Introduction to OpenCL™ for Intel® FPGAs

Agenda and Objectives

Describe high-level parallel computing concepts and challenges

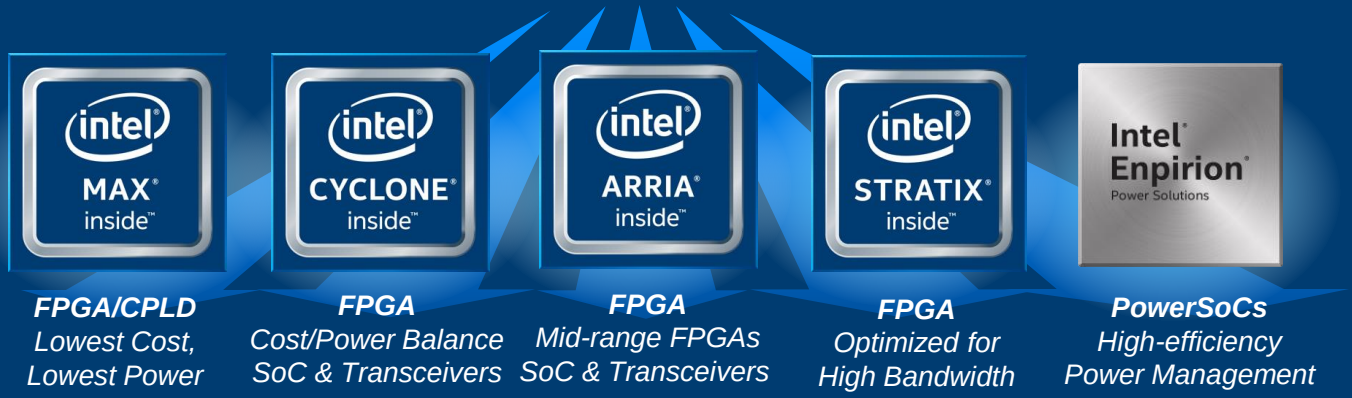
Understand the advantages of using OpenCL™ with Intel® FPGAs

Know the basics of the OpenCL standard

Write simple OpenCL programs

Compile and run OpenCL programs using the Intel FPGA SDK for OpenCL

Innovation Across the Board



RESOURCES

Embedded Soft and Hard Processors

Nios® II

Arm*

Design Software

Intel® Quartus® Prime Design Software

Intel® FPGA SDK for OpenCL™

Development Kits

Intellectual Property (IP)

- Industrial
- Computing
- Enterprise

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

Executing OpenCL Kernels

NDRange Kernels

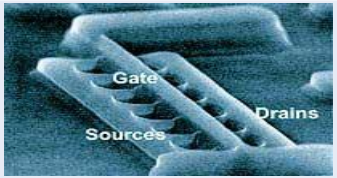
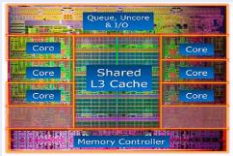
OpenCL on Intel® FPGAs

Parallel Computing

"A form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (in parallel)"

~ Highly Parallel Computing, Amasi/Gottlieb (1989)

Need for Parallel Computing

Power Wall	
	Unable to scale power consumption with reduction in process node Maximum processor frequency capped
Instruction-Level Parallelism Wall	
	Compilers and processors can't extract enough parallelism from a single instruction stream to keep processor architecture busy
Memory Wall	
	Memory architectures have limited bandwidth Can't keep up with the processor

Programmer-Specified Parallelism

Allow software programmer to define and control parallelism

- Programmers know the algorithm the best
- Allow programmers to find activities that can be executed in parallel
- Expressed explicitly or implicitly
- Expressed at different levels that are higher than instruction-level parallelism
- Likely more effective than compiler/processor extracted parallelism

Types of Parallelism

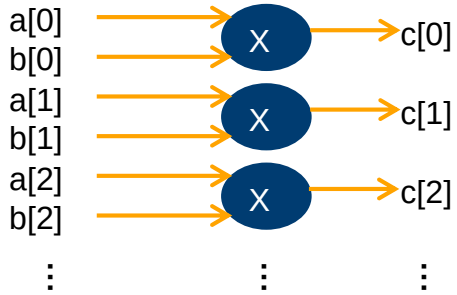
- Data Parallelism
  - Input data separated and sent to parallel resources, results recombined
  - Scatter-gather
- Task Parallelism
  - Decompose problem into sub-problems that run well on available compute resources
  - Divide-and-conquer
- Pipeline Parallelism
  - Task parallelism where tasks have a producer consumer relationship
  - Different tasks operate in parallel on different data

Data Parallelism

Same operation(s) applied across different data in parallel

- Single Program Multiple Data (SPMD)
- Single Instruction Multiple Data (SIMD)

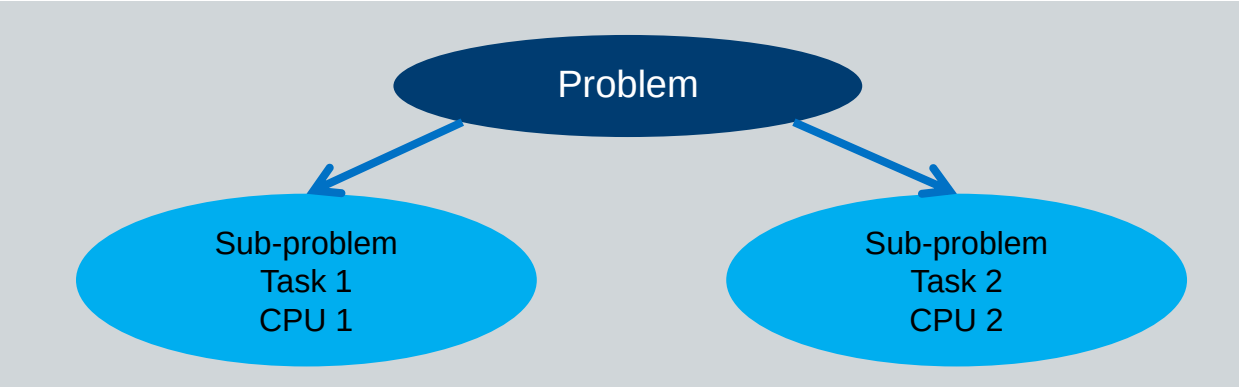
```
for (i = 0; i < N ; i++)
     c[i] = a[i] * b[i]
```



Task Parallelism

Decompose problem into sub-problems (tasks)

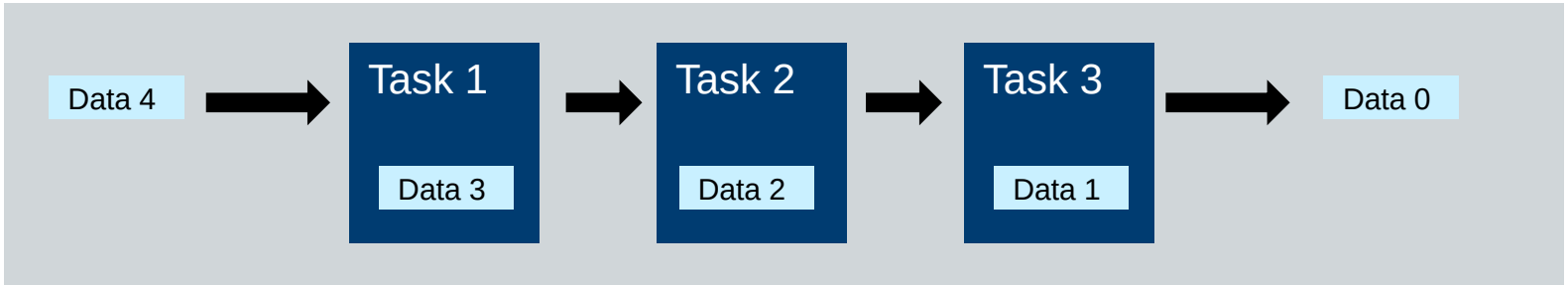
- Tasks operate on same or different data
- Example: Multi-CPU system where each CPU execute a different thread
- A.K.A. Simultaneous Multithreading (SMT), Thread/Function Parallelism



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

- Fundamental challenge of parallel programming
- 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 shared
- Synchronization mechanisms
  - Barriers
    - Stop tasks at certain point until all tasks reach the barrier
  - Locks
    - Enforce limits on access of particular resources
  - Parallel computing environment must handle this effectively

Heterogeneous Computing Systems

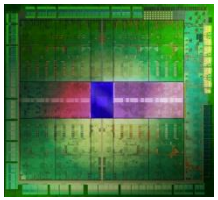
- Modern systems contain more than one kind of processor
- Applications exhibit different behaviors
  - Control intensive (Searching, parsing, etc…)
  - Data intensive (Image processing, data mining, etc…)
  - Compute intensive (Iterative methods, financial modeling, etc…)
- Gain performance by using specialized capabilities of different types of processors

Example Heterogeneous System

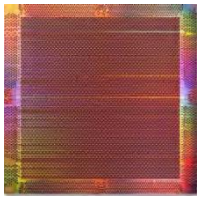
- Modern computing platform contains many dissimilar processors
  - Multi-core, general purpose, central processing units (CPUs)
  - Digital Signal Processing (DSPs) processors
  - Graphics Processing units (GPUs)
  - Field Programmable Gate Arrays (FPGAs)

CPUs

DSPs



GPUs



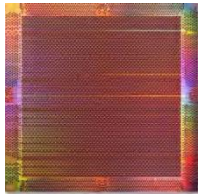
FPGAs

- Challenge: How to build a software ecosystem for a heterogeneous platform?

Traditional Approach to Heterogeneous Computing

- Write software for each software programmable architecture CPU, GPU, DSP
  - Using different languages and vendor specific tools

- Develop custom parallel hardware for FPGA
  - Fine-grained parallelism
  - Write HDL
  - Simulation, timing closure, on-chip verification etc.



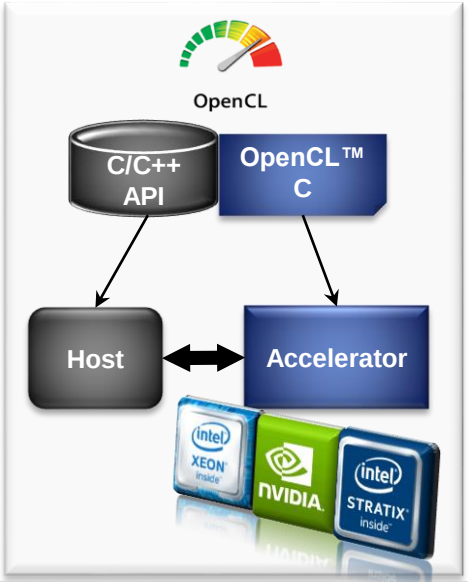
Programmers Dilemma

"The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two yeah; four not really; eight, forget it."

~ Steve Jobs, 1955-2011

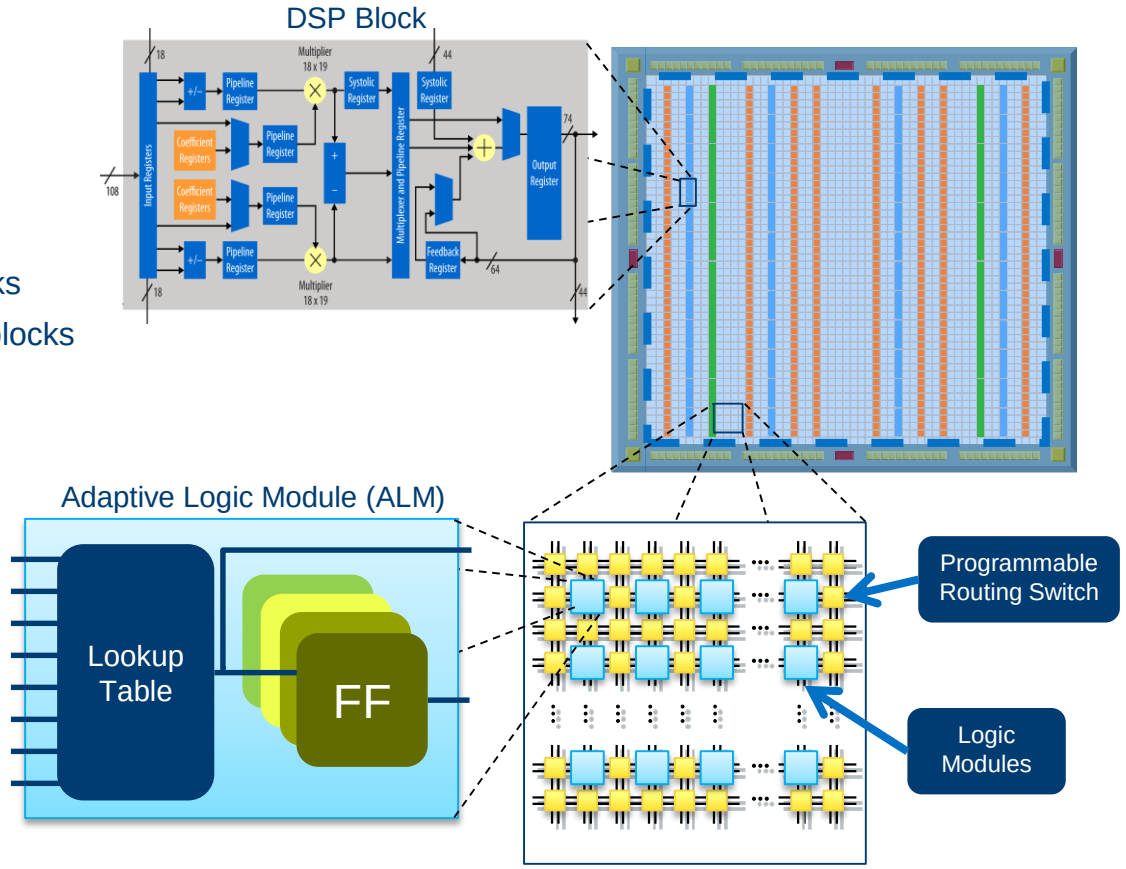
What is OpenCL™?

- Open Computing Language (OpenCL™) - Framework for heterogeneous computing
  - General purpose programming model for multiple platforms
  - Host API and kernel language
  - Low-level Programming language based on C/C++
  - Provides increased performance with hardware acceleration
- Open, royalty-free standard
  - Managed by Khronos* Group
    - Intel® is an active member
  - http://www.khronos.org

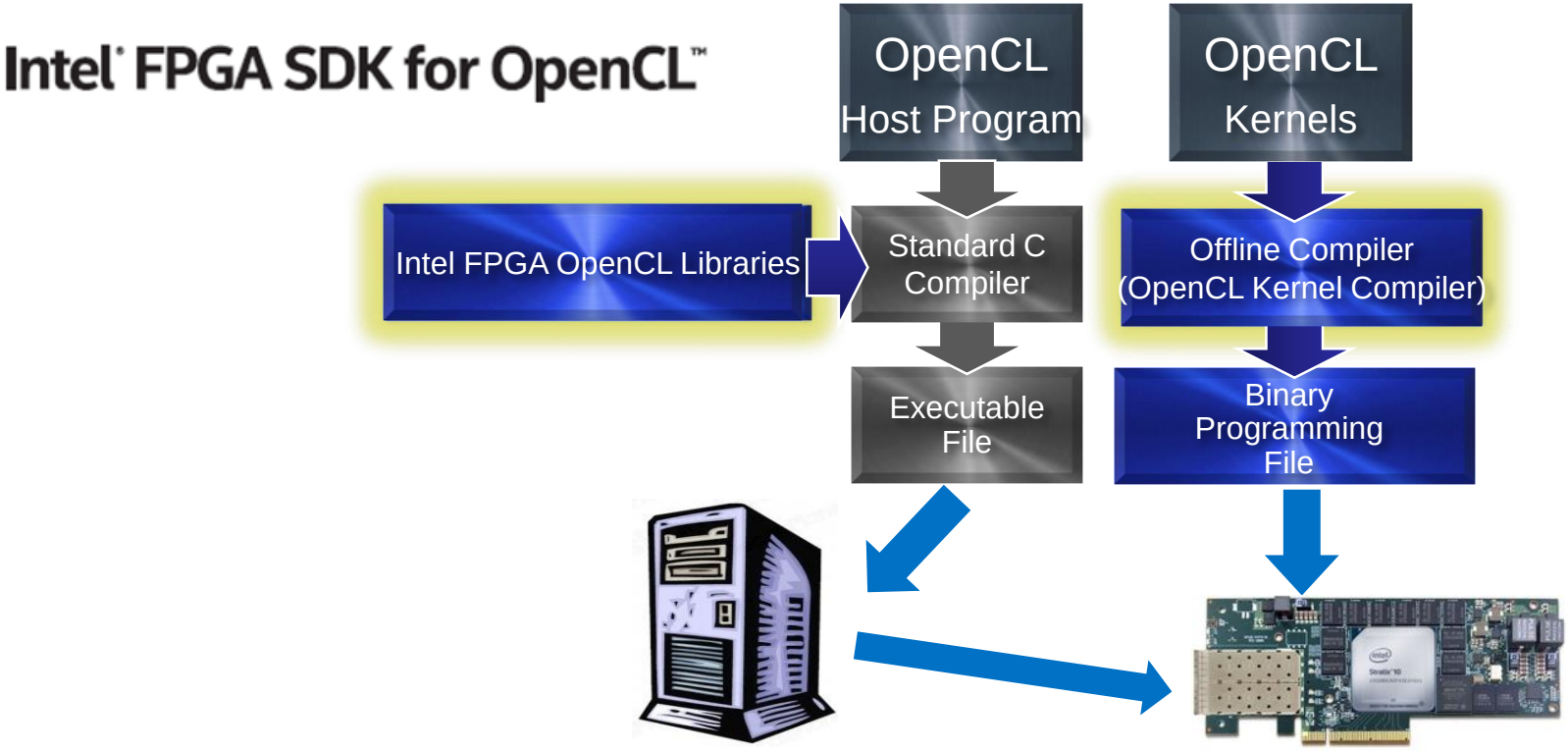


FPGA Architecture

- Massive Parallelism
  - Millions of logic elements
  - Thousands of embedded memory blocks
  - Thousands of Variable Precision DSP blocks
  - Programmable routing
  - Dozens of High-speed transceivers
  - Various built-in hardened IP
- FPGA Advantages
  - **Custom hardware!**
  - Efficient processing
  - Low power
  - Ability to reconfigure
  - Fast time-to-market

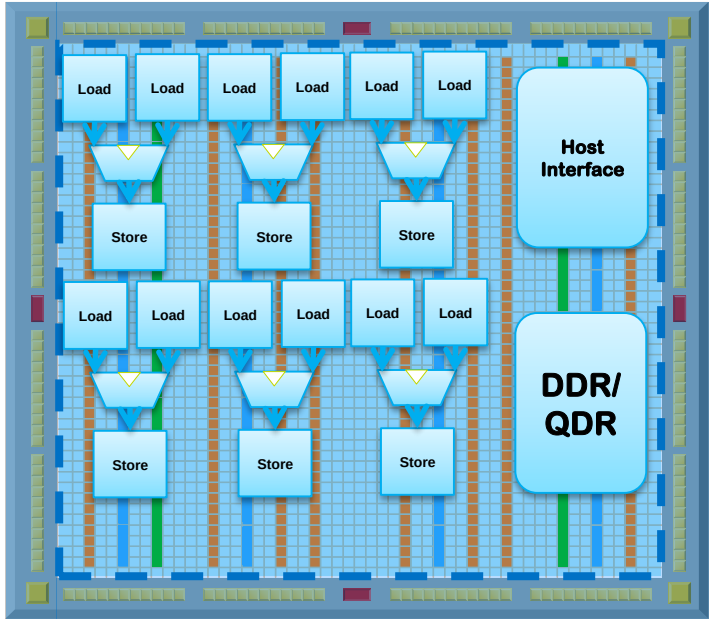


Intel® FPGA SDK for OpenCL™ Usage



Compiling OpenCL™ to Intel® FPGA

- Custom hardware generated automatically for each kernel
  - Get the advantages of the FPGA without the lengthy design process
- Organized into functional units based on operation
- Able to execute OpenCL™ threads in parallel



Benefits of OpenCL™ on FPGAs

- Faster software-centric development flow
  - OpenCL™ flow abstracts away FPGA hardware flow
  - C-based design leads to shorter architectural exploration and development time
  - Allows software developers to develop FPGA hardware
- Obtain performance and power advantages of an FPGA
- Portability & Obsolescence free
  - Portability between different HW accelerators (CPU, GPU, FPGA, etc)
  - Ports easily to new generations of the FPGA
    - Compiler automatically optimizes for new architectural features

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

- **OpenCL overview**
- Platform Layer API
- Runtime Layer API

Executing OpenCL Kernels

NDRange Kernels

OpenCL on Intel® FPGAs

OpenCL™ Characteristics

- Provides parallel computing using task- and data-based parallelism
- Includes a C99 based language for writing functions that execute on OpenCL™ accelerators
- Provides abstract models
  - Generic: able to be mapped on to significantly different architectures
  - Flexible: able to extract high performance from every architecture
  - Portable: vendor and device independent

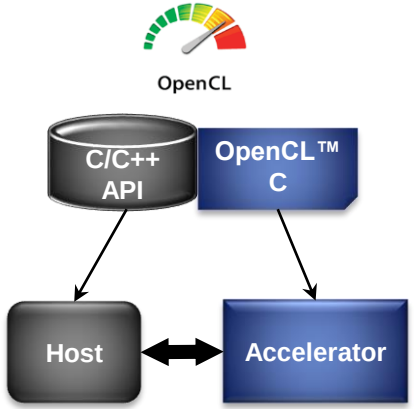
OpenCL™ Versions

- 1.0 (2008)
- 1.1 (2010)
  - Image data type
- 1.2 (2011)
  - Printf support
- 2.0 (2013)
  - Shared virtual memory
  - Pipes
- 2.2 (2017)
- Backwards compatible

OpenCL™ Specification Defined by Four Models

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

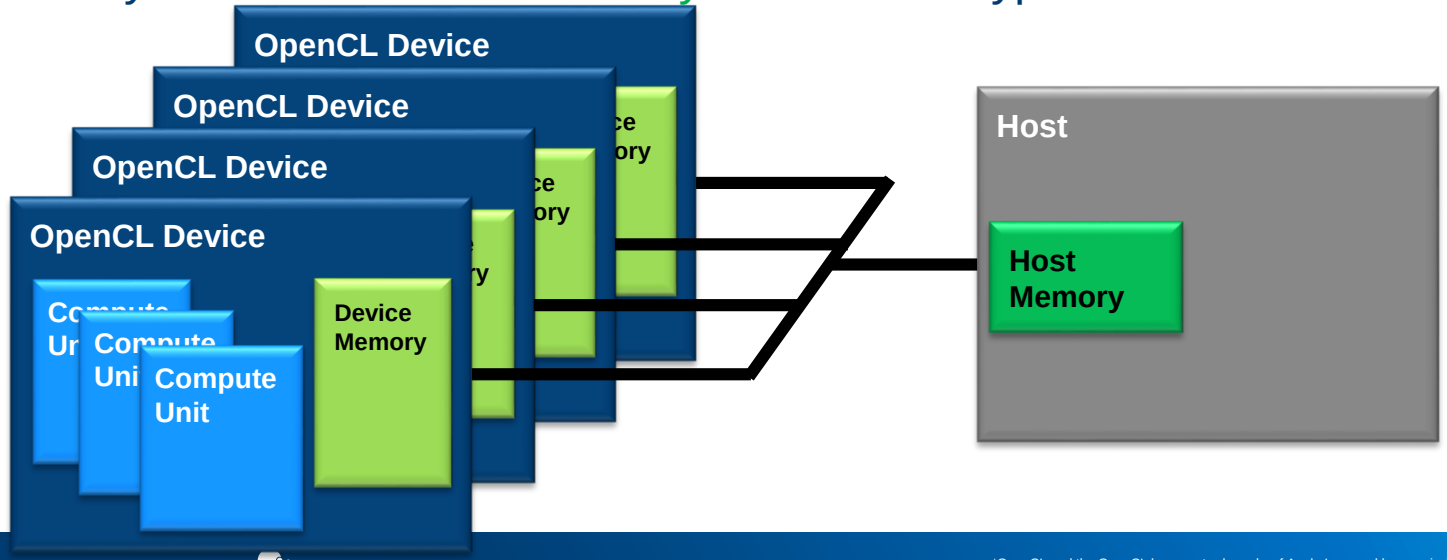
Two Sides of OpenCL™ Standard



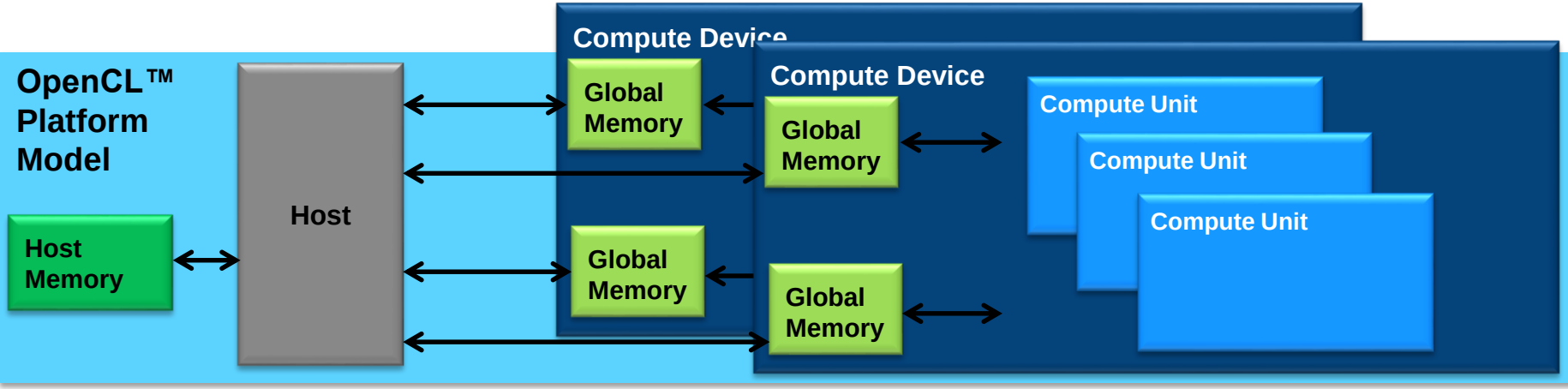
- Kernel Function
  - OpenCL™ C
  - Software that runs on accelerators (OpenCL devices)
  - Usually used for computationally intensive tasks
- Host Program
  - Software running on conventional microprocessor
  - Supports efficient plumbing of complicated concurrent programs with low overhead
    - Through OpenCL host API
- Used together to efficiently implement algorithms

