Access Intel® FPGAs for Acceleration

Karl Qi

CERN 2018
Agenda

- High-level synthesis with the Intel® HLS Compiler
- Intel® FPGA SDK for OpenCL™
- Acceleration Stack for Intel® Xeon CPUs and FPGAs
- Deep Learning Inference on FPGAs
FPGA Overview

- Field Programmable Gate Array (FPGA)
  - Millions of logic elements
  - Thousands of embedded memory blocks
  - Thousands of DSP blocks
  - Programmable routing
  - High speed transceivers
  - Various built-in hardened IP

- Used to create **Custom Hardware!**
Traditional FPGA Design Process

Potentially Time-Consuming Effort

Behavioral Simulation

Synthesis

Place & Route / Timing Analysis / Timing Closure

HDL

Design Software

Intel® Quartus® Prime

Stratix® FPGA Device
Why HLS?

Designing IP at a higher level of abstraction = increase productivity

- Debugging software is much faster than hardware
- Easier to specify functions in software
- Productivity tool for RTL designers
HLS Use Model

C/C++ Code

Standard gcc/g++ Compiler

100% Makefile compatible

HLS Compiler

HDL IP

FPGA

src.c

lib.h

i++ <options>

a.exe

Intel® Quartus® Ecosystem

Programmable Solutions Group
HLS Procedure

1. **Create Component and Testbench in C/C++**
   - Use `-march=x86-64`
   - Both compilers compatible with GDB

2. **Functional Verification with g++ or i++**
   - Use `-march=x86-64`
   - Both compilers compatible with GDB

3. **Compile with i++ `-march=<FPGA fam>` for HLS**
   - Generates IP
   - Examine compiler generated reports
   - Verify design in simulation

4. **Run Quartus® Prime Compilation on Generated IP**
   - Generate QoR metrics

5. **Integrate IP with rest of your FPGA system**

---

Flowchart:

- **C/C++ Source** → **Intel® HLS Compiler** → **HDL IP**
- **Emulation** → **Cosimulation** → **Functional Iterations**
- **Architectural Iterations**
Emulation Mode

- Just like any executing any other software
- Debug with
  - printf/cout
  - gdb
  - Valgrind

Develop with C/C++:

```
src.c
lib.h
```

```
i++ -march=x86-64 src.c
```

GDB-Compatible Executable
Cosimulation: Synthesize Component Function into RTL

```c
#include "HLS/hls.h"
#include "assert.h"
#include "HLS/stdio.h"
#include "stdlib.h"

component int accelerate(int a, int b) {
    return a+b;
}

int main() {
    srand(0);
    for (int i=0; i<10; ++i) {
        int x=rand() % 10;
        int y=rand() % 10;
        int z=accelerate(x, y);
        printf("%d + %d = %d\n", x, y, z);
        assert(z == x + y);
    }
    return 0;
}
```

accelerate() becomes an FPGA component

- Use --component i++ argument or component attribute in source

main() becomes testbench for component accelerate()
The Cosimulation Flow

**Run Compiler for HLS:**

```
i++ -march=<fpga fam> --component func src.c
```

Executable which will run the testbench and calls to `func` in simulation of synthesized IP

- `a.prj/ components/func/`
- `a.prj/ reports/`
- `a.prj/ verification/`
- `a.prj/ quartus/`

All the files necessary to include IP in a Quartus project. i.e. .qsys, .ip, .v etc

Component hardware implementation reports

Simulation testbench

Quartus project to compile all IP

*a* is the default output name, `-o` option can be used to specify a non-default output name.
Cosimulation Verifying HLS IP

The Intel® HLS compiler automatically compiles and links C++ testbench with an instance of the component running in an RTL simulator

- To verify RTL behavior of IP, just run the executable generated by the HLS compiler targeting the FPGA architecture
  - Any calls to the component function becomes calls the simulator through DPI

```
src.c
lib.h

i++ -march=<fpga family> src.c

Data

a.exe|out

a.prj/verification/
```
C/C++ Functions to Dataflow Circuits

