EEE4120F

High Performance Embedded Systems

Lecture 4: OpenCL

A Language for Programming Digital Accelerators

How to Program In OpenCL

Presented by

Simon Winberg

Some slides have voice annotations

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

EEE4120F Lab Sessions / Bookings?

Term 1: Thursday 3pm - 5pm (Blue Lab)
Tuesday 11am – 12pm (catchup/additional slot)

Term 2: same (presumably)

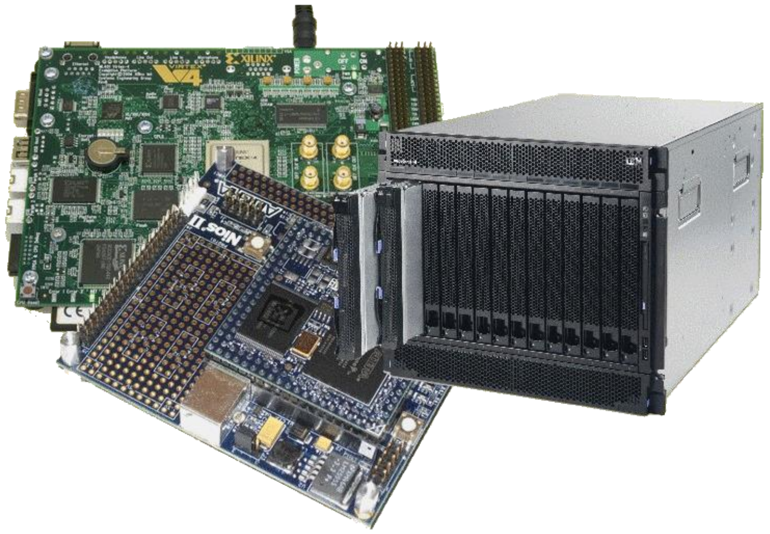
Note: there was availability of Monday 11am also (wanted Mon or Tue option, not both)

OpenCL Lecture Overview

- Why OpenCL
- Brief overview of OpenCL
  - Abstractions & platform model
  - OpenCL Scenario
- OpenCL Programming
  (Preparation for Prac 2)

The following slides are based on a presentation prepared by Dr. Suleyman Demirsoy, DSP Technology Specialist, Intel Programmable Solutions Group

See also paper in reading list:
"06502816 - OpenCL Overview, Implementation, and Performance Comparison.pdf"
(this paper is supplementary reading, not prescribed)



Licensing details last slide

Note: to save time and avoid boredom,
some of these slides are assigned as:

Prescribed reading

May come up in a test

Supplementary reading

Won't be examined

Why OpenCL?

EEE4120F

Why OpenCL...

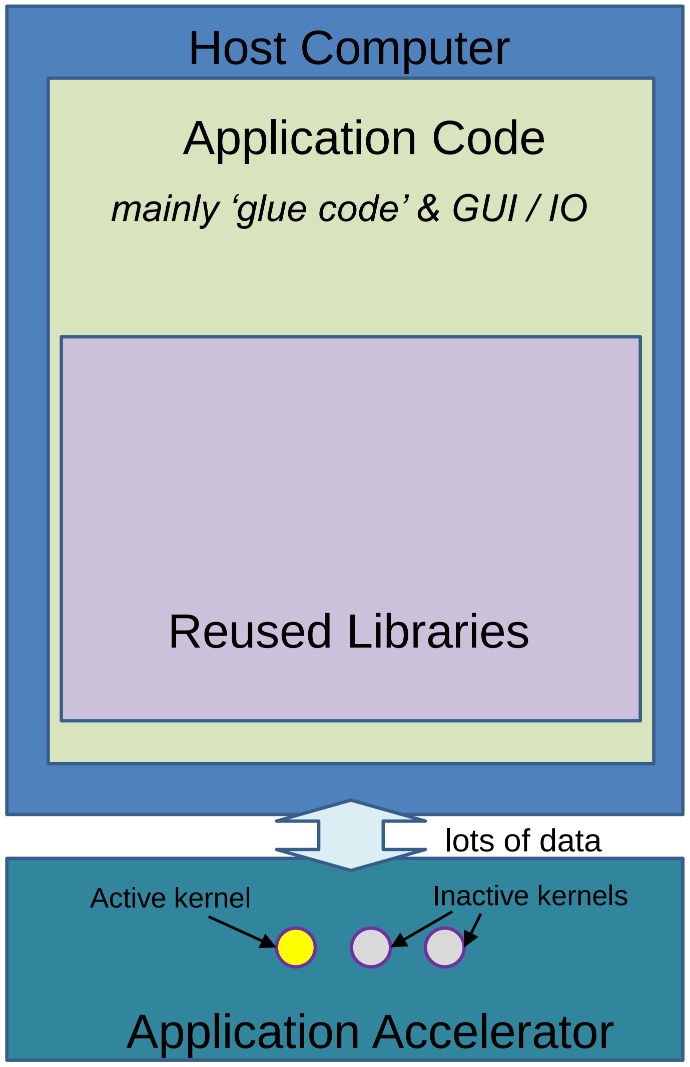
- Main reason is because there's massive Increasing demands for more functionality and performance
- ... and clients also want it soon (i.e. asap to beat the competition etc).
- But it's not just that, it's also to try and streamline and perhaps even mitigate the complexity of modern designs.

OpenCL
– for acceleration & other kernels

- It is also a response for needs to cater for
  - BIG DATA
  - HIGHLY INTEGRATED SYSTEMS
  - SPECIAL-PURPOSE ACCELERATORS