OpenCL™ Platform Model

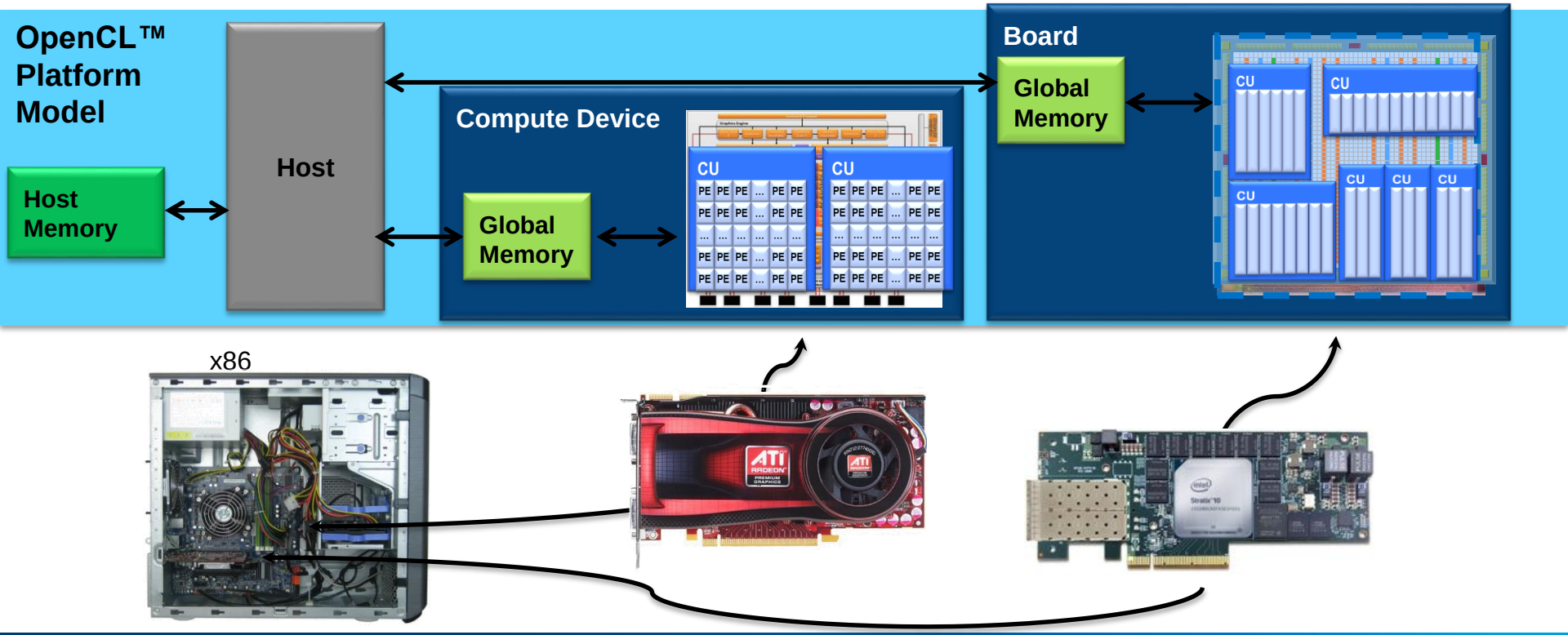
- One Host with one or more OpenCL™ Devices
  - Each Device is composed of one or more compute units
- Memory divided into Host Memory and various types of Device Memory



Heterogeneous Platform Model



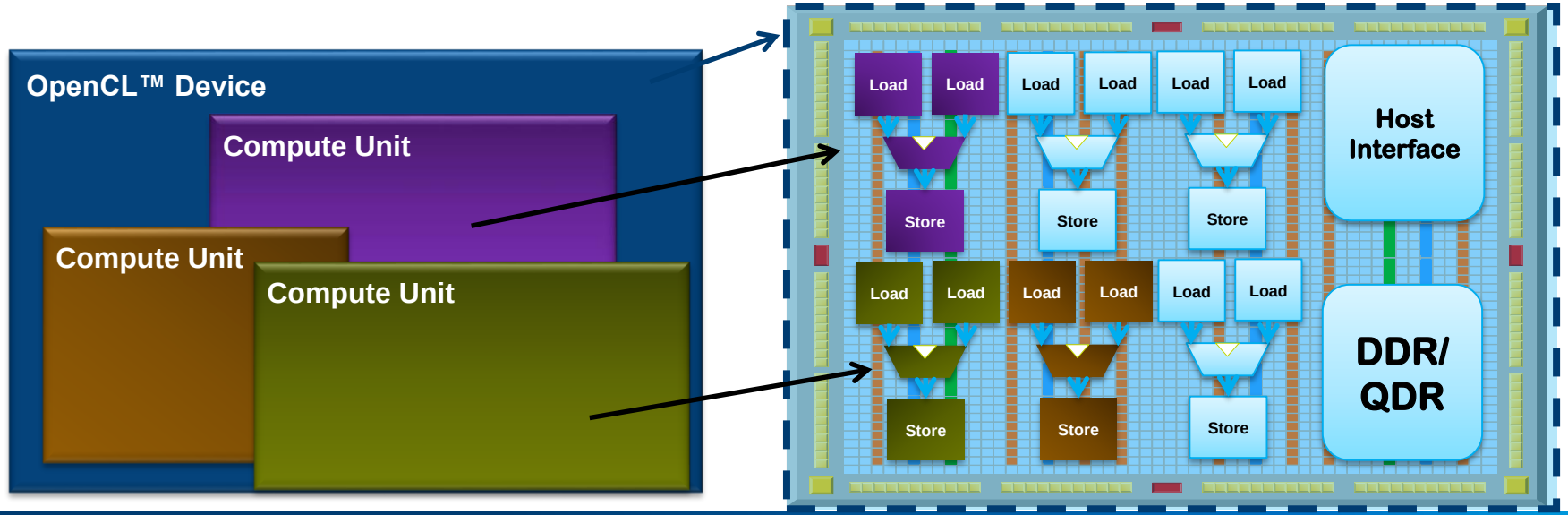
Heterogeneous Platform Model



Intel® FPGA OpenCL™ Device

Each device is made of many independent compute units

- Each compute unit is custom built from kernel code



Data Parallel Execution Model

Execute a single kernel with multiple threads

Implicit Parallelism

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

Data Parallelism (SPMD)

```
myqueue.enqueueWriteBuffer(Buffer_In,…)
myqueue.enqueueNDRangeKernel(…)
myqueue.enqueueReadBuffer(Buffer_Out,…)
```

```
__kernel void _foo
   (__global float *x)
{
   int i = get_global_id(0);
   u[i] = foo(x[i]);
}
```

Task Parallel Execution Model

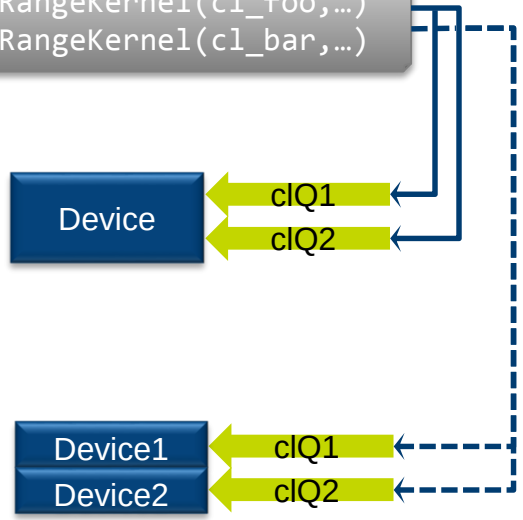
Execute multiple kernels in parallel

Implicit Parallelism

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

Task Parallelism (SMT)

```
clQ1.enqueueNDRangeKernel(cl_foo,…)
clQ2.enqueueNDRangeKernel(cl_bar,…)
```



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
  - Loop pipeline parallelism is created when the compiler analyzes dependencies between iterations of a loop and pipelines each iteration for acceleration
- Data storage and movement is explicit
  - Hierarchical abstract memory model
  - Up to the programmer to manage memories and bandwidth efficiently

OpenCL™ Host APIs

The host program, through a set of OpenCL™ APIs, setup the environment and manages the execution of kernels on the devices

- Defined by the standard in a C header file (`opencl.h`)
  - Provided along with implementation by individual solution vendors
- C++ API (`cl.hpp`)
  - Wrapper around the C API
  - Uses classes and C++ standard library containers
  - Simpler
  - More error checking capabilities
  - Focus of this class (C API covered in appendix)

OpenCL™ Platform Layer and Runtime Layer API

OpenCL™ API divided into two layers

- Platform Layer API
  - Discover platform and device capabilities
  - Setup execution environment
- Runtime Layer API
  - Executes compute kernels on devices
  - Manage device memory

Host Managed OpenCL™ Objects

OpenCL™ API controls device execution using the following objects

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

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

- OpenCL overview
- Platform Layer API
- Runtime Layer API

Executing OpenCL Kernels

NDRange Kernels

OpenCL on Intel® FPGAs

Platform Layer API

Setup device execution environment

- Necessary to allow for heterogeneous environments and multiple devices

- Tasks
  - Allows host to discover devices and capabilities
  - Query, select and initialize compute devices
  - Create compute contexts to manage OpenCL™ objects

Typical Platform Layer Steps

1. Query platforms
2. Query devices
3. Create a context for the devices

- Setup code written once and can be reused for all project with the same HW

Platform IDs

Platform: Vendor-specific implementation of OpenCL™

- Obtain the list of platforms available

```
static cl_int cl::Platform::get(VECTOR_CLASS<Platform>* platforms)
```

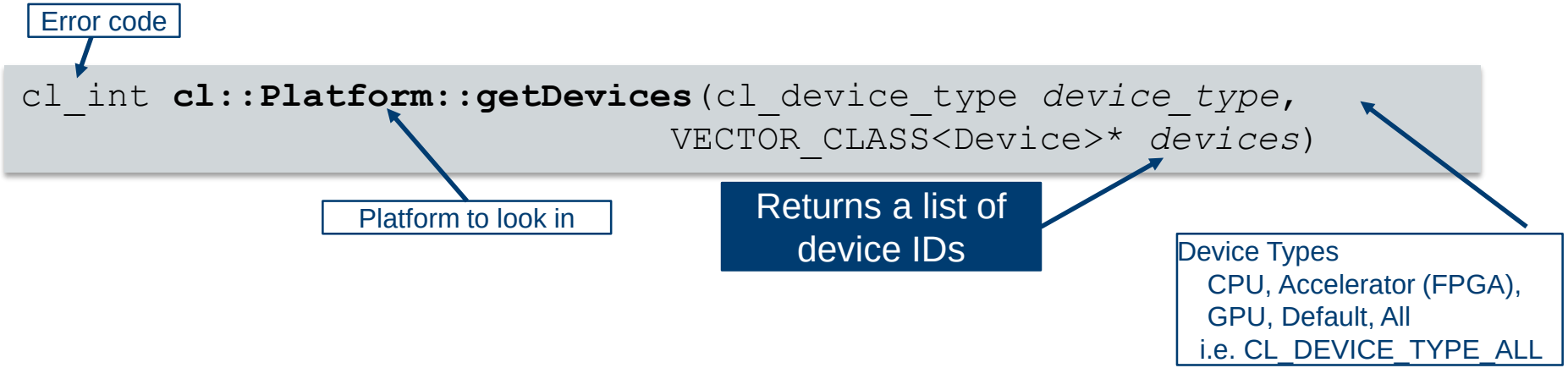
Error code

Returns a list of platform IDs

Device IDs

Device: An OpenCL™ accelerator supported by a platform

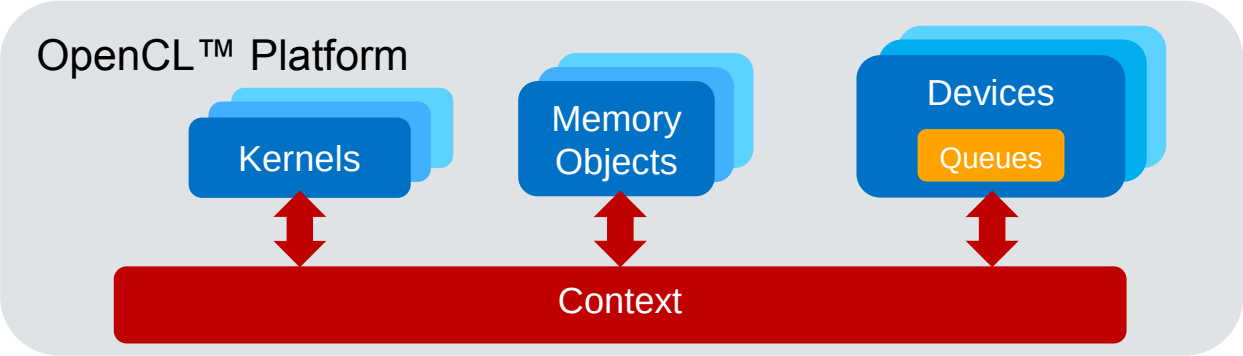
- Obtain the list of devices available



Context

Abstract containers that manage host device interaction

- Purpose
  - Coordinates the mechanisms for host-device interaction
  - Manages the device memory
  - Keeps track of kernels to be executed on each device



Create Context

Create a context with one or more devices

Constructor

```
cl::Context::Context (VECTOR_CLASS<Device> devices,
                      cl_context_properties *properties=NULL,
                      void (CL_CALLBACK  *pfn_notify)(
                                  const char *errinfo,
                                  const void *private_info,
                                  ::size_t cb,
                                  void *user_data)=NULL,
                      void *user_data=NULL,
                      cl_int *err=NULL)
```

List of devices in context

Properties that define context behavior

Callback function to be registered to handle errors in the context

Data argument for *pfn_notify*

Error code

May also use alternative constructor with device type instead of devices

Platform Layer APIs Called to Setup Environment

1. Call `cl::Platform::get` to retrieve a list of platforms
2. Call `cl::Platform::getDevices` to retrieve devices in a given platform
3. Create `cl::Context` object that manages kernel execution

Example Platform Layer Code

```
//Get the Platforms
std::vector<cl::Platform> plist;
err=cl::Platform::get(&plist);

// Get the FPGA devices in the first platform
std::vector<cl::Device> mydevlist;
err=plist[0].getDevices(CL_DEVICE_TYPE_ACCELERATOR, &mydevlist);

//Create an OpenCL™ context for the FPGA devices
cl::Context mycontext (mydevlist);
```

Query Info About a Platform, Device, or Context

- Use `cl::Platform::getInfo` to return information about a platform
  - Vendor, OpenCL™ version, platform profile, extensions, etc…
  - Use `cl_platform_info` enum type
- Use `cl::Device::getInfo` to return information about a device
  - Memory sizes, Bus widths, Device Type, Endianness, etc.
  - Use `cl_device_info` enum type
- Use `cl::Context::getInfo` to return information about the context
  - Number of devices in context, context properties, and reference count
  - Use `cl_context_info` enum type

Error Codes

- Every OpenCL™ host functions call will generate a error code
  - Either as the return value or as a pointer function argument
- Host functions designed to return even when they error out
  - Check error code to debug
  - Error codes are negative
  - CL_SUCCESS==0
- Defined by the OpenCL™ specification
  - Definitions located in cl.h

```
/* 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
```

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

- OpenCL overview
- Platform Layer API
- Runtime Layer API

Executing OpenCL Kernels

NDRange Kernels

OpenCL on Intel® FPGAs

Runtime Layer API

Execute kernels on the device

- Tasks
  - Memory management
    - Allocate/deallocate device memory
    - Read/write to the device
  - Run kernels on the device
  - Host/device synchronization

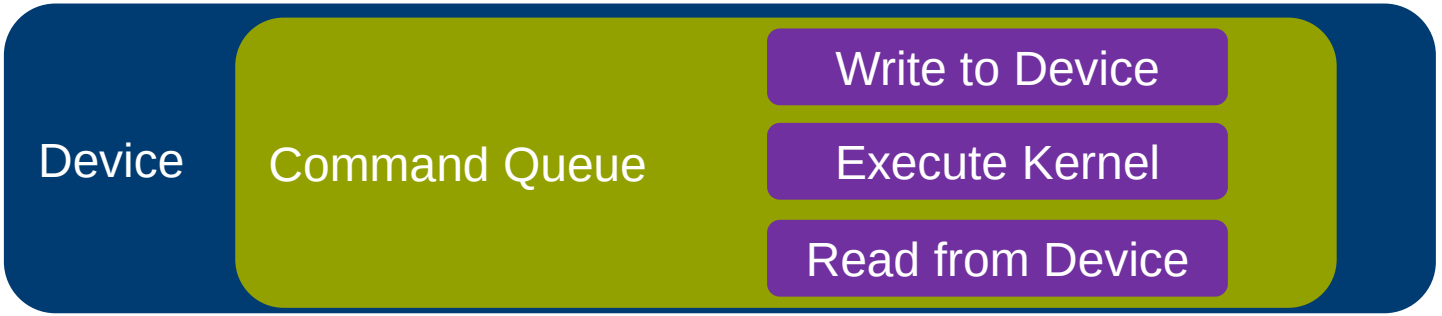
Typical Runtime Layer Steps

1. Create a command queue
2. Write to the device
3. Launch kernel
4. Read results back from the device

Command Queue

Mechanism for host to request action by the device

- Each command queue associated with one device
  - Each device can have one or more command queues
- Host submits commands to the appropriate queue
- Operations in the queue will execute in-order for Intel® FPGAs



Create a Command Queue

Creates a command queue associated with a device

Constructor

```
cl::CommandQueue::CommandQueue(
                        const Context& context,
                        const Device& device,
                        cl_command_queue_properties properties=0,
                        cl_int *errcode_ret=NULL)
```

Valid context

A device associated with *context*

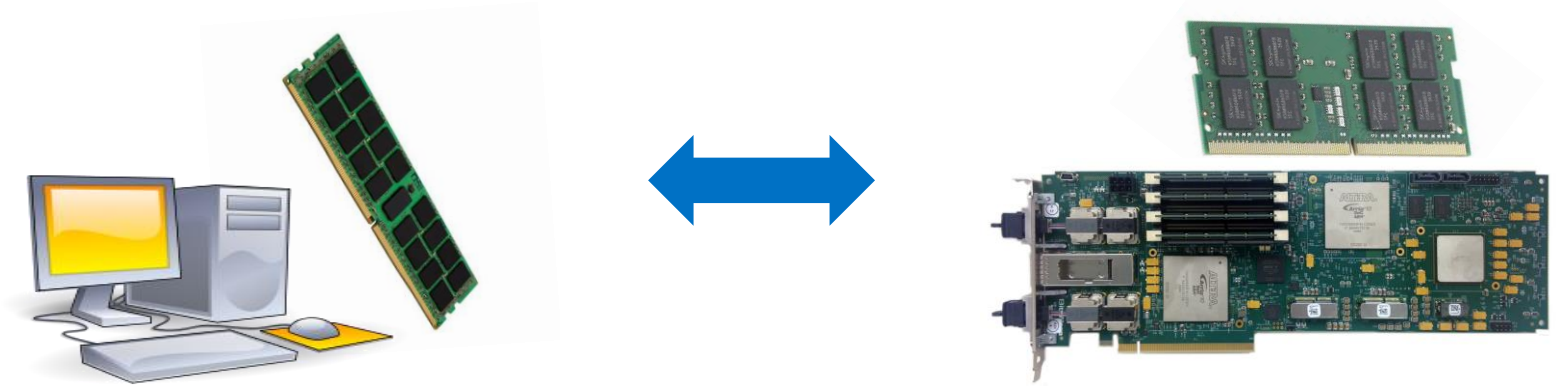
Error code

Queue properties
e.g. Turn on profiling

- Host will submit commands to the device through the command queue
  - Using `CommandQueue::enqueue…` commands
    - e.g. read, write, execute kernel, etc..

Host / Device Physical Memory Space

- The host and the device each has its own physical memory space
  - Data needs to be physically located on a device before kernel execution
- Use OpenCL™ API functions to allocate, transfer, and free device memory
  - Using **memory objects** through command queues



Memory Objects

Representation of device memory on the host

- Data encapsulated as memory objects in order to be transferred to/from device
- Valid within one context
  - Runtime manages the memory objects and actual location on devices
- OpenCL™ specification defines two types
  - Buffers (One dimensional collection of elements)
    - Can be scalars (int, float), vector data types, or user-defined structures
  - Images
    - Simplifies the process of representing and accessing images
    - Not discussed in this class

Buffer Creation

Allocates and creates a buffer memory object

- Similar to `malloc` and `new`
  - Except buffer is not a pointer on the host

- A buffer is passed as a kernel argument and converted to **pointers in the kernel**

Constructor

```
cl::Buffer::Buffer( const Context& context,
                    cl_mem_flags flags,
                    ::size_t size,
                    void *host_ptr=NULL,
                    cl_int *err=NULL)
```

Valid context

Allocation and usage information *(Upcoming slide)*

Size in bytes

Pointer to data already allocated in the host application (Optional)

Error code

Memory Management Buffer Flags

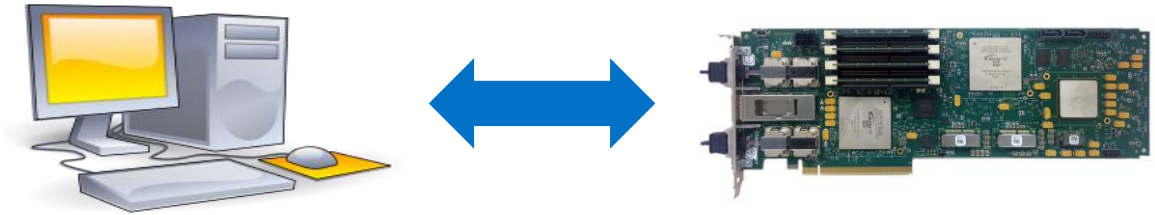
- Kernel access permissions (from the kernel perspective)
  - `CL_MEM_READ_WRITE` (default), `CL_MEM_WRITE_ONLY`, `CL_MEM_READ_ONLY`
- Host access permissions (from the host perspective)
  - `CL_MEM_HOST_WRITE_ONLY`, `CL_MEM_HOST_READ_ONLY`, `CL_MEM_HOST_NO_ACCESS`
- Host pointer options
  - `CL_MEM_COPY_HOST_PTR`
    - Data is copied from *`host_ptr`* to allocated device memory one time
  - `CL_MEM_USE_HOST_PTR`
    - *`host_ptr`* location is used for storage for the memory object
  - `CL_MEM_ALLOC_HOST_PTR`
    - Allocate in host accessible memory, used in SoC devices with shared physical memory

```
cl::Buffer mybuf(mycontext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                 sizeof(float)*32, host_vector, &error)
```

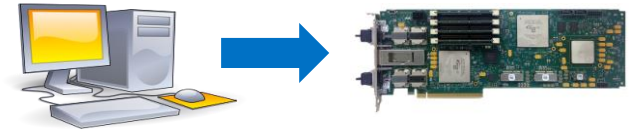
Data Transfers Calls

Use Read and Write Host API calls to explicitly transfer data from/to the device

- Commands placed on the command queue
- If kernel dependent on the buffer is executed on the accelerator device, buffer is transferred to the device
- Runtime determines precise timing of data movement



Writing to Device Buffer

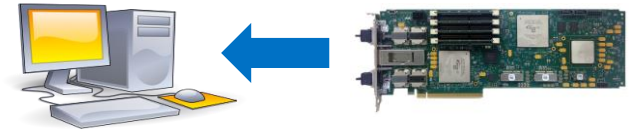


Write from host memory to buffer object (device)

```
cl_int cl::CommandQueue::enqueueWriteBuffer(
    const Buffer& buffer,
    cl_bool blocking_write,
    ::size_t offset,
    ::size_t size,
    const void *ptr,
    const VECTOR_CLASS<Event> *events=NULL,
    Event* event=NULL)
```

- Error code
- Destination Buffer
- Source host pointer
- Set to CL_TRUE blocks call until *ptr* can be reused by the host
- Offset in bytes in the buffer
- Size in bytes of data to be written
- Events used for synchronization. Discussed later

Reading from Device Buffer



Read from buffer object (device) to host memory

```
cl_int cl::CommandQueue::enqueueReadBuffer(
    const Buffer& buffer,
    cl_bool blocking_read,
    ::size_t offset,
    ::size_t size,
    const void *ptr,
    const VECTOR_CLASS<Event> *events=NULL,
    cl_event *event=NULL)
```

