INTEL® FPGA

OpenCL for FPGA Compute Acceleration

Argonne Training Program on Extreme Scale Computing

July 30, 2018

LEGAL NOTICES & DISCLAIMERS

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer. No computer system can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel, the Intel logo, and Stratix10 are trademarks of Intel Corporation in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.

© Intel Corporation.

INTEL PROGRAMMABLE SOLUTIONS GROUP

FPAG/CPLD
Lowest Cost,
Lowest Power

FPGA
Cost/Power Balance
SoC & Transceivers

FPGA
Mid-range FPGAs
SoC & Transceivers

FPGA
Optimized for
High Bandwidth

PowerSoCs
High-efficiency
Power Management

RESOURCES

Embedded Soft and Hard Processors

Nios® II

ARM®

Design Software

Intel® Quartus® Prime
Design Software

Intel® FPGA SDK for OpenCL™

Development Kits

Intellectual Property (IP)

- Industrial
- Computing
- Enterprise

WHAT IS AN FPGA?

- An advanced, multi-function accelerator
- Flexible for highly differentiated products
- Reprogrammable as market dynamics or standards change

What is an FPGA?

An FPGA is an advanced, user-customizable chip

- An FPGA's functions can be reprogrammed based on the computing workload

DSPs

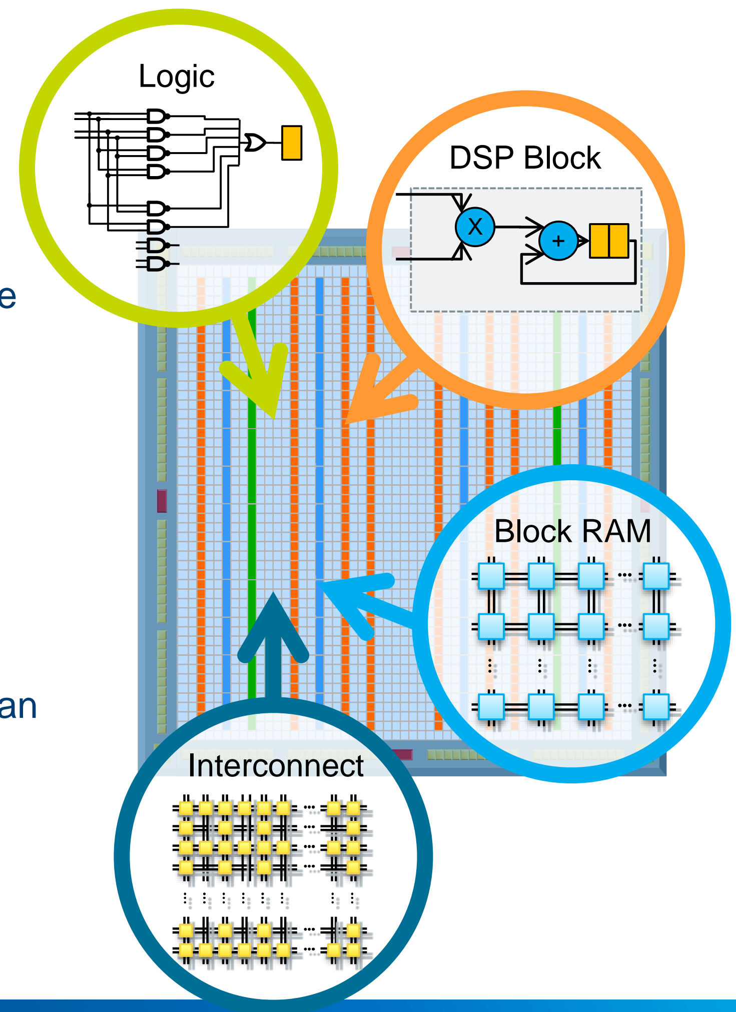
- Dedicated single-precision floating point multiply and accumulators

Block RAMs

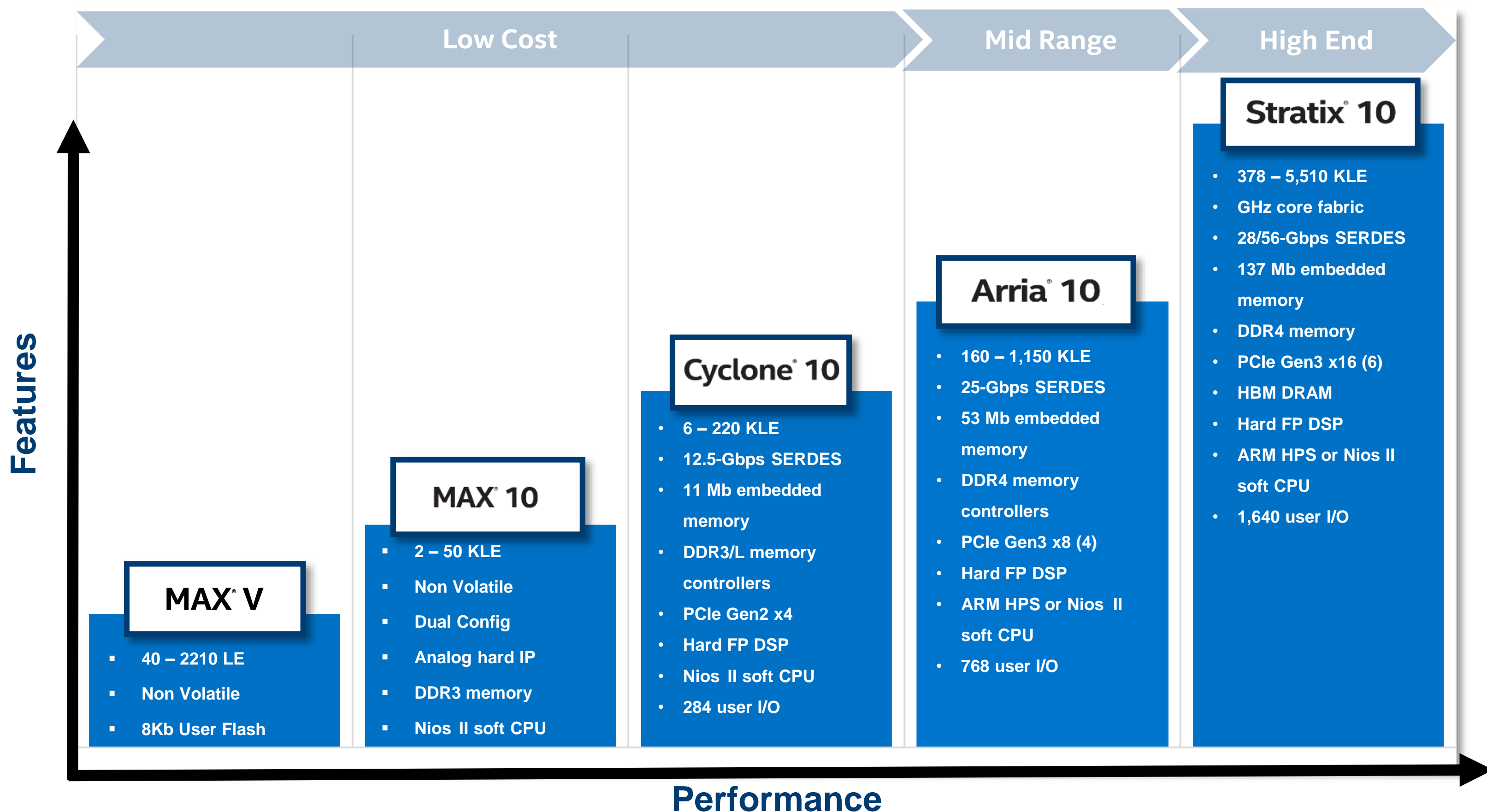
- Small embedded memories that can be stitched to form an arbitrary memory system

Programmable Interconnect

- Programmable logic and routing that can build arbitrary topologies

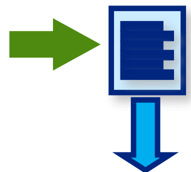


Intel FPGA Portfolio **Options**



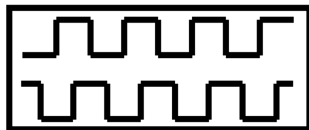
Typical FPGA Design Flow (1/2)

Design specification



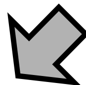
RTL coding (VHDL or Verilog)

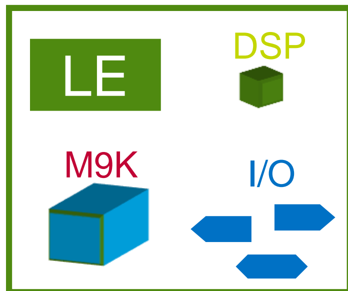
- Behavioral or structural description of design
- Possibly with the help of high level tools



RTL functional simulation

- Mentor Graphics ModelSim* - Intel® FPGA Edition or other 3rd party simulators
- Verify logic model & data flow (no timing delays)





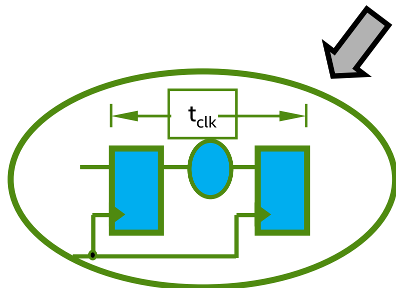
Synthesis (Mapping)

- Translate design into device specific primitives
- Optimization to meet required area & performance constraints
- Intel® Quartus® Prime Software synthesis or those available from 3rd party vendors
- *Result*: Post-synthesis netlist

Typical FPGA Design Flow (2/2)

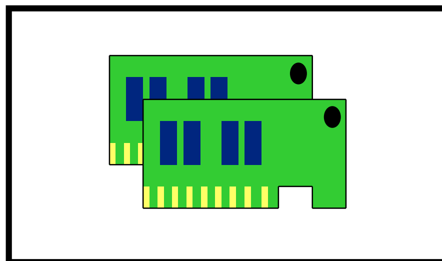
Place & route (Fitting)

- Map primitives to specific locations inside target technology with reference to area & performance constraints
- Specify routing resources to be used
- *Result:* Post-fit netlist



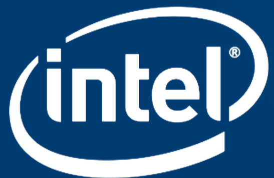
Timing analysis

- Verify performance specifications were met
- Static timing analysis



PC board simulation & test

- Simulate board design
- Program & test device on board
- Use on-chip tools for debugging



INTRODUCTION TO PARALLEL COMPUTING WITH OPENCL™

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos

Agenda

Approaches to Parallel Programming

Data Sharing and Synchronization

Overview of OpenCL™

OpenCL Platform and Host-side Software

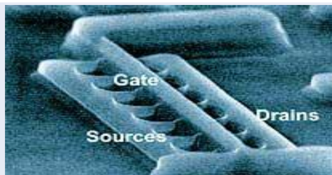
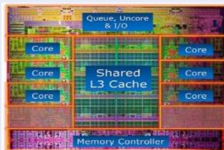
Executing OpenCL Kernels

Compiling software into circuits

The Intel® FPGA SDK for OpenCL™

Hands-on Example

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

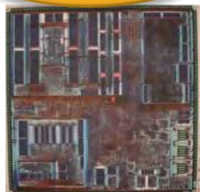
Heterogeneous Computing Systems

- 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

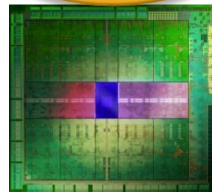
Traditional Approach to Heterogeneous Computing

- Write software for each software programmable architecture CPU, GPU, DSP
  - Using different languages and vendor specific tools
- Develop custom parallel hardware for FPGA
  - Fine-grained parallelism
  - Write HDL, Simulate, close timing, in-system debug, etc.

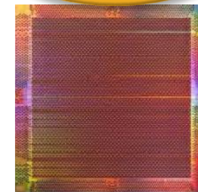
CPUs



DSPs



GPUs



FPGAs

Programmer-Specified Parallelism

Allow software programmer to define and control parallelism

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

Types of Parallelism

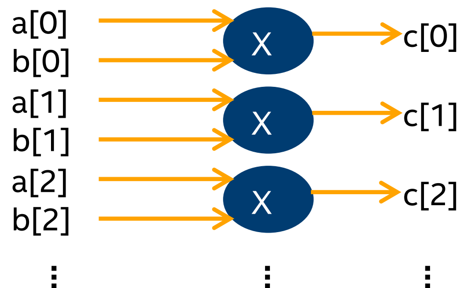
- Data Parallelism
  - Input data separated and sent to parallel resources, results recombined
  - Scatter-gather
- Task Parallelism
  - Decompose problem into sub-problems that run well on available compute resources
  - Divide-and-conquer
- Pipeline Parallelism
  - Task parallelism where tasks have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

Data Parallelism

Same operation(s) applied across different data in parallel

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

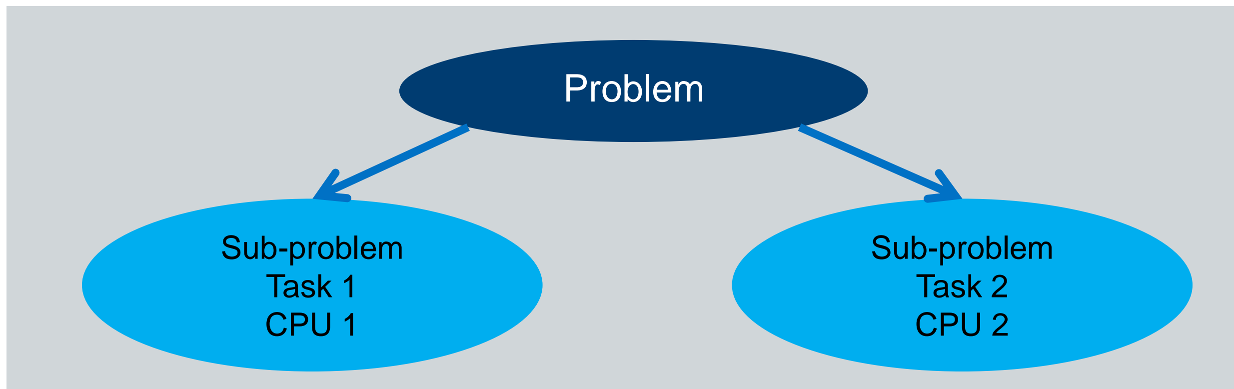
```
for (i = 0; i < N ; i++)
      c[i] = a[i] * b[i]
```



Task Parallelism

Decompose problem into sub-problems (tasks)

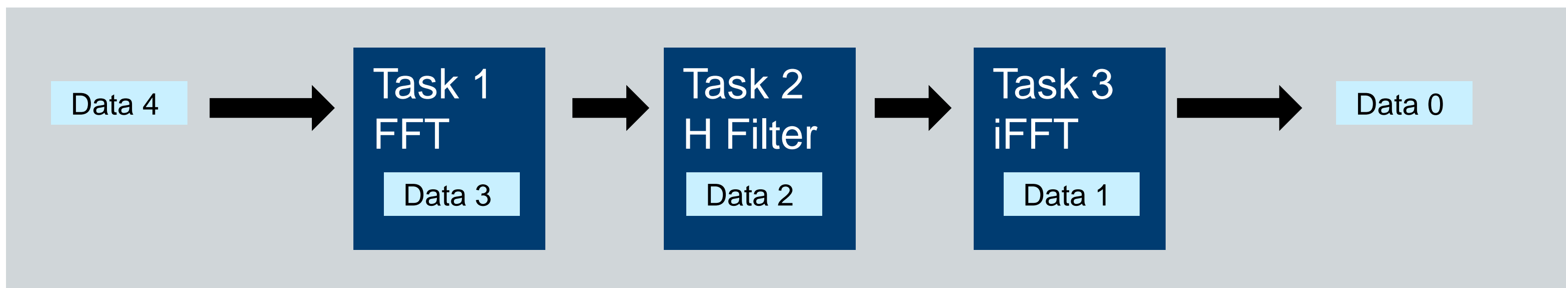
- Tasks operate on same or different data
- Example: Multi-CPU system where each CPU execute a different thread
- A.K.A. Simultaneous Multithreading (SMT), Thread/Function Parallelism