Each component function is converted into custom dataflow hardware

- Gain the benefits of Intel® FPGAs without the length design process
- Implement C/C++ operators as circuits
  - HDL code located in `<HLS 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 function
Compilation Example

Software compiled into dataflow circuit with flow control

- Include branch and merge units

```c
void my_component(int *a, int *b, int *c, int N)
{
    int i;
    for (i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```
The Default Interfaces

```
component int add(int a, int b) {
    return a + b;
}
```

<table>
<thead>
<tr>
<th>C++ Construct</th>
<th>HDL Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar arguments</td>
<td>Conduits associated with the default start/busy interface</td>
</tr>
<tr>
<td>Pointer arguments</td>
<td>Avalon memory master interface</td>
</tr>
<tr>
<td>Global scalars and arrays</td>
<td>Avalon memory master interface</td>
</tr>
</tbody>
</table>

Note: more on interfaces later.
Other Custom Interfaces

- Customizable Avalon Streaming Interfaces
  - Explicit ready/valid signals for each data argument

- Explicit Memory-Mapped Master interfaces
  - Create a number of master interfaces with customizable features

- Slave Registers
  - Slave port for scalar values

- Slave Memory

- Slave Control
  - Call/Return interface done through register
MM HLS Component with Streaming Interfaces

- HLS Component
  - Stream In
  - Stream Out
  - Valid data ready

- Platform Designer Interconnect
  - Processor
    - Data Master

- Upstream Component
  - Valid data ready
  - Slave CSR

- Downstream Component
  - Valid data ready
  - Slave CSR

- HLS Component
Viewing Waveforms in Modelsim

Locate Component

Add Signals to Waveform
Intel® Quartus® Software Integration

- `a.prj/components` directory contains all the files to integrate
  - One subdirectory for each component
    - Portable, can be moved to a different location if desire

- 2 use scenarios
  1. Instantiate in HDL
  2. Adding IP to a Platform Designer system integration tool system
Main HTML Optimization Report

Fast generation of optimization report
Loops

Serial loop execution hinders function dataflow circuit performance

- Use Loop Analysis report to see if and how each loop is optimized
  - Helps identify component pipeline bottlenecks

![Loop Analysis Flowchart]

- Unrolled?
  - Yes: Automatically unrolled? Fully unrolled? Partially unrolled? #pragma unroll implemented?
  - No: Pipelined?
    - Yes: What’s the Initiation Interval (launch frequency of new iteration)? Are there dependency preventing optimal II?
    - No: Reason for serial execution?
Loop Unrolling

Loop unrolling: Replicate hardware to execute multiple loop iterations at once

- Simple loops unrolled by the compiler automatically
- User may use \#pragma unroll to control loop unrolling
- Dependencies resolved through scheduling of operations
Loop-Pipelining and Dependencies

- Execute next iteration as soon as possible
- Dependencies can be resolved by the compiler
  - Values transferred between loop iterations with FPGA resources

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

- Automatically Generated
- Reports status of loop pipelining
- Displays dependency information

- Part of HTML Report
  - `<prj folder>/reports/report.html`
Loop Pipelining Optimization Report

Reports shows pipeline status of each loop

- Minimizing II is the key to loop pipelining optimization

- Report shows
  - If loops are pipelined
    - Reason given if loop not pipelined
