Accelerating Algorithms on FPGAs using OpenCL™
Class Agenda

FPGA Basics
OpenCL™ Platform and Host-side Software
Executing OpenCL Kernels
NDRange Kernels
OpenCL on Intel® FPGAs
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
- Fast time-to-market
- Many available I/O standards
Traditional FPGA Design Flow

**Design specification**

**Design entry/RTL coding**
- Behavioral or structural description of design
- Possibly with the help of high-level tools

```verilog
module dut(...);
always @(...) ...
endmodule;
```

**RTL functional simulation**
- Use 3rd party simulators
- Verify logic model & data flow

**Synthesis (Mapping)**
- Translate design into device-specific primitives
- Design optimized
- Quartus® 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

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

- **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
Class Agenda

FPGA Basics

OpenCL™ Platform and Host-side Software

Executing OpenCL Kernels

NDRange Kernels

OpenCL on Intel® FPGAs
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](http://www.khronos.org)
Intel® FPGA SDK for OpenCL™ Usage

**Intel® FPGA SDK for OpenCL™**

- **OpenCL Host Program**
  - Standard C Compiler
  - Executable File
- **OpenCL Kernels**
  - Offline Compiler (OpenCL Kernel Compiler)
  - Binary Programming File

Intel FPGA OpenCL Libraries
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**
Intel® FPGA OpenCL™ Device

Each device is made of many independent compute units

- Each compute unit custom built from kernel code
Data Parallel Execution Model

Execute a single kernel with multiple threads

Implicit Parallelism

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

Data Parallelism (SPMD)

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

clEnqueueWriteBuffer(clQ,x,...)
clEnqueueNDrangeKernel(clQ,_foo,...)
clEnqueueReadBuffer(clQ,u,...)

NDRange

MD

Queue

Device

Array Processor (GPU)
Pipeline Processor (FPGA)
Task Parallel Execution Model

Execute multiple kernels in parallel

Implicit Parallelism

\[ u = \text{foo}(x); \]
\[ y = \text{bar}(x); \]

Task Parallelism (SMT)

\[
\text{clEnqueueNDRangeKernel(clQ1, cl\_foo, \ldots)} \\
\text{clEnqueueNDRangeKernel(clQ2, cl\_bar, \ldots)}
\]
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 also available (*cl.hpp*)
  - Wrapper around the C API
  - Uses classes and C++ standard library containers
  - Simpler
  - More error checking capabilities
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 Layer APIs Called to Setup Environment (C++)

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 (C++)

```cpp
// 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);
```
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
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

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
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`
- Kernel language derived from ISO C99 with certain restrictions
  - Supports C operators, `math.h` operations, and flow control constructs
  - Supports C data types, vector data types, and structs

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

```c
// kernel.cl
__kernel void KernelName(...)
{
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```
FPGA Architecture for OpenCL™ Implementation

- Processor
- Host Interface
- External Memory Controller & PHY
- External Memory Controller & PHY
- DDR
- Precompiled periphery (BSP)

Custom Built Kernel System

Global Memory Interconnect

On-Chip Memory

Kernel Pipeline

Kernel Pipeline

Local Memory Interconnect
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]
```

Custom datapath

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

Setup Kernels

Create Data & Arguments

Send to Execution

OpenCL™ Platform

Programs (aocx)
- Kernels

Memory Objects (Buffers)

Compute Devices
- Queues

Context
Kernel Execution Complete Example (C++)

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

// 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);
}
Exercise 2

Writing a Simple Kernel
Class Agenda

Heterogeneous Parallel Computing
OpenCL™ Platform and Host-side Software
Executing OpenCL Kernels
NDRange Kernels
OpenCL on Intel® FPGAs
Data Parallelism