OpenCL – for acceleration & other kernels

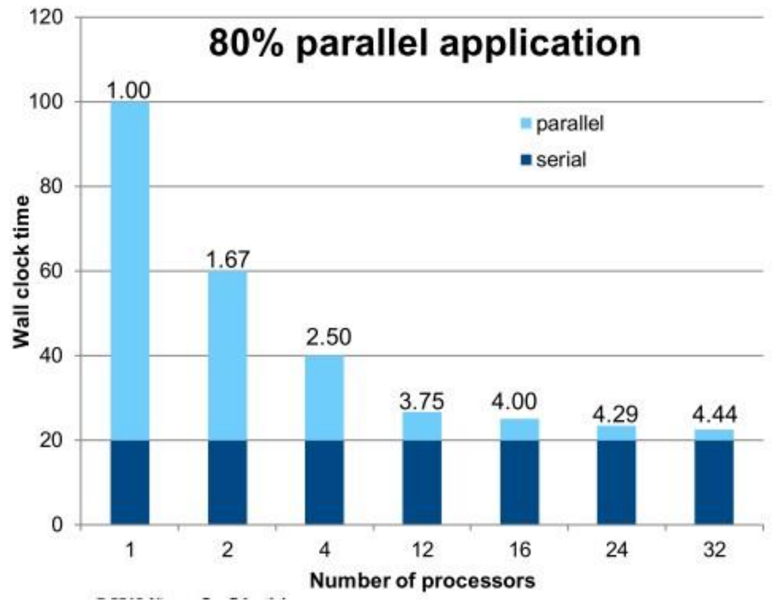
- OpenCL is not (currently) meant for creating most of the application code.
- It is designed around developing accelerated kernels that are highly parallel or exercise the specialized hardware (not necessarily an application accelerator)
- The diagram on right explains where OpenCL fits in (it is the same as to where Digital Accelerators generally fit in to a computer system)

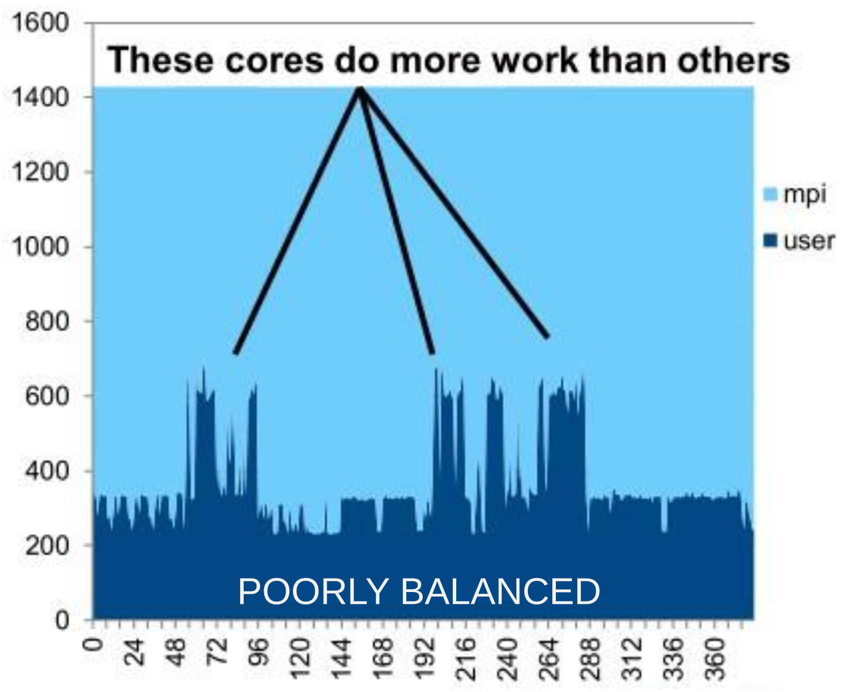


Why OpenCL…

- Not all applications scale well linearly
- Not all apps load balance across all cores equally

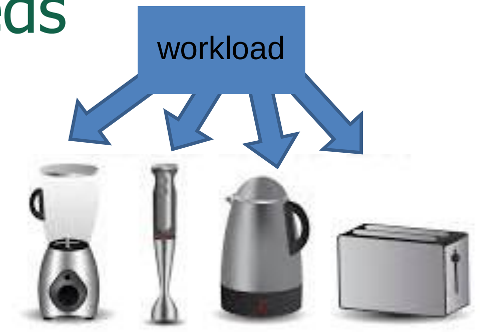
LIMITED SCALABILITY





Why OpenCL…

- Heterogeneous Computing Systems as a potential solution to the problems of limited scalability and balancing
  - Distribute your different processing needs among processing cores better suited to the particular types of processing.

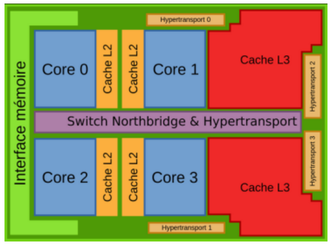


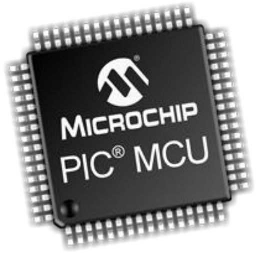
Essentially like matching the equipment with the task

Definition:
Heterogeneous computing refers to systems that use more than one kind of processor. These are systems that gain performance not just by adding the same type of processors, but by adding dissimilar processors, usually incorporating specialized processing capabilities to handle particular tasks (or entire applications).

Examples of devices form which heterogeneous computing systems are composed

- Multi-core, general purpose, central processing units (CPUs)
  - Include multiple execution units ("cores") on the same chip
- Digital Signal Processing (DSPs) processors
  - Optimized for the operational needs of digital signal processing
- Graphics Processing units (GPUs)
  - Heavily optimized for graphics processing
- Field Programmable Gate Arrays (FPGAs)
  - Custom architectures for each problem solved
  - SoC FPGAs combine CPU+FPGA in single device





Challenges to developers

- Various applications becoming bottlenecked by scalable performance requirements
  - e.g. Object detection and recognition, image tracking and processing, cryptography, cloud, search engines, deep packet inspection, etc…
- Overloading CPUs capabilities
  - Frequencies capped
  - Processors keep adding additional cores
  - Need to coordinate all the cores and manage data

Source: points based on course notes by B. Subramanian, Department of Computer Science, University of Texas, Austin

Challenges to developers (cont.)

