier via clCreateBuffer() earlier

// Set the kernel args (refer your kernel source code!) Note err flag trick.
err = clSetKernelArg( kernel, 0, sizeof(cl_int), &n );
err |= clSetKernelArg( kernel, 1, sizeof(cl_mem), &vecA_d ); // Input mem obj
err |= clSetKernelArg( kernel, 2, sizeof(cl_mem), &vecB_d ); // Input mem obj
err |= clSetKernelArg( kernel, 3, sizeof(cl_mem), &vecC_d ); // Output mem obj
if ( err != CL_SUCCESS ) ...

```c
__kernel void arrayAddOCL( const int n, __global const float *x,
                           __global const float *y, __global float *z )
{
    int i = get_global_id(0);
    z[i] = x[i] + y[i];
}
```
Add kernel launch to command queue
- Think back to the NDRange layout of work-items (threads)
- A non-blocking (async) function – returns immediately to host code

```
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 )
```

// 1-D example: Do the vector add for 64000 element arrays
int globalSize = 64000; // Number of elements in our arrays
int localSize = 128; // Work-groups contain 128 work-items (500 wg's)

// No work offset (work-item ids start at 0). No events to wait for or returned.
err = clEnqueueNDRangeKernel( cmd_queue, kernel, 1, NULL, &globalSize &localSize, 0, NULL, NULL );

// Command returns immediately to host, which can carry on working (see later)
The `work_dim` arg specifies array sizes for `work_xx` args
- `global_work_offset` changes initial value of `get_global_id()`
  - Pass in NULL to keep at 0 (0, 0) (OpenCL 1.1)
- `global_work_size` gives total number of work-items reqd
- `local_work_size` gives work-group size

// 2-D example: Processing a matrix of size 2048 rows x 4096 columns
// Number of columns is 4096. This is the X dimension (or width)
// Number of rows is 2048. This is the Y dimension (or height)

```c
int globalSize[2], localSize[2]; // `work_dim` is 2 so all args need two dimension
globalSize[0] = 4096; // Dimension [0] is x (num cols)
globalSize[1] = 2048; // Dimension [1] is y (num rows)
localSize[0] = 32; // 32x32 = 1024 work-items in a work-group
localSize[1] = 32; // Hence there will be 128x64 = 8192 work-groups
```

// No work offset (work-item ids start at 0). No events to wait for or returned.
```c
err = clEnqueueNDRangeKernel( cmd_queue, kernel, 2, NULL, globalSize
localSize, 0, NULL, NULL );
```
Number of work-items and work-groups

- OpenCL calculates number of work-groups using
  - \( \text{global\_work\_size}[0] / \text{local\_work\_size}[0] \)
  - \( \text{global\_work\_size}[1] / \text{local\_work\_size}[1] \) …
  - Your \text{global\_work\_size} numbers \textbf{must} be evenly divisible by your \text{local\_work\_size} numbers
    - err return flag will be \text{CL\_INVALID\_WORK\_GROUP\_SIZE} if not
    - Lots of other error flag values indicating where dimension info is wrong

- Your numbers must be within the hardware limits
  - Individual limits ([0], [1], …) and total limits ([0] x [1] x … )
  - \text{clGetDeviceInfo()} and ask for
    - \text{CL\_DEVICE\_MAX\_WORK\_ITEM\_SIZES},
    - \text{CL\_DEVICE\_MAX\_WORK\_GROUP\_SIZE} etc
Will need to experiment per device

- Num work-groups > Num Compute Units (e.g., Nvidia SM)
  - Each CU has at least one work-group to execute
- Num work-groups / Num Compute Units > 2
  - Each CU can run multiple work-groups
  - Allows swapping of work-groups if one stalls (e.g., on a barrier – see later)
- Num work-groups / Num Compute Units > 100 (Nvidia recommendation)
  - Your code will scale on future devices (won't run out of work)
More work-items than needed

- What if the global / local numbers don't divide?
  - Silly example: your array length is prime
  - More realistic: $\text{global}_\text{work}_\text{size}[0] = 10000$
    $\text{local}_\text{work}_\text{size}[0] = 64$
    - Asking OpenCL for 156.25 work-groups!

- Specify more work-items than reqd

```c
int num_work_groups = (\text{global}_\text{work}_\text{size}[0] + \text{local}_\text{work}_\text{size}[0] - 1) / \text{local}_\text{work}_\text{size}[0];
global\_work\_size[0] = num\_work\_groups * \text{local}_\text{work}_\text{size}[0];
```

- Set $\text{global}_\text{work}_\text{size}[0] = 10048$
- Kernel must check for going out of bounds in your arrays

```c
__kernel void arrayAddOCL( const int n, __global const float *x,
                           __global const float *y, __global float *z )
{
    int i = get_global_id(0); // 10048 work-items hence i could = 10000+
    if ( i < n ) // Pass in n=10000 i.e., the true array len
        z[i] = x[i] + y[i];
}
```
Fewer work-items than needed?

- Could also iterate within a kernel
  - Work-item processes more than one element
  - Decouples index-space from data

```c
__kernel void vecAdd( const int n, __global const float *x, ... )
{
    int i = get_global_id(0);
    int stride = get_global_size(0);
    for ( ; i<n; i+=stride )
        z[i] = x[i] + y[i];
}
```

- Work-item 0 accesses array elements 0, 64, 128, ...
- Work-item 1 accesses array elements 1, 65, 129, ...
- Work-item 25 accesses array elements 25, 89, 153, ...
- Work-item 63 accesses array elements 63, 127, 191, ...

---

\[
\text{1D index-space of 64 work-items (4 work-groups of 16)}
\]

- 01 25 63

\[
\text{Work-item 0 accesses array elements 0, 64, 128, ...}
\]

\[
\text{Work-item 1 accesses array elements 1, 65, 129, ...}
\]

\[
\text{Work-item 25 accesses array elements 25, 89, 153, ...}
\]

\[
\text{Work-item 63 accesses array elements 63, 127, 191, ...}
\]

---

\[
\text{n = 1000000;} \quad \text{// Number of elements in our arrays}
\]

\[
\text{globalSize[0] = 64;} \quad \text{// Unrealistically low (point is that it is < 1000000)}
\]

\[
\text{localSize[0] = 16;} \quad \text{// Number of work-items in a work-group}
\]

\[
\text{clSetKernelArg(kernel, 0, sizeof(cl_int), &n);} \quad \text{// Set other args too ...}
\]

\[
\text{clEnqueueNDRangeKernel(q, kernel, 1, NULL, globalSize, localSize, 0, 0, 0);}
\]
Choosing the work-group size

- **local_work_size** arg can be NULL
  - OpenCL will choose the "best" value
  - Don't rely on it being the "best"
  - Often recommended to be multiple of warp/wavefront size
    - *Nvidia suggest multiple warps per work-group*

- You'll need to experiment to find the best value
  - Run kernel with several values and time it (more later)
  - Can get the value the device thinks is the "best"

```c
// Get device-specific kernel info
size_t max_wg_size; // Your local_work_size[0] x local_work_size[1] ...
                      // should equal this value.

err = clGetKernelWorkGroupInfo( kernel, device_id, CL_KERNEL_WORK_GROUP_SIZE,
                                 sizeof(size_t), &max_wg_size, NULL );
```
12. Transfer results back to Host