Pipeline Parallelism

Task parallelism where tasks have a producer consumer relationship

- Operates on pipelined data
  - Different tasks operate in parallel on different data
- Example
  - Task 1 – FFT, Task 2 – Frequency Filter, Task 3 – Inverse FFT



Agenda

Approaches to Parallel Programming

Data Sharing and Synchronization

Overview of OpenCL™

OpenCL Platform and Host-side Software

Executing OpenCL Kernels

Compiling software into circuits

The Intel® FPGA SDK for OpenCL™

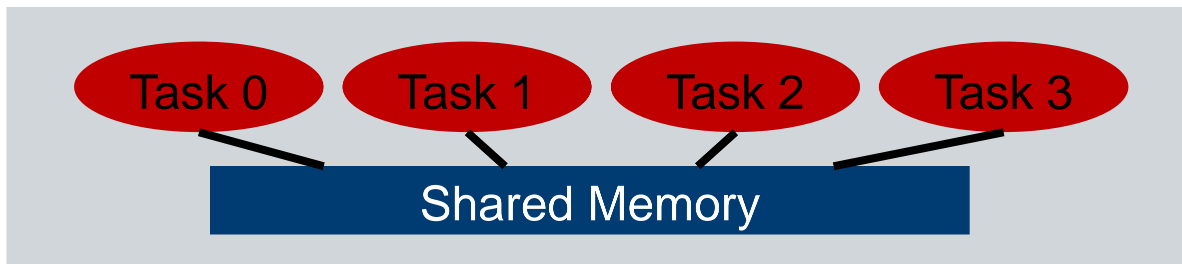
Hands-on Example

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 (mutex)
    - Enforce limits on access of particular resources
  - Parallel computing environment must handle this effectively

Shared Memory Model

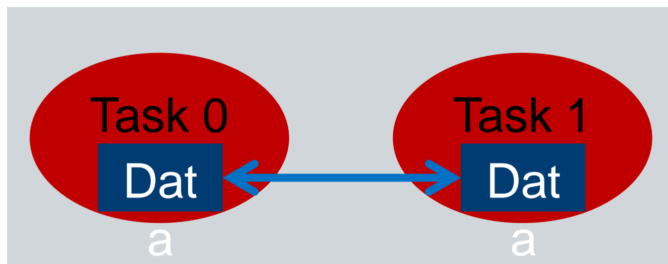
- Operates on Global view of memory accessible by tasks
  - Used for inter-task communication
  - Maybe guarded by semaphores or mutexes (barriers and locks)



Advantage	Drawback
Programmer not required to manage data movement, so code development is simpler	The overhead of shared busses and coherency becomes the limiting factor

Message Passing Model

- Explicit communication between concurrent tasks



Advantage

- Scalable
  - Tasks can run on an arbitrary number of devices

Drawbacks

- Programmer needs to explicitly manage communications
- Difficult to make portable since it uses specific libraries

Agenda

Approaches to Parallel Programming

Data Sharing and Synchronization

Overview of OpenCL™

OpenCL Platform and Host-side Software

Executing OpenCL Kernels

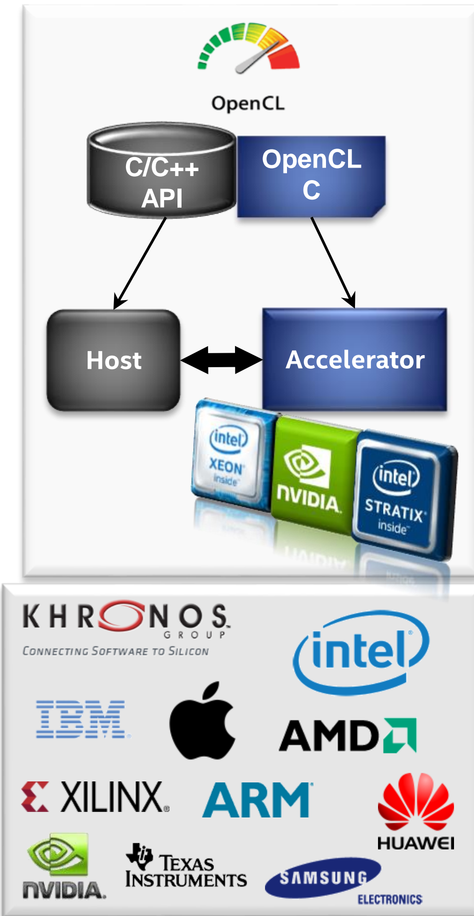
Compiling software into circuits

The Intel® FPGA SDK for OpenCL™

Hands-on Example

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

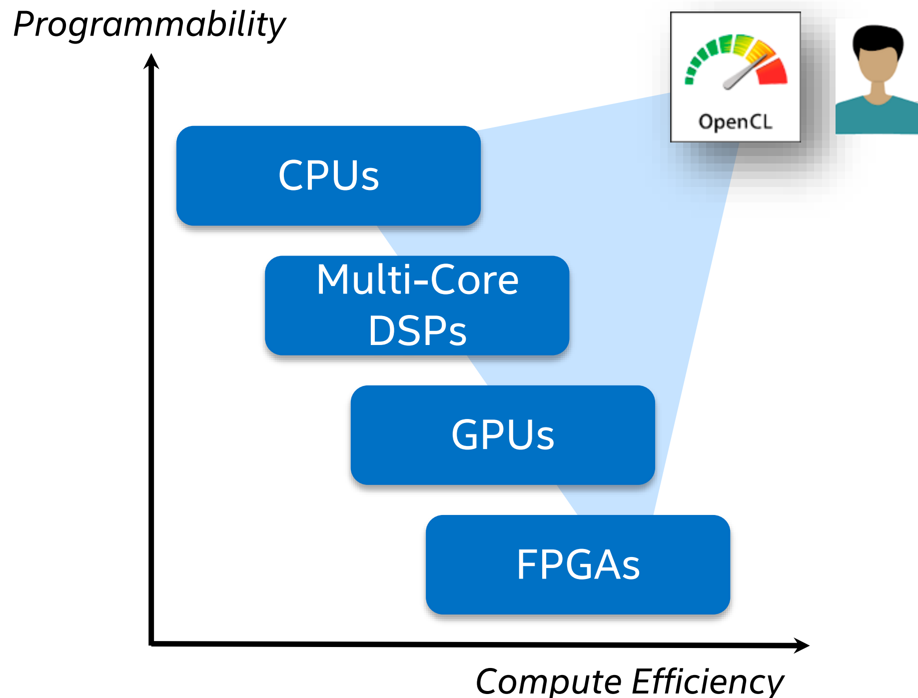


*Other names and brands may be claimed as the property of others

OpenCL Provides A Single Environment for Heterogeneous Platforms

Enables a single user with one skillset to target multiple platforms

- Enables developer to optimize for performance
- Automates heterogenous system stitching



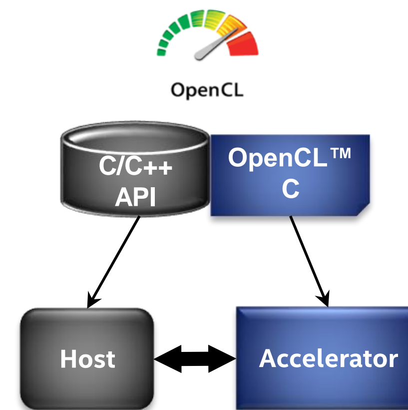
OpenCL™ Characteristics

- 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™ 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

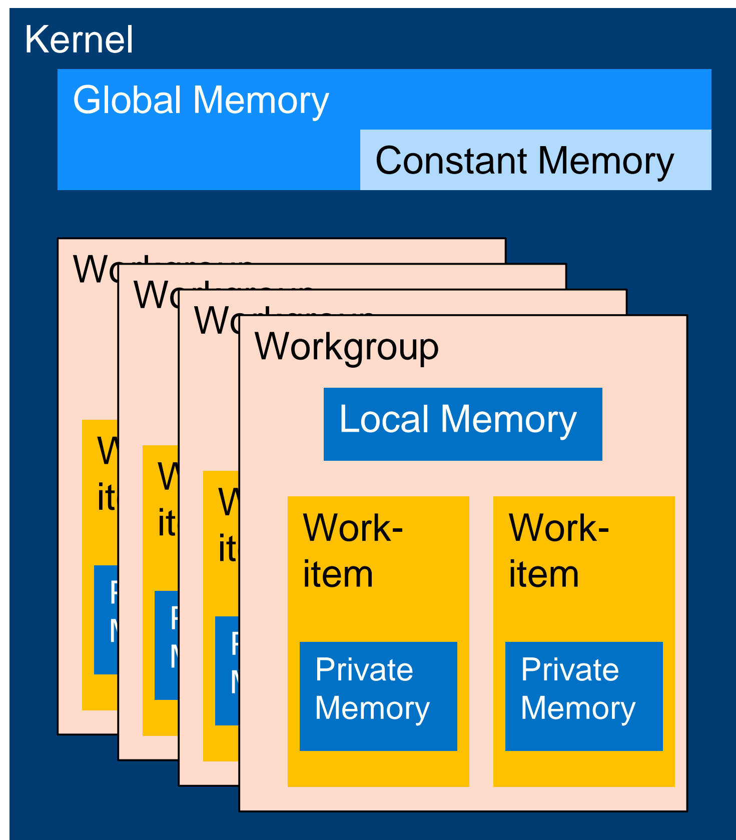
Two Sides of OpenCL™ Standard



- 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™ 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



OpenCL™ by Example

Consider the following C program

C

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

Called in this manner

```
void main()
{
  …
  vecadd(a,b,c,N);
}
```

Kernels

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
- When compiled for an Intel® FPGA, kernels can provide pipeline parallelism

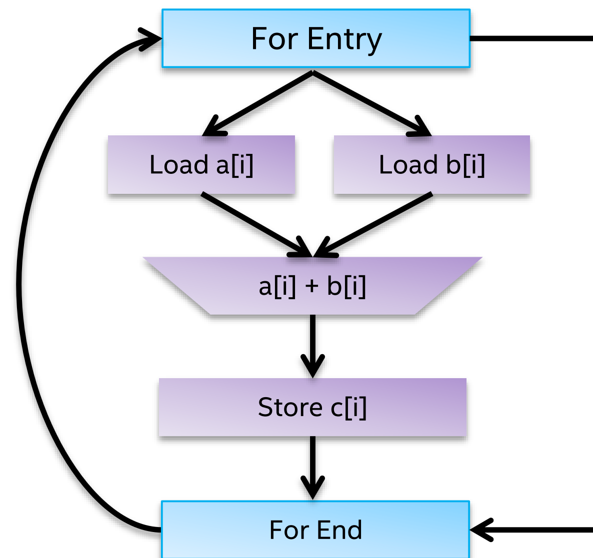
OpenCL™ by Example – Single Work-Item Kernel

Kernel compiled into dataflow circuit with flow control

- Either **single work-item** or **NDRange**

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

aoc



See OpenCL: Single-threaded vs. Multi-threaded Kernels for more information:
https://www.altera.com/support/training/course/oopnclkern.html

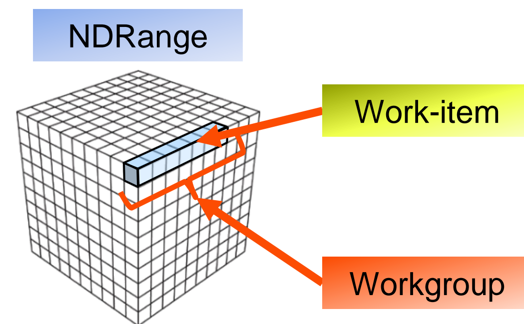
NDRange Kernels

Execute an OpenCL™ kernel across multiple data-parallel threads

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

Hierarchy of work-items

- Work-items are grouped into workgroups
- Work-items within a workgroup can explicitly synchronize and share data
- Workgroups are always independent



OpenCL™ by Example – NDRange Kernel

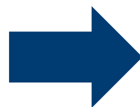
Kernel represents a single iteration of loop to perform vector operation