Error code

Source Buffer

Set to CL_TRUE blocks call until buffer data copied to *ptr*

Offset in bytes in the buffer

Size in bytes of data to be read

Destination host pointer

Events used for synchronization. Discussed later

Memory Management – Code Example

```
const int N = 5;
int nBytes = N*sizeof(int);
int hostarr [N] = {3,1,4,1,5};

//Create an OpenCL™ command queue
cl::CommandQueue myq(mycontext, mydevlist[0]);

// Allocate memory on device
cl::Buffer buf_a(mycontext, CL_MEM_READ_WRITE, nBytes);

// Transfer Memory
cl_int err;
err=myq.enqueueWriteBuffer(buf_a, CL_FALSE, 0, nBytes, hostarr);
```

Exercise 1

Setting Up OpenCL™ Host-Side Application

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

Executing OpenCL Kernels

- **Writing kernels**
- Launching kernels

NDRange Kernels

OpenCL on Intel® FPGAs

OpenCL™ Kernels

Functions that run on OpenCL™ devices

- Begins with the keyword `__kernel`
- Returns `void`
- Pointers in kernels should be qualified with an address space
  - `__private`, `__local`, `__global`, or `__constant`
  - Discussed later
- Kernel language derived from ISO C99 with certain restrictions

```
__kernel void my_kernel (__global float *data) {
}
```

OpenCL™ Kernel Statements

- C Operators
  - +, *, %, <<, ?:, &, &&, ~, !, ++, ==, etc.
- Math functions (math.h functions included implicitly)
  - `sin, acos, log, exp, pow, floor, fabs, fma, fmod`, etc.
- Call user-defined non-kernel functions
- Flow-control statements
  - if-then-else, loops, etc
- Preprocessing directives defined by C99
  - e.g. #include

OpenCL™ Kernel Restrictions

- No pointers to functions
- No recursion
- No predefined identifiers
- No writable static variables

OpenCL™ Data Types

- Scalar data types
  - `char, ushort, int, uint, long, float, double, bool`, etc
  - On the host, recommended to use `cl_` prefixed data types to ensure size compatibility and maximum portability
    - e.g. `cl_float, cl_int, cl_ulong`, etc…
- Image types
  - `image2d_t, image3d_t, sampler_t`
- User-defined structures
- Vector data types
  - Next slide

Vector Data Types

OpenCL™ supports vector variants of basic data types

- Supported size of vectors: 2, 3, 4, 8, 16
- Available in host and kernel code
  - Kernel type example: `char2, ushort3, int8, float16`, etc
  - Host type example: `cl_char2, cl_ushort3, cl_int8, cl_float16`, etc
- Aligned at vector length

Accessing Vector Data Type Components

- Vector data types with 1 to 4 components can be addressed as .xyzw

```
float4 c;
c.xyzw = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
c.z = 1.0f;
c.xy = (float2)(3.0f, 4.0f);
```

- All vector components can be addressed using a numeric index
  - <vector_name>.s<index> notation
    - Index range is 0 up to f (or F)
    - x.sa (or x.sA) refers to the 11th element of the variable x

```
float4 f, a;
a.s3 = f.y;
a.xyzw = f.s0123;
```

- Vector operations

```
int4 a, b, c;
c = a + b;
```

=

```
c.x = a.x + b.x;
c.y = a.y + b.y;
c.z = a.z + b.z;
c.w = a.w + b.w;
```

Kernel Example

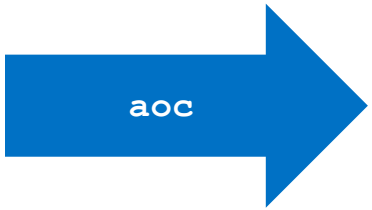
```
__kernel void my_kernel ( __global float *a,
                          __global float *b,
                          __global float *c,
                          int N)
{
    int index;
    for (index = 0; index < N; index++)
        c[index] = a[index] + b[index];
}
```

Compiling OpenCL™ Kernel to FPGAs

Kernels are compiled offline using an Offline Compiler (AOC)

- Kernels are first translated into an AOC Object file (.aoco)
  - Represents the FPGA hardware system
- Object file used to generate the AOC Executable file (.aocx)
  - Used to program the FPGA or Flash

```
// kernel.cl
__kernel void KernelName(…)
{
   int index;
   for (index = 0; index < N; index++)
      c[index] = a[index] + b[index];
}
```



Compile Kernels

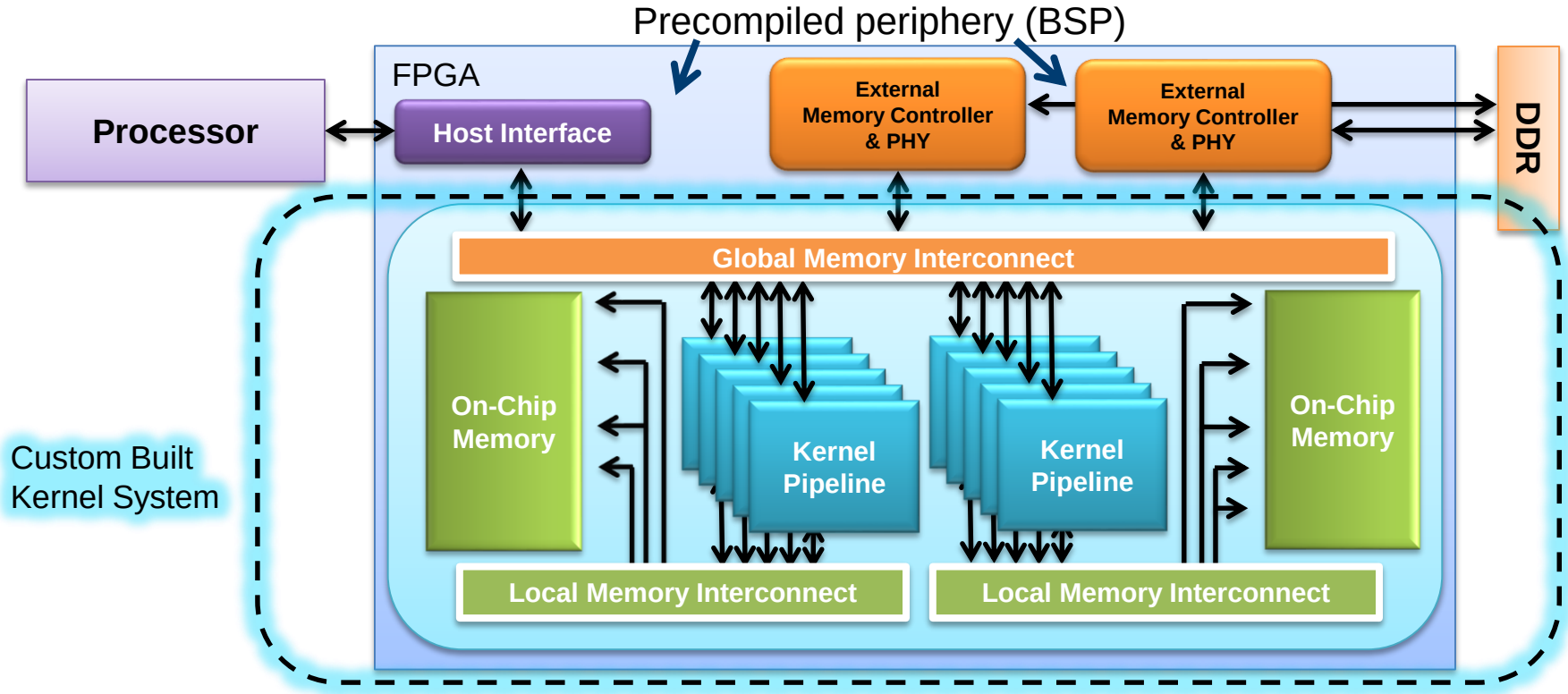
Run the Offline Compiler

- `aoc -list-boards`
  - List boards available to target within the installed board packages
- `aoc -board=<board> <kernel file>`
  - Compile the kernel to the specified board in the board package
  - Generates the kernel hardware system and compiles it using the Quartus® Prime software targeting a specific board

aoc Output Files

- <kernel file>.aoco
  - Intermediate object file representing the created hardware system for each kernel
- <kernel file>.aocr
  - Intermediate object file representing the created hardware system for the entire system
- <kernel file>.aocx
  - Kernel executable file used to program FPGA
- Inside <kernel file> folder
  - <kernel file folder>\reports\report.html
    - Interactive HTML report
    - Static report showing optimization, detailed area, and architectural information
  - <kernel file>.log compilation log
  - Intel® Quartus® Prime software generated source and report files

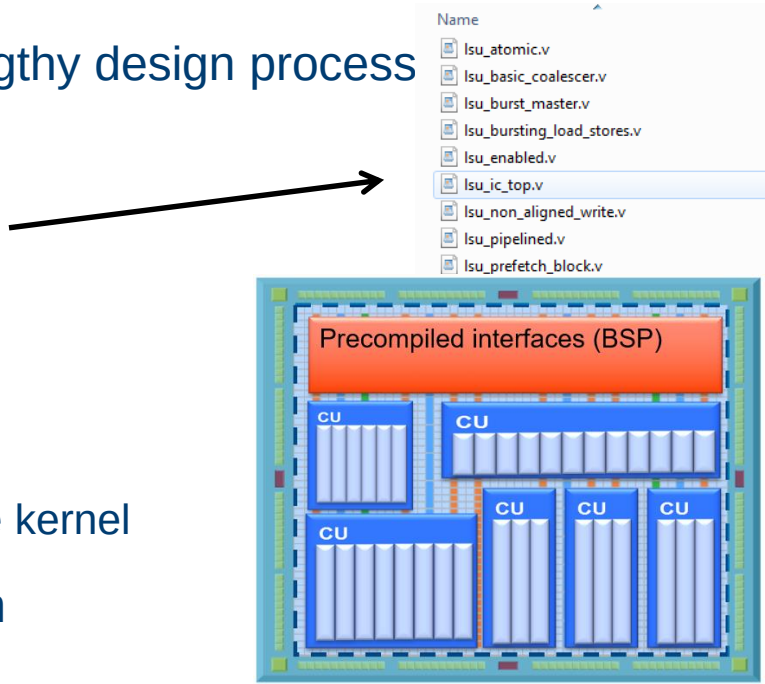
FPGA Architecture for OpenCL™ Implementation



OpenCL™ Kernels to Dataflow Circuits

Each kernel is converted into custom dataflow hardware (Compute Unit)

- Gain the benefits of FPGAs without the lengthy design process
- Implement C operators as circuits
  - HDL code located in <SDK Installation>/ip
  - Load Store units to read/write memory
  - Arithmetic units to perform calculations
  - Flow control units
  - Connect circuits according to data flow in the kernel
- May replicate circuit to accelerate algorithm



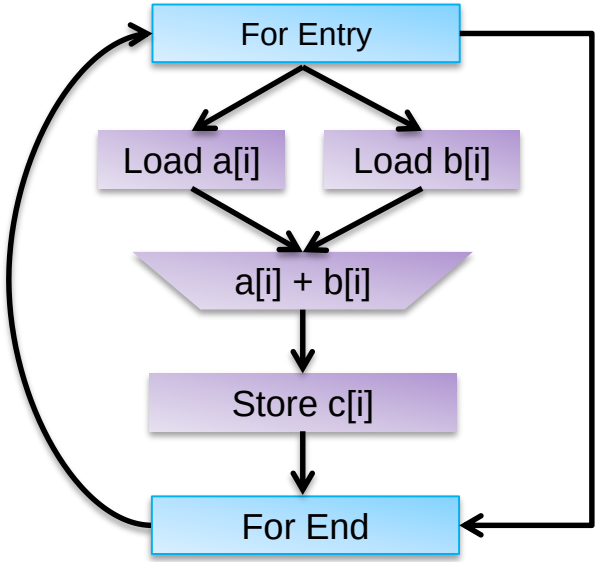
Compilation Example

Kernel compiled into dataflow circuit with flow control

- Includes branch and merge units

```
__kernel void my_kernel ( __global float *a,
                          __global float *b,
                          __global float *c,
                          int N)
{
    int i;
    for (i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

aoc



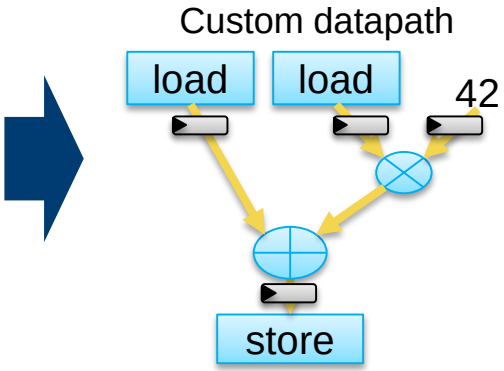
FPGA Custom Hardware

Custom Datapath on the FPGA Matches Your Algorithm!

- Creates typically very deeply pipelined version of a kernel
  - Huge number of operations simultaneously inflight
- Data can more easily be localized on chip

High-level code

```
Mem[100] += 42 * Mem[101]
```



Build exactly what you need:

Operations

Data widths

Memory size & configuration

Efficiency:

Throughput / Latency / Power

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

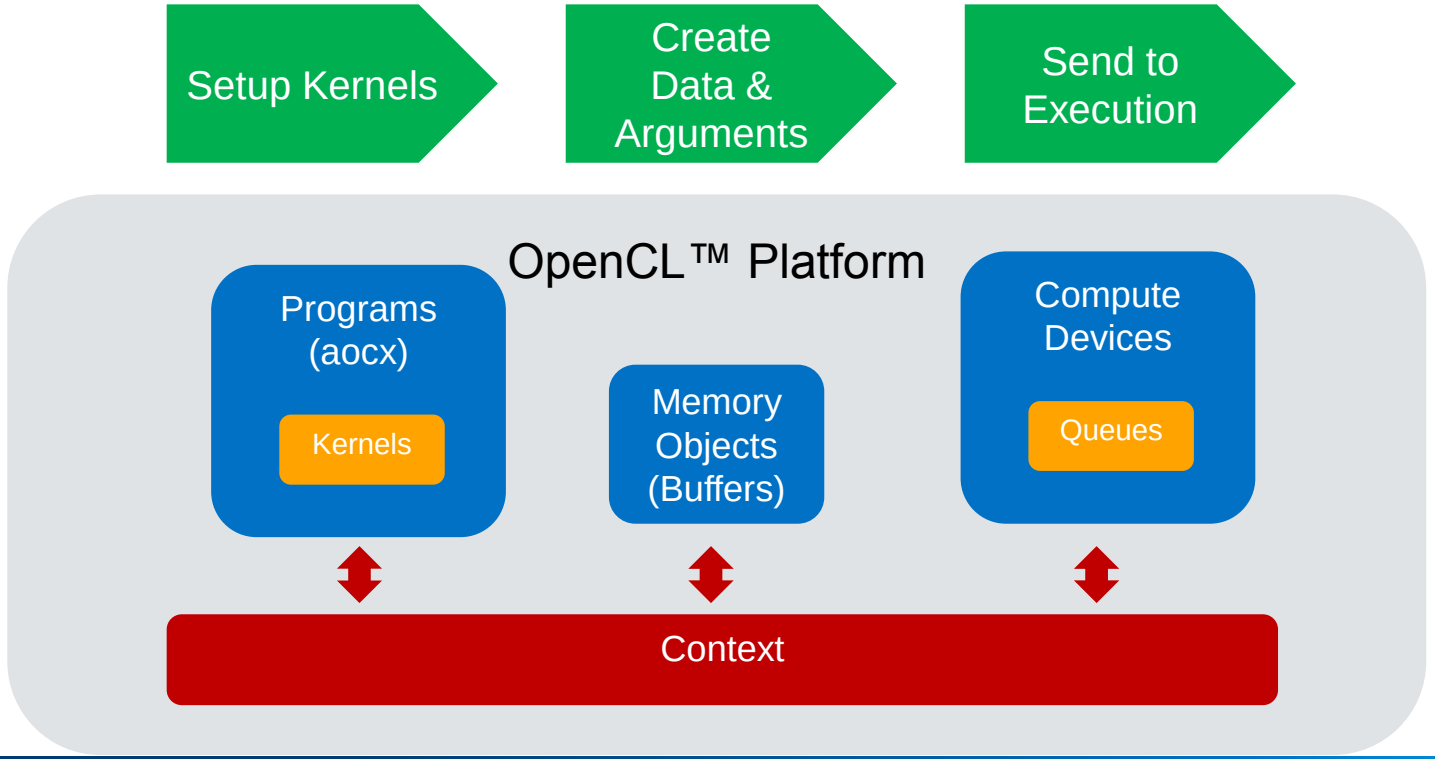
Executing OpenCL Kernels

- Writing kernels
- **Launching kernels**

NDRange Kernels

OpenCL on Intel® FPGAs

OpenCL™ Execution Flow



Programs and Kernels

Program – collection of kernels

- Process for host to execute a kernel on a 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 `enqueue…` methods

Code Example

```
__kernel void increment ( __global float *a, float c, int N)
{
     int i;
     for (i = 0; i < N; i++)
          a[i] = a[i] + c;
}
```

```
void main()
{
   cl::Context mycontext… // or cl_context mycontext
   std::vector<cl::Device> mydevlist // or 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 contains one or more kernels

- GPU/CPU vendors support creation of programs from source code
  - Online compilation of kernels (host runtime compilation)
  - Not supported by Intel® FPGA
- Intel FPGA only supports creation of programs from pre-compiled **binaries**
  - Binary implementation is vendor specific
  - **`aocx`** files supported
    - `aocx` is essentially the FPGA programming image

Creating Programs from Binary for FPGAs

Constructor

```
cl::Program::Program( const Context& context,
                      const VECTOR_CLASS<Device>& *devices,
                      const Binaries& binaries,
                      VECTOR_CLASS<cl_int> *binaryStatus=NULL,
                      cl_int *err=NULL)
```

Valid context

Compatible devices

aocx binary

Status of binary loading

Error code

- For Intel® FPGA
  - `Binaries` is `typedef VECTOR_CLASS<std::pair<const void*, ::size_t>>`
    - `binaries` should contain contents of the `aocx` file
  - When kernels from the `aocx` is run, the host will configure the FPGA with the `aocx`

Content

Length of content

Building Programs

Compiles and links a program executable from the program source or binary

- For Intel® FPGA, needs to be called to conform to the standards, but nothing meaningful done

```
cl_int cl::Program::build(const VECTOR_CLASS<Device> devices,…)
```

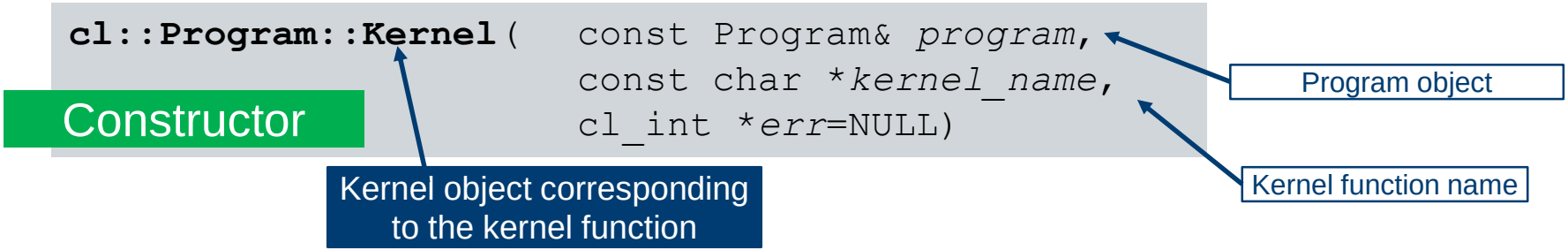
Create and Build Program - Code Example

```
void main()
{
...

//Create ifstream
std::ifstream aocx_stream("program.aocx", std::ios::in|std::ios::binary);

//Read file into string
std::string prog(std::istreambuf_iterator<char>(aocx_stream),
                 (std::istreambuf_iterator<char>()));

//Create ::Binaries which is just a vector of content, length pairings
cl::Program::Binaries mybinaries(1, std::make_pair(prog.c_str(), prog.length()+1));

// 1. Create then build the program
cl::Program program(mycontext, mydevlist, mybinaries);
err = program.build(mydevlist);
```

Creating Kernels

Create kernels from programs with **`clCreateKernel`**

- For Intel® FPGA, able to load any of the kernels compiled into the `aocx` file by the offline compiler



Creating Kernels – Code Example

```
__kernel void increment ( __global float *a, float c, int N)
{
      int i;
      for (i = 0; i < N; i++)
           a[i] = a[i] + c;
}
```

```
void main()
{

   ...
   // 1. Create then build the program
   ...
   // 2. Create kernels from the program
   cl::Kernel mykernel(program, "increment");

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

Set Kernel Arguments

Set the value for a specific argument of a kernel

```
cl_int cl::Kernel::setArg( cl_uint arg_index,
                           Type arg_value)
```

Argument index
From 0 (leftmost arg) to *N*-1 (*N* is the total number of args)

- Important to set the *arg_index* correctly
  - Hard to debug errors can result if a mistake is made

Setting Up Kernel Argument List - Code Example

```
__kernel void increment ( __global float *a, float c, int N)
{
      int i;
      for (i = 0; i < N; i++)
           a[i] = a[i] + c;
}
```

```
void main()
{
   ...
   //
   // 3. Allocate and write buffers to device
   cl::Buffer a_device(…);
   cl_float c_host = 10.8; …

   // 4. Set up the kernel argument list
   err = mykernel.setArg(0, a_device); // Set up 'a'
   err = mykernel.setArg(1, c_host); // Set up 'c'
   err = mykernel.setArg(2, NUM_ELEMENTS); // Set up 'N'

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

Execute Kernel

Use `enqueueNDRangeKernel` or `enqueueTask` to run kernel on device

```
cl_int cl::CommandQueue::enqueueTask(
                const Kernel& kernel,
                const VECTOR_CLASS<Event>* events=NULL,
                Event* event=NULL)
```

Error code

Where kernel will be queued for execution

Kernel to be executed

Events used for synchronization. Discussed later

- `enqueueNDRangeKernel` discussed later

Kernel Launch - Code Example

```
void main()
{
    // 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
    err=myqueue.enqueueTask(mykernel);

    // 6. Transfer result buffer back
}
```

Kernel Execution Complete Example

```
void main()
{   ...
   // 1. Create then build program
   c::Program myprogram = (mycontext, mydevlist, mybinaries);
   err = myprogram.build(mydevlist);

   // 2. Create kernels from the program
   cl::Kernel mykernel(myprogram, "increment", &err);

   // 3. Allocate and transfer buffers on/to device
   float* a_host = ...
   cl::Buffer a_device(mycontext, CL_MEM_READ_WRITE, nBytes);
   err=myqueue.enqueueWriteBuffer(a_device, CL_FALSE, 0, size, a_host);

   cl_float c_host = 10.8;

   // 4. Set up the kernel argument list
   err = mykernel.setArg(0, a_device);
   err = mykernel.setArg(1, c_host);
   err = mykernel.setArg(2, NUM_ELEMENTS);
```

Kernel Execution Complete Example Cont.

