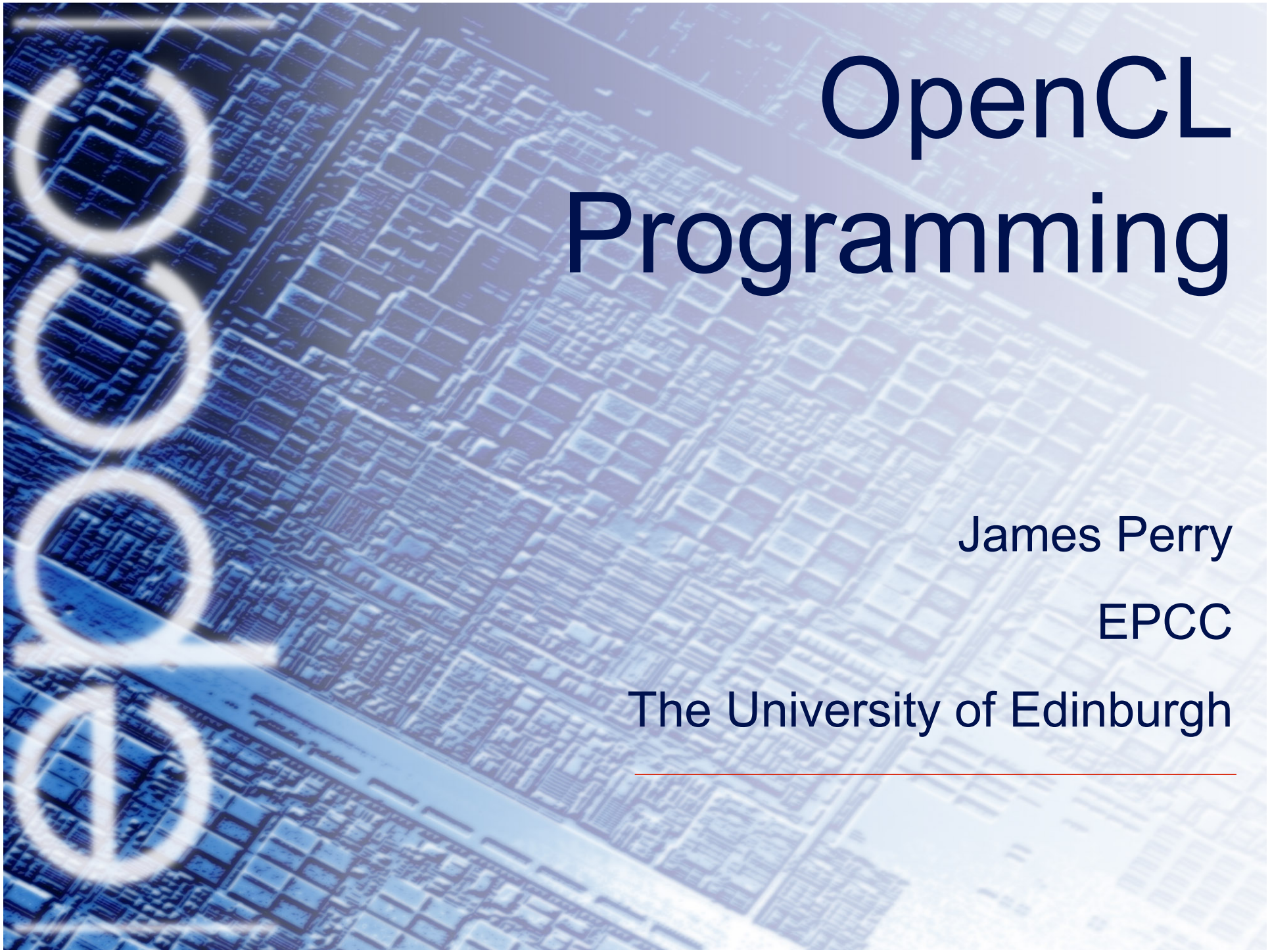


# OpenCL Programming

James Perry

EPCC

The University of Edinburgh

# OpenCL Programming

- Introducing OpenCL
- Important concepts
  - Work groups and work items
- Programming with OpenCL
  - Initialising OpenCL and finding devices
  - Allocating and copying memory
  - Defining kernels
  - Launching kernels
- OpenCL vs. CUDA

# Introducing OpenCL

# What is OpenCL?

- OpenCL is an open, royalty-free standard for parallel programming across heterogeneous devices
  - Originally developed by Apple
  - Now maintained by the Khronos Group
  - Adopted by many major chip vendors => cross platform support
    - Including Intel, ARM, AMD, NVIDIA
  - Same program can run on CPUs and various types of accelerator
- OpenCL 1.0 released in 2008
- Current version is 1.2, released November 2011
  - Further details at http://www.khronos.org/opencl/