- Product life cycles are long
  - GPUs lifespan is short (which goes with the problem of getting replacements of the same model in the future)
  - Require re-optimization and regression testing between generations
  - (not limited to accelerators/GPUs!! It could be some cryptic control code for a robot, e.g. multiple processors of an integrated device)
- Maintaining coherency throughout scalable system
- Support agreement for GPUs costly
- Power dissipation of CPUs and GPUs limits system size

Source: points based on course notes by B. Subramanian, Department of Computer Science, University of Texas, Austin

OpenCL – abstractions and platform model

EEE4120F

Overview of OpenCL

- Open standard for parallel programming across heterogeneous devices
- Devices can consist of CPUs, GPUs, embedded processors etc. – uses all the processing resources available
- Includes a language based on C99 for writing kernels and API used to define and control the devices
- Parallel computing (or hardware acceleration*) through task-based and data-based parallelism

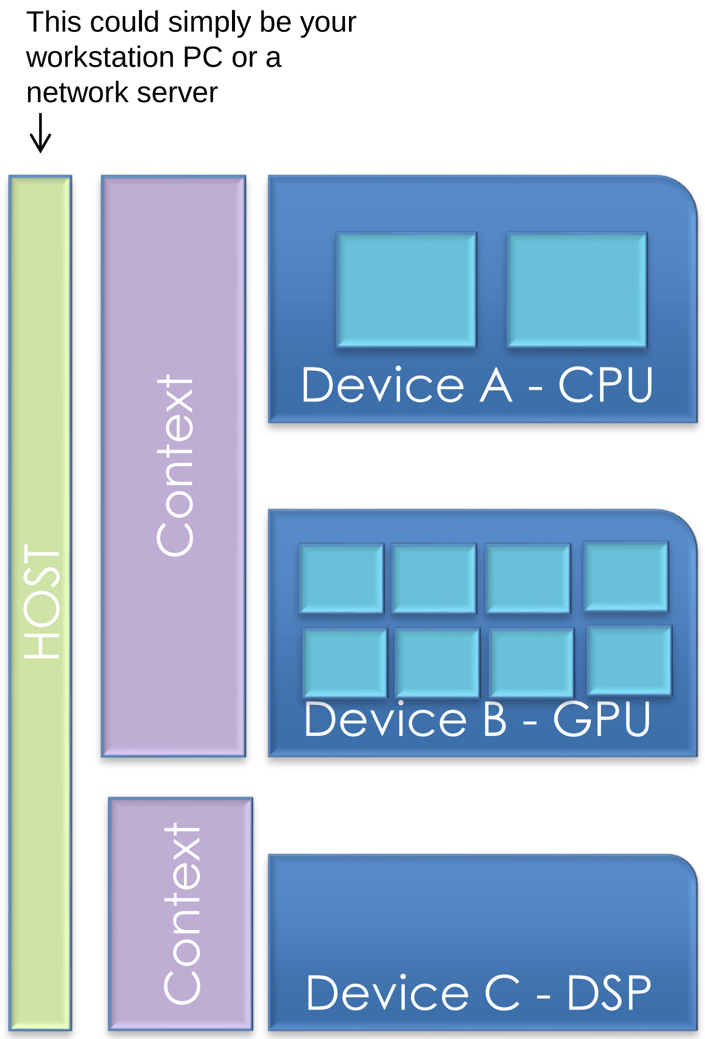
* Which is more generic e.g. if the hardware is faster but not necessarily more parallel

Purposes of OpenCL

- Use all computational resources of the system via a consistent programming language
- Greater platform independence
- Provides both a data and task parallel computational model
- A programming model which abstracts the specifics of the underlying hardware
- Much flexibility in specifying accuracy of floating-point computations
- Supports desktop, server, mobile, custom, etc.

The OpenCL Platform Model

- Host connected to one or more OpenCL devices
- Device consists of one or more cores
- Execution per processor may be SIMD or SPMD*
- Contexts group together devices and enable inter-device communication


This could simply be your workstation PC or a network server
HOST
Context
Context
Device A - CPU
Device B - GPU
Device C - DSP

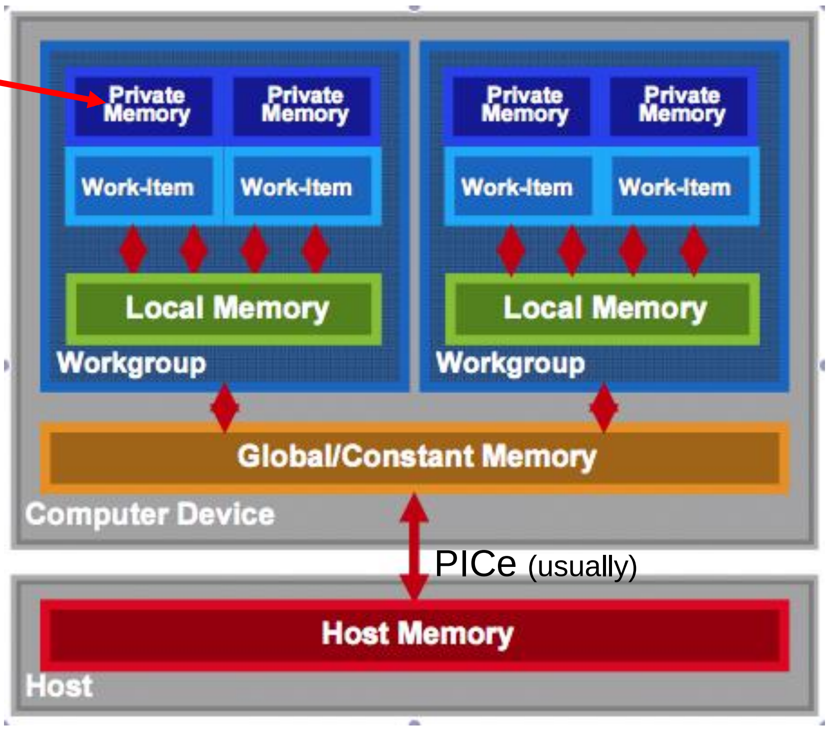
* Single Program, Multiple Data - where cores are running the same program but not necessarily the same instructions at the same time

Slide adapted from course notes by B. Subramanian, Department of Computer Science, University of Texas, Austin

OpenCL Memory model

Prescribed reading

- Private memory: available per work item
- Local memory: shared in workgroup
- NB: No synchronization between workgroups
- Synchronization possible between work items in a common workgroup
- Global/constant memory accessible by any work-items (no guarantee to be synchronized)
- Host memory: access through the CPU