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

\[
\begin{align*}
\text{data}[0] & \rightarrow f() & \rightarrow \text{result}[0] \\
\text{data}[1] & \rightarrow f() & \rightarrow \text{result}[1] \\
& \cdots & \\
\text{data}[n] & \rightarrow f() & \rightarrow \text{result}[n]
\end{align*}
\]
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

```c
__kernel void fookernel(__global float *x, ...)
{
    int i = get_global_id(0);
    u[i] = foo(x[i]);
}
```
Multi-Threaded Execution

- Work-Items
- Compute Unit A (1)
  - On-Chip Memory
- Compute Unit A (2)
  - On-Chip Memory
- Compute Unit B (1)
  - On-Chip Memory
- Compute Unit B (2)
  - On-Chip Memory
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

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

// N work-items to be created
__kernel void vecadd(__global int *C,
                     __global int *A,
                     __global int *B)
{
    int tid = get_global_id(0);
}
Kernel Launch - Code Example

//1D C++ 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(1dkernel, cl::NullRange,
   cl::NDRange(512,512,512),
   cl::NDRange(16,8,2));
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

\[
globalid(n) = \text{groupid}(n) \times \text{localsize}(n) + \text{localid}(n)
\]
## Work-item Identification Example

### One-dimension launch

<table>
<thead>
<tr>
<th>Work-items</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_global_id(0)</code></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td><code>get_local_id(0)</code></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><code>get_group_id(0)</code></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- `get_work_dim() = 1`,
- `get_global_size(0) = 12`,
- `get_local_size(0) = 4`,
- `get_num_groups(0) = 3`
Work-Item Identification Continued

One-dimension launch

```
__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

\[ \text{get}\_work\_dim()=1, \text{get}\_global\_size(0)=12, \text{get}\_local\_size(0)=4, \text{get}\_num\_groups(0)=3} \]

- \( x[0-11] \): \( 8 \quad 8 \quad 8 \quad 8 \quad 8 \quad 8 \quad 8 \quad 8 \quad 8 \quad 8 \quad 8 \quad 8 \)
- \( x[0-11] \): \( 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 2 \quad 2 \quad 2 \quad 2 \)
- \( x[0-11] \): \( 0 \quad 1 \quad 2 \quad 3 \quad 0 \quad 1 \quad 2 \quad 3 \quad 0 \quad 1 \quad 2 \quad 3 \)
- \( x[0-11] \): \( 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \)
Pipeline execution

- By default AOC implements a deep pipeline for each kernel
- On each cycle the portions of the pipeline are processing different threads

Example **Workgroup** with 8 work-items

**Thread IDs**

```
7 7 7 7 7 6 5 4
```

```
3 2 1 0
3 2
```
OpenCL™ Memory Model

- **Private Memory (On-chip memory)**
  - Unique to work-item

- **Local Memory (On-chip memory)**
  - Shared within workgroup

- **Global/Constant Memory (Off-chip memory)**
  - Visible to all workgroups

- **Host Memory**
  - Visible to the host CPU
  - May be shared with device in unique cases
Memory Qualifier Syntax Example

```c
__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], ...)
    ...
}
```
FPGA Memory Implementation Advantage

- Tailored global memory interconnect
- Ability to partition or interleave global memory space
  - Partitioning allows allocation of variables into specific banks (controllers)
- Ability to assign variable to different types of global memory
- Custom Local/Private Memory Systems!
  - Each individual array/variable gets its own system
  - Custom banked, coalesced, replicated, double-pumped for never-stall accesses
  - Conversions to shift-registers, registers, ROMs, when appropriate
- Takes advantage of on-chip memory bandwidth of FPGAs
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
  - SDK Contents
  - AOCL Utility
- Kernel Compilation
- Host Compilation
- Runtime
- Debug Tools
- FPGA-specific Features
Intel® FPGA SDK Overview

Intel® FPGA SDK for OpenCL™

- Intel FPGA OpenCL™ Libraries
- OpenCL™ Host Program
- Standard C Compiler
- Executable File
- Offline Compiler (OpenCL™ Kernel Compiler)
- Binary Programming File
- Intel FPGA OpenCL™ Kernels
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
  - Linked against when compiling the host using a generic C compiler

- AOCL Utility
  - Perform various tasks related to the board, drivers, and compile process

- Software Requirements
  - Intel Quartus® Prime tool with the appropriate devices
  - Generic C compiler for the host program
Offline Kernel Compiler (aoc)

- Compiles kernels for a specific board defined by a board support package
- Generates aocx and aoco 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.

