
# Introduction to OpenCL™ on FPGAs for Parallel Programmers

# Agenda

FPGA Basics

The OpenCL™ Model

Setting Up the Device at the Host

OpenCL Host-side Execution

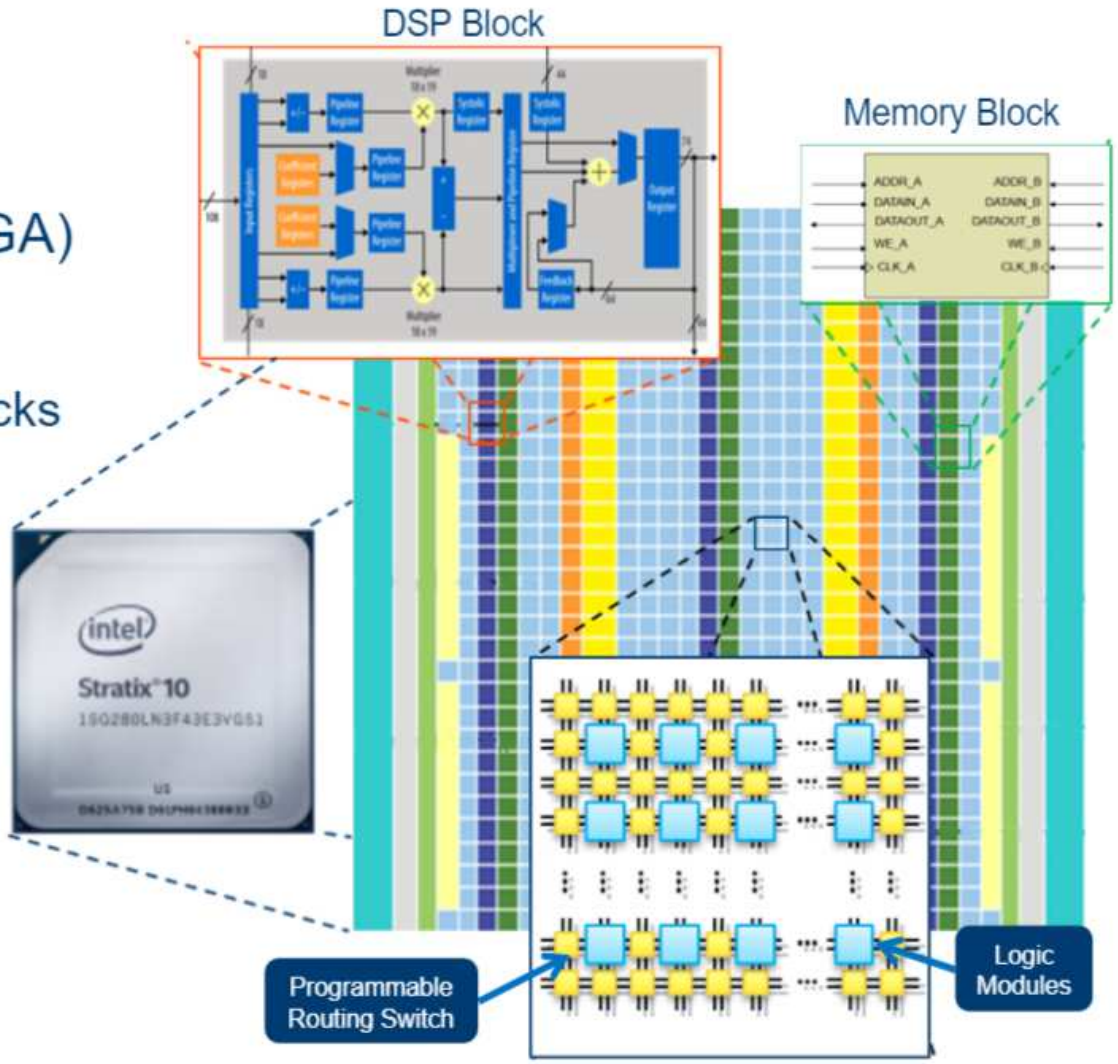
OpenCL Kernels

# FPGA Basics

What does the hardware look like? How does the software toolset work?