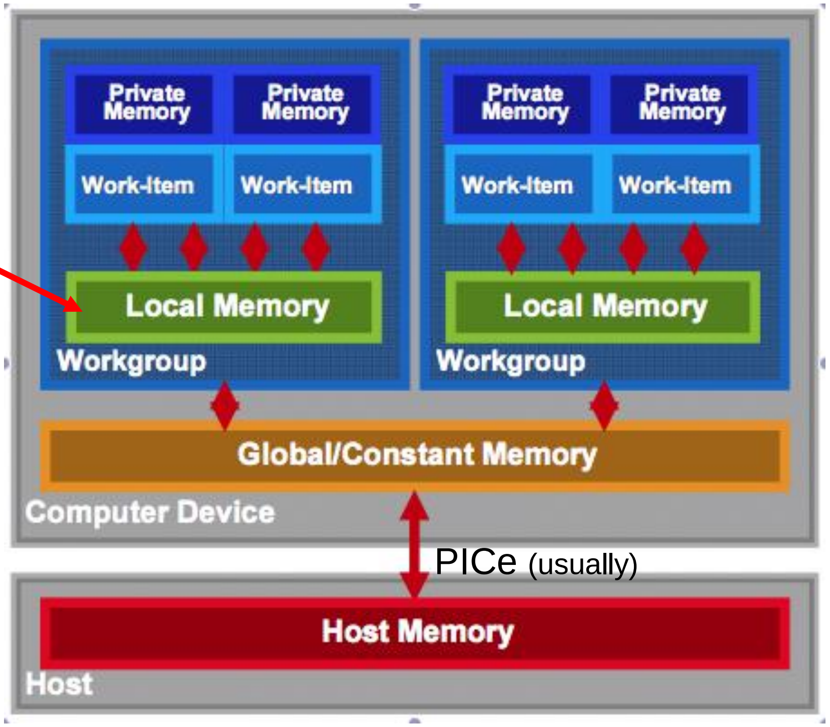


- Memory management is explicit…
- Data moved from:
 host->global->local and back

Prescribed reading

OpenCL Memory model

- Private memory: available per work item
- Local memory: shared in workgroup
- NB: No synchronization between workgroups
- Synchronization possible between work items in a common workgroup
- Global/constant memory accessible by any work-items (no guarantee to be synchronized)
- Host memory: access through the CPU

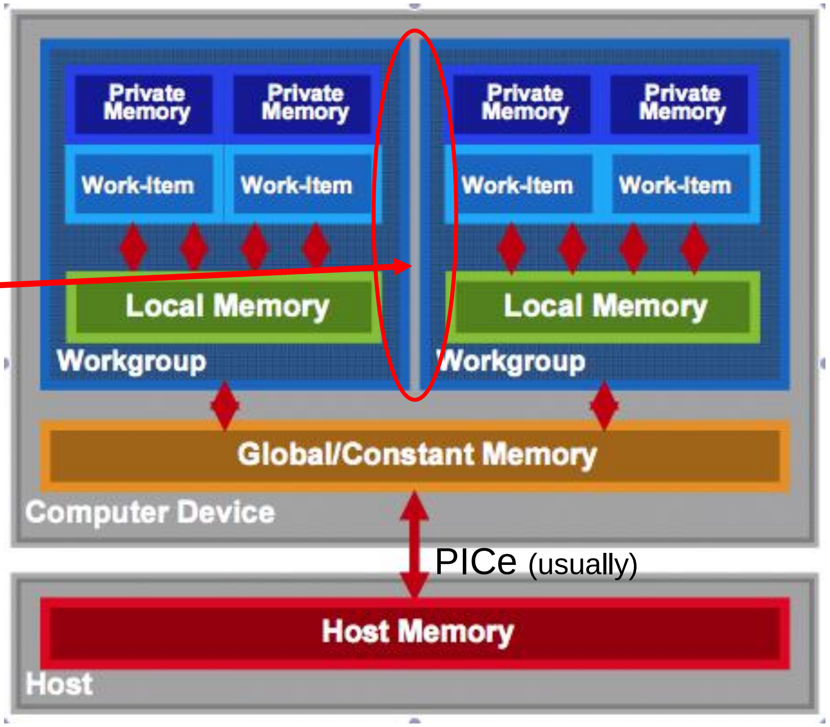


- Memory management is explicit…
- Data moved from:
  host->global->local and back

OpenCL Memory model

Prescribed reading

- Private memory: available per work item
- Local memory: shared in workgroup
- NB: No synchronization between workgroups
- Synchronization possible between work items in a common workgroup
- Global/constant memory accessible by any work-items (no guarantee to be synchronized)
- Host memory: access through the CPU

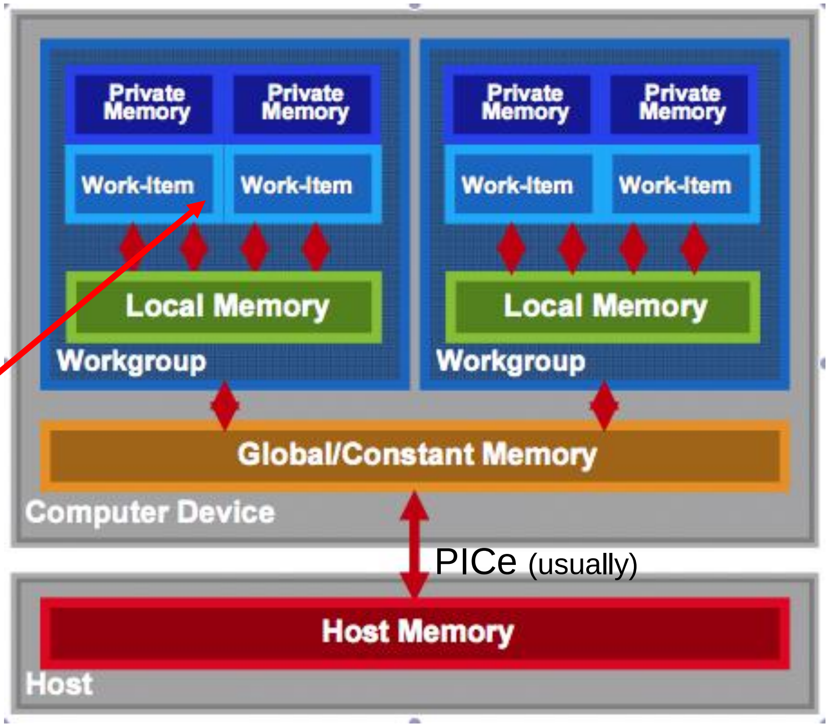


- Memory management is explicit…
- Data moved from: host->global->local and back

OpenCL Memory model

Prescribed reading

- Private memory: available per work item
- Local memory: shared in workgroup
- NB: No synchronization between workgroups
- Synchronization possible between work items in a common workgroup
- Global/constant memory accessible by any work-items (no guarantee to be synchronized)
- Host memory: access through the CPU

