ECE3270

Information for these slides taken from Altera OpenCL documentation and learning materials

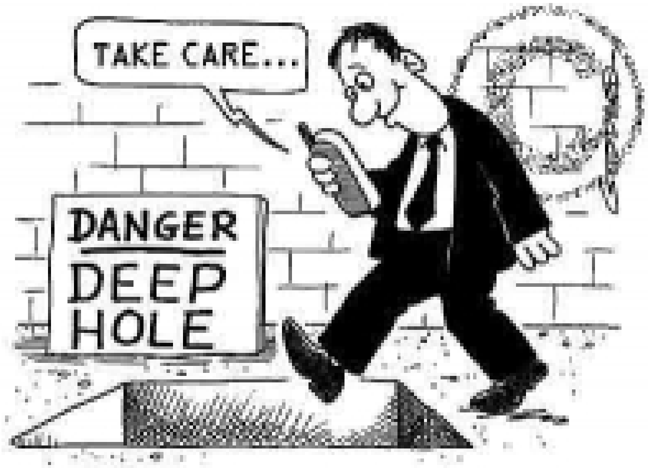
OPENCL

Need for Parallel Computing

- "Moore's Law"
  - Silicon performance expected to double within two years
  - Traditionally achieved by shrinking transistor's dimensions
- Physics limitations "power wall"
  - Starting in mid 2000s, unable to scale operating voltage lower and increase frequency while maintaining reasonable power densities
  - Frequencies are capped
- Performance now come from parallelism
  - Multiple processor cores and other compute resources

Challenges in Parallel Programming

- Finding Parallelism
  - What activities can be executed concurrently?
    - Is parallelism explicit (programmer specified) or implicit?
- Data sharing and synchronization
  - What happens if two activities access the same data at the same time?
    - Hardware design implications
      - eg. Uniform address spaces, cache coherency
- Applications exhibit different behaviors
  - Control
    - Searching, parsing, etc…
  - Data intensive
    - Image processing, data mining, etc…
  - Compute intensive
    - Iterative methods, financial modeling, etc…



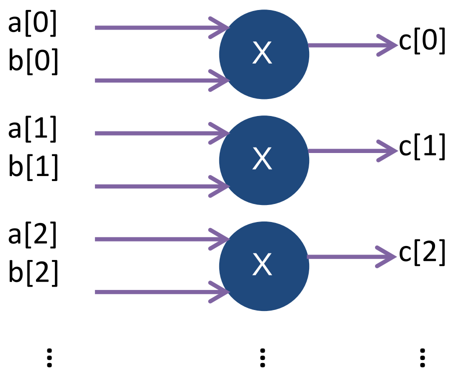
Finding Parallelism

- Scatter-gather
  - Separate input data into subsets that are sent to each parallel resource and then combine the results
  - Data parallelism
- Divide-and-conquer
  - Decompose problem into sub-problems that run well on available compute resources
  - Task parallelism
- Pipeline Parallelism
  - Task parallelism where tasks have a producer consumer relationship
  - Different tasks operate in parallel on different data

Data Parallelism (Scatter-Gather)

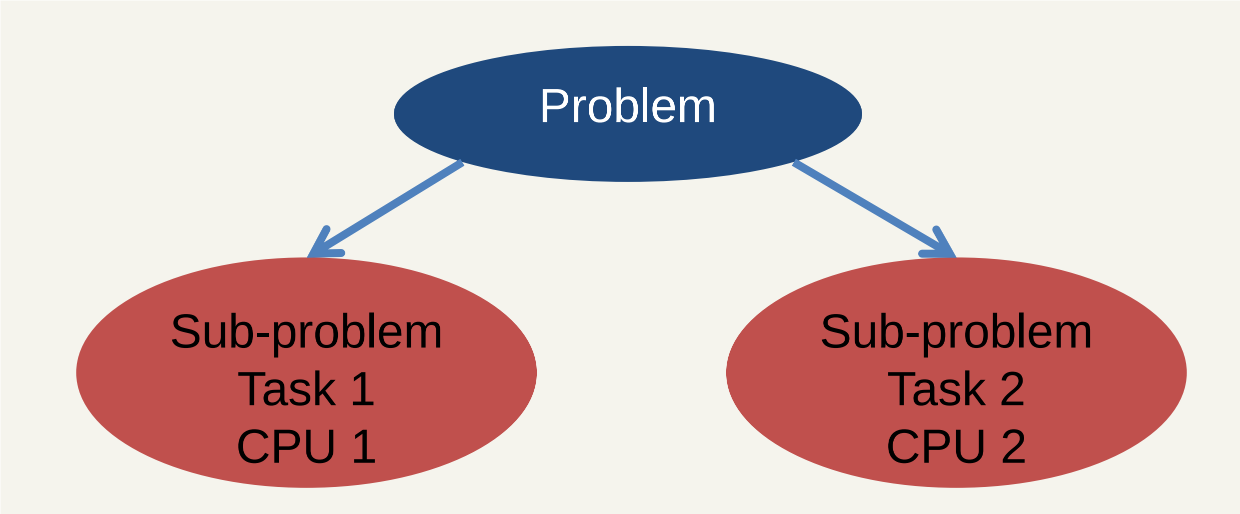
- Same operation can be independently applied across different data in parallel
  - Single Program Multiple Data (SPMD)
  - Single Instruction Multiple Data (SIMD)

```
for (i = 0; i < N ; i++)
        c[i] = a[i] * b[i]
```



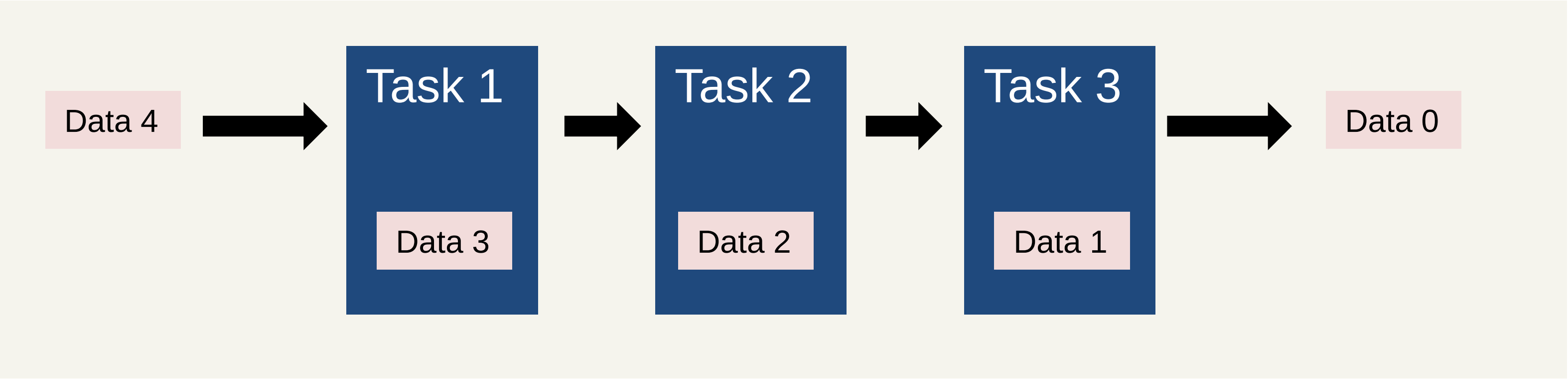
Task Parallelism (Divide and Conquer)

- A.k.a. Thread/Function Parallelism
- Tasks operate on same or different data
- Example
  - Multi-CPU system where each CPU execute a different thread
- Simultaneous Multithreading (SMT)



Pipeline Parallelism

- Task parallelism where tasks have a producer consumer relationship
- Operates on pipelined data
  - Different tasks operate in parallel on different data
- Example
  - Task1 – FFT, Task 2 – Frequency Filter, Task3-Inverse FFT

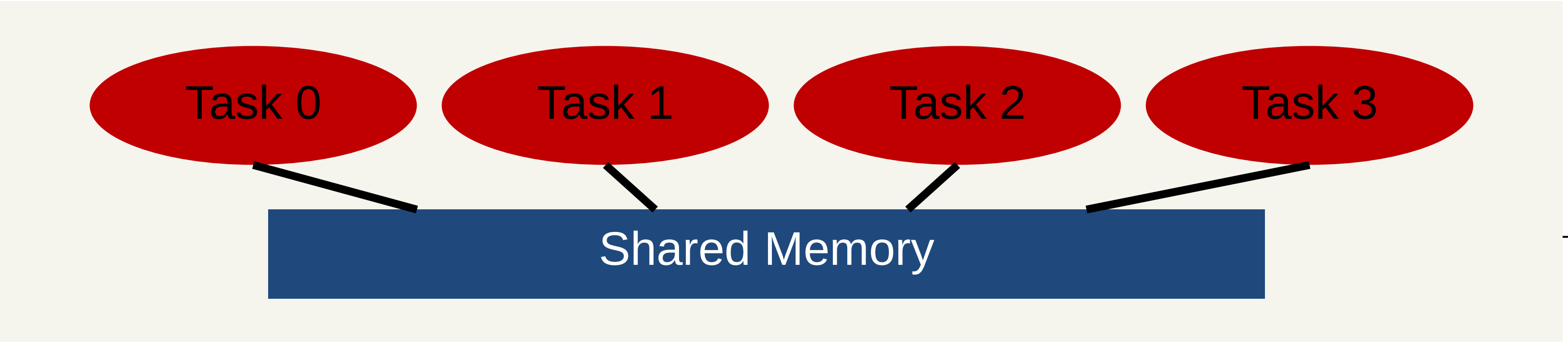


Data Sharing and Synchronization

- Tasks that do not share data can run in parallel without synchronization
- Data dependencies require synchronization
  - Input of one task dependent on result of another
  - Intermediate results are combined together
- Synchronization mechanisms
  - Barriers
    - Stop tasks at certain point until all tasks reach the barrier
  - Locks
    - Enforce limits on access of particular resources

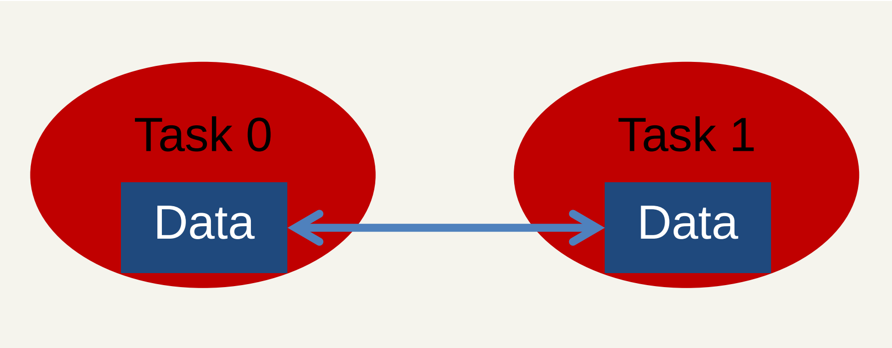
Shared Memory Model

- Global view of memory accessible by tasks
  - Used for inter-task communication
  - May be guarded by semaphores or mutexes (barriers and locks)
- Advantages
  - Programmer not required to manage data movement
  - Code development simplified
- Drawbacks
  - Shared buses and coherency overheads become limiting factors



Message Passing Model

- Explicit communication between concurrent tasks
- Advantages
  - Scalable
    - Tasks can run on arbitrary number of devices
- Drawbacks
  - Programmer needs to explicitly manage communications
  - Uses a specific library of routines for sending and receiving
  - Difficult to make portable