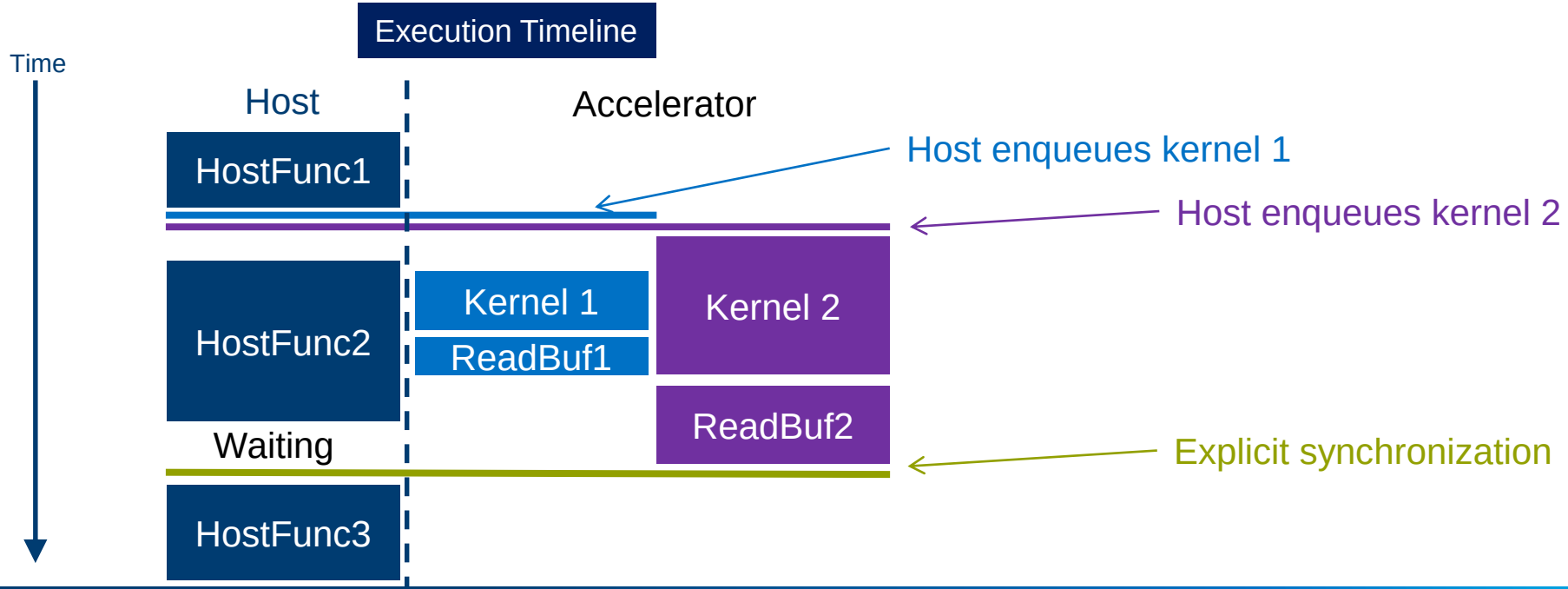
```
  …
  // 5. Launch the kernel
  err = myqueue.enqueueTask(mykernel);

  // 6. Transfer result buffer back
  err = myqueue.enqueueReadBuffer(a_device, CL_TRUE, 0, NUM_ELEMENTS*sizeof(cl_float), a_host);
}
```

Host and Kernel Execution

- Kernels execute on one or more OpenCL™ devices
- Host program executes on the host
- With `enqueue` commands, the host launches device tasks asynchronously
  - Control returns to host immediately
  - Unless explicit synchronization specified
- The host needs to manage synchronization among device tasks
  - In additional to memory management and error handling tasks

Asynchronous Kernel Execution

By default, host launches device but execution is not synchronized



Host-side Explicit Synchronization Point

- `cl::CommandQueue::finish` or `clFinish(queue)`
  - Blocks until all commands in a given queue have finished execution
- Events
  - Each `enqueue` task assigned an event id that can be used as a prerequisite for another `enqueue` 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 have finished executing

Event Dependencies

- Each `enqueue`
  - Can **depend on** an array/vector of (previously created) `Event`s
    - To ensures synchronization
  - Can **generate** a `Event` (to be used later)
  - The `enqueue` command itself does not block, just the execution of the associated task on the device

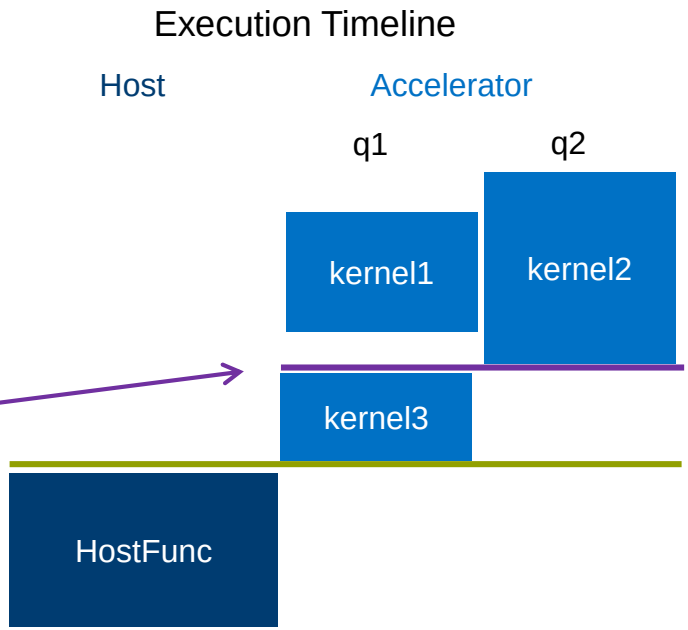
```
cl_int  cl::CommandQueue::enqueue… (…
                        const VECTOR_CLASS<Event>* events=NULL,
                        Event *event=NULL)
```

Wait for these to finish

Generate new event, to be used later. (If not NULL)

Synchronization Example

```
cl::CommandQueue q1(…);
cl::CommandQueue q2(…);
cl::Event e1, e2;

q1.enqueueTask(k1,…, &e1);

q2.enqueueTask(k2,…, &e2);

std::vector<cl::Event> elist;
elist.push_back(e1);
elist.push_back(e2);

q1.enqueueTask(k3,…,elist,NULL);

q1.finish();
q2.finish();

HostFunc();
```



Clean Up

- OpenCL™ C++ objects have destructors that are called by the host API when the object goes out of scope

Exercise 2

Writing a Simple Kernel

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

Executing OpenCL Kernels

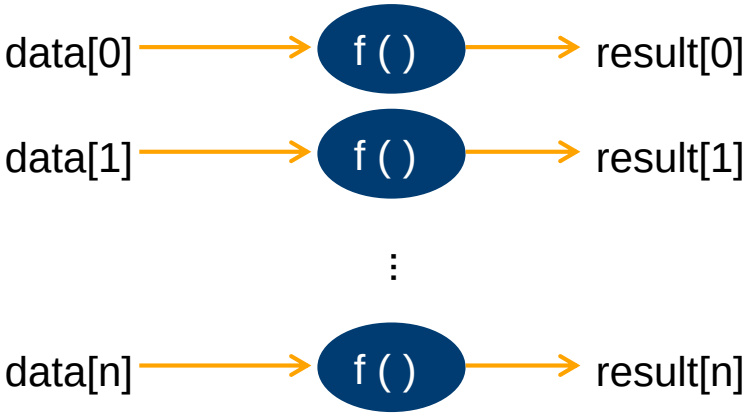
NDRange Kernels

- **Kernels and work-item hierarchy**
- Memory model

OpenCL on Intel® FPGAs

Data Parallelism (Review)

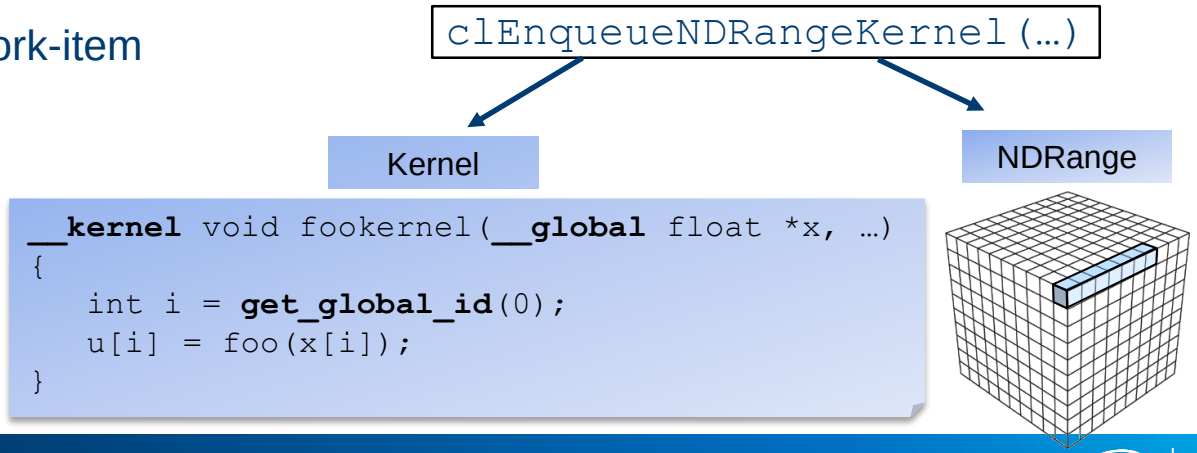
- Same operation applied to multiple, independent data concurrently
  - Data dependency hinders data parallelism



NDRange Kernels

Execute an OpenCL™ kernel across multiple data-parallel threads

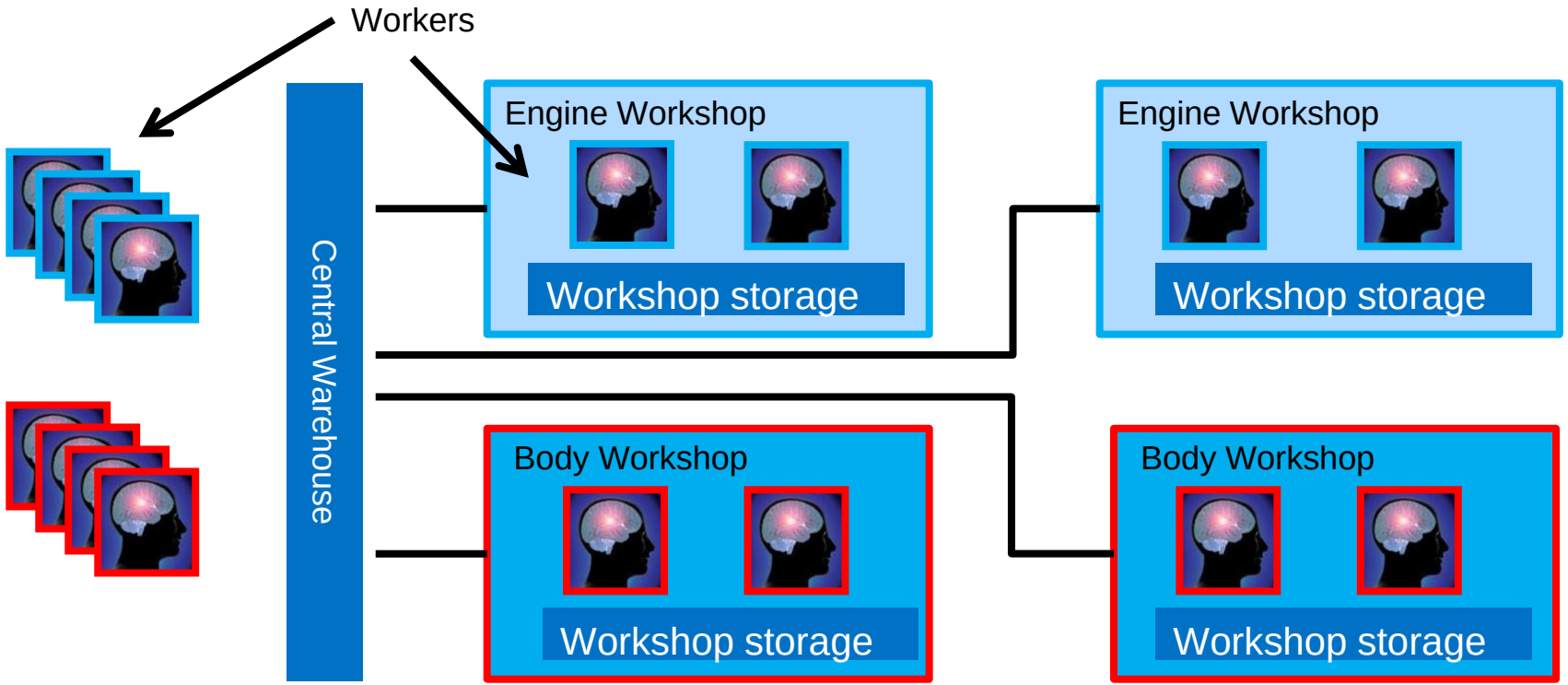
- "Traditional" OpenCL
- Executed in a single program (kernel) multiple data SPMD fashion
  - Explicitly declares data parallelism
  - Each thread called a work-item



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

- Global Dimensions: Total number of cars to build
- Work-item: Worker
- Workgroup (local dimensions): 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 with NDRange launches
- Represent parallelism at the finest granularity possible
- Same kernel executed by all the different data-parallel threads of a single launch

Work-item

Factory worker

Unit of concurrent execution in OpenCL™ standard

- A Thread
- 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

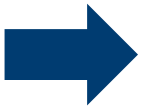
Kernel represents a single iteration of loop to perform vector operation

- N work-items will be generated to match array size
- `get_global_id(0)` function returns index of work-item which represent the loop counter

Vectored addition of A and B example

C

```
for (int i=0; i<N; i++)
{
    C[i] = A[i] + B[i];
}
```



OpenCL™ Kernel

```
// N work-items to be created
__kernel void vecadd(__global int *C,
                     __global int *A,
                     __global int *B)
{
    int tid = get_global_id(0);
    C[tid] = A[tid] + B[tid];
}
```

Global Dimensions

Total number of cars to build

- N-dimensional range of all threads that execute the kernel
- 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
  - `cl::NDRange(512, 1, 1);`
  - 2nd and 3rd dimensions can be omitted if size is 1 as in the case here
- Set at kernel launch time

Data Division Examples

- Audio
  - Use one-dimensional series of samples
    - Process each sample independently (e.g. volume change)
  - Global size = total number of samples
- Images
  - Maps well to two-dimensional data
  - Global dimensions = total number of pixels
- Physics Simulation
  - Use three-dimensional data for stress simulation to model behavior of materials
  - Global dimensions = representation of the 3D space

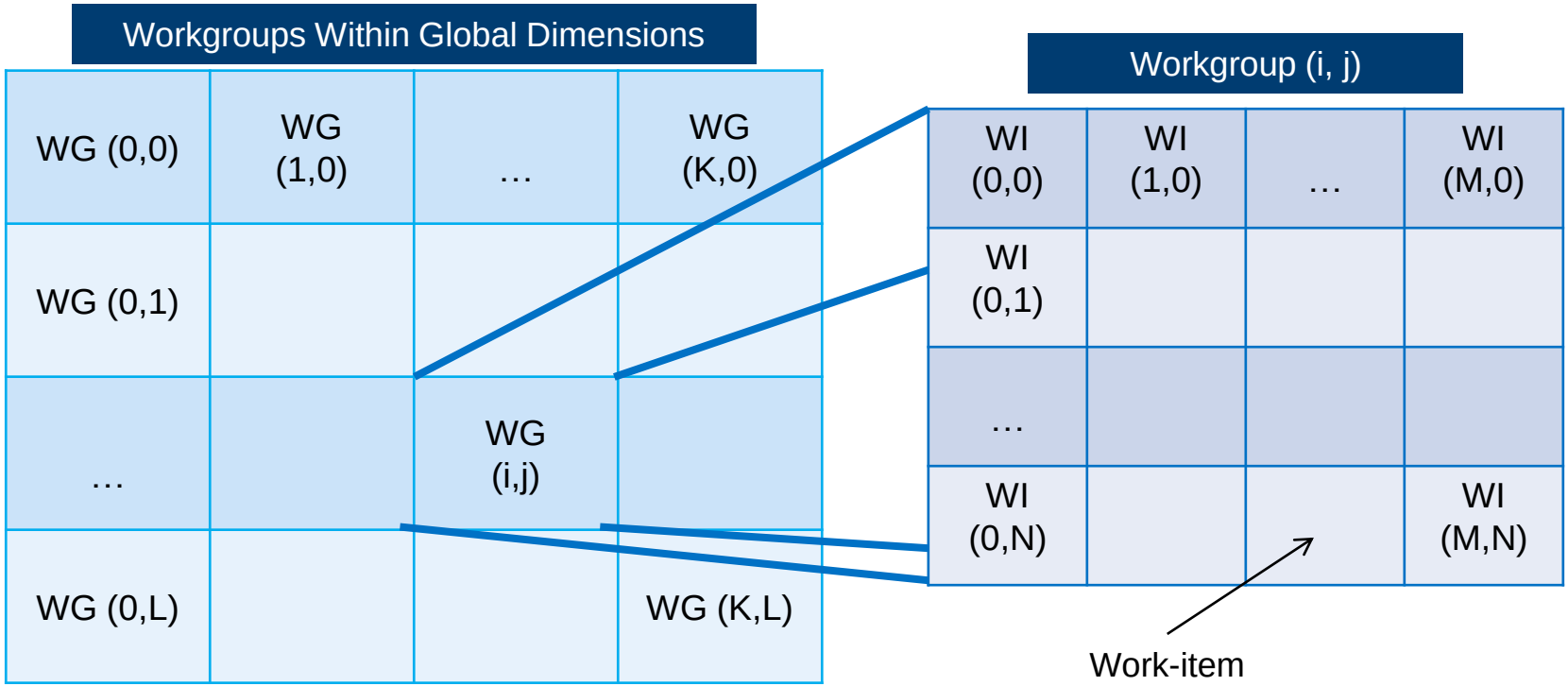
Workgroup

Group of workers able to communicate

Dividing work-items of global dimensions into smaller equally sized workgroups

- Local Dimensions
- Same dimension as NDRange index space
  - If we have 64 work-items per work group
    - `size_t workGroupSize[3] = {64, 1, 1}`
  - NDRange size must be **evenly divisible** by workgroup size in each dimension
  - Set at kernel launch time
- Synchronization among work-items possible only within workgroups
- Optimal workgroup size usually determined by hardware

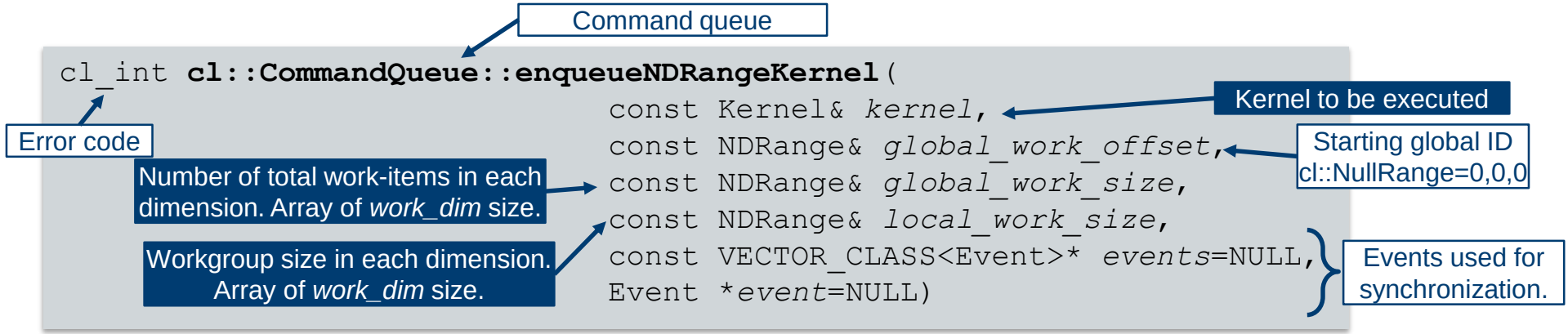
Work-Item Hierarchy (2D Example)



Total number of work-items = (N+1) * (M+1) * (K+1) * (L+1)

NDRange Launch

Use `enqueueNDRangeKernel` to launch a kernel with multiple work-items



Total work-items = *global_work_size*[0] * *global_work_size*[1] * *global_work_size*[2]

Kernel Launch - Code Example

```
//1D Work-Group Example
int err;
size_t const globalWorkSize = 1920;
size_t const localWorkSize = 8;
err=myqueue.enqueueNDRangeKernel(1dkernel, cl::NullRange, cl::NDRange(globalWorkSize),
                                 cl::NDRange(localWorkSize));


//3D C Work-Group Example
err=clEnqueueNDRangeKernel(myqueue, 3dkernel, 3, NULL, NDRange(512,512,512),
                                 NDRange(16,8,2),0, NULL, NULL);
```

Identifying NDRange Launch Properties in Kernels

- OpenCL™ kernels have functions to query the NDRange properties determined at kernel launch time
  - Most take the dimension as an `uint` argument (0-2)
  - `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 In the Kernel

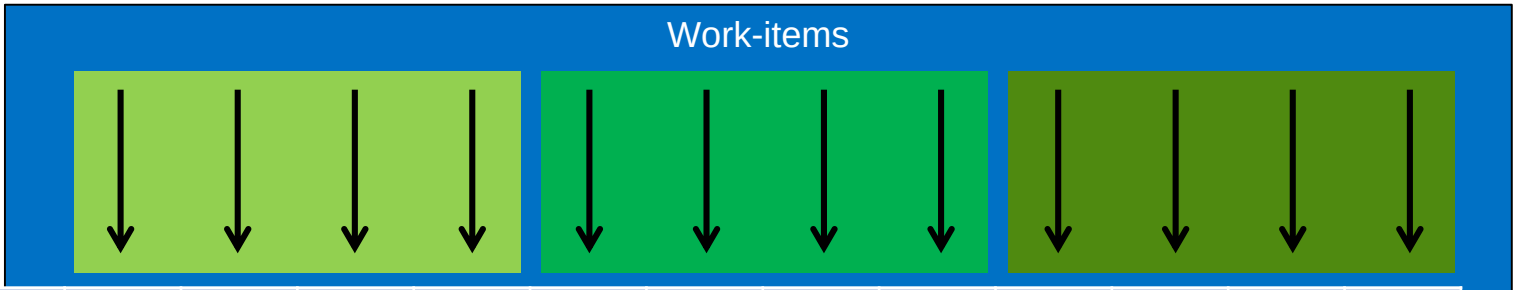
- OpenCL™ kernels have functions to identify the current work-item executing the kernel
  - Take the dimension as an `uint` argument (0-2)
  - Often used to dereference data pointers
  - `get_global_id(dim)`
    - Index of work-item in the global space
  - `get_local_id(dim)`
    - Index of work-item within workgroup
  - `get_group_id(dim)`
    - Index of current workgroup

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

Work-item Identification Example

One-dimension launch

get_work_dim()=1, get_ global_size(0)=12,
get_local_size(0)=4, get_num_groups(0)=3



get_global_id(0)	0	1	2	3	4	5	6	7	8	9	10	11
get_local_id(0)	0	1	2	3	0	1	2	3	0	1	2	3
get_group_id(0)	0	0	0	0	1	1	1	1	2	2	2	2

Work-Item Identification Continued

One-dimension launch

get_work_dim()=1, get_ global_size(0)=12,
get_local_size(0)=4, get_num_groups(0)=3

```
__kernel void MyKernel(…) {
      int i = get_global_id(0);
      x[i] = 8;
}

__kernel void MyKernel(…) {
      int i = get_global_id(0);
      x[i] = get_group_id(0);
}

__kernel void MyKernel(…) {
      int i = get_global_id(0);
      x[i] = get_local_id(0);
}

__kernel void MyKernel(…) {
      int i = get_global_id(0);
      x[i] = get_global_id(0);
}
```

Results

```
x[0-11]: 8 8 8 8 8 8 8 8 8 8 8 8
```

```
x[0-11]: 0 0 0 0 1 1 1 1 2 2 2 2
```

```
x[0-11]: 0 1 2 3 0 1 2 3 0 1 2 3
```

```
x[0-11]: 0 1 2 3 4 5 6 7 8 9 10 11
```

Mapping NDRange Kernels to FPGAs (Naïve)

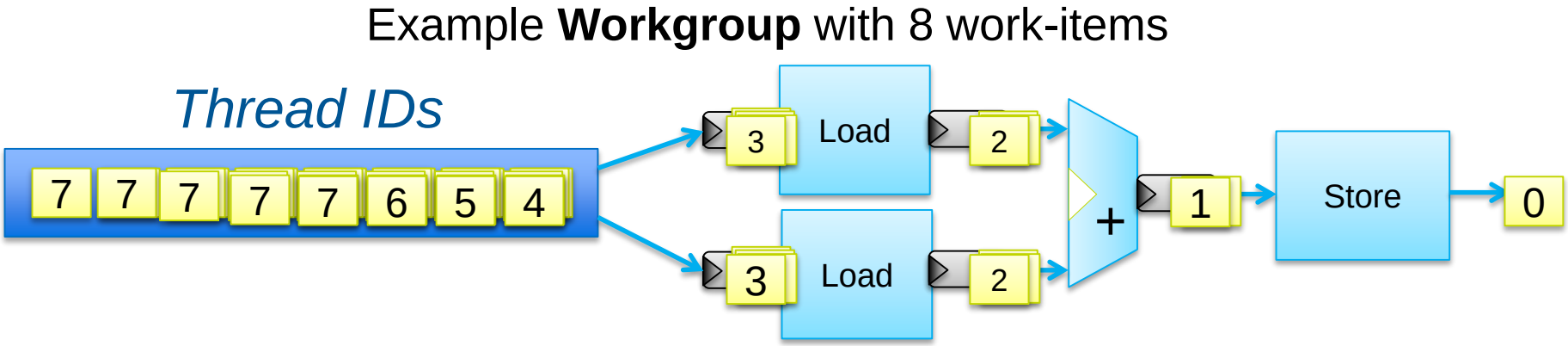
- ~~Simplest way of mapping kernel functions to FPGAs may appear to be replicating hardware for each work-item (thread)~~
- Problems:
  - Global size tends to be really large
    - Can we implement millions of kernel pipelines at once on an FPGA?
  - Inefficient and wasteful
    - FPGA compute bandwidth is often NOT the bottleneck of system
    - Difficult to keep all the stages of all the pipelines busy
  - Unknown at kernel compile time the number of work-items to run
- AOC may optionally replicate HW to process multiple work-items in parallel
  - See the best practices guide

Mapping NDRange Kernels to FPGAs

- Better method involves taking advantage of pipeline parallelism
  - Attempt to create a deeply pipelined representation of a kernel
  - On each clock cycle, we attempt to send in input data for a new thread
  - Map coarse grained thread parallelism to fine-grained FPGA parallelism
  - A typical kernel pipeline will consist of **hundreds** of stages
    - Hundreds of work-items executing concurrently in pipelined fashion
  - No need to worry about atomic operations since threads are not exactly in parallel

