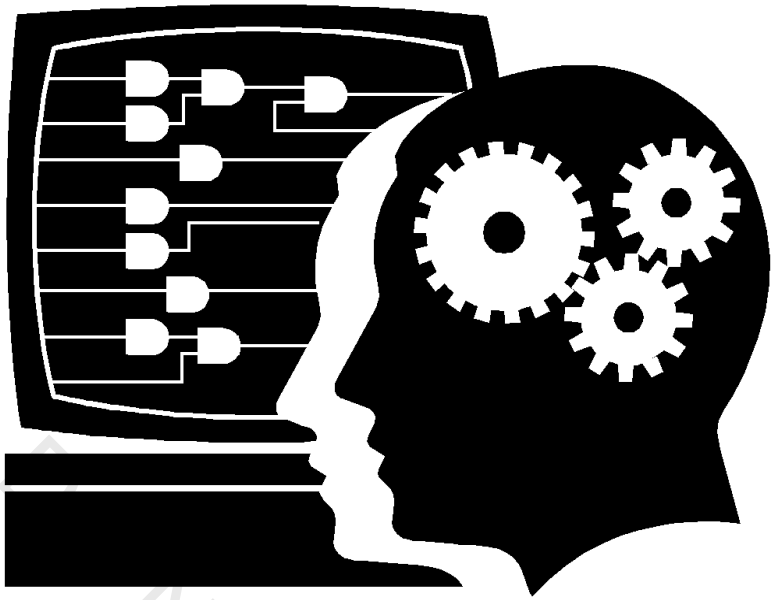


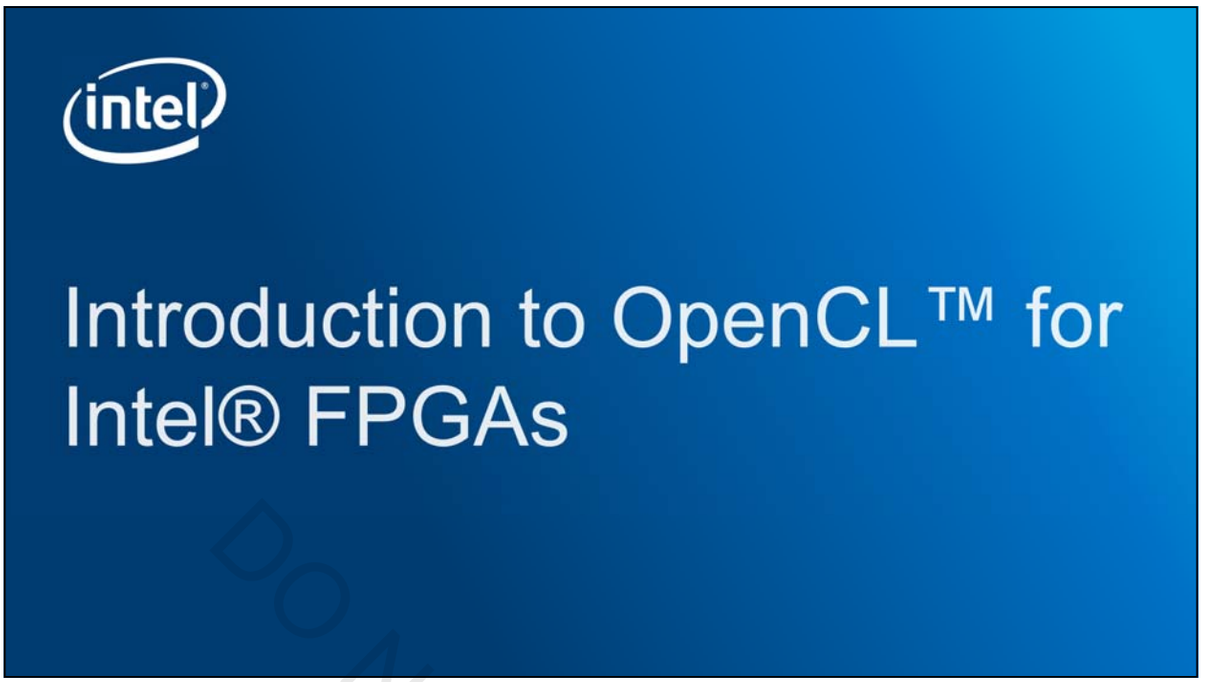
Customer Training

Introduction to OpenCL for Intel FPGAs

A-MNL-OPNCL-16-1-v1

http://www.altera.com/customertraining/ILT/P-CSTN-OPNCL-16-1-v1.zip

Table of Contents **Introduction to OpenCL for Intel FPGAs**	**Page Numbers**



Agenda and Objectives

Describe high-level parallel computing concepts and challenges

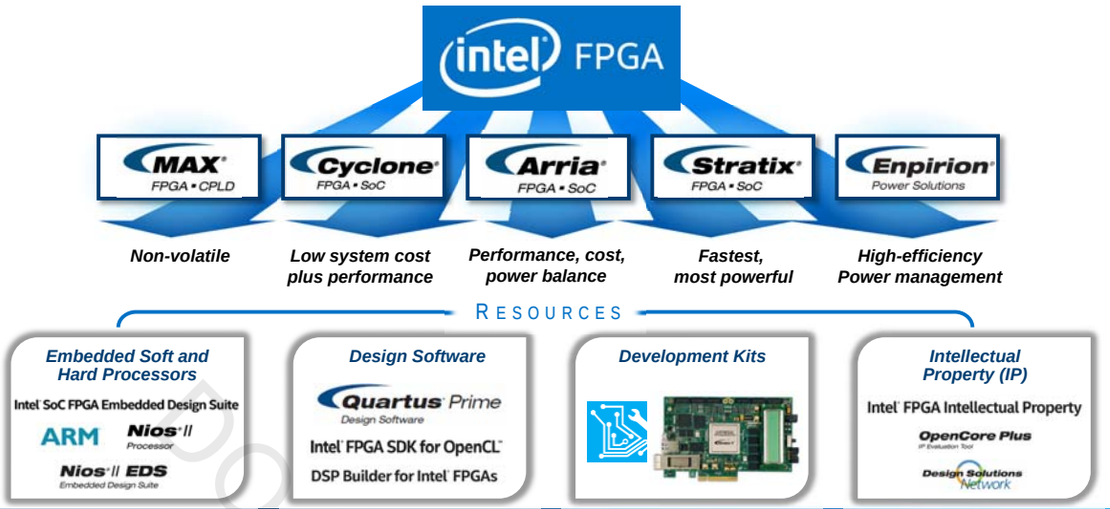
Understand the advantages of using OpenCL™ with Intel® FPGAs

Know the basics of the OpenCL standard

Write simple OpenCL programs

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

Product Portfolio Overview



Programmable Solutions Group | Intel Confidential | 3

Class Agenda

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

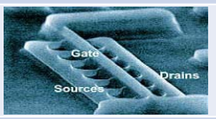
Programmable Solutions Group | Intel Confidential | 4

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 differently levels that are higher than instruction-level parallelism
- Likely more effective than compiler/processor extracted parallelism

Types of Parallelism

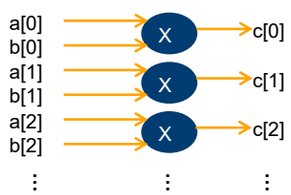
- Data Parallelism
  - Input data separated and sent to parallel resources, results recombined
  - Scatter-gather
- Task Parallelism
  - Decompose problem into sub-problems that run well on available compute resources
  - Divide-and-conquer
- Pipeline Parallelism
  - Task parallelism where tasks have 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)

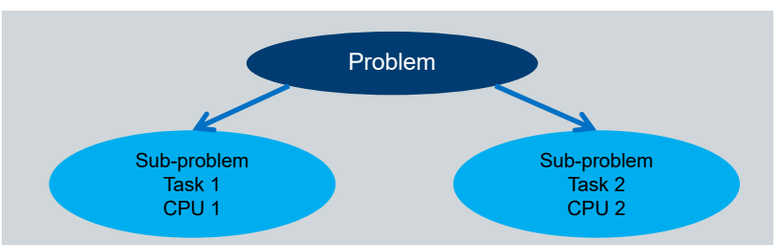
```
for (i = 0; i < N ; i++)
     c[i] = a[i] * b[i]
```



Task Parallelism

Decompose problem into sub-problems (tasks)

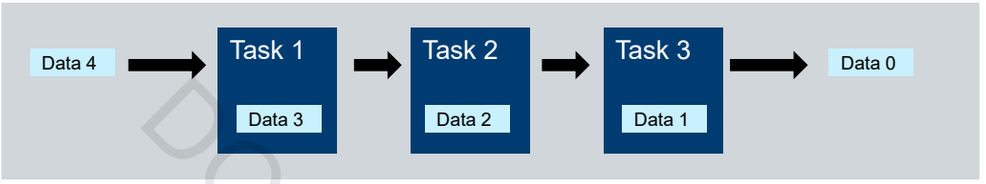
- Tasks operate on same or different data
- Example: Multi-CPU system where each CPU execute a different thread
- A.K.A. Simultaneous Multithreading (SMT), Thread/Function Parallelism



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

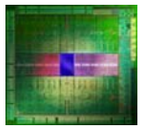
- Modern systems contain more than one kind of processor
- Applications exhibit different behaviors
  - Control (Searching, parsing, etc…)
  - Data intensive (Image processing, data mining, etc…)
  - Compute intensive (Iterative methods, financial modeling, etc…)
- Gain performance by utilizing specialized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

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

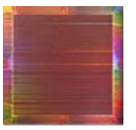

FPGAs

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

Traditional Approach to Heterogeneous Computing

- Write software for each software programmable architecture CPU, CPU, 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.



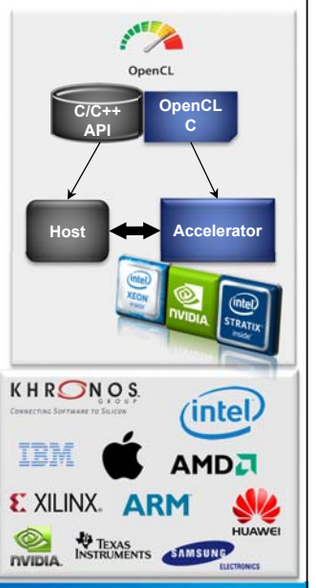
Programmers Dilemma

"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

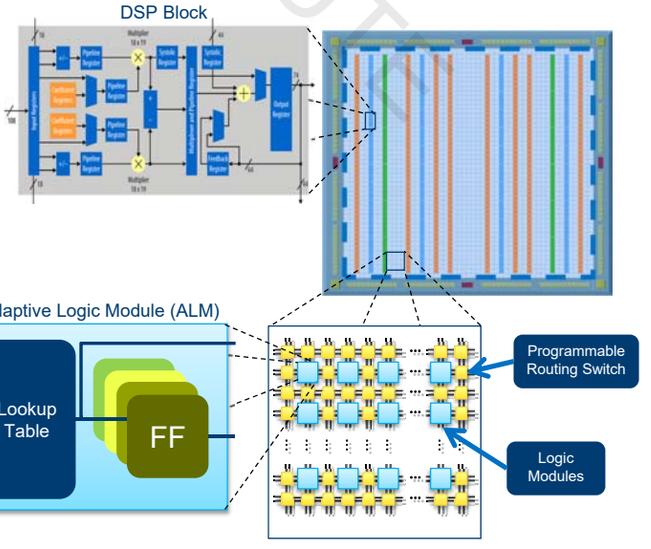
What is OpenCL™?

- 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

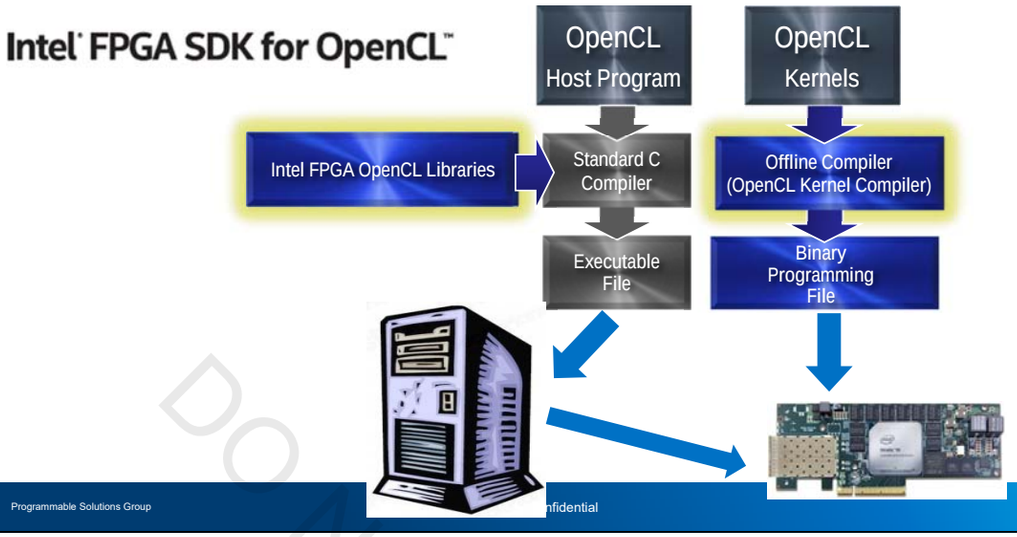


FPGA Architecture

- 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

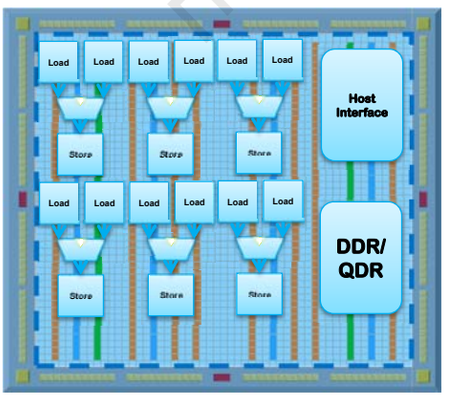


Intel® FPGA SDK for OpenCL™ Usage



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, GPGPU, 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

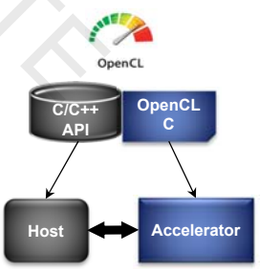
OpenCL™ Versions

- 1.0 (2008)
- 1.1 (2010)
  - Image data type
- 1.2 (2011)
  - Device Partitioning
  - Printf support
- 2.0 (2013)
  - Shared virtual memory
  - Pipes
- 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