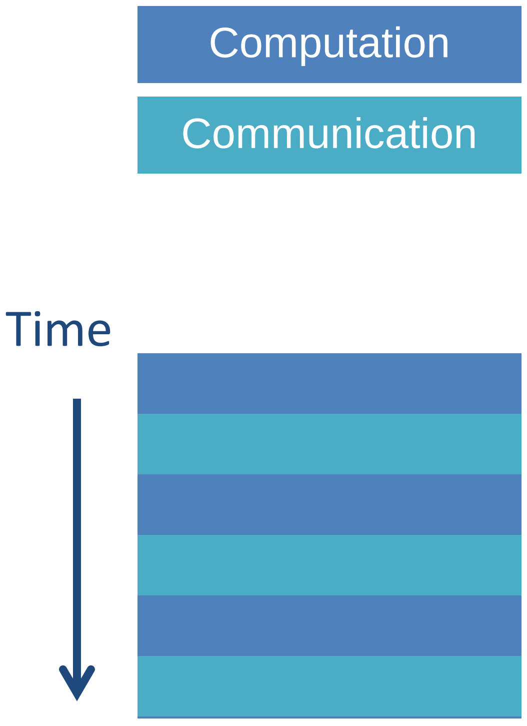


Granularity of Parallelism

- Ratio of computation to communication
- Fine-grain vs. Coarse-grain
- Most efficient granularity heavily depends on application and hardware environment

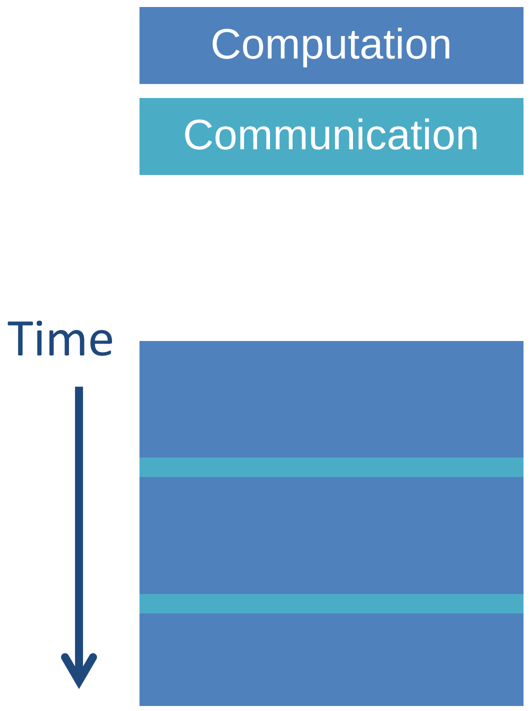
Fine-grained parallelism

- Low computation intensity
- Small task size
- Data transferred frequently
- Benefits
  - Easy to load balance among task
- Drawback
  - Synchronization and communication overhead can overshadow benefits of parallelism
- Example
  - GPU threads
  - Instruction level parallelism



Coarse-grained parallelism

- High arithmetic intensity
- Data transferred infrequently
- Benefit
  - Low overhead
- Drawback
  - Difficult to load balance
- Example
  - Threads running on different CPUs



Parallelism in OpenCL

- Parallelism is explicit
  - Identify parallel regions of algorithm and implement as kernels executed by many work-items
  - Task (SMT) or Data (SPMD)
- Hierarchy of work-items (threads)
  - Work-items are grouped into workgroups
    - Size of workgroups is usually restricted by hardware implementation (256-1024)
  - Work-items within a workgroup can explicitly synchronize and share data
    - Otherwise free to executed independently
  - Work-groups are always independent
- Explicit memory hierarchy
  - Global memory visible to all workgroups and work-items
  - Local memory visible only to work-items in a workgroup
  - Private memory visible only to a single work-item

Discussion

- Pair up and talk about parallelism!

- How do OpenCL and C differ?
  - Memory
  - Parallelism

- What are some examples of Fine and Coarse Grain Parallelism?

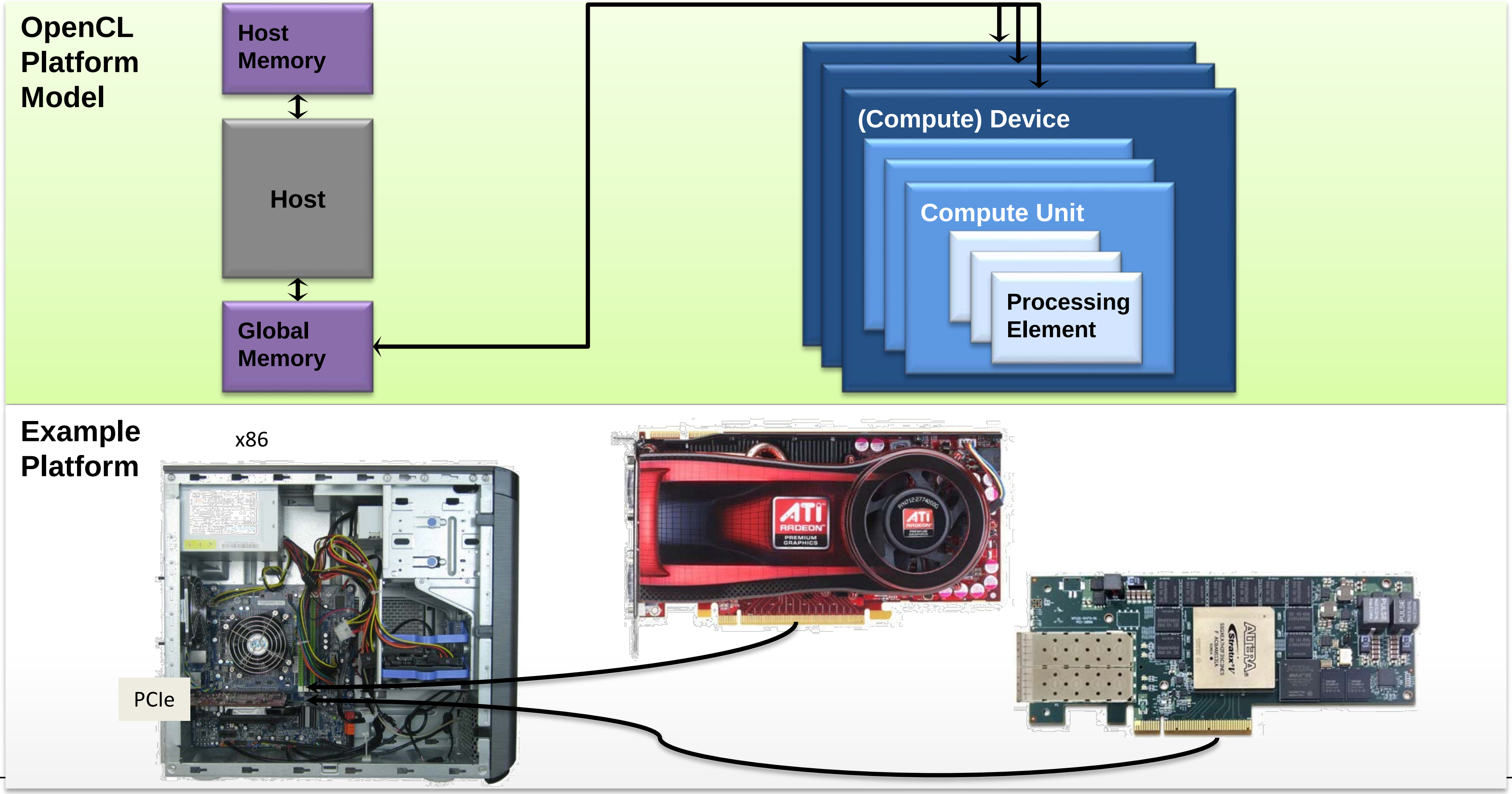
OpenCL Specification Defined by Four Models

- Platform model
  - Defines abstract hardware model
- Execution model
  - Defines the execution environment
  - Concurrency model
  - Host-device interaction
- Memory model
  - Defines abstract memory hierarchy
- Programming model
  - Defines how concurrency model is mapped to hardware

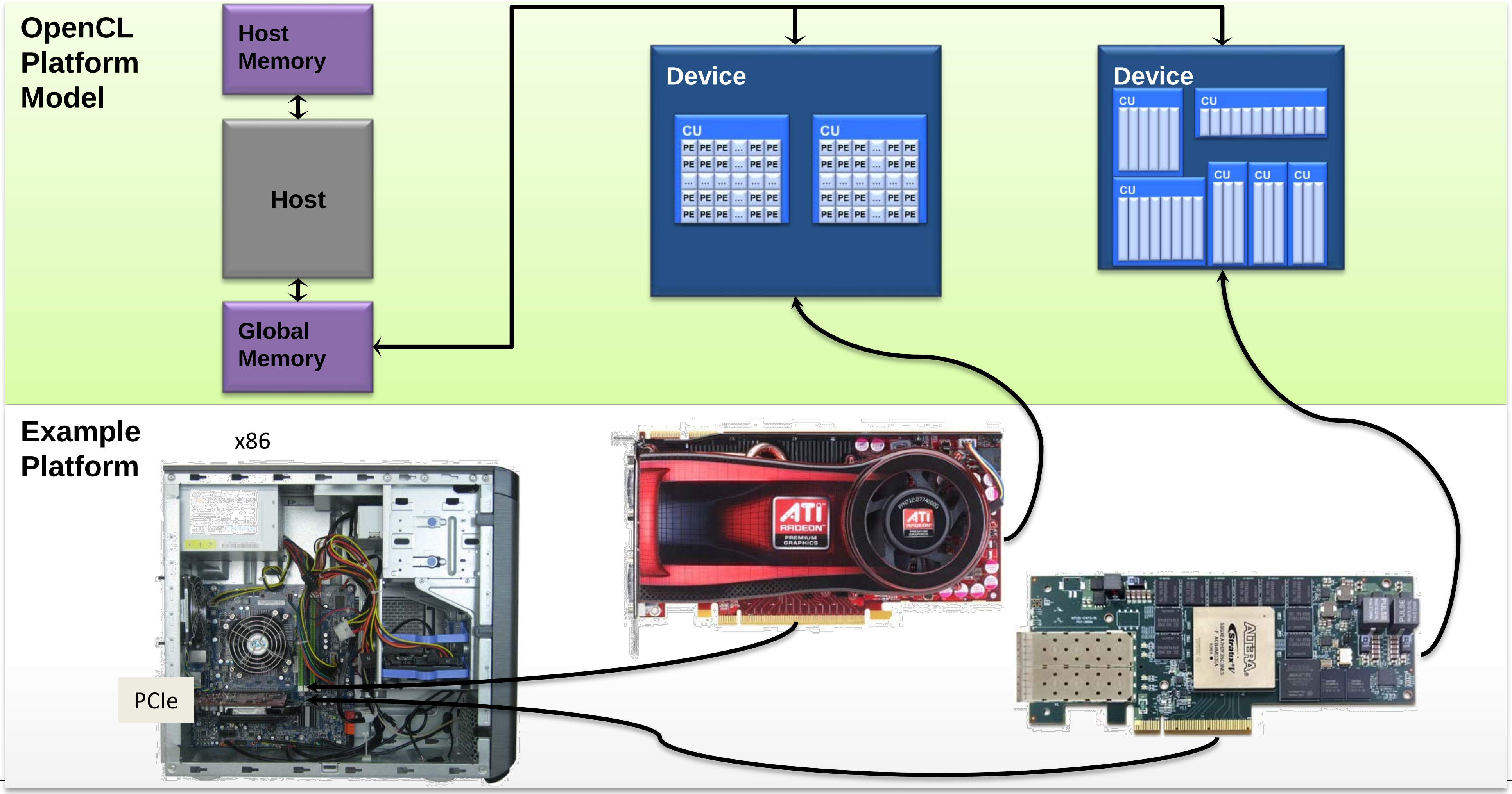
The OpenCL Standard Agenda

- OpenCL overview
- **Platform and execution models**
- Setup API
- OpenCL mechanisms