- Enqueue a read command to get data back from device
  - As before, transfer can be blocking or non-blocking
  - Don't transfer back if simply passing cl_mem obj to another kernel
  - An in-order queue guarantees the kernel has finished

```c
long count;               // Example number of elements in my array
float *vecC_h;           // Host array to be populated with results
cl_mem vecC_d;           // Device mem obj written to by kernel

// Kernel was Enqueued earlier and may be running...

// Alloc host array to hold results from device (usually allocated earlier)
vecC_h = (float *)malloc(count*sizeof(float));

// Enqueue a blocking read from device to host
err = clEnqueueReadBuffer( cmd_queue, vecC_d, CL_TRUE, 0, count*sizeof(float),
                           vecC_h, 0, NULL, NULL );

// The above call will not return until transfer is complete
writeMyArrayToFile(vecC_h, count);
```
How do we know the kernel has finished?
- In-order queue: the clEnqueueReadBuffer() command won't run until previous commands in queue have completed
- Out-of-order queue: Need to wait on an event or barrier (see later)

```
// 1-D example: Do the vector add for 64000 element arrays
int globalSize = 64000;    // Number of elements in our arrays
int localSize = 128;       // Work-groups contain 128 work-items (500 wg's)
cl_event event;           // Indicate when kernel has finished

// No work offset. No events to wait for. An event is written to when complete.
err = clEnqueueNDRangeKernel( cmd_queue, kernel, 1, NULL, &globalSize
                            &localSize, 0, NULL, &event );

// Command returns immediately to host, which can carry on working
...
// Wait for one event then transfer data back from device to host (blocking call)
err = clEnqueueReadBuffer( cmd_queue, vecC_d, CL_TRUE, 0, count*sizeof(float),
                          vecC_h, 1, &event, NULL );
```
Free resources on device as you would on host

- Free up device memory (buffers, objects etc)
  - At end of program and between kernels if no longer required
  - 'Creation' commands have a 'release' equivalent
  - Typically call in reverse order (although reference counting is used)

```
// This ensures all previous commands have finished (may not be needed)
clFinish( cmd_queue );

// Release device objects
clReleaseMemObject( vecA_d );
clReleaseMemObject( vecB_d );
clReleaseMemObject( vecC_d );
clReleaseProgram( prog );
clReleaseKernel( kernel );
clReleaseCommandQueue( cmd_queue );
clReleaseContext( ctx );
```

// Don't forget host objects - let's be tidy
free( vecC_h );
Sample app. No error checking (not recommended!)

```c
#include <CL/opencl.h>

// Kernel as one long single string
const char *ksource = " 
  __kernel void myKernel( const int n, __global float *inArr, __global float *outArr ) { \n  int idx = get_global_id(0); \n  // Do something wonderful in OpenCL \n  outArr[idx] = inArr[idx] * inArr[idx]; \n}\n"

#define VSIZE 1024*1024
int main( void )
{
    cl_int err;
    cl_platform_id p_id;
    cl_device_id d_id;
    cl_context_properties cprops = {CL_CONTEXT_PLATFORM, 0, 0};
    cl_command_queue cq;
    cl_program prog;
    cl_kernel kernel;
    cl_mem A_d, B_d;
    float A_h[VSIZE], B_h[VSIZE];
    long nbytes;
    int vsize = VSIZE;

    nbytes = VSIZE*sizeof(float); // Num bytes to alloc
    initArray( A_h, VSIZE); // Random values?

    // Assuming just one platform, one GPU
    err = clGetPlatformIDs( 1, &p_id, &num_platforms );
    clGetDeviceIDs(p_id, CL_DEVICE_TYPE_GPU, 1, &d_id, NULL);
    cprops[1] = p_id;
    cq = clCreateContext( cprops, 1, &d_id, NULL, NULL, NULL);
    prog = clCreateProgramWithSource( ctx, 1, &ksource, NULL, NULL);
    clBuildProgram( prog, 0, NULL, NULL, NULL, NULL );

    kernel = clCreateKernel( prog, "myKernel", NULL);
    A_d = clCreateBuffer( ctx, CL_MEM_READ_ONLY, nbytes, 0, 0);
    B_d = clCreateBuffer( ctx, CL_MEM_WRITE_ONLY, nbytes, 0, 0);

    // Enqueue a blocking write from host to device mem obj
    clEnqueueWriteBuffer( cq, A_d, CL_TRUE, 0, nbytes, A_h, 0, 0, 0);

    // Set the kernel args
    clSetKernelArg(kernel, 0, sizeof(int), &vsize);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &A_d);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &B_d);

    // Enqueue a kernel launch using a 1-D NDRange. Auto W-G size.
    clEnqueueNDRangeKernel( cq, kernel, 1, NULL, &vsize, 0, 0, 0, 0);

    // Enqueue a blocking readback from the device
    clEnqueueReadBuffer( cq, B_d, CL_TRUE, 0, nbytes, B_h, 0, 0, 0);

    // Could do something with the results in B_h now ...

    // Tidy up
    clReleaseMemObject( A_d );
    clReleaseMemObject( B_d );
    clReleaseProgram( prog );
    clReleaseKernel( kernel );
    clReleaseCommandQueue( cq );
    clReleaseContext( ctx );
}
```
Exercise 1 (part 3)
Exercise 2
OpenCL C and Kernels
Written in OpenCL C

- Restricted version of ISO C99
  - No recursion, function pointers, or functions from the standard headers
- Preprocessor (C99) supported
  - #ifdef etc
- Scalar datatypes plus vector types
  - E.g., float4, int16 etc (with host equivalents: cl_float4, cl_int16)
- Image types: image2d_t, image3d_t etc
  - Alternative to memory buffers for images (not covered)
- Built-in functions: already seen work-item/group id functions
  - Many math.h functions, synchronisation functions
  - Optional functions up to vendors: e.g., double precision
OpenCL Kernels

- Remember: one instance of the kernel created for each work-item (thread)

  - Must have __kernel keyword
    - otherwise is a function only a kernel can call

  __kernel void simpleOCL(const int n, __global float *x )
  {
      int i = get_global_id(0);
      if ( i < n )
          x[i] = sin((float)i);
  }

  - No return value allowed (void)
  - Declare address space of memory object args

- Lots of builtin maths / geometry functions

- Work out which data a work-item uses
  - get_global_id(0), get_local_id(1) etc
  - 2D: column = x = get_global_id(0), row = y = get_global_id(1)
- Raise all elements of a vector by the given exponent
  - 1-D index-space

```c
__kernel void exponentor(__global int* data, const uint exponent)
{
    int tid = get_global_id(0);
    int base = data[tid];
    int i;
    for (i=1; i < exponent; ++i) {
        data[tid] *= base;
    }
}
```

- See trick later to avoid passing in the exponent param

---

- Rotate a 2-D image about its centre

  - OpenCL has built-in image types and functions. For this simple example we use an ordinary memory buffer.