Example Pipeline for Vector Add

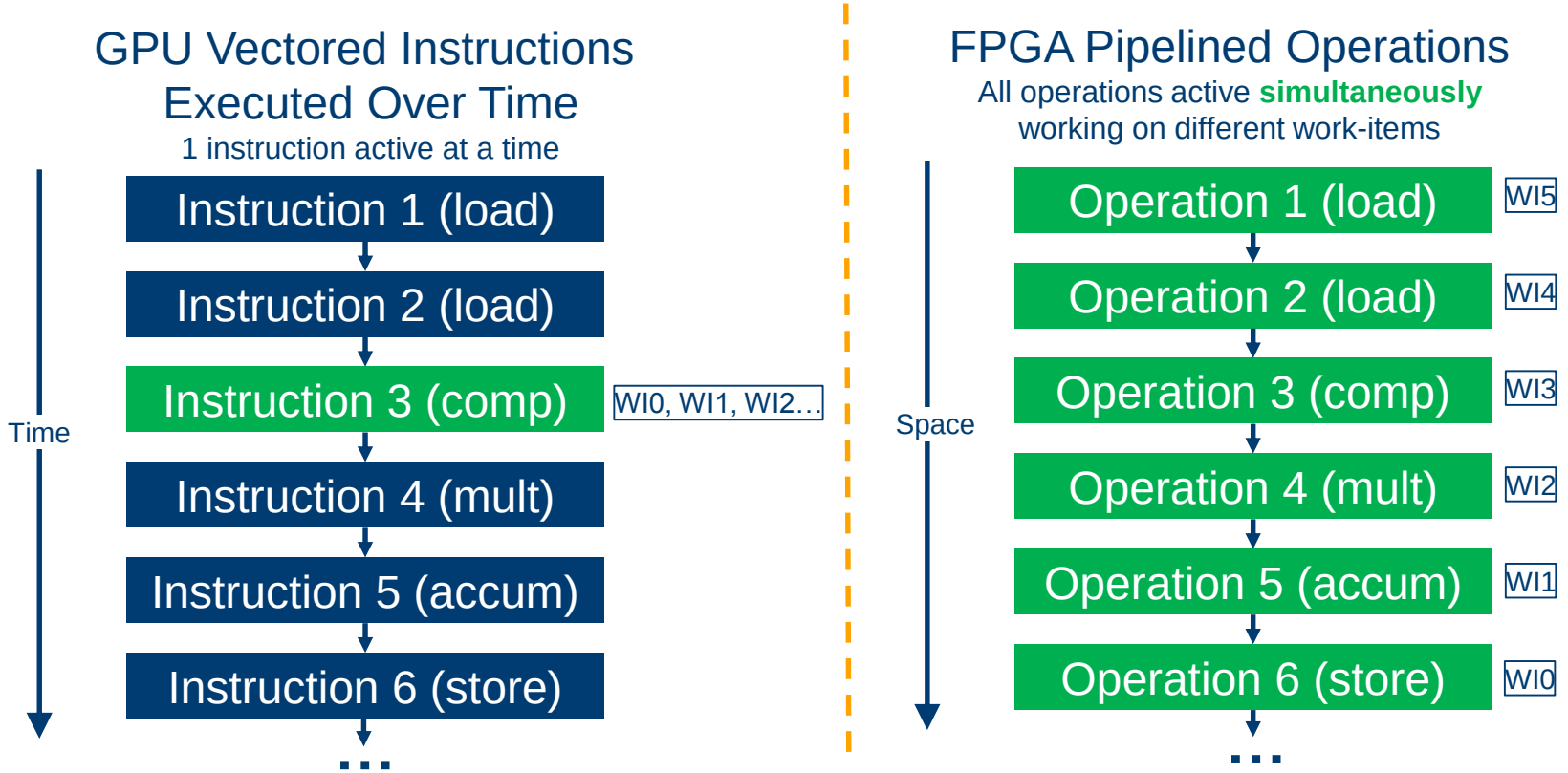
- On each cycle the portions of the pipeline are processing different threads
- While work-item 2 is being loaded, work-item 1 is being added, and work-item 0 is being stored



Mapping of Multi-threaded Kernels Comparison

- Pipelined parallelism achieved from FPGA kernel pipeline implementation
  - 1 work-item / clock cycle throughput
    - Default implementation assuming no other bottlenecks
    - Regardless of kernel complexity
  - Can optionally enable vectorization or compute unit replication if appropriate
- Array processing with GPUs
  - A workgroup of work-items processed together over a large number of cycles
    - Kernel throughput slows with increase in kernel complexity

GPUs vs FPGA Execution



Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

Executing OpenCL Kernels

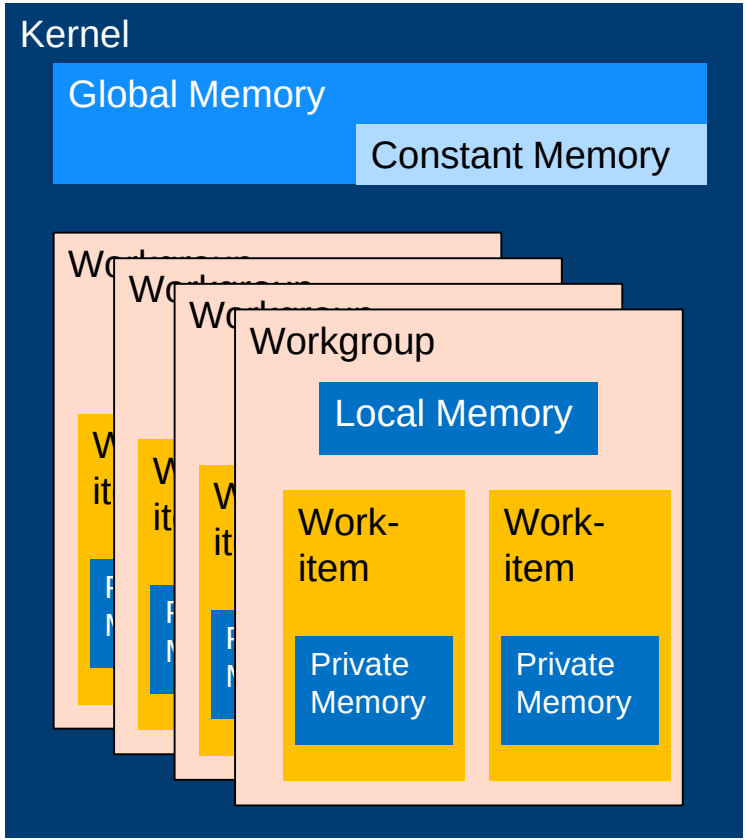
NDRange Kernels

- Kernels and work-item hierarchy
- Memory model

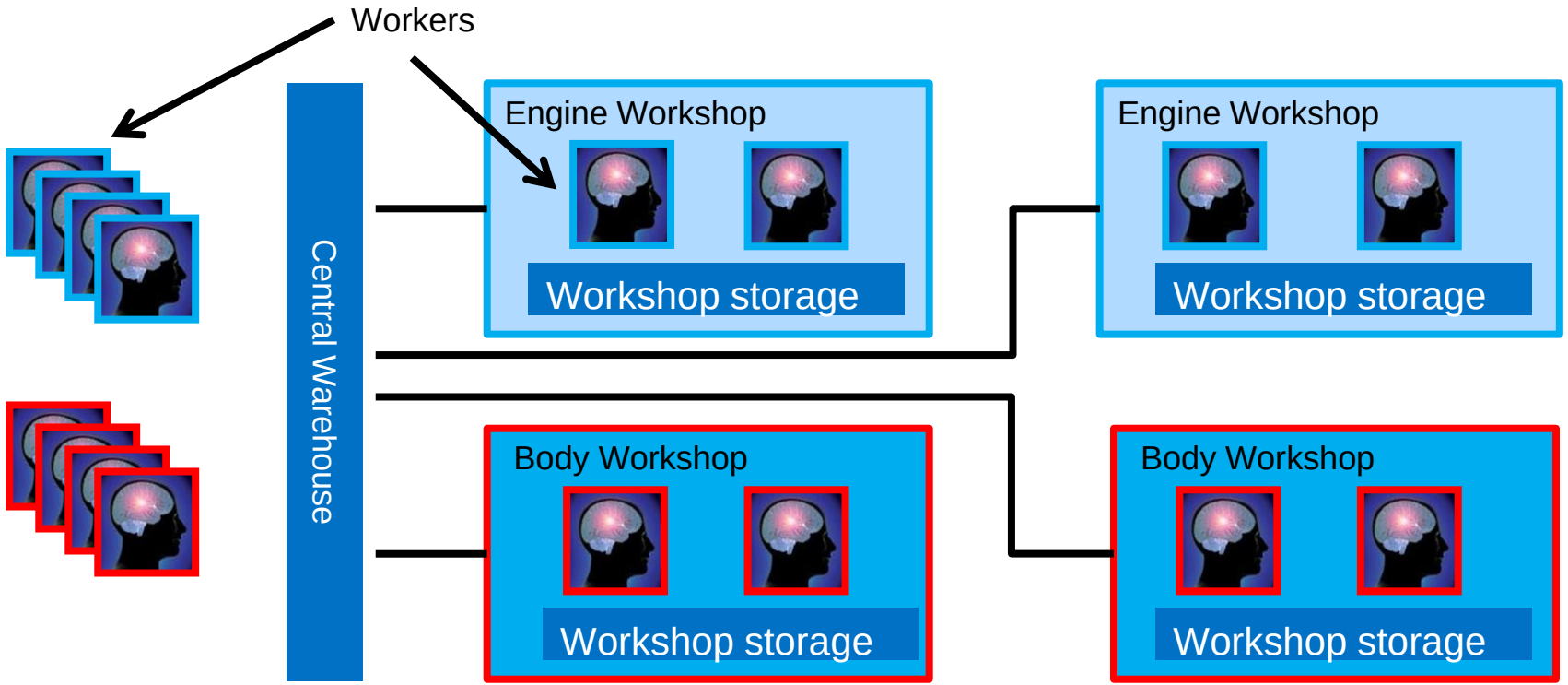
OpenCL on Intel® FPGAs

OpenCL™ Memory Model

- Private Memory
  - Unique to work-item
- Local Memory
  - Shared within workgroup
- Global/Constant Memory
  - Visible to all workgroups
- Host Memory
  - Visible to the host CPU
  - May be shared with device in unique cases



Car Factory Analogy Revisited



Car Factory Analogy Explained

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

Mapping OpenCL™ Memory to FPGAs

- Global
  - Off-chip memory (DDR SDRAM, QDR SRAM, HMC) defined by the BSP
- Constant
  - Resides in off-chip memory and accessed through cache shared by all kernels
- Local
  - On-chip memory
    - Custom memory system built for each variable
    - Much higher bandwidth and lower latency than global memory
- Private
  - On-chip registers or block RAM depending on usage

Kernel Pointer Qualifiers

- All pointers in kernels needs to be qualified to indicate memory address space
  - `__global`, `__local`, `__constant`, or `__private`
  - `__private` is the default
- Pointers in kernel arguments can be `__global`, `__constant`, or `__local`
  - Allows host to allocate and read/write to `__global` and `__constant` memory
  - Allows host to allocate `__local` memory
- Non-pointer data types are always private
- The only type of program scope (static) variables are `__constant` pointers

Data Sharing and Synchronization

Data Sharing and Synchronization only possible among work-items of the **same work-group**

- Use **barriers** to synchronize
  - Ensures all work-items within a work-group must execute the barrier function before any work-item can continue
  - Implies **memory fence** that provides ordering between memory operations
  - Ensures consistency of memory at the barrier

Memory Qualifier Syntax Example

Increase performance by using local memory to cache data

```
__kernel void MyKernel(__global float* g_data) {
   l_Index=get_local_id(0);
   g_Index=get_global_id(0);

   //Shared by all work-items in the workgroup
   __local float l_Data[256];

   //Cache in l_data for current work-item
   l_Data[l_Index] = g_data[g_Index];

   barrier(CLK_LOCAL_MEM_FENCE);
   //All elements of  l_Data available
   process_data(l_data[l_Index], l_data[l_Index+1], …)
   …
}
```

Exercise 3

Writing a Simple NDRange Kernel

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

Executing OpenCL Kernels

NDRange Kernels

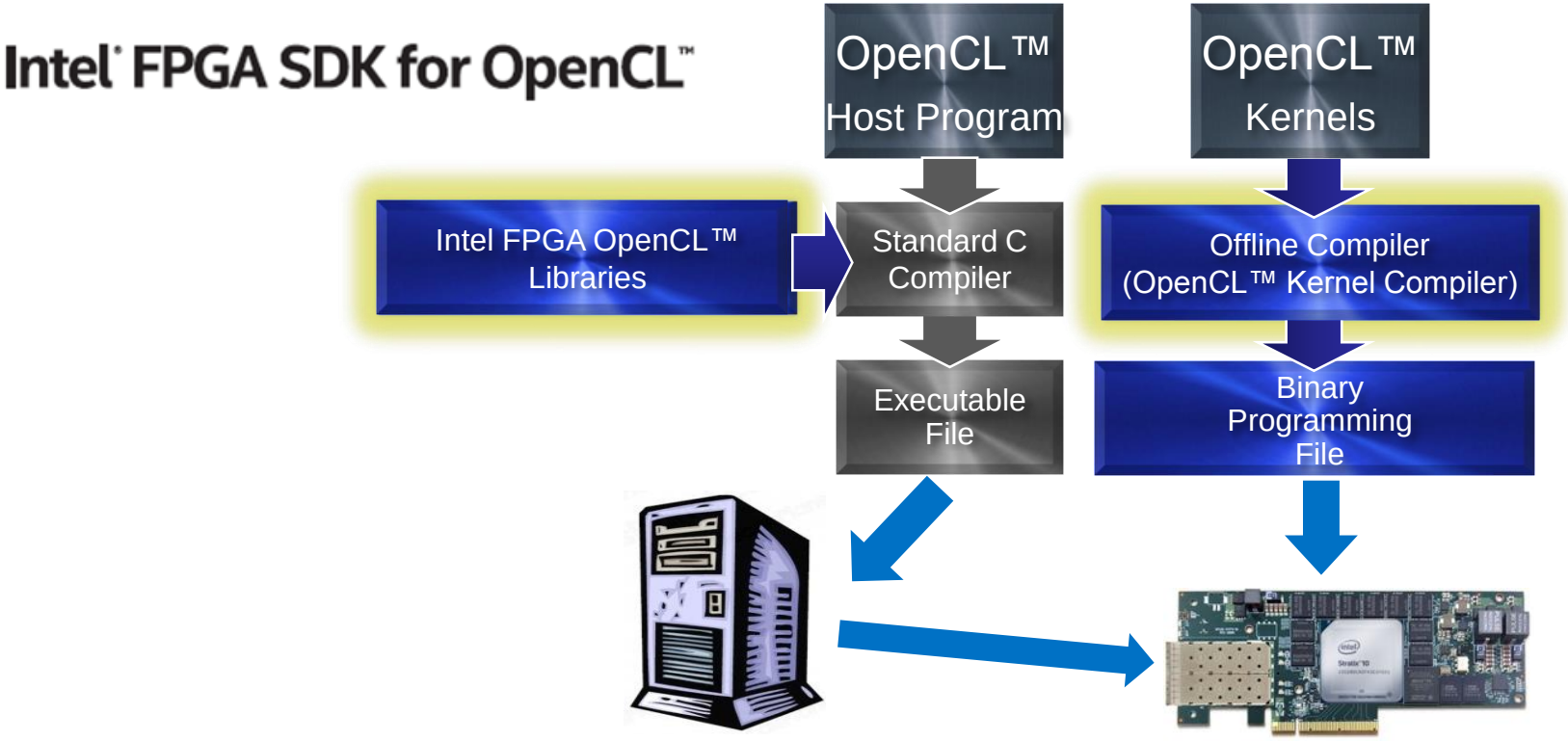
OpenCL on Intel® FPGAs

- The Intel FPGA SDK for OpenCL
- Debug Tools
- FPGA-specific Features

Intel® FPGA SDK for OpenCL™ Agenda

- SDK Content
- Kernel Compilation
- Host Compilation
- AOCL Utility
- Runtime

Intel® FPGA SDK Overview



SDK Components

- Offline Compiler (AOC)
  - Translates your OpenCL™ C kernel source file into an Intel® FPGA hardware image
- Host Libraries
  - Provides the OpenCL host API to be used by OpenCL host applications
- AOCL Utility
  - Perform various tasks related to the board, drivers, and compile process
- Intel® Code Builder for OpenCL API with FPGA kernel development framework
  - Provides Microsoft* Visual Studio or Eclipse-based IDE for code development

Software Requirements

To compile and run the host program and OpenCL™ kernels

- Intel® Quartus® Prime software with the appropriate devices
- Generic C compiler for the host program

To use Intel® Code Builder for OpenCL™ API

- Microsoft* Visual Studio or Eclipse

Intel® FPGA SDK for OpenCL™ Directory Structure

Directory	Description
bin	Main compiler and utility executables
windows64/bin linux64/bin	Runtime DLLs and other executables This should be in your path.
board	Design files related to specific supported boards
ip	IP cores required for kernel compilation
host	Files used by the compilation flow for user programs.
host/include	OpenCL™ API header files, and the interface files used to compile and link a user host program. Add this directory to the include file search path when compiling an OpenCL host program.
host/windows64/lib host/linux64/lib host/arm32/lib	The OpenCL host runtime libraries. Add this directory to the library file search path when linking an OpenCL host program.

Offline Kernel Compiler (aoc)

```
aoc –board=<my board> <my kernel file>
```

Option	Description
`-help or -h`	Help for the tool
`-c`	Creates .aoco object file and sets up a Quartus® Prime hardware design project
`-rtl`	Creates .aocr file that links all of the .aoco files
`-board=<board name>`	Compile for the specified board
`-list-boards`	Prints a list of available boards

- Compiles kernels for a specific board defined by a board support package
- Generates aoco, aocr, and aocx files
- For detailed info on supported kernel constructs see the Intel® FPGA SDK for OpenCL™ programming Guide

There are many other debugging, optimization, and build options.

Intel FPGA Preferred Board for OpenCL

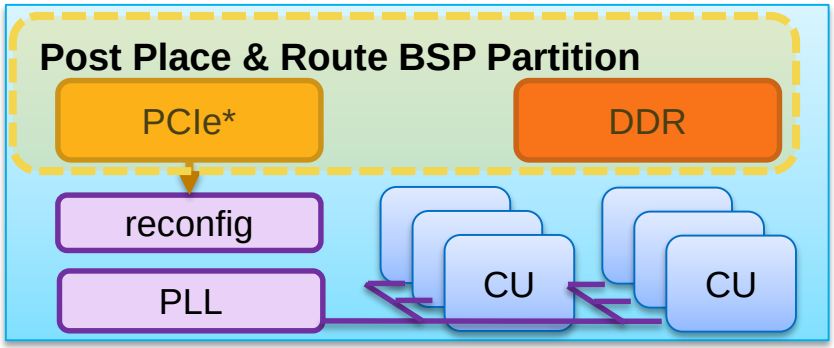
- Intel® FPGA Preferred Board for OpenCL™
  - Available for purchase from preferred partners
  - Passes conformance testing
- Download and install Intel FPGA OpenCL compatible BSP from vendor
  - Supplies board information required by the offline compiler
  - Provides software layer necessary to interact with the host code including drivers

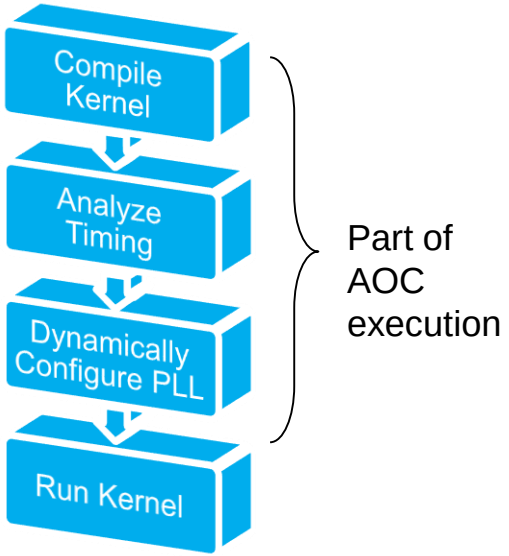
Nallatech GIDEL terasIC REFLEX CES Custom Embedded Systems BittWare picocomputing




Guaranteed Timing Closure

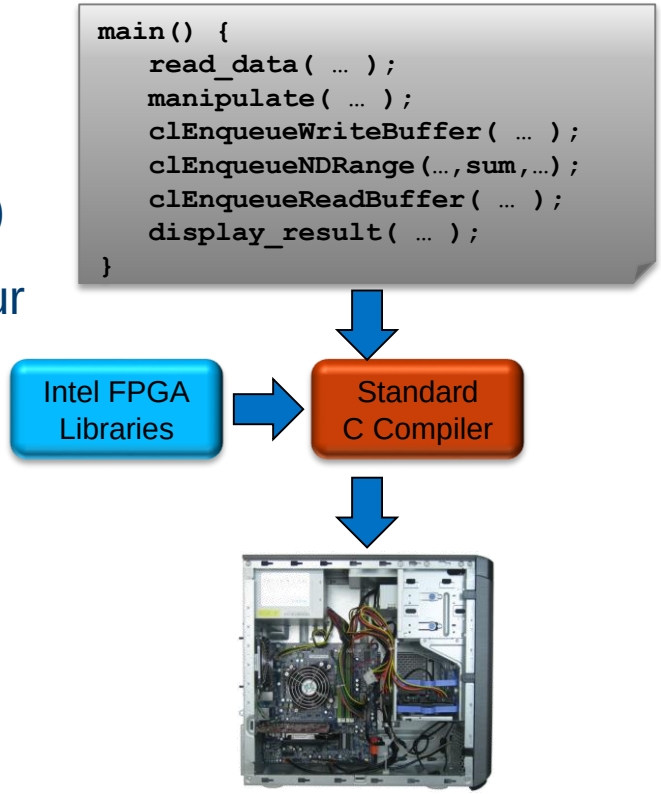
- Interfaces have BSP defined clock frequencies
  - PCIe* e.g. 250 MHz
  - DDR e.g. 800 MHz
- Kernel compute unit frequency determined by AOC
  - <= BSP-specified reference frequency





Compiling the Host Program

- Include `CL/opencl.h` or `CL/cl.hpp`
- Use a conventional C compiler (Visual Studio*/GCC)
- Add `$INTELFPGASDKROOT/host/include` to your file search path
  - Recommended to use `aocl compile-config`
- Link to Intel® FPGA OpenCL™ libraries
  - Link to libraries located in the `$INTELFPGASDKROOT/host/<OS>/lib` directory
    - Recommended to use `aocl link-config`



AOCL Utility

Host Compilation Commands (Use in your makefile)	
`aocl compile-config`	Displays the compiler flags for compiling your host program
`aocl link-config`	Shows the link options needed by the host program to link with libraries
`aocl makefile`	Shows example Makefile fragments for compiling and linking a host program

Board Management Commands (Functionality Provided by BSP)	
`aocl install`	Installs a board driver onto your host system
`aocl diagnose`	Runs the board vendor's test program
`aocl flash <.aocx>`	Programs the on-board flash with the FPGA image over JTAG

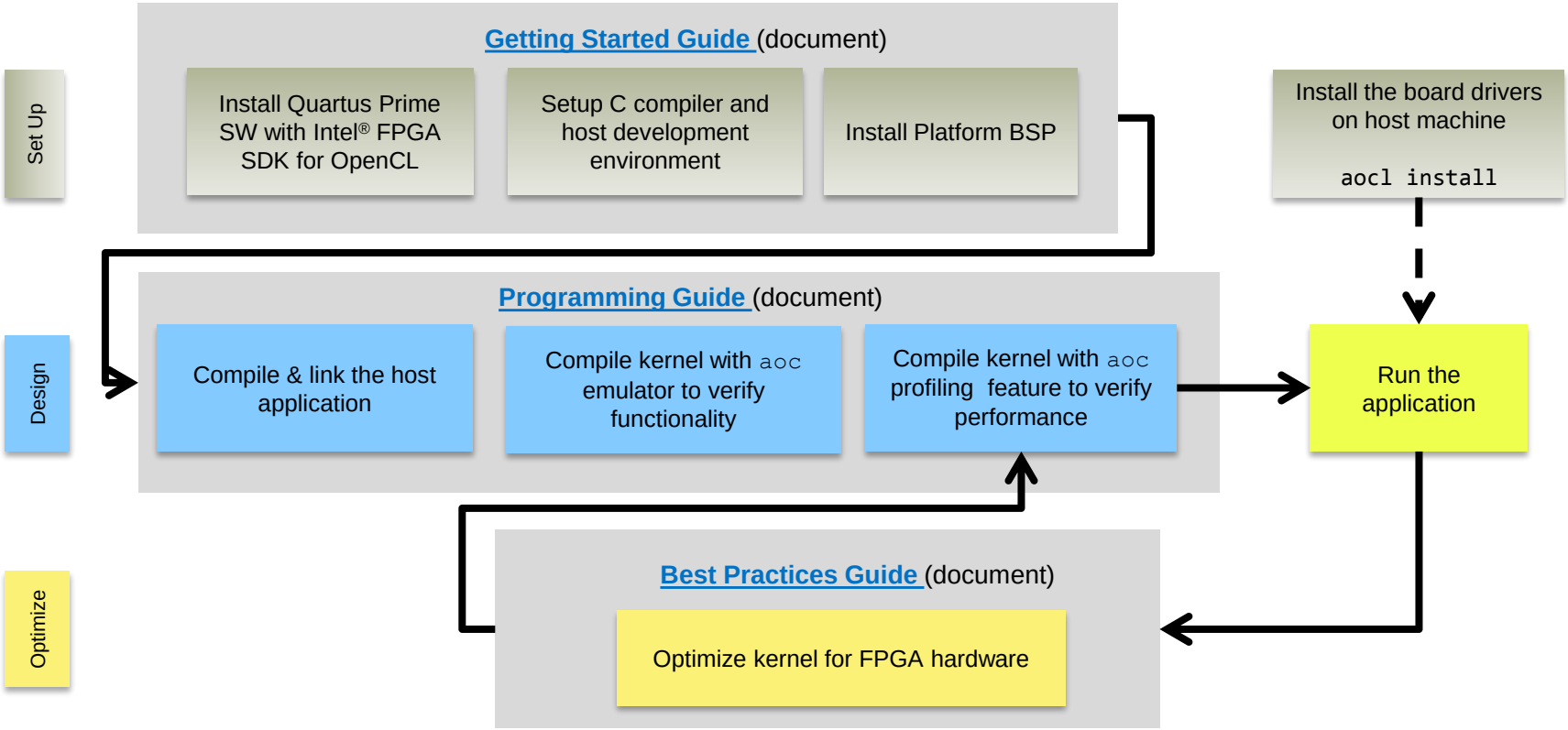
View Kernel Compilation Report	
`aocl report`	Displays kernel execution profiler data