- N work-items will be generated to match array size
- `get_global_id(0)` function returns index of work-items 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];
}
```

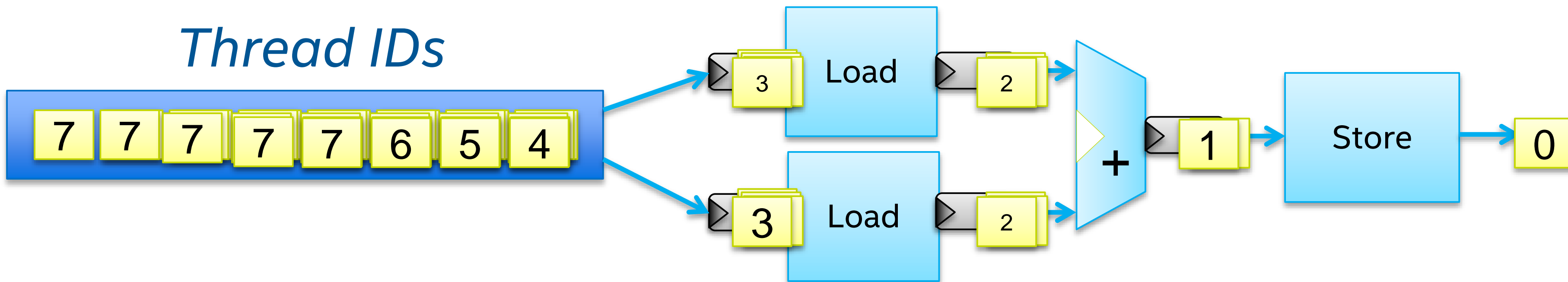


OpenCL™ Kernel

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

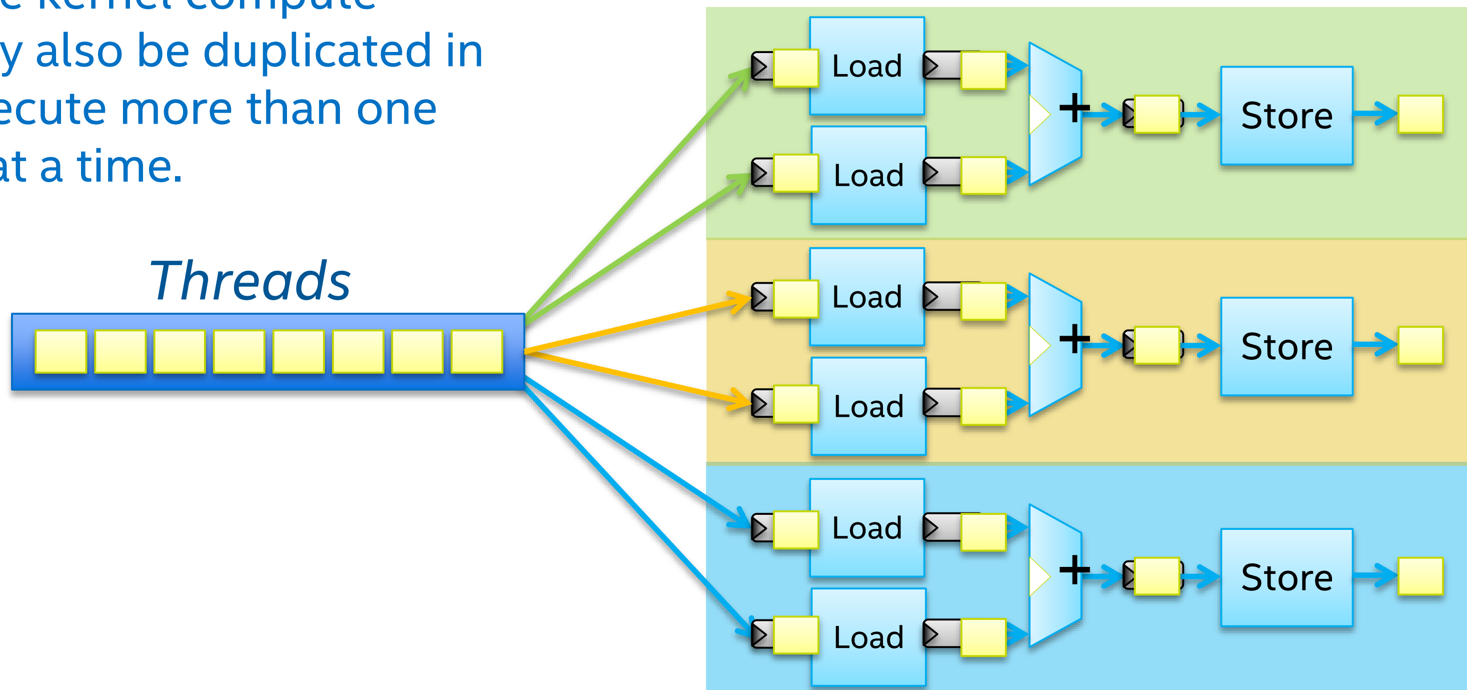
Pipeline Parallelization of NDRange Kernels

- 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



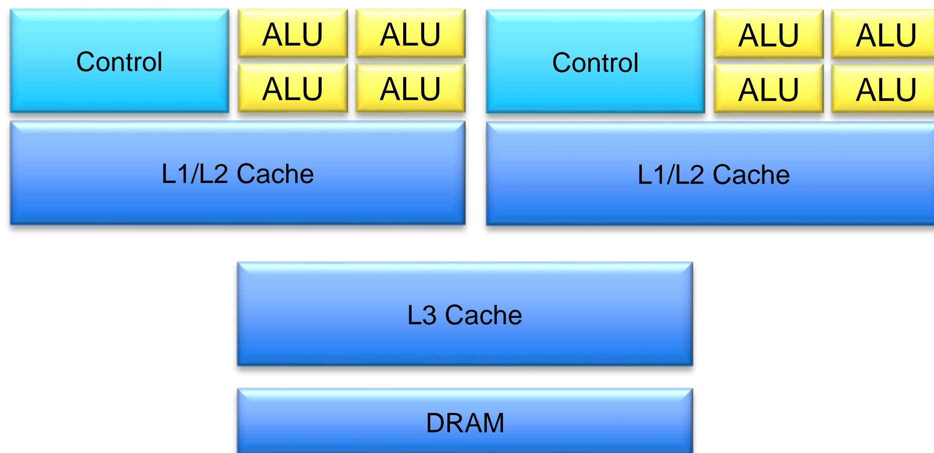
Data Parallelization of NDRange Kernels

An NDRange kernel compute engines may also be duplicated in order to execute more than one work-item at a time.



CPU Architectures

- **Optimized for latency:** large caches, hardware prefetch
- **Complicated control:** superscalar, out-of-order execution
- **Comparatively few execution units**



Running OpenCL™ on CPU Architectures

- Workgroups executed across different cores
- Target vector units
  - Fuse work-items together
  - Vectorize kernels to work with explicit vector types
- Work-item synchronization is handled in software
- Data-sharing between work-items in a workgroup relies on caches

GPU Architectures

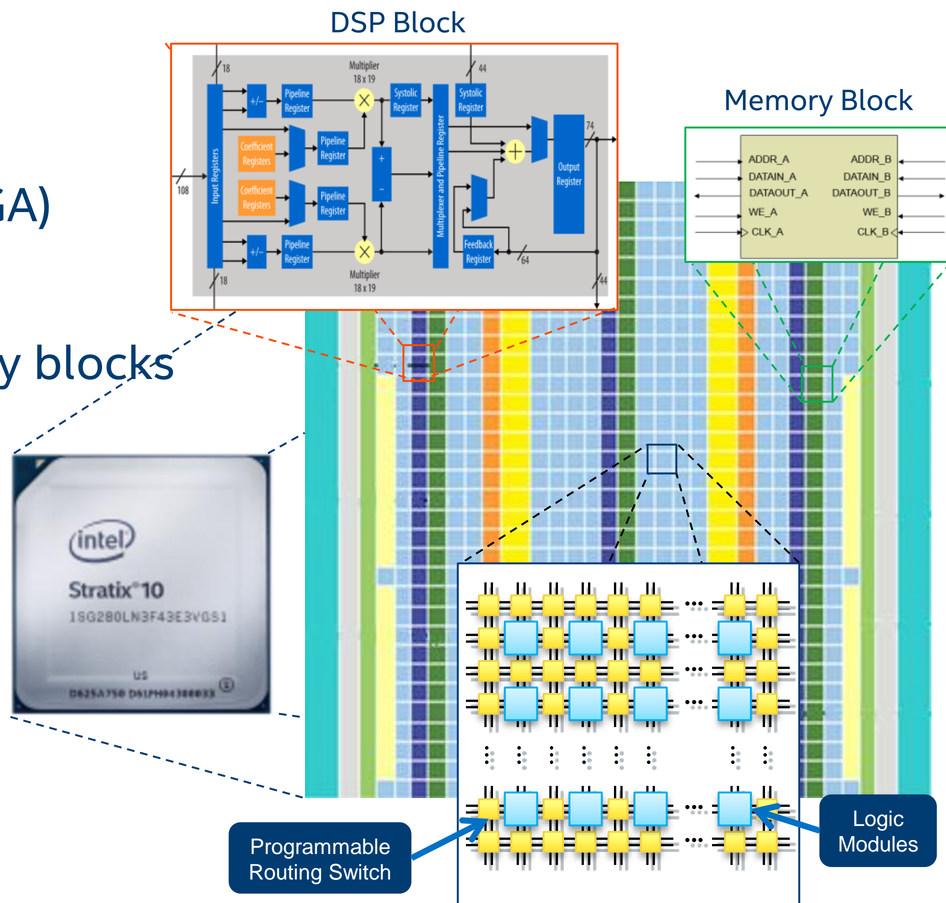
- Many Compute Units
  - Up to 128
- Wide memory bus
  - Multiple channels
  - High bandwidth
  - High latency
- Small read/write caches
- PCIe* board

Running OpenCL™ on GPU Architectures

- **Workgroups** distributed across compute units
- **Work-items** execute in parallel on the vector-like cores
  - Dedicated hardware resources used for synchronization and data-sharing between work-items within a workgroup
- Run **enough** work-items per compute unit to **mask latencies**
  - Translates into a requirement for thousands of work-items!
- Work-items contend for fixed resources (registers, local memory)
- Hardware limits how many work-items/workgroup per compute unit

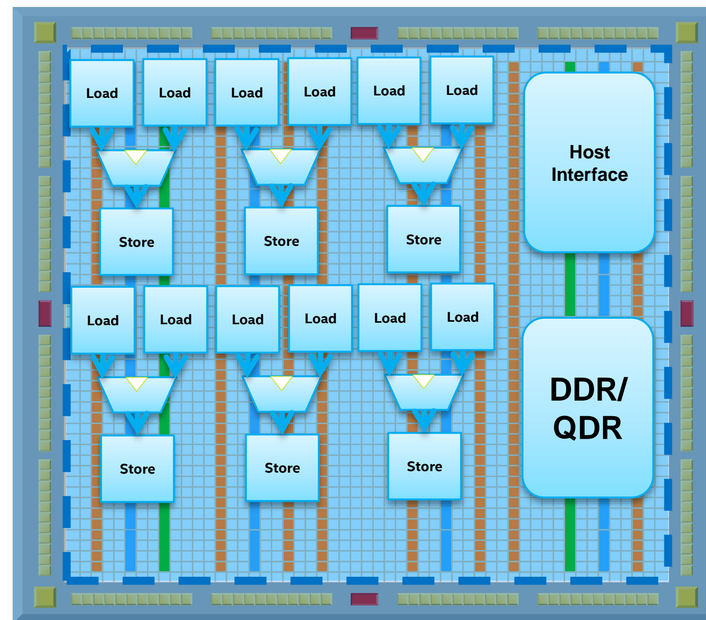
FPGA Architecture

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

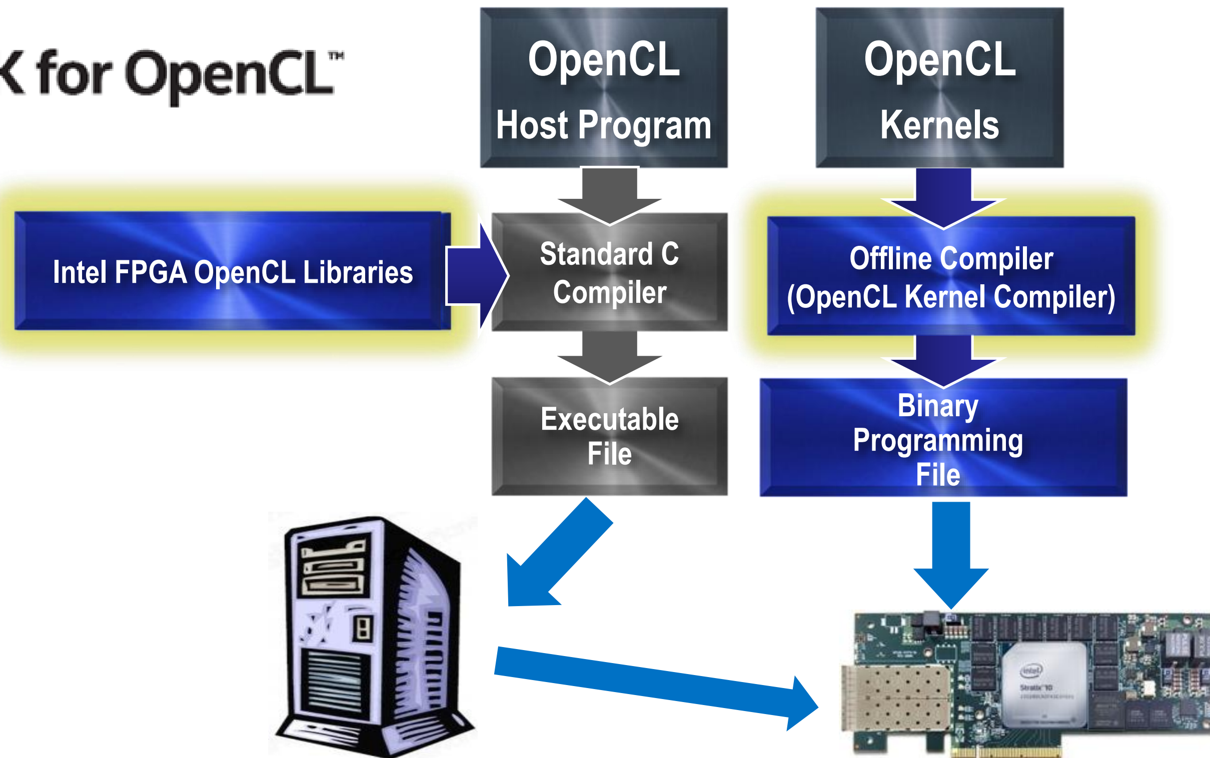


Compiling OpenCL™ to Intel® FPGA

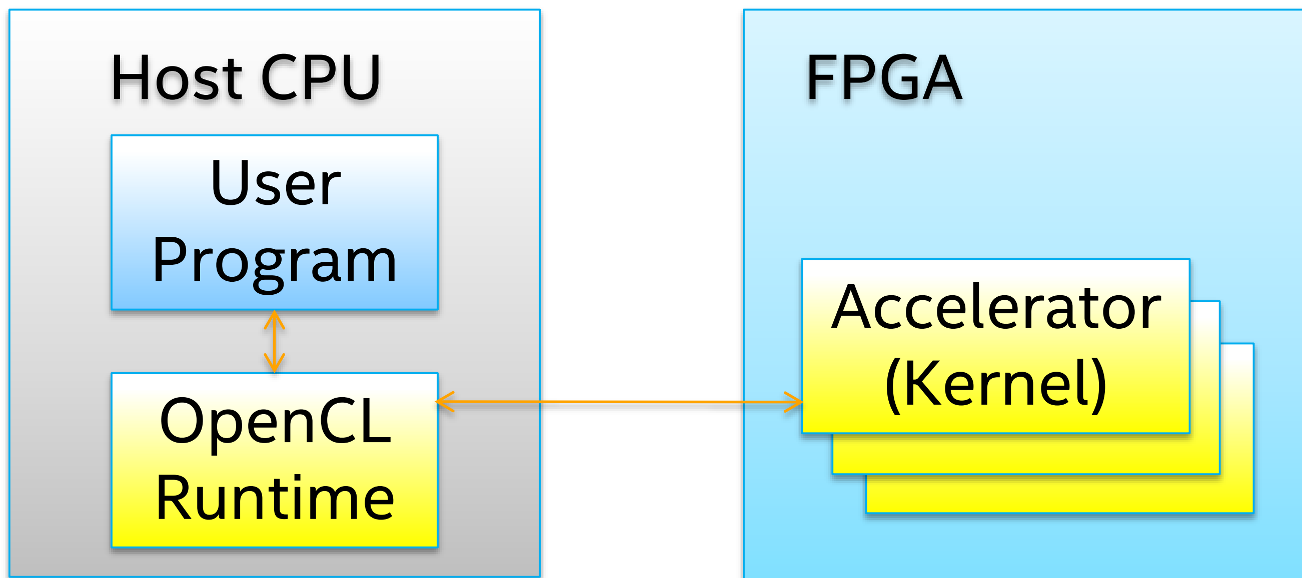
- Custom hardware generated automatically for each kernel
  - Get the advantages of the FPGA without the lengthy design process
- Organized into functional units based on operation
- Able to execute OpenCL™ threads in parallel



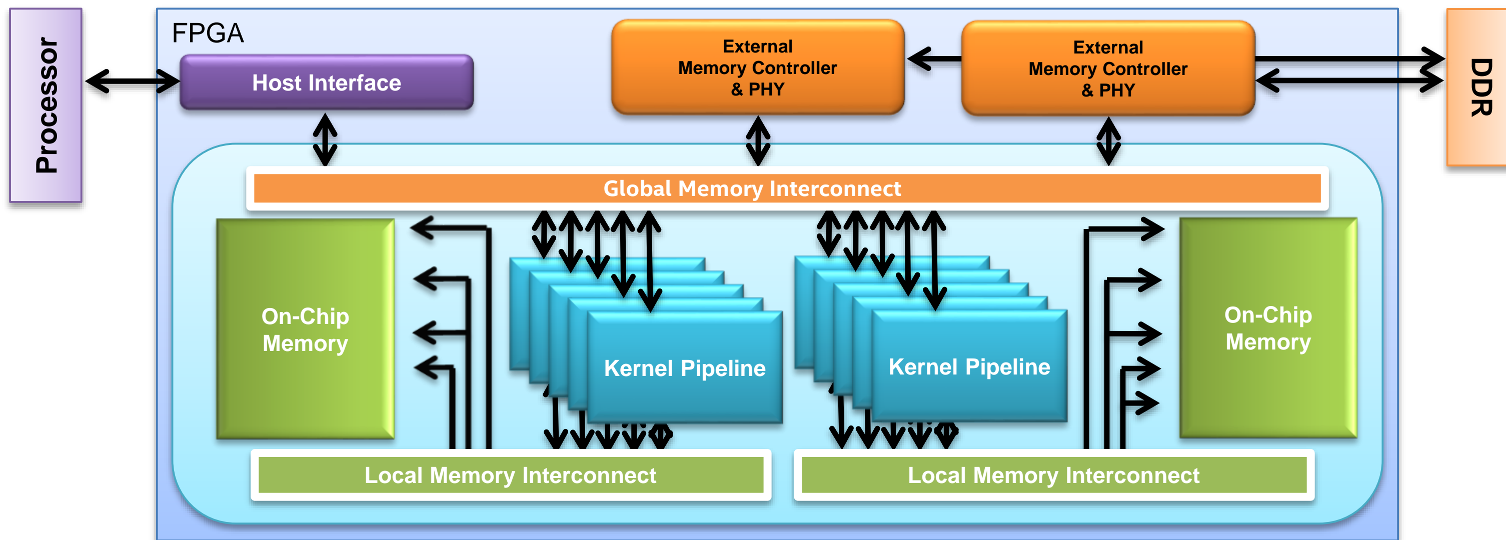
Intel® FPGA SDK for OpenCL™ Usage



OpenCL™ and Intel® FPGAs

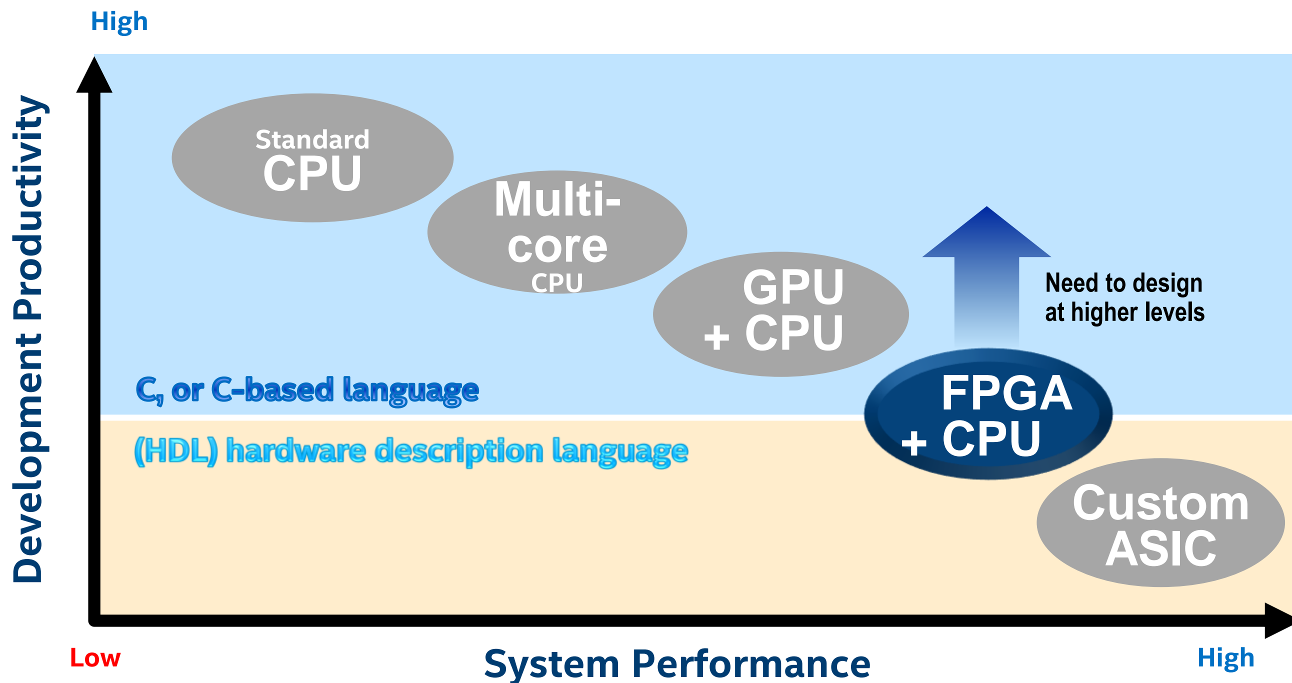


Compiler Builds the Complete FPGA



The Intel® FPGA SDK for OpenCL™ builds the FPGA:
accelerators ~ all the data paths ~ all memory structures

Improving FPGA Development Productivity

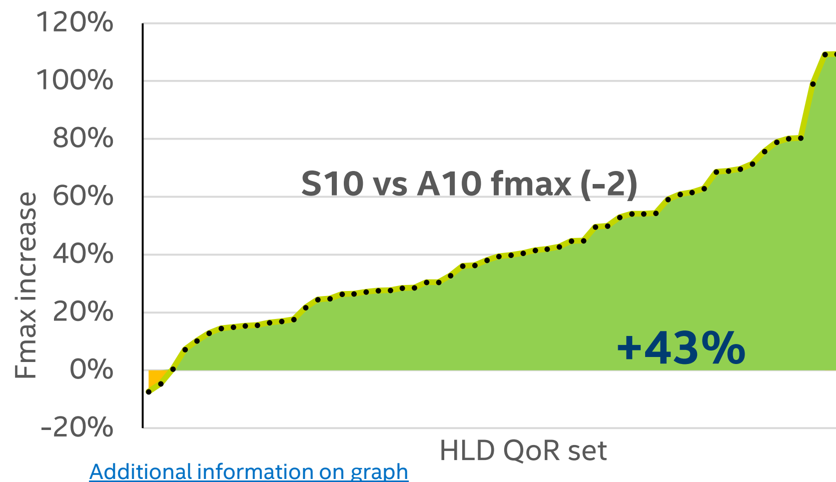


OpenCL™ results with HLD Stratix 10 compiler

- Out of the box designs, *optimized* RTL
  - Demonstrating the Stratix 10 advantage
  - Customers can reproduce PoC results

Design	Scaling vs. A10	Devkit (-2) demo
FFT	8x	0.4 Gpts/s
Back projection	3.4x	14.6 ms
GZIP	7.2x	4 x 4.9 GB/s
MM 32x14	3.8x	1.77/2.5 Tflops
MM 50x14	5.1x	2.27/3.7 Tflops

Additional information on table



Agenda

Approaches to Parallel Programming

Data Sharing and Synchronization

Overview of OpenCL™

OpenCL Platform and Host-side Software

Executing OpenCL Kernels

Compiling software into circuits

The Intel® FPGA SDK for OpenCL™

Hands-on Example

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

Platform Model Device Structure

OpenCL™ Platform Model

Host Memory

Host

Compute Device

Global Memory

CU

PE PE PE ... PE PE

CU

PE PE PE ... PE PE

Board

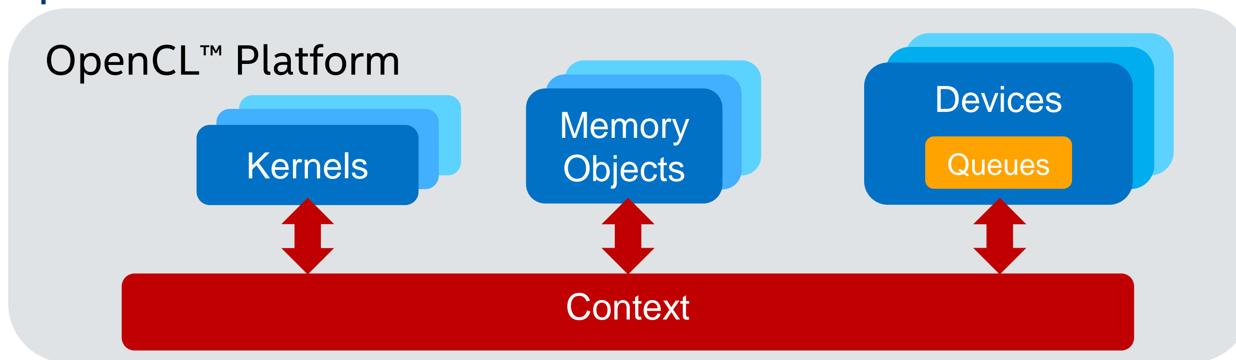
Global Memory

CU

Context

Environment within which kernels execute

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



Platform Layer 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

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*: Properties that define context behavior
- *num_devices*: Number of elements in *devices*
- *devices*: List of devices in context
- *pfn_notify*: Callback function to be registered to handle errors in the context
- *user_data*: Data argument for *pfn_notify*
- *errcode_ret*: 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

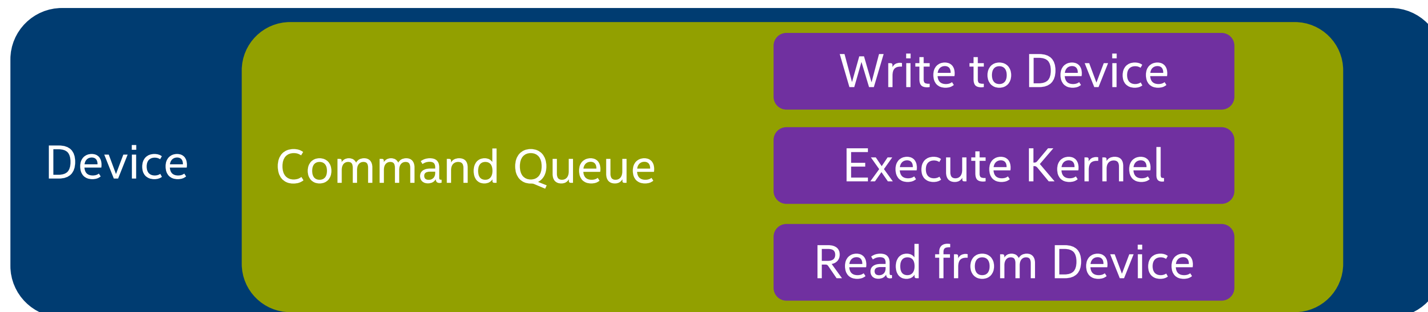
Example Platform Layer Code