  - Initiation interval of pipelined loops
    - If II>1, shows operations that contributes to loop-carried dependency
      - Data computation or memory dependencies
      - Dependencies increases II
Arbitrary Precision Datatypes

- Algorithmic C (AC) datatypes
  - From Mentor Graphics under the Apache License
  - User Guide shipped with the HLS tool
    - `<path_to_HLS_installation>/include/ref/ac_datatypes_ref.pdf`

- Templated classes that allows instantiation of **arbitrary sized integers** and **arbitrary precision fixed-point** datatypes

- `ac_int` and `ac_fixed` are supported by the Intel® HLS Compiler

- Two implementations shipped with the Intel® HLS Compiler
  - `ref/ac_int.h, ac_fixed.h`: Mentor Graphics reference implementation
  - `HLS/ac_int.h, ac_fixed.h`: Intel-optimized implementation for HLS
Local Component Memories

- Local component memories implemented with on-chip RAM resources
  - Much better performance than off-chip system memories

- Local memory system is customized to your application at compile time
  - Dependent on data type and usage
  - Banking configuration (number of banks, width), and interconnect customized to minimize contention
  - Big advantage over fixed-architecture accelerators

- Note: local memory cannot be dynamically allocated inside the component
Agenda

- High-level synthesis with the Intel® HLS Compiler
- Intel® FPGA SDK for OpenCL™
- Acceleration Stack for Intel® Xeon CPUs and FPGAs
- Deep Learning Inference on FPGAs
Intel® FPGA SDK for OpenCL™ Usage

Intel® FPGA SDK for OpenCL™ Libraries

OpenCL Host Program

Standard C Compiler

Executable File

Offline Compiler (OpenCL Kernel Compiler)

Binary Programming File

Intel® FPGA SDK for OpenCL™ Libraries

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos
Compiling OpenCL™ Kernel to Intel® FPGA

- Using similar concepts and optimization techniques as HLS

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

*aoc -board=a10_ref*
Benefits of OpenCL™ on FPGAs

- For software developers
- Faster software-centric development flow
  - C-based design leads to shorter architectural exploration and development time
- Obtain performance and power advantages of an FPGA
- Portability between different HW accelerators (CPU, GPGPU, FPGA, etc)

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos*
FPGA Architecture for OpenCL™ Implementation

- Processor
- Host Interface
- External Memory Controller & PHY
- External Memory Controller & PHY
- Global Memory Interconnect
- Local Memory Interconnect
- On-Chip Memory

Precompiled periphery (BSP)

Custom Built Kernel System

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos*
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 folder>\reports\report.html`
    - Interactive HTML report
    - Static report showing optimization, detailed area, and architectural information

  - `<kernel file>.log` compilation log

- Intel® Quartus® Prime software generated source and report files
void main()
{
    ...
    // 1. Create then build program
    c::Program myprogram = (... mybinaries_of_aocx...);
    err = myprogram.build(mydevlist);

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

    // 3. Transfer buffers on/to device
    err = myqueue.enqueueWriteBuffer(a_device, CL_FALSE, 0, size, a_host);
    ...

    // 4. Set up the kernel argument list
    err = mykernel.setArg(0, buffer);

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

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

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