Heterogeneous Platform Model

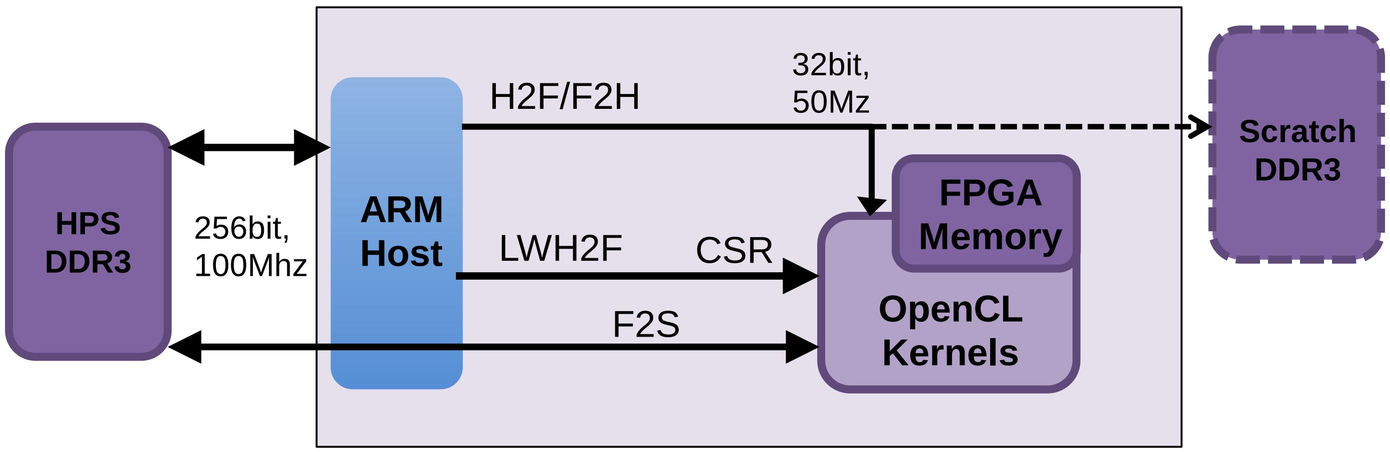


Heterogeneous Platform Model



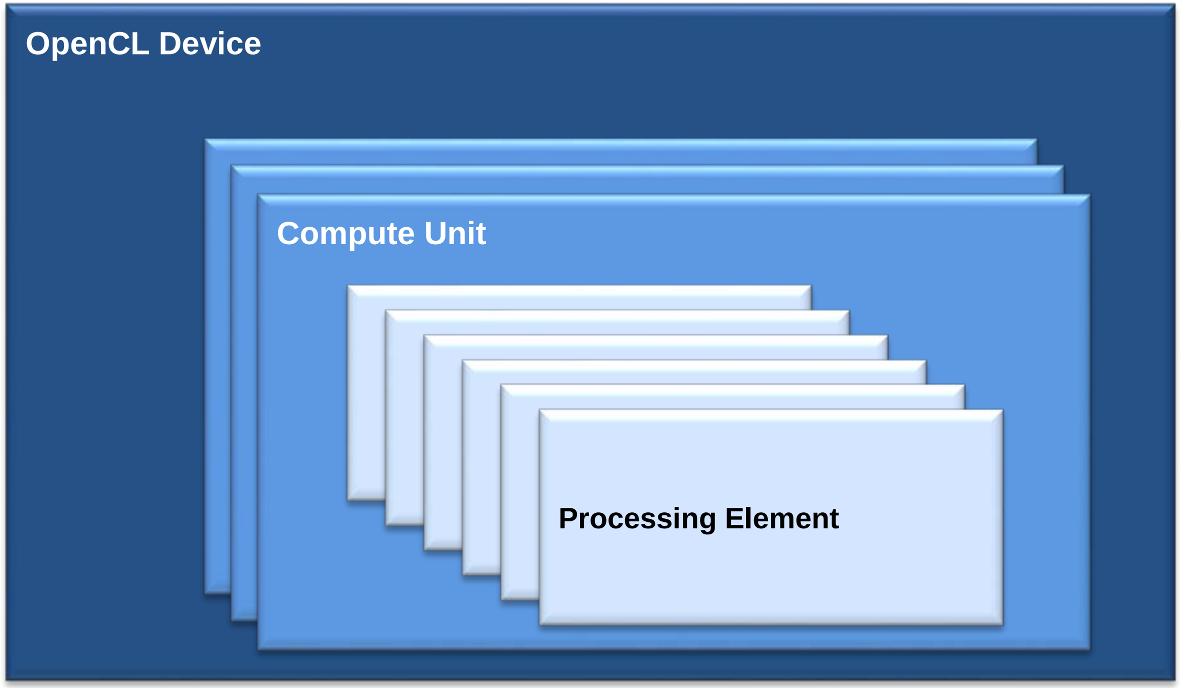
Heterogeneous Platform Model - SoC Platforms

- Running embedded host on the ARM Cortex®-A9 Processors of SoC Devices
  - Cyclone® V, Arria® V, or Arria 10 SoC devices



OpenCL Device

- Device is an array of functionally independent **compute units**
- Compute units divided into **processing elements**



OpenCL Execution Model

- The host defines a context to control the device

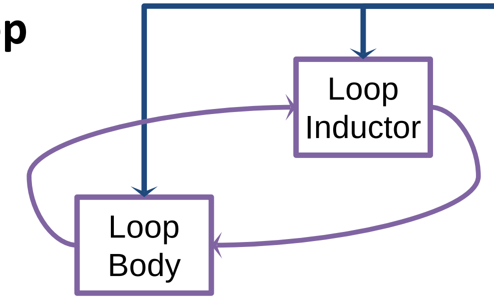
- The context manages the following resources:
  - Devices – hardware to run on
  - Kernels – functions to run on the hardware
  - Program Objects – device executables
  - Memory Objects – memory visible to host and device
  - Command Queues - schedule commands for execution on the device

Execution Model (Data Parallelism)

Implicit Parallelism

```
for (i=0;i<M;i++) {
  u[i] = foo(x[i]);
}
```

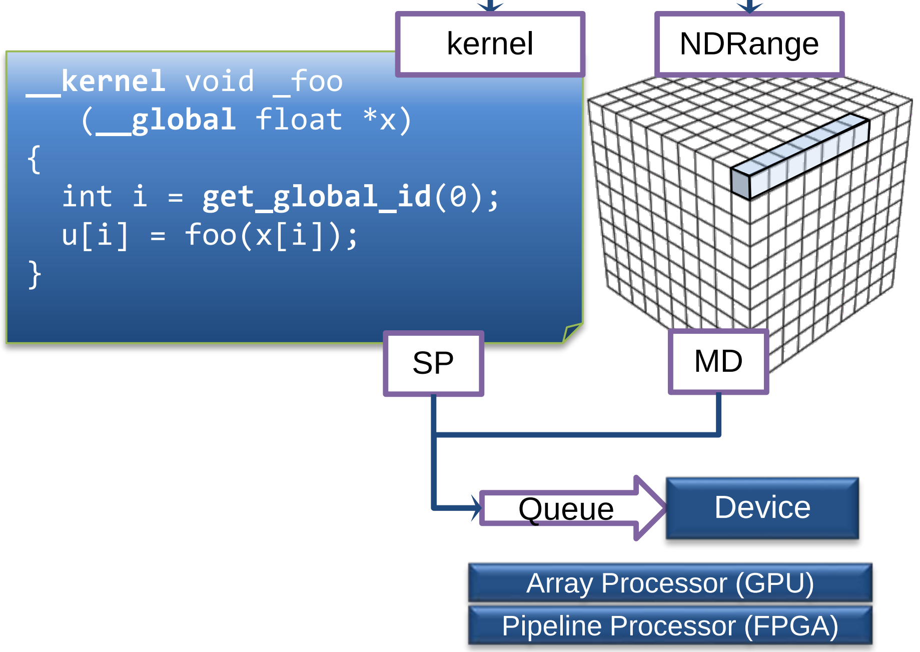
- **Loop**



Data Parallelism (SPMD)

```
clEnqueueWriteBuffer(clQ,x,…)
clEnqueueNDrangeKernel(clQ,_foo,…)
clEnqueueReadBuffer(clQ,u,…)
```

- Kernel



Execution Model (Task Parallelism)

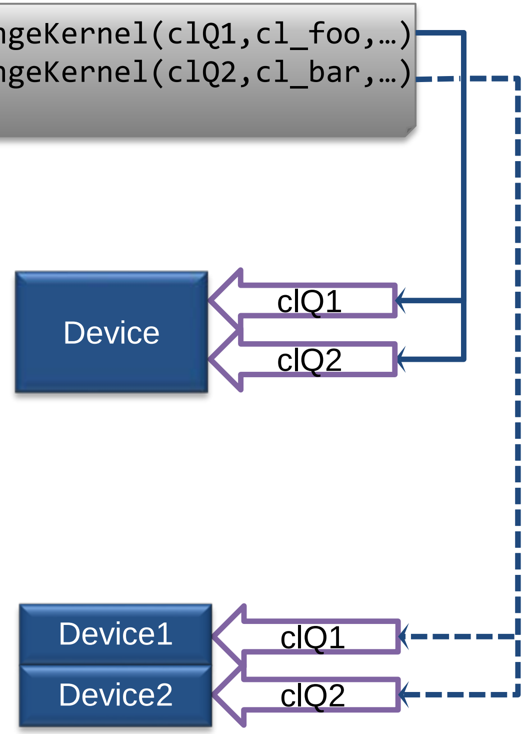
Implicit Parallelism

```
u = foo(x);
y = bar(x);
```

Task Parallelism (SMT)

```
clEnqueueNDrangeKernel(clQ1,cl_foo,…)
clEnqueueNDrangeKernel(clQ2,cl_bar,…)
```

- Queues
- Devices



OpenCL Properties

- Parallelism is declared by the programmer
  - Data parallelism is expressed through the notion of parallel threads which are instances of computational kernels
  - Task parallelism is accomplished with the use of queues and events that coordinate the coarse-grained control flow
- Data storage and movement is explicit
  - Hierarchical abstract memory model
    - Various memory spaces
  - Up to the programmer to manage memories and bandwidth efficiently