  - **WARNING:** there is a horrible inefficiency in this kernel. Spotted it?

```c
// We should be launched with a 2-D index space. For example:
int globalSize[2] = {2048,2048};  // Input 'image' array is 2k x 2k values
int localSize[2] = {64,64};        // Divides evenly the globalSize[]
__kernel void rotArray( const float theta, const int w, const int h,

  __global const int *inImage, __global int *outImage)

{
    int dstx = get_global_id(0);    // This work-item's point in index-space
    int dsty = get_global_id(1);    // is where we will write to (not read from)

    // We read from elsewhere in the image data
    int srcx = (int)floor( ((float)dstx)*cos(theta) + ((float)dsty)*sin(theta) );
    int srcy = (int)floor( ((float)dsty)*cos(theta) - ((float)dstx)*sin(theta) );

    // Check bounds before reading from elsewhere
    if ( (srcx>=0 && srcx<w) && (srcy>=0 && srcy<h) ) {
      // Can access memory buffers using 1-D indexing or 2-D (as with C)
      outImage[dsty*w+dstx] = inImage[srcy*w+srcx];
    }
}
```

Based on example by Perhaad Mistry, Daba Schaa, Benedict Gaster, AMD OpenCL University Kit
Some features of OpenCL are optional

- Not all vendors support them (yet)
- May be promoted to core features in subsequent specs
- Many exist. Check the string returned by
  - `clGetDeviceInfo(…, CL_DEVICE_EXTENSIONS, …);`
- To control the OpenCL compiler's use of extensions use `#pragma`

```c
#ifdef cl_khr_fp64
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#else
#error Kernel requires double precision
#endif

__kernel void DoubleKernel(__global const double *vecA, …)
{
    double accum;
    int i = get_local_id(0);
    …
}
```
- Portable version of scalar datatypes extended to vectors
  - Various types: char, uchar, short, ushort, int, uint, long, ulong, float, (double)
  - Various lengths: 2, 4, 8, 16
  - E.g. in kernels: uchar2, int4, float16 etc
  - On the host: cl_uchar2, cl_int4, cl_float16 etc
  - H/W may optimize (e.g., map to SIMD units)
    - On CPU (e.g., Intel OpenCL) map to SSE instructions
    - On future APUs (CPU+GPU)
  - OpenCL compiler will map to supported (non-vector) instructions if h/w doesn't support a particular type
  - Aligned to size of vector. E.g., float4 aligned to 16-byte boundary
Vector Notation and Ops in Kernels

- **Vector literals**

  ```c
  float4 fvec = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
  uint4 uvec = (uint4)(1); // uvec will be (1,1,1,1)
  float4 fvec = (float4)((float2)(1.0f, 2.0f), (float2)(3.0f, 4.0f));
  float4 fvec = (float4)(1.0f, 2.0f); // ERROR (need 4 components)
  ```

- **Components: if 4, access using .xyzw, if 2, use .xy**

  ```c
  float4 fvec = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
  fvec.z = 1.0f // Single component is a float
  fvec.xy = (float2)(5.0f, 8.0f); // Two components are a float2
  int2 ivec = (int2)(3, 5);
  ivec.z = 7; // ERROR (no .zw when int2)
  ```

- **Components: numeric index sn where n=0,1,...,vecsize-1**

  ```c
  float4 avec, fvec = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
  fvec.s0 = 1.0f;
  fvec.s23 = (float2)(8.0f, 9.0f);
  avec.xyzw = fvec.s0123;
  a = f.x12w; // ERROR (can't mix xyzw and sn)
  ```
### Vector Ops...

- **Swizzling**
  ```
  float4 fvec = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
  float4 swiz = fvec.wzxy       // swiz = (4.0, 3.0, 1.0, 2.0)
  float4 gvec = fvec.xxxy;     // gvec = (1.0, 1.0, 2.0, 2.0)
  ```

- **Assignment of components**
  ```
  float4 fvec = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
  fvec.xw = (float2)(5.0, 6.0)    // fvec = (5.0, 2.0, 3.0, 6.0)
  fvec.wx = (float2)(7.0, 8.0)    // fvec = (8.0, 2.0, 3.0, 7.0)
  fvec.xy = (float4)(1.0f, 2.0f, 3.0f, 4.0f) // ERROR (xy is float2)
  ```

- **low, hi, odd, even**
  ```
  float4 fvec = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
  float2 low = fvec.lo         // = (1.0, 2.0) (fvec.xy)
  float2 high = fvec.hi;       // = (3.0, 4.0) (fvec.zw)
  float2 even = fvec.even;     // = (1.0, 3.0) (fvec.xz)
  float2 odd = fvec.odd;       // = (2.0, 4.0) (fvec.yw)
  float2 avez = (float2)(fvec.odd.lo, fvec.even.hi); // = (2.0, 3.0) (yz)
  ```
- Operations and functions accepts vectors

```c
float4 fvec = (float4)(1.0f, 2.0f, 7.0f, 8.0f);
float4 gvec = (float4)(4.0f, 3.0f, 2.0f, 1.0f);
fvec -= gvec; // fvec = (-3.0, -1.0, 5.0, 7.0)
gvec = abs(fvec); // gvec = (3.0, 1.0, 5.0, 7.0)
float c = 3.0f;
fvec = gvec + c; // fvec = (6.0, 4.0, 8.0, 10.0)
fvec *= 3.142; // Component-wise mult
gvec = sin(fvec); // Sin() on each component
float4 svec = sin(fvec.xxyy); // Sin() on x,x,y,y components
float3 crsp = cross(fvec, gvec); // Cross product
float dotp = dot(fvec, gvec); // Dot product
float sum = dot(fvec, (float4)1.0f ); // Sum elements of fvec
```

- Lots of built-in mathematical and geometric functions
  - Accept scalar and vector types
  - E.g., sin(a), cos(a), exp(a), fabs(a), hypot(a,b), sqrt(a), rsqrt(a) (inverse square root), isless(a,b) (scalars: 0=false, 1=true, vectors: component-wise flags 0=false, -1=true)
Memory and Kernels

Memory hierarchy
The OpenCL memory model

- **Private mem: __private**
  - R/W per work-item
- **Local mem: __local**
  - R/W by the work-items in a work-group
- **Global mem: __global**
  - R/W by work-items in all work-groups
- **Const mem: __constant**
  - R-only global mem visible to all work-items
    Host allocs and inits
- **Host mem:**
  - On the CPU

Explicit memory management: You move data host->global->local and back
A kernel's local variables and a function's args are private

- E.g., `x, y, n, a, b, temp, p, i` all automatically `__private` to each work-item

```c
float myFunc( float x, float y ) {
    return 2.0*x*x*x+3.0*y*y-1.0;
}
__kernel void myKernel( const int n, ... ) {
    float a, b;
    float temp[4];
    __global int *p;
    int i = get_global_id(0);
    // Kernel body will call myFunc(a,b) at some point
}
```