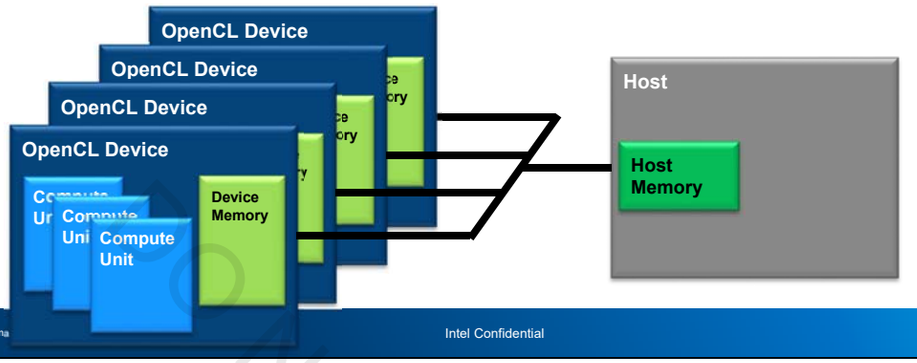
Two Sides of OpenCL™ Standard



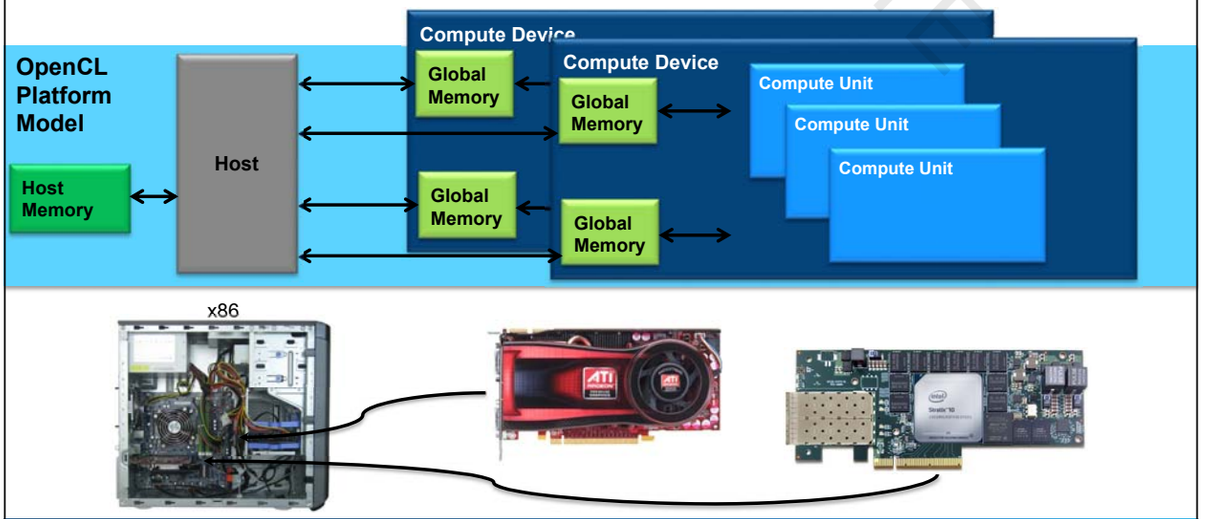
- Kernel Function
  - OpenCL C
  - Software that runs on accelerators (OpenCL devices)
  - Usually used for computationally intensive tasks
- Host Program
  - Software running conventional microprocessor
  - Supports efficient plumbing of complicated concurrent programs with low overhead
    - Through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

OpenCL Platform Model

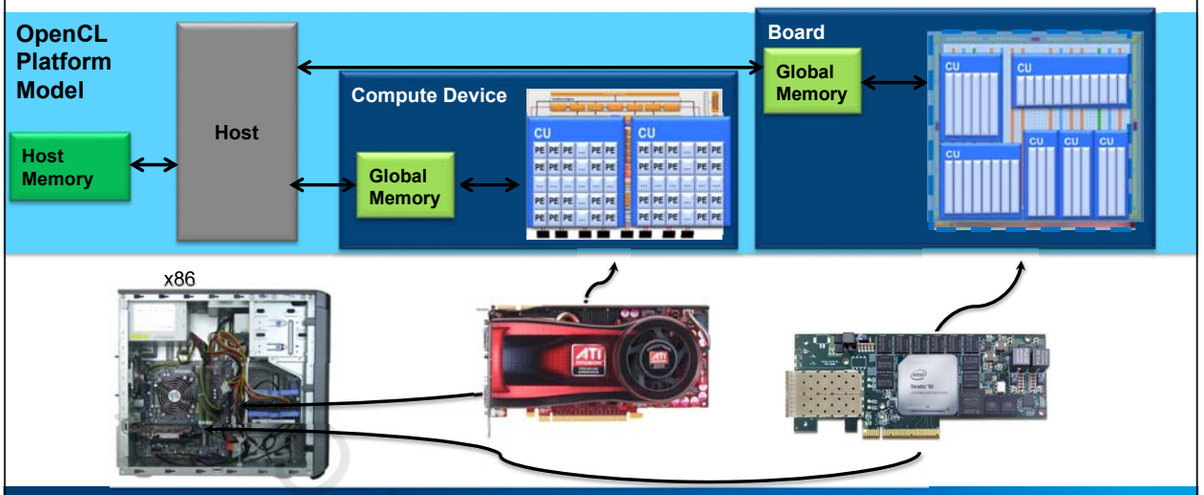
- One Host with one or more OpenCL Devices
  - Each Device is composed of one or more compute units
- Memory divided into Host Memory and various types of Device Memory



Heterogeneous Platform Model



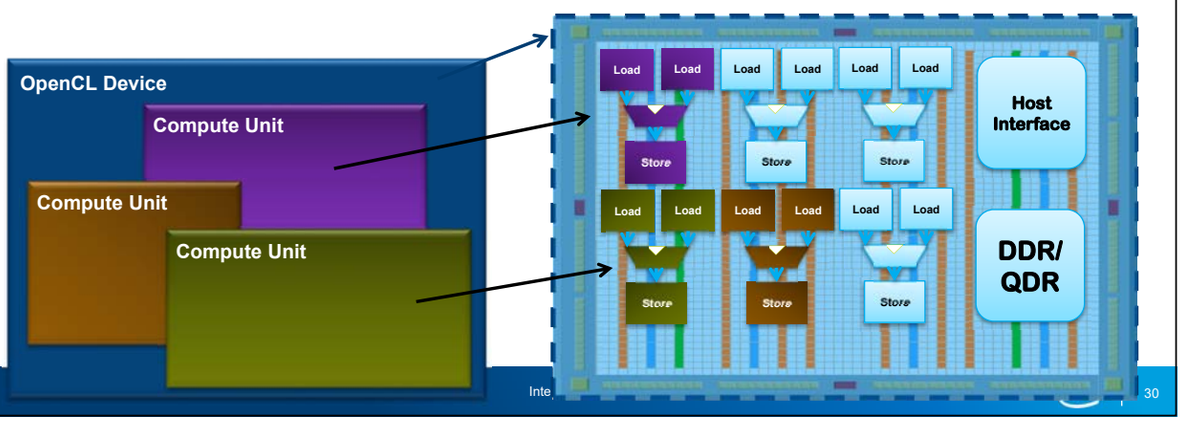
Heterogeneous Platform Model



Intel® FPGA OpenCL™ Device

Each device is made of many independent compute units

- Each compute unit custom built from kernel code



Data Parallel Execution Model

Execute a single kernel with multiple threads

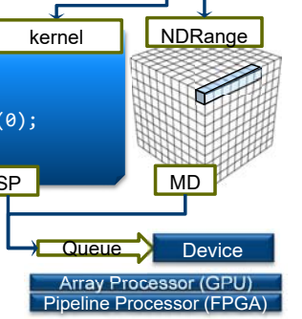
Implicit Parallelism

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

Data Parallelism (SPMD)

```
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]);
}
```



Task Parallel Execution Model

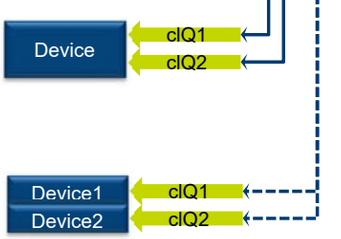
Execute multiple kernels in parallel

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

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
  - Wrapper that maps to the C API (`cl.hpp`)
  - No additional execution overhead
  - Much simpler

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

- 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 and select the platforms
2. Query and select the devices
3. Create a context for the devices

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

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

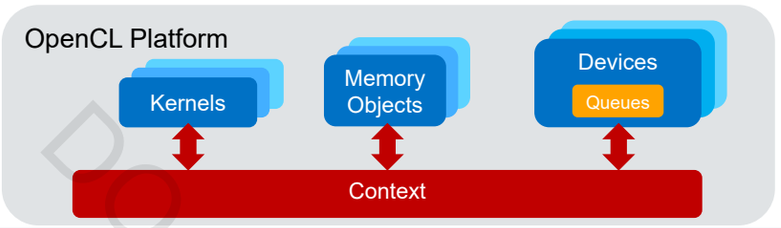
Returns a list of device IDs

Returns the total number of devices available

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



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 device
  - Number of devices in context, context properties, and reference count
  - Pass in `cl_context` and `cl_context_info` Enum type

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

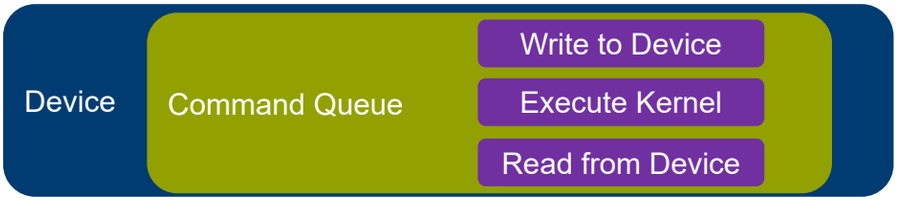
- Tasks
  - Memory management
    - Allocate/deallocate device memory
    - Read/write to the device
  - Run kernels on the device
  - Host/device synchronization

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

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

Valid context

A device associated with *context*

Returns the command queue

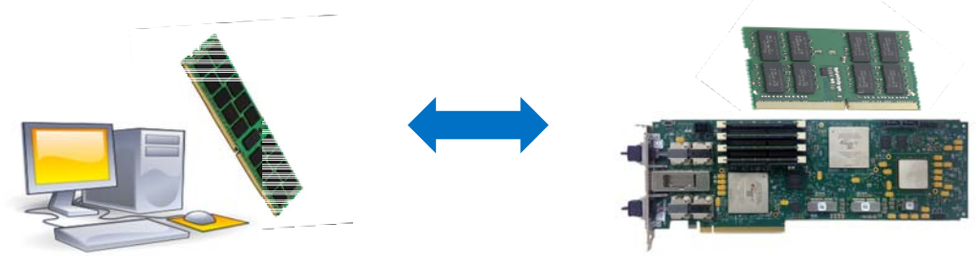
Error code

Queue properties
e.g. Turn on profiling

- Host will submit commands to the device through the command queue
  - Using *clEnqueue…* 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)
  - Images
    - Stores an image or array of images
    - Simplifies the process of representing and accessing images
    - Not discussed in this class

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

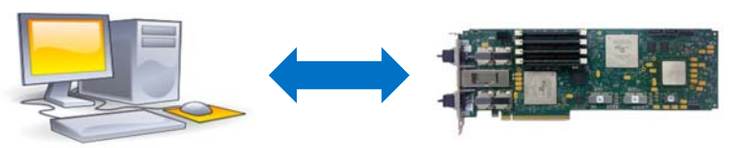
- Kernel access permissions (from the kernel perspective)
  - `CL_MEM_READ_WRITE` (default), `CL_MEM_WRITE_ONLY`, `CL_MEM_READ_ONLY`
- Host access permissions (from the host perspective)
  - `CL_MEM_HOST_WRITE_ONLY`, `CL_MEM_HOST_READ_ONLY`, `CL_MEM_HOST_NO_ACCESS`
- Host pointer options
  - `CL_MEM_COPY_HOST_PTR`
    - 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)
```


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

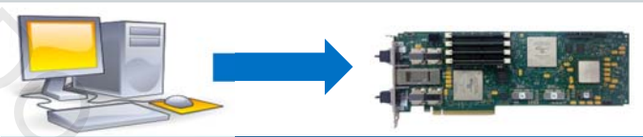


clEnqueueWriteBuffer

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_read,
                           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);
```


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
```

Test Your Knowledge

- What is the OpenCL mechanism that manages tasks for one device?
  a) Context
  b) Command Queue
  c) Kernel
  d) Platform
- How is device memory represented on the host?
  a) As pointers
  b) As linked lists
  c) As memory objects
  d) It can't be accessed from the host

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
  - `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];
}
```

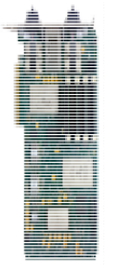
Compiling OpenCL™ Kernel to FPGAs

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 i = get_global_id(0);
   c[i] = a[i] + b[i];
}
```

aoc



Compile Kernels

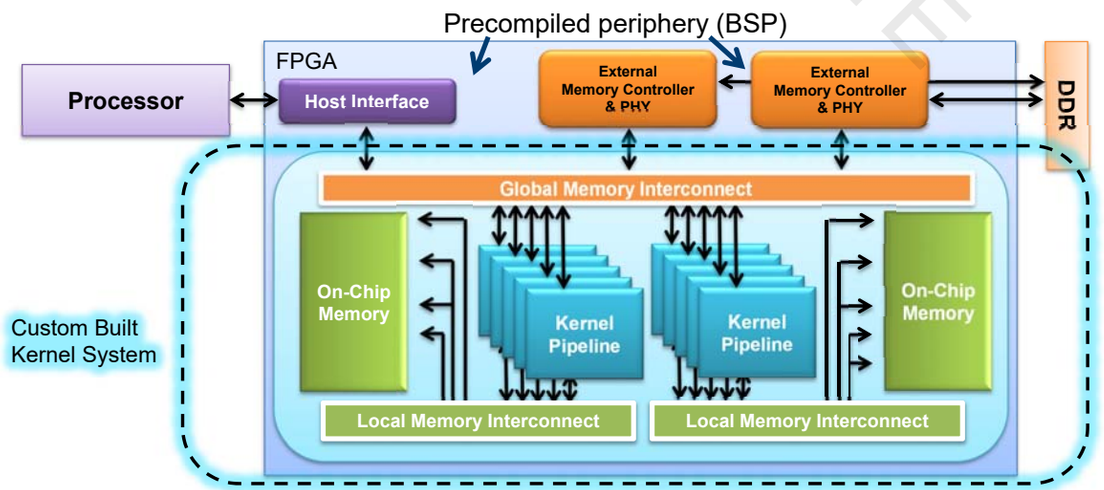
Run the Offline Compiler

- Set `AOCL_BOARD_PACKAGE_ROOT` environment variable to the path of the board support package
  - Usually `%ALTERAOCLSDKROOT%/board/<name_of_BSP>`
- `aoc --list-boards`
  - List available boards within the current board package
- `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
- <kernel file>.aocx
  - Kernel executable file used to program FPGA
- Inside <kernel file> folder
  - <kernel file>.log
    - Compile log including estimated resource usage, optimization report, and compile messages
  - <kernel file folder>\reports\report.html
    - Interactive HTML report
    - Static report showing optimization, detailed area, and architectural information
  - Quartus generated source and report files
    - Timing information, actual resource usage, etc.

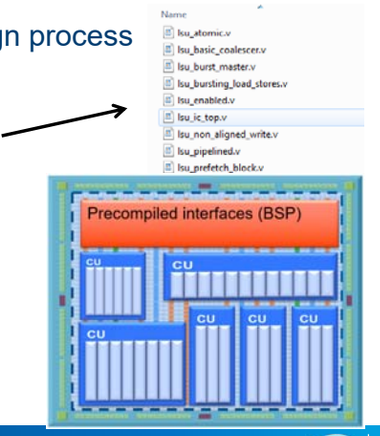
FPGA Architecture for OpenCL™ Implementation



OpenCL™ Kernels to Dataflow Circuits

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

- Gain the benefits of FPGAs without the length design process
- Implement C operators as circuits
  - HDL code located in <OpenCL 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



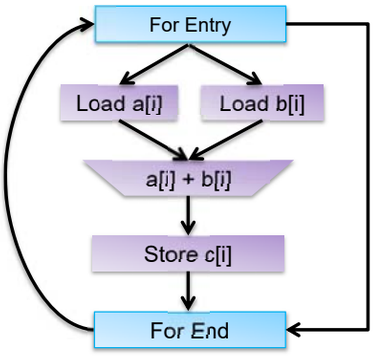
Compilation Example

Kernel compiled into dataflow circuit with flow control

- Include 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



Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

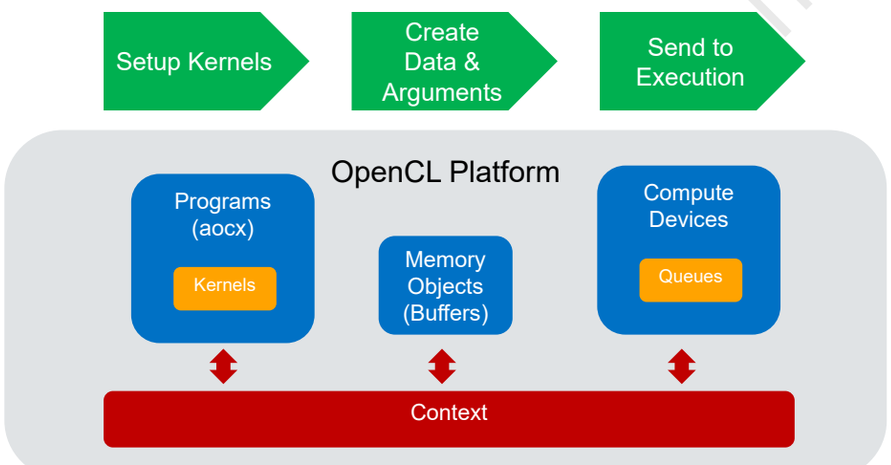
Executing OpenCL Kernels

- 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 `clEnqueue…` function

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 a Program

A program object (`cl_program`) contains one or more kernels (`cl_kernel`)

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

Creating Programs from Binary for FPGAs