```
//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);
```

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



Runtime Layer API

Execute kernels on the device

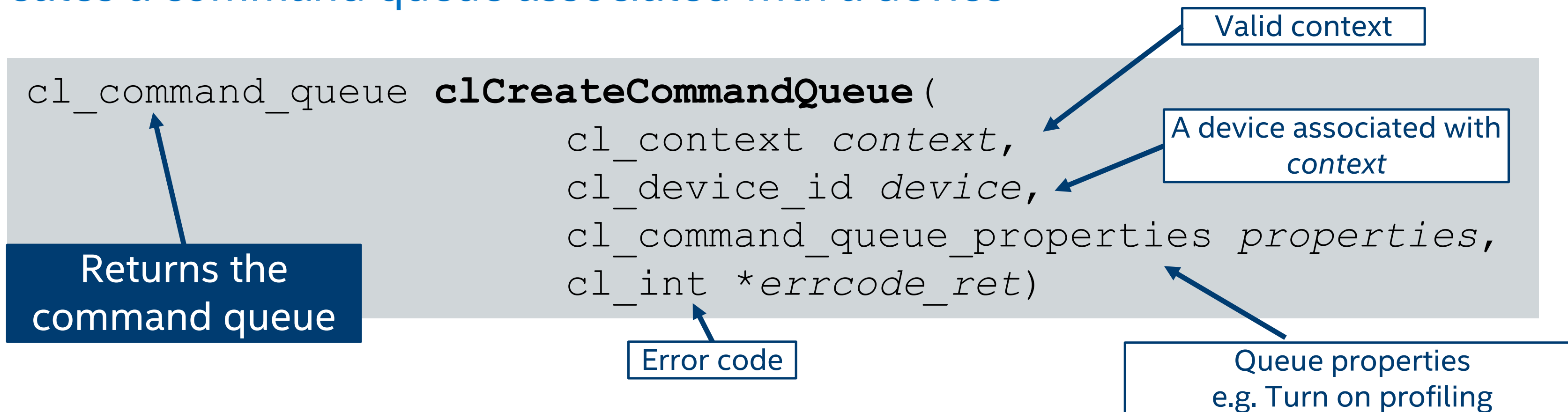
- Tasks
  - Memory management
    - Allocate/deallocate device memory
    - Read/write to the device
  - Run kernels on the device
  - Host/device synchronization

Typical Runtime Layer Steps

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

Create a Command Queue

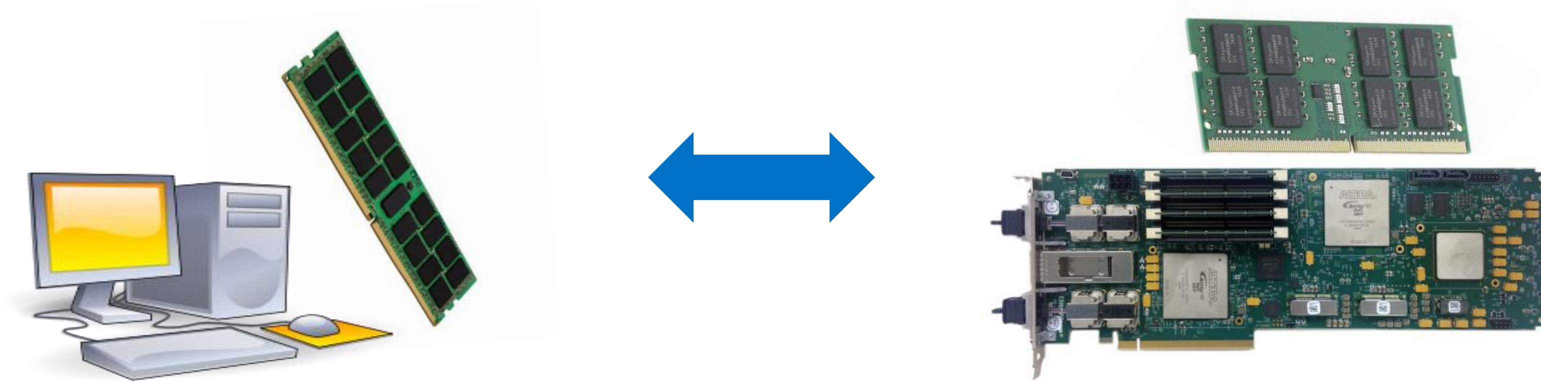
Creates a command queue associated with a device



- 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

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)

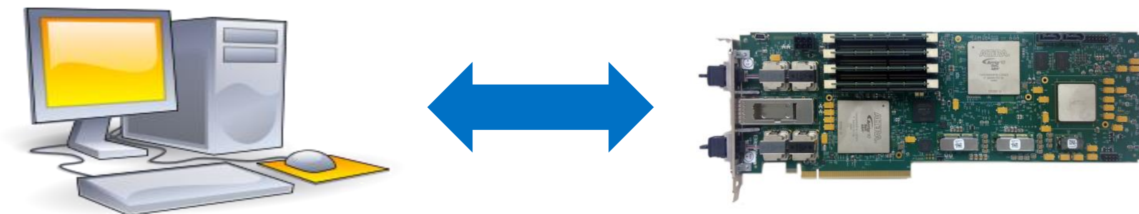
Error code

- 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

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

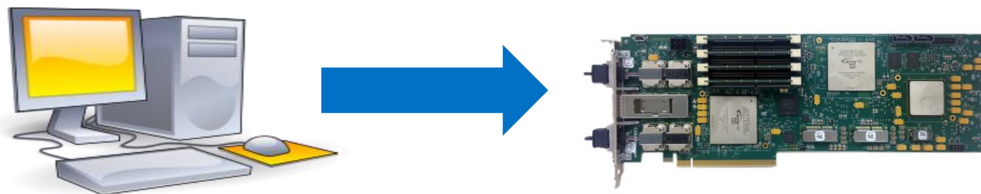


clEnqueueWriteBuffer

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 → `command_queue`
- Error code → `cl_int`
- Destination Buffer → `buffer`
- Set to CL_TRUE blocks call until *ptr* can be reused by the host → `blocking_write`
- Offset in bytes in the buffer → `offset`
- Size in bytes of data to be written → `cb`
- Source host pointer → `ptr`
- Events used for synchronization. Discussed later → `num_events_in_wait_list`, `event_wait_list`, `event`



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 → `command_queue`
- Error code → `cl_int`
- Source Buffer → `buffer`
- Set to CL_TRUE blocks call until buffer data copied to *ptr* → `blocking_read`
- Offset in bytes in the buffer → `offset`
- Size in bytes of data to be read → `cb`
- Destination host pointer → `ptr`
- Events used for synchronization. Discussed later → `num_events_in_wait_list`, `event_wait_list`, `event`

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);
```

Agenda

Approaches to Parallel Programming

Data Sharing and Synchronization

Overview of OpenCL™

OpenCL Platform and Host-side Software

Executing OpenCL Kernels

Compiling software into circuits

The Intel® FPGA SDK for OpenCL™

Hands-on Example

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 Restrictions

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

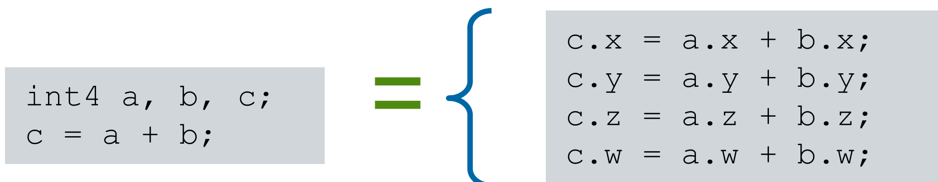
OpenCL™ Data Types

- 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
- Use variable for vector operation
- Or, use the components