Run `aocl help` or `aocl help <subcommand>` for detailed information about the tool

Host to FPGA Programming

- A valid OpenCL™ compatible image must be configured on the FPGA prior to host application execution
  - To establish Host-Device communication
  - Host may overwrite the core of the FPGA with new kernel circuit
- `aocl program <board instance> <BSP compatible image>.aocx`
  - boardtest.aocx included with the BSPs can be used as the image
  - Programs FPGA directly, usually across PCIe* interface or JTAG download cable
- `aocl flash <board instance> <BSP compatible image>.aocx`
  - Programs the user region of the flash on the board
  - FPGA loads from flash upon power-up

Intel® FPGA SDK for OpenCL™ Design Flow



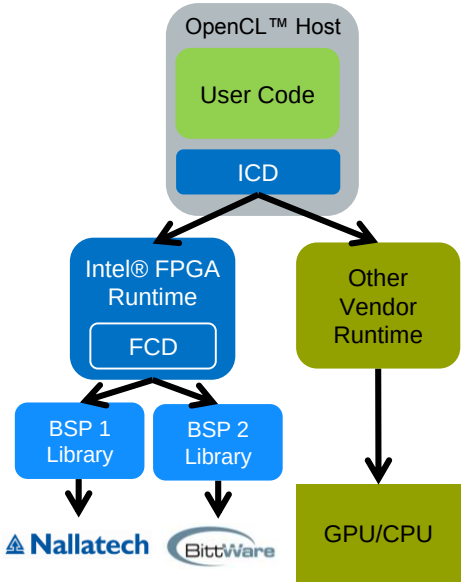
Host Execution

Prior to executing the compiled host program:

- Install Intel® FPGA Runtime Environment for OpenCL™
  - If host is not the kernel development system
- Ensure the following directories are part of the PATH
  - `$INTELFPGAOCLSDKROOT/bin`
  - `$INTELFPGAOCLSDKROOT/<OS>/bin`
  - `$INTELFPGAOCLSDKROOT/host/<OS>/bin`
- Run `aocl install` installs the drivers needed for the current board package

FCD / ICD Support

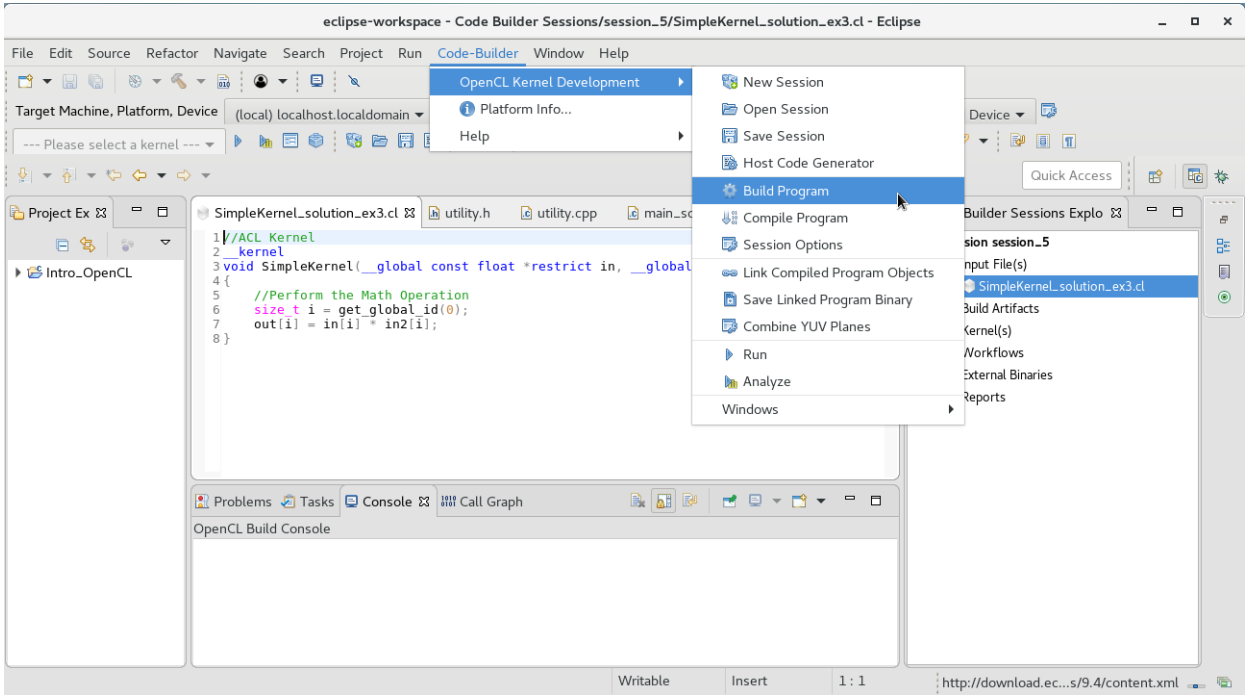
- Installable Client Driver (ICD)
  - Allows multiple vendor runtime libraries to be dynamically loaded by a single host
- FPGA Client Driver (FCD)
  - Allows multiple BSP runtime libraries to be dynamically loaded by a single host



Intel® Code Builder for OpenCL™ API Now with FPGA Support

Plugin for Eclipse or Microsoft* Visual Studio*

- Compile kernel within Eclipse
- Target FPGA or other Intel device
- Syntax highlighting



Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

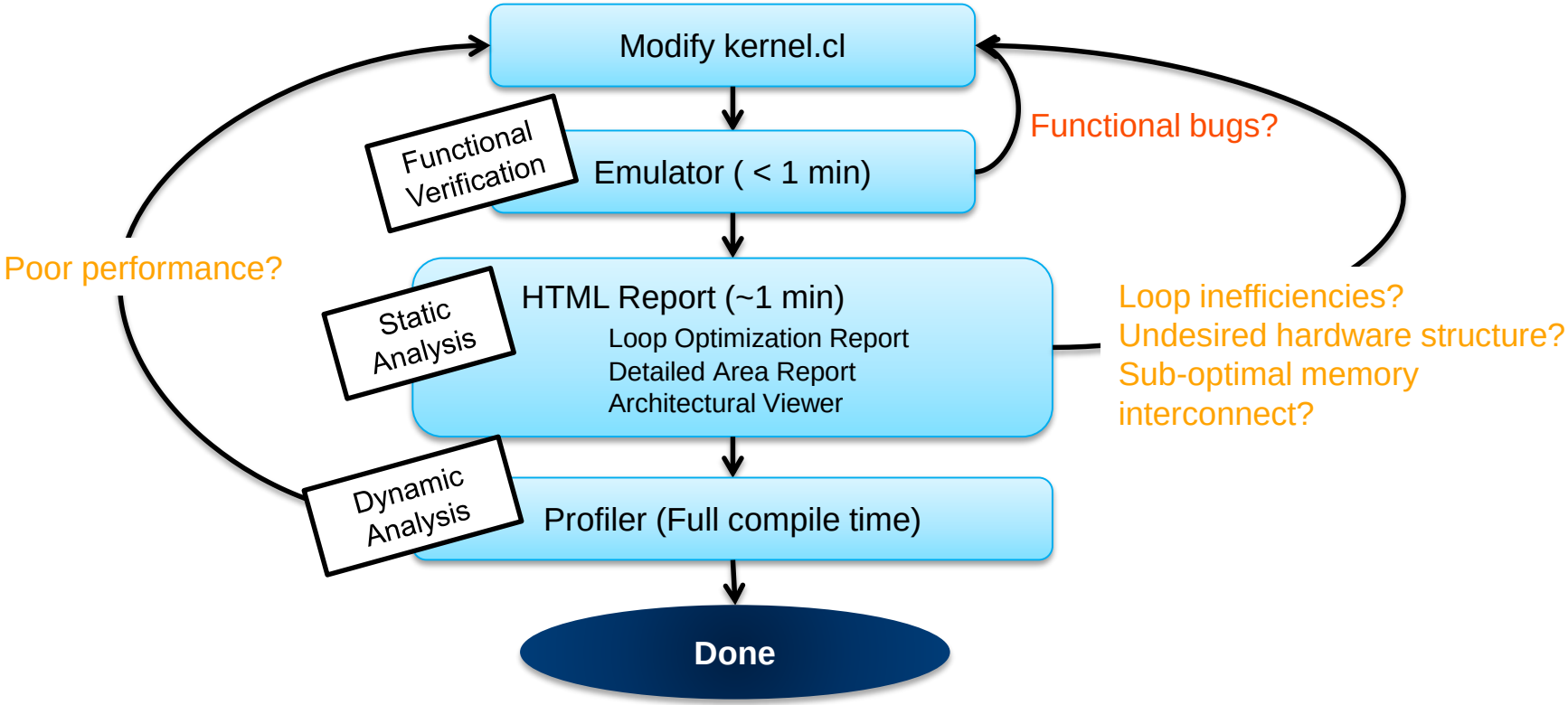
Executing OpenCL Kernels

NDRange Kernels

OpenCL on Intel® FPGAs

- The Intel FPGA SDK for OpenCL
- Debug Tools
- FPGA-specific Features

Kernel Development Flow and Tools



Emulator

Enable kernel functional debug on x86 systems

- Quickly generate x86 executables that represent the kernel

```
aoc –march=emulator <kernel file>
```

```
kernel void accel(…) {
  …
  gid = get_global_id(0);
  out[gid]=proc(data[gid]);
  …
}
```

```
./kernel_tb…
…
Running …
```

- Debug support for
  - Standard OpenCL™ syntax, Channels, Printf statements

Emulating an OpenCL™ Kernel Steps

1. Generate the .aocx file with `aoc -march=emulator`
   - Make sure the right board is used
2. Compile and link the host
3. Set Emulator Environment for the number of boards

```
set CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=<num_devices>
```

4. Run the host program

```
c:\opencl>aoc -march=emulator conv.cl
c:\opencl>dir
host.exe  conv.cl conv.aocx
c:\opencl>host.exe
running…
Done!
```

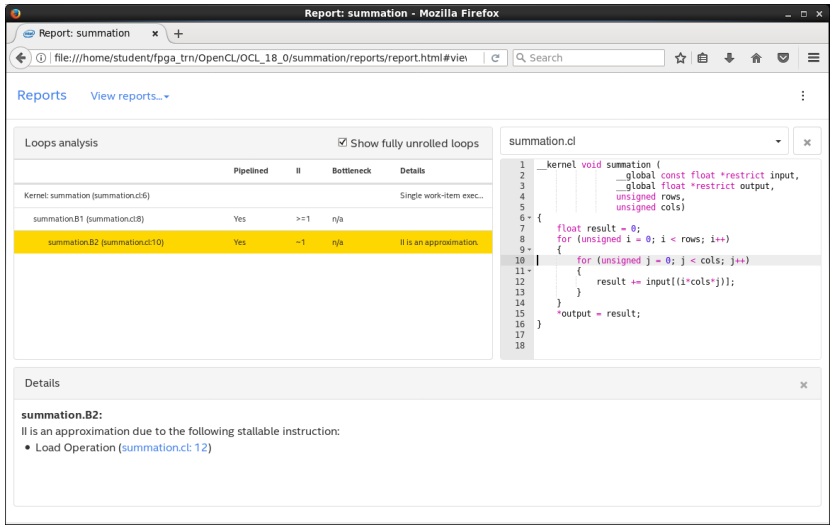
HTML Report

Static report showing optimization, area, and architectural information

- Automatically generated with the object file (`aoc -rtl`)
  - Located in **`<kernel file folder>\reports\report.html`**
- Dynamic reference information to original source code
- Loop Analysis Optimization report
  - Information on how loops are implemented
- Area report
  - Detailed FPGA resource utilization by source code or system block
- Architectural viewer
  - Memory access implementation and kernel pipeline information

HTML Loop Analysis Optimization Report

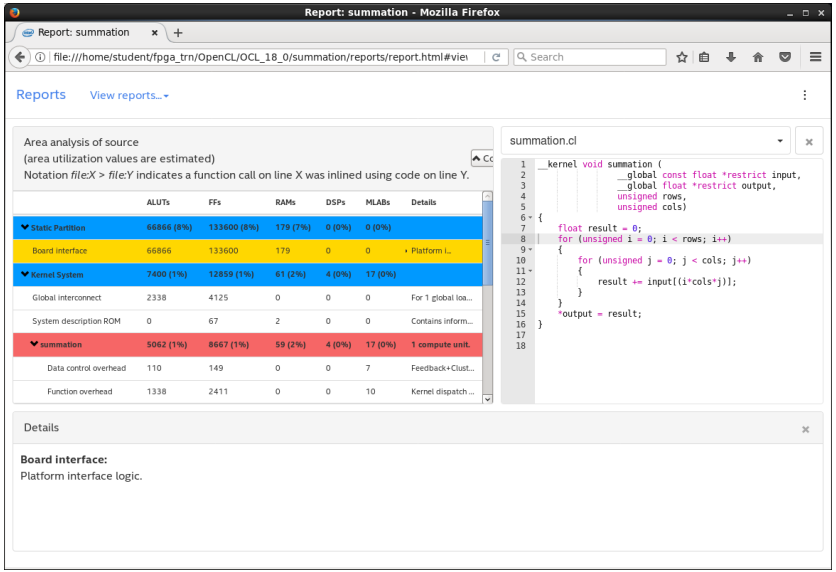
- Actionable feedback on pipeline status of loops
  - Shows loop carried dependencies and bottlenecks
  - Especially important for single work-item kernels since they have an outer loop
- Shows loop unrolling status
- Shows loop nesting relationship



HTML Area Report

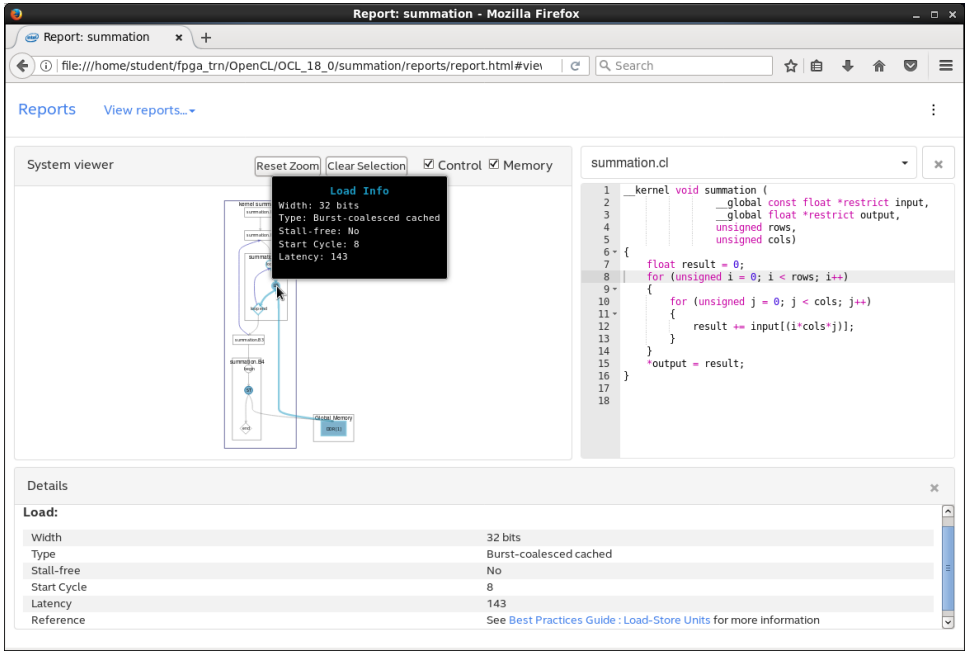
Generate detailed estimated area utilization report of kernel code

- Detailed breakdown of resources by source line or by system blocks
- Provides architectural details of HW
  - Suggestions to resolve inefficiencies



HTML System Viewer

- Displays kernel pipeline implementation and memory access implementation
- Visualize
  - Off-chip memory
    - Load-store units
    - Accesses
  - Stalls
  - Latencies
  - On-chip memory
    - Implementation
    - Accesses

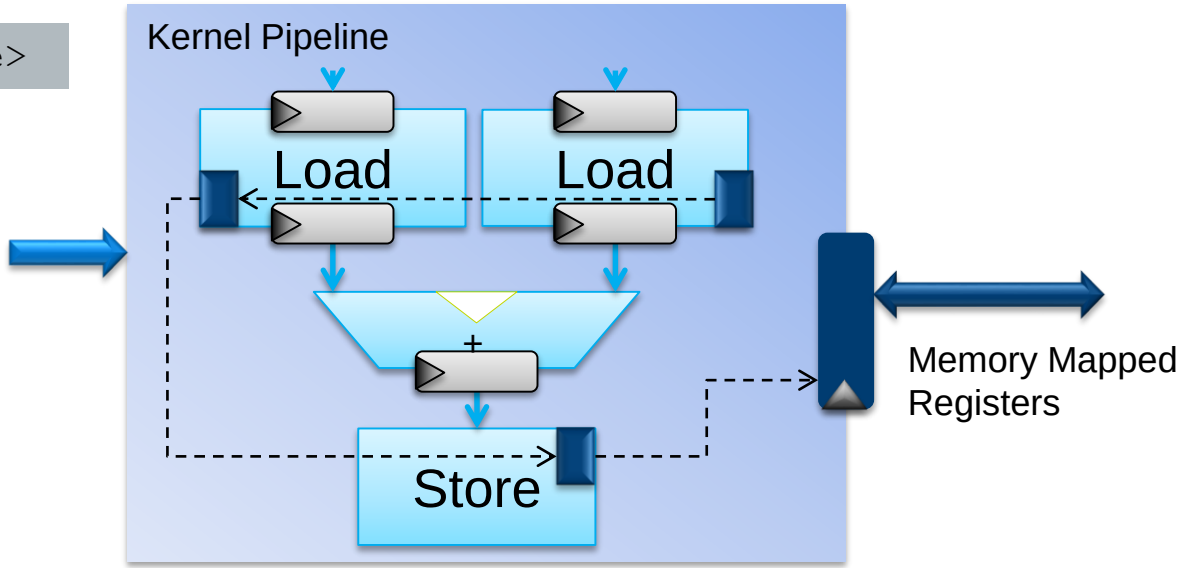


Profiler

- Inserts counters and profiling logic into the HW design
- Dynamically reports the performance of kernels

```
aoc --profile <kernel file>
```

```
kernel void accel(…) {
  …
  gid = get_global_id(0);
  out[gid] = a[gid]+b[gid];
  …
}
```



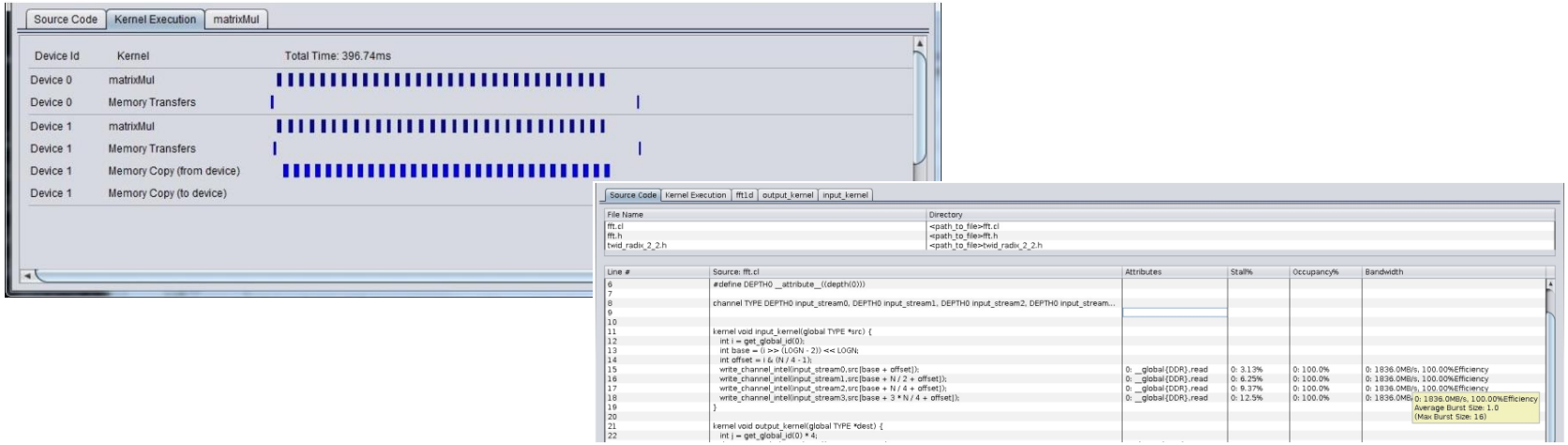
Collecting and Viewing Profile Information

- Compile kernel with `aoc --profile` option
  - `.source` file generated containing source information
- Run host application with generated aocx file
  - Performance counters will collect profile information
  - Host saves a `profile.mon` monitor description file to working directory
- View statistical data using the profiler GUI
  - Optionally provide `.source` file to view source code of profiled application

```
aocl report <kernel file>.aocx profile.mon [<kernel file>.source]
```

Profiler Reports

- Get runtime information about kernel performance
- Reports bottlenecks, bandwidth, saturation, and pipeline occupancy
  - At data access points



Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

Executing OpenCL Kernels

NDRange Kernels

OpenCL on Intel® FPGAs

- The Intel FPGA SDK for OpenCL
- Debug Tools
- FPGA-specific Features

Intel® FPGA-Specific Features

- Single Work-Item Execution
- Channels
- Controlling Hardware Generation
- Libraries
- SoC Platforms
- Shared Virtual Memory
- Custom Boards

Single Work-Item Execution

- Launching kernels with global size of (1,1,1)
  - A kernel executed on a compute unit with exactly one work-item
  - Or use `cl::CommandQueue::enqueueTask`
- Defined as a **Task** in OpenCL™
- **Single work-item kernels almost always have an outer loop**
  - **Loops in kernels automatically parallelized by the Intel® FPGA OpenCL Offline Compiler**
  - ***Entire kernel gets pipeline parallelized!***
- Intel FPGA specific feature that wouldn't run well on other architectures

Single-Threaded Kernels Motivation

- Data parallelism isn't always easy to extract
- NDRange execution may not be suitable for certain situations
  - Difficulties partitioning data into workgroups
  - Streaming application where data cannot arrive in parallel
- Some algorithms that are inherently sequential and depend on previous results
  - E.g. FIR filters, compression algorithms
- Sequential programming model of tasks more similar to C programming
  - Certain usage scenario more suited for sequential programming model
  - Easier to port

Data Parallelization Review

OpenCL™ NDRange execution best suited for applications where each loop iteration is independent

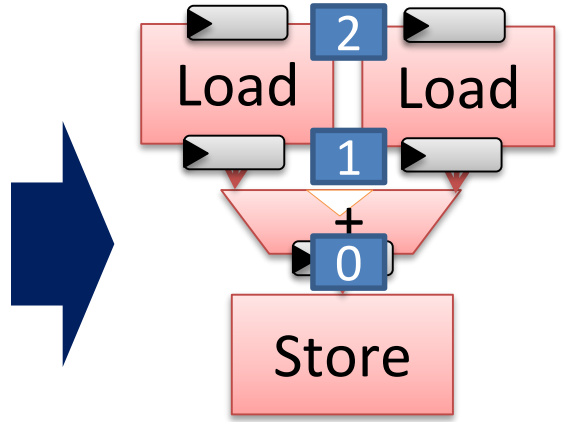
Algorithm

```
for (int i=0; i < n; i++)
    answer[i] = a[i] + b[i];
```

OpenCL™ Implementation

```
__kernel void sum(__global const float *a,
                  __global const float *b,
                  __global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}
```

FPGA Acceleration through Pipelined Execution

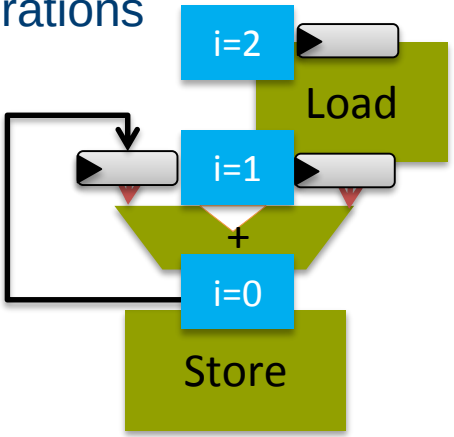


Tasks and Loop-pipelining Implementation

- Allow users to express programs as a single-thread kernel

```
for (int i=1; i < n; i++) {
     c[i] = c[i-1] + b[i];
}
```

- Compiler will infer parallel pipelined execution across loop iterations
  - Pipeline parallelism still leveraged to efficiently execute loops
  - Dependencies resolved by the compiler
  - Values transferred between loop iterations with FPGA resources
    - No need to buffer up data
    - Easy and cheap to share data through feedbacks in the pipeline



Loop Pipelining

AOC will pipeline each iteration of the loop for acceleration

- Analyze any dependencies between iterations
- Schedule these operations
- Launch the next iteration as soon as possible

```
float array[M];
for (int i=0; i < n; i++)
{
   for (int j=0; j < M-1; j++)
        array[j] = array[j+1];
   array[M-1] = a[i];

   for (int j=0; j < M; j++)
      answer[i] += array[j] * coefs[j];
}
```

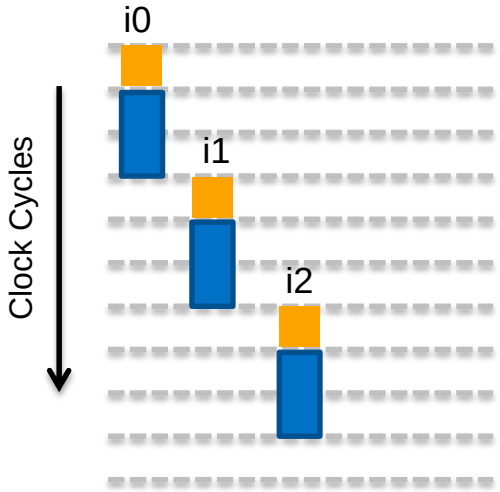
Shift Register `array`
(Dependency for next iteration)

At this point, launch the next iteration

Reduction on `array`
(Not a dependency)

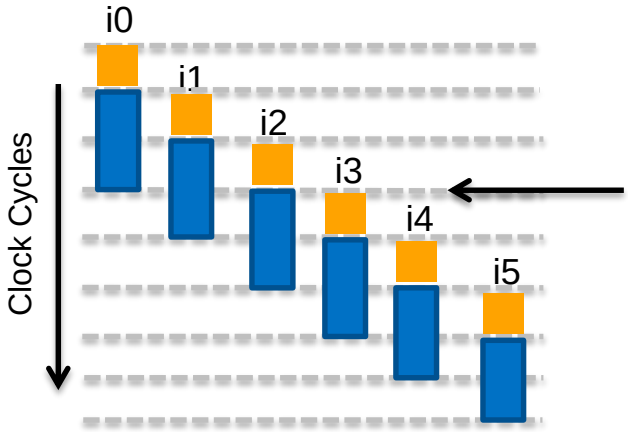
Loop Pipelining Example

No Loop Pipelining



No Overlap of Iterations!