```
cl_program clCreateProgramWithBinary(cl_context context,
                                     cl_uint num_devices,
                                     const cl_device_id *device_list,
                                     const size_t *lengths,
                                     const unsigned char **binaries,
                                     cl_int *binary_status,
                                     cl_int *errorcode_ret)
```

Valid context

Number of devices in *device_list*

Compatible devices

Program object

Lengths of binaries

aocx binary

Status of binary loading

Error code

- 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

```
cl_int clEnqueueTask(cl_command_queue command_queue,
                     cl_kernel kernel,
                     cl_uint num_events_in_wait_list,
                     const cl_event *event_wait_list,
                     cl_event *event)
```

Error code

Where kernel will be queued for execution

Kernel to be executed

Events used for synchronization. Discussed later

- `clEnqueueNDRangeKernel` discussed later

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.

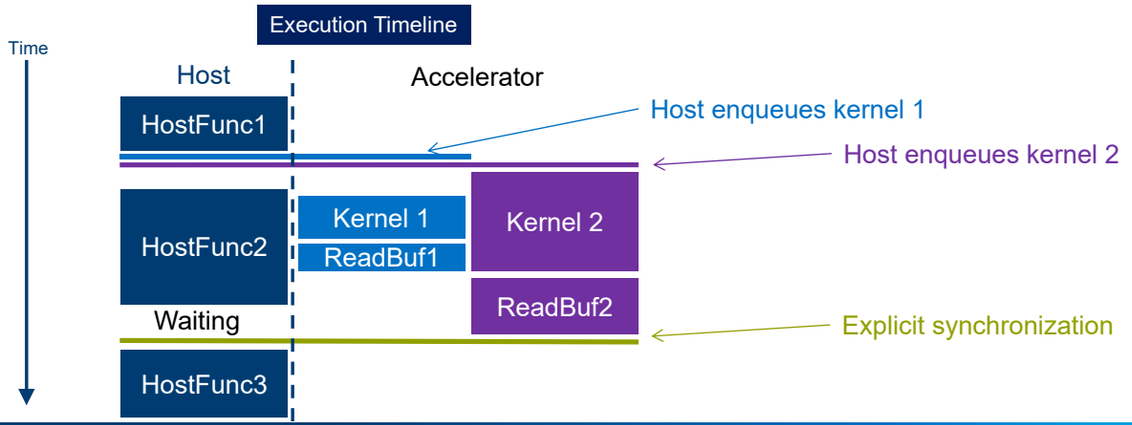
```
   …
   // 5. Launch the kernel
   err = clEnqueueTask( queue, kernel, 0, NULL, NULL);

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

Host and Kernel Execution

- Kernels execute on one or more OpenCL devices
- Host program executes on the host
- With `clEnqueue` 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

Asynchronous Kernel Execution

By default, host launches device but execution is not synchronized



Host-side Explicit Synchronization Point

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

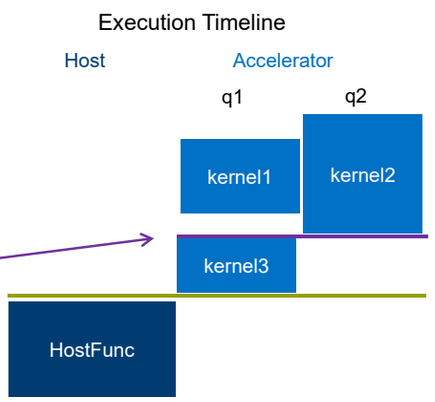
Event Dependencies

- Each `clEnqueue`
  - Can **depend on** an array of (previously created) `cl_events`
    - 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,        } Wait for these
                      const cl_event *event_wait_list,        } to finish
                      cl_event *event)                        } 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);
```

Test Your Knowledge

- What does the host need to do before launching a kernel?
  a) Create program
  b) Create kernel
  c) Setup kernel arguments
  d) All of the above

Exercise 2

Writing a Simple Kernel

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

Executing OpenCL Kernels

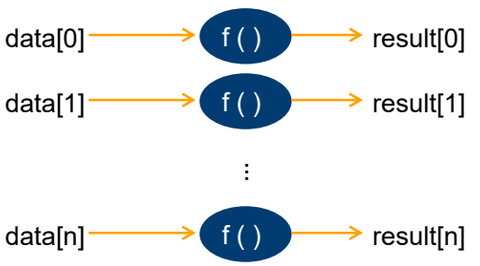
NDRange Kernels

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

OpenCL on Intel® FPGAs

Data Parallelism (Review)

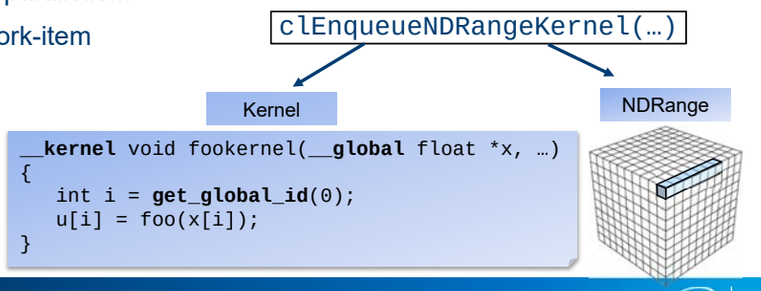
- Same operation applied to multiple, independent data concurrently
  - Data dependency hinders data parallelism



NDRange Kernels

Execute an OpenCL™ kernel across multiple data-parallel threads (NDRange)

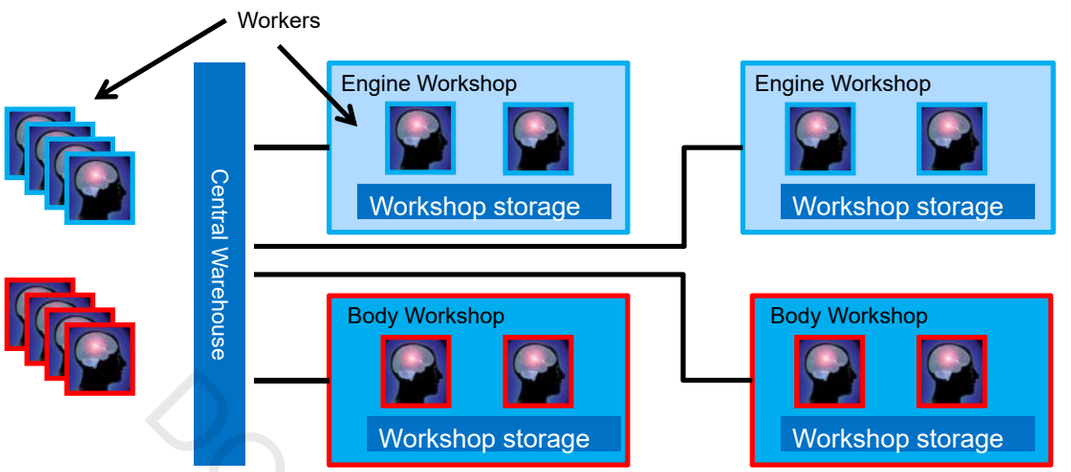
- "Traditional" OpenCL
- Executed in a single program (kernel) multiple data (NDRange) SPMD fashion
  - Explicitly declares data parallelism
  - Each thread called a work-item



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

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

Kernels

What to do on the assembly line

OpenCL™ C code written to run on OpenCL devices

- Kernels provide data parallelism 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

- 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];
}
```


NDRange

Total number of cars to build

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

NDRange Data Division Examples

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

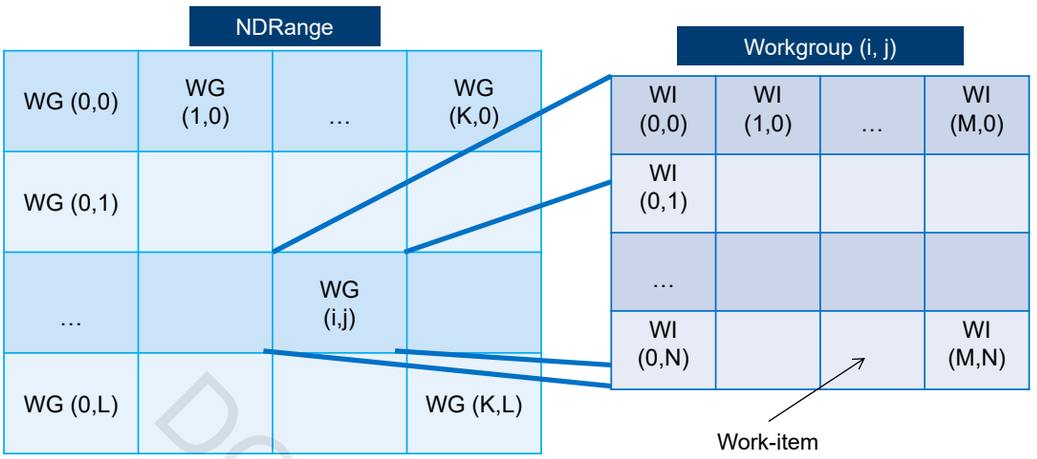
Workgroup

Group of workers able to communicate

Dividing work-items of an NDRange into smaller equally sized workgroups

- Local Dimension
- Same dimension as NDRange index space
  - 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 `clEnqueueNDRangeKernel` to launch a kernel with multiple work-items

```
cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue,
                              cl_kernel kernel,
                              cl_uint work_dim,
                              const size_t *global_work_offset,
                              const size_t *global_work_size,
                              const size_t *local_work_size,
                              cl_uint num_events_in_wait_list,
                              const cl_event *event_wait_list,
                              cl_event *event)
```

Command queue

Kernel to be executed

Error code

Number of dimensions (1-3)

Starting global ID NULL=0,0,0

Number of total work-items in each dimension. Array of *work_dim* size.

Workgroup size in each dimension. Array of *work_dim* size.

Events used for synchronization.

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);
```


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

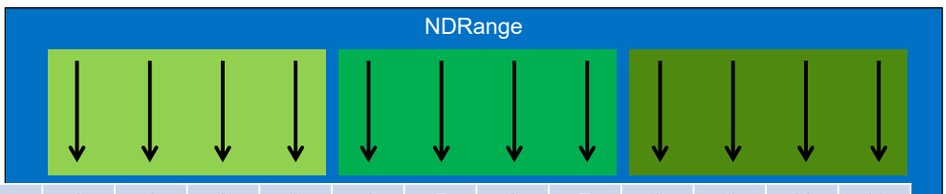
- OpenCL™ kernels have functions to identify the current work-item executing the kernel
  - Take the dimension as an `uint` argument (0-2)
  - Often used to dereference data pointers
  - `get_global_id(dim)`
    - Index of work-item in the global space
  - `get_local_id(dim)`
    - Index of work-item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

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

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

```
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)

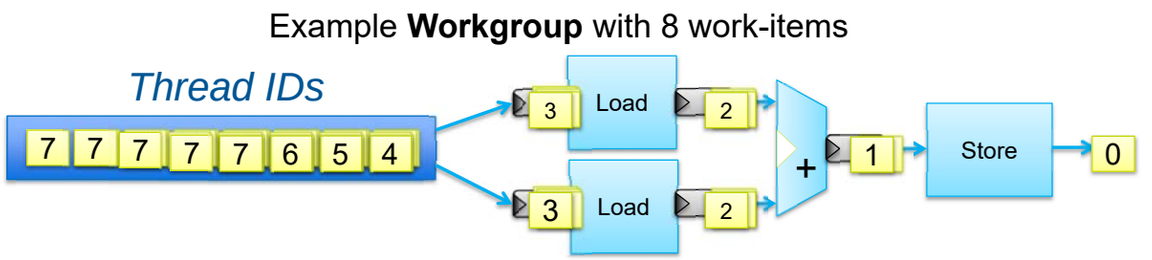
- ~~Simplest way of mapping kernel functions to FPGAs may appear to be replicating hardware for each work-item (thread)~~
- Problems:
  - NDRange size tend to be really large
    - Can we implement millions of kernel pipelines at once on an FPGA?
  - Inefficient and wasteful
    - FPGA compute bandwidth 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 Optimizing OpenCL™ instructor-led training or 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