GPUs alloc `__privates` in Compute Unit's register file

- Fastest access times
- Limited number of registers per CU (and hence per work-item)
- More registers implies fewer work-groups executing concurrently
- Use to share items between work-items in a work-group
  - Allows work-items to communicate (within their work-group)
  - Can save re-loading data from global memory
  - Can replace a `_private` (register) if same value in all work-items
  - Useful for (temp) arrays – quicker than using global memory

```c
__kernel void myKernel( const int n, ... )
{
  __local float tmpScale;       // Cannot initialize here
  __local float wgValues[64];   // Hard-coded size in kernel
  int g_idx = get_global_id(0); // global work-item id
  int l_idx = get_local_id(0);  // local work-item id in work-group
  tmpScale = 123.456;           // Bad: All work-items do this! Use __constant?

  wgValues[l_idx] = myComputation(g_idx, ...);
  ...
}
```
Local memory highlights a potential problem
- All work-items in a work-group writing to a local array
- How do we know when all work-items have written (or read, etc)?

```c
__local float wgValues[64];    // Assume work-group size of 64
wgValues[get_local_id(0)] = myComputation(...);
// Have all work-items in work-group written to wgValues[]? See later
if ( get_local_id(0) == 0 )      // Lots of idle work-items!
    for ( i=0; i<64; i++ )
        sum += wgValues[i];
```

Memory model has \textit{relaxed consistency}
- Different work-items may see a different view of local and global memory as computation progresses
- Load/store consistency within a work-item
- Global & local mem mem consistent within a work-group at barrier
- Command queue events ensure consistency between kernel calls
Local Memory cont…

- Do we have to hard-code a __local mem array size?
  - E.g., Often want size equal to number of work-items in work-group
  - Hard-coding the size is not portable in this case
  - No dynamic mem alloc (e.g., malloc) in a kernel
  - Could use a preprocessor value (reqs kernel recompiles)

- Pass size as kernel arg from host (in a slightly odd way)
  - Setting last arg of clSetKernelArg() to NULL means __local mem
    
    ```c
    err |= clSetKernelArg( kernel, 0, sizeof(int), &numElems ); // As earlier
    err |= clSetKernelArg( kernel, 1, sizeof(cl_mem), &vecA_d ); // As earlier
    err |= clSetKernelArg( kernel, 2, sizeof(cl_float)*localSize, NULL );
    ```

```c
__kernel void myKernel(const int n, __global float *x, __local float *localArray)
{
    int local_idx = get_local_id(0);
    // 0 ... get_local_size(0)-1
    localArray[local_idx] = ...;
    // work-items in work-group init array
}```
- Used to refer to memory objects (buffers or images)
  - Memory allocated with clCreateBuffer() is in global memory
  - Usually passed in to kernel as an arg
  - Can pass to functions

```c
float myFunc( int i, __global const float *x ) {
    float tmp = *(x+i);          // Could also use x[i]
    return 2.0*tmp*tmp*tmp+3.0*sin(tmp)-1.0;
}

__kernel void myKernel( const int n,
                        __global const float *inArr,
                        __global float *outArr )
{
    int i = get_global_id(0);
    outArr[i] = myFunc(i, inArr);
}
```
Constant Memory

- Similar to __global but read-only within a kernel
  - May be allocated in same global memory
  - Device may have separate constant mem cache (e.g., 64Kb)
  - Vars with 'program scope' must be declared __constant

```c
// Create a read-only cl_mem obj. Initialize with these host values.
const float table_h[5] = {1.0, 3.0, 5.0, 3.0, 1.0};
cl_mem table_d = clCreateBuffer( ctx, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
                                 sizeof(float)*5, table_h, NULL );

// Pass read-only buffer to kernel just like any other cl_mem obj
err = clSetKernelArg( kernel, 0, sizeof(int),   &numElems ); // As earlier
err |= clSetKernelArg( kernel, 1, sizeof(cl_mem), &vecA_d ); // As earlier
err |= clSetKernelArg( kernel, 2, sizeof(cl_mem), &table_d ); // constant

__constant int myConst = 123;    // Var has 'program scope' (use pre-processor)

__kernel void myKernel(const int n, __global float *x, __constant float *table)
{
    for ( i=0; i<5; i++ )
        // Do something with table[i]
}
Coalesced __global memory access for efficiency
- work-item i to access array[i]
  work-item i+1 to access array[i+1]

Access to C matrices (row-major)
- E.g., a 1D index-space where a single work-item processes an entire row of a C matrix (seen in mat-mul examples)
- Rules are slightly more relaxed
  - Accessing sequential elements in a half-warp = 1 mem op!

```
<table>
<thead>
<tr>
<th>half-warp</th>
<th>half-warp</th>
</tr>
</thead>
</table>
```

- Allowed mixed/offset access within a half warp (still one op)

```
<table>
<thead>
<tr>
<th>half-warp</th>
<th>half-warp</th>
</tr>
</thead>
</table>
```

- Access all elements in a block even if not using them all
  - Allows coalescing
- Limits to various memory spaces
  - Can spill from faster to slower memory (or crash)

```c
cl_ulong dinfo; // Properties below of this type unless indicated
clGetDeviceInfo( &device_id, PROPERTY, sizeof(dinfo), &dinfo, NULL );
```

<table>
<thead>
<tr>
<th>Memory Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_DEVICE_GLOBAL_MEM_SIZE</td>
<td>Size of global device memory in bytes.</td>
</tr>
<tr>
<td>CL_DEVICE_MAX_MEM_ALLOC_SIZE</td>
<td>Max memory object allocation size in bytes. The minimum value is $\max\left(\frac{1}{4} \text{CL_DEVICE_GLOBAL_MEM_SIZE}, 128\text{MB}\right)$</td>
</tr>
<tr>
<td>CL_DEVICE_LOCAL_MEM_SIZE</td>
<td>Size of local memory arena in bytes. Minimum value is 32KB.</td>
</tr>
<tr>
<td>CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE</td>
<td>Size of constant buffer memory arena in bytes. Minimum value is 64KB</td>
</tr>
<tr>
<td>CL_DEVICE_MAX_CONSTANT_ARGS</td>
<td>(cl_uint) Max number of arguments declared with __constant qualifier in a kernel. The minimum value is 8.</td>
</tr>
<tr>
<td>CL_DEVICE_LOCAL_MEM_TYPE</td>
<td>(cl_device_local_mem_type) Type of local memory supported. Can be CL_LOCAL implying dedicated local mem storage such as SRAM, or CL_GLOBAL.</td>
</tr>
</tbody>
</table>
Introduction to OpenCL
Part 2 (half day)

June 2012

George Leaver
IT Services
george.leaver@manchester.ac.uk
Example 2-D grid of work-items (e.g., image processing)

- Global dimensions: 1024 x 1024 (the entire problem domain)
- Local dimensions: 128 x 128 (work-group size)

Single work-item (instance of kernel)

Work-items within a work-group can synchronize (barriers) and share local memory.