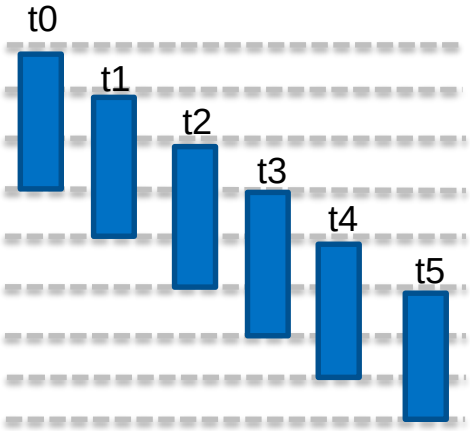
With Loop Pipelining



Finishes Faster because Iterations Are Overlapped

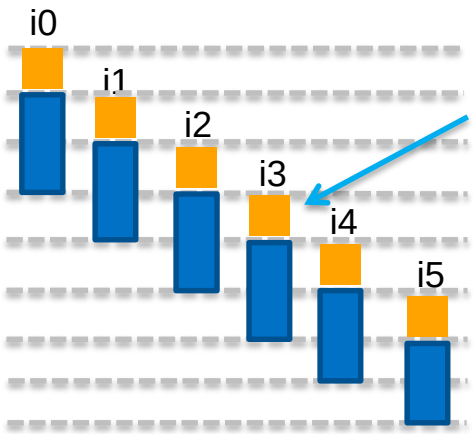
Parallel Threads vs Loop Pipelining

NDRange Parallel Threads



Parallel threads launch 1 thread per clock cycle in pipelined fashion

Loop Pipelining



If loop dependency resolved in 1 clock cycle

- Loop Pipelining enables Pipeline Parallelism AND the communication of state information between iterations.
  - If dependency resolved in 1 clock cycle, then the throughput is the same
  - *Data dependency resolved without adding extra compute time!*

Single Work-Item vs. NDRange Kernels

- One approach is not better than the other
  - Depends on application!
  - Determined early in the design stage
- Create single work-item kernels if
  - Data processing sequencing is critical
  - Algorithm cannot break down into work-items easily due to data dependencies
  - Data do not arrive in parallel prior to kernel launches
  - Data cannot be easily partitioned into workgroups
- Create NDRange kernels if
  - Algorithm does not have dependencies on previous calculations or data that isn't ready
  - Kernel can execute multiple work-items in parallel efficiently
    - Able to take advantage of SIMD processing and multiple compute units

Recognition of Single Work-Item Kernels

AOC assumes single work-item kernel if kernel does not query any work-item information

- No `get_global_id(), get_local_id(), or get_group_id()` calls
- Enables AOC to automatically perform outer loop pipelining and memory dependency analysis on the kernel
- Many C-based algorithms can directly compile to OpenCL™ Task

```
__kernel void mykernel (…) {
     for (i=0; i< FFT_POINTS; i++) {
          …
     }
}
```

Launching Single Work-Item Kernels (Tasks)

- Use `clEnqueueNDRangeKernel` with `work_dim`, `global_work_size`, and `local_work_size` set to 1
- Or `clEnqueueTask` in host code
  - Equivalent to the above `clEnqueueNDRangeKernel` call

Host Code

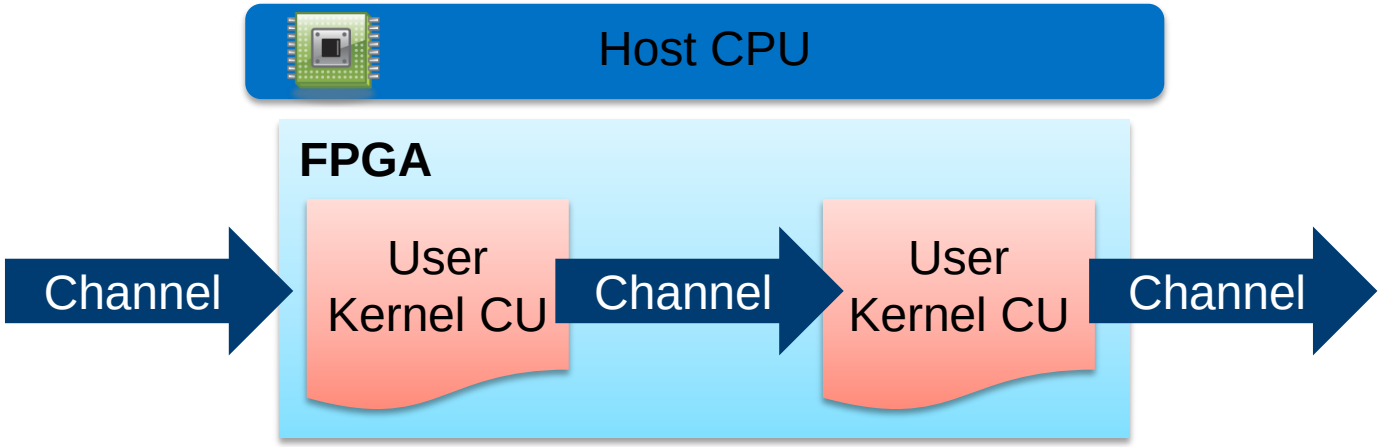
```
setup_memory_buffers();
transfer_data_to_fpga();

clEnqueueTask(myqueue, mykernel, …);

read_data_from_fpga();
```

*Note: `clEnqueueTask` deprecated in OpenCL™ 2.0

Channels / Pipes

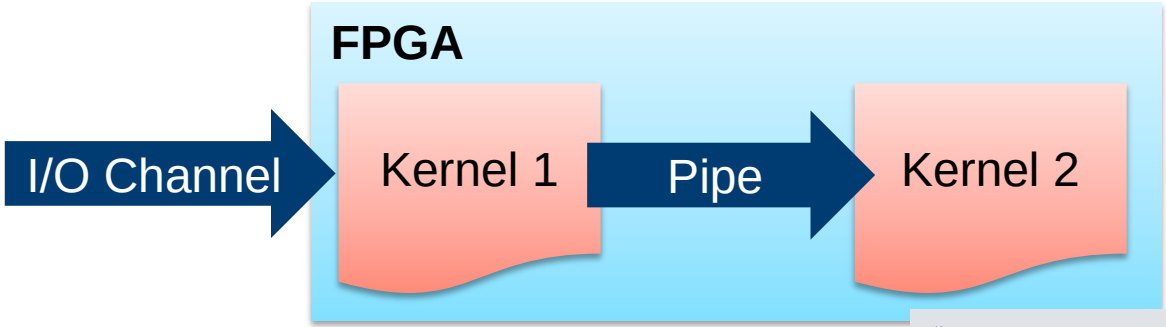
Allows I/O-to-kernel and kernel-to-kernel communication without going through global memory

- Enable `aoc` to implement custom FIFOs to stream data in/out of kernels



See the best practices guide for more information

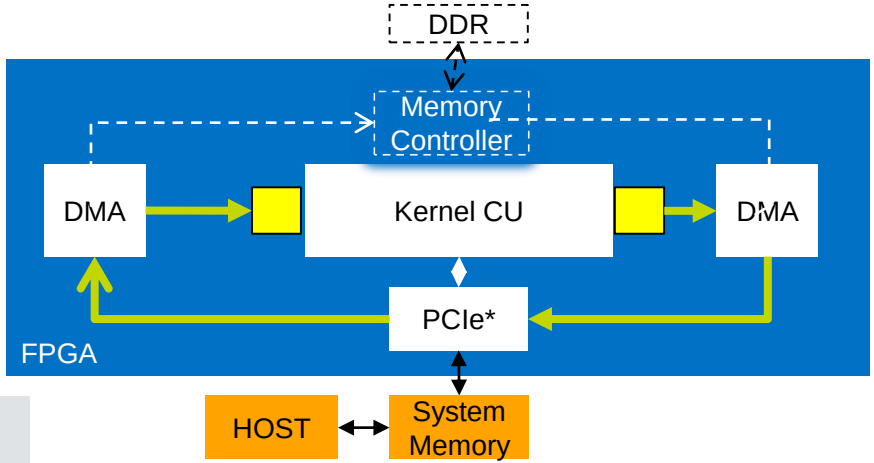
Channel / Pipe Implementation Example




```
#pragma OPENCL EXTENSION cl_intel_channels : enable
channel uint c0 __attribute__((io("eth0_in")));

kernel void kernel1(write_only pipe uint p1) {
      …
      iData = read_channel_intel(c0);
      …
      write_pipe(p1, &oData);
      }
}
kernel void kernel2(read_only pipe uint p1) {
      …
      read_pipe(p1, &value);
}
```

Host Pipes

- Allow host to send/receive data to/from the kernels without global memory
  - Performance advantage
  - Achieve peak host-to device bandwidth



Kernel Code

```
#pragma OPENCL EXTENSION cl_intel_fpga_host_pipe : enable
kernel void reader(__attribute__((intel_host_accessible))
                   __read_only pipe ulong4 host_in) {....}
kernel void writer(__attribute__((intel_host_accessible))
                   __write_only pipe ulong4 device_out) {....}
```

Host Code

```
cl_mem read_pipe = clCreatePipe( context,CL_MEM_HOST_READ_ONLY, …);
cl_mem write_pipe = clCreatePipe( context, CL_MEM_HOST_WRITE_ONLY, …);
clReadPipeIntelFPGA (read_pipe, &val);
clWritePipeIntelFPGA (write_pipe, &val);
```

Control Generation of Compute Unit Hardware

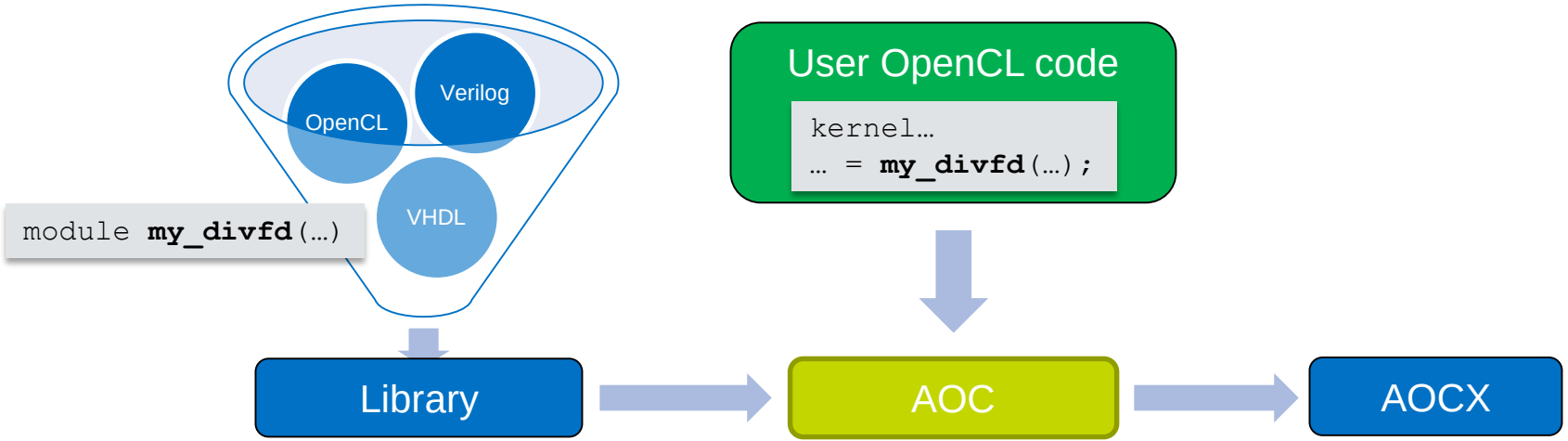
Kernel developer can control hardware generation through attributes

- Compute Unit Replication (Number of CU per kernel)
- Compute Unit Vectorization (Width of SIMD Lane)
- Autorun Kernels (Kernels that start running without the host)
- Maximum Workgroup Size (Synchronization Hardware needed)
- Loop Hardware Unrolling (Number loop iterations to execute at once)
- Control Memory Implementation (Control memory system topology)

See the best practices guide for more information

OpenCL™ Libraries

Create libraries from RTL or OpenCL™ source and call those library functions from user OpenCL code



See the Intel® FPGA SDK for OpenCL Programming Guide for detailed examples

RTL Function Component

- XML File
  - Describes properties of the RTL components
    - Used by aoc to integrate with rest of the OpenCL™ pipeline
  - Maps RTL ports to streaming interfaces supported by aoc
  - Specify RTL and OpenCL model files
- RTL Source File
  - Verilog, System Verilog, or VHDL file for the component
- OpenCL emulation model file (.cl)
  - OpenCL model implementation for the RTL component used by the emulator
- Header file (<library>.h)
  - C-style header file declaring the signatures of the function implemented

```
double my_divfd(double a, double b);
```

RTL Functions XML File

```
<RTL_SPEC>
   <FUNCTION name="my_sqrtfd" module="my_fp_sqrt_double">
   <ATTRIBUTES> <PARAMETER name="WIDTH" value="32"/>
      <IS_STALL_FREE value="yes"/>
      <IS_FIXED_LATENCY value="yes"/>
      <EXPECTED_LATENCY value="31"/>
      <CAPACITY value="1"/>
      <HAS_SIDE_EFFECTS value="no"/>
      <ALLOW_MERGING value="yes"/>
   </ATTRIBUTES>
   <INTERFACE>
      <AVALON port="clock" type="clock"/>
      <AVALON port="resetn" type="resetn"/>
      <AVALON port="ivalid" type="ivalid"/>
      <AVALON port="iready" type="iready"/>
      <AVALON port="ovalid" type="ovalid"/>
      <AVALON port="oready" type="oready"/>
      <INPUT port="datain" width="64"/>
      <OUTPUT port="dataout" width="64"/>
   </INTERFACE>
```

IP Properties

Interface signals and properties

Associated emulation model file and HDL source file

```
<C_MODEL>
   <FILE name="c_model.cl" />
</C_MODEL>
<REQUIREMENTS>
   <FILE name="my_fp_sqrt_double_s5.v" />
   <FILE name="fp_sqrt_double_s5.vhd" />
</REQUIREMENTS>
</FUNCTION>
</RTL_SPEC>
```

Library Function Usage

- Include <library>.h
- Call the library function as if it's any other function

```
#include "lib_header.h"

kernel void test_lib ( __global double * restrict in,
                       __global double * restrict out,
                       int N)
{
     int i = get_global_id(0);
     for (int k =0; k < N; k++)
     {
          double x = in[i*N + k];
          out[i*N + k] = my_divfd(my_rsqrtfd(x), my_sqrtfd(my_rsqrtfd (x)));
     }
}
```

Creating Library and Compiling with Library

1a. Package a single RTL component into an object file

```
aocl -rtl comp_spec.xml -o add.aoco
```

XML file describing RTL component

"object file" containing a single library component

1b. Package an OpenCL™ file with helper functions into an object file

```
aoc -rtl -shared sub.cl -o sub.aoco
```

Flag to compile OpenCL file for library inclusion.

2. Package multiple object files into a library file

```
aocl library create -o mylib.aoclib add.aoco sub.aoco
```

Library file

List of object files

3. Use a library during OpenCL kernel compilation

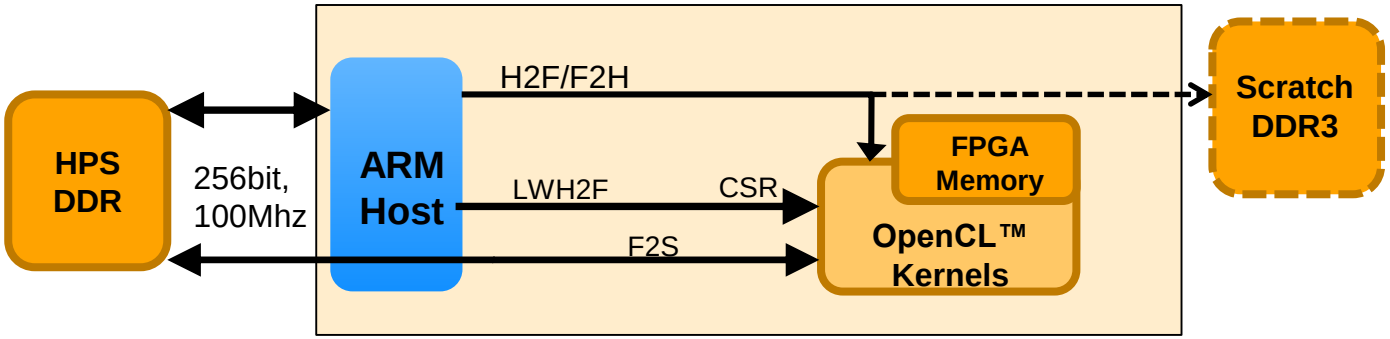
```
aoc -l mylib.aoclib [-L <lib_dir>] mykernel.cl
```

Multiple instances of *<library file name> and <library directory> permitted*

Intel® SoC FPGA Platforms

Running embedded host on the ARM* Cortex*-A9 Processors of SoC Devices

- Intel® Cyclone® V, Arria® V, or Intel Arria 10 SoC devices
- Take advantage of zero-copy shared physical memory using the HPS DDR memory and FPGA-to-SDRAM bridge



Shared Physical Memory - SoC Devices

ARM* CPU and the FPGA can share DDR memory on the SoC device

- Using shared memory instead of dedicated FPGA DDR is recommended
- Mark the shared buffers between kernels as volatile
  - Ensures that buffer modification by one kernel is visible to another

```
__kernel void producer(__global volatile uint * restrict shared_mem)
```

- Use `clEnqueueMapBuffer` function to create host pointer

Host Code for Allocating Shared Memory (SoC)

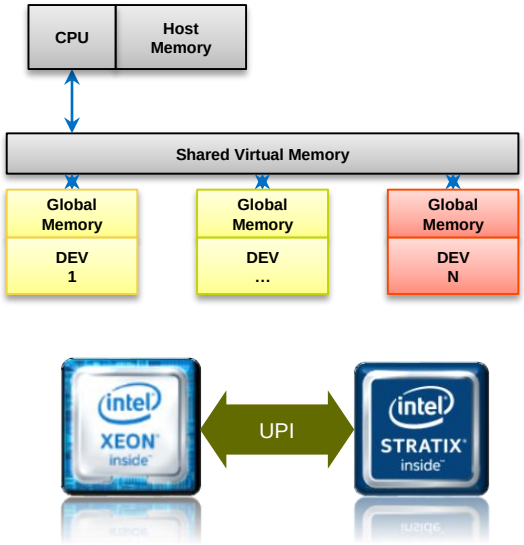
- Shared memory must be physically contiguous
- CPU caching is disabled for the shared memory
- Use the `CL_MEM_ALLOC_HOST_PTR` flag
- `clEnqueueMapBuffer` function required to create pointer to allocated space
  - `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`, `malloc`, `new` not used

```
cl_mem src = clCreateBuffer(…, CL_MEM_ALLOC_HOST_PTR, size, …);
int *src_ptr = (int*)clEnqueueMapBuffer (…, src, size, …);
*src_ptr = input_value; //host writes to ptr directly
clSetKernelArg (…, src);
clEnqueueNDRangeKernel(…);
clFinish();
printf ("Result = %d\n", *dst_ptr); //result is available immediately
clEnqueueUnmapMemObject(…, src, src_ptr, …);
clReleaseMemObject(src); // actually frees physical memory
```

Shared Virtual Memory

Shared Virtual Memory (SVM) is used to share buffers between host and FPGA rather than explicitly transferring data to and from

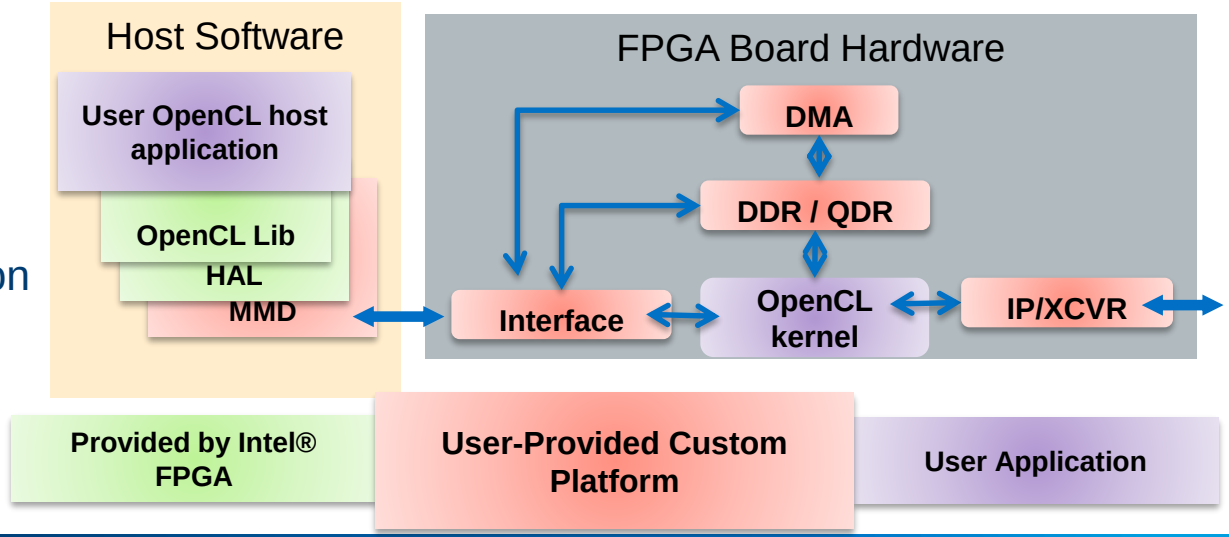
- Enables pointer sharing
- Available over cache-coherent interfaces
  - e.g. Intel® multi-chip package (MCP) over UltraPath Interconnect (UPI)
- OpenCL™ 2.0 Shared Virtual Memory Standard



Custom Platform

Framework of host software and FPGA interface design to enable the use of OpenCL™ on a custom board

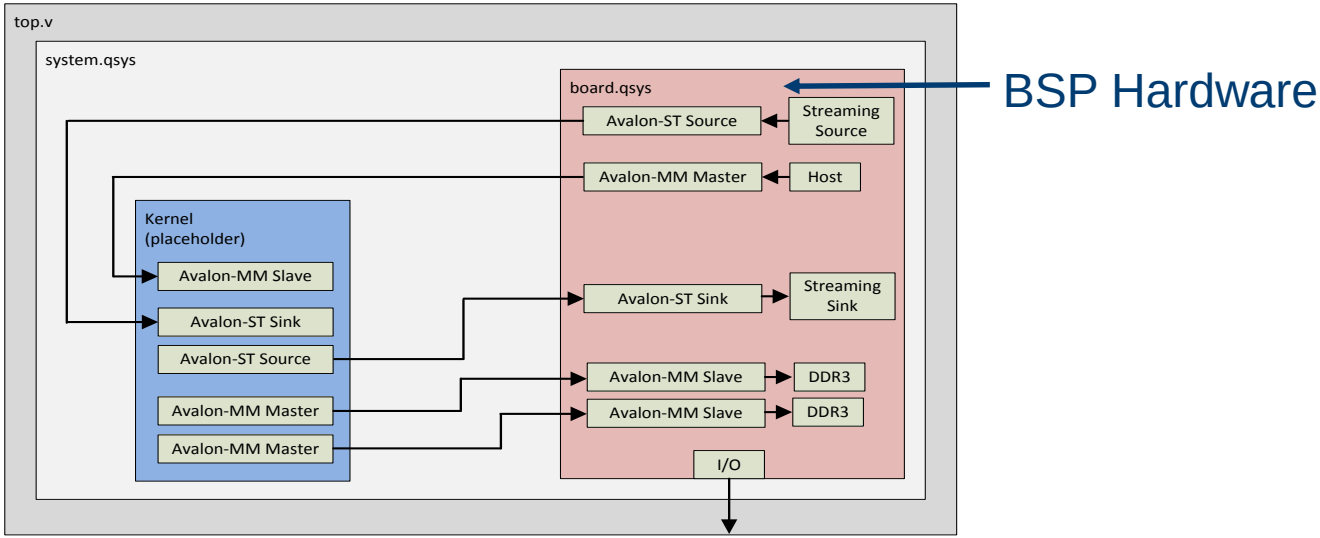
- FPGA design, software, and board bring up skills required
- Custom BSP provides
  - Timing-closed Hardware
  - MMD software layer
  - Some AOCL utility function