Example Pipeline for Vector Add

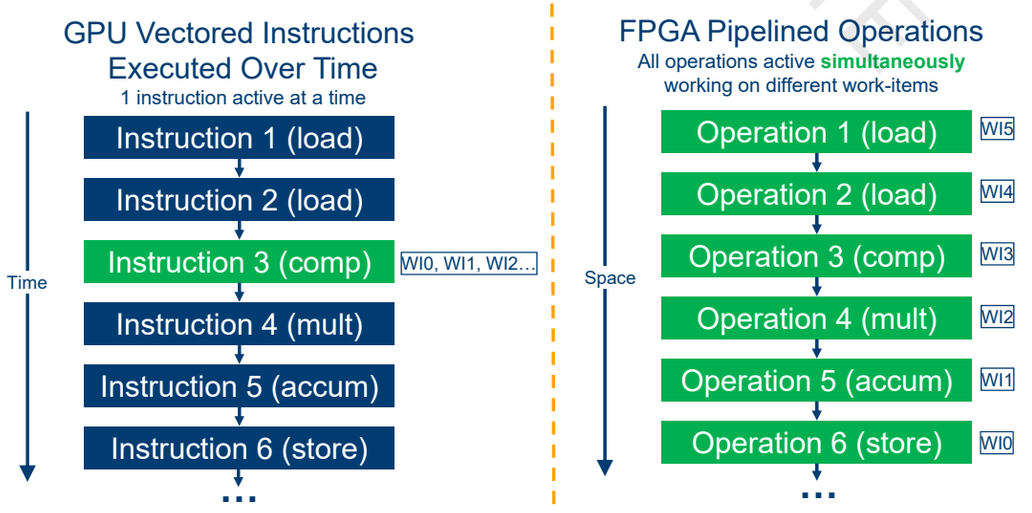
- On each cycle the portions of the pipeline are processing different threads
- While work-item 2 is being loaded, work-item 1 is being added, and work-item 0 is being stored



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

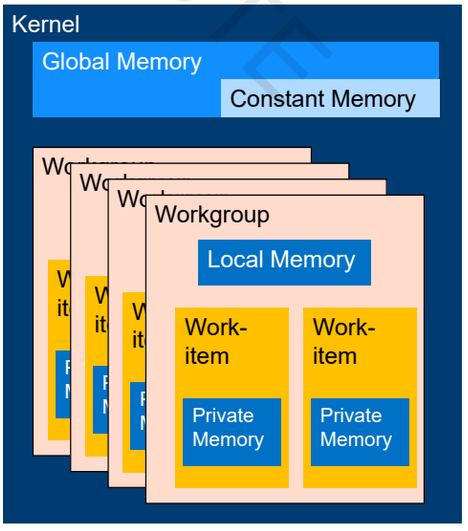
NDRange Kernels

- Kernels and work-item hierarchy
- Memory model

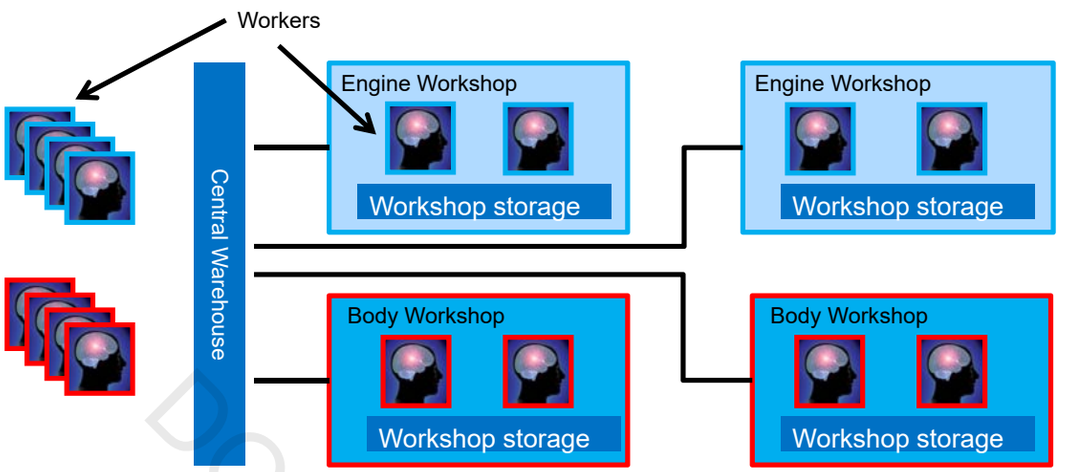
OpenCL on Intel® FPGAs

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], …)
   …
}
```

Test Your Knowledge

- How does a kernel identify the current work-item?
  a) It doesn't need to
  b) Magic, it just knows
  c) Global attribute variable
  d) Through get_local_id(), get_global_id(), get_group_id() calls

Exercise 3

Writing a Simple NDRange Kernel

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

Executing OpenCL Kernels

NDRange Kernels

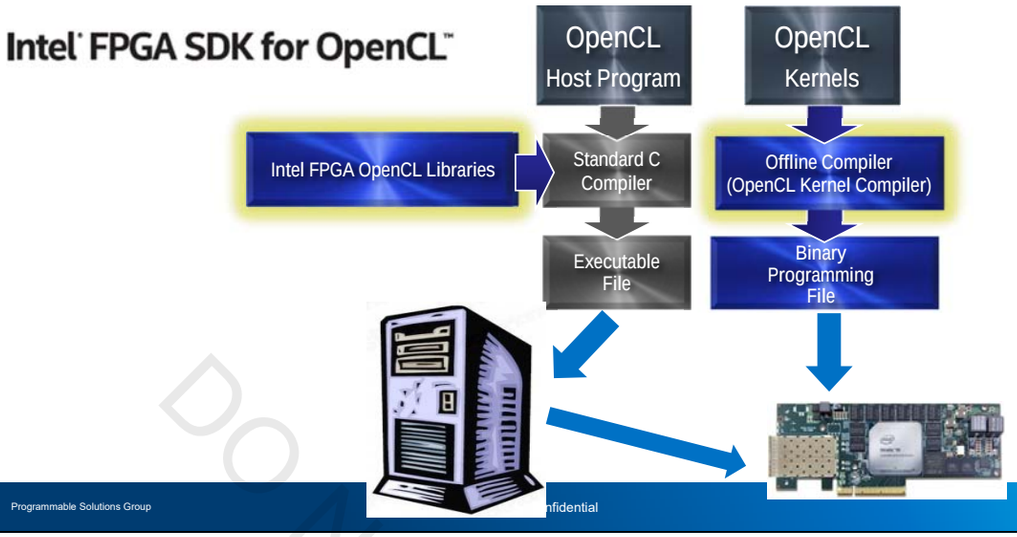
OpenCL on Intel® FPGAs

- **The Intel SDK for OpenCL**
- Single Work-Item Kernels
- Debug Tools
- Libraries
- Custom Boards

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
- Software Requirements
  - Quartus® Prime tool with the appropriate devices
  - Licenses for the Intel FPGA SDK for OpenCL and Quartus tool
  - Generic C compiler for the host program

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
`--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 aocx and aoco files
- For detailed info on supported kernel constructs see the Intel® FPGA SDK for OpenCL™ programming Guide

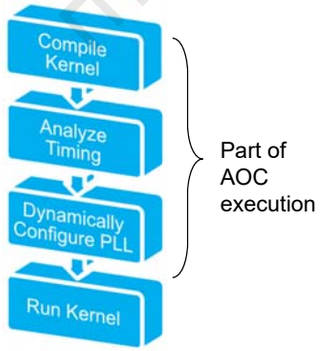
There are many other debugging, optimization, and build options.

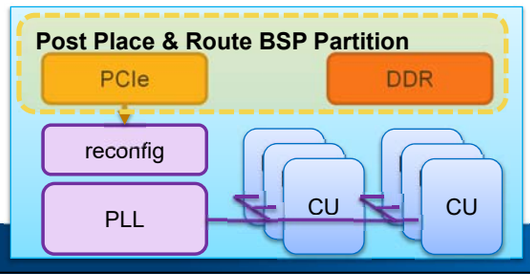
Intel® FPGA Preferred Board for OpenCL™

- 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

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

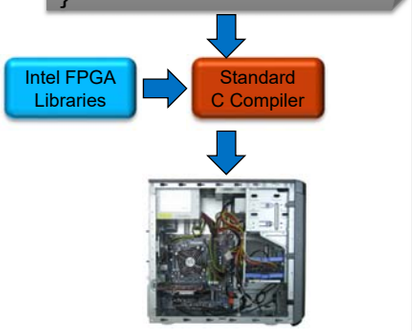




Compiling the Host Program

- Use a conventional C compiler (Visual Studio/GCC)
- Add `%ALTERAOCLSDKROOT%/host/include` to your file search path
  - Recommended to use `aocl compile-config`
- Include CL/opencl.h in your source code
- Link to Intel FPGA OpenCL libraries
  - Link to libraries located in the `%ALTERAOCLSDKROOT%/host/<OS>/lib` directory
    - Recommended to use `aocl link-config`

```
main() {
   read_data( … );
   manipulate( … );
   clEnqueueWriteBuffer( … );
   clEnqueueNDRange(…,sum,…);
   clEnqueueReadBuffer( … );
   display_result( … );
}
```



AOCL Utility

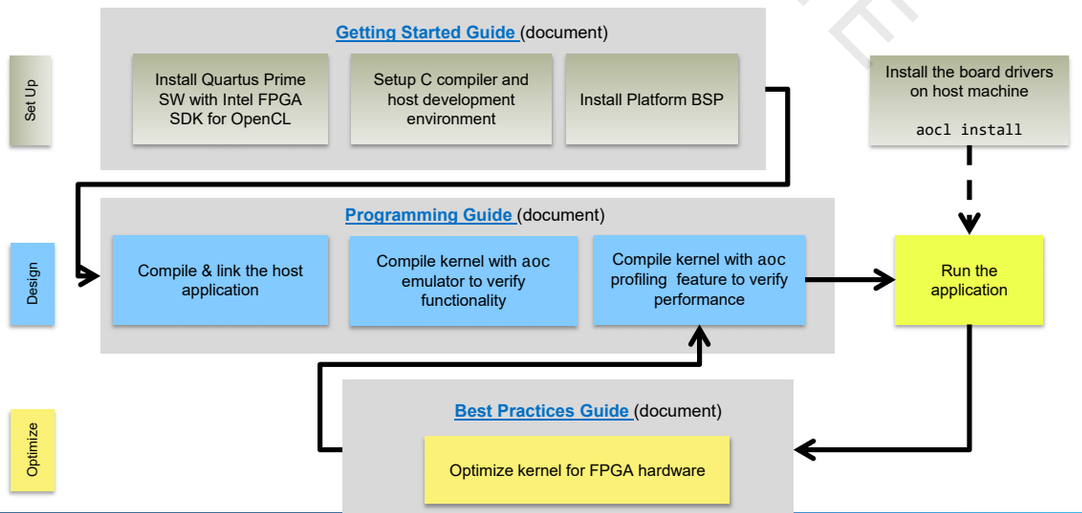
Host Compilation Commands (Use in your makefile)	
`aocl compile-config`	Displays the compiler flags for compiling your host program
`aocl link-config`	Shows the link options needed by the host program to link with libraries
`aocl makefile`	Shows example Makefile fragments for compiling and linking a 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`
  - Programs FPGA directly, usually across PCIe* or JTAG download cable
- `aocl flash <board instance> <BSP compatible image>.aocx`
  - Programs the user region of the flash on the board with boardtest.aocx
  - FPGA loads from flash upon power-up

Intel® FPGA SDK for OpenCL™ Design Flow



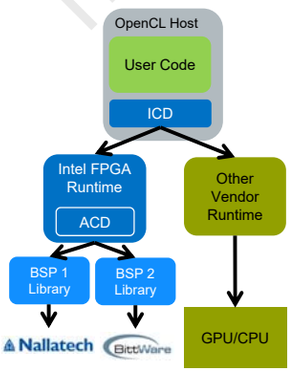
Host Execution

Prior to executing the compiled host program:

- Install Intel® FPGA Runtime Environment for OpenCL™
  - If host is not the kernel development system
- Run `aocl install` installs the drivers needed for the current board package
  - Needs to be done before running the host
- Ensure `%ALTERAOCLSDKROOT%\<OS>\bin` is part of the PATH
  - Contains the dynamic libraries provided by AOCL needed by the host
- Ensure `%AOCL_BOARD_PACKAGE_ROOT%\<OS>\bin` is part of the PATH
  - Contains the dynamic libraries provided by the BSP needed by the host

ACD / ICD Support

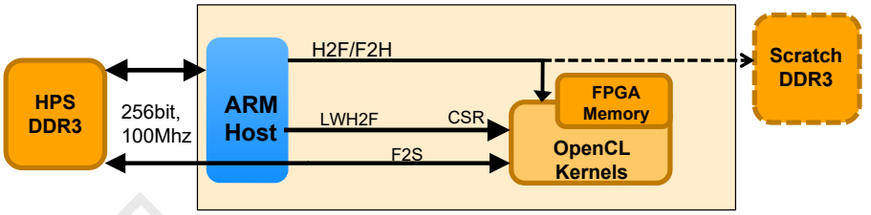
- Installable Client Driver (ICD)
  - Allows multiple vendor runtime libraries to be dynamically loaded by a single host
- Altera Client Driver (ACD)
  - Allows multiple BSP runtime libraries to be dynamically loaded by a single host



Intel® SoC FPGA Platforms

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

- Cyclone® V, Arria® V, or Arria 10 SoC devices



Shared Physical Memory - SoC Devices

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

- Intel® FPGA recommend using shared memory instead of dedicated FPGA DDR
- 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` to create host pointer

Host Code for Allocating Shared Memory (SoC)

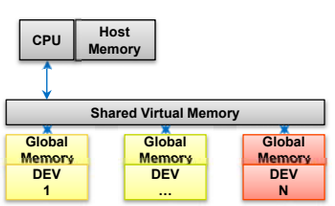
- Shared memory must be physically contiguous
- CPU caching is disabled for the shared memory
- Use the `CL_MEM_ALLOC_HOST_PTR` flag
- `clEnqueueMapBuffer` function required to create pointer to allocated space
  - `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` not used
  - Cannot use **malloc** or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

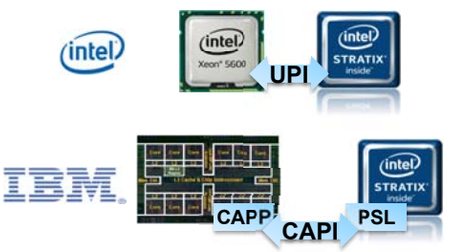
```
cl_mem src = clCreateBuffer(…, CL_MEM_ALLOC_HOST_PTR, size, …);
int *src_ptr = (int*)clEnqueueMapBuffer (…, src, size, …);
*src_ptr = input_value; //host writes to ptr directly
clSetKernelArg (…, src);
clEnqueueNDRangeKernel(…);
clFinish();
printf (“Result = %d\n”, *dst_ptr); //result is available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


Shared Virtual Memory (SVM)

Hosted Heterogeneous Platform with SVM

- Works over cache-coherent interfaces
- Allows use of pointer-based data types

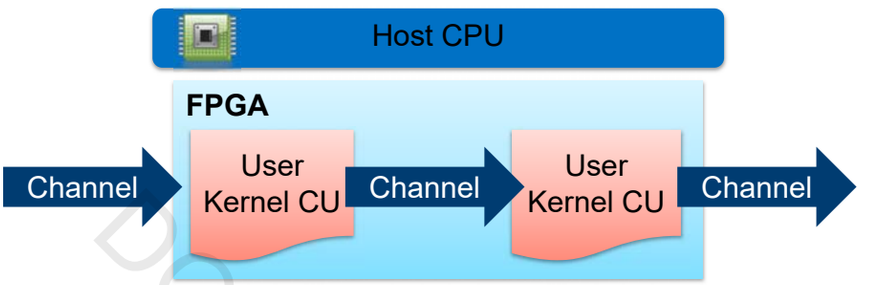




Channels / Pipes

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

- Enable aoc to implement FIFOs



See the Optimizing OpenCL instructor-led training or the best practices guide for more information