```c
main() {
    read_data( ... );
    manipulate( ... );
    clEnqueueWriteBuffer( ... );
    clEnqueueNDRange(...,sum,...);
    clEnqueueReadBuffer( ... );
    display_result( ... );
}
```
Kernel Development Flow and Tools

- Modify kernel.cl
- Emulator (~1 min)
  - Functional bugs?
- HTML Report (~1 min)
  - Loop Optimization Report
  - Detailed Area Report
  - Architectural Viewer
  - Loop inefficiencies?
  - Undesired hardware structure?
  - Sub-optimal memory interconnect?
- Profiler (Full compile time)
- Dynamic Analysis
- Static Analysis
- Poor performance?

Done
Intel® FPGA-Specific Features

- Single Work-Item Execution
- Channels
- Controlling Hardware Generation with Attributes
  - Autorun Kernels, Vectorization Factor, Compute Unit replication, etc...
- Libraries (Calling custom RTL)
- SoC Platforms
- Shared Virtual Memory
- Custom Boards
Agenda

- High-level synthesis with the Intel® HLS Compiler
- Intel® FPGA SDK for OpenCL™
- **Acceleration Stack for Intel® Xeon CPUs and FPGAs**
- Deep Learning Inference on FPGAs
Acceleration Stack for Intel® Xeon® CPU with FPGAs

- Dynamically Allocate Intel® FPGAs for Workload Optimization
- Simplified Application Development
- Leverage Common Frameworks
- Fast-Track Your Performance
- Workload Optimization with Less Effort
- Common Developer Interface for Intel FPGA Data Center Products

Rack-Level Solutions

User Applications

Industry Standard SW Frameworks

Acceleration Libraries

Intel Developer Tools
- (Intel Parallel Studio XE, Intel FPGA SDK for OpenCL™, Intel Quartus® Prime)

Acceleration Environment
- (Intel Acceleration Engine with OPAE Technology, FPGA Interface Manager (FIM))

Intel® Hardware

Intel® delivers a system-optimized solution stack for your data center workloads

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos
Some names pending final approval and may change in the future.
Logos and names provided for illustrative purposes only. Current availability may be different.
Intel® Xeon® with FPGA Virtualization Framework

Developed by User

Application

Libraries

Drivers

Intel® Xeon®
Software

User, Intel, and 3rd Party

Open Programmable Acceleration Engine (OPAE)
Provided by Intel

FPGA Interface Manager
Provided by Intel

Accelerator Function Unit (AFU)

Signal Bridge and Management

FPGA Hardware

User, Intel, or 3rd-Party IP
Plugs into Standard Slot

Simplifies the use of FPGAs in virtualized cloud environments
Intel® Programmable Acceleration Card with Intel Arria® 10 GX FPGA

Intel’s 1st versatile FPGA PCIe acceleration card that offers inline & look-aside acceleration for workloads requiring up to 45W

1st acceleration card to offer the Acceleration Stack for Intel Xeon CPU with FPGAs enabling broader FPGA adoption in data center

Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA
Open Programmable Acceleration Engine (OPAE)

Consistent API across product generations and platforms
• Abstraction for hardware specific FPGA resource details

Designed for minimal software overhead and latency
• Lightweight user-space library (libfpga)

Open ecosystem for industry and developer community
• License: FPGA API (BSD), FPGA driver (GPLv2)

FPGA driver being upstreamed into Linux kernel

Supports both virtual machines and bare metal platforms

Faster development and debugging of Accelerator Functions with the included AFU Simulation Environment (ASE)

Includes guides, command-line utilities and sample code

Start developing for Intel FPGAs with OPAE today: http://01.org/OPAE
OPAE FPGA API – Enumerate, Manage & Access

1:1

property

fpgaEnumerate()

fpgaOpen()