Hardware System Overview

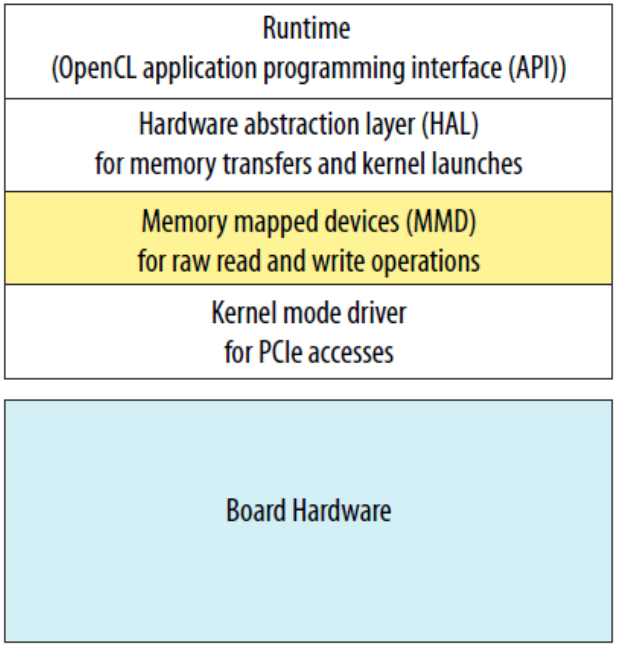
Board support package hardware (board.qsys) need to be provided

- Timing closed and fitting locked



Memory Mapped Devices (MMD) Software Layer

- Software layer for communicating with board
- Used by host programs and board utilities
- API functions specified in `aocl_mmd.h`
  - Header file is <custom board>\source\include\aocl_mmd.h
  - Includes details of the API function prototype
- File I/O like interface needs to be implemented
  - Read/write/open/close/reprogram etc.
- Compiled MMD library needs to be provided
  - To be linked to by the host program



AOCL Utilities

Utility executables delivered in the custom platform

- Set utilbindir in Board XML to containing directory
- Install/Uninstall (`aocl install`)
  - Installs kernel driver into the host operating system
- Program (`aocl program <device> <kernel file>.aocx`)
  - Programs FPGA from provided aocx file using `aocl_mmd_reprogram` MMD API call
- Flash (`aocl flash <device> <kernel_filename>.aocx`)
  - Programs base programming image into Flash
- Diagnose (`aocl diagnose <device_name>`)
  - Useful in identifying memory transfer performance and board issues

Reference Platforms

- Contains both the hardware and software platform layers and reference designs
  - Modify to create custom FPGA accelerator boards
- Reference platforms available for Cyclone® V, Stratix® V, Intel® Arria® 10, Intel Stratix 10 FPGAs
  - Ships with the Intel FPGA SDK for OpenCL™
- Skeleton design template project
- Download the appropriate reference platform porting guide at the support page
  - https://www.altera.com/products/design-software/embedded-software-developers/opencl/support.html

OpenCL™ References

- Intel® FPGA OpenCL collateral
  - https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html
  - Intel FPGA SDK for OpenCL™ Getting Started Guide
  - Intel FPGA SDK for OpenCL Programming Guide
  - Intel FPGA SDK for OpenCL Best Practices Guide
  - Free Intel FPGA OpenCL Online Trainings
- Khronos* Group OpenCL Page
- OpenCL Reference Card
  - https://www.khronos.org/files/OpenCLPP12-reference-card.pdf

Exercise 4

Examining Kernel Compilation Reports

Intel® FPGA Technical Support Resources

- Intel® FPGA Technical Training materials
- Intel® Programmable Solutions Group (PSG) community forum for self-help
- Intel PSG wiki site for design examples
- Intel PSG Knowledge Base Solutions
- Intel PSG Self Servicing License Center
- Design Service Network (DSN) www.elcamino.de
- Please contact your sales and field support if you need further assistance

Appendix: C API

Platforms IDs

Platform: Vendor-specific implementation of OpenCL™

- Obtain the list of platforms available with `clGetPlatformIDs`

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

Error code

Size of *platforms* array

Returns a list of platform IDs

Returns the total number of platforms available

Device IDs

Device: An OpenCL™ accelerator supported by a platform

- Obtain the list of devices available with `clGetDeviceIDs`

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

Error code

Platform to look in

Device Types
CPU, Accelerator (FPGA),
GPU, Default, All
i.e. CL_DEVICE_TYPE_ALL

Size of *devices* array

Returns a list of device IDs

Returns the total number of devices available

Create Context

Use `clCreateContext` to create a context with one or more devices

```
cl_context clCreateContext(cl_context_properties *properties,
                           cl_uint num_devices,
                           const cl_device_id *devices,
                           void CL_CALLBACK  *pfn_notify (
                                   const char *errinfo,
                                   const void *private_info,
                                   size_t cb,
                                   void *user_data),
                           void *user_data,
                           cl_int *errcode_ret)
```

Returns the context

Properties that define context behavior

Number of elements in *devices*

List of devices in context

Callback function to be registered to handle errors in the context

Data argument for *pfn_notify*

Error code

May also use `clCreateContextFromType`

Platform Layer APIs Called to Setup Environment

1. Call `clGetPlatformIDs` to get available number of platforms
   - If unknown
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
   - If unknown
5. Allocate space to hold device information
6. Call `clGetDeviceIDs` again to fill in devices
7. Call `clCreateContext` to create a context that manages kernel execution

Example Platform Layer Code

```
//Get the first platform ID
cl_platform_id myp;
err=clGetPlatformIDs(1, &myp, NULL);

// Get the first FPGA device in the platform
cl_device_id mydev;
err=clGetDeviceIDs(myp, CL_DEVICE_TYPE_ACCELERATOR, 1, &mydev, NULL);

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

Note: Error checking should be performed, but is not shown here.

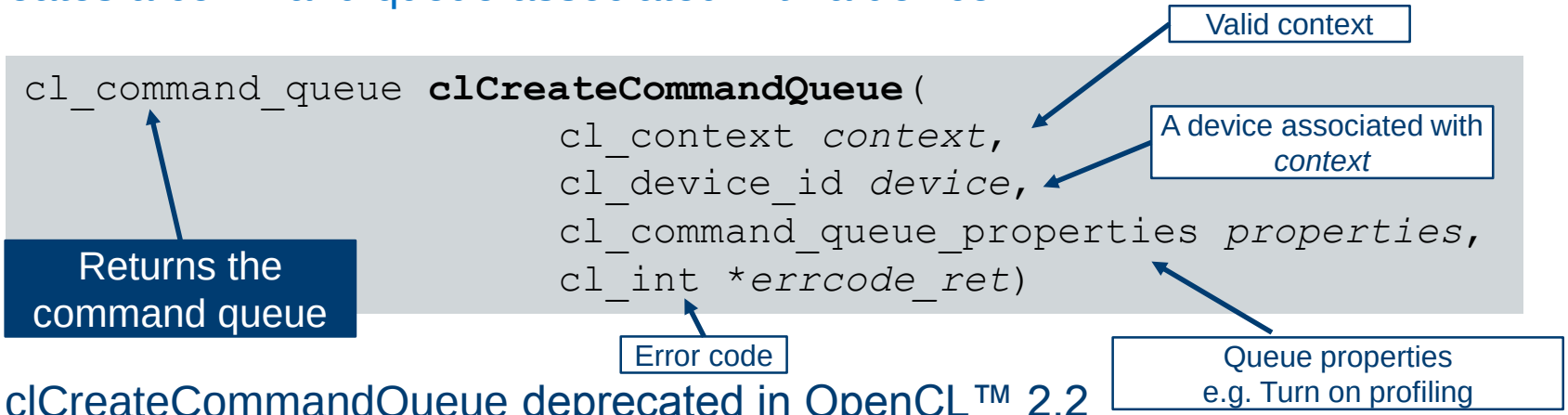
clGet<Platform/Device/Context>Info

Query information about a platform, device, or context

- Use `clGetPlatformInfo` to return information about an OpenCL™ platform
  - Vendor, OpenCL version, platform profile, extensions, etc…
  - Pass in `cl_platform_id` and `cl_platform_info` Enum type
- Use `clGetDeviceInfo` to return information about a device
  - Memory sizes, Bus widths, Device Type, Endianness, etc.
  - Pass in `cl_device_id` and `cl_device_info` Enum type
- Use `clGetContextInfo` to return information about a context
  - Number of devices in context, context properties, and reference count
  - Pass in `cl_context` and `cl_context_info` Enum type

Create a Command Queue

Creates a command queue associated with a device




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

- clCreateCommandQueue deprecated in OpenCL™ 2.2
  - Use clCreateCommandQueueWithProperties instead
- Host will submit commands to the device through the command queue
  - Using *clEnqueue…* commands

clCreateBuffer

Allocates and creates a buffer memory object

- One dimensional collection of elements that can be scalars (int, float), vector data types, or user-defined structures
- 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)
```

Returns the buffer object

Valid context

Allocation and usage information *(See next slide)*

Size in bytes

Pointer to data already allocated in the host application (Optional)

Error code

- A buffer is passed to the kernel argument and converted to a pointer in the kernel
- In the host, a buffer is not a pointer. i.e. `mybuffer[3]=…` is not legal

Memory Management Buffer Flags

- Kernel access permissions (from the kernel perspective)
  - `CL_MEM_READ_WRITE` (default), `CL_MEM_WRITE_ONLY`, `CL_MEM_READ_ONLY`
- Host access permissions (from the host perspective)
  - `CL_MEM_HOST_WRITE_ONLY`, `CL_MEM_HOST_READ_ONLY`, `CL_MEM_HOST_NO_ACCESS`
- Host pointer options
  - `CL_MEM_COPY_HOST_PTR`
    - Data is copied from *`host_ptr`* to allocated device memory one time
  - `CL_MEM_USE_HOST_PTR`
    - *`host_ptr`* location is used for storage for the memory object
  - `CL_MEM_ALLOC_HOST_PTR`
    - Allocate in host accessible memory, used in SoC devices with shared physical memory

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

clEnqueueWriteBuffer

Write from host memory to buffer object (device)

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

Valid command queue

Error code

Destination Buffer

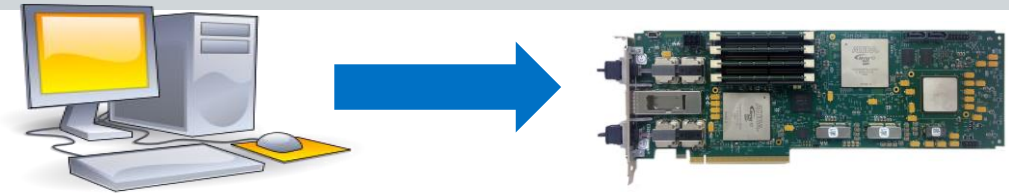
Set to CL_TRUE blocks call until *ptr* can be reused by the host

Offset in bytes in the buffer

Size in bytes of data to be written

Source host pointer

Events used for synchronization. Discussed later



clEnqueueReadBuffer

Read from buffer object (device) to host memory

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

Valid command queue

Error code

Source Buffer

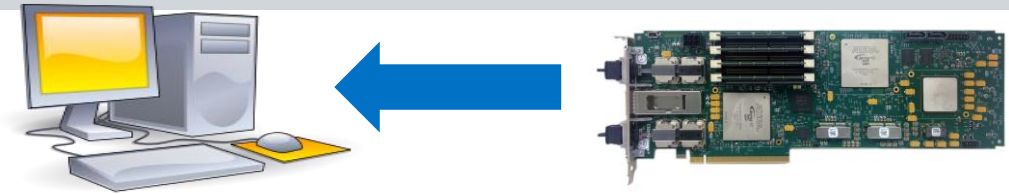
Set to CL_TRUE blocks call until buffer data copied to *ptr*

Offset in bytes in the buffer

Size in bytes of data to be read

Destination host pointer

Events used for synchronization. Discussed later



Memory Management – Code Example

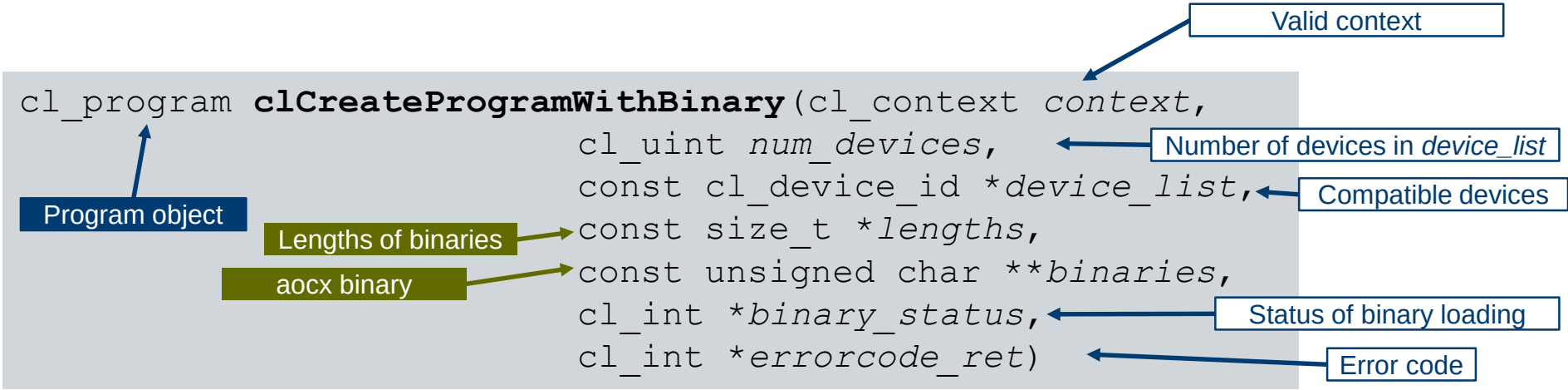
```
const int N = 5;
int nBytes = N*sizeof(int);
int hostarr [N] = {3,1,4,1,5};

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

// Allocate memory on device
cl_mem a;
a = clCreateBuffer(context, CL_MEM_READ_WRITE, nBytes, NULL, &err);

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

Code Example

```
__kernel void increment ( __global float *a, float c, int N)
{
     int i;
     for (i = 0; i < N; i++)
          a[i] = a[i] + c;
}
```

```
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 Programs from Binary for FPGAs



- For Intel® FPGA
  - *lengths* is the size of the `aocx` file in bytes
  - *binaries* is the contents of the `aocx` file
  - When kernels from the `aocx` is run, the host will configure the FPGA with the `aocx`

Building Programs

Compiles and links a program executable from the program source or binary

- For Intel® FPGA, needs to be called to conform to the standards, but nothing meaningful done

```
cl_int clBuildProgram(cl_program program,
                      cl_uint num_devices,
                      const cl_device_id *device_list,
                      const char *options,
                      void (*pfn_notify)(cl_program,
                                         void *user_data),
                      void *user_data)
```

Error code

Program object to build

Number of devices in *device_list*

Compatible devices

Build Options

Callback function for when build has completed

Data for callback function

Create and Build Program - Code Example

```
void main()
{
...

//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);

// 1. Create then build the program
cl_program program = clCreateProgramWithBinary(context, 1, &myDevice, &length,
                                               (const unsigned char**)&binaries, &status,
                                               &err);
err = clBuildProgram(program, 1, &myDevice, "", NULL, NULL);
```

Creating Kernels

Create kernels from programs with **`clCreateKernel`**

- For Intel® FPGA, able to load any of the kernels compiled into the `aocx` file by the offline compiler

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

Program object

Kernel function name

Kernel object corresponding to the kernel function

Error code

Creating Kernels – Code Example

```
__kernel void increment ( __global float *a, float c, int N)
{
      int i;
      for (i = 0; i < N; i++)
           a[i] = a[i] + c;
}
```

```
void main()
{

   ...
   // 1. Create then build the program
   cl_program program = clCreateProgramWithBinary(...);
   err = 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
}
```

Set Kernel Arguments

Use `clSetKernelArg` to set the value for a specific argument of a kernel

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

Error code

Kernel object

Argument index
From 0 (leftmost arg) to *N*-1 (*N* is the total number of args)

Size of argument value

Pointer to data used as argument value

- Important to set the *arg_index* correctly
  - Limited error checks done

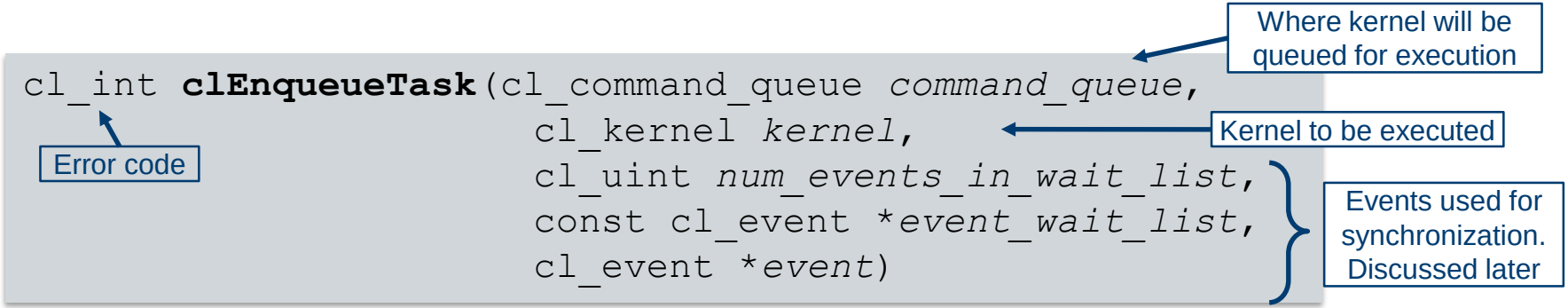
Setting Up Kernel Argument List - Code Example

```
__kernel void increment ( __global float *a, float c, int N)
{
      int i;
      for (i = 0; i < N; i++)
           a[i] = a[i] + c;
}
```

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

   // 3. Allocate and transfer buffers on/to device
   cl_mem a_device = clCreateBuffer(…);
   cl_float c_host = 10.8;
   ...

   // 4. Set up the kernel argument list
   err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_device);      // Set up ‘a’
   err = clSetKernelArg(kernel, 1, sizeof(cl_float), (void *)&c_host);      // Set up ‘c’
   err = clSetKernelArg(kernel, 2, sizeof(cl_int), (void *)&NUM_ELEMENTS);      // Set up ‘N’

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

Execute Kernel

Use `clEnqueueNDRangeKernel` or `clEnqueueTask` to run kernel on device



- `clEnqueueNDRangeKernel` discussed later

*Note: `clEnqueueTask` deprecated in OpenCL™ 2.0

Kernel Launch - Code Example

```
void main()
{
    …
    cl_program program = clCreateProgramWithBinary(...);
    err = clBuildProgram(...);
    cl_kernel kernel = clCreateKernel(program, ”increment”, &err);
    …
    err = clSetKernelArg(…)
    …
    // 5. Launch the kernel
    err = clEnqueueTask(queue, kernel, 0, NULL, NULL);

    // 6. Transfer result buffer back
}
```

Kernel Execution Complete Example

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

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

   // 3. Allocate and transfer buffers on/to device
   float* a_host = ...
   cl_mem a_device = clCreateBuffer(..., CL_MEM_COPY_HOST_PTR, a_host, ...);
   cl_float c_host = 10.8;

   // 4. Set up the kernel argument list
   err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_device);
   err = clSetKernelArg(kernel, 1, sizeof(cl_float), (void *)&c_host);
   err = clSetKernelArg(kernel, 2, sizeof(cl_int), (void *)&NUM_ELEMENTS);
```

Kernel Execution Complete Example Cont.

```
  …
  // 5. Launch the kernel
  err = clEnqueueTask( queue, kernel, 0, NULL, NULL);

  // 6. Transfer result buffer back
  err = clEnqueueReadBuffer( queue, a_device, CL_TRUE, 0, NUM_ELEMENTS*sizeof(cl_float),
                             a_host, 0, NULL, NULL);
}
```

Event Dependencies

- Each `clEnqueue`
  - Can **depend on** an array of (previously created) `cl_event`s
    - To ensures synchronization of data.
  - Can **generate** a `cl_event`
    - To be used later
  - The `clEnqueue` command itself does not block, just the execution of the associated task on the device

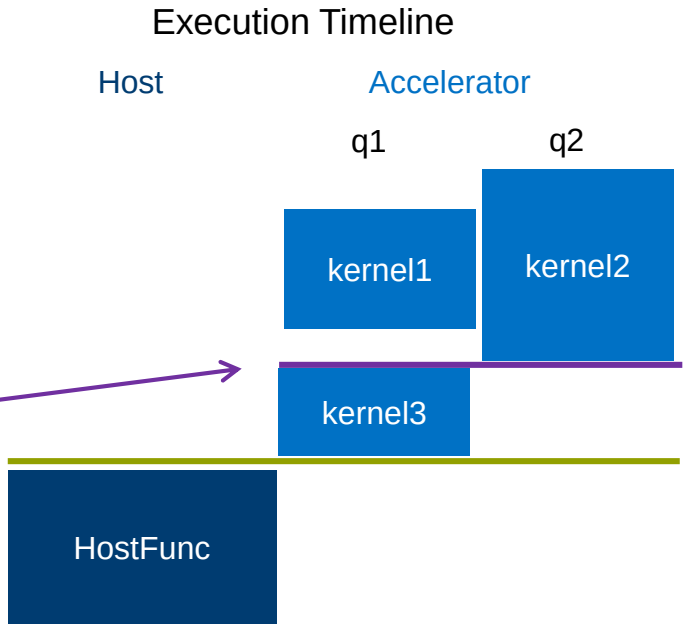
```
cl_int  clEnqueue… (  cl_command_queue command_queue,
                      …
                      cl_uint num_events_in_wait_list,
                      const cl_event *event_wait_list,
                      cl_event *event)
```

Wait for these to finish

Generate new event, to be used later. (If not NULL)

Synchronization Example

```
cl_command_queue q1, q2;
cl_event e1, e2;

clEnqueueNDRangeKernel(q1,k1,…, &e1);

clEnqueueNDRangeKernel(q2,k2,…, &e2);

cl_event elist[2];
elist[0]=e1;
elist[1]=e2;

clEnqueueNDRangeKernel(q1,k3,…,2,elist,NULL);

clFinish(q1);
clFinish(q2);

HostFunc();
```



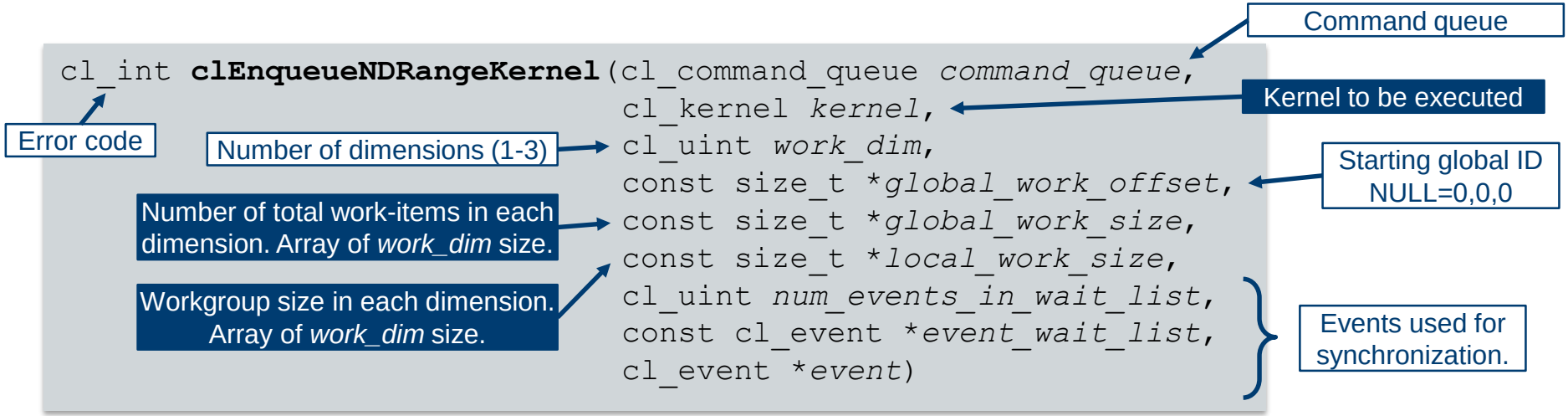
Clean Up

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

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

NDRange Launch

Use `clEnqueueNDRangeKernel` to launch a kernel with multiple work-items



Total work-items = *global_work_size*[0] * *global_work_size*[1] * *global_work_size*[2]

Kernel Launch - Code Example

```
//1D Work-Group
int err;
size_t const globalWorkSize = 1920;
size_t const localWorkSize = 8;
err=clEnqueueNDRangeKernel(queue, 1dkernel, 1, NULL, &globalWorkSize, &localWorkSize,
                0, NULL, NULL);

//3D Work-Group
size_t const globalWorkSize[3] = {512,512,512};
size_t const localWorkSize[3] = {16, 8, 2};
err=clEnqueueNDRangeKernel(queue, 3dkernel, 3, NULL, globalWorkSize, localWorkSize,
                0, NULL, NULL);
```

Legal Disclaimers/Acknowledgements

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at www.intel.com.

Intel, the Intel logo, Intel Inside, the Intel Inside logo, MAX, Stratix, Cyclone, Arria, Quartus, HyperFlex, Intel Atom, Intel Xeon and Enpirion are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

OpenCL is the trademark of Apple Inc. used by permission by Khronos

*Other names and brands may be claimed as the property of others


intel
experience
what's inside™