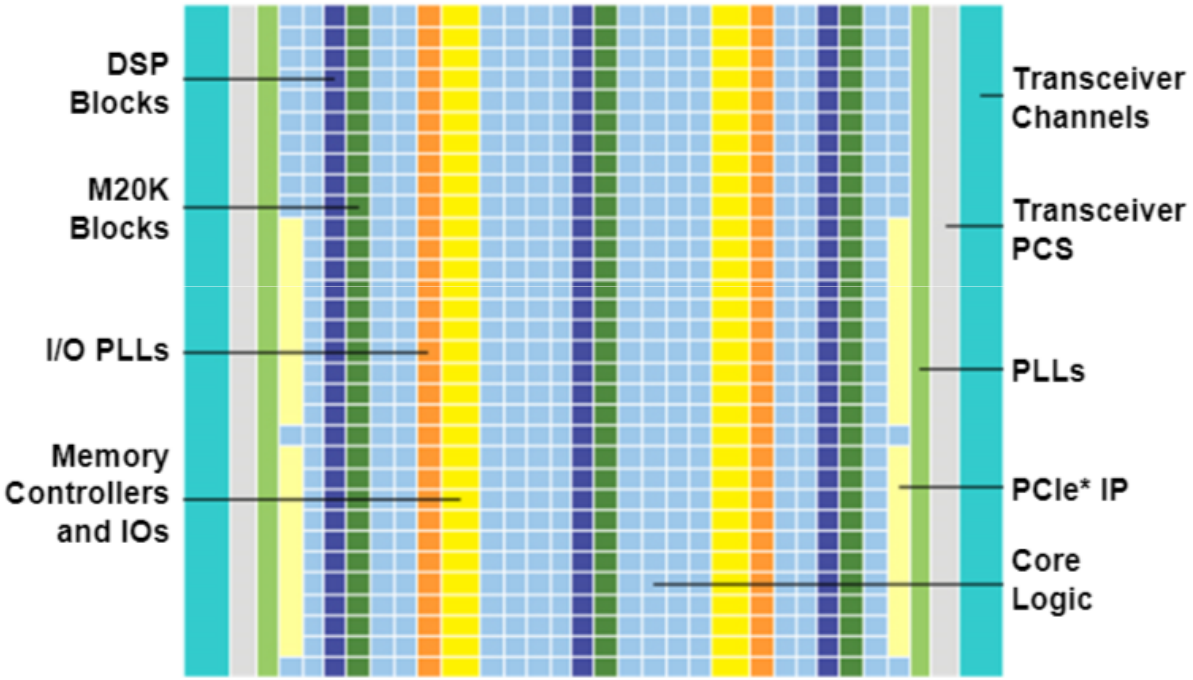
# FPGA Overview

- Field Programmable Gate Array (FPGA)
  - Millions of logic elements
  - Thousands of embedded memory blocks
  - Thousands of DSP blocks
  - Programmable routing
  - High speed transceivers
  - Various built-in hardened IP
- Used to create **Custom Hardware!**



# Advantages of Custom Hardware with FPGAs

- **Custom hardware!**
- Efficient processing
- Fine-grained parallelism
- Low power
- Flexible silicon
- Ability to reconfigure
- Many available I/O standards



# Traditional FPGA Design Flow

Design specification

```
module dut(…);
  always @(…)
  …
endmodule;
```

## Design entry/RTL coding

- Behavioral or structural description of design
- Possibly with the help of high-level tools

## RTL functional simulation

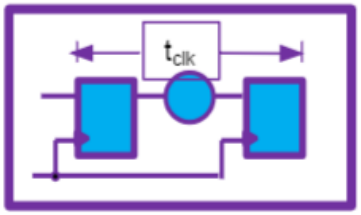
- Use 3rd party simulators
- Verify logic model & data flow


## Synthesis (Mapping)

- Translate design into device-specific primitives
- Design optimized
- Intel® Quartus® Prime synthesis or other 3rd party tools

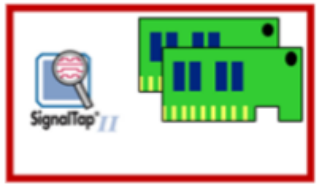

## Place & route (Fitting)

- Assign primitives to locations
- Route the resources




## Static timing analysis

- Verify performance specs can be met




## Board simulation & test

- Simulate board design
- Program & test device on board

# FPGA High Level Design with OpenCL™

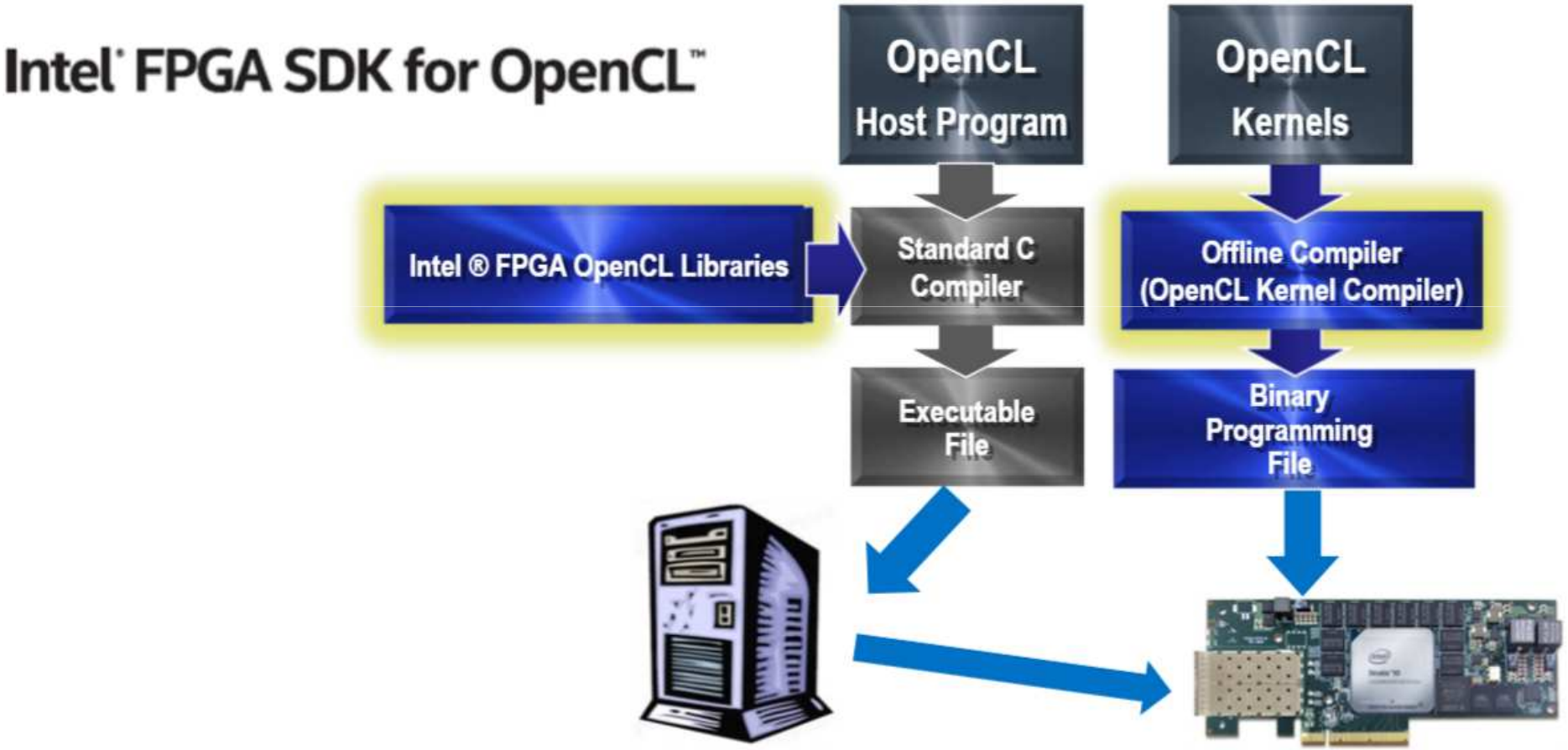
## Goal: Design FPGA custom hardware with C-based software language

```
__kernel void _foo (__global float *x) {
   int i …
}
```