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

Executing OpenCL Kernels

NDRange Kernels

OpenCL on Intel® FPGAs

- The Intel SDK for OpenCL
- **Single Work-Item Kernels**
- Debug Tools
- Libraries
- Custom Boards

Single Work-Item Execution

- Launching kernels with NDRange of (1,1,1)
  - A kernel executed on a compute unit always with NDRange of one work-item
- Defined as a **Task** in OpenCL™
- **Loops in single work-item kernels automatically parallelized by the Intel® FPGA OpenCL Compiler**
- 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

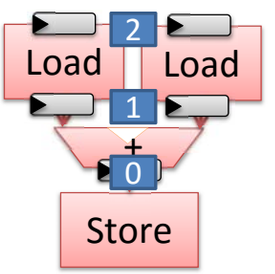
Algorithm

```
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

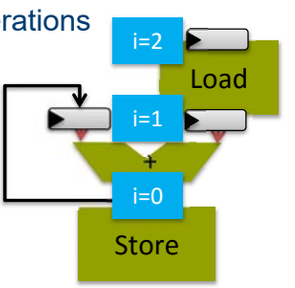


Tasks and Loop-pipelining Implementation

- 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

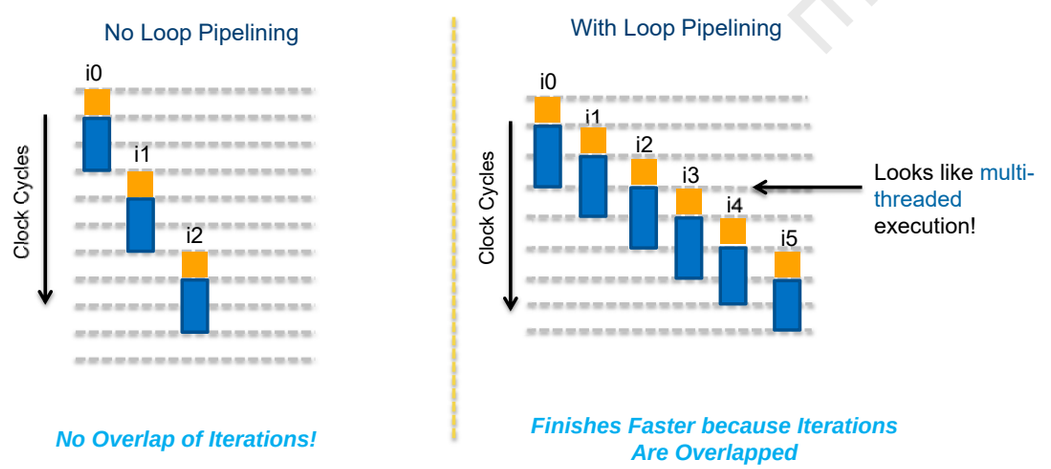
```
float array[M];
for (int i=0; i < n; i++)
{
   for (int j=0; j < M-1; j++)
        array[j] = array[j+1];
   array[M-1] = a[i];

   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

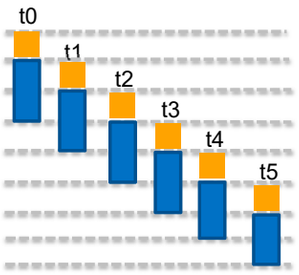
Reduction on array (Not a dependency)

Loop Pipelining Example



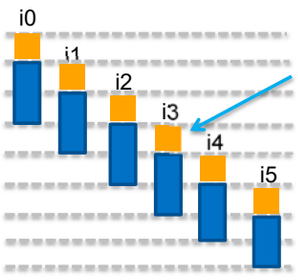
Parallel Threads vs Loop Pipelining

NDRange Parallel Threads



Parallel threads launch 1 thread per clock cycle in pipelined fashion

Loop Pipelining



Loop dependencies may not be resolved in 1 clock cycle

- Loop Pipelining enables Pipeline Parallelism AND the communication of state information between iterations.
  - If dependency can be resolved in 1 clock cycle, then the resulting computational throughput is the same

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
  - Kernel does not have loop and memory dependencies
  - 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 loop pipelining and memory dependence 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();
```

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

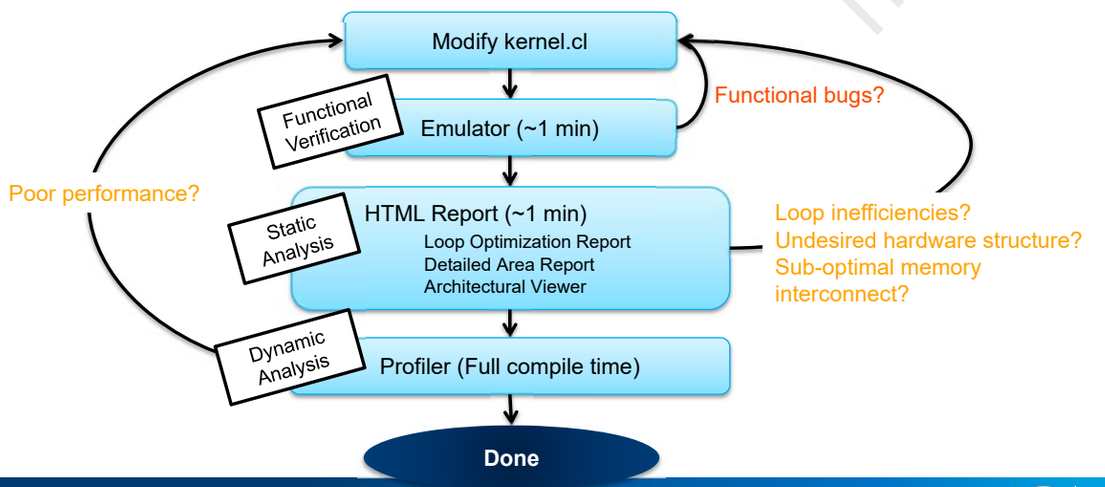
Executing OpenCL Kernels

NDRange Kernels

OpenCL on Intel® FPGAs

- The Intel SDK for OpenCL
- Single Work-Item Kernels
- **Debug Tools**
- Libraries
- Custom Boards

Kernel Development Flow and Tools



Emulator

Enable kernel functional debug on x86 systems

- Quickly generate x86 executables that represent the kernel

```
aoc –march=emulator [-g] <kernel file>
```

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

- Debug support for
  - Standard OpenCL™ syntax, Channels, Printf statements
  - Emulates one device at a time

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 variable to your board
   - Same board option used when compiling the kernel

```
set CL_CONTEXT_EMULATOR_DEVICE_ALTERA=<board_name>
```

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!
```

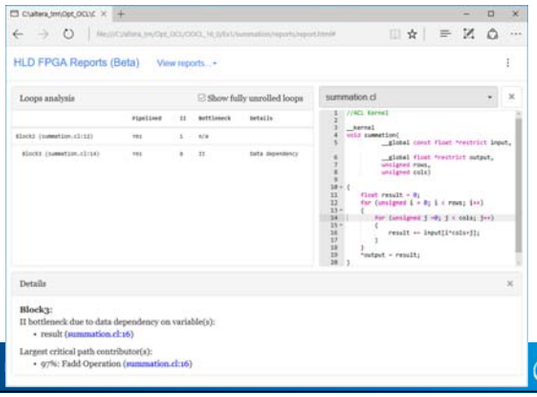
HTML Report

Static report showing optimization, area, and architectural information

- Automatically generated with the object file (`aoc -c`)
  - 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

Loop Analysis Optimization Report

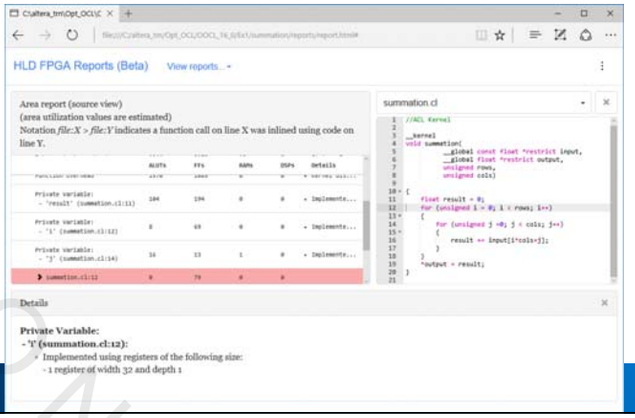
- Static actionable feedback on pipeline status of loops in single work-item kernels
  - Shows loop carried dependencies and bottlenecks
- Shows loop unrolling status



Area Report

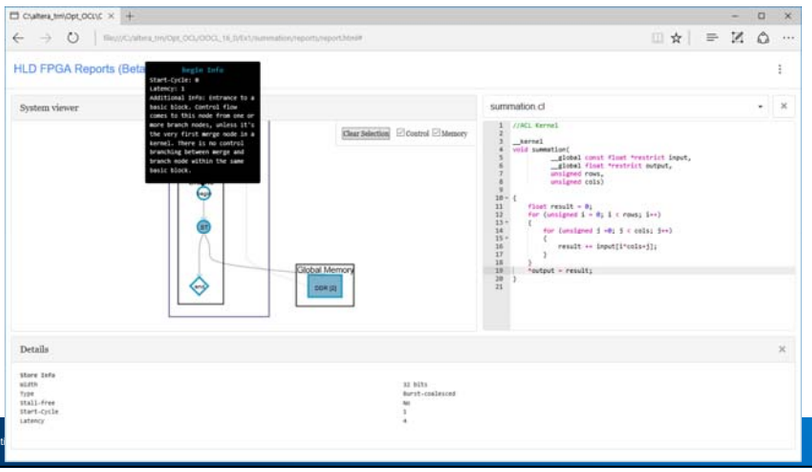
Generate detailed area utilization report of kernel code

- Breakdown by source line available



Architectural Viewer

- Displays kernel pipeline implementation and memory access implementation

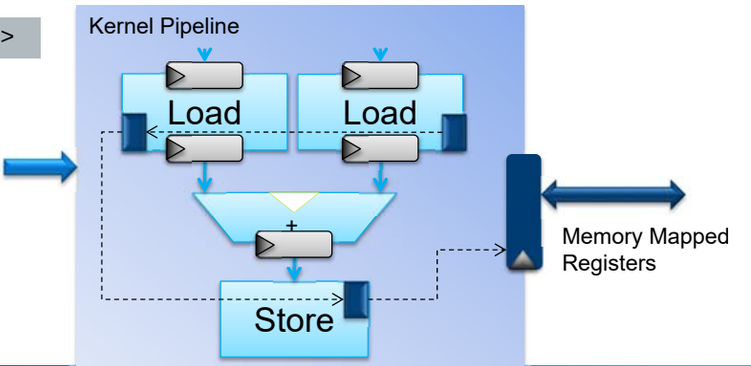


Profiler

- 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];
  …
}
```



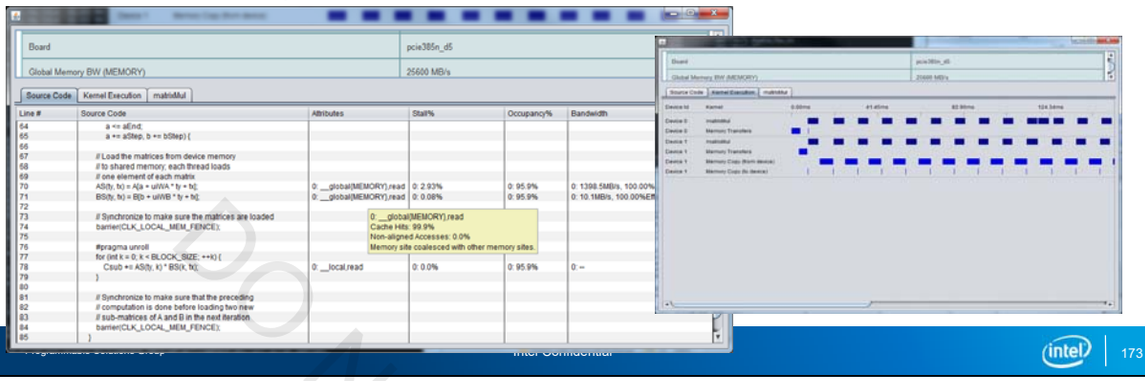
Collecting and Viewing Profile Information

- Compile kernel with `aoc --profile` option
- 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

```
aocl report <kernel file>.aocx profile.mon
```

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

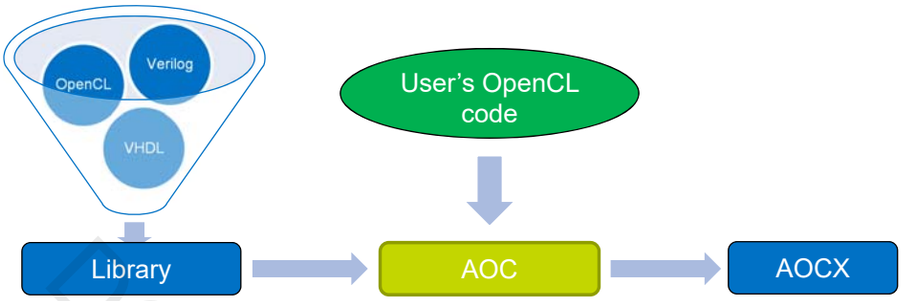
NDRange Kernels

OpenCL on Intel® FPGAs

- The Intel SDK for OpenCL
- Single Work-Item Kernels
- Debug Tools
- **Libraries**
- Custom Boards

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 the 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

```
aoc -c 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 -c -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

Class Agenda

Heterogeneous Parallel Computing

OpenCL™ Platform and Host-side Software

Executing OpenCL Kernels

NDRange Kernels

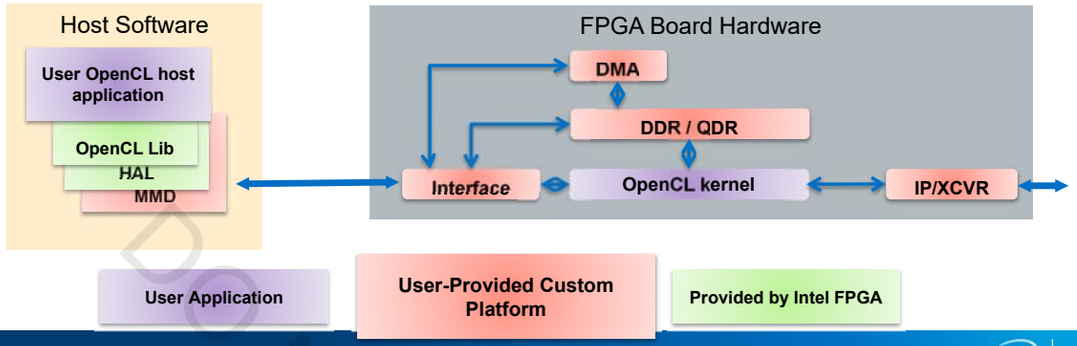
OpenCL on Intel® FPGAs

- The Intel SDK for OpenCL
- Single Work-Item Kernels
- Debug Tools
- Libraries
- Custom Boards