Kernel Example

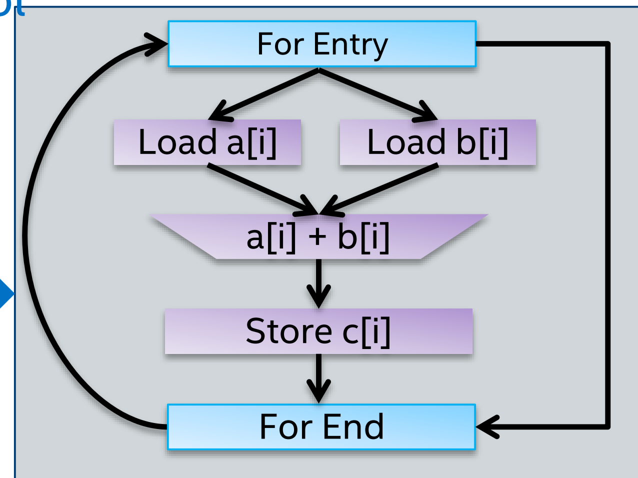
```
__kernel void my_kernel ( __global float *a,
                          __global float *b,
                          __global float *c,
                          int N)
{
    int index;
    for (index = 0; index < N; index++)
        c[index] = a[index] + b[index];
}
```

Compilation Example

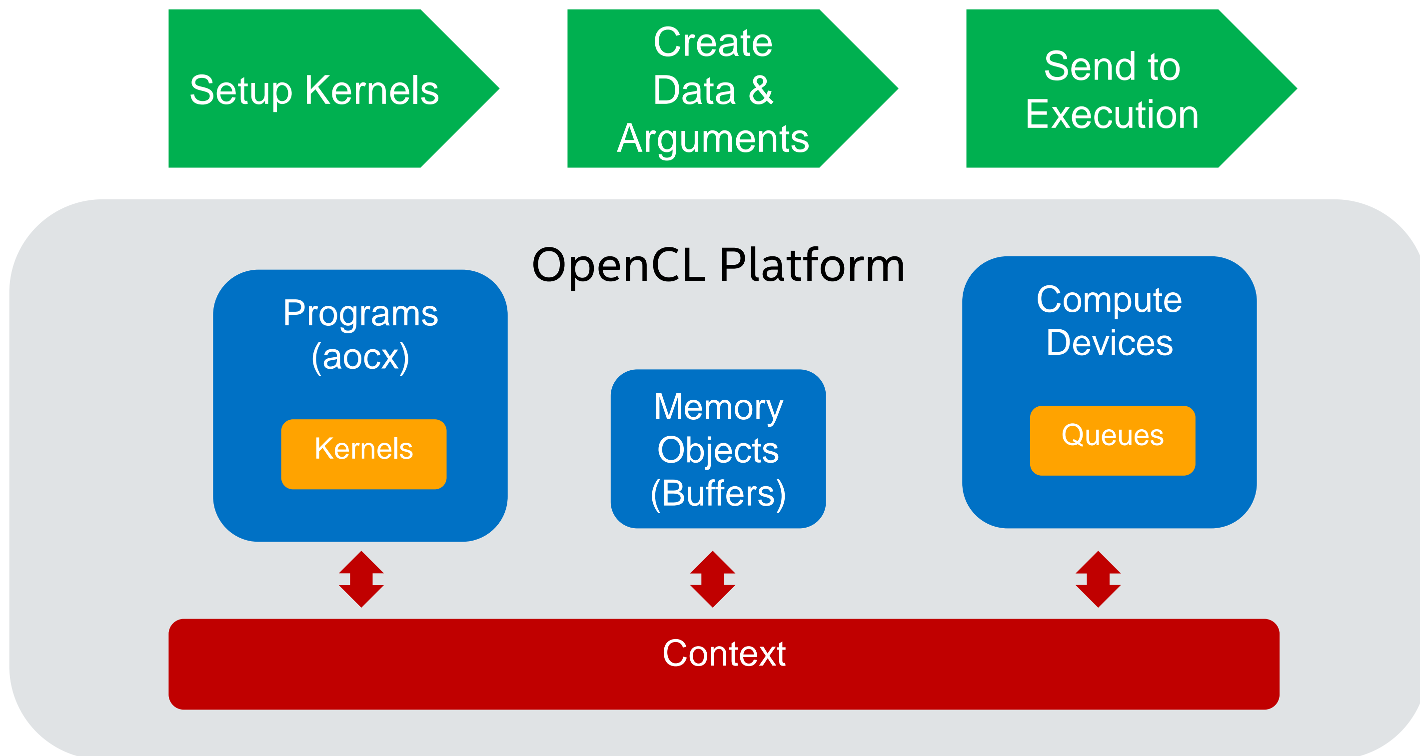
Kernel compiled into dataflow circuit with flow control

- 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];
}
```



OpenCL™ Execution Flow



Programs and Kernels

Process for host to execute a kernel on a device

1. Create program (a program is a collection of kernels)
   – 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

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)
```


- 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)
```

Program object to build

Number of devices in *device_list*

Compatible devices

Build Options

Error code

Callback function for when build has completed

Data for callback function

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

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)
```

- `cl_int` – Error code
- `kernel` – Kernel object
- `arg_index` – Argument index: from 0 (leftmost arg) to *N*-1 (*N* is the total number of args)
- `arg_size` – Size of argument value
- `arg_val` – Pointer to data used as argument value

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

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)
```


– `clEnqueueNDRangeKernel` discussed later

Kernel Execution 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 program
  cl_program program =CreateProgramWithBinary(context, 1, &device, &binary_length,
                                    (const unsigned char**)&binaries, &kernel_status,
&clError);

   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;
   err = clEnqueueWriteBuffer(queue, a_device, CL_TRUE, 0,
     NUM_ELEMENTS*sizeof(cl_float), a_host, 0, NULL, NULL);
```

Kernel Execution Example Cont.

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

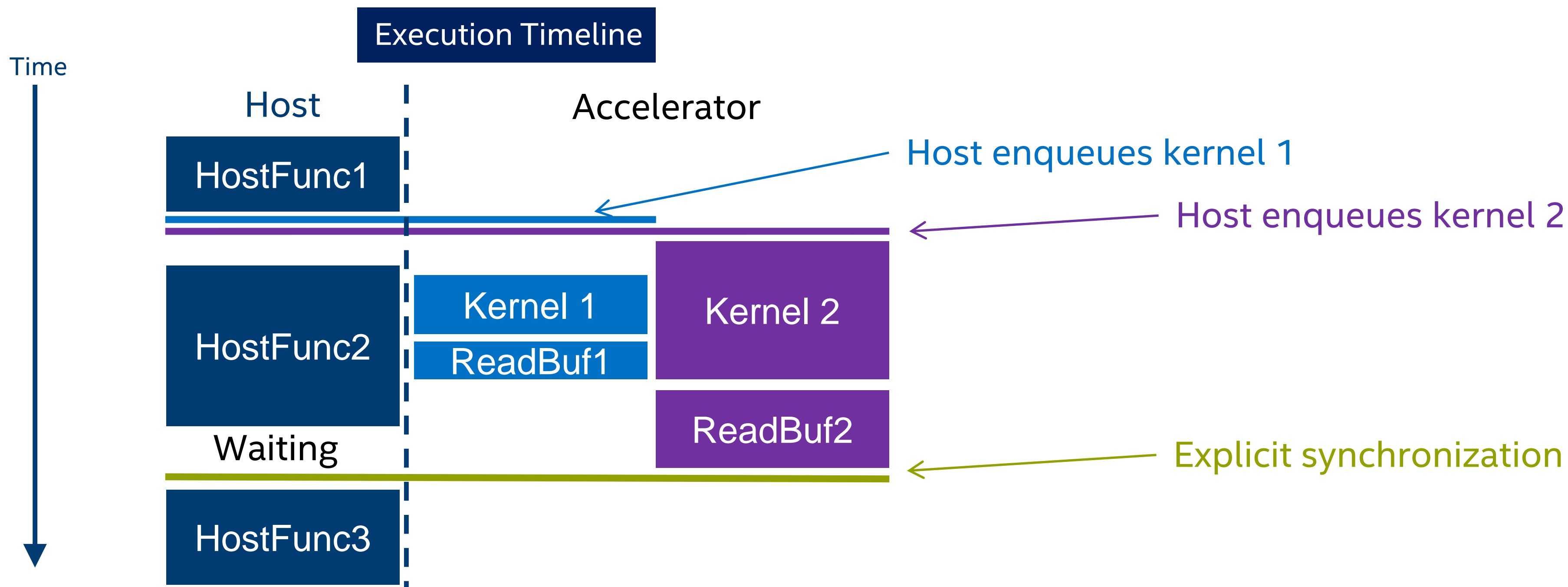
```
    ...

    // 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);


   // 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



Explicit Synchronization Points

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

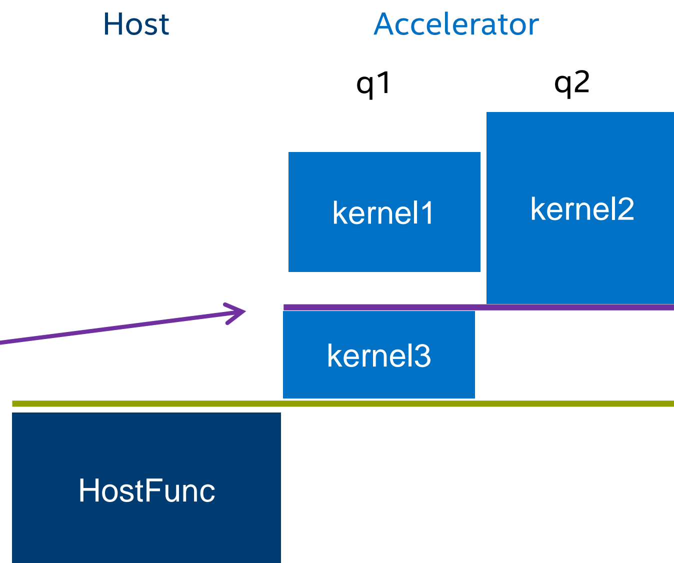
Event Dependencies

- 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);
```

Agenda

Approaches to Parallel Programming

Data Sharing and Synchronization

Overview of OpenCL™

OpenCL Platform and Host-side Software

Executing OpenCL Kernels

Compiling software into circuits

The Intel® FPGA SDK for OpenCL™

Hands-on Example

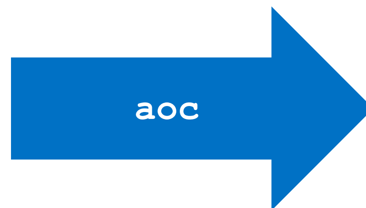
*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos

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];
}
```



Compile Kernels

Run the Offline Compiler

- Set `AOCL_BOARD_PACKAGE_ROOT` environment variable to the path of the board support package
  - Usually `%INTELFPGAOCLSDKROOT%/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

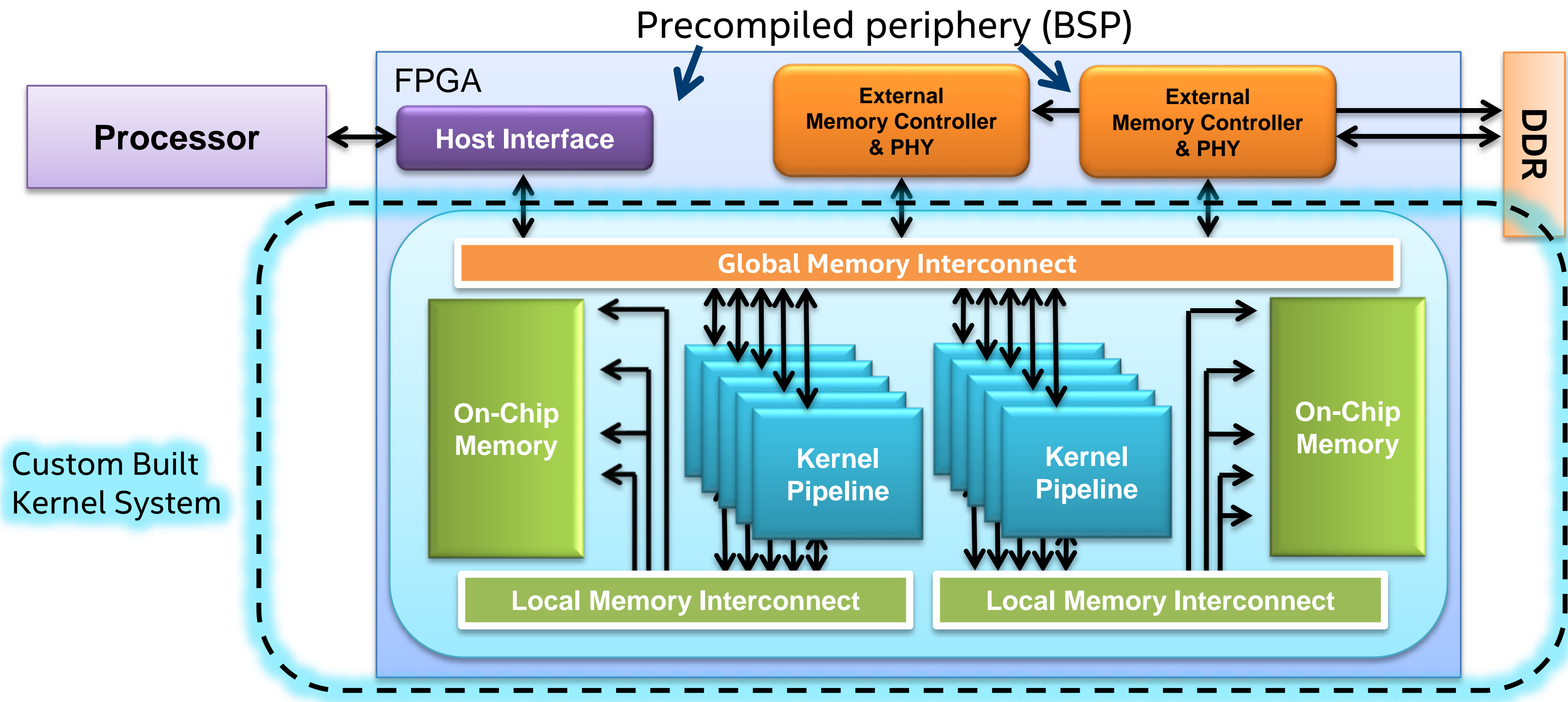
- <kernel file>.aoco
  - Intermediate object file representing the created hardware system
- <kernel file>.aocx
  - Kernel executable file used to program FPGA
- Inside <kernel file> folder
  - <kernel file>.log
    - Compile log including estimated resource usage, optimization report, and compile messages
  - <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

Each kernel is converted into custom hardware.

- Precompiled hardware interfaces used
  - Provided by the Board Support Package (BSP)
  - PCIe* or Hard Processor System (HPS), memory controller, kernel interface, clock generator, DMA
- OpenCL™ memory model implemented
  - Global -> DDR, QDR
  - Local -> On-chip memory
  - Private -> On-chip registers

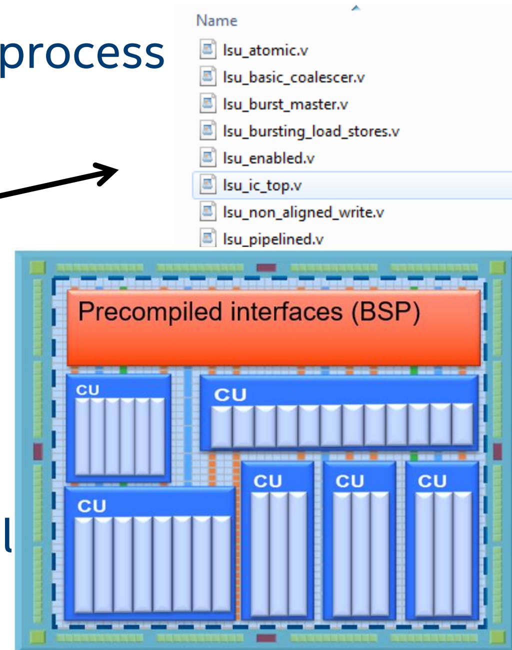
FPGA Architecture for OpenCL™ Implementation



OpenCL™ Kernels to Dataflow Circuits

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

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



Mapping Multi-Threaded Kernels to FPGAs (Naïve)

~~Simplest way of mapping kernel functions to FPGA may appear to be replicating hardware for each work-item~~ (thread)

- Problems:
  - NDRange size tend to be really large
  - Inefficient and wasteful
    - FPGA compute bandwidth is often NOT the bottleneck of system
  - Unknown at kernel compile time the number of work-items to run
- A better method involves taking advantage of pipeline parallelism
- AOC may optionally replicate HW to process multiple work-items in parallel
  - See the *Intel FPGA SDK for OpenCL Best Practices Guide*

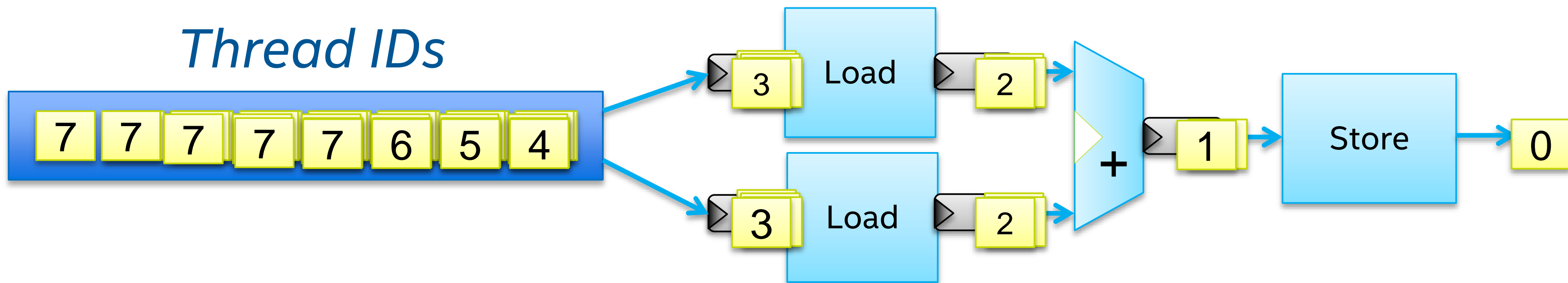
Mapping Multi-Threaded Kernels to FPGAs

Take 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

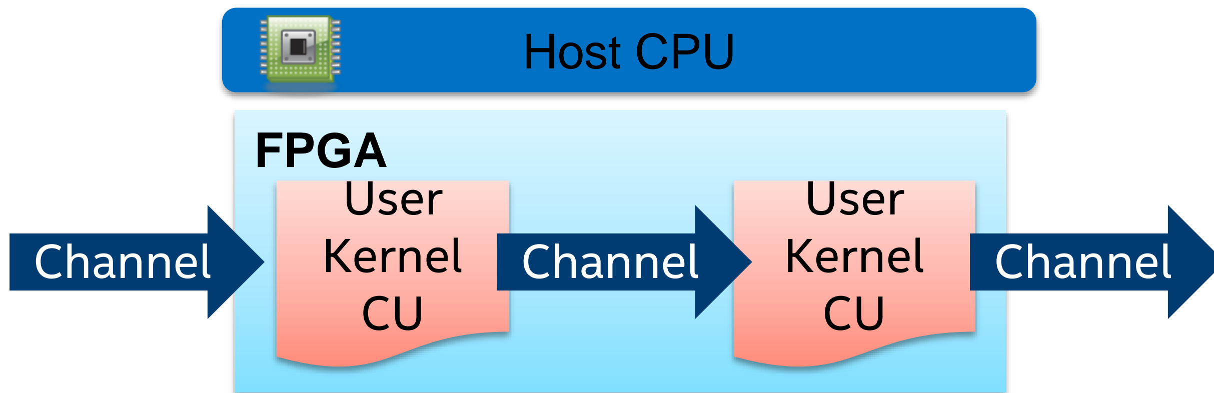
- On each cycle the portions of the pipeline are processing different threads
- While work-item 2 is being loaded, work-item 1 is being added, and work-item 0 is being stored



Channels / Pipes

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

- Implemented with FIFOs by AOC
- Kernels have the ability to autorun if host interaction is not needed

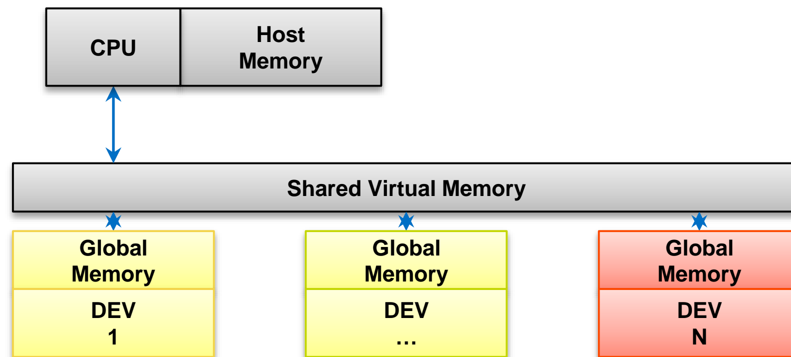


See the Optimizing OpenCL™ instructor-led training or the best practices guide for more information

Shared Virtual Memory (SVM)

Hosted Heterogeneous Platform with SVM

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



Agenda

Approaches to Parallel Programming

Data Sharing and Synchronization

Overview of OpenCL™

OpenCL Platform and Host-side Software

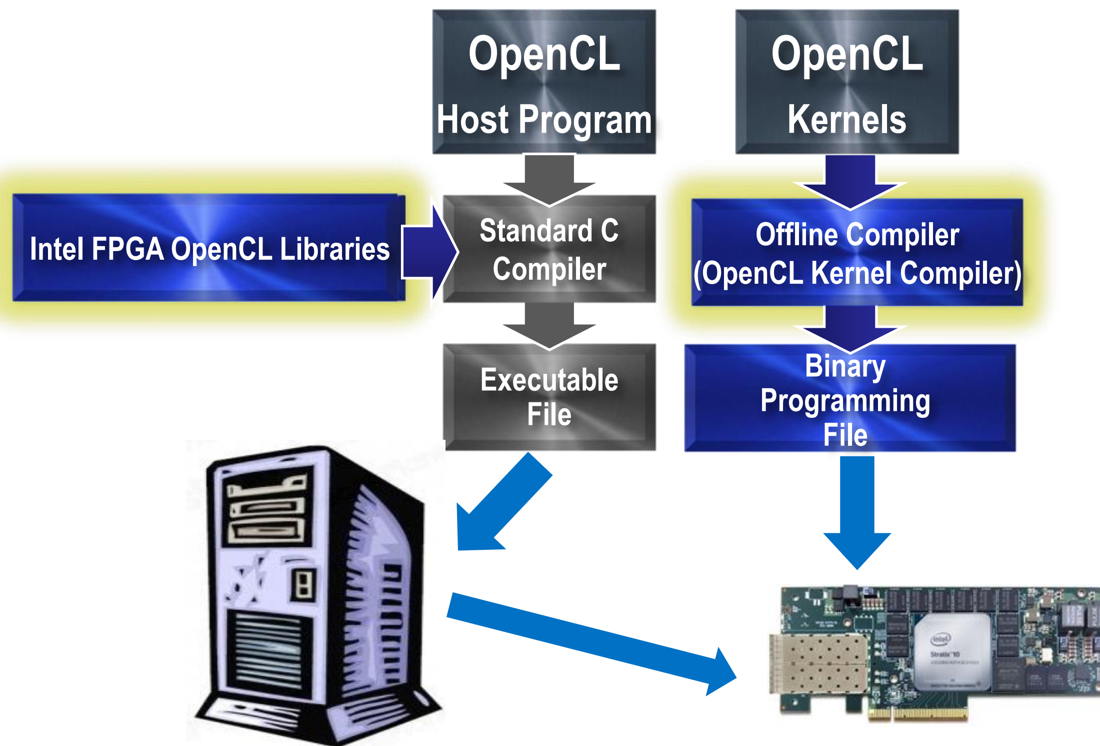
Executing OpenCL Kernels

Compiling software into circuits

The Intel® FPGA SDK for OpenCL™

Hands-on Example

Intel® FPGA SDK for OpenCL™ 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

Software Requirements

- Operating system: 64-bit, Windows* or Linux* (RHEL* or CentOS*)
- Intel® FPGA SDK for OpenCL™, plus license
- Intel Quartus® Prime Software, plus license
  - For most boards, the Pro edition will be required
  - Appropriate devices installed
  - Same version as SDK
- C compiler
  - E.g. Microsoft* Visual Studio* or GCC*
  - Needed to compile the host program
  - Able to compile and link 64-bit code (except when targeting a SoC host)

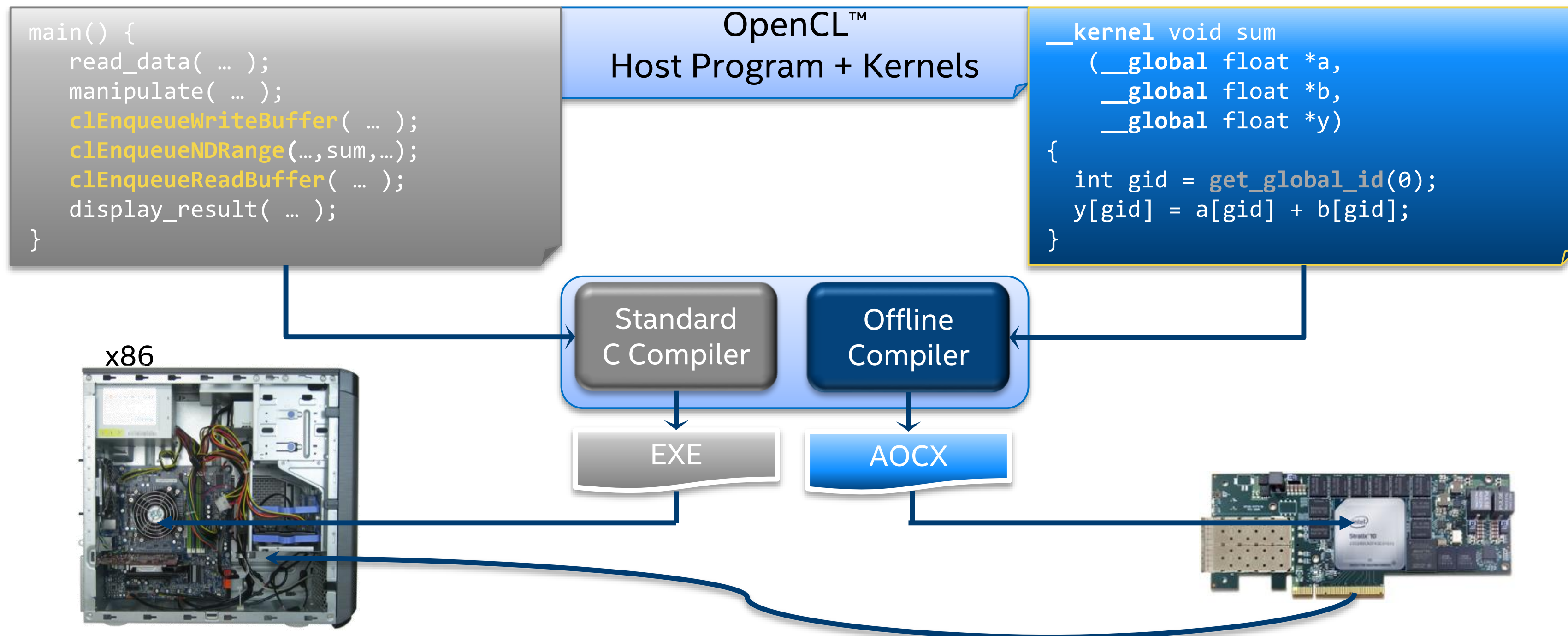
*Other names and brands may be claimed as the property of others

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 dri

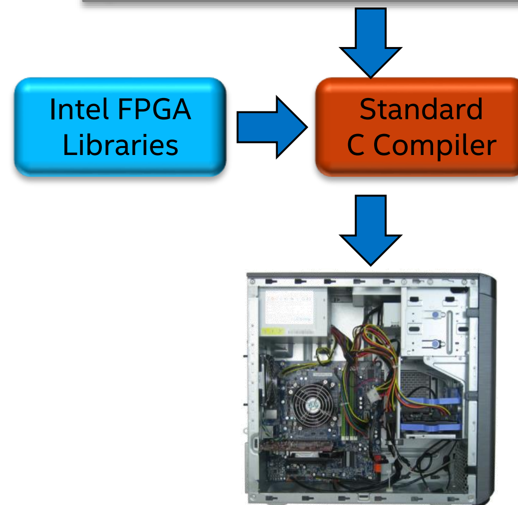
*Other names and brands may be claimed as the property of others

SDK Compile Flow



Compiling the Host Program

- Use a conventional C compiler (Visual Studio*/GCC*)
- Add `%INTELFPGAOCLSDKROOT%/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 `%INTELFPGAOCLSDKROOT%/host/<OS>/lib` directory
    - Recommended to use `aocl link-config`



Compiling Kernels with the Offline Kernel Compiler

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

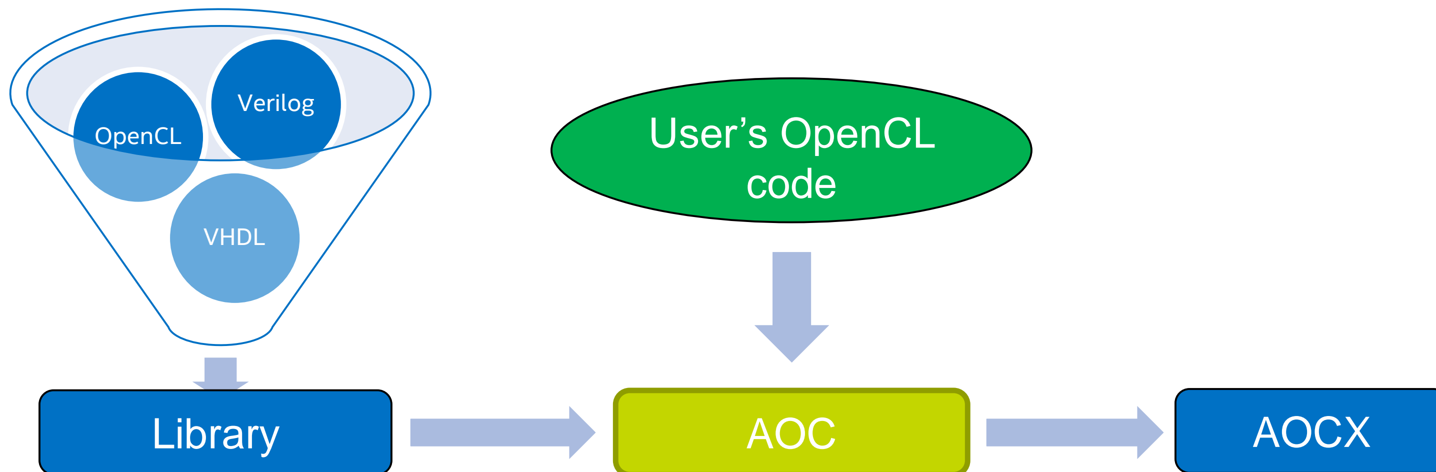
Option	Description
`--help or -h`	Help for the tool
`-c`	Creates .aoco object file and sets up a Quartus Prime hardware design project
`--board <board name>`	Compile for the specified board
`--list-boards`	Prints a list of available boards

- 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.

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

Offline Compiler (aoc) Options

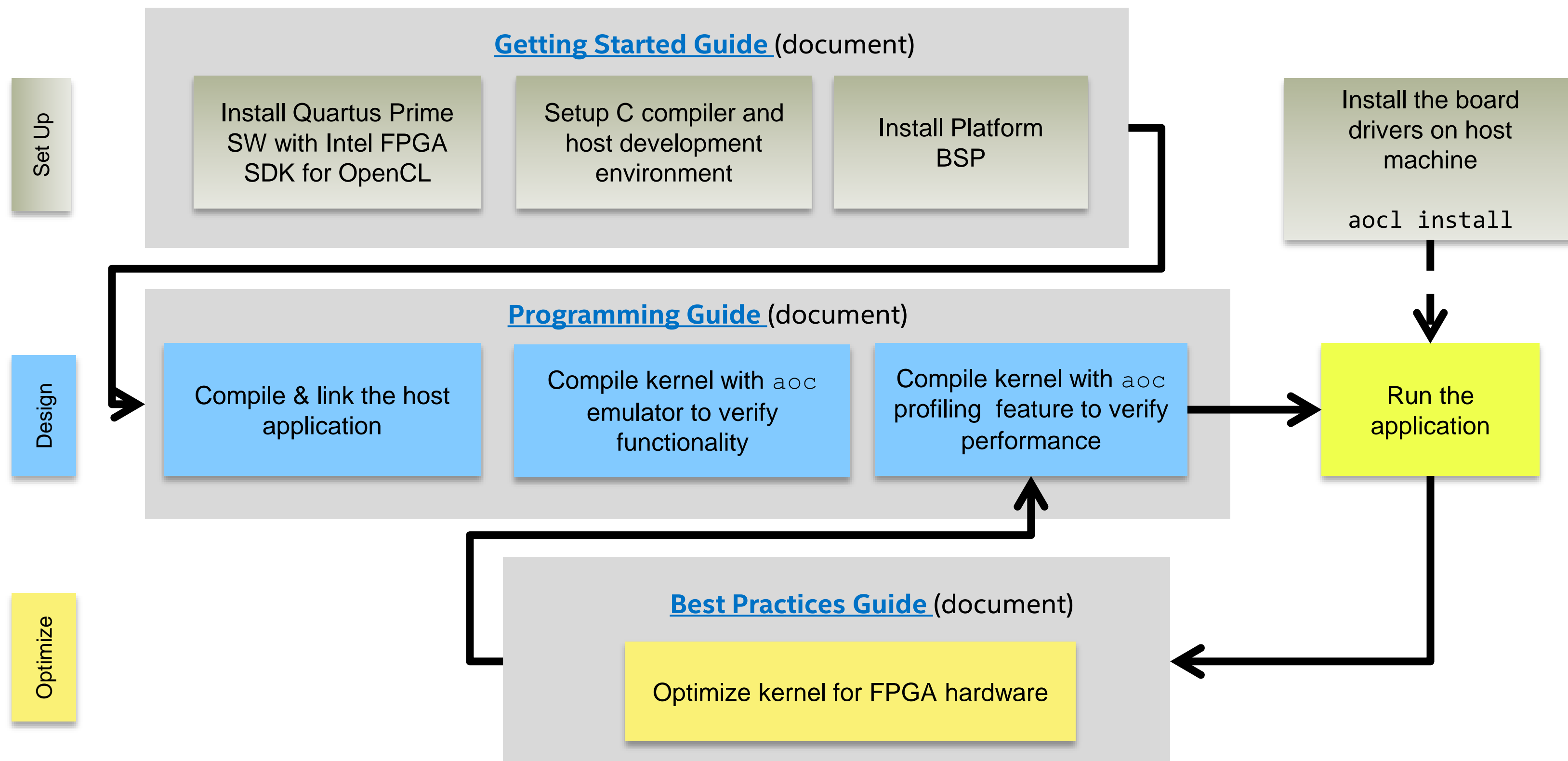
Option	Description
`--help or -h`	Help for the tool
`-v`	Reports the progress of the compilation
`--report`	Print area estimates to screen
`-c`	Creates .aoco object file and sets up a Quartus® Prime hardware design project
`-g`	Add debug data to reports.
`-o <file>`	Use to specify a non-default name for the output file
`-I/-L <directory>`	Adds <directory> to header search path
`-l <library.aoclib>`	Specify OpenCL™ library file
`-D <name>`	Defines a macro called <name>
`--board <board name>`	Compile for the specified board
`--list-boards`	Prints a list of available boards
`-march=emulator`	Create kernels that can be executed and debugged on the host PC without the board
`--profile`	Enable profile support when generating aocx file

Kernel Language Support

- Supports OpenCL™ 1.0
  - Passed Khronos* Conformance Testing Process
- Full description of the supported kernel language features is located in the *Intel® FPGA SDK for OpenCL Programming Guide*
- Built-in scalar data types supported as follows:
  - All integer types are supported, including 64-bit types
  - Single precision floating point is supported
  - Double precision floating point is supported
  - *The half type is supported only in the add, subtract, and multiply operations.*
- Built-in vector data types including three-element types supported

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

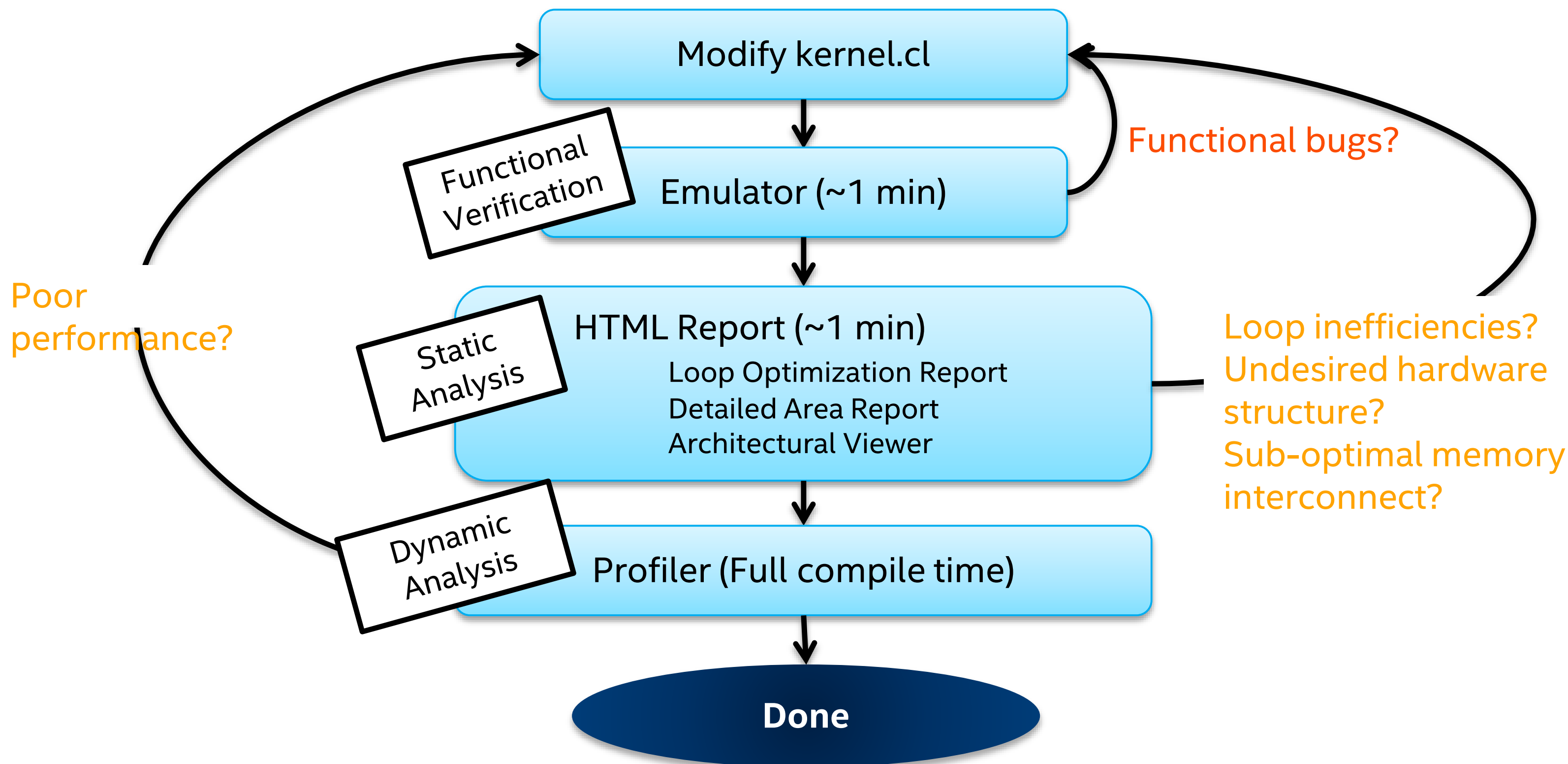
Intel® FPGA SDK for OpenCL™ Design Flow



Execution Considerations

- Ensure the .aocx file is available for host executable to read and send to clCreateProgramWithBinary
  - Kernels in the aocx file must match those called by the host
  - Allow the OpenCL™ implementation to program the FPGA
- The timing of reading/writing tasks to/from the device memory is done on an as needed basis and not necessarily when those actions are enqueued onto the command queue
  - Consistent with the OpenCL specifications
- Many factors can affect FPGA accelerator performance
  - Memory bandwidth, caching scheme, workgroup size, etc
  - Please consult the best practices guide for more details

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]);
  …
}
```

```
./kernel_tb…
…
Running …
```