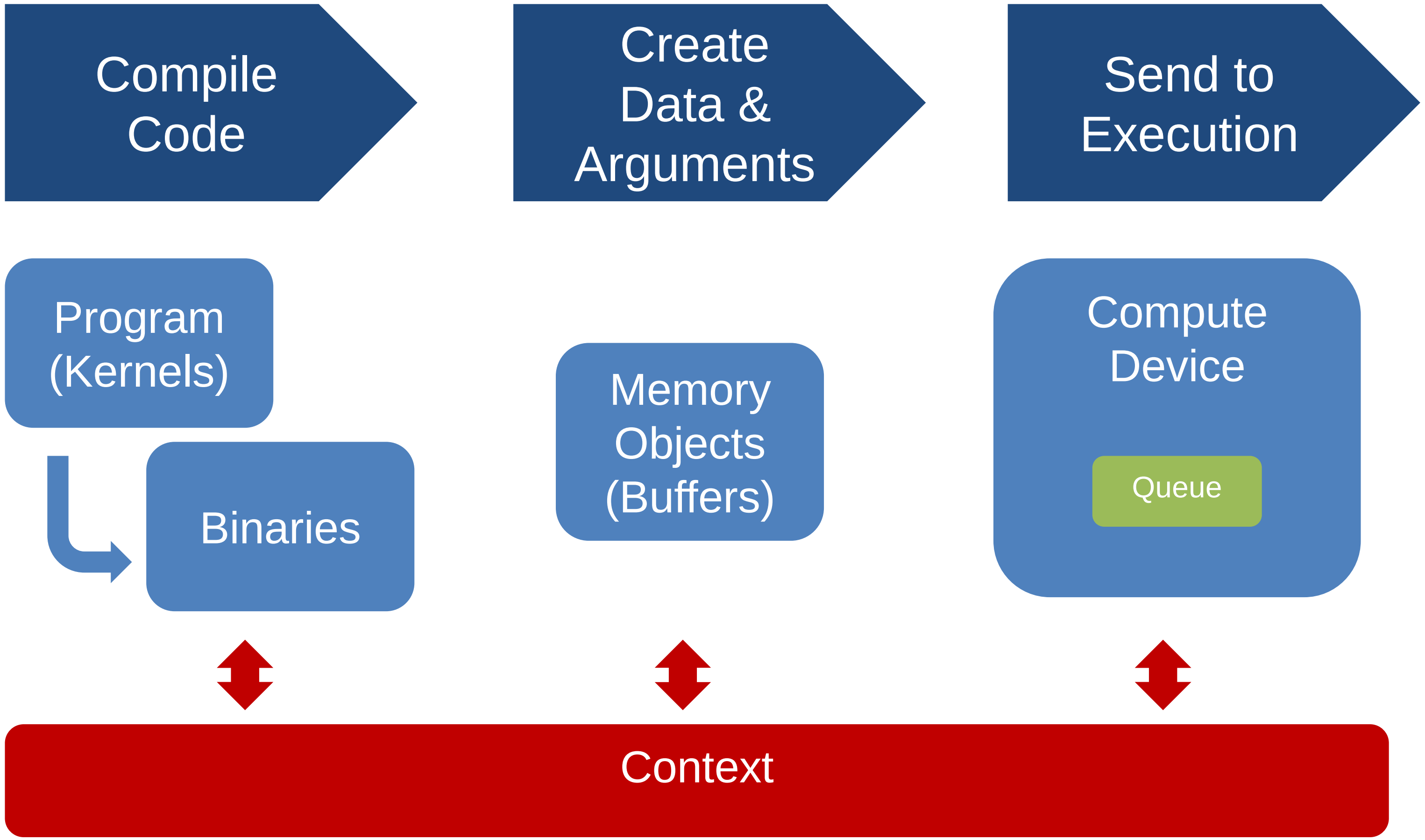
Two Sides of OpenCL Standard

- Device-side language
  - "Kernel Code" or OpenCL C
  - Maps to a wide range of accelerators
  - Usually used for computationally intensive tasks
- Host language
  - Supports efficient plumbing of complicated concurrent programs with low overhead
  - Runs on conventional microprocessor
    - Soft processor, embedded processor, external x86 processor
- Used together to efficiently implement algorithms

Host Managed OpenCL Objects

- Setup
  - Devices – CPU, FPGA, GPU…
  - Contexts – Collection of devices
  - Queue – Work for the device
- Memory
  - Buffers – Blocks of memory
- Execution
  - Programs – Collections of kernels
  - Kernels – Argument/execution instances
- Synchronization/profiling
  - Events

OpenCL Flow


Compile Code
Create Data & Arguments
Send to Execution
Program (Kernels)
Binaries
Memory Objects (Buffers)
Compute Device
Queue
Context

OpenCL APIs

- Access OpenCL features through C API
  - Provided by solutions vendor
  - Single include file (**`opencl.h`**)
- C++ API also available
  - Wrapper that maps to the C API (**`cl.hpp`**)

OpenCL API

- Divided into Platform Layer API and Runtime API
- Platform Layer API
  - Allows host to discover devices and capabilities
  - Query, select and initialize compute devices
  - Create compute contexts
- Runtime Layer API
  - Executes compute kernels
  - Allows the host to work with created contexts

Program Set Up

CLEMSON UNIVERSITY

- To set up an OpenCL program, the typical steps are as follows:

Platform Layer

1. Query and select the platforms (e.g., Altera)
2. Query the devices
3. Create a context

Runtime Layer

4. Create a command queue
5. Read/Write to the device
6. Launch the kernel

The OpenCL Standard Agenda

- OpenCL overview
- Platform and execution models
- **Setup API**
- OpenCL mechanisms

Query and Select the Platform

- Steps
  1. Call `clGetPlatformIDs()` to get available number of platforms
  2. Allocate space to hold platform information
  3. Call `clGetPlatformIDs()` again to fill in platforms
  4. Call `clGetDeviceIds()` to get available number of device in a platform
  5. Allocate space to hold device information
  6. Call `clGetDeviceIDs()` again to fill in devices

Platform Profile

- Platforms support one of the two following profiles:
  - FULL_PROFILE – supports the OpenCL specification
    - Functionality defined as part of the core specification
    - Does not require any extensions to be supported
  - EMBEDDED_PROFILE – supports a subset of the OpenCL specification
- Altera’s OpenCL platform supports the EMBEDDED_PROFILE

OpenCL Platform Layer

- Get platform identifiers with `clGetPlatformIDs`
  - Arguments:
    - *`cl_uint num_entries`* – the length of the *platforms* list
    - *`cl_platform_id *platforms`* – the list of OpenCL platforms
    - *`cl_uint *num_platforms`* – the total number of platforms
  - The function is typically called twice
    - First to determine the number of platforms
      - `clGetPlatformIDs(0, NULL, &num_platforms)`
    - Second to get the platform IDs
      - `clGetPlatformIDs(num_platforms, &platforms, NULL)`

Note: Pointer arguments such as *`cl_uint *num_platforms`* are assigned *`NULL`* when unused and all other arguments use 0.

- Hardware vendors provide platforms to manage their devices:
  - A system may have several platforms (e.g., Altera and Intel)

OpenCL Platform Layer – Devices

- Enumerate the devices present in the system with `clGetDeviceIDs`
  - Arguments:
    - *`cl_platform_id platform`* – the target platform
    - *`cl_device_type device_type`* – the field that identifies the type of the device (e.g., CL_DEVICE_TYPE_CPU, CL_DEVICE_TYPE_ALL)
      - Use CL_DEVICE_TYPE_ACCELERATOR for Altera FPGA
    - *`cl_uint num_entries`* – the length of the *devices* list
    - *`cl_device_id *devices`* – the list of OpenCL devices
    - *`cl_uint num_devices`* – the total number of devices

Query Platform and Device Info

- Use `clGetPlatformInfo()` and `clGetDeviceInfo()`
- Print hardware details
  - Platform
    - Vendor
    - Version
    - Etc…
  - Devices
    - Memory sizes
    - Bus widths
    - Device Type
    - Endianess
    - Etc…

Contexts

- Abstract container
  - Exists on the host
  - Coordinates the mechanisms for host-device interaction
  - Manages the memory objects
  - Keeps track of programs created for each device

Context Creation

- Create context and associate it with the devices
  - Use `clCreateContext()`
  - Arguments
    - *`const cl_context_properties *properties`*
      - Restricts the scope of context
    - *`cl_uint num_devices`* – Number of devices
    - *`const cl_device_id *devices`* – pointer to devices
    - *`void *pfn_notify()`* – optional callback function to report error info
    - *`void *user_data`* – supplied data for callback
    - *`cl_int *errcode_ret`* – error code
- Use **`clGetContextInfo()`** function to query information after creating a context
  - E.g. number of devices and device structures

Context Creation from Device Type

- Use `clCreateContextFromType()`
  - Same arguments as clCreateContext()but pass in device type enum instead of number of and list of devices
  - Possible types:
    - `CL_DEVICE_TYPE_CPU`
    - `CL_DEVICE_TYPE_GPU`
    - `CL_DEVICE_TYPE_ACCELERATOR (FPGAs)`
    - `CL_DEVICE_TYPE_DEFAULT`
    - `CL_DEVICE_TYPE_ALL`

Sample Platform Layer C Code

```
//Get the platform ID
cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);

// Get the first FPGA device associated with the platform
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ACCELERATOR, 1,
&device, NULL);

//Create an OpenCL context for the FPGA device
cl_context context;
context = clCreateContext(NULL, 1, &device, NULL, NULL,
NULL);
```

Setup Code

- Discovering platforms and devices and setting up a context seems tedious

- Written once and reused for almost any project

The OpenCL Standard Agenda

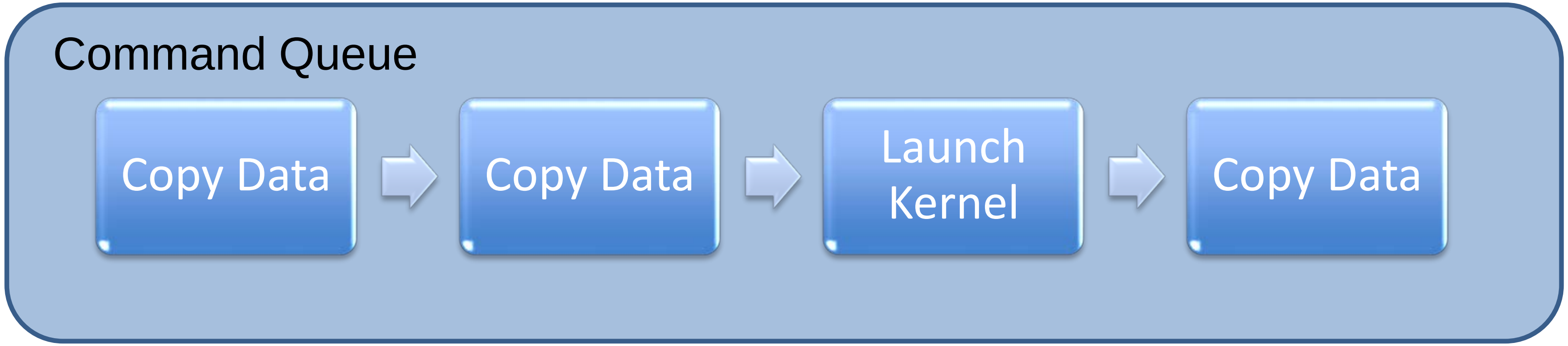
- OpenCL overview
- Platform and execution models
- Setup API
- **OpenCL mechanisms**
  - **Command queues**
  - **Events**
  - **Memory objects**

Command Queues

- Mechanism for host to request action by the device
- Each command queue associated with one device
- Host submits commands to the appropriate queue
  - `clEnqueue` commands

OpenCL Command Queue

- A command queue operates on contexts, memory, and program object
- Each device can have one or more command queues
- Operations in the command will execute in-order unless the out-of-order mode (currently not supported by Altera) is enabled



Queue Creation API

- Creates a command queue and associate it with a device

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

- Arguments
  - *cl_context context* – valid context
  - *cl_device_id device*– device associated with context
  - *cl_command_queue_properties properties* – bit-field enabling profiling and/or out-of-order execution

Events

- Operations that add command to the command queue (`clEnqueue`) produce an **event**
  - Represent dependencies
    - When executing `clEnqueue`, can pass in a blocking "wait list" of events as parameter
  - Provide mechanism for profiling
    - Using associated timers

Memory Objects

- OpenCL applications tend to work with large arrays or multidimensional matrices
- Data needs to be physically located on a device before execution
- Data encapsulated as **memory objects** in order to be transferred
- Valid within only one context
- OpenCL specification defines two types
  - Buffers and images

Buffer Object

- A buffer stores a one dimensional collection of elements
  - Any type of data that doesn't involve images
  - Elements can be scalar (int, float), vector data type, or user-defined structure

- Buffer objects use the `cl_mem` type
  - `cl_mem` is an abstract memory container (i.e., a handle)
  - The buffer object cannot be dereferenced on the host
    - `cl_mem a; a[0] = 5; // Not allowed`

- Specific OpenCL commands required to interact with buffers

Allocating Buffer

- Allocate buffer and return memory object
  - Similar to **`malloc`** and **`new`**

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

  - Arguments
    - *`cl_context context`* - where buffer will be allocated
    - *`cl_mem_flags flags`* - optionally supply flags
      - Upcoming slide
    - *`size_t size`* in bytes
    - *`void *host_ptr`* - pointer to buffer data that may already be allocated

Image Object

- An image[1] object stores an image or array of images

- Simplifies the process of representing and accessing images
  - Built-in support for processing image data
  - Native support for a multitude of image formats

- Image objects also use the `cl_mem` type

1. Partial OpenCL 1.2 image support available in Altera SDK version 15.0

Allocating Images

- Allocate image and return memory object

```
cl_mem clCreateImage (cl_context context,
    cl_mem_flags flags,
    const cl_image_format *image_format,
    const cl_image_desc *image_desc,
    void *host_ptr,
    cl_int *errcode_ret)
```