Custom Platform

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

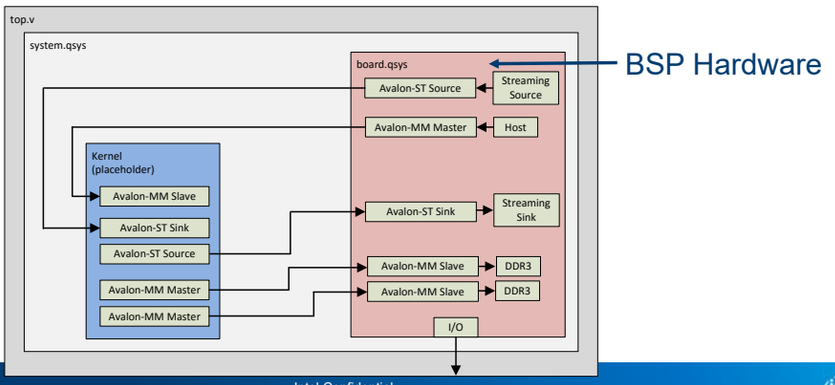
- FPGA design, software, and board bring up skills required



Hardware System Overview

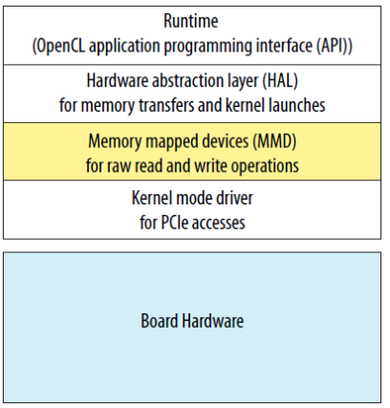
Board support package hardware (board.qsys) need to be provided

- 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



OpenCL 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
- Arria® 10 Reference Platform, Stratix® V Reference Platform, and Cyclone® V SoC Reference Platforms
  - Ships with the Intel® FPGA SDK for OpenCL™
- Stratix V Network Reference Platform
  - Download from OpenCL board platforms landing page
- Template project - Custom BSP toolkit deliverable
  - Skeleton design

Test Your Knowledge

- How are kernels compiled into a FPGA bitstream?
  a) Offline, using the Offline Compiler aoc
  b) Online, by the host as it runs
  c) Using a generic C compiler
  d) Using the Quartus software directly
- How are OpenCL host code compiled?
  a) Using the Offline Compiler aoc
  b) With Python scripts
  c) Using a generic C compiler linked to Intel FPGA libraries
  d) None of the above

Course Summary

- Heterogeneous Parallel Computing
  - Boost performance with heterogeneous parallel systems
- 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

OpenCL® References

- Intel® FPGA OpenCL collateral
  - 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
  - http://www.khronos.org/files/opencl-quick-reference-card.pdf

Training on Advanced Topics

Follow-on Intel® FPGA trainings

- Optimizing OpenCL™ for Intel FPGAs instructor-led training
- Single-Threaded vs. Multi-Threaded Kernels online training
- Custom Platform instructor-led or online training

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

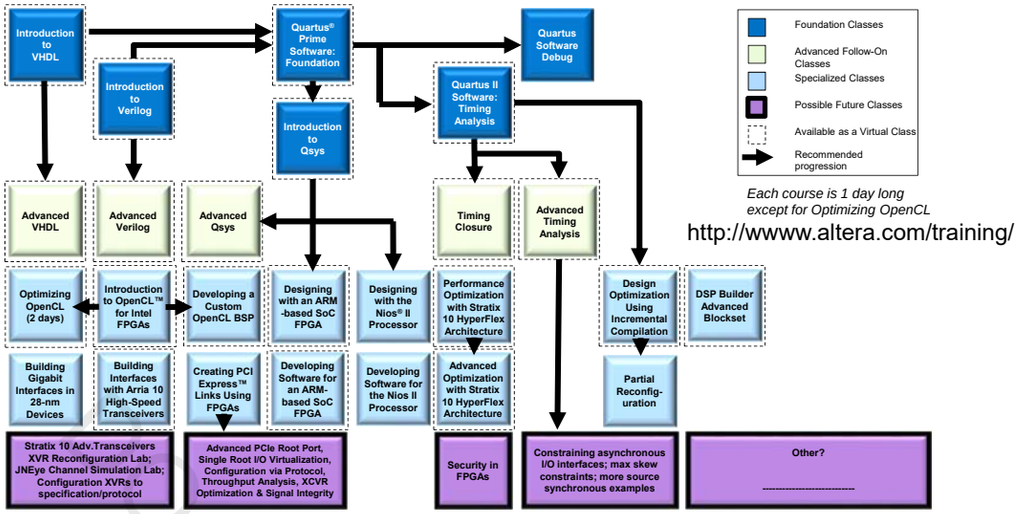
Virtual Classes

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

Instructor-Led and Virtual Training Curriculum



Each course is 1 day long except for Optimizing OpenCL

http://wwww.altera.com/training/

Exercise 4

Examining Kernel Compilation Reports

Intel® FPGA Technical Support

Quartus® Prime software on-line help

Quartus Prime Handbook

www.altera.com

- Search for answers to problems with Knowledge Database
- Download literature
- View design examples
- View online trainings
- Intellectual property support

mySupport: www.altera.com/myAltera

Intel FPGA Wiki: www.alterawiki.com

Intel FPGA Forum: www.alteraforum.com

Field applications engineers: contact your local Intel FPGA sales office

Exercise Manual

for

Introduction to OpenCL for Intel FPGAs

Software Requirements

Microsoft® Visual C++® 2010 (Express)
Intel® FPGA SDK for OpenCL™ version 16.1
Quartus® Prime Standard software version 16.1

http://www.altera.com/customertraining/ILT/Introduction_to_OpenCL_for_Intel_FPGAs_16_1_v1.zip

Exercise 1

Setting Up OpenCL Host-Side Application

In this exercise, you will practice writing an OpenCL host-side application including constructs to get the OpenCL platform and device; create an OpenCL context, command queue, and buffers.

Step 1. Verify Project Environment

____ 1. Unzip the project files using the following steps.

a. In an Explorer window, go to **C:\altera_trn\OpenCL**.

This will be your exercise directory.

b. Delete any old lab file folders that may already exist that start with ***OCL_***.

c. Double-click the executable file (**Introduction_to_OpenCL_for_Intel FPGAs_16_1_v1.exe**).

d. In the WinZip dialog box, click **Unzip** to automatically extract the files to the directory mentioned above.

e. Click **Close** when finished.

____ 2. Launch Visual C++: From the Windows taskbar select **All Programs → Microsoft Visual Studio 2010 (Express) → Microsoft Visual C++ 2010 (Express)**

____ 3. Open **OpenCL Exercise** solution using the following steps.

a. Go to **File -> Open -> Project/Solution**.

b. Navigate to **C:\altera_trn\OpenCL\OCL_16_1**

c. Select **OpenCLExercise_soln.sln**.

d. Click on **Open.**

*This Solution contains one project: **SimpleOpenCL**. This is the project we're going to use to compile our OpenCL program. You should notice that there are three source files. **utility.cpp** contains functions that performs some useful tasks such as printing information about our OpenCL environment. **SimpleKernel.cl** contains the kernel code, we will modify this file in the next exercise. **main.c** has main() that will contain all of our host side code, this is what we're going to modify in this exercise.*

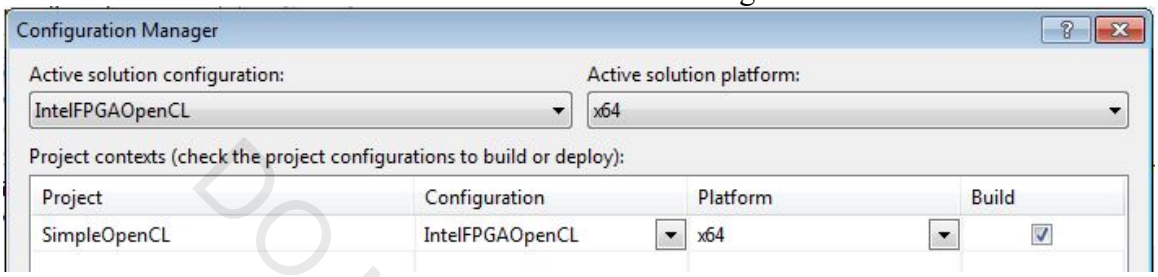
____ 4. Configure the project using the following steps.

a. Right click on the Project **SimpleOpenCL** in the Solution Explorer.

b. Click on **Properties**.

c. At the top of the Property Pages, click on **Configuration Manager…**

d. From the dropdown menus, change the **Active solution configuration** and **Active solution Platform** to match the following.



The IntelFPGAOpenCL configuration contains setting that points to the Intel FPGA include directories and links to the Intel FPGA SDK for OpenCL libraries. With this configuration, our kernel (after we write it in exercise 2) will run on the FPGA.

e. Click **Close** to close the Configuration Manager.

____ 5. In the Property Pages, go to **Configuration Properties** -> **General**. Verify that the **Platform Toolset** has been set to **Windows7.1SDK**. This is what allows us to compile 64-bit executables.

____ 6. In the Property Pages, go to **C/C++** -> **General**. You should see that the **Additional Include Directories** should include a path to the host include files.

This directory allows our compile to be aware of the OpenCL header files.

____ 7. Now go to **Linker->General**. You should see Intel FPGA's 64-bit library path listed under A**dditional Library Directories**.

This points the linker to the library directory where Intel's libraries are located.

____ 8. Go to **Linker->Input**. In the **Additional Dependencies** field, you should see **opencl.lib**. This is Intel FPGA's provided static library for compiling and running OpenCL kernels.

____ 9. Click **OK** to close the Property Pages.

Step 2. Write the host side code

____ 1. Open **main.cpp** from the **Solution Explorer** by expanding **Source Files** and double-click on **main.cpp**.

____ 2. Look for the comment "Exercise 1 Step 2.3" in **main.cpp**.

____ 3. Right below the comment, complete the line of code that will set the platform ID to the variable **myPlatform.** Set the value returned to the variable **err**.

Hint: Since we statically linked to the Intel FPGA libraries, we know for sure there's only 1 platform so we don't need to use the third argument to query for the number of total platforms.

____ 4. Look for the comment "Exercise 1 Step 2.5" in **main.cpp**

____ 5. Write the code to set the Device ID to the variable **myDevice** below the comment "Exercise 1, Step 2.5"

*Hint: Use the device type **CL_DEVICE_TYPE_ALL**. Since we're going to execute this host program in emulation mode, there will only be 1 device so there's no need to query for the total number of devices available.*

____ 6. Create a context named **context** below the comment "Exercise 1, Step 2.6."

Everything we wrote up until this point is considered setup code using OpenCL platform layer APIs. This is code that only needs to be written once and can be used in many application scenarios. In your own code, you would likely store the setup code in a function so it can be easily reused.

____ 7. Create a command queue named **queue** below the comment "Exercise 1, Step 2.7."

____ 8. Below the comment "Exercise 1, Step 2.8", create three buffers named **kernelIn**, **kernelIn2,** and **kernelOut**. The first two buffers are READ_ONLY and kernelOut is WRITE_ONLY. The size of the buffer for all three should be sizeof(cl_float)*vectorSize which is the total size of the array in bytes.

These are used to represent memory on the device and will be used as arguments to the kernel later.

____ 9. Below the comment "Exercise 1, Step 2.9", write two lines that would transfer the contents of **X** and **Y** into the buffers **kernelIn** and **kernelIn2** on the device respectively.

*X, Y, and Z are void pointers allocated on the host. The function **allocate_generate**, allocates the memory for all three to be of size **vectorSize**, it also randomly generates the float content that will fill the arrays **X** and **Y**. By default, numbers generated will be from 0 to 1000, you can change this if you wish to any range you want by changing the values of **LO** and **HI**.*

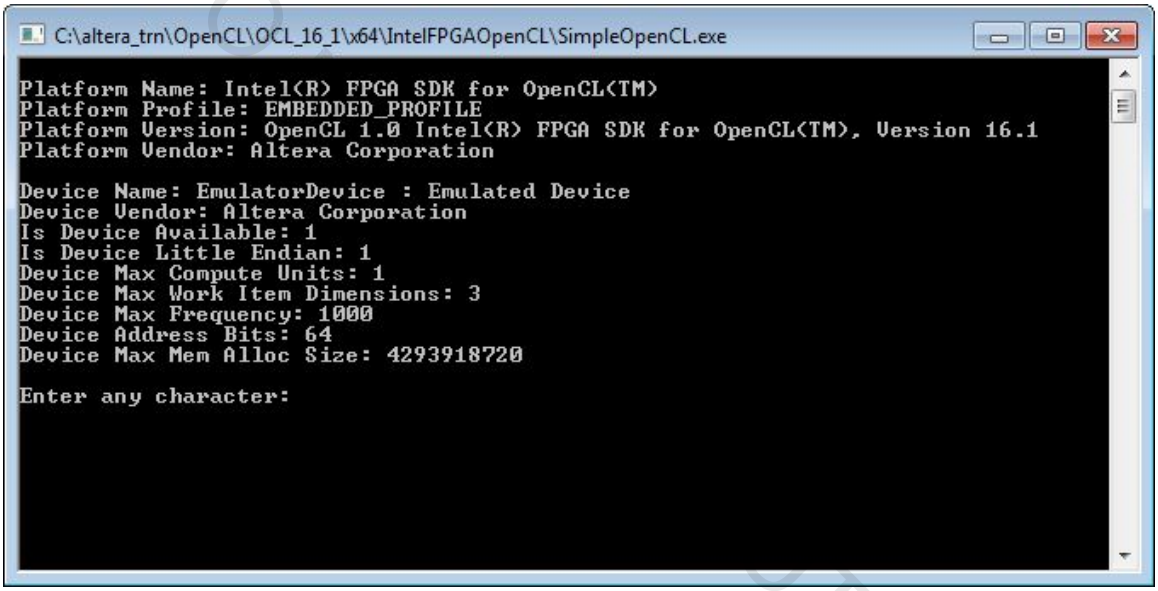
Step 3. Run and debug the host-side program

____ 1. Save **main.cpp**.

____ 2. Compile the project by pressing **F7** (invoked with **Fn F7** when FnLk is off). Fix any errors you may encounter. You should see the "Build: 1 succeeded" message from the output console.

____ 3. Run the program. Go to the menu **Debug->Start Debugging** or press **F5** (or **Fn F5).** When the program runs toward the end, you should see information regarding the Platform and Device printed in the output console, which appears in a separate window. You'll need to use the Windows Task Bar to switch to it.

Here you see the various properties of your OpenCL Platform as well as the device which is the Stratix V reference board.



____ 4. Stop the program by entering any character in the execution window or in the Visual C++ window, press  .

Exercise Summary

- Practiced writing OpenCL host-side code that moved content into the device memory.