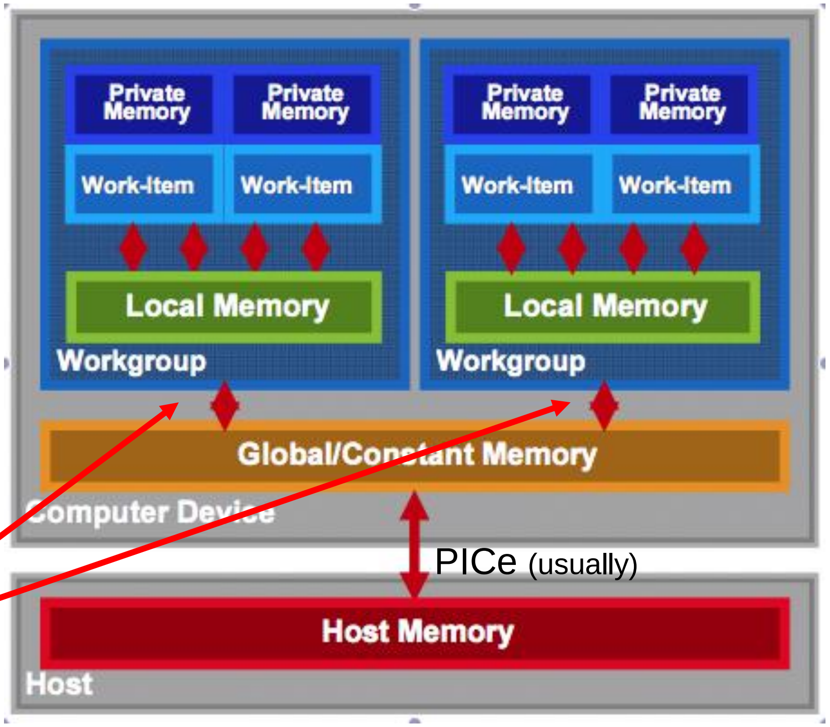


- Memory management is explicit…
- Data moved from: host->global->local and back

OpenCL Memory model

Prescribed reading

- Private memory: available per work item
- Local memory: shared in workgroup
- NB: No synchronization between workgroups
- Synchronization possible between work items in a common workgroup
- Global/constant memory accessible by any work-items (no guarantee to be synchronized)
- Host memory: access through the CPU

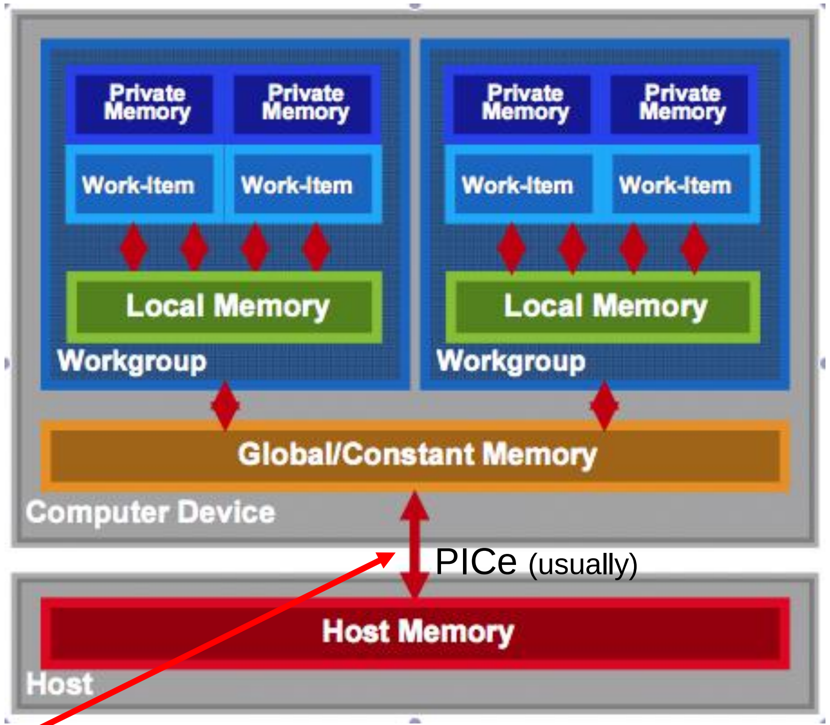


- Memory management is explicit…
- Data moved from: host->global->local and back

OpenCL Memory model

Prescribed reading

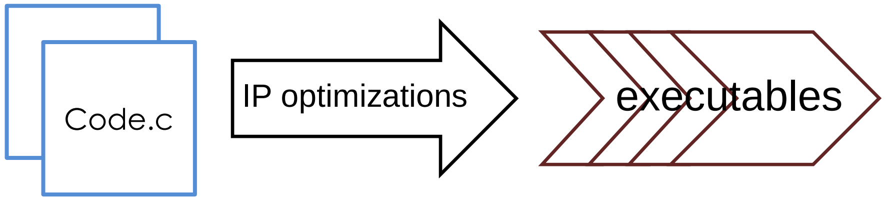
- Private memory: available per work item
- Local memory: shared in workgroup
- NB: No synchronization between workgroups
- Synchronization possible between work items in a common workgroup
- Global/constant memory accessible by any work-items (no guarantee to be synchronized)
- Host memory: access through the CPU



- Memory management is explicit…
- Data moved from: host->global->local and back

Advantages of the OpenCL JIT* Compiler

- HLS (high level synthesis) phase leverages the OpenCL compiler
- You can (if you want to) still specify and adjust:
  - High performance datapaths
  - Automatic pipelining
  - New QoR enhancements as they are developed
  - DSPB back-end optimizations
  - Memory optimizations
- In addition to:
  - Control constraints (latency, fmax, area, …)
  - Control interfaces (stall, valid, Avalon-MM, …)
  - Control architecture (memory configuration, scheduling, …)
  - IP core verification flows