– Arguments
  - `cl_context context` - same as buffer
  - `cl_mem_flags flags` - same as buffer
  - `const cl_image_format *image_format` - describes format properties of the image to be allocated
    – See the OpenCL 1.2 specification for options
  - `const cl_image_desc *image_desc` - describes the type and dimensions of the image
    – See the OpenCL 1.2 specification for options
  - `void *host_ptr` - same as buffer

Memory Management Flags

- Flags:
  - CL_MEM_READ_WRITE – default
  - CL_MEM_WRITE_ONLY
  - CL_MEM_READ_ONLY
  - CL_MEM_USE_HOST_PTR – the host ptr value contains the storage for the data; the device may cache the memory
  - CL_MEM_COPY_HOST_PTR – memory is allocated and the values stored in host memory are copied into the device
- Read/write permissions constrain the device access, <u>not</u> the host access
- Flags may be combined

```
buffer = cl_mem clCreateBuffer (context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    sizeof(float)*32, host_vector, &error)
```

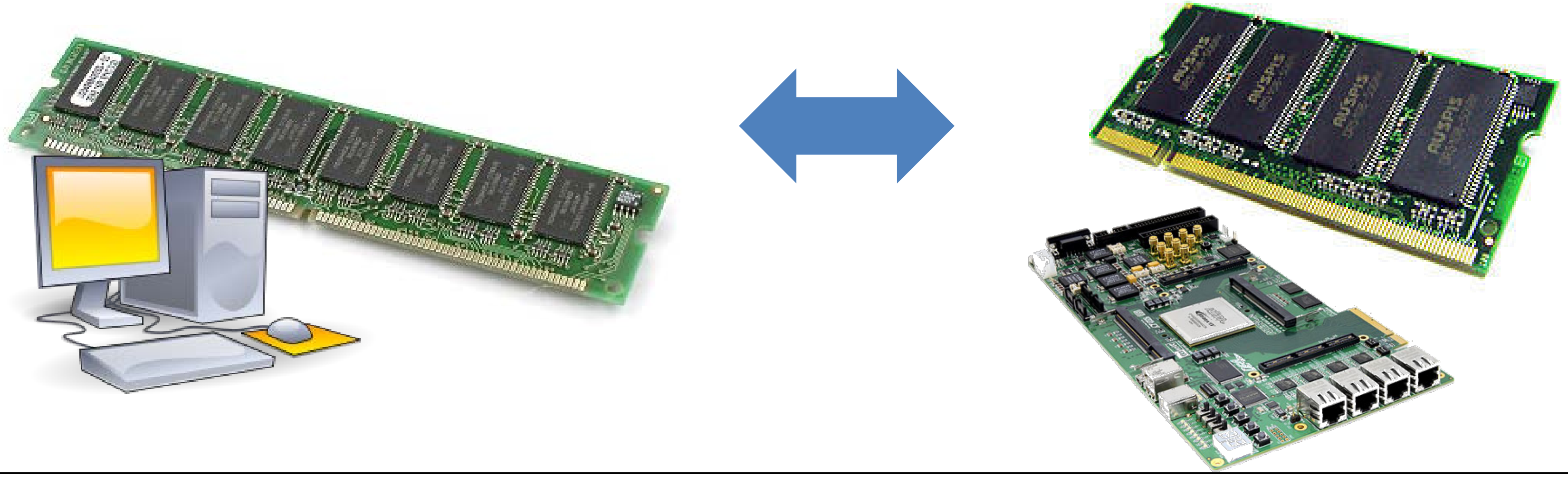
Memory Management Error Codes

- Host functions return error code or has pointer argument for error code
  - Defined in `cl.h`
  - Error codes are negative, `CL_SUCCESS == 0`

```
/* Error Codes */
#define CL_SUCCESS                                  0
#define CL_DEVICE_NOT_FOUND                         -1
#define CL_DEVICE_NOT_AVAILABLE                     -2
#define CL_COMPILER_NOT_AVAILABLE                   -3
#define CL_MEM_OBJECT_ALLOCATION_FAILURE            -4
#define CL_OUT_OF_RESOURCES                         -5
#define CL_OUT_OF_HOST_MEMORY                       -6
#define CL_PROFILING_INFO_NOT_AVAILABLE             -7
#define CL_MEM_COPY_OVERLAP                         -8
#define CL_IMAGE_FORMAT_MISMATCH                    -9
#define CL_IMAGE_FORMAT_NOT_SUPPORTED               -10
#define CL_BUILD_PROGRAM_FAILURE                    -11
#define CL_MAP_FAILURE                              -12
#define CL_MISALIGNED_SUB_BUFFER_OFFSET             -13
#define CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST -14
```

Physical Memory Space

- The host and the device each has its own physical memory space
- Use OpenCL API functions to allocate, transfer, and free device memory
- Data transferred between host and device



Data Transfers

- Data transferred between host memory and OpenCL buffer explicitly
  - API on next slide
  - Commands placed on the command queue
- If kernel dependent on the buffer is executed on accelerator device, buffer is transferred to the device
- Runtime determines precise time data is moved

Data Transfers

- Host code manages data transfers to and from the device with the `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`:
  - The arguments:
    - *`cl_command_queue command_queue`* – a valid queue of instructions
    - *`cl_mem buffer`* – a valid buffer handle
    - *`cl_bool blocking`* – if CL_TRUE, the function will block
    - *`size_t offset`* – offset in bytes into the *buffer* array
    - *`size_t cb`* – size of data to transfer in bytes
    - *`void *ptr`* – pointer to host memory
    - *`cl_uint num_events_in_wait_list`* – length of the event list
    - *`const cl_event *event_wait_list – the event list`*
    - *`cl_event *event –`* an event pointer for the transfer

    Events specify data dependencies between commands

Memory Management – Code Example

```
//Create an OpenCL command queue
cl_int err;
cl_command_queue queue;
queue = clCreateCommandQueue(context, device, 0, &err);

// Allocate memory on device
const int N = 5;
int nBytes = N*sizeof(int);
cl_mem a = clCreateBuffer(context, CL_MEM_READ_WRITE,
                          nBytes, NULL, &err);

int hostarr [N] = {3,1,4,1,5};

// Transfer Memory
err = clEnqueueWriteBuffer(queue, a, CL_TRUE, 0,
                           nBytes, hostarr, 0, NULL,
                           NULL);
```

OpenCL Software - Summary

- Open CL
  - Provides an API to coordinate parallel computation across heterogeneous processors
  - Defines a cross-platform programming language
- Platform- Host and a collection of devices
- Device - Hardware that the kernel is run on
- Context - Manages devices, kernels, program and memory objects
- Command Queue - Schedules commands for execution on the device
- `cl_mem` is a handle for buffers on device
- Host manages memory allocation, memory transfer and error checking

Discussion

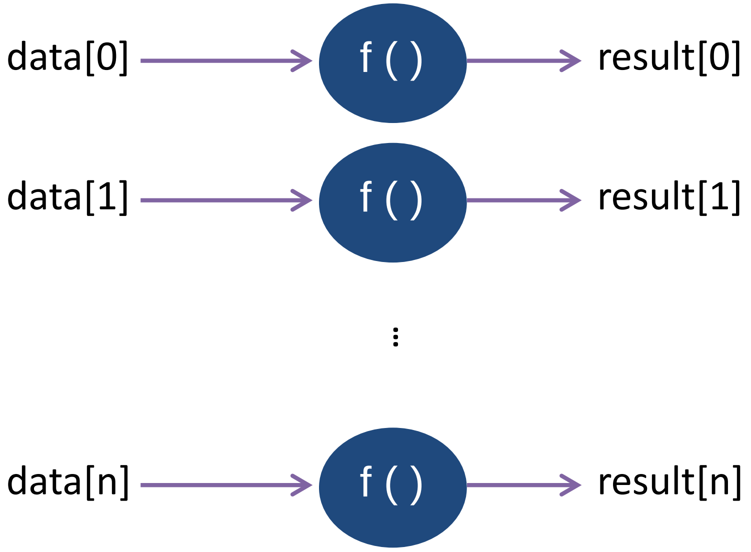
- Discuss the OpenCL API so far
- What is the difference between the runtime and platform layers?
- How is memory different here than in traditional programs?
- With what you know, how are FPGAs different from other accelerators?

Writing OpenCL Programs Agenda

- **Kernels and work-item hierarchy**
- Launching kernels
- Kernel code
- Memory model
- Clean up

Data Parallelism

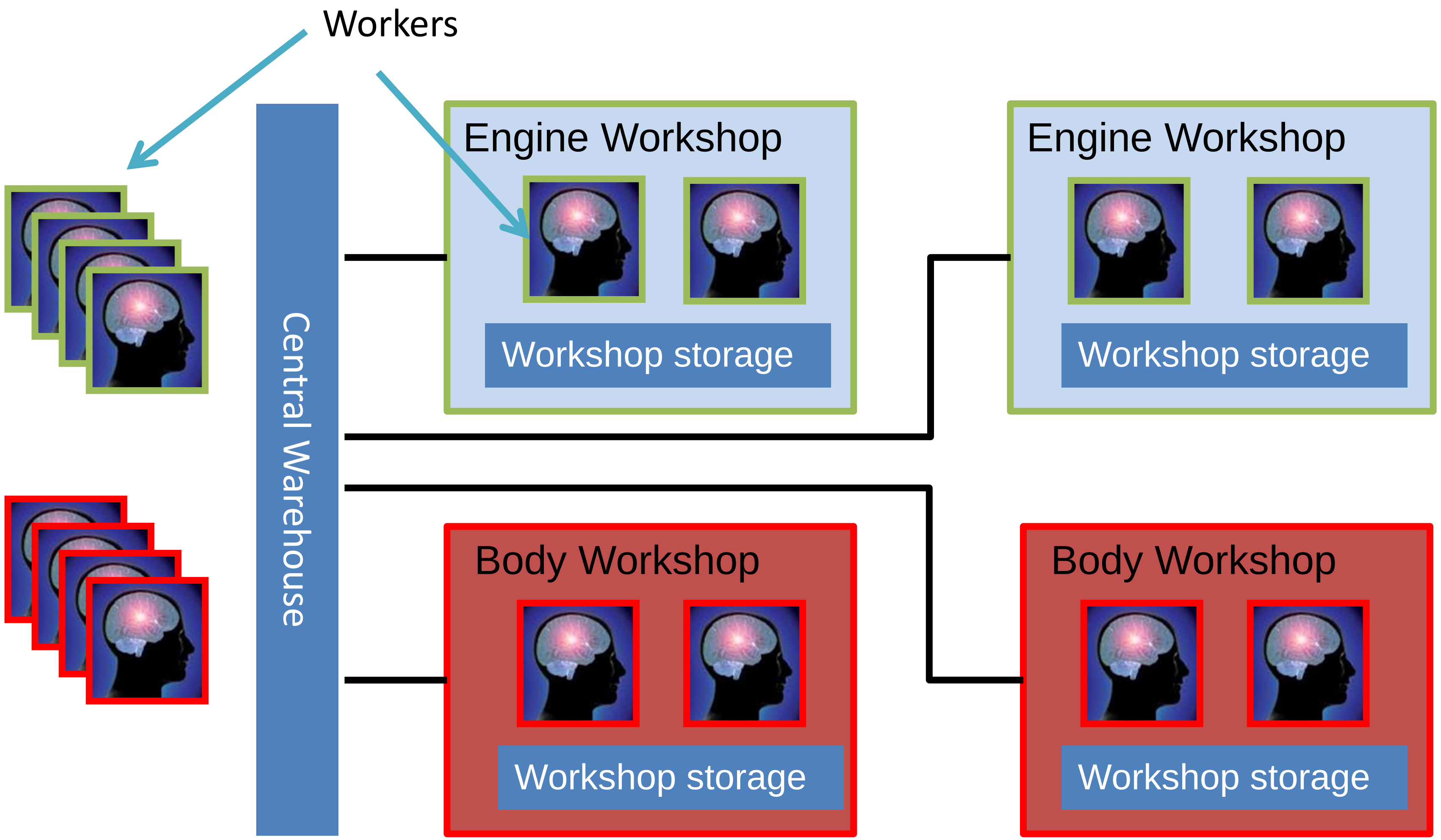
- Data Parallelism (Review)
  - Same operation applied to multiple, independent data concurrently
  - Data dependency hinders data parallelism