Work-items outside work-groups cannot synchronize. Work-groups are independent.
The OpenCL memory model

- **Private mem: __private**
  - R/W per work-item

- **Local mem: __local**
  - R/W by the work-items in a work-group

- **Global mem: __global**
  - R/W by work-items in all work-groups

- **Const mem: __constant**
  - R-only global mem visible to all work-items
    Host allocs and inits

- **Host mem:**
  - On the CPU

- Explicit memory management: You move data host->global->local and back
Synchronization

In-kernel barriers, queue barriers, events, timing
Various synchronization points (seen some already)

- Between work-items in a work-group
- Between kernels (and other commands) in a queue
- Between kernels (and other commands) in separate queues
- Between the host and the queues

- **Sync-ing can be enforced/implied by OpenCL**
  - An in-order queue: commands executed in order submitted

- **Or explicitly requested by user**
  - Out-of-order queue: commands scheduled by OpenCL
  - Barriers in kernels and queues
  - Events in queues
Work-item Synchronization

- Only possible within a work-group
  - Can't sync with work-items in other work-groups
  - Can't sync one work-group with another

- Use `barrier(type)` in kernel where `type` is
  - `CLK_LOCAL_MEM_FENCE`: ensure consistency in local mem
  - `CLK_GLOBAL_MEM_FENCE`: ensure consistency in global mem

- All work-items in work-group must issue the barrier() call and same number of calls

```c
__kernel void BadKernel(...) {
    int i = get_global_id(0);
    ...
    // ERROR: Not all WIs reach barrier
    if ( i % 2 )
        barrier(CLK_GLOBAL_MEM_FENCE);
}
```

```c
__kernel void BadKernel(...) {
    int i = get_local_id(0);
    ...
    // ERROR: WIs issue different number
    for ( j=0; j<i; j++ )
        barrier(CLK_LOCAL_MEM_FENCE);
}
```
Use with __local memory

- Barrier often used when initializing __local memory
  - Kernel must initialize local memory

```c
__kernel void kMat( const int n, __global float *A, __local float *tmp_arr )
{

    __local float tmp_arr[64];

    int gbl_id = get_global_id(0); // ID within entire index space
    int loc_id = get_local_id(0); // ID within this work-group
    int loc_sz = get_local_size(0); // Size of this work-group

    // For some reason we want to fill up the first half of the local array
    if (loc_id < loc_sz/2)
        tmp_arr[loc_id] = A[gbl_id];

    // All work-items must hit barrier. They'll all see a consistent tmp_arr[]
    barrier(CLK_LOCAL_MEM_FENCE);

    // Each work-item can now use the elements from tmp_arr[] safely.
    // Often used if we'd be repeatedly accessing the same A[] elements.
    for ( j=0; j<loc_sz; j++ )
        my_compute( gbl_id, tmp_arr[j], A );
}
```

Introduction to OpenCL
*Example: Parallel Reduction (CPU)*

- **Reduction of an array of numbers to a single value**
  - E.g. sum:
  - OpenMP on host forms partial sums in each thread
    - *Linear (serial) sum within thread*

```
void sumCPU( const int n, const float *x, float *res )
{
    float sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for ( int i=0; i<n; i++ )
        sum += x[i];
    *res = sum;
}
```

| input | 9 | 2 | 1 | 7 | 9 | 8 | 4 | 5 | 2 | 1 | 0 | 2 | 6 | 6 | 3 | 0 | 7 | 9 | 5 | 5 | 2 | 8 | 0 | 2 | 3 | 7 | 6 | 4 | 1 | 2 | 0 | 9 |
| partial sums | 45 | 20 | 38 | 32 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

```
\sum = 135
```

*OpenMP performs the final reduction for you*

- **Can recreate in OpenCL using work-groups**
Parallel Sum on GPU (1\textsuperscript{st} attempt)

- Use one work-item to perform a sum within a work-group
  - Inefficient – only one work-item forms the partial sum

```c
__kernel void sumGPU1( const uint n, __global const float *x,
                       __global float *partialSums ) {
    if ( get_local_id(0) == 0 ) { // Many idle work-items!
        float group_sum = x[get_global_id(0)];
        for ( int i=1; i<get_local_size(0); i++ )
            group_sum += x[get_global_id(0)+i]; // Should check (gid+i) < n
        partialSums[get_group_id(0)] = group_sum; // Write sum to output array
    }
    // Add barrier(CLK_GLOBAL_MEM_FENCE) if doing other work in kernel
}
```
Parallel Sum on GPU (2nd attempt)

- Memory optimization – copy to __local memory in parallel
  - Still inefficient – only one work-item forms the partial sum

```c
__kernel void sumGPU2( const uint n, __global const float *x,
                      __global float *partialSums, __local float *localCopy ) {
    localCopy[get_local_id(0)] = x[get_global_id(0)]; // Init the localCopy array
    barrier(CLK_LOCAL_MEM_FENCE); // All work-items must call
    if ( get_local_id(0) == 0 ) { // Many idle work-items!
        float group_sum = localCopy[0];
        for ( int i=1; i<get_local_size(0); i++ )
            group_sum += localCopy[i]; // Sum up the local copy
        partialSums[get_group_id(0)] = group_sum; // Write sum to output array
    }
    // Add barrier(CLK_GLOBAL_MEM_FENCE) if doing other work in kernel
}
```
Parallel Reduction (3\textsuperscript{rd} attempt)

- Improve on linear sum – binary sum

\begin{itemize}
  \item \texttt{global memory} \hspace{1cm} work-group\textsubscript{0} \hspace{1cm} work-group\textsubscript{1}
  \item \texttt{local memory}
\end{itemize}

\begin{align*}
\text{global memory} & : 9 \ 2 \ 1 \ 7 \ 9 \ 8 \ 4 \ 5 \ 2 \ 1\ 0 \ 2 \ 6 \ 6 \ 3 \ 0 \ 7 \ 9 \ 5 \ 5 \ 2 \ 8 \ 0 \ 2 \ 3 \ 7 \ 6 \ 4 \ 1 \ 2 \ 0 \ 9 \\
\text{local memory} & : 9 \ 2 \ 1 \ 7 \ 9 \ 8 \ 4 \ 5 \ 2 \ 1 \ 0 \ 2 \ 6 \ 6 \ 3 \ 0 \ 11 \ 3 \ 1 \ 9 \ 15 \ 14 \ 7 \ 5 \ 2 \ 1 \ 0 \ 2 \ 6 \ 6 \ 3 \ 0 \\
& \quad \vdots \\
& : 34 \ 31 \ 8 \ 14 \ 15 \ 14 \ 7 \ 5 \ 2 \ 1 \ 0 \ 2 \ 6 \ 6 \ 3 \ 0 \\
& \quad \vdots \\
& : 65 \quad 65 \quad 70
\end{align*}
Parallel Reduction (3rd attempt) Kernel

- Copy all work-group's values in to __local memory
  - Repeatedly half the work-group, adding one half to the other

```c
__kernel void sumGPU3( const uint n, __global const float *x,
                      __global float *partialSums, __local float *localSums )
{
    uint local_id   = get_local_id(0);
    uint group_size = get_local_size(0);

    // Copy from global mem in to local memory (should check for out of bounds)
    localSums[local_id] = x[get_global_id(0)];
    for (uint stride=group_size/2; stride>0; stride /= 2) { // stride halved at loop

        // Synchronize all work-items so we know all writes to localSums have occurred
        barrier(CLK_LOCAL_MEM_FENCE);

        // First n work-items read from second n work-items (n=stride)
        if ( local_id < stride )
            localSums[local_id] += localSums[local_id + stride]
    }

    // Write result to nth position in global output array (n=work-group-id)
    if ( local_id == 0 )
        partialSum[get_group_id(0)] = localSums[0];

    //... }
```
Improved Reduction (4\textsuperscript{th} attempt) Kernel