* JIT = Just In Time

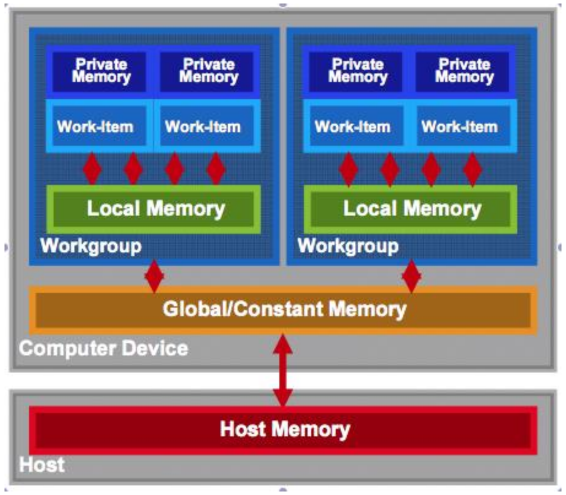
OpenCL Coding

EEE4120F

OpenCL Programming Model

Prescribed reading

- Kernel
  - basic unit of execution – data parallel
- Program
  - collection of kernels and related functions
- Kernels executed across a collection of work-items
  - one work-item per computation
- Work-items
  - Independent processing tasks; grouped into workgroups
- Workgroups
  - Work-items that executed together on one device
- Workgroups are executed independently can take place simultaneously or via specific schedule
- Applications structuring
  - Queue kernel instances for execution in-order, but they may be executed in-order or out-of-order



Work items and OpenCL memory model

OpenCL Objects (work task elements)

Prescribed reading

- Devices: multiple cores on CPU/GPU together taken as a single device
  - Kernels executed across all cores in a data-parallel manner
- Contexts: Enable sharing between different devices
  - Devices must be within the same context to be able to share
- Queues: used for submitting work, one per device
- Buffers: simple chunks of memory like arrays; read-write access
- Images: 2D/3D data structures
  - Access using read_image(), write_image()
  - Either read or write within a kernel, but not both

OpenCL Kernel Objects

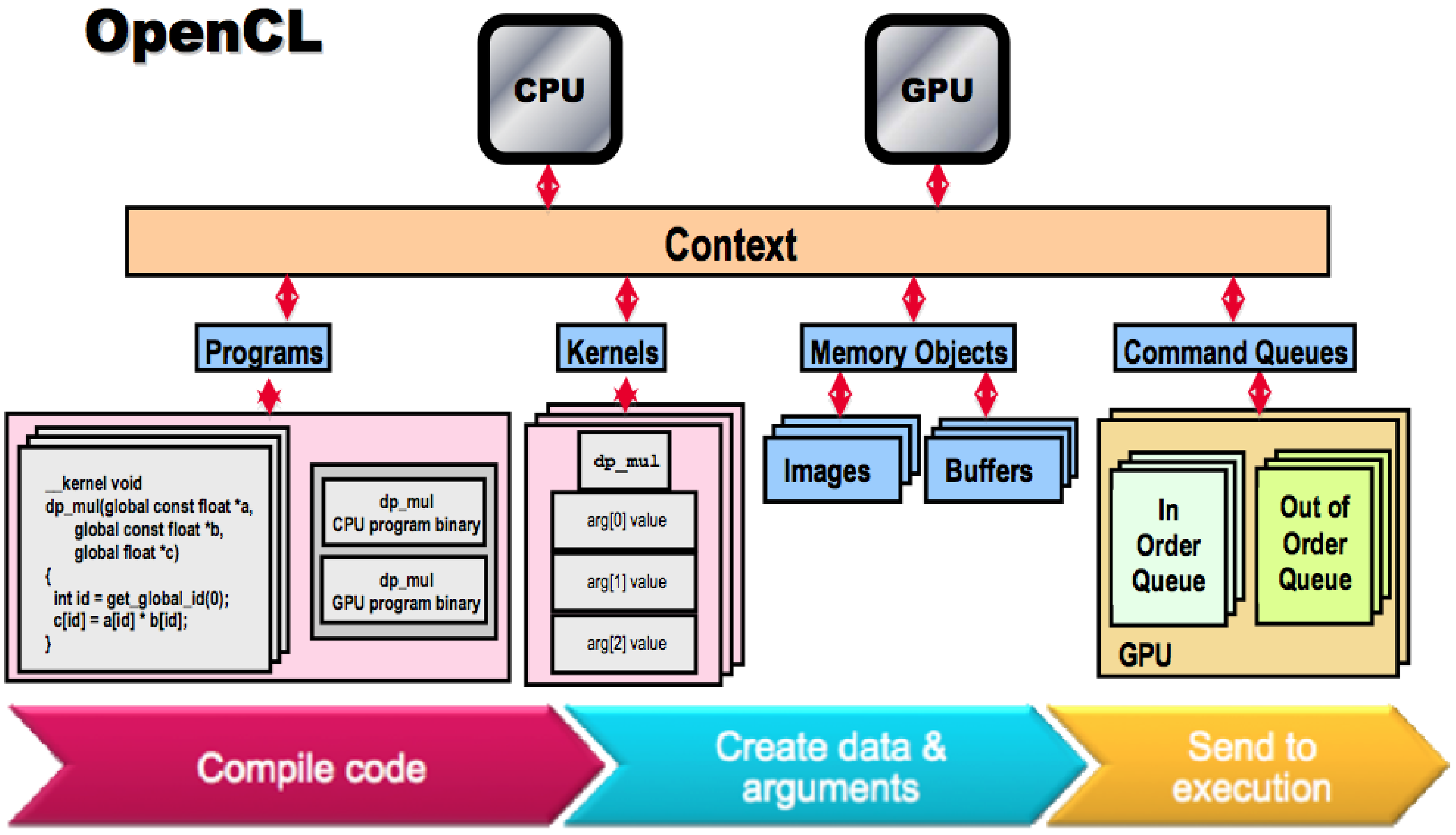
- Declared with a __kernel qualifier
- Encapsulate a kernel function
- The kernel objects created after the executable is built
- Execution
  - Set the kernel arguments
  - Enqueue the kernel
- Kernels are executed asynchronously
- Events used to track the execution status
  - Used for synchronizing execution of two kernels
  - clWaitForEvents(), clEnqueueMarker() etc.