Intel® FPGA SDK for OpenCL™


- Benefits
  - Makes FPGA acceleration available to software engineers
  - Debug and optimize in a software-like environment
  - Significant productivity gains compared to hardware-centric flow
  - Easier to perform design exploration
  - Abstracts away FPGA design flow and FPGA hardware

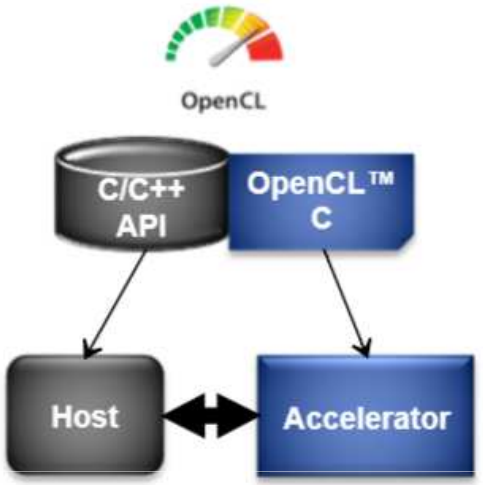
# Intel® FPGA SDK for OpenCL™ Usage



# The OpenCL™ Model

## Basics of how OpenCL models your system

# 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 conventional microprocessor
  - Supports efficient plumbing of complicated concurrent programs with low overhead
    - Through OpenCL host API
  - C / C++ supported. (Only C functions listed in this PPT)
- Used together to efficiently implement algorithms

# OpenCL™ vs CUDA* C

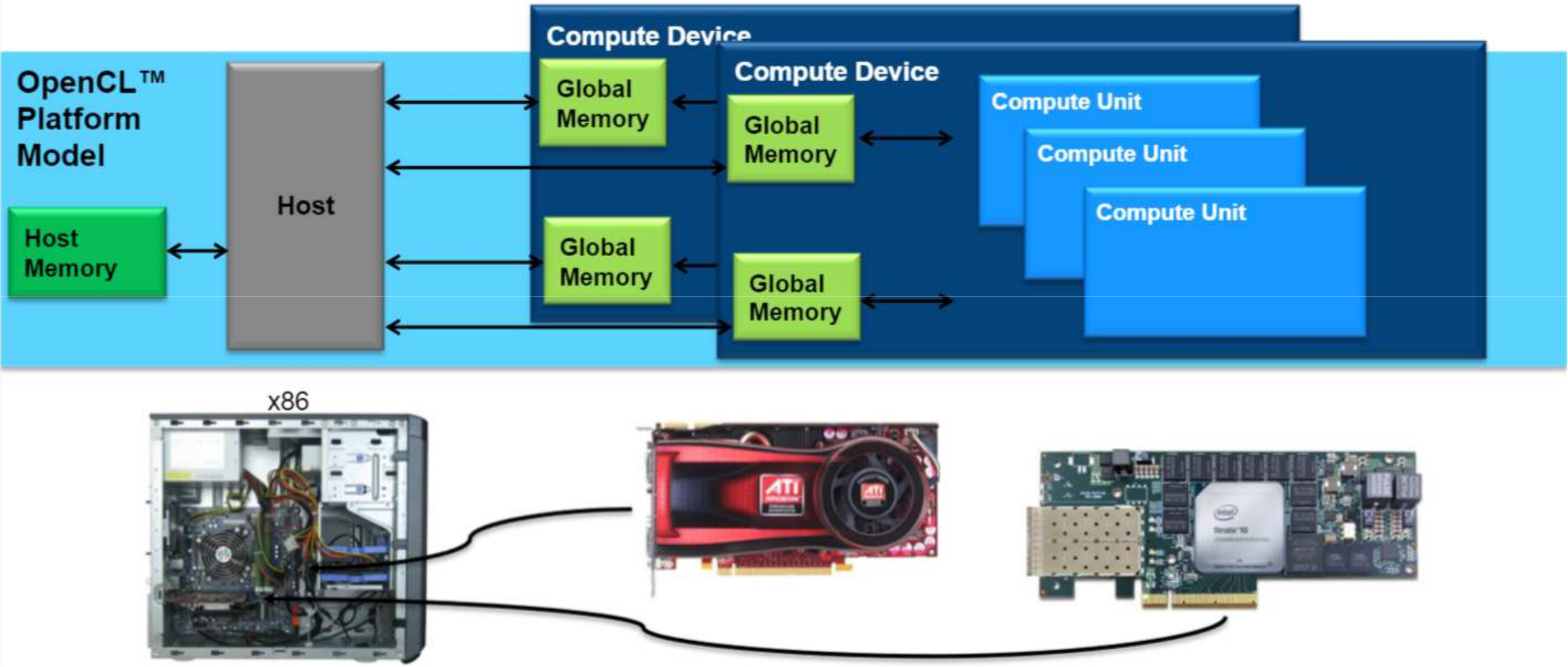
Majority of features are similar across the frameworks

OpenCL

- Open Standard
- Heterogeneous parallel computing framework
- Works with FPGAs, CPUs, many core processors (e.g. Xeon Phi™), and GPUs from several vendors including Nvidia*
  - Functionally portable across large range of devices from different vendors
- More flexible API with regard to synchronization, communication, compiler and other features


- Proprietary Solution
- Parallel computing platform
- Works with Nvidia* GPUs
- Designed only for GPUs

