INTRO TO FPGA FLOWS OVERVIEW

Bill Jenkins
Intel Programmable Solutions Group
Agenda

9:00 am    Welcome
9:15 am    Introduction to FPGAs
9:45 am    FPGA Programming models: RTL
10:15 am   FPGA Programming models: HLS
11:00 am   Lab 1 HLS Flow
11:45 am   Lunch
12:30 pm   FPGA Programming models: OpenCL
1:00 pm    High Performance Data Flow Concepts
1:30 pm    Lab 2 OpenCL Flow
2:15 pm    Introduction to DSP Builder
3:00 pm    Introduction to Acceleration Stack
4:00 pm    Lab 3 Acceleration Stack
4:30 pm    Curriculum & University Program Coordination
INTRODUCTION
The Problem: Flood of Data

By 2020

The average internet user will generate
~1.5 GB of traffic per day

Smart hospitals will be generating over
3 TB per day

Self-driving cars will be generating over
4 TB per day... each

A connected plane will be generating over
40 TB per day

A connected factory will be generating over
1 PB per day

All numbers are approximated.

Radar ~10-100 KB per second

Sonar ~10-100 KB per second

GPS ~50 KB per second

LiDAR ~10-70 MB per second

Cameras ~20-40 MB per second

1 car 5 exaflops per hour


The Problem: Flood of Data

By 2020

The average internet user will generate
~1.5 GB of traffic per day

Smart hospitals will be generating over
3 TB per day

Self-driving cars will be generating over
4 TB per day... each

A connected plane will be generating over
40 TB per day

A connected factory will be generating over
1 PB per day

All numbers are approximated.

Radar ~10-100 KB per second

Sonar ~10-100 KB per second

GPS ~50 KB per second

LiDAR ~10-70 MB per second

Cameras ~20-40 MB per second

1 car 5 exaflops per hour

Typical HPC Workloads

- **ASTROPHYSICS**
- **GENOMICS / BIO-INFORMATICS**
- **ARTIFICIAL INTELLIGENCE**
- **MOLECULAR DYNAMICS***
- **BIG DATA ANALYTICS**
- **FINANCIAL**
- **WEATHER & CLIMATE**
- **CYBER SECURITY**

* Source: [https://comp-physics-lincoln.org/2013/01/17/molecular-dynamics-simulations-of-amphiphilic-macromolecules-at-interfaces/]
Fast Evolution of Technology

We now have the compute to solve these problems today in near real-time

<table>
<thead>
<tr>
<th>Bigger Data</th>
<th>Better Hardware</th>
<th>Smarter Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image: 50 MB / picture</td>
<td>Transistor density doubles every 18 months</td>
<td>Advances in neural networks leading to better accuracy in training models</td>
</tr>
<tr>
<td>Audio: 5 MB / song</td>
<td>Cost / GB in 1995: $1000.00</td>
<td></td>
</tr>
<tr>
<td>Video: 47 GB / movie</td>
<td>Cost / GB in 2015: $0.03</td>
<td></td>
</tr>
</tbody>
</table>
50+ Years of Moore’s Law
Computing has Changed...
If engineers keep building processors the way we do now, CPUs will get even faster but they’ll require so much power that they won’t be usable.

—Patrick Gelsinger, former Intel Chief Technology Officer, February 7, 2001

Source: http://www.cnn.com/2001/tech/ptech/02/07/hot.chips.idg/
Implications to High Performance Computing

- 50 GFLOPS/W
- ~100MW

- 1.5x from transistor
- 670x from parallelism

- 8x from transistor
- 128x from parallelism

- 32x from transistor
- 32x from parallelism
Challenges Scaling Systems to Higher Performance

- **CPU Intensive**
  - Result: Excessive power requirements

- **Memory Intensive**
  - Result: Slow Performance
  - Bottleneck

- **IO Intensive**
  - Result: Slow Performance (high latency)
  - Bottleneck

Need to think about Compute Offload as well as Ingress/Egress Processing
Diverse Application Demands

Accelerators can increase Performance at lower TCO for targeted workloads

Intel estimates; bubble size is relative CPU intensity
The Intel Vision

Heterogeneous Systems:

- Span from CPU to GPU to FPGA to dedicated devices with consistent programming models, languages, and tools
Heterogeneous Computing Systems

Modern systems contain more than one kind of processor

- Applications exhibit different behaviors:
  - Control intensive (Searching, parsing, etc...)
  - Data intensive (Image processing, data mining, etc...)
  - Compute intensive (Iterative methods, financial modeling, etc...)

- Gain performance by using specialized capabilities of different types of processors
Separation of Concerns

Two groups of developers:

- Domain experts concerned with getting a result
  - Host application developers leverage optimized libraries
- Tuning experts concerned with performance
  - Typical FPGA developers that create optimized libraries

Intel® Math Kernel Library a simple example of raising the level of abstraction to the math operations

- Domain experts focus on formulating their problems
- Tuning experts focus on vectorization and parallelization