END OF EXERCISE 1

Exercise 2

Write a simple OpenCL kernel

In this exercise, you will write an OpenCL kernel and launch it from the host-code that you started writing in exercise 1.

Step 1. Writing and Compiling the Kernel

____ 1. Reopen the **SimpleOpenCL** solution in Visual C++ if it's not already open.

____ 2. Open **SimpleKernel.cl** by double-clicking on it in the Solution Explorer

____ 3. Write a kernel named **SimpleKernel** with **two** float input arguments, **one** float output argument, and **one** uint argument for the size of the arrays.

a. The 1st three arguments of the kernel should be pointers stored in global memory.

b. The 4th argument is passed in by value.

c. Run a loop iterating through all the elements of the arrays

d. Inside the loop, perform any vector math function you want on the two inputs and store the result to the output argument. Use the loop index to dereference the input and output arrays.

Refer to the slides to see the available math operations..

Your kernel can be something very simple, for example just a multiply or add operation or it can be something more involved.

____ 4. Using Windows Explorer, go to the **C:\altera_trn\OpenCL\OCL_16_1** folder.

____ 5. Double-click the **helpful_commands.txt** file to open it.

This file can be used to cut and paste commands to save time.

____ 6. Double-click the **OpenCL Intro** shortcut.

*This opens a command prompt in the **C:\altera_trn\OpenCL\OCL_16_1** directory.*

____ 7. Type the following command (or use helpful_commands.txt) at the prompt

```
opencl_init.bat
```

This script sets the environment variables needed for the Intel FPGA OpenCL Kernel Compile. It points the compiler to the proper board support package and configures the Microsoft compiler to be in 64 bit mode.

____ 8. To compile the kernel with the following command (or use **helpful_commands.txt**)

```
aoc -march=emulator --board s5_ref SimpleKernel.cl
```

This compiles the kernel using the offline compiler targeting the Stratix V reference board in emulation mode.

You likely will see warnings regarding the absence of the restrict keyword.

```
C:\altera_trn\OpenCL\OCL_16_1>aoc -march=emulator SimpleKernel.cl
c:/altera_trn/OpenCL/OCL_16_1/SimpleKernel.cl:3:42: warning: declaring kernel argument with no 'rest
rict' may lead to low kernel performance
void SimpleKernel(__global const float * in, __global const float * in2, __global float * out, uint
N)
                                         ^
c:/altera_trn/OpenCL/OCL_16_1/SimpleKernel.cl:3:69: warning: declaring kernel argument with no 'rest
rict' may lead to low kernel performance
void SimpleKernel(__global const float * in, __global const float * in2, __global float * out, uint
N)
                                                                        ^
c:/altera_trn/OpenCL/OCL_16_1/SimpleKernel.cl:3:91: warning: declaring kernel argument with no 'rest
rict' may lead to low kernel performance
void SimpleKernel(__global const float * in, __global const float * in2, __global float * out, uint
N)
                                                                                          ^
3 warnings generated.
```

This is because the AOC compiler makes a lot of dependency analysis to optimize the pipelined circuit created. Multiple pointers can technically point to the same area, which prevents the compiler from creating the optimal circuit. Using the "restrict" keyword lets the compiler know that in pointers won't point to the same area and that it can treat pointers as separate and make optimizations accordingly.

*To resolve the warning, simply add the **restrict** keyword to each of the pointers in the kernel.*

*e.g. **__global const float * restrict in***

Remember if you change the kernel code, you will need to rerun the aoc compiler on the kernel file.

Step 2. Write the host code that launches the kernel

____ 1. Reopen **main.cpp** in Visual C++.

____ 2. Find "#define EXERCISE1" around line 9 in **main.c** and comment it out.

Current line numbers appear at the bottom status bar in Visual Studio, however you may enable line numbers in general by going to Tools->Options->Text Editor->All Languages->General

Commenting enables the portion of the code that launches the kernel and does a performance profiling of the kernel along with the verification of the results.

____ 3. Find the comment "Exercise 2 Step 2.3". Right beneath it, write the code that would create the program from binary. This is used for the program meant to be run on the FPGA. Use the arguments (context, 1, &myDevice, &lengths, (const unsigned char**)&binaries, &kernel_status, &err)

In this code, the variable named "binaries" already stores the information from the .aocx file used to program the FPGA via CvP. In the emulation mode, the aocx file is just a software library that can be dynamically linked with the host code.

____ 4. After the comment "Exercise 2 Step 2.4", call the function that **builds** the program from the program created in the previous two steps.

For Intel FPGA, this is only done as a formality. Set the build options to be ""

____ 5. Call the OpenCL function to **Create** the Kernel from the **program** after the comment "Exercise 2 Step 2.5."

You can use the kernel_name variable in your argument which is set to "SimpleKernel" and should match your kernel name.

____ 6. Call the function to setup the Kernel arguments four times, once for each buffer: **kernelIn**, **kernelIn2**, and **kernelOut**. And another to pass in the variable vectorSize. Do this after the comment "Exercise 2 Step 2.6."

For vectorSize, use sizeof(cl_uint) as the size of the parameter.

____ 7. After the comment "Exercise 2 Step 2.7," **Launch** the kernel using **clEnqueueTask**.

Use the command queue we created in the first exercise. Set Event wait list and event both to be NULL

This command will execute the kernel on the OpenCL device

____ 8. Following the comment "Exercise 2 Step 2.8," read back the results of the kernel execution by copying the contents of the **kernelOut** buffer to the array **Z**.

*The contents of Z will be verified later in **main**.*

Make sure the blocking argument of the call is set to CL_TRUE. This guarantees that after read function, Z will be ready to be used.

____ 9. After the comment "Exercise 2 Step 2.10," write the **same** math operation you have in the kernel. This time, however, it should operate on X[i] and Y[i] and store the result into CalcZ[i].

a. You'll need to type convert X, Y, and CalcZ using the construct ((float*) X)[i], ((float *)Y)[i], and ((float*) CalcZ)[i]

Here we're doing the same calculation the kernel is doing, except that we're using a regular C. The contents are then stored in CalcZ, which will be compared to the contents of Z.

If you're attempting the extra credit moving average design, set CalcZ to the moving average of X and disregard Y.

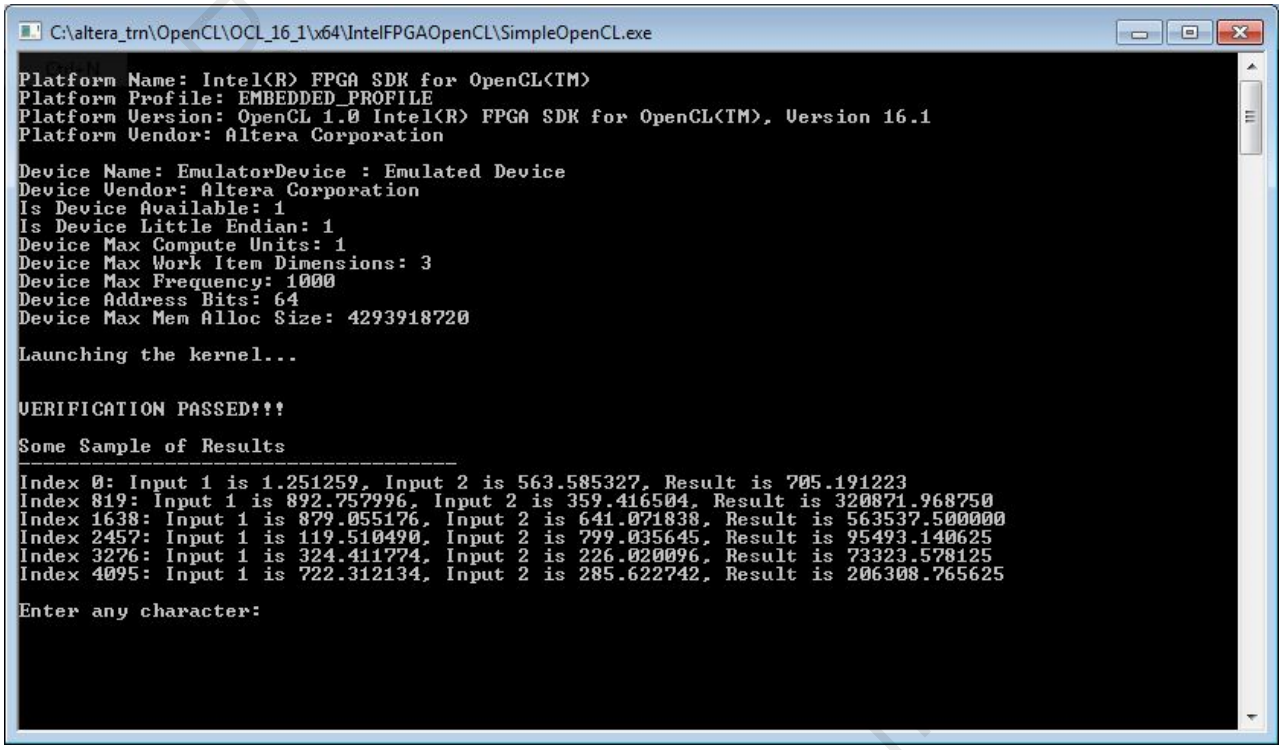
____ 10. Save **main.cpp.**

Step 3. Run and debug the kernel

____ 1. Press **F7** (invoked with **Fn F7** when FnLk is off) to compile the project. Fix any errors you may encounter.

____ 2. Run the program. Go to the menu **Debug->Start Debugging** or press **F5** (or **Fn F5**). When the program runs to the breakpoint, you should see performance results in the output console (a separate window) as well as the "VERIFICATION PASSED!!!" message, along with some samples of results.

The Verification Passed message means the contents of Z and CalcZ are the same. You can also verify that the sample of results matches the math operation you've performed.



____ 3. Finish the running of the program.

Exercise Summary

- Practiced writing a kernel
- Wrote host code that setup the arguments to the kernel
- Launched the kernel using clEnqueueTask

END OF EXERCISE 2

Exercise 3

NDRange Kernel

In this exercise we will convert kernel that you created in exercise 2 into an NDRange kernel

Step 1. Convert and Compile the Kernel

____ 1. Reopen the **SimpleOpenCL** solution in Visual C++ if it's not already open.

____ 2. Open **SimpleKernel.cl** by double-clicking on it in the Solution Explorer

____ 3. Save it as **SimpleKernel_For.cl**

We will come back to the for loop version of the kernel in the next exercise.

____ 4. Reopen SimpleKernel.cl

____ 5. Convert the kernel into a ND Range Kernel.

Follow these steps if you need assistance

a. Remove the 4th argument that represented the number of elements

b. Remove the for loop while keeping the statement(s) inside it

c. At the beginning of the kernel, write the line of code that retrieves the current global index in the X (0) dimension and assigned it to a variable i of type size_t

d. Use i to dereference the input and output arrays.

e. Save the file.

____ 6. If you closed the command prompt

a. Go to the **C:\altera_trn\OpenCL\OCL_16_1** folder in a Windows Explorer window

b. Double-click the **OpenCL Intro** shortcut.

c. Run the initialization script by typing the following in the command prompt.

```
opencl_init.bat
```

____ 7. If you didn't close the command prompt, make sure the command prompt is in the following directory.

```
C:/altera_trn/OpenCL/OCL_16_1
```

____ 8. In the command prompt, type the following command (or use helpful_commands.txt) to compile the kernel and verify that it is error free.

```
aoc -march=emulator --board s5_ref SimpleKernel.cl
```


Step 2. Change the host code to launch a single work-item kernel

____ 1. Reopen **main.cpp** in Visual C++ if not already open.

____ 2. Comment out the 4th clSetKernelArg call that passes in the vectorSize

This is no longer needed

____ 3. Comment out the clEnqueueTask call

____ 4. Right below it add the **clEnqueueNDRangeKernel** function call to launch the kernel in a multi-threaded fashion

Use the following steps if you need help

b. Dimensionality of the work-items and workgroups is 1

c. Global work offset should be **NULL**

d. Global work size is stored in the variable **vectorSize**

e. Work-item per workgroup is stored in the variable **workSize**

f. Event wait list and event can both be NULL

*With this command, you'll be launching the kernel **vectorSize** times. This is how data parallelism is achieved in OpenCL.*

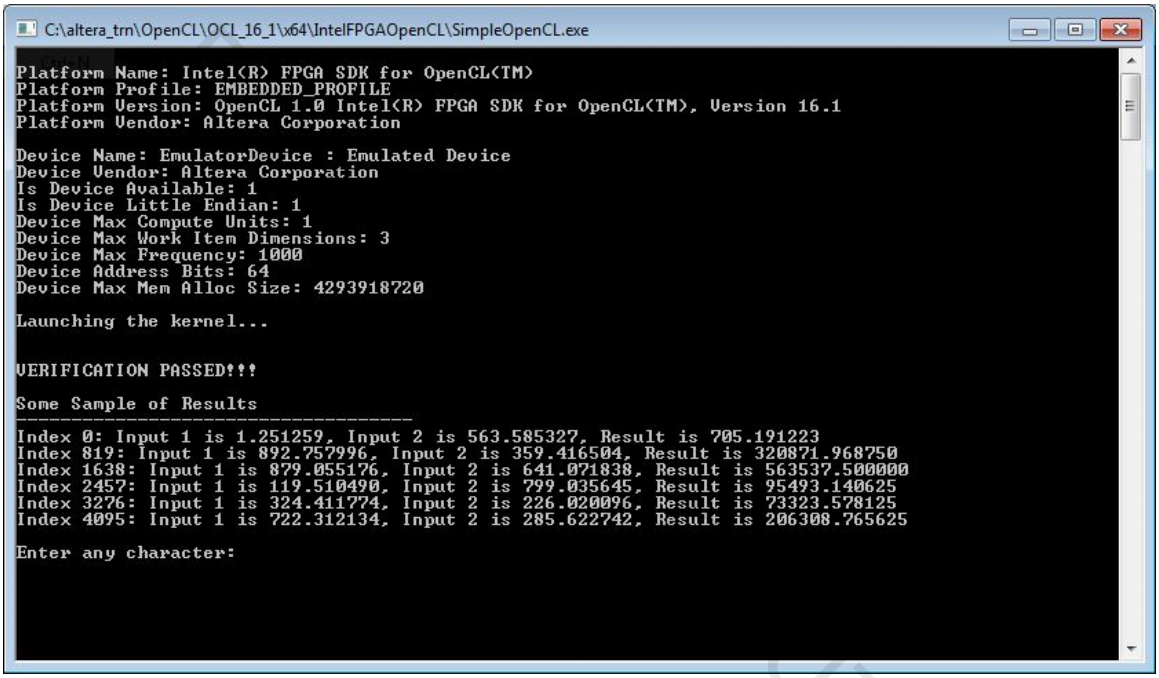
____ 5. Save **main.cpp**

Step 3. Run and debug the kernel

____ 1. Press **F7** (invoked with **Fn F7** when FnLk is off) to compile the project. Fix any errors you may encounter.

____ 2. Run the program. Go to the menu **Debug->Start Debugging** or press **F5 (**or **Fn F5)**. When the program runs to the breakpoint, you should see performance results in the output console (a separate window) as well as the "VERIFICATION PASSED!!!" message, along with some samples of results.