Work-Item Hierarchy Analogy

- Factory building cars from parts
- Two step process, assemble engine and assemble body
- Each worker assembles one engine or one body
  - Organized into groups though each works independently
  - Can leave workshop once everyone in the group is done
- Several identical but separate workshops for assembling engines and also for body
- Central warehouse stores parts and finished cars

Car Factory Analogy



Car Factory Analogy Explained

- NDRange: Total number of cars to build
- Work-item: Worker
- Workgroup: Group of workers
  - Workers in different groups can't talk to each other
- Kernel: what to do on the assembly line
  - Two kernels, one for engine and one for body
- Device: Entire factory
- Compute Unit : Workshops and machinery inside

Kernels

What to do on the assembly line

- OpenCL C code written to run on OpenCL devices
- Kernels provide data parallelism
- Syntactically similar to standard C function
  - Set of additional keywords
  - Some restrictions
- General interface and low-level language maps efficiently to wide range of hardware
- Represent parallelism at the finest granularity possible

Work-item

Factory worker

CLEMSON UNIVERSITY

- Unit of concurrent execution in OpenCL standard
  - Data parallel task
- Each work-item executes the same kernel function body independently
- Writing the kernel
  - Usually map single iteration of loop to a work-item
  - Generate as many work-items as elements in the input and output array
- Mapped to hardware during runtime

Example Kernel

- Kernel often represents a single iteration of loop
- N work-items will be generated to match array size
  - `get_global_id(0)` function returns position of work-item which represent the loop counter

```
// What does this do?
// N work-items to be created
__kernel void vecadd(__global int *C, __global int *A, __global int *B)
{
	int tid = get_global_id(0); // OpenCL function to retrieve index
	C[tid] = A[tid] + B[tid];
}
```

OpenCL Work-Item Hierarchy

- OpenCL is designed to execute millions of work-items
- Work-items are grouped together into **workgroups**
  - Default workgroup size is 256
  - Query `CL_DEVICE_MAX_WORK_GROUP_SIZE` in `clGetDeviceInfo`
- The entire collection of work-items is called the N-Dimensional Range (NDRange)

NDRange

Total number of cars to build

- N-dimensional range
- Global Dimension
- One-, two-, or three-dimensional index space of work-items
- Often maps to dimensions of input or output data
  - If we have 512 work-items, NDRange can be specified as
    - `size_t worksize[3] = {512, 1, 1};`
    - 2nd and 3rd dimensions can be omitted if size is 1 as in the case here
  - Set at kernel launch time

NDRange Data Division Examples

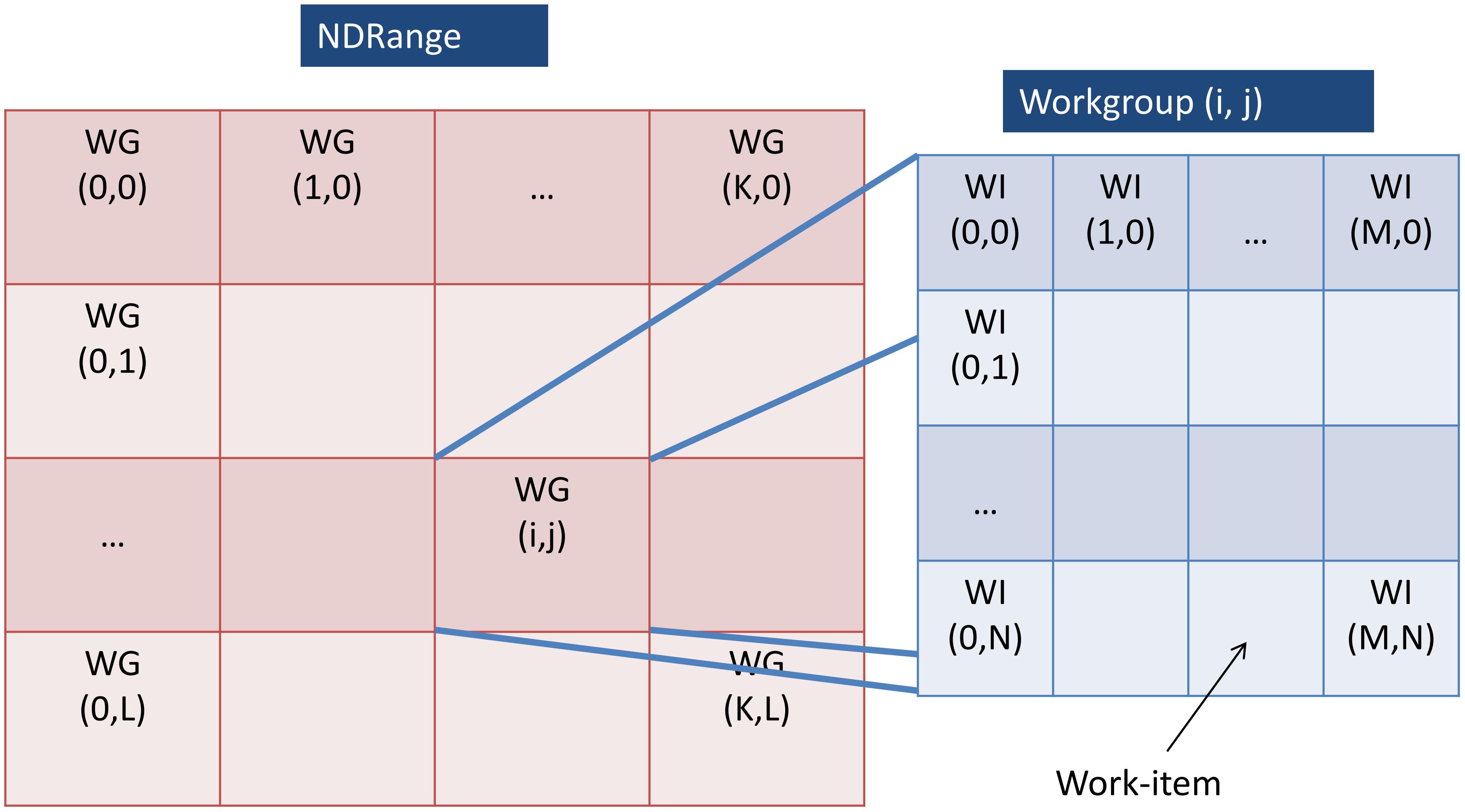
- Audio
  - Series of samples
    - Process each sample independently (e.g. volume change)
  - One dimensional data
  - NDRange = total number of samples
- Images
  - Maps well to two-dimensional data
  - NDRange = total number of pixels
- Physics Simulation
  - Simulate stresses to model behavior of materials
  - Use three-dimensional data
  - NDRange = representation of the 3D space

Workgroup

Group of workers able to communicate

- Dividing work-items of an NDRange into smaller equally sized workgroups
- Local Dimension
- Same dimension as NDRange index space
  - For vector add example if we have 64 work-items per work group
    - `size_t workGroupSize[3] = {64, 1, 1}`
  - NDRange index space size must be evenly divisible by workgroup sizes in each dimension
  - May be automatically generated
  - Set at kernel launch time
- Synchronization between work-items possible only within workgroups
- Optimal workgroup size usually determined by hardware

Work-Item Hierarchy (2D Example)



Writing OpenCL Programs Agenda

- Kernels and work-item hierarchy
- **Launching kernels**
- Kernel code
- Memory model
- Clean up

Programs and Kernels

- Program – collection of kernels
- Process for host to execute a kernel on device

1. Create program
    - Turn source code or precompiled binary into program object
2. Compile program
3. Create kernel by extracting it from program object
    - Similar to obtaining exported function from dynamic library
4. Setup kernel arguments individually
    - Also require memory objects to be transferred to the device
5. Dispatch kernel through `clEnqueue` function

Code Example

- Let's complete the host code

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

```
void main()
{
   cl_context context;   cl_device_id device;
   ...
   // 1. Create then build the program
   // 2. Create kernels from the program
   // 3. Allocate and transfer buffers on/to device
   // 4. Set up the kernel argument list
   // 5. Launch the kernel
   // 6. Transfer result buffer back
 }
```

Creating a Program

- A program object (`cl_program`) contains one or more kernels (`cl_kernel`)
- GPU/CPU vendors support creating of program with source code via `clCreateProgramWithSource` function
  - Online compilation of kernels (runtime compilation)
  - Not supported by Altera
- Altera only supports creating of program from pre-compiled binaries
  - Binary implementation is vendor specific
  - Altera supports aocx files
    - Represent generated library in emulation mode or FPGA programming image

Creating Programs from Binary for FPGAs

- Arguments for `clCreateProgramWithBinary`
  - `cl_context context` – the context
  - `cl_uint num_devices` – number of devices associated
  - `cl_device_id *device_list` – pointer to a list of devices
  - `size_t *lengths` – size of binary in bytes
    - For Altera, size of the aocx file in bytes
      - Use the C `ftell()` function to determine the size.
  - `unsigned char **binaries` – program binaries
    - For Altera, contents of the aocx file
      - Use the C `fread()` function to get the content from aocx file
  - `cl_int *binary_status` – status of binary loading
  - `cl_int *errcode_ret` – the error code
- Returns cl_program object
- For Altera, when this function is called, the host will configure the FPGA with the binaries

Creating Programs from Binary - Code Example

```
//Read aocx file into unsinged char array
FILE *fp = fopen(“program.aocx”, “rb”);    //Open aocx file for binary read
fseek(fp, 0, SEEK_END);
size_t length=ftell(fp);                   //Determine size of aocx file
unsigned char* binaries =
        (unsigned char*)malloc(sizeof(unsigned char) * length);
rewind(fp);
fread(binaries, length, 1, fp);
fclose(fp);

cl_program program = clCreateProgramWithBinary(context,
                                  num_device,
                                  device_list,
                                  length,
                                  (const unsigned char**) binaries,
                                  status,
                                  &error);

clBuildProgram(program, 1, device_list options, NULL, NULL);
```

Building Programs

- Compiles and links a program executable from the program source or binary
- Arguments for `clBuildProgram`
  - `cl_program program` – the program object created earlier
  - `cl_uint num_devices` – number of devices associated
  - `cl_device_id *device_list` – pointer to a list of devices
  - `const char *options` – build options
  - `void *pfn_notify()` – optional callback function to report error info
  - `void * user_data` – data for callback
- Returns `cl_int` error code
- For Altera, needs to be called to conform to the standards, but nothing meaningful done

Building Programs - Code Example

```
void main()
{
   ...
   cl_int clError;
   cl_context context;   cl_device_id device;
   ...

   // 1. Create then build the program
   cl_program program = clCreateProgramWithBinary(context,1, &myDevice,
                                    &BinLength, &binaries, &status,
                                    &clError);

   clError = clBuildProgram(program, 1, &device, compilerOptions, NULL, NULL);


   // 2. Create kernels from the program
   // 3. Allocate and transfer buffers on/to device
   // 4. Set up the kernel argument list
   // 5. Launch the kernel
   // 6. Transfer result buffer back
 }
```

OpenCL API – Creating Kernels

- Create kernels from programs with `cl_kernel clCreateKernel()`. Arguments are:
  - *`cl_program program`* – the program
  - *`const char* kernel_name`* – the kernel name
  - *`cl_int* errcode_ret`*

- For Altera implementation, you should be able to load any of the kernels compiled via the AOC kernel compiler

Creating Kernels – Code Example

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

```
void main()
{

   ...
   // 1. Create then build the program
   cl_program program = clCreateProgramWith…(...);
   clError = clBuildProgram(...);

   // 2. Create kernels from the program
   cl_kernel kernel = clCreateKernel(program, ”increment”, &err);

   // 3. Allocate and transfer buffers on/to device
   // 4. Set up the kernel argument list
   // 5. Launch the kernel
   // 6. Transfer result buffer back

}
```

Setting Up Kernel Argument List

- All values passed into a kernel must be set up using the `clSetKernelArg` command
  - ```
    cl_int clSetKernelArg (
          cl_kernel kernel, cl_uint arg_index,
          size_t arg_size, const void *arg_val)
    ```

- The value of arg_index matters!
  - Passing in the wrong argument index will put the wrong values in the wrong arguments
  - Potentially a difficult problem to debug
    - No error would be reported by the `clSetKernelArg` call
    ```

Setting Up Kernel Argument List - Code Example

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

```
void main()
{
   ...
   cl_program program = clCreateProgramWith…(...);
   clError = clBuildProgram(...);
   cl_kernel kernel = clCreateKernel(program, ”increment”, &err);

   // 3. Allocate and transfer buffers on/to device
   cl_mem aD;
   float bH = 10.8;
   ...