OpenCL Program Objects

- Encapsulate
  - A program source/binary
  - List of devices and latest successfully built executable for each device
  - List of kernel objects
- Kernel source specified as a string can be provided and compiled at runtime using clCreateProgramWithSource() – platform independence
- Overhead – compiling programs can be expensive
  - OpenCL allows for reusing precompiled binaries

OpenCL Overall Pipeline

A view on how some of the objects fit in to the execution model



```
__kernel void
dp_mul(global const float *a,
      global const float *b,
      global float *c)
{
 int id = get_global_id(0);
 c[id] = a[id] * b[id];
}
```

(can skip this slide, may be useful to refer to when writing a OpenCL program)

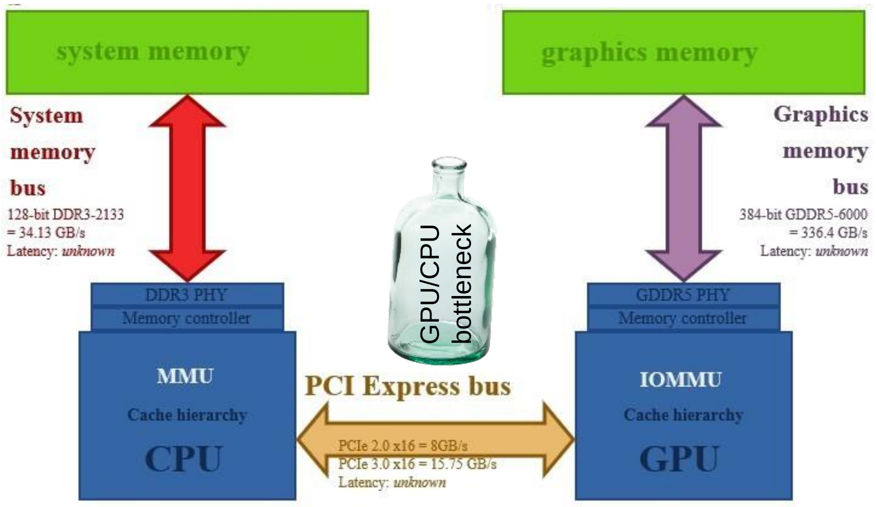
OpenCL C Language

C99 (previously 'C9X') is an informal name for ***ISO/IEC 9899:1999***, a past version of the C programming language standard. It extends the previous version (C90) with new features for the language and the standard library. Helps implementations make better use of available computer hardware and compiler technology. Source & more info: https://en.wikipedia.org/wiki/C99

- Derived from ISO C99
- Non standard headers, function pointers, recursion, variable length arrays, bit fields
- Added features: work-items, workgroups, vector types, synchronization
- Address space qualifiers
- Optimized image access
- Built-in functions specific to OpenCL
- Data-types
  - Char, uchar, short, ushort, int, uint, long, ulong
  - Bool, intptr_t, ptrdiff_t, size_t, uintptr_t, half
  - Image2d_t, image3d_t, sampler_t
  - Vector types – portable, varying length (2,4,8,16), endian safe
  - Char2,ushort4,int8,float16,double2 etc.

Many of these are recognizable from C!

Supplementary reading



OpenCL Programming

EEE4120F

How to code an OpenCL kernel…

OpenCL C Language: dealing with address spaces

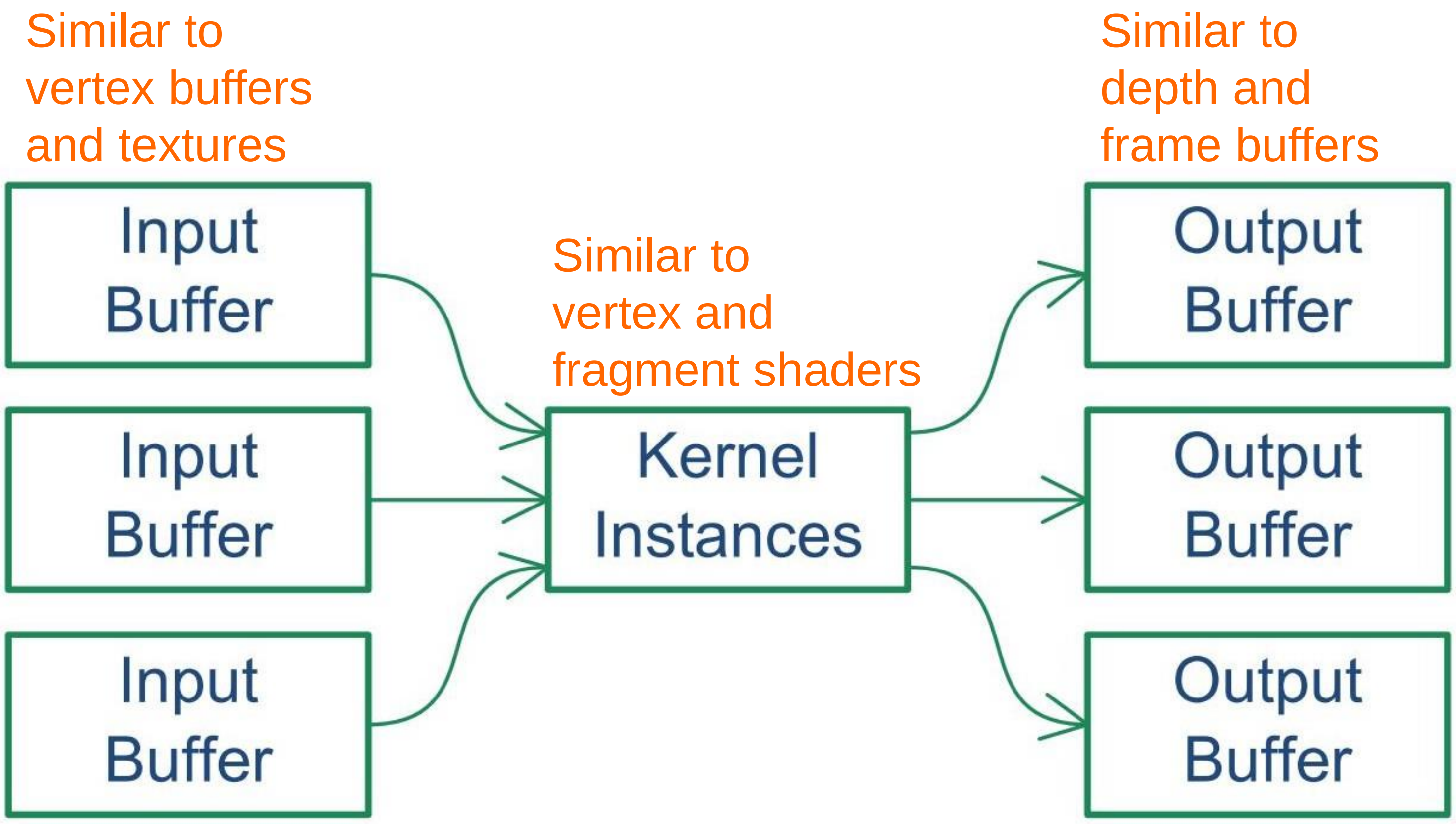
Supplementary reading

- Address spaces
  - Kernel pointer arguments must use global, local or constant
  - Default for local variables is private
  - Image2d_t and image3d_t are always in global address space
  - Global variables must be in constant address space
  - NB: Casting between different address spaces undefined