- Slight re-order to remove a couple of loop iterations
  - NB: set global\_work\_size to be half the input array length

```c
__kernel void sumGPU4( const uint n, __global float *x,
                      __global float *partialSums, __local float *localSums ) {

    uint global_id = get_global_id(0);          // Gives where to read from
    uint global_size = get_global_size(0);      // Used to calc where to read from
    uint local_id = get_local_id(0);            // Gives where to read/write local mem
    uint group_size = get_local_size(0);        // Used to calc initial stride

    // Copy from global mem in to local memory (doing first iteration)
    localSums[local_id] = x[global_id] + x[global_id + global_size];
    barrier(CLK_LOCAL_MEM_FENCE);
    for (uint stride=group_size/2; stride>=1; stride>>=1 ) { // >>=1 does same as /=2
        // First n work-items read from second n work-items (n=stride)
        if ( local_id < stride )
            localSums[local_id] += localSums[local_id + stride];

        // Synchronize so we know all work-items have written to localSums
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    // Last iter: write result to nth position in global x array (n=work\_group id)
    if ( local_id == 0 )
        x[get_group_id(0)] = localSums[0]+localSums[1];
}
```

Kernel based on slides by Tim Mattson, Intel, ISC2011
Still have \( n \) partial sums (\( n=\text{number of work-groups} \))
- Sum on host
- Sum on GPU
  - Linear sum (use one work-item)
  - Parallel reduction (use one work-group) provided \( n \) is small enough. Iterate if not.

- Both GPU options can be done with another kernel call
  - Data is still on the GPU (in the partialSums array)
    - Avoid a device-to-host transfer
    - Simply make another kernel call passing in the device memory object
    - DO NOT transfer back to host then pass back to device!
Exercise 3
Queue Synchronization

- Can sync between commands in a queue
- In-order execution (an in-order queue)
  - Each command executes after previous one finishes
  - Kernels will not run concurrently
  - Any memory transactions have a consistent view
  - Synchronization is implicit

![Queue Synchronization Diagram](image-url)
Out-of-order execution

- OpenCL can re-order commands in queue
- E.g., run ASAP scheduling on resource availability
- Useful for independent commands (kernels)

Some commands will need specific ordering / sync

- Memory transactions may overlap (clobber)
- Kernel execution order
Queue Synchronization

- Command queue sync functions work on individual queues (NB: all async – return to host immediately)

- Barrier

  ```c
  cl_int clEnqueueBarrier( cl_command_queue command_queue )
  ```

  - Ensures all commands in queue before this one will complete before any further commands execute

- Marker

  ```c
  cl_int clEnqueueMarker( cl_command_queue command_queue, 
                     cl_event *event )
  ```

  - Like Barrier but returns an event for other commands to wait on

- Events

  ```c
  cl_int clEnqueueWaitForEvents( cl_command_queue command_queue, 
                                 cl_uint num_events, 
                                 const cl_event *event_list )
  ```

  - Ensures a list of events already in the queue have completed before any further commands execute
Events provide status info about commands in the queue
- Want info about execution. Not same as *when* enqueued.
- A command can **wait on earlier events** and/or **generate** its own event

```c
cl_int clEnqueueReadBuffer( cl_command_queue command_queue,
    cl_mem buffer, cl_bool blocking_read,
    size_t offset, size_t cb,
    void *ptr, cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event )

cl_int clEnqueueWriteBuffer( cl_command_queue command_queue,
    cl_mem buffer, cl_bool blocking_write,
    size_t offset, size_t cb,
    const void *ptr, cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event )

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 )
```
Commonly used with mem transfers

- Ensure transfer completes before kernel runs in o-o-o queue
- Also required if commands are in separate queues

```c
cl_event memEvent; // One command writes status to this, others read it

// Create out-of-order queue
cq = clCreateCommandQueue(ctx, &d_id, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, 0);

// Enqueue non-blocking write-to-device. Returns immediately.
clEnqueueWriteBuffer(cq, A_d, CL_FALSE, 0, nbytes, A_h, 0, NULL, &memEvent);

// Enqueue kernel. Returns immediately. Kernel won't run until write has finished.
clEnqueueNDRangeKernel(cq, kernel, 1, NULL, &gSize, NULL, 1, &memEvent, NULL);
```
- Commands can wait on multiple events
  - All commands associated with the events in list must complete

```c
cl_event events[3]; // Going to wait on three commands

// Create out-of-order queue
cq = clCreateCommandQueue(ctx, &d_id, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, 0);

... // Enqueue kernels. o-o-o queue will run them in any order.
clEnqueueNDRangeKernel(cq, kernel1, 1, NULL, &gSize, NULL, 0, NULL, &events[0]);
clEnqueueNDRangeKernel(cq, kernel2, 1, NULL, &gSize, NULL, 0, NULL, &events[1]);
clEnqueueNDRangeKernel(cq, kernel3, 1, NULL, &gSize, NULL, 0, NULL, &events[2]);

// This kernel won't run until the previous three kernels have completed
clEnqueueNDRangeKernel(cq, kernel4, 1, NULL, &gSize, NULL, 3, events, NULL);

// Host can carry on - previous Enqueue calls are asynchronous
```
Event marker can sync between queues (in same ctx)
  - Placing a barrier in each queue won't work
  - Must make one queue wait on an event from another queue

```
Device 0

various queued commands
clEnqueueNDRangeKernel(cq1,...)
clEnqueueReadBuffer(cq1,...)
clEnqueueBarrier(cq1)
clEnqueueWaitForEvents(cq1,1,&marker)
clEnqueueWriteBuffer(cq1,...)
various queued commands

Device 1

various queued commands
clEnqueueReadBuffer(cq2,...)
clEnqueueMarker(cq2,&marker)
various queued commands
```

Will not run until both queues have reached their barrier/marker
Events contain status information (useful?)