- 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!
```

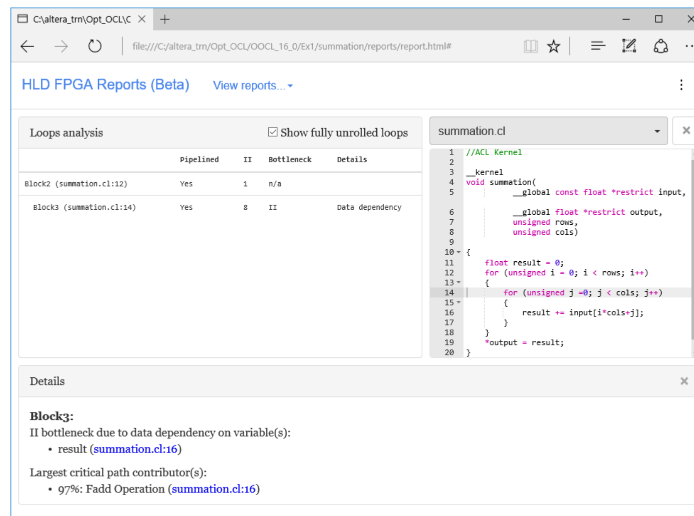
HTML Report

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

HLD FPGA Reports (Beta) View reports...

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
Private Variable: - 'result' (summation.cl:11)	104	194	0	0	• Implemente...
Private Variable: - 'i' (summation.cl:12)	8	69	0	0	• Implemente...
Private Variable: - 'j' (summation.cl:14)	16	13	1	0	• Implemente...
summation.cl:12	0	79	0	0	

summation.cl

```
//ACL Kernel

__kernel
void summation(
        __global const float *restrict input,
        __global float *restrict output,
        unsigned rows,
        unsigned cols)

{
    float result = 0;
    for (unsigned i = 0; i < rows; i++)
    {
        for (unsigned j =0; j < cols; j++)
        {
            result += input[i*cols+j];
        }
    }
    *output = result;
}
```

Details

Private Variable:
- 'i' (summation.cl:12):
- Implemented using registers of the following size:
  - 1 register of width 32 and depth 1

Architectural Viewer

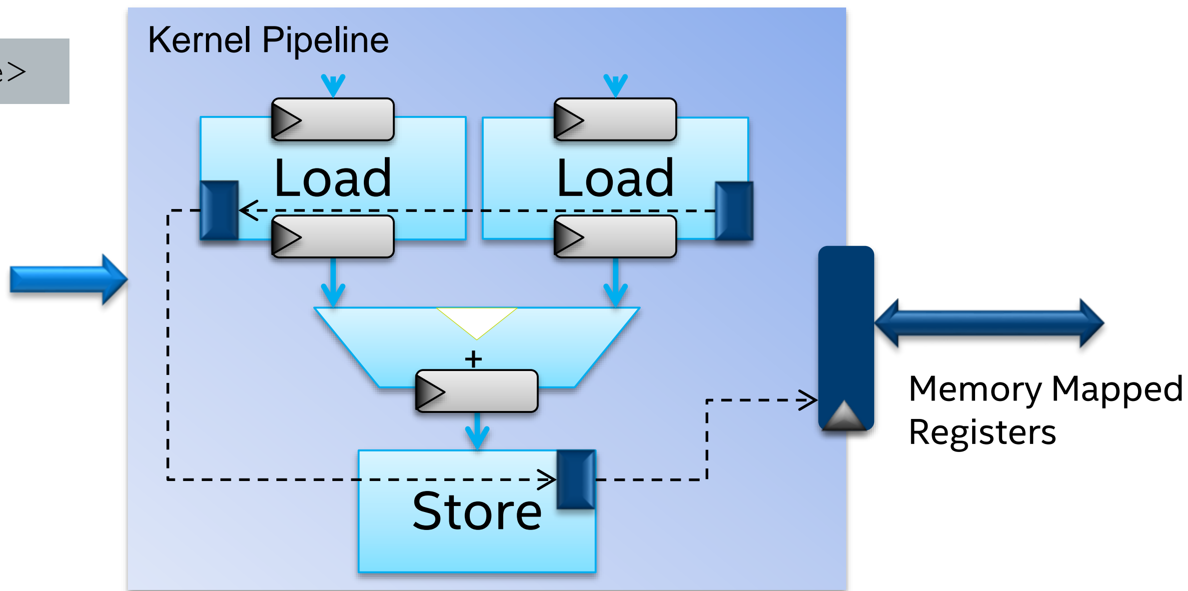
- Displays kernel pipeline implementation and memory access implementation

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];
  …
}
```



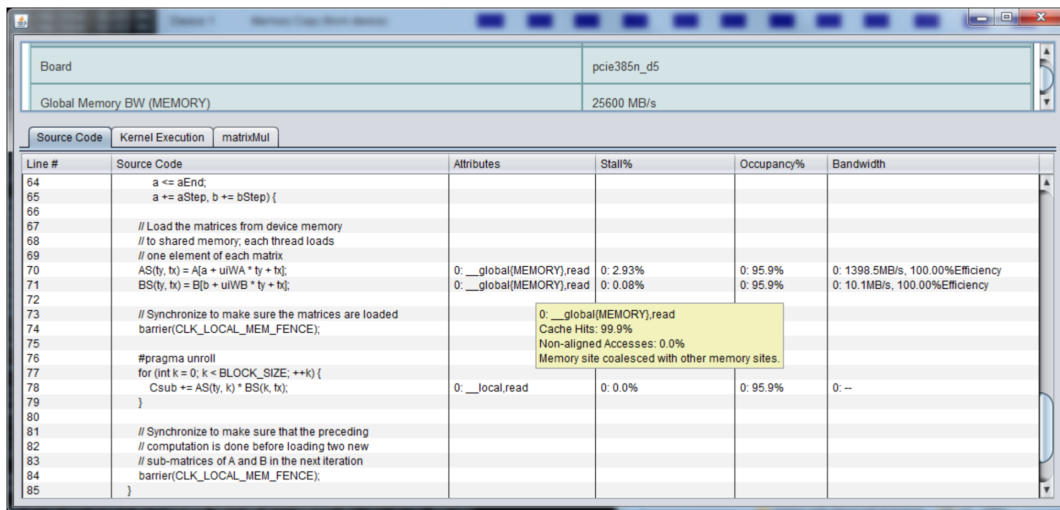
Collecting and Viewing Profile Information