# What is OpenCL? (2)

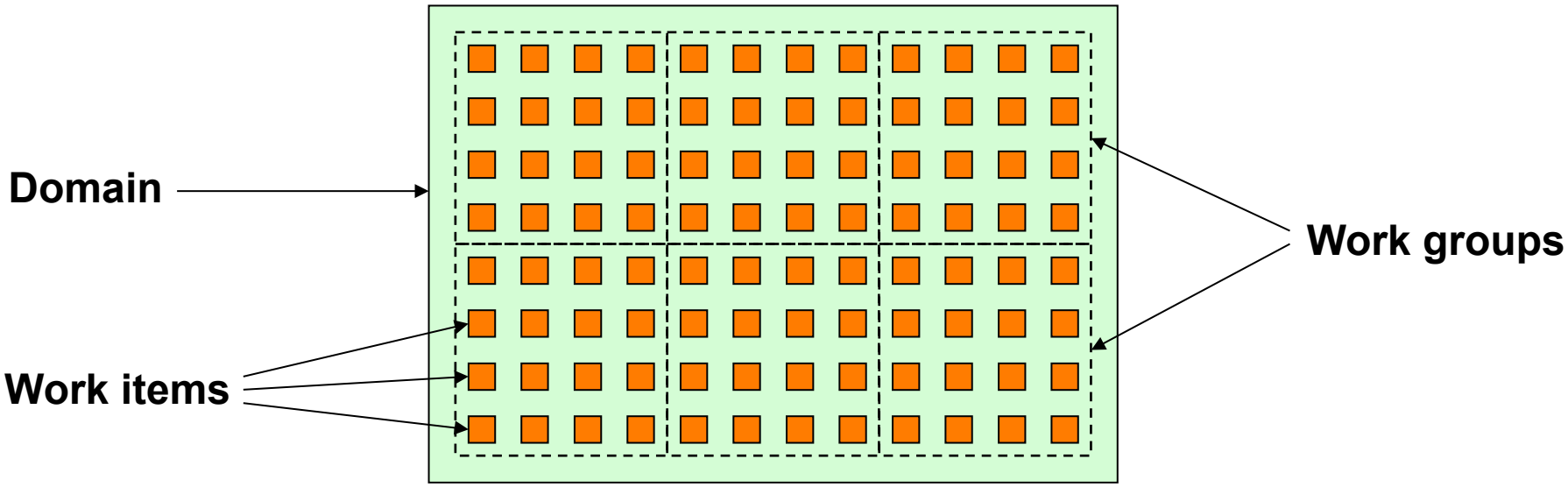
- Consists of:
  - A programming language for writing *kernels* (code fragments that run on devices), based on standard C
  - An API (used for querying and initialising devices, managing memory, launching kernels, etc.)

- Kernel functions are compiled at runtime for whatever device is available
  - Same code can run on NVIDIA GPU, AMD GPU, Intel CPU, more exotic accelerators…

# Important Concepts

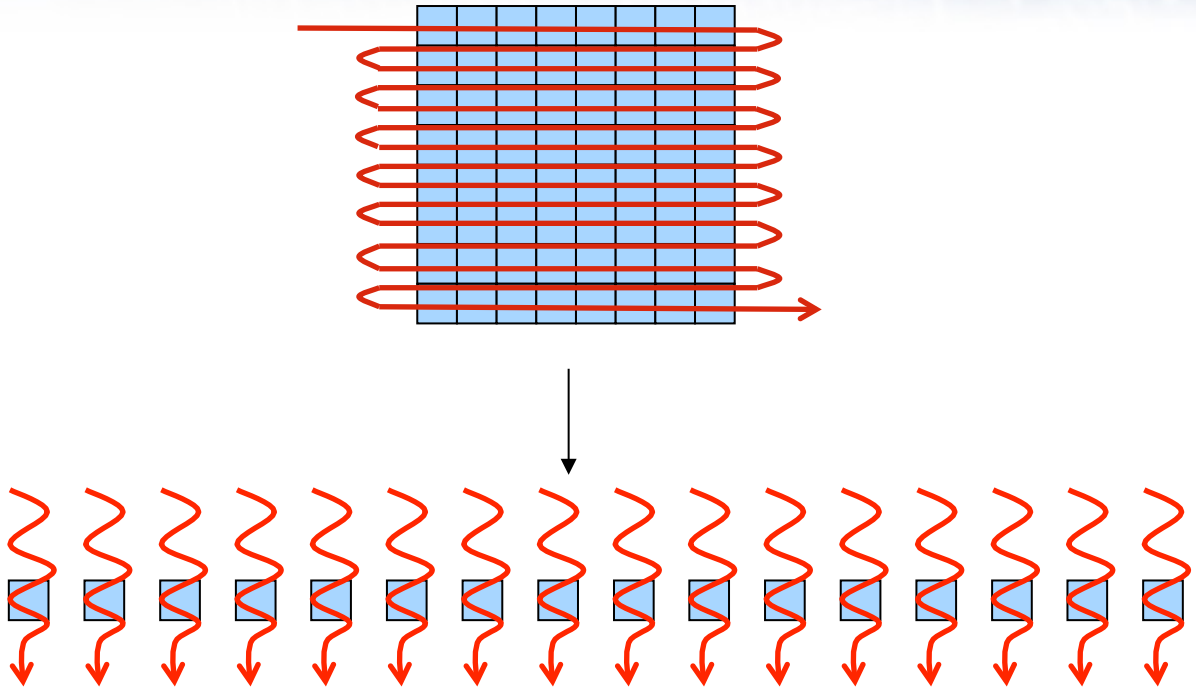
# Work Items and Work Groups

- For OpenCL, each problem is composed of an array of *work items*
  - May be a 1D, 2D or 3D array
- The domain is sub-divided into *work groups*, each consisting of a number of work items