- OpenCL host functions are asynchronous (or block until complete)
  - Impossible for them to return certain execution info to the host
- Request info via a command's event, after enqueuing

```c
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 )
```

```c
cl_int evinfo;    // Event flag is returned in an int
cGetEventInfo( event, CL_EVENT_COMMAND_EXECUTION_STATUS, sizeof(evinfo), &evinfo, NULL );
```

<table>
<thead>
<tr>
<th>CL_QUEUEED</th>
<th>Command has been enqueued in the command queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_SUBMITTED</td>
<td>Enqueued command has been submitted to the device</td>
</tr>
<tr>
<td>CL_RUNNING</td>
<td>Device is currently executing the command</td>
</tr>
<tr>
<td>CL_COMPLETE</td>
<td>The command has completed</td>
</tr>
<tr>
<td>Negative integer</td>
<td>Corresponds to an OpenCL error code</td>
</tr>
</tbody>
</table>

Event Information
Host / Queue Synchronization

- Can have the host wait (block) until enqueued commands have completed

- **Finish**

  ```c
  cl_int clFinish( cl_command_queue command_queue )
  ```

  - Host waits (blocks) until **all** commands in **the** queue have completed

- **Events**

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

  - Host waits (blocks) until all commands associated with the listed events have completed
  - Commands can be in different queues (same ctx)
- Events contain start/end time info (more useful?)
  - Must enable profiling in the queue, \textit{at creation}
  - Device times reported in nanoseconds

\begin{verbatim}
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 )

cl_ulong startT, endT, elapsedT;  // Timing info returned in a cl_ulong
cl_event event;                 // Event associated with the kernel exec

// Create a command queue with profiling enabled
cq = clCreateCommandQueue( ctx, &d_id, CL_QUEUE_PROFILING_ENABLE, NULL);
...
clEnqueueNDRangeKernel( cq, kernel, 1, NULL, &gSize, NULL, 0, NULL, &event );
clWaitForEvents( 1, &event );      // Could also use clFinish(cq);
clGetEventProfilingInfo( event, CL_PROFILING_COMMAND_START, sizeof(startT),
                        &startT, NULL);
clGetEventProfilingInfo( event, CL_PROFILING_COMMAND_END, sizeof(endT),
                        &endT, NULL );

elapsedT = endT - startT;
printf( "Elapsed time (ms): %10.5f\n", elapsedT*1.0e-06 );
\end{verbatim}
Profiling events contain several time stats

- All times are in nanoseconds, returned in a cl_ulong
- Can get timer resolution using

  \[
  \text{clGetDeviceInfo}(\ldots, \text{CL\_DEVICE\_PROFILING\_TIMER\_RESOLUTION}, \ldots);
  \]

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_PROFILING_COMMAND_QUEUED</td>
<td>Device time when the command was enqueued in a command queue by the host</td>
</tr>
<tr>
<td>CL_PROFILING_COMMAND_SUBMIT</td>
<td>Device time when the command was submitted to the compute device</td>
</tr>
<tr>
<td>CL_PROFILING_COMMAND_START</td>
<td>Device time when the command started execution on the device</td>
</tr>
<tr>
<td>CL_PROFILINH_COMMAND_END</td>
<td>Device time when the command finished execution on the device</td>
</tr>
</tbody>
</table>
- For completeness (and being tidy)
- Release
  ```c
  cl_int clReleaseEvent( cl_event event )
  ```
  - Free up memory when finished with event obj
Exercise 4
Miscellaneous

Task parallelism, pre-processor opt, warps divergence, async I/O
- Generally been concerned with data-parallel
- Use of queues is tasked oriented
  - Enqueue a data-parallel task
- OpenCL specifically supports tasks
  - Run a kernel using one work-item only

```c
cl_int clEnqueueTask( cl_command_queue command_queue, cl_kernel kernel,
                     cl_uint num_events_in_wait_list,
                     const cl_event *event_wait_list, cl_event *event )
```

- Equivalent to calling clEnqueueNDRangeKernel() with
  - `work_dim = 1, global_work_offset = NULL, global_work_size[0] = 1, local_work_size[0] = 1`
  - The get_global_id(0) etc functions behave as expected
If you find yourself doing initialization in one work-item

```c
__kernel void myKernel( const int n, ... )
{
    int i = get_global_id(0);
    if ( i == 0 ) {
        // Do some one-off initialization compute
        // E.g., init a _global lookup-table
    }
    // All work-items must wait for first one to do the init code
    barrier(CLK_GLOBAL_MEM_FENCE);
    ...
}
```

Create a separate kernel and use clEnqueueTask()
- Or initialize memory on the host (where possible)

Also useful on CPU devices with only a few cores
- Enqueue independent tasks to be scheduled by OpenCL
Can also enqueue a native C/C++ function in the command queue

- Not all devices allow this (CPUs might). Check using
  - `clGetDeviceInfo( ..., CL_DEVICE_EXECUTION_CAPABILITIES, ... )` for a value of `CL_EXEC_NATIVE_KERNEL`

```c
cl_int clEnqueueNativeKernel( cl_command_queue command_queue,
    void (*user_func)(void *), void *args,
    size_t cb_args, cl_uint num_mem_objects,
    const cl_mem *mem_list,
    const void **arg_mem_loc,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event )
```
Device resources (memory, registers etc) are limited

- Register usage by a work-item can be reduced
- Allows more work-items per work-group for example
- Use pre-processor constants when possible (see slide 84)

```c
// Compile kernel to use a pre-processor constant
char preproc[64];    // String to hold compiler flags
int exponent = 193;  // Whatever value you require
sprintf( preproc, "-DEXPONENT=%d", exponent );
clBuildProgram( program, 0, NULL, preproc, NULL, NULL );
```

```c
// Saw this kernel earlier. Now removed the 'exponent' arg.
__kernel void exponentor(__global int* data)
{
    int tid = get_global_id(0);
    int base = data[tid];
    int i;
    for (i=1; i < EXPONENT; ++i) {
        data[tid] *= base;
    }
}
```
Divergent Control Flow

- Recall: Nvidia warps (32 work-items), AMD Wavefronts (64 w-i's)
- Threads in a warp/wavefront execute in lock-step
- What about different execution paths?

Both paths computed

- All work-items in a warp will do both `func_a()` and `func_b()`
- The predicate is then used to mask out invalid values: Predicate is `(i\%2==0)`
- Instructions with false predicate don't read or write
- OK for _short_ conditionals

```c
__kernel void BadKernel()
{
    int i = get_local_id(0);
    if (i\%2 == 0) func_a();
    else         func_b();
}
```

Calc predicate mask

```
1010101010101010
```

keep valid values (blue)

invert predicate mask

```
```

keep valid values (green)
- Predicate mask may select an entire warp/wavefront
  - Lucky?
  - This case may be optimized
  - No divergence

```c
__kernel void NotSoBadKernel()
{
    int i = get_local_id(0);
    
    // Assuming warp of 32 work-items
    if (i/32 == 0) func_a();
    else func_b();
}
```

Calc predicate mask

| 1111111111111111 | 0000000000000000 |

func_a()  

func_b()
- Example use of two queues feeding one device
  - Overlap I/O in one queue with compute in another
  - Can't have dependencies on kernels

---

Asynchronous I/O

---

Example use of two queues feeding one device

- Overlap I/O in one queue with compute in another
- Can't have dependencies on kernels

---

Case study by Perhaad Mistry, Daba Schaa, Benedict Gaster, AMD OpenCL University Kit
- Dual I/O (e.g., Nvidia Fermi DMA)
  - Simultaneous bidirectional I/O
Example: Simulation kernel outputting results from each time-step to host