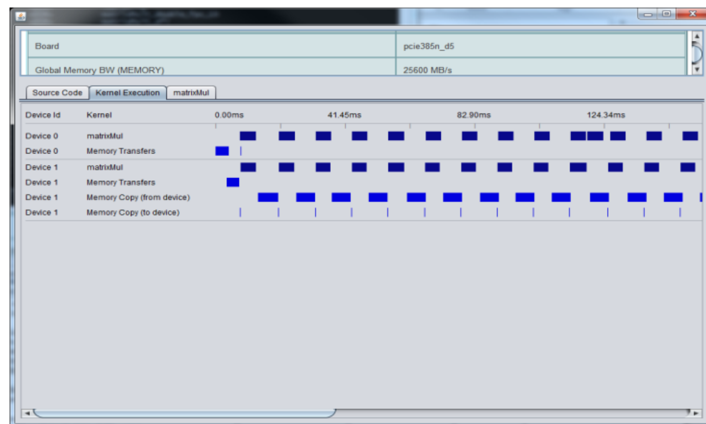
- Compile kernel with `aoc --profile` option
- Run host application with generated aocx file
  - Performance counters will collect profile information
  - Host saves a `profile.mon` monitor description file to working directory
- View statistical data using the profiler GUI

```
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





OpenCL™ Support is Everywhere

- CPUs, GPUs, FPGAs, mobile devices
- List of conformant products
  - http://www.khronos.org/conformance/adopters/conformant-products#topencl
- Short list of OpenCL adopters
  - Intel
  - ARM
  - Qualcomm
  - NVIDIA

Benefits of OpenCL™ on FPGAs

Productivity

- Shorter development time and faster time to market
- Shorter architecture and design exploration time
- Portability across multiple devices

Performance

- CPU offload of computationally intensive applications
- Compiler and runtime routines can distribute the workloads

Performance per watt

- FPGA fabric flexibility
- Generate custom hardware for your kernel

Getting Started with Intel® FPGA OpenCL™ SDK

Resources	Where to find them...
Intel PSG OpenCL Website	http://www.altera.com/opencl
Intel PSG OpenCL Dev Zone Videos, Examples, Trainings, Board Partners Optimization Techniques (SHA-1) Optimization Techniques (Image Processing)	http://www.altera.com/products/design-software/embedded-software-developers/opencl/developer-zone.html https://www.altera.com/opencl-optimization-sha1 https://www.altera.com/opencl-image-processing-optimization
Instructor-Led Training Parallel Computing with OpenCL - 1 day Optimization of OpenCL for FPGAs – 2 days Developing a Custom BSP – 1 day	https://www.altera.com/support/training/catalog.html http://www.altera.com/education/training/courses/IOPNCL110 http://www.altera.com/education/training/courses/IOPNCL210 http://www.altera.com/education/training/courses/IOPNCLBSP
Free On-Line Training Intro to Parallel Computing with OpenCL Writing OpenCL Programs for FPGAs Running OpenCL on FPGAs Single-Threaded vs Multi-Threaded Kernels Building Custom Platforms for FPGAs	 http://www.altera.com/education/training/courses/OOPNCL100 http://www.altera.com/education/training/courses/OOPNCL300 http://www.altera.com/education/training/courses/OOPNCL300 http://www.altera.com/education/training/courses/OOPNCLKERN http://www.altera.com/education/training/courses/OOPNCLCSTBOARD

OpenCL™ References

- Intel® FPGA OpenCL collateral
  https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html
- 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 https://www.khronos.org/opencl/
- OpenCL Reference Card http://www.khronos.org/files/opencl-quick-reference-card.pdf

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos
*Other names and brands may be claimed as the property of others

Agenda

Approaches to Parallel Programming

Data Sharing and Synchronization

Overview of OpenCL™

OpenCL Platform and Host-side Software

Executing OpenCL Kernels

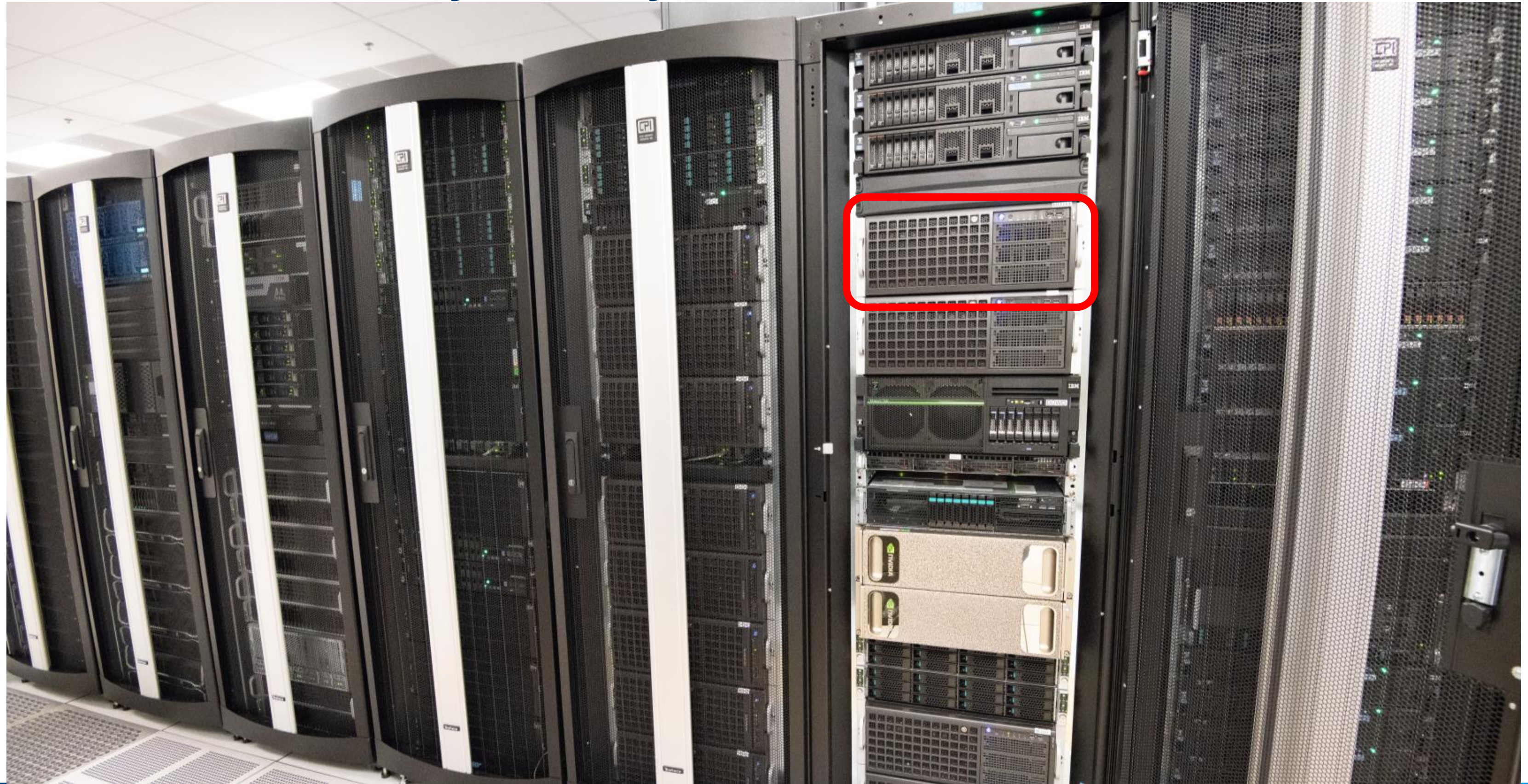
Compiling software into circuits

The Intel® FPGA SDK for OpenCL™

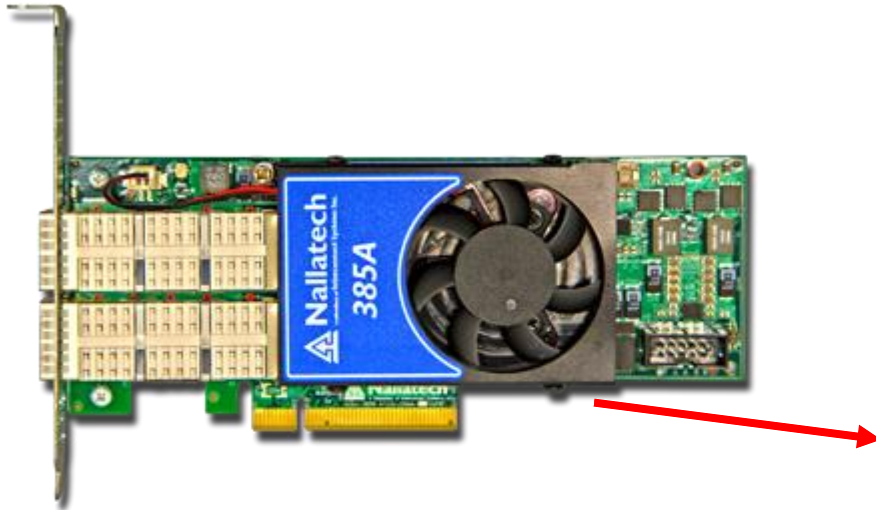
Hands-on Example

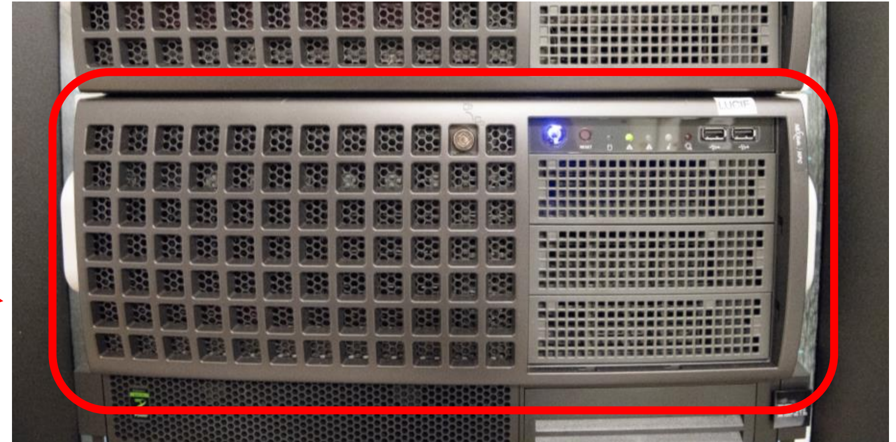
*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos

Joint Laboratory for System Evaluation (JLSE)



JLSE: Ruth
Nallatech® 385A FPGA Accelerator Card with Intel® Arria® 10 FPGA





Hands-on Example (running on JLSE)

Part A: set up the env and copy the lab files

1. Login to a shell on JLSE:
   `$ ssh login.jlse.anl.gov`

2. Please submit for an interacive queue so that we don't tie up resources on the jlse login node. E.g.:
   `$ qsub -q gomez` (or `it` or `skylake_8180` -- some queue you have access to) `-I -t 120 -n 1`

3. If you do not wish to work out of your home directory, cd into another directory. E.g.
   ```
   $ mkdir atpesc_lab
   $ cd atpesc_lab
   ```

4. Obtain the lab files from my home directory and cd into that directory:
   ```
   $ cp -r /home/jmoawad/fpga_opencl_trn/ .
   $ cd fpga_opencl_trn
   ```

5. Source the environment setup script:
   `$ source aoc-env.sh`
   Note that this is a slightly modified version of `/soft/fpga/altera/pro/18.0.0.219/aoc-env.sh` which sets the environment to use the OpenCL emulator rather than an FPGA board

aoc-env.sh

```
#!/bin/bash

TOPDIR=/soft/fpga/altera


export QUARTUS_ROOTDIR="$TOPDIR/pro/18.0.0.219"
#export AOCL_BOARD_PACKAGE_ROOT=${QUARTUS_ROOTDIR}/hld/board/nalla_pcie
export AOCL_BOARD_PACKAGE_ROOT=${QUARTUS_ROOTDIR}/hld/board/a10_ref
export CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=a10gx

#
# COMMON
#

export INTELFPGAOCLSDKROOT=${QUARTUS_ROOTDIR}"/hld"

export PATH="$QUARTUS_ROOTDIR"/quartus/linux64/jre64/bin:$PATH
export PATH=$PATH:"$QUARTUS_ROOTDIR"/bin:"$INTELFPGAOCLSDKROOT"/linux64/bin
export PATH=$PATH:"$INTELFPGAOCLSDKROOT"/bin
export PATH=$PATH:"$QUARTUS_ROOTDIR"/quartus/bin
# for HLS
export QUARTUS_ROOTDIR_OVERRIDE=$QUARTUS_ROOTDIR/quartus
export PATH=$PATH:"$QUARTUS_ROOTDIR"/quartus/sopc_builder/bin/ # for qsys-script
export PATH=$PATH:"$QUARTUS_ROOTDIR"/modelsim_ase/linuxaloem/ # for vsim

export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:"$INTELFPGAOCLSDKROOT"/linux64/lib
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:"$INTELFPGAOCLSDKROOT"/host/linux64/lib
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${AOCL_BOARD_PACKAGE_ROOT}/linux64/lib
```

Hands-on Example (running on JLSE)

Part B: compile a single work-item kernel

1. Change directories into the *OpenCL_single* directory. Here we have a simple OpenCL example showing a single work-item kernel:
   ```
   $ cd OpenCL_single
   ```

2. Change directories into the *device* directory where our OpenCL kernel code resides:
   ```
   $ cd device
   ```

3. Compile the kernel SimpleKernel.cl by executing the following command:
   ```
   $ aoc -v -march=emulator SimpleKernel.cl
   ```

4. Double check the kernel compiled successfully. You should now have a *.aocx file which is the kernel executable FPGA image:
   ```
   $ ls
   SimpleKernel/  SimpleKernel.aoco  SimpleKernel.aocr  SimpleKernel.aocx
   SimpleKernel.cl
   ```

SimpleKernel.cl

```
//OpenCL AOC Kernel
__kernel void my_kernel (__global float * restrict a,
                         __global float * restrict b,
                         __global float * restrict x,
                         uint array_size) {
int i;
for (i = 0; i < array_size; i++)
       x[i] = a[i] * b[i];
}
```

Hands-on Example (running on JLSE)

Part C: compile and execute the host application

1. Change directory into the folder containing our main() c++ source project for the host:
```
$ cd ../host/
```
2. List the directory files and you should see 3 source files and a Makefile:
```
$ ls
main.cpp  Makefile  utility.cpp  utility.h
```
3. Compile the software using a make command:
```
$ make
```
4. Run the executable that was just created:
```
$ ./SimpleOpenCLApp
```

Hands-on Example (running on JLSE)

Part D: compile a NDRange kernel

1. Change directories into the *OpenCL_ndrange* directory. Here we have a simple OpenCL example showing a single work-item kernel:
   ```
   $ cd ../../OpenCL_ndrange
   ```
2. Change directories into the *device* directory where our OpenCL kernel code resides:
   ```
   $ cd device
   ```
3. Compile the kernel SimpleKernel.cl by executing the following command:
   ```
   $ aoc -v -march=emulator SimpleKernel.cl
   ```
4. Double check the kernel compiled successfully. You should now have a *.aocx file which is the kernel executable FPGA image:
   ```
   $ ls
   SimpleKernel/  SimpleKernel.aoco  SimpleKernel.aocr  SimpleKernel.aocx
   SimpleKernel.cl
   ```

SimpleKernel.cl

```
//OpenCL AOC NDRange Kernel
__kernel void my_kernel (__global float * restrict a,
                         __global float * restrict b,
                         __global float * restrict x) {

       size_t i = get_global_id(0);
       x[i] = a[i] * b[i];
}
```

Hands-on Example (running on JLSE)

Part E: compile and execute the host application

1. Change directory into the folder containing our main() c++ source project for the host:
```
$ cd ../host/
```

2. List the directory files and you should see 3 source files and a Makefile:
```
$ ls
main.cpp  Makefile  utility.cpp  utility.h
```

3. Compile the software using a make command:
```
$ make
```

4. Run the executable that was just created:
```
$ ./SimpleOpenCLApp
```

Follow on

Today we used targeted our OpenCL kernel to the emulator. To target an actual FPGA card, there is a Nallatech 385A with Intel Arria 10 on “Ruth”. The queue name is fpga_385a

```
$ qsub -q fpga_385a -I -t 120 -n 1
```

Launch aoc without the emulator flag. Adding the -report flag will generate the html report.

```
$ aoc -v –report SimpleKernel.cl
```

Follow on by using design examples

Design examples available:

- 1D FFT https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/fft-1d.html
- 2D FFT https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/fft-2d.html
- Finite Difference Computation https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/fdtd-3d.html
- Many others at https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/developer-zone.html#design-examples

intel®
experience
what's inside™

OpenCL results with HLD Stratix 10 compiler

Designs

- Compiled directly from OpenCL source code available internally
- Functionally equivalent source code used to compile for A10 and S10

Fmax

- Max achieved among several seeds when compiled through OpenCL flow on a pre-release candidate build (RC2)
- Targeting the reference S10 BSP for the -1 GX 280 production silicon

Scaling

- Increase in design capability against a compile of the source code for A10. For matrix multiplication, backprojection and gzip, this is an increase in throughput, between a projection of the S10 performance (accounting for the fmax measurement and assuming a -1 BSP platform) and a measurement of the A10 performance (on the existing reference -1 0.95V platform). For FFT, it is an increase in the number of design points possible due to the device size

Devkit demo

- Design performance as measured in hardware on the Stratix10 GX H-tile development kit, which uses an early silicon -2 speed grade GX280 part

Continued on next slide

S10 vs A10 fmax (-2) graph

- HLD QoR set (see above)
    - Same designs have been used for both data points
- Both data points targeted the same device (-2 production part) and have been compiled with the same 18.0 pre-release candidate
    - A10 targeted the reference BSP, retargeted to a -2 production part 0.9V
    - 18.0 used a pre-release candidate and targeted a -2 production part 0.9V