token

property

token

fpgaReconfigureSlot()
fpgaReset()

fpgaPrepareBuffer(); fpgaReleaseBuffer(); fpgaGetIOVA();

fpgaMapMMIO() fpgaUnmapMMIO() fpgaWriteMMIO32() fpgaReadMMIO32() fpgaWriteMMIO64() fpgaReadMMIO64()

fpgaGetNumUmsg() fpgaSetUmsgAttributes() fpgaGetUmsgPtr()

handle

fpgaClose()
Two Development Approaches

**HDL Programming**
- **ASE from Intel®**
- **OPAE from Intel**
- **C** → **SW Compiler** → **exe** → **AFU Simulation Environment (ASE)**
- **C** → **HDL** → **Syn. PAR** → **AFU Bitstream** → **FPGA**

**OpenCL* Programming**
- **Intel® FPGA SDK for OpenCL™**
- **OpenCL Host** → **OpenCL Kernels** → **SW Compiler** → **exe** → **AFU Bitstream** → **FPGA**

**Intel®**
- **OpenCL Emulator**
- **Application** → **OPAE Software** → **FIM** → **AFU** → **FPGA**
- **Application** → **OpenCL BSP** → **FIM + OpenCL BSP** → **AFU** → **FPGA**
OpenCL™ Flow

- Usage no different from traditional OpenCL™ flow
  - C based development and optimization flow to create AFUs and Host Application
  - Standard OpenCL FPGA application using the Intel® FPGA SDK for OpenCL
    - FPGA OpenCL debug and profiling tools supported

- The Acceleration Stack abstracted away from user
  - OPAE part of the Host Run-Time
OpenCL™ Support Package for Intel® PAC

OpenCL Host

OpenCL Runtime

SP MMD

OPAE

CPU

Software Stack

FPGA Interface Manager (BBS)

FPGA Interface Unit

PCIe*

External Memory Interface

CCI-P

OpenCL Support Package IP

OpenCL Kernel

AFU (GBS)

External Memory Interface

DDR

DDR
**RTL AFU**

- Develop RTL AFU with standard FPGA development tools
- Interface with the acceleration stack through Core Cache Interconnect (CCI-P)
  - Provides a base platform memory interface
    - Simple request/response interface (Simple Read/Write)
    - Physical addresses
    - No order guarantees
  - These minimal requirements satisfy major classes of algorithms, e.g.:
    - Double buffered kernels that read from and write to different buffers
    - Streaming kernels that read from one memory-mapped FIFO and write to another
AFU Simulation Environment (ASE) enables seamless portability to real HW

- Allows fast verification of OPAE software together with AFU RTL without HW
  - SW Application loads ASE library and connects to RTL simulation
- For execution on HW, application loads Runtime library and RTL is compiled by Intel® Quartus into FPGA bitstream
Agenda

- High-level synthesis with the Intel® HLS Compiler
- Intel® FPGA SDK for OpenCL™
- Acceleration Stack for Intel® Xeon CPUs and FPGAs
- Deep Learning Inference on FPGAs
Design Flow with Machine Learning

**Data Collection**

- **Selection**

**Data Store**

- **Train Network**

**Inference Engine**

- **Improvement Strategies**
  - Collect more data
  - Improve network

**Choose Network**

- **Architecture**

**Parameters**

**Train Network**

- A high-performance computing (HPC) workload from large dataset
- Weeks to months process

**Inference Engine (FPGA Focus)**

- Implementation of the neural network performing real-time inferencing

Choose Network topology

- Use framework (e.g. Caffe, Tensor Flow)
Solving Machine Learning Challenges with FPGA

**EASE-OF-USE**
Software Abstraction, Platforms & Libraries

*Intel FPGA solutions enable software-defined programming of customized machine learning accelerator libraries.*

**REAL-TIME**
Deterministic Low Latency

*Intel FPGA hardware implements a deterministic low latency data path unlike any other competing compute device.*

**FLEXIBILITY**
Customizable Hardware for Next Gen DNN Architectures