# Heterogeneous Platform Model



# Data Parallel Execution Model

Execute a single kernel with multiple threads

Implicit Parallelism

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

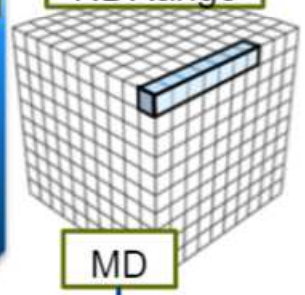
Data Parallelism (SPMD)

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

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

kernel

NDRange



SP

MD

Queue

Device

Array Processor (GPU)

Pipeline Processor (FPGA)

# Task Parallel Execution Model

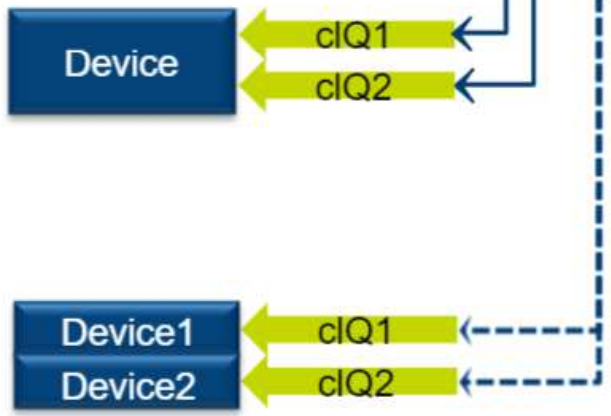
Execute multiple kernels in parallel

Implicit Parallelism

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

Task Parallelism (SMT)

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

# Setting Up the Device at the Host

Setting up your OpenCL™ platform using the host API

# OpenCL™ Device vs CUDA* Devices

- In CUDA* C, after `cudaSetDevice` is called, all CUDA * API commands go to that GPU
- OpenCL™ devices are explicitly associated with contexts and command queues where OpenCL API commands are applied
  - Designed to concurrently and easily work with multiple devices

# Device Execution Environment Setup

- 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**

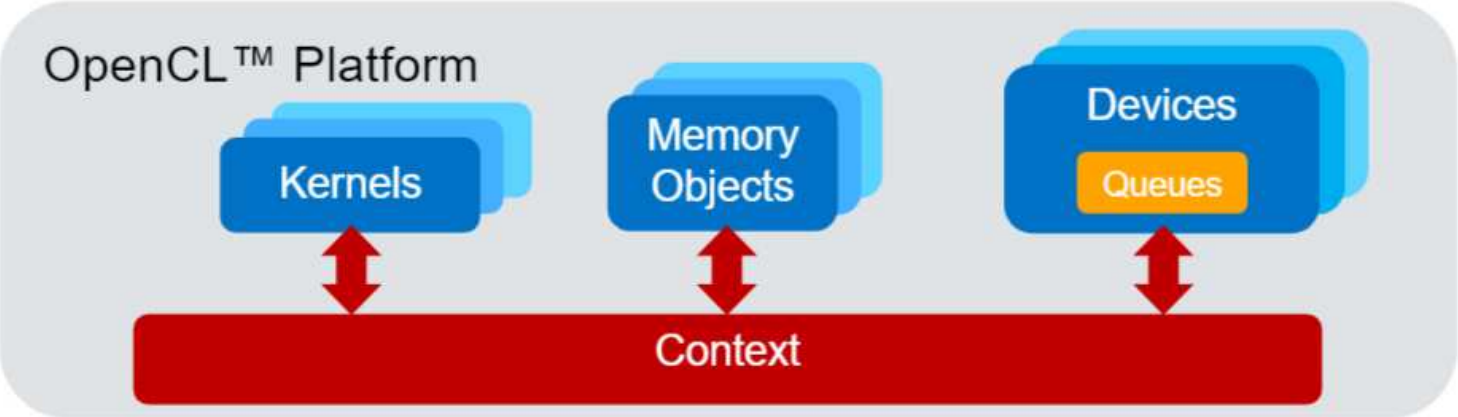
1. Query platforms
2. Query devices
3. Create a context for the devices

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

# Context

## Environment within which kernels execute

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



# 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

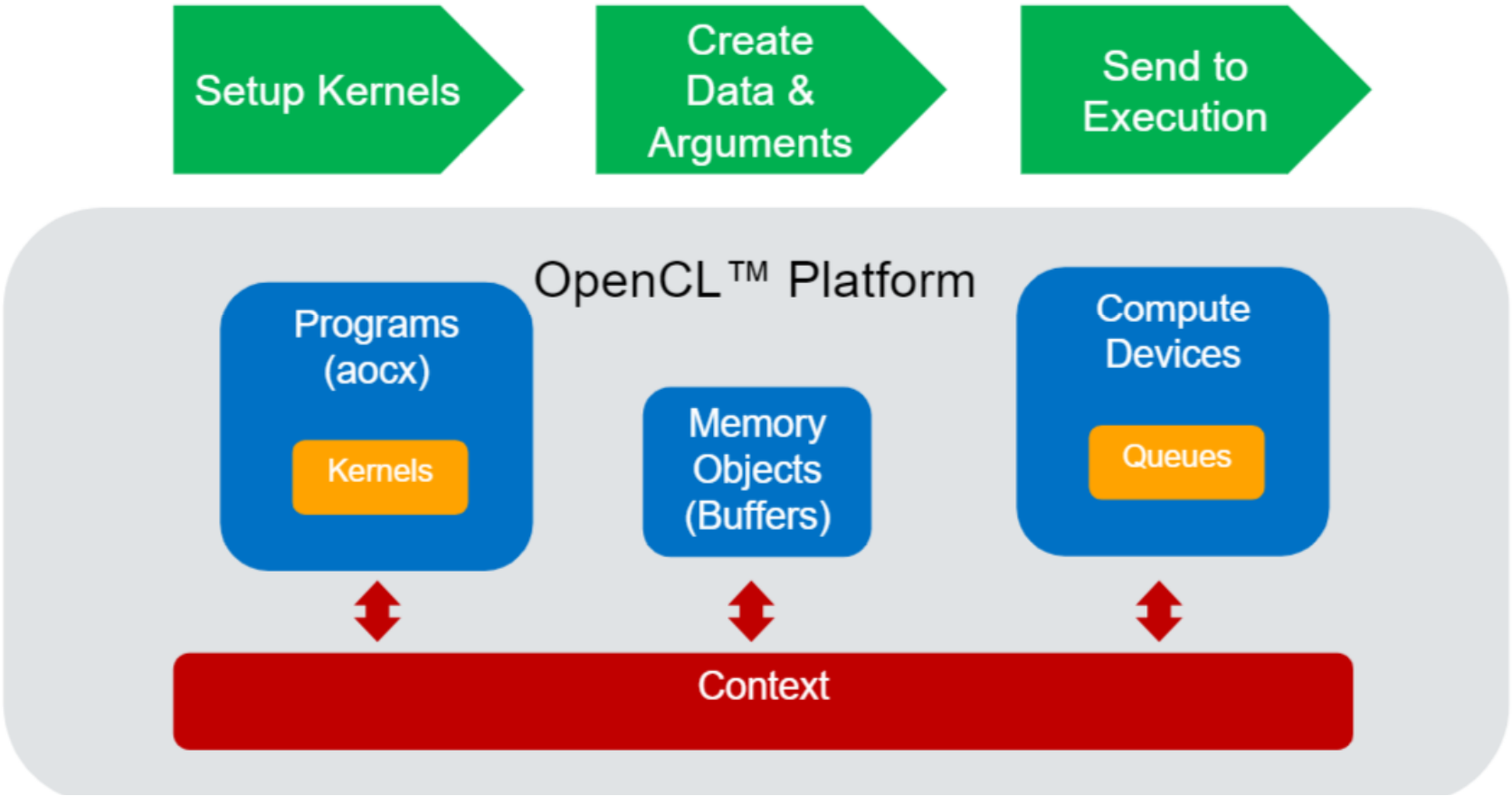
# Example Platform Layer Code

```
//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);
```