- Host will store each time-step result (e.g. a 2-D matrix)
- No host-to-device transfer (kernel operates on device buffers)
  - One kernel execution = one time-step
- Two queues, one for compute, one for device-to-host transfer

```c
cl_event elist[N]; // N iterations of simulation kernel

// Enqueue kernel in 'compute' queue to calculate first timestep (updates R_d[0])
clEnqueueNDRangeKernel(q0, kernel, 1, NULL, &gSize, NULL, 0, NULL, &elist[0]);
for (i=1; i<N; i++)
{
    // Enqueue a non-blocking read to copy device data to unique host array.
    // Won't execute until previous kernel execution has finished.
    clEnqueueReadBuffer(q1, R_d[i-1], CL_FALSE, 0, nbytes, R_h[i-1], 1, &elist[i-1], NULL);

    // Enqueue another timestep kernel in the 'compute' queue
    clEnqueueNDRangeKernel(q0, kernel, 1, NULL, &gSize, NULL, 0, NULL, &elist[i]);
}
```
Calling OpenCL from Fortran
Calling OpenCL from Fortran(90)

- No native OpenCL fortran library
- Call via intermediate C layer
  - C code maintains OpenCL state
  - Can return OpenCL objects (as pointers) to Fortran
  - Fortran passes args by-reference and *column-major* arrays!
  - USE iso_c_binding
  - Add ‘_’ to C function names in C code (not fortran)
    - int copy_data_to_gpu_( const int *nelems, const int *elemsize,
      const void *hostdata, void **device_ptr )
  - Compile using usual flags, link libOpenCL.so
    - pgf90 -i/path/to/modules -c ocltest.f90
    - pgcc -i/path/to/opencl/include -c opencl_layer.c
    - pgf90 ocltest.o opencl_layer.o -o ocltest -L/path/to/opencl/lib -I.OpenCL
Example Fortran Code

USE iso_c_binding  ! C-types

INTEGER, EXTERNAL :: init_opencl, stop_opencl
INTEGER, EXTERNAL :: allocate_memory_on_gpu, free_memory_on_gpu
INTEGER, EXTERNAL :: copy_data_to_gpu, copy_data_from_gpu
INTEGER, EXTERNAL :: compile_kernel_from_file, run_kernel

! C-types / variables
type (c_ptr) :: device_memobj1 = C_NULL_PTR
type (c_ptr) :: device_memobj2 = C_NULL_PTR
INTEGER :: dsize = sizeof(0.0d0)
CHARACTER(LEN=32) :: srcfilename = "kernels.cl"
CHARACTER(LEN=32), dimension(2) :: kernelnames = (/ "mykernel_method1"
                           //CHAR(0), &
                           "mykernel_method2"
                           //CHAR(0)  /)

! Create OpenCL context, alloc device memory objects
status = init_opencl()
status = allocate_memory_on_gpu(nelems1, dsize, device_memobj1)
status = allocate_memory_on_gpu(nelems2, dsize, device_memobj2)
...

! Populate host arrays, matrices etc
status = copy_data_to_gpu(nelems1, dsize, host_matrix, device_memobj1)
status = copy_data_to_gpu(nelems2, dsize, host_vector, device_memobj2)
...

! Compile kernel
status = compile_kernel_from_file( srcfilename, kernelnames(1), C_NULL_PTR )
status = run_kernel(nelems1, nelems2, device_memobj1, device_memobj2 )
...

! Retrieve results
status = copy_data_from_gpu( nelems2, dsize, device_memobj2, host_result_vector )
...

! Cleanup
status = free_memory_on_gpu( device_memobj1 )
status = free_memory_on_gpu( device_memobj2 )
status = stop_opencl()
#include <CL/cl.h>

// Private OpenCL state not exposed to fortran
typedef struct {
    cl_platform_id platform;
    cl_device_id device;
    cl_context_properties props[3]; // {CL_CONTEXT_PLATFORM, 0, 0}
    cl_context ctx;
    cl_command_queue queue;
    cl_program program;
    cl_kernel kernel;
} OCLinfo;
static OCLinfo *g_ocl = NULL;

// Initialise OpenCL context etc
int init_opencl_( void ) {
    cl_int err;

    g_ocl = (OCLinfo *)__malloc(1, sizeof(OCLinfo));

    err = clGetPlatformIDs(1, &g_ocl->platform, NULL );
    err = clGetDeviceIDs(g_ocl->platform, CL_DEVICE_TYPE_GPU, 1, &g_ocl->device, NULL );

    ... return 1;
}

// Allocate device memory object
int allocate_memory_on_gpu_(const int *nelems,
    const int *elemsize,
    void **device_ptr)
{
    cl_mem memobj;
    cl_int err;

    memobj = clCreateBuffer( g_ocl->ctx, CL_MEM_READ_WRITE,
        (*nelems * *elemsize), NULL, &err );
    *device_ptr = (void *)memobj; // Set output arg
    return 1;
}

// Copy host-to-device
int copy_data_to_gpu( const int *nelems,
    const int *elemsize,
    const void *hostdata,
    void **device_ptr )
{
    cl_mem memobj;
    cl_int err;

    memobj = *(cl_mem *)device_ptr; // Cast device ptr
    err = clEnqueueWriteBuffer(g_ocl->queue,
        memobj, CL_TRUE, 0,
        (*nelems * *elemsize),
        hostdata, 0, NULL, NULL);

    return ( err == CL_SUCCESS );
}

// Run a kernel (not generic)
int run_kernel(const int *karg1, const int *karg2,
    const void **device_ptr1,
    void **device_ptr2 )
{
    cl_int err;
    size_t global, local;

    // Set kernel args
    err = clSetKernelArg(g_ocl->kernel, 0, sizeof(int),
        (const void *)karg1);
    err = clSetKernelArg(g_ocl->kernel, 1, sizeof(int),
        (const void *)karg2);
    err = clSetKernelArg(g_ocl->kernel, 2, sizeof(cl_mem),
        (cl_mem *)device_ptr1);
    err = clSetKernelArg(g_ocl->kernel, 3, sizeof(cl_mem),
        (cl_mem *)device_ptr2);

    global = ...; // global work size
    local = ...; // work-group size
    err = clEnqueueNDRangeKernel(g_ocl->queue, g_ocl->kernel,
        1, NULL, &global, &local, 0, NULL, NULL );
    clFinish(g_ocl->queue);
    return 1;
}
For CSF (Nvidia), docs are available in

```
$CUDA_HOME/doc/
(don't forget module load libs/cuda/)
```

OpenCL_Programming_Overview.pdf
OpenCL_Programming_Guide.pdf
OpenCL_Best_Practices_Guide.pdf

- Do read the OpenCL specification available at

`http://www.khronos.org/registry/cl/`
GETTING MORE HELP

> reminder...
OpenCL pt2 and Drop-in

- **OpenCL Part2**
  - Tuesday 3rd July 10am - 1pm
  - Please book on ROPENCLADV

- **ITS research computing courses drop-in session**
  - Thursday 5th July 10am-2pm
  - Come along to get final help with our courses, exercises etc.
  - Book on if you want us to look at something in advance
Please give feedback on this course
  - Feedback is important for us to improve our courses.

The link for feedback is on the main courses page
  - http://wiki.rcs.manchester.ac.uk/community/Courses/
Research Computing Support

- Research Computing Community Website
  - Service information and support for research computing at The University of Manchester

http://wiki.rcs.manchester.ac.uk/community/

ITS-Research@manchester.ac.uk
Directorate of IT Services
Research Computing Services
http://www.manchester.ac.uk/researchcomputing
rcs@manchester.ac.uk