(skip in lecture, may be useful to refer to when writing a OpenCL program)

Conceptual view of how an OpenCL kernel fits in

Prescribed reading



Similar to vertex buffers and textures
Similar to depth and frame buffers
Input Buffer
Input Buffer
Input Buffer
Similar to vertex and fragment shaders
Kernel Instances
Output Buffer
Output Buffer
Output Buffer

(1) Set up & transfer buffers to GPU

(2) Invoke the kernel

(3) Transfer output buffer to CPU

… comparing to GPU / CUDA type approach …

OpenCL C Language

Prescribed reading

- Work-item and workgroup functions
  - get_work_dim() : number dimensions of tasks, returns 1 for a queue added with clEnqueueTask
  - get_global_size() : work-item ID
  - get_group_id(), get_local_id() :
- Vector operations and components are pre-defined
- Kernel functions
  - get_global_id() – gets the work item ID
- Conversions
  - Explicit – convert_destType<_sat><_roundingMode>
  - Reinterpret – as_destType
  - Scalar and pointer conversions follow C99 rules
  - No implicit conversions/casts for vector typs

See image on slide 25.

Many of these are not so common in C!

(won't be asked about in tests)

(skip in lecture, may be useful to refer to when writing a OpenCL program)

https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueTask.html

Initialisation – on CPU side

1. Get platform ID
2. Get device ID
3. Create Context
4. Load kernel .cl code file(s)
5. Create OpenCL program (may have multiple kernels, provide list of kernel names)
6. Build program
7. Compile kernels function (creates a kernel from one program function)
8. Create command queue to the target device
9. Create memory buffer(s) that both host and target can access
10. Set up kernel memory arguments (to send to kernel)
11. Enqueue kernel, added to list of kernels to deploy on target
12. Get data from output buffer
13. Do any checks / further processing on output obtained
14. Wait for kernel queue to finish
    - Wait for the OpenCL system to finish performing the commands in the queue and the release the kernel, memory, queues, program and context.

OpenCL Kernel .cl Code Example

```
/** This Kernel function has one buffer that is defined as a
float array. It also has a constant input n. */
__kernel void KernelFunctionName (
    __global float*  x,        // float array to process
    const    unsigned int n ) // number of elements in x
{
 // get the core number and
 const int i = get_global_id(0); // number of this work item
 const int g = get_group_id(0);  // group that work item is in

 // if the x array is long enough, just put this item's group
 // number into the array at index of work item number
 if (i<n) x[i] = g;
}
```

Put this in a file called e.g. **Kernel.cl**

Think of method to check kernel

- Generally, it is useful to have an equivalent implementation, which gives an accurate result, for testing the kernel function (e.g. ‘Golden Measure’ written in Python).

Coding each step

- Prac2.1 will take you through the process of setting up an OpenCL kernel

Thanks to Chris Hill for EEE4120F OpenCL Prac2 updates and refinements.

Info & Procedure for Prac2.1

- Each student should run through the prac.
- The bluelab is planned to be set up for remote access, the machines would need to be shared via ssh remote login.
- The machines each have a small but fast SSD (so keep a separate main copy of your files)

Run the OpenCL version

- A brief view of Prac2.1 to be given in the lecture…

Compare CPU & GPU kernel results

- E.g. comparing results obtained from the (e.g.) Python sequential implementation to those generated by the parallel kernel version.

Further Reading

Supplementary reading

- R Wright, N Haemel and G Sellers
  - OpenGL SuperBible, 6 th ed
  - Addison-Wesley, 2014, ISBN 978-0-321-90294-8
- A Munshi, B R Gaster, T G Mattson, J Fung and D Ginsburg
  - OpenCL Programming Guide
  - Addison-Wesley, 2012, ISBN 978-0-321-74964-2
- Mac Developer Library
  - OpenCL Hello World Example
- Altera
  - OpenCL SDK for FPGA

closing remarks & reminders…

Coding Kernels: OpenCL, C++ → HDL

EEE4120F

Advantages of OpenCL vs C++

Supplementary reading

OpenCL

- Targets CPU, GPU and FPGAs
- Target user is Software developer
- Implements FPGA in software development flow
- Performance is determined by resources allocated
- Host Required

C++ → HDL translators

- Targets FPGA
- Target user is FPGA designer
- Implements FGPA in traditional FPGA development flow
- Performance is defined and amount of resource to achieve is reported
- Host not required

(skip in lecture, may be useful to refer to when writing a OpenCL program)

CUDA vs OpenCL correspondence

Supplementary reading

CUDA		OpenCL
Thread	⟷	Work-item
Thread-block	⟷	Work-group
Global memory	⟷	Global memory
Constant memory	⟷	Constant memory
Shared memory	⟷	Local memory
Local memory	⟷	Private memory
__global__ function	⟷	__kernel function
__device__ function	⟷	no qualification needed
__constant__ variable	⟷	__constant variable
__device__ variable	⟷	__global variable
__shared__ variable	⟷	__local variable

(skip in lecture, may be useful to refer to when writing a OpenCL program)

End of Lecture

Disclaimers and copyright/licensing details

I have tried to follow the correct practices concerning copyright and licensing of material, particularly image sources that have been used in this presentation. I have put much effort into trying to make this material open access so that it can be of benefit to others in their teaching and learning practice. Any mistakes or omissions with regards to these issues I will correct when notified. To the best of my understanding the material in these slides can be shared according to the Creative Commons “Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)” license, and that is why I selected that license to apply to this presentation (it’s not because I particularly want my slides referenced but more to acknowledge the sources and generosity of others who have provided free material such as the images I have used).

Image sources:
Wikipedia (open commons) commons.wikimedia.org
flickr.com
Gadgets, Block diagrams for Altera OpenCL models – fair usage
public domain CC0 (http://pixabay.com/)