# OpenCL™ Host-side Execution

Using the OpenCL host API to control an OpenCL device during runtime

# OpenCL™ Execution Flow



# 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
- Unlike CUDA*, having a program object allows OpenCL™ to be flexible with how programs are compiled by different vendors even within the same host

# Programs and Kernels

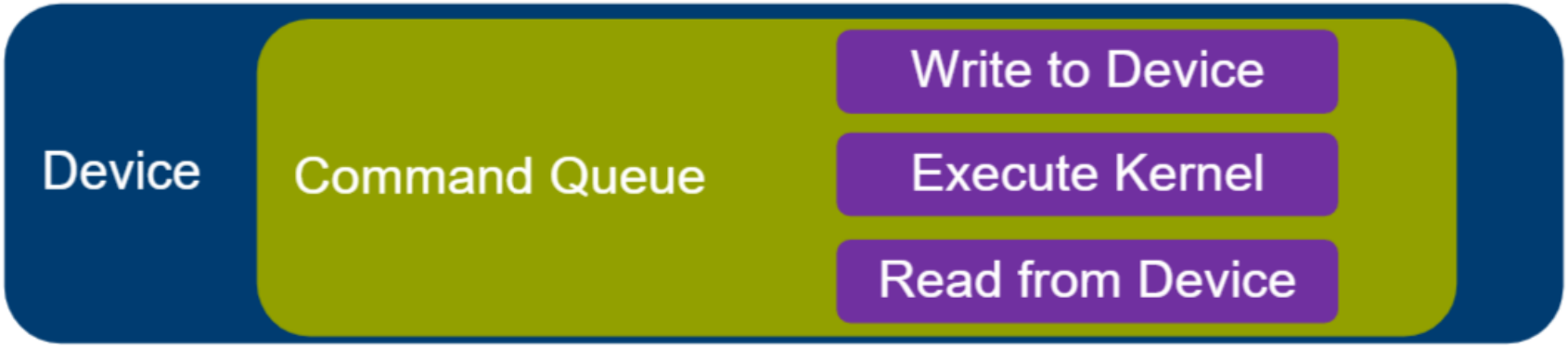
## Program – collection of kernels

- 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 `cl::CommandQueue::enqueue…` method

# 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



# Command Queue vs CUDA* Streams

## OpenCL™ command queue is similar to CUDA* streams

- Differences
  - No default command queue
    - Always have to create at least one
  - Allows operations in different queues to be synchronized using events
  - OpenCL allows out-of-order execution for operations on the same queue
    - Intel® FPGA does not

# Host / Device Physical Memory Space

- The host and the device each have their 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

# Kernel Execution Complete Example

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

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

    // 3. Create a command queue
   cl::CommandQueue myqueue(mycontext, mydevlist[0]);

   // 4. 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);
```

# Kernel Execution Complete Example Cont.

```
   …
    cl_float c_host = 10.8;

   // 5. Set up the kernel argument list
   err = mykernel.setArg(0, a_device);
   err = mykernel.setArg(1, c_host);
   err = mykernel.setArg(2, NUM_ELEMENTS);

   // 6. Launch the kernel
   err = myqueue.enqueueTask(mykernel);

   // 7. Transfer result buffer back
   err = myqueue.enqueueReadBuffer(a_device, CL_TRUE, 0, NUM_ELEMENTS*sizeof(cl_float), a_host);
}
```

# Kernel Execution Compared to CUDA*

- In CUDA* C, kernel functions are launched similar to a C function call
  - Pass in pointers created with `cudaMalloc` directly
  - No need to setup arguments individually
  - No need to deal with program and kernel objects