The Verification Passed message means the contents of Z and CalcZ are the same. You can also verify that the sample of results matches the math operation you've performed.



____ 3. Finish the running of the program.

____ 4. Close Visual C++

Exercise Summary

- Created an ND Range Kernel
- Launched the kernel using clEnqueuNDRange Kernel

Congratulations!

You have completed Lab 3

Exercise 4

Examining Kernel Compile Results

In this exercise, we will examine some of the outputs of the Intel FPGA offline compiler that can help us debug and optimize the kernels.

Step 1. Convert and Compile the Kernel

____ 1. Go to the c:\altera_trn\OpenCL\OCL_16_1 command prompt if it's not already open

a. Use the **OpenCL Intro** short cut if necessary

b. Run **`opencl_init.bat`** if it hasn't been run

____ 2. Compile **SimpleKernel_For.cl,** this time NOT in the emulation mode, but we'll stop the compiler after the object file is generated. Either type the following command or use helpful_commands.txt

```
aoc --board s5_ref -c SimpleKernel_For.cl
```

We're going to recompile the for loop version of the kernel from Lab 2. The -c option stops the compiler after the object stage which is a much quicker compile than the full FPGA compile. Stopping after the object file is created allows us to see the compilation results and all the detailed static reports.

____ 3. In the windows explore, navigate to c:\altera_trn\OpenCL\OCL_16_1\SimpleKernel_For

This folder named after the kernel file contains all the compilation results including the generated hardware source files.

____ 4. Examine the log file

a. Inside the SimpleKernel_For folder, look for SimpleKernel_For.log

This is the main compile log which contains the Optimization Report, Estimated Resource Utilization Report and Compile Messages

b. Right click on the log file and open it in Notepad or Notepad++

____ 5. Is your kernel compiled as a single work-item or ND Range kernel? _____

This report contains information how each kernel is compiled.

____ 6. You should also see the Loop Report. How every loop is implemented is shown in the optimization report.

```
Loop Report:

 + Loop "Block2" (file SimpleKernel_For.cl line 8)
   Pipelined well. Successive iterations are launched every cycle.
```


Since this is a single-work-item kernel, AOC will attempt to pipeline the loop and fill the pipeline. In the report shown, our loop is pipelined well, meaning every clock cycle a new iteration is able to enter the pipeline.

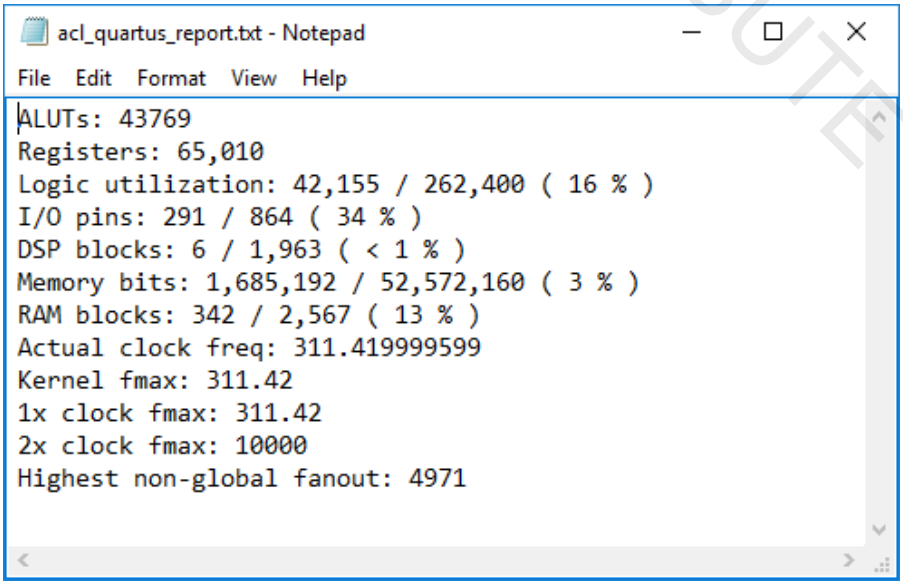
How well did your loop get pipelined? _____

____ 7. Finally the log file also include an estimated resource usage report, this report is an early estimate prior to Quartus compilation.

```
+--------------------------------------------------------------------+
; Estimated Resource Usage Summary                                  ;
+----------------------------------------+---------------------------+
; Resource                               + Usage                     ;
+----------------------------------------+---------------------------+
; Logic utilization                      ;   16%                     ;
; ALUTs                                 ;   10%                     ;
; Dedicated logic registers             ;    6%                     ;
; Memory blocks                         ;   15%                     ;
; DSP blocks                            ;    0%                     ;
+----------------------------------------+---------------------------;
System name: SimpleKernel_For
```

The screen capture is for a vector multiply example. You may wonder why such a kernel consumes this much resource. The reasons is most of the logic is actually occupied by the board support package to support the various interfaces.

____ 8. If this was a full compilation you will see acl_quartus_report.txt which contains information on the actual resource utilization as well as the actual clock frequency used in the Kernel subsystem. The screen capture below is from a report of the vector multiply compile.



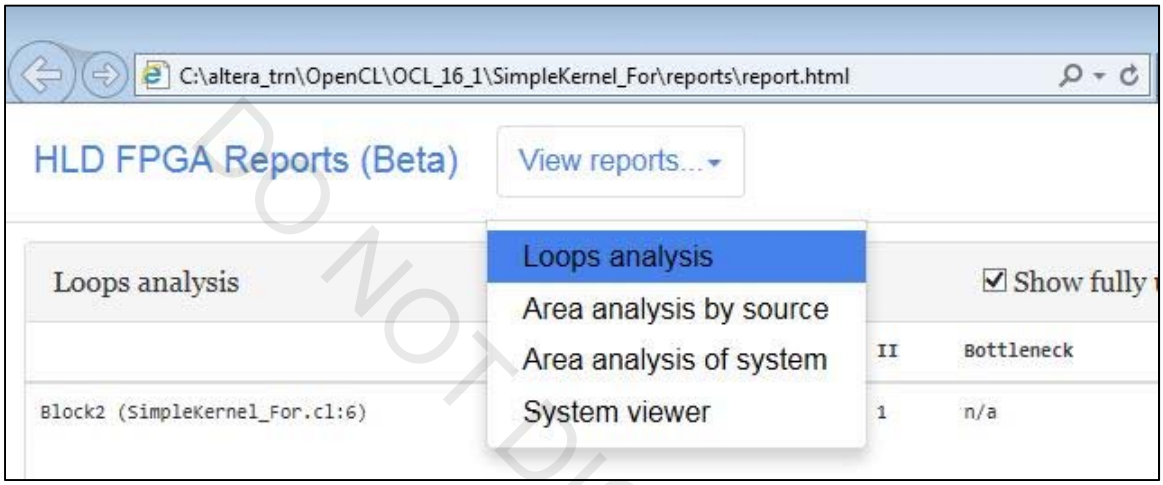
As you can see full compilation resource usage matches very closely with the early estimate.

____ 9. In the Windows explorer, navigate to c:\altera_trn\OpenCL\OCL_16_1\SimpleKernel_For\reports

____ 10. Double click on **report.html** to open it in a web browser

This is the unified kernel compilation static report that's capable of showing many different types of information.

____ 11. First, we'll examine the Loop implementation information by choosing **Loop analysis** in the drop down



This report will show how every loop is implemented.

Since we're dealing with a single-work-item kernel, we expect our loops to be pipelined which is the case here. You'll also see an II value, which is the initialization interval or cycles between iteration launches. II=1 is the best the compiler can do. If the II is > 1 the analysis will also show details about that report.

____ 12. Click on the loop in question. In the example the loop is called Block2.

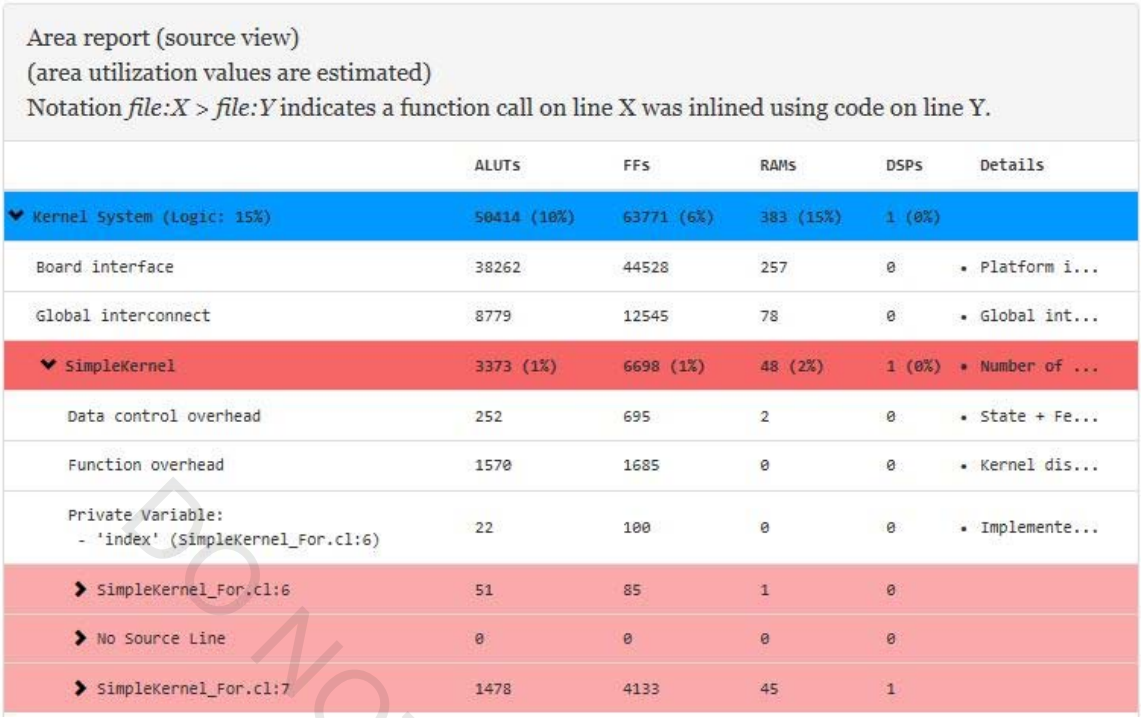
Clicking on the loop will show the actual location of the loop in the source file.

____ 13. Next, let's examine the detailed area report by choosing **Area analysis by source** from the drop-down menu

There are two ways to view the detailed area report by source code or by system blocks. Source code will correspond well to the original source cl file. Blocks will correspond well to the actual circuit created.

____ 14. Expand "Kernel System" and then expand "SimpleKernel"

Now we will see some detailed information and breakdown of resource usage.



Area report (source view)
(area utilization values are estimated)
Notation *file:X > file:Y* indicates a function call on line X was inlined using code on line Y.

	ALUTs	FFs	RAMs	DSPs	Details
Kernel System (Logic: 15%)	50414 (10%)	63771 (6%)	383 (15%)	1 (0%)	
Board interface	38262	44528	257	0	• Platform i...
Global interconnect	8779	12545	78	0	• Global int...
SimpleKernel	3373 (1%)	6698 (1%)	48 (2%)	1 (0%)	• Number of ...
Data control overhead	252	695	2	0	• State + Fe...
Function overhead	1570	1685	0	0	• Kernel dis...
Private Variable: - 'index' (SimpleKernel_For.cl:6)	22	100	0	0	• Implemente...
SimpleKernel_For.cl:6	51	85	1	0	
No Source Line	0	0	0	0	
SimpleKernel_For.cl:7	1478	4133	45	1	

As you can see for our simplekernel, the majority of resource was taking up by the Board interface, which is the logic used by the board support package interfaces and components.

____ 15. Click on several of the pink blocks inside simple kernel.

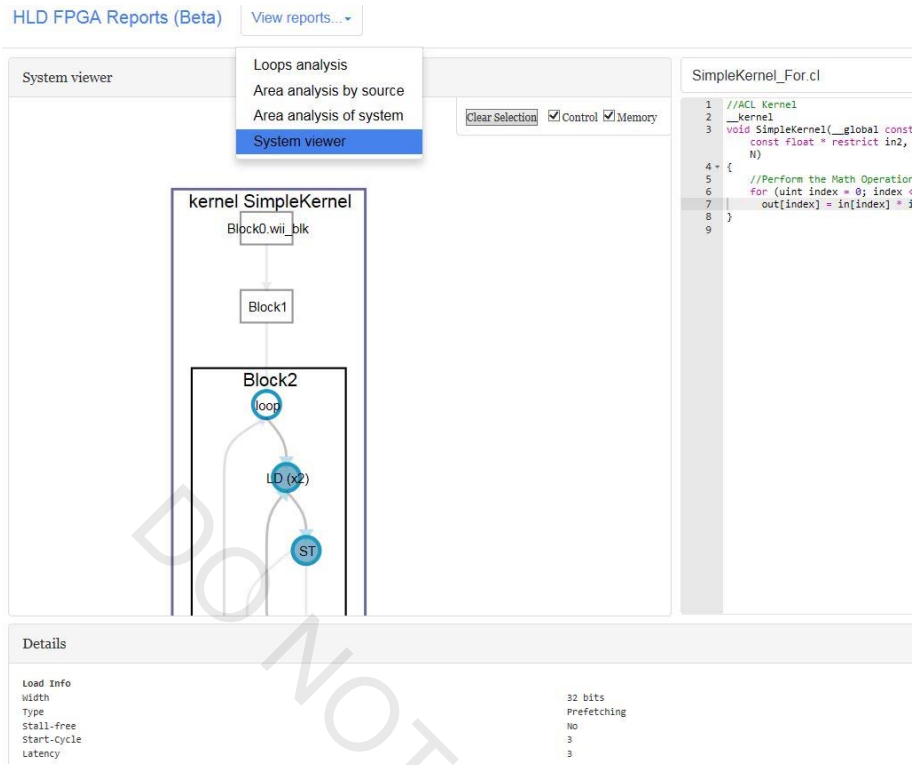
The corresponding line in the source file should be highlighted. You will also see a detailed breakdown of resource utilization by each operator at each of the line numbers.

____ 16. Switch to "Area analysis of system"

Look around this report. You should see information very similar to the Area report (source view) except that the report breaks down to Basic Blocks, which are just branch-free areas of the kernel pipeline. This will correspond very well with the actual implementation on the FPGA.

____ 17. Switch to the "System viewer"

The System view shows the latencies of the individual blocks in the kernel pipeline as well as the load and store to the memory.



____ 18. Click the various blocks in the pipeline including the loop and loop end blocks.

Pay attention to the latency information as well as other information under "Details"

____ 19. Click on the LD and ST units

Notice the type of Load and Store units created. In the AOC implementation, each variable get its own load store units with its own cache. The access to global memory however is arbitrated. Here you can also see the number of access to each memory. If your global or local memory is banked, you will see the number of accesses to each bank.

____ 20. Once you've finished examining this report close the web browser.

Exercise Summary

- Examined the various reports created by the AOC tool which will help you debug and optimize your kernel.

Congratulations!

You have completed Lab 4