*Intel FPGAs can be customized to enable advances in machine learning algorithms.*
### Why Intel® FPGAs for Machine Learning?

**Convolutional Neural Networks are Compute Intensive**

**Fine-grained & low latency**

between compute and memory

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highly parallel architecture</td>
<td>Facilitates efficient low-batch video stream processing and reduces latency</td>
</tr>
<tr>
<td>Configurable Distributed Floating Point DSP Blocks</td>
<td>FP32 9Tflops, FP16, FP11 Accelerates computation by tuning compute performance</td>
</tr>
<tr>
<td>Tightly coupled high-bandwidth memory</td>
<td>&gt;50TB/s on chip SRAM bandwidth, random access, reduces latency, minimizes external memory access</td>
</tr>
<tr>
<td>Programmable Data Path</td>
<td>Reduces unnecessary data movement, improving latency and efficiency</td>
</tr>
<tr>
<td>Configurability</td>
<td>Support for variable precision (trade-off throughput and accuracy). Future proof designs, and system connectivity</td>
</tr>
</tbody>
</table>
FPGAs Provide Deterministic System Latency

FPGAs can leveraging parallelism across the entire chip reducing the compute time to a fraction

\[ \text{System Latency} = \text{I/O Latency} + \text{Compute Latency} \]

**Compute Latency** 0.3ms

**I/O Latency** 0.25ms

### GoogleNet (Batch 1) Compute Latency

<table>
<thead>
<tr>
<th>FPGA Type</th>
<th>Compute Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arria 10 FP11</td>
<td>1.25 ms</td>
</tr>
<tr>
<td>Stratix 10 FP11</td>
<td>0.27 ms</td>
</tr>
<tr>
<td>Stratix 10 Binary</td>
<td>0.02 ms</td>
</tr>
<tr>
<td>P4/P40 (RT)</td>
<td>5 ms</td>
</tr>
</tbody>
</table>
Intel® FPGA Deep Learning Acceleration Suite

- CNN acceleration engine for common topologies executed in a graph loop architecture
  - AlexNet, GoogleNet, LeNet, SqueezeNet, VGG16, ResNet, Yolo, SSD, LSTM...
- Software Deployment
  - No FPGA compile required
  - Run-time reconfigurable
- Customized Hardware Development
  - Custom architecture creation w/ parameters
  - Custom primitives using OpenCL™ flow
Intel® FPGA DLA Suite Usage

- Supports common software frameworks (Caffe, TensorFlow)
- Intel DL software stack provides graph optimizations
- Intel FPGA Deep Learning Acceleration Suite provides turn-key or customized CNN acceleration for common topologies

Intel Deep Learning Deployment Toolkit
Part of OpenVINO™ toolkit

Standard ML Frameworks

- Caffe
- TensorFlow

Intel® Xeon® Processor

Intel® FPGA

Heterogenous CPU/FPGA Deployment

Model Optimizer

Inference Engine

DLA SW API

- Pre-compiled Graph Architectures
  - GoogleNet Optimized Template
  - ResNet Optimized Template
  - SqueezeNet Optimized Template
  - VGG Optimized Template
  - Additional, Generic CNN Templates

Optimized Acceleration Engine

Pre-compiled Graph Architectures

- Feature Map Cache
- Conv PE Array
- Crossbar
- Memory Reader/Writer
- Config Engine

Hardware Customization Supported
Mapping a Topology to the Architecture in FPGA

Using the Intel® DL Deployment Toolkit component of the OpenVINO™ toolkit to enable deployment of trained model on all Intel® architectures

- CPU, GPU, FPGA, …
- Optimize for best execution
- Enable users to validate and tune
- Easy-to-use runtime API across all devices

Trained Model

Model Optimizer
- FP Quantize
- Model Compress
- Model Analysis

Optimize a deploy-ready model

Inference Engine
- DLA Runtime Engine
- MKL-DNN
- MKL-DNN/cLDNN

FPGA
CPU
GEN

Intermediate Representation
- .bin
- .xml

Runtime Programming
Using the Inference Engine API

IR → Parse (using CNNNetReader)

→ Create Engine Instance

→ Load Network

→ Infer