```
mycudakernel<<<1,1>>>(a_device, b_device, c_device);
```

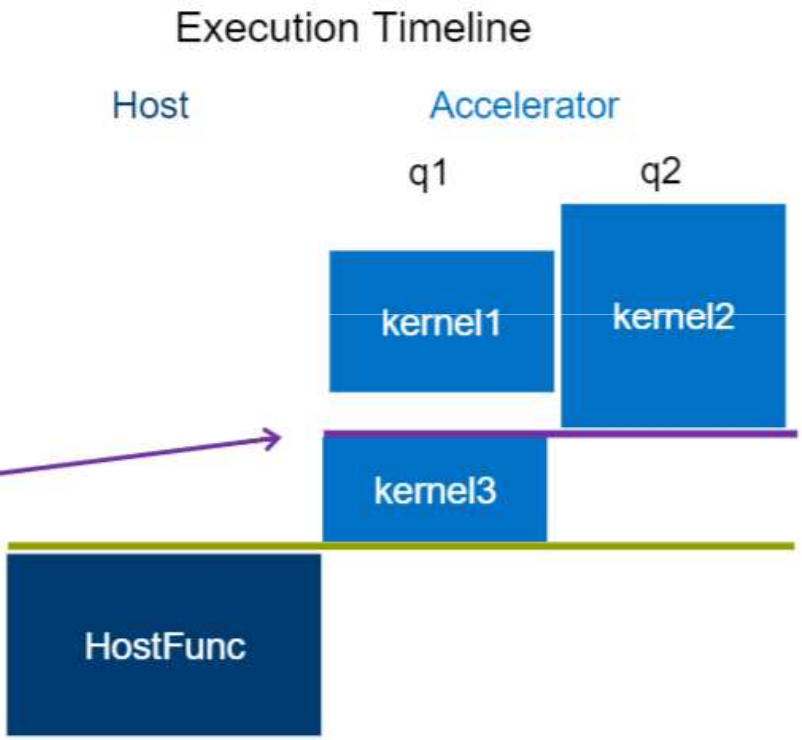
- Drawbacks
  - Not as flexible regarding how kernels are compiled and run

# Host and Kernel: Need for Synchronization

- 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 manages synchronization among device tasks
  - In additional to memory management and error handling tasks
- OpenCL allows more flexible and powerful synchronization options compared to CUDA*

# 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();
```



# OpenCL™ Kernels

## Writing code that will execute on the FPGA

# 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`
- Kernel language derived from ISO C99 with certain restrictions

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

# Compared to CUDA* kernels


- Denoted by the `__kernel` keyword
- Returns `void`
- Compiled separately from the host
  - Sometimes the host is the compiler
  - Separate host compiler needed
  - Compiled with the `aoc` compiler for FPGAs
- Pointer arguments needs to be qualified


- Denoted by the `__global__` keyword
- Returns `void`
- Compiled with the host code together at the top level by NVCC
  - Separate host compiler called by NVCC
- Pointer arguments are global

# OpenCL™ Kernel Statements

- C Operators
  - -, +, *, %, <<, ?:, &, &&, ~, !, ++, ==, etc.
- Math functions
  - `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™ 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

# OpenCL™ Kernel Restrictions

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

# NDRange Kernels

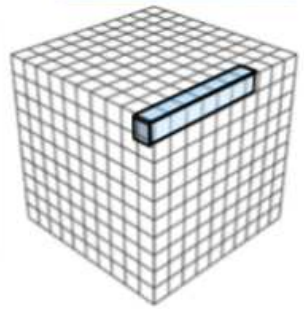
Execute an OpenCL™ kernel across multiple data-parallel threads