- This domain has *global dimensions* 12x8 and *local dimensions* 4x4

# Work items and kernels



- When kernel function is invoked, it runs on every work item in the domain independently
- Many work items will be processed in parallel
  - Depending on the hardware

# Traditional looped vector addition

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

# OpenCL vector addition

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

- No loop – each kernel instance operates on a single element
- Global ID determines position in domain

# Work groups

- All the work items within a work group will be scheduled together on the same processing unit
  - E.g. same SM on NVIDIA GPU
- Synchronisation is possible between items in the same work group
- But not between different work groups
- Items in the same work group also share the same local memory

# Programming with OpenCL

# Initialising OpenCL

- Quite a long-winded process; typically need to:
  - Find the *platform* you wish to target
    - System may have multiple platforms (e.g. CPU and GPU)
  - Find the target *device* within that platform
  - Create a *context* and a *command queue* on the target device
  - Compile your kernels for the target device
    - Kernels typically distributed as source so they can be compiled optimally for whatever device is present at runtime
- OpenCL has many API functions to help with this
- See practical for an example of how to do this

# Allocating Memory

- Global device memory can be allocated with:

```
mem = clCreateBuffer(context, flags, size,
                     host_ptr, error);
```

- and freed with:

```
clReleaseMemObject(mem);
```

# Copying to and from device memory

- Copying between host and device memory is initiated by `clEnqueueWriteBuffer` and `clEnqueueReadBuffer`
- Example:

```
clEnqueueWriteBuffer(queue, d_input, CL_TRUE,
    0, size, h_input, 0, NULL, NULL);
// ... perform processing on device ...
clEnqueueReadBuffer(queue, d_output, CL_TRUE,
    0, size, h_output, 0, NULL, NULL);
```

- Can also perform non-blocking transfers, synchronise with events, partial transfers, etc.

# Writing kernels

- OpenCL kernels are functions that run on the device
  - Kernel processes a single work item
- Written in separate source file from host C code
  - Often `.cl` file extension
  - Usually compiled at runtime
- Mostly C syntax but with some extensions and some restrictions
- Various OpenCL API calls available
  - For syncronisation etc.

# Declaring a kernel

- Like a normal C function declaration, but proceeded by `__kernel`:

```
__kernel void add_vectors(__global float *a,
                          __global float *b,
                          __global float *c,
                          int n)
```

- `__global` specifies that the pointers point to global memory on the device – i.e. main memory that is accessible to all work groups
- `__local` and `__constant` qualifiers also available

# Some OpenCL APIs

- `get_global_id(dimension)` returns the position of the current work item within the whole domain, e.g.:

```
int x = get_global_id(0);
int y = get_global_id(1);
```

- `get_local_id(dimension)` returns position within the local work group
- `barrier()` synchronises all items within the local work group
  - useful for co-ordinating accesses to local memory

# Vector datatypes

- Vectors of length 2, 4, 8 or 16 are supported
  - Both integer and floating point
- May improve performance on hardware that has vector instructions
- Simple to use:

```
float4 v1 = (float4)(1.0, 2.0, 3.0, 4.0);
float4 v2 = (float4)(0.0, -0.5, -0.3, 7.9);
v1 -= v2;
```

- Various specific vector operations and functions also available

# Launching Kernels

- Call `clSetKernelArg` to set each argument
- Call `clEnqueueNDRangeKernel` to launch the kernel
- `clFinish` can be used to wait for all work groups to complete
- Example:

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_input);
clSetKernelArg(kernel, 1, sizeof(int), &size);
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, globalsize,
                       localsize, 0, NULL, NULL);
clFinish(queue);
```

# OpenCL vs. CUDA

# OpenCL vs. CUDA

- Similar in many ways
- Different terminology
  - CUDA has *threads* and *blocks*
  - OpenCL has *work items* and *work groups*
- OpenCL is cross platform, CUDA is NVIDIA-only
  - OpenCL supports NVIDIA and AMD GPUs, CPUs, etc.
- CUDA currently more mature
  - On NVIDIA hardware, CUDA is faster and better supported
- OpenCL API tends to be more verbose
  - But also more flexible
- OpenCL kernels normally compiled at runtime

# Summary

- OpenCL allows programming of heterogeneous processors and accelerators using a common language and API
  - Supports both NVIDIA and AMD GPUs, as well as CPUs

- For more information, see:

  http://www.khronos.org/opencl/
  http://developer.nvidia.com/opencl
  http://developer.amd.com/zones/OpenCLZone/Pages/default.aspx