```cpp
auto netBuilder = new InferenceEngine::CNNNetReader();
netBuilder->ReadNetwork("Model.xml");
netBuilder->ReadWeights("Model.bin");

auto enginePtr = new InferenceEngine::InferenceEnginePluginPtr(getSuitablePlugin(eFPGA));
enginePtr->LoadNetwork(*netBuilder->network, &resp);

InferenceEngine::TBlob<float> output;
InferenceEngine::SizeVector inputDims;
netBuilder->getInputDimensions(inputDims);
InferenceEngine::TBlob<short> input(inputDims);
input.allocate();
enginePtr->Infer(input, output, &resp);
```
User Flow

**Turnkey Software Deployment Flow**
- Data Scientist
  - Design
  - OpenVINO™ toolkit
  - Run
  - Program

**FPGA Architecture Development Flow**
- IP Architect
  - Design
  - User Customization of DLA Suite Source Code
  - Compile
  - Intel® FPGA SDK for OpenCL™
  - Bitstream Library
Customization for Architecture Developers

Add a custom primitive into crossbar

- **Three primitive types supported:**
  - Unary (ReLU, Tanh)
  - Binary (Eltwise Add, Mult)
  - Window (Pool, LRN, Norm)
  - Unary w/ coefficients
    - Scale/Dropout (a couple of coefficients per layer: coefficients loaded via layer config)
    - BatchNorm (dozen or more coefficients per layer: coefficients loaded via DDR)
Machine Learning on Intel® FPGA Platform

Acceleration Stack Platform Solution

ML Framework (Caffe*, TensorFlow*)

Software Stack
- Application
- DL Deployment Toolkit
- DLA Runtime Engine
- OpenCL™ Runtime
- Acceleration Stack

Hardware Platform & IP
- DLA Workload
- BBS

Intel® Xeon CPU

PAC Family Boards

For more information on the Acceleration Stack for Intel® Xeon® CPU with FPGAs on the Intel® Programmable Acceleration Card, visit the Intel® FPGA Acceleration Hub
DLA Architecture: Built for Performance

- Maximize Parallelism on the FPGA
  - Filter Parallelism (Processing Elements)
  - Input-Depth Parallelism
  - Winogrand Transformation
  - Batching
  - Feature Stream Buffer
  - Filter Cache

- Choosing FPGA Bitstream
  - Data Type / Design Exploration
  - Primitive Support
CNN Computation in One Slide

\[
I_{\text{new}}[x][y] = \sum_{x'=-1}^{1} \sum_{y'=-1}^{1} I_{\text{old}}[x+x'][y+y'] \times F[x'][y']
\]

Input Feature Map (Set of 2D Images)
Filter (3D Space)
Output Feature Map

Repeat for Multiple Filters to Create Multiple “Layers” of Output Feature Map
Mapping Graphs in DLA

AlexNet Graph

Blocks are run-time reconfigurable and bypassable
Mapping Graphs in DLA

AlexNet Graph

Blocks are run-time reconfigurable and bypassable
Mapping Graphs in DLA

AlexNet Graph

Blocks are run-time reconfigurable and bypassable
Mapping Graphs in DLA

AlexNet Graph

Blocks are run-time reconfigurable and bypassable
Mapping Graphs in DLA

AlexNet Graph

Blocks are run-time reconfigurable and bypassable
Mapping Graphs in DLA

AlexNet Graph

Blocks are run-time reconfigurable and bypassable
Mapping Graphs in DLA

AlexNet Graph

Blocks are run-time reconfigurable and bypassable
Mapping Graphs in DLA

AlexNet Graph

Blocks are run-time reconfigurable and bypassable
Mapping Graphs in DLA

**AlexNet Graph**

Blocks are run-time reconfigurable and bypassable
Efficient Parallel Execution of Convolutions

- **Parallel Convolutions**
  - Different filters of the same convolution layer processed in parallel in different processing elements (PEs)

- **Vectored Operations**
  - Across the depth of feature map

- **PE Array geometry can be customized to hyperparameters of given topology**
Winograd Transformation

- Perform convolutions with fewer multiplication
  - Allows more convolutions to be done on FPGA
- Take 6 input features elements and 3 filter elements
  - Standard convolution requires 12 multiplies
  - Transformed convolution requires just 6 multiplies
Fully Connected Computation and Batching

- Fully Connected Layer computation does not allow for data reuse of weights
  - Different from convolutions
  - Very memory bandwidth intensive

- Solution: Batch up images
  - Weights reused across multiple images

\[ o = I_{vec} \ast W_{vec} \]

\[ O_{vec} = I_{mat} \ast W_{vec} \]
Feature Cache

Feature data cached on-chip

- Streamed to a daisy chain of parallel processing elements
- Double buffered
  - Overlap convolution with cache updates
  - Output of one subgraph becomes input of another
  - Eliminates unnecessary external memory accesses
Filter Cache

Filter weights cached in each processing element

- Double buffered in order to support prefetching
  - While one set is used to calculate output feature maps, another set is prefetched
### DLA Architecture Selection

- Find ideal FPGA image that meets your needs
- Create custom FPGA image based on need

<table>
<thead>
<tr>
<th>Arch Name</th>
<th>ALEXNET</th>
<th>GOOGLENET</th>
<th>SQUEEZENET</th>
<th>VGG</th>
<th>RESNET18</th>
<th>RESNET18_MANUAL</th>
<th>RESNET50</th>
<th>RESNET101</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-8-1_rc_fp32_8x8_arch02</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>0-8-1_rc_fp16_4x4_arch03</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>0-8-1_rc_fp16_8x32_arch09</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>0-8-1_rc_fp16_8x32_arch10</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>0-8-1_rc_fp16_8x32_arch11</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>0-8-1_rc_fp16_8x32_arch12</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>0-8-1_rc_fp11_16x32_arch17</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>0-8-1_rc_fp11_16x32_arch18</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>0-8-1_rc_fp11_16x32_arch19</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>0-8-1_rc_fp11_16x32_arch20</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>0-8-1_rc_fp10_16x32_arch23</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>0-8-1_rc_fp9_16x32_arch25</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>0-8-1_rc_fp8_16x32_arch26</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
Support for Different Topologies

Tradeoff between features and performance

Feature Map Cache
Convolution PE Array
Memor y Reader /Writer
Crossbar
ReLU
Norm
MaxPool
Config Engine

Convolution PE Array
Feature Map Cache
Memor y Reader /Writer
Crossbar
SoftMax
LRN
Concat
Flatten
Reshape
ReLU
Norm
Permute
MaxPool
Config Engine
## Supported Primitives and Topologies

### Primitives
- ✓ batch norm
- ✓ max pool
- ✓ average pool
- ✓ inner product
- ✓ reshape
- ✓ priorbox
- ✓ bias
- ✓ local conv
- ✓ power
- ✓ slice
- ✓ tanh
- ✓ concat
- ✓ relu, leaky relu
- ✓ scale
- ✓ permute
- ✓ detection output
- ✓ fully connected
- ✓ group conv
- ✓ sigmoid
- ✓ crop
- ✓ depthwise conv
- ✓ dilated conv
- ✓ deconv
- ✓ flatten
- ✓ lrn normalization
- ✓ softmax
- ✓ prelu
- ✓ conv
- ✓ eltwise
- ✓ depthwise conv
- ✓ elu
- ✓ proposal
- ✓ roi pooling
- ✓ deconv

### Topologies
- ✓ AlexNet
- ✓ GoogleNet v1
- ✓ ResNet18
- ✓ ResNet50
- ✓ ResNet101
- ✓ SqueezeNet
- ✓ VGG16
- ✓ Tiny Yolo
- ✓ LeNet
- ✓ SSD
- ✓ Supported
- ✓ Upon Request
- ✓ Future
### Design Exploration with Reduced Precision

**Tradeoff between performance and accuracy**

- Reduced precision allows more processing to be done in parallel
- Using smaller Floating Point format does not require retraining of network
- FP11 benefit over using INT8/9
  - No need to retrain, better performance, less accuracy loss

<table>
<thead>
<tr>
<th>Floating Point Format</th>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP16</td>
<td>5</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>FP11</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>FP10</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>FP9</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>FP8</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- FP16: Sign, 5-bit exponent, 10-bit mantissa
- FP11: Sign, 5-bit exponent, 5-bit mantissa
- FP10: Sign, 5-bit exponent, 4-bit mantissa
- FP9: Sign, 5-bit exponent, 3-bit mantissa
- FP8: Sign, 5-bit exponent, 2-bit mantissa
Summary

- Use HLS Compiler to generate excelleration IP for HW Developers
- Use OpenCL to accelerate for software developers
  - May use over Acceleration Stack
- Acceleration stack enables data center acceleration
  - Supports RTL and OpenCL flow, HLS in the future
- Use Deep Learning Acceleration Suite to easily deploy inference tasks on the FPGA
  - Supported for the Acceleration Stack
  - In the future will support custom platforms
Legal Disclaimers/Acknowledgements

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

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

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

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

© Intel Corporation