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

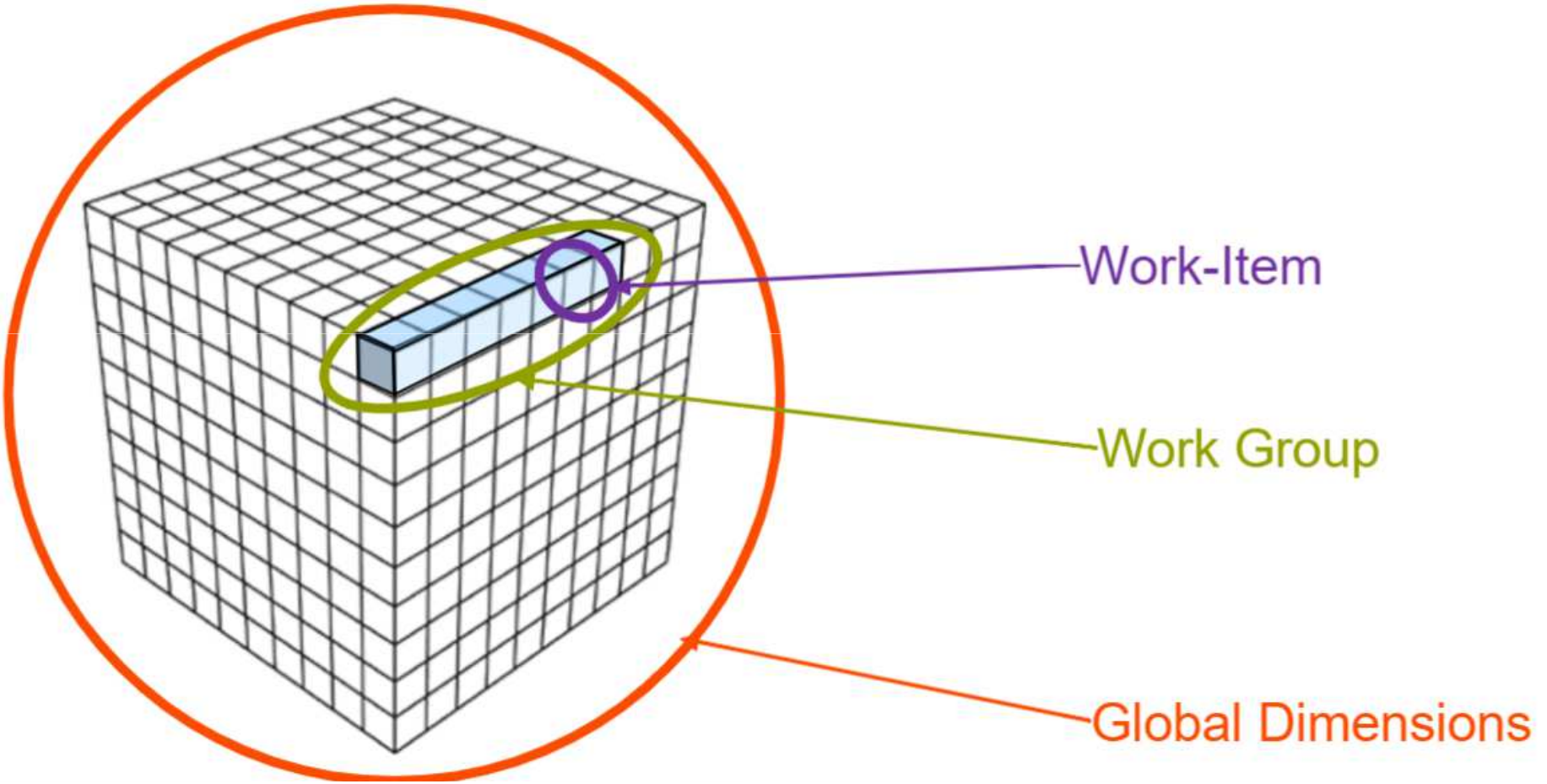

Kernel

Work-Items

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



# Work to be Done by NDRange Kernel



# OpenCL™ vs CUDA* Terminology

| OpenCL™ | CUDA* |
|---|---|
| Work-item | Thread |
| Workgroup | Thread block |
| Workgroups in a kernel launch | Grid (blocks in a kernel launch) |
| Compute Unit | GPU (or Streaming) Multiprocessor |

# 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=myqueue.enqueueNDRangeKernel(3dkernel, cl::NullRange, cl::NDRange(512,512,512),
                                 cl::NDRange(16,8,2));
```

# 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];
}
```

# Identifying Work-Items In the Kernel

OpenCL™ kernels have functions to identify the current work-item

- `get_global_id(dim)`
  - Index of work-item in the global space
- `get_local_id(dim)`
  - Index of work-item within workgroup
- `get_group_id(dim)`
  - Index of current workgroup

```
global_id =
     (group_id * local_size) +
     get_local_id(n)
```

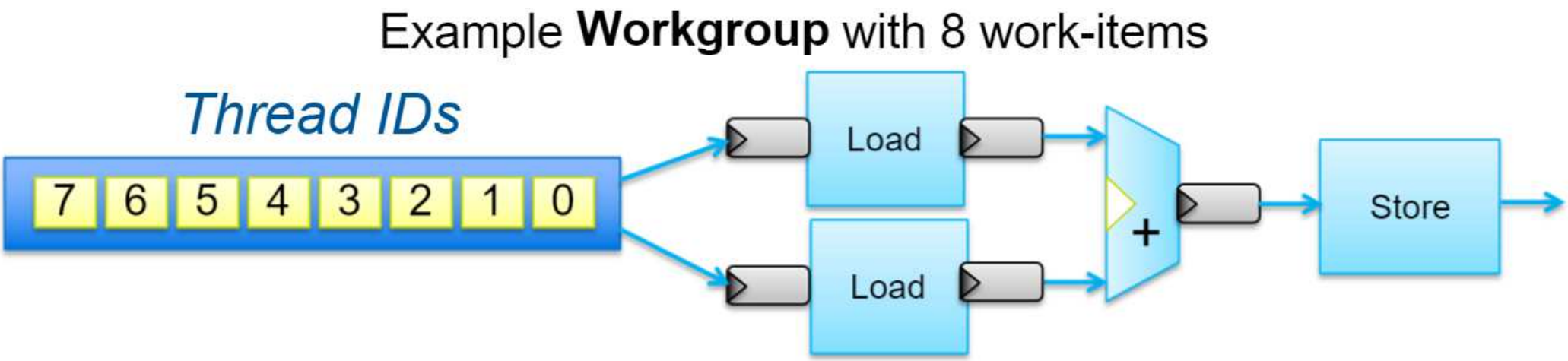
# Mapping NDRange Kernels to FPGAs

Better method involves creating a deeply pipelined representation of a kernel

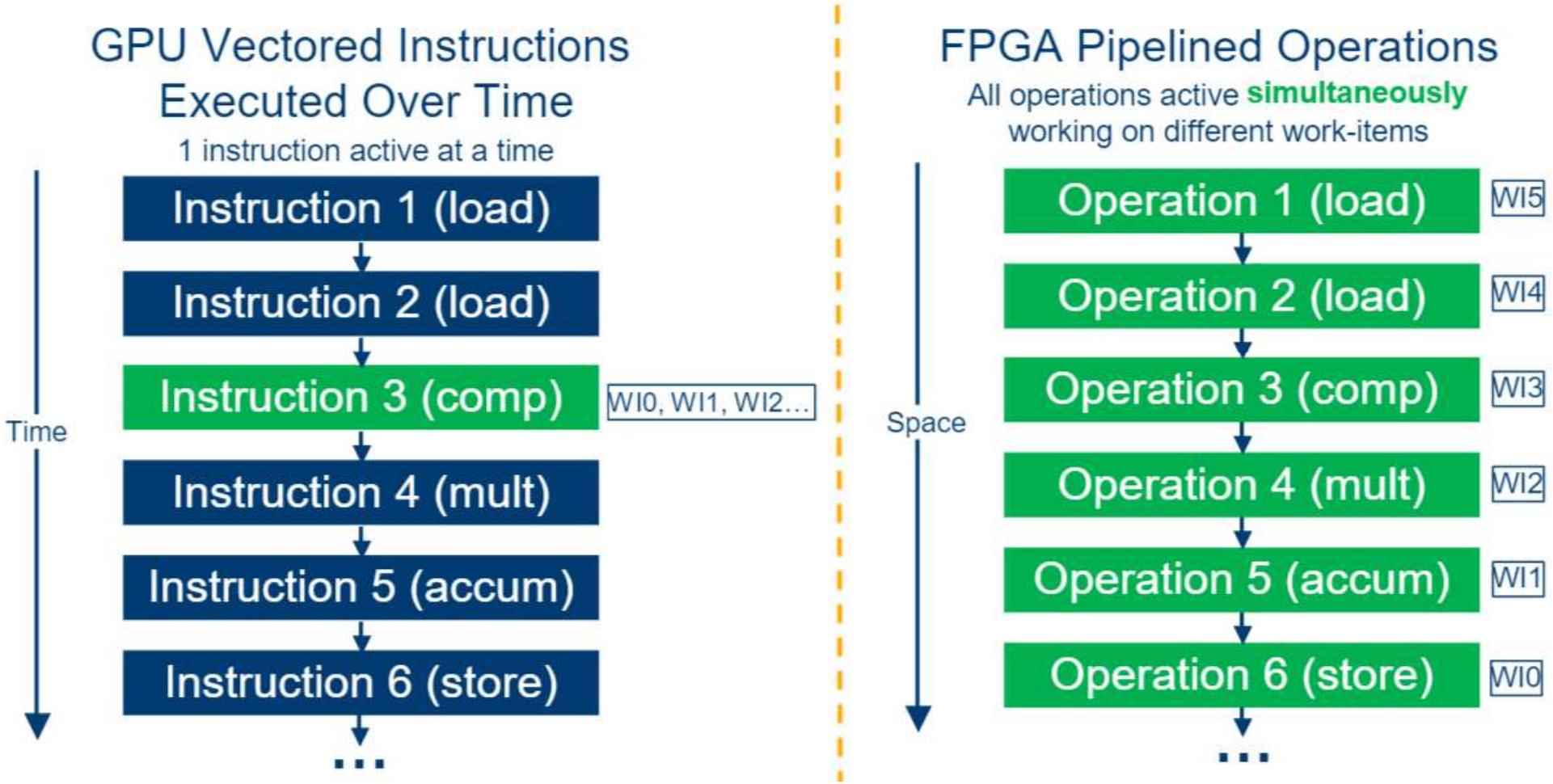
- On each clock cycle, we attempt to send in input data for a new thread
- A typical kernel pipeline will consist of **hundreds** of stages
  - Hundreds of work-items concurrently in the pipeline
- No need to worry about atomic operations since threads are not exactly in parallel
- Can duplicate the kernel compute engine if desired

# 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

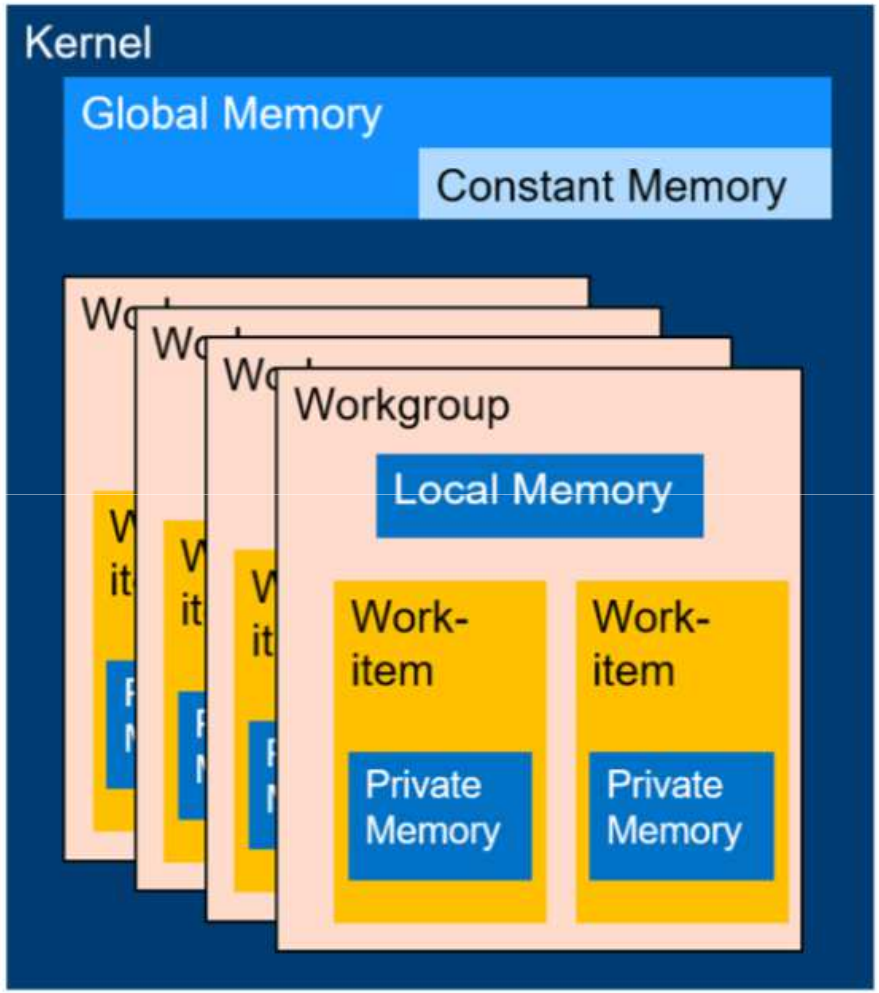


# GPUs vs FPGA Execution



# OpenCL™ Memory Model

- Private Memory – FPGA registers
  - Unique to work-item
- Local Memory – FPGA on-chip memory
  - Shared within workgroup
- Global/Constant Memory – DDR on FPGA board
  - Visible to all workgroups
- Host Memory
  - Visible to the host CPU



# CUDA* Equivalent Memory Qualifiers

| OpenCL™ | CUDA* C |
|---|---|
| Global memory<br>`__global` | Global memory<br>`__device__` |
| Constant memory<br>`__constant` | Constant memory<br>`__constant__` |
| Local memory<br>`__local` | Shared memory<br>`__shared__` |
| Private memory<br>`__private` | Local memory<br>`__local__`<br>Usage discouraged |