```
aoc --board <my board> <my kernel file>
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--help or -h</td>
<td>Help for the tool</td>
</tr>
<tr>
<td>-c</td>
<td>Creates .aoco object file and sets up a Quartus® Prime hardware design project</td>
</tr>
<tr>
<td>--board &lt;board name&gt;</td>
<td>Compile for the specified board</td>
</tr>
<tr>
<td>--list-boards</td>
<td>Prints a list of available boards</td>
</tr>
</tbody>
</table>
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
Kernel Development Flow and Tools

1. Modify kernel.cl
2. Emulator (secs)
   - Functional bugs?
   - Loop inefficiencies?
   - Undesired hardware structure?
   - Sub-optimal memory interconnect?
3. HTML Report (~1 min)
   - Loop Optimization Report
   - Detailed Area Report
   - Architectural Viewer
4. Profiler (full compile time)
5. Done

Poor performance?
Enable kernel functional debug on x86 systems

- Quickly generate x86 executables that represent the kernel

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

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

Static report showing optimization, area, and architectural information

- Automatically generated with the object file \( \texttt{aoc -c} \)
  - Located in \(<\text{kernel file folder}>\text\textbackslash reports\text\textbackslash report.html\)

- 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 in single work-item kernels
  - Shows loop carried dependencies and bottlenecks
- 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>
```

```c
kernel void accel(...) {
  ...
  gid = get_global_id(0);
  out[gid] = a[gid] + b[gid];
  ...
}
```
Profiler Reports

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

- Launching kernels with global size of (1,1,1)
- NDRRange execution may not be suitable for certain situations
  - Data parallelism isn’t always easy to extract
    - 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
- Loops in single work-item kernels automatically parallelized by the offline Compiler
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 Example

No Loop Pipelining

No Overlap of Iterations!

With Loop Pipelining

Finishes Faster because Iterations Are Overlapped

Looks like multi-threaded execution!
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

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

kernel void kernel1(write_only pipe uint p1) {
  ...  
  iData = read_channel_altera(c0);
  ...  
  write_pipe(p1, &oData);
}

kernel void kernel2(read_only pipe uint p1) {
  ...  
  read_pipe(p1, &value);
}
```
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)
Intel® FPGA-Specific Features

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

- FPGA Basics
- OpenCL™ Platform and Host-side Software
  - Platform and Runtime Layer API
- Executing OpenCL Kernels
  - Writing and Launching Kernels
- NDRange Kernels
  - Multi-threaded Kernels and associated memory model
- OpenCL on Intel® FPGAs
Exercise 4

*Examining Kernel Compilation Reports*
OpenCL™ References

- Intel® FPGA OpenCL collateral [www.altera.com/OpenCL](http://www.altera.com/OpenCL)
  - White papers
  - Demos and Design Examples
  - 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

*Other names and brands may be claimed as the property of others
Many Ways to Learn

Videos
Free
Always available
~4 minutes long
YouTube* videos

Online Training
Free
Always available
~30 minutes long
>200 topics
English, Chinese, Japanese

Virtual Classes
Live over Webex
Ask questions to Intel FPGA expert
Hands on labs
Taught in ½ day sessions
Class schedules at
www.altera.com/training

Instructor-led Training
In-person
Ask questions to Intel FPGA expert
Hands on labs
1 day long
Class schedules at
www.altera.com/training
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 Corporation