   // 4. Set up the kernel argument list
   // Setup up ‘a’ first
   clError = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&aD);

   // Set up ‘b’ second
   clError = clSetKernelArg(kernel, 1, sizeof(float), (void *)&bH);

   // 5. Launch the kernel
   // 6. Transfer result buffer back
}
```

cl_mem converted to pointers

- `cl_mem` handles are passed to the kernel argument list and are converted to pointers within the kernel

```
cl_mem aD;

// C Kernel Argument
clSetKernelArg (..., (void *)&aD);
```


```
__kernel void MyKernel ( __global float *a ...)
{
...
}
```

OpenCL Kernel Launch

- OpenCL kernels are launched by the host using the following call: `clEnqueueNDRangeKernel`
- The arguments:
  - *`cl_command_queue command_queue`* – a valid queue of instructions
  - *`cl_kernel kernel`* – a valid compiled program object
  - *`cl_uint work_dim`* – dimensionality of the work-items and workgroups
  - *`const size_t *global_work_offset`* – offset for global ID in each dimension (NULL is no offset)
  - *`const size_t *global_work_size`* – total number of work-items in each dimension
    - Total work-items = *`global_work_size[0]`* * *`global_work_size[1]`* * …

OpenCL Kernel Launch (Continued)

- Arguments (continued):
  - `const size_t *local_work_size` – defines the number of work-items per workgroup
    - If NULL, the OpenCL implementation will determine how to divide up the global_work_size
  - `cl_uint num_events_in_wait_list` – the number of events in the wait list
  - `const cl_event *event_wait_list` – the list of events that must be completed before the kernel launches
  - `cl_event *event` – the event attached to this kernel launch
- If the event list is NULL, the number of events must be zero

Kernel Launch - Code Example

```
//3D Work-Group, let OpenCL Runtime determine
//local work size.
size_t const globalWorkSize[3] = {512,512,512};
clEnqueueNDRangeKernel(queue, kernel, 3, NULL,
              globalWorkSize, NULL,
                     0, NULL, NULL);

//2D Work-Group, specify local work size
size_t const globalWorkSize[2] = {512,512};
size_t const localWorkSize[2] = {16, 16};
clEnqueueNDRangeKernel(queue, kernel, 2, NULL,
                     globalWorkSize, localWorkSize,
                     0, NULL, NULL);
```

Putting It All Together – Code Example

```
void main()
{
   ...
   // 1. Create then build program
   cl_program program = clCreateProgramWith…(…);
   clError = clBuildProgram(program, 1, &device, compilerOptions, NULL, NULL);

   // 2. Create kernels from the program
   cl_kernel kernel = clCreateKernel(program, ”increment”, &err);

   // 3. Allocate and transfer buffers on/to device
    float* aH = ...;
    cl_mem aD = clCreateBuffer(..., CL_MEM_COPY_HOST_PTR, aH, ...);
    cl_float bH = 10.8;

   // 4. Set up the kernel argument list
    clError = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&aD);
    clError = clSetKernelArg(kernel, 1, sizeof(cl_float), (void *)&bH);
```

Putting It All Together – Code Example (2)

```
  ...

   // 5. Launch the kernel
    size_t const globalWorkSize = 8;
    clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalWorkSize, NULL, 0, NULL, NULL);

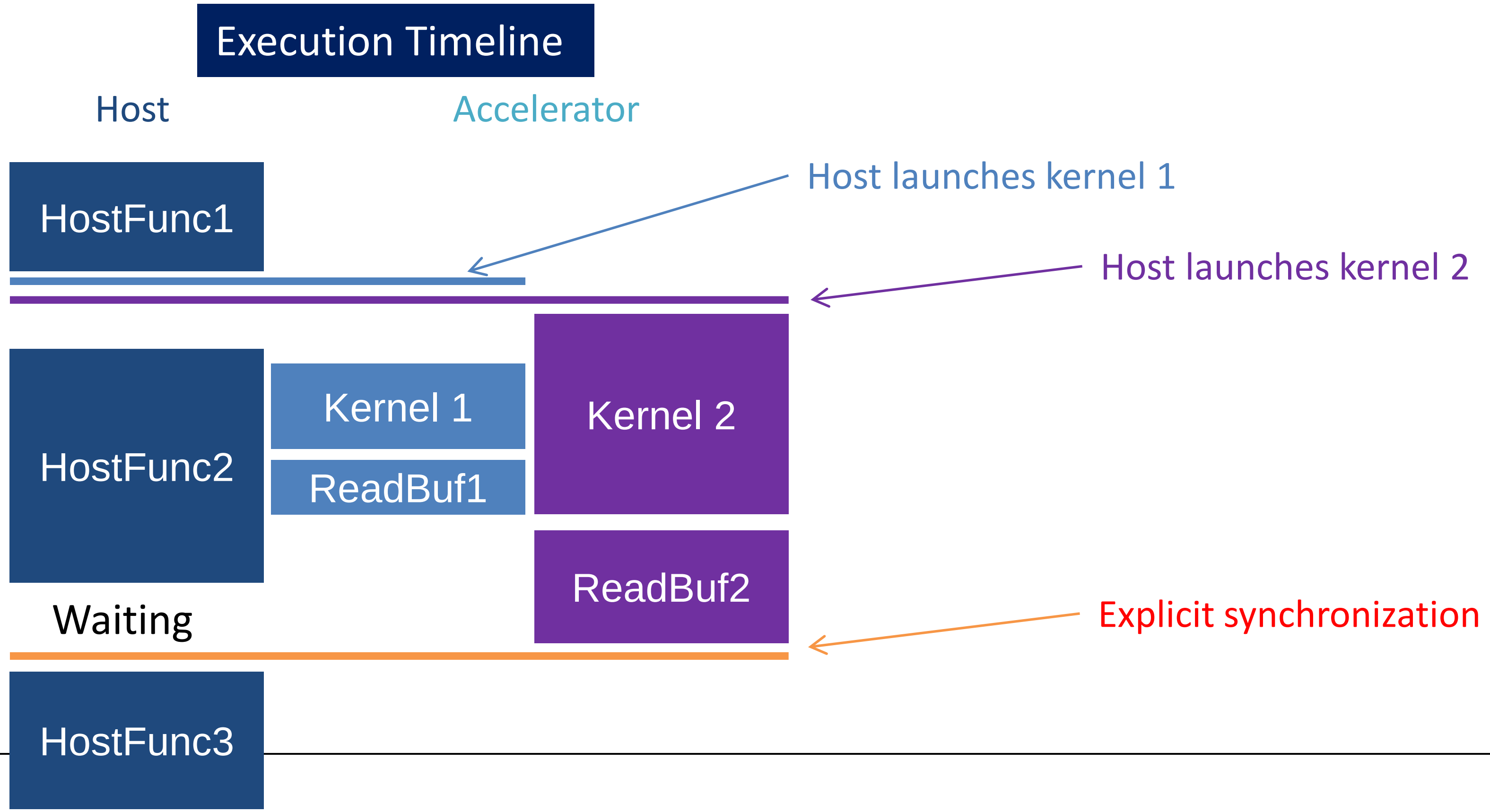
   // 6. Transfer result buffer back
    clError = clEnqueueReadBuffer(queue, aD, CL_TRUE, 0,                    8*sizeof(cl_float), aH,
    0, NULL, NULL);
}
```

Host and Kernel Execution

- Kernels execute on one or more OpenCL devices
- Host program executes on the host
- The host launches device commands asynchronously
  - Control returns to host immediately
  - Unless explicit synchronization specified
- The host manages tasks between device kernel execution
  - Memory management
  - Error handling

Asynchronous Kernel Execution

- By default, host launches device but execution is not synchronized
  - Unless explicit synchronization mechanisms are used



Explicit Synchronization Point

- Host Side
  - `clFinish(queue)`
    - Blocks until all commands in a given queue have finished execution
  - Events
    - Each `clEnqueue` task assigned an event id that can be used as a prerequisite for another `clEnqueue` task
  - Blocking memory commands
  - In-order command queue
    - All commands in an in-order queue will not execute until all commands enqueued before it in the same queue has finished executing
- Kernel Side
  - `barrier()`
    - Ensures work-items will not progress beyond a barrier until all other work-items in the same workgroup has executed the barrier

Discussion

- What are the steps required for setting up an OpenCL problem?
  - What layer does each step belong to?
- How does OpenCL for Altera differ than normal OpenCL?

Writing OpenCL Programs Agenda

- Kernels and work-item hierarchy
- Launching kernels
- **Kernel code**
- Memory model
- Clean up

Writing Kernels

- Executed once for every work-item created
- Begins with the keyword `__kernel`
- Returns `void`
- Address space of any pointer argument must be specified
  - `__local`, `__global`, or `__constant`

Kernel Restrictions

- No pointers to functions
- No recursion
- No predefined identifiers
- No static variables
- No writes to pointer or arrays of types that are less than 32-bits in size

Identifying Work-Items

CLEMSON UNIVERSITY

- OpenCL kernels have functions to identify the position of a work item in the execution range
  - Most take a dimension as a `uint` argument (0-2)

- Functions that return NDRange properties
  - Determined at kernel launch time
  - `get_work_dim()`
    - Number of dimensions used
  - `get_global_size(dim)`
    - Total number of work-items in dimension
  - `get_local_size(dim)`
    - Return size of workgroup in dimension
  - `get_num_groups(dim)`
    - Number of workgroups in dimension

Number of cars to build

Factory workers per group

Number of groups

Identifying Work-Items Continued

- Kernel functions that return appropriate index of current work-item
  - `get_global_id(dim)`
    - Index of work-item in the global space
  - `get_local_id(dim)`
    - Index of work-item within workgroup
  - `get_group_id(dim)`
    - Index of current workgroup

OpenCL Syntax - Kernels

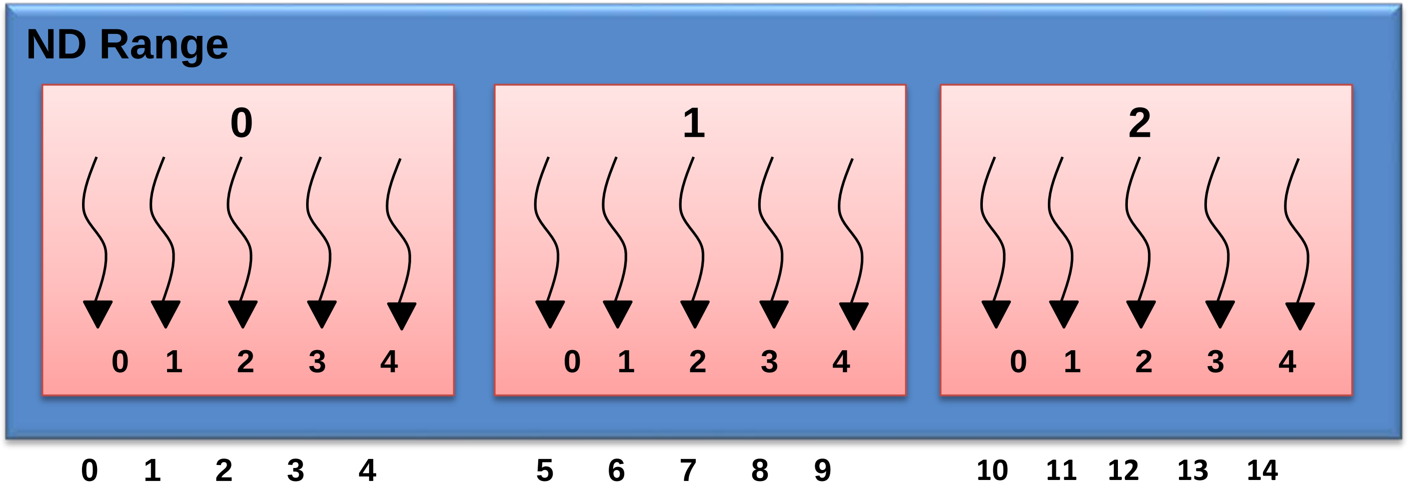
- Built-in functions are typically used to determine unique work-item identifiers:

One Dimensional Array
get_work_dim () == 1
get_local_size(0) = 5
get_global_size(0) = 15

`get_group_id(0)`

`get_local_id(0)`

`get_global_id(0)`




```
get_global_id(0) == get_group_id(0) * get_local_size(0) + get_local_id(0)
```

OpenCL Syntax – Thread Identifiers

- Result for each kernel launched with the following execution configuration:
  - Dimension = 1 Global work size = 12 Local Work Size = 4

```
__kernel void MyKernel(__global int* a)
{
   int idx = get_global_id(0);
   a[idx] = 7;
}

__kernel void MyKernel(__global int* a)
{
   int idx = get_global_id(0);
   a[idx] = get_group_id(0);
}

__kernel void MyKernel(__global int* a)
{
   int idx = get_global_id(0);
   a[idx] = get_local_id(0);
}
```

```
a: 7 7 7 7 7 7 7 7 7 7 7 7

a: 0 0 0 0 1 1 1 1 2 2 2 2

a: 0 1 2 3 0 1 2 3 0 1 2 3
```

OpenCL Kernel Statements

- C Operators
- Math Functions
  - Floating Point Operations Support
  - sin, log, exp, pow, etc.
- Call non-kernel functions
  - Create your own functions
- Flow-control statements
  - if-then-else, loops, etc.

OpenCL Syntax - Data Types

- Available in host and device code

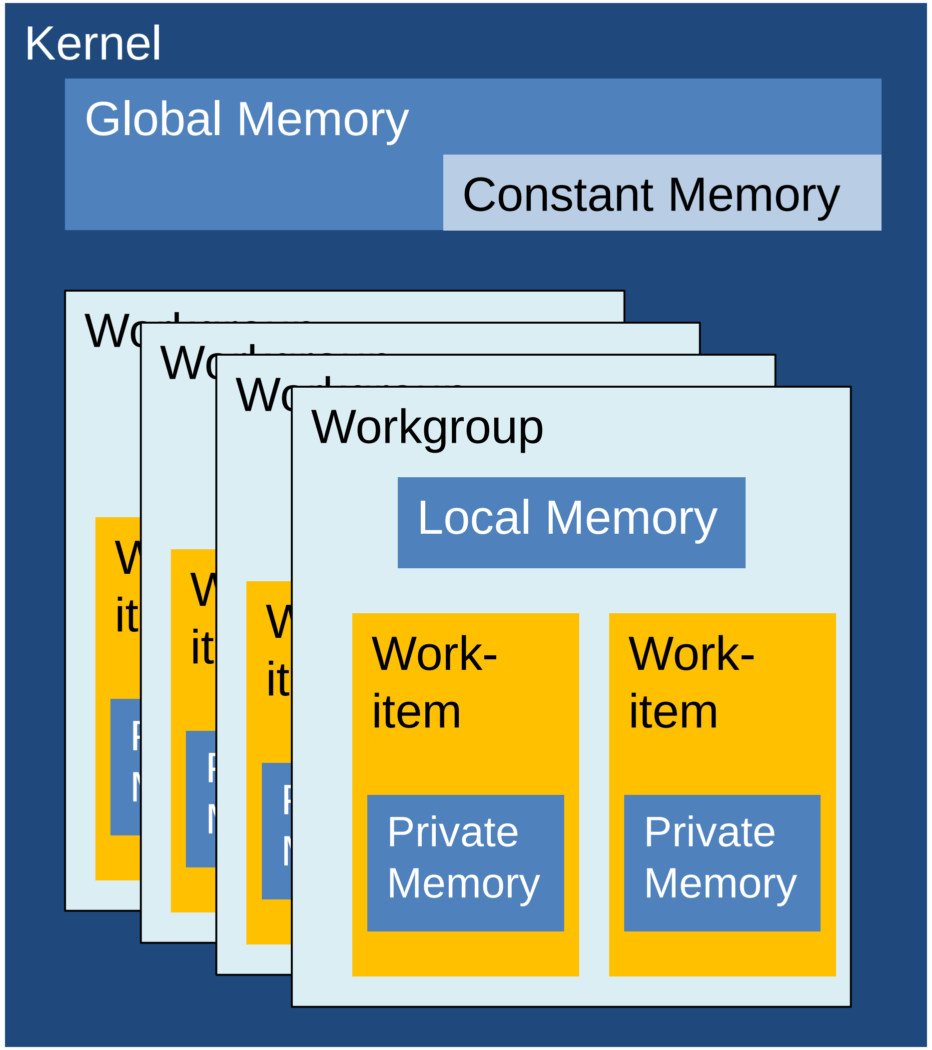
- Defines vector variants of basic integer (signed and unsigned) and floating-point types:
  - eg. `char1, char4, uchar3, float4, double2`
  - 1st, 2nd, 3rd, and 4th individual components accessed as x, y, z, and w members:
    - eg. `float3 a,b; a.x = 1.0f;  b.y = a.x;`
  - constructor function
    - Eg. `float3 a = float3(1.0f, 2.0f, 3.0f);`

Writing OpenCL Programs Agenda

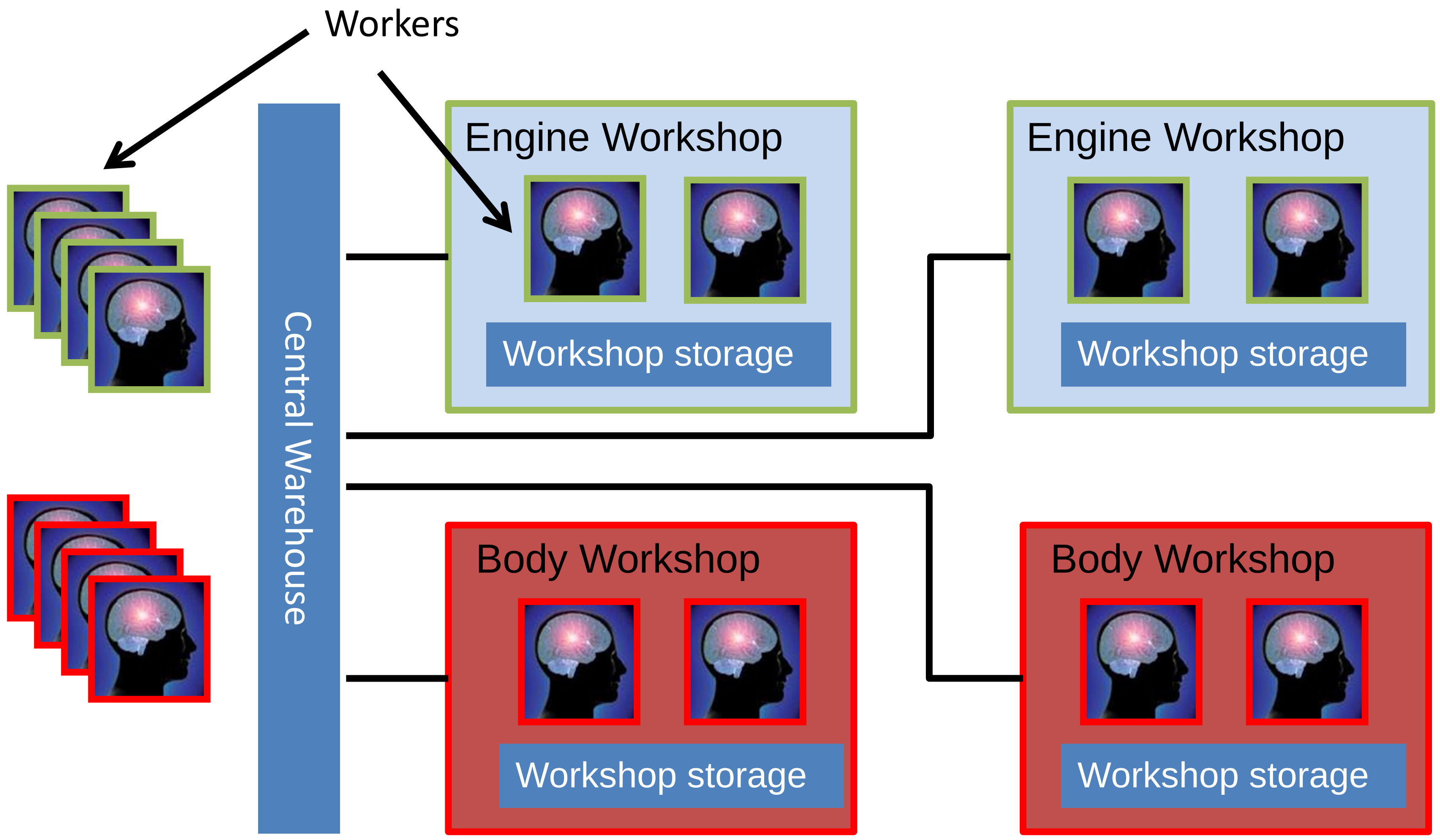
- Kernels and work-item hierarchy
- Launching kernels
- Kernel code
- **Memory model**
- Clean up

Memory Model

- Private Memory
  - Unique to work-item
- Local Memory
  - Shared within workgroup
- Global/Constant Memory
  - Visible to all workgroups
- Host Memory
  - One the host CPU



Car Factory Analogy Revisited



Car Factory Analogy Explained (2)

- Global Memory: Central warehouse
- Local Memory: Workshop storage
- Private Memory: Worker’s brain

Mapping OpenCL Memory to FPGAs

- Global
  - Off-chip DDR/QDR memory
- Constant
  - Resides in off-chip memory
  - Accessed through cache shared by all kernels
- Local
  - On-chip memory
  - Much higher bandwidth and lower latency than global memory
- Private
  - On-chip registers

OpenCL Memory Qualifiers

- OpenCL standard defines qualifiers to specify memory regions:
  - `global`, `local`, `constant`, or `private`
- Pointer arguments in kernel functions must be qualified with global, local, or constant
- Non-pointer arguments cannot be qualified with global, local, and constant
- Private is the default qualifier for function arguments and “C-local” variables
  - Any variables declared inside a kernel without a qualifier will be private – likely stored in a register
- Program scope (“C-global”) variables must be declared as constant

Memory Qualifier Syntax Example

- Increase performance by using local memory to cache data

```
__kernel void MyKernel(__global float* data)
{
   int i= get_global_id(0);

   //Shared by all work-items in the workgroup
   __local float lData[256];
   lData[get_local_id(0)] = data[i];


   …
}
```

Writing OpenCL Programs Agenda

- Kernels and work-item hierarchy
- Launching kernels
- Kernel code
- Memory model
- **Clean up**

Clean Up

- Clean up memory, release all OpenCL objects
- Check reference count to ensure it equals zero

```
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmd_queue);
clReleaseMemObject(memobj);
clReleaseContext(context);
```

Discussion

- In your pair, detail all the steps required to build an OpenCL program that completes Matrix-Vector Multiplication
  - Assume the Vector has N elements

Discussion

- Which language would you use to solve the following problems, and why?
  - Vector Addition
  - Image Thresholding
  - State Machine Design
  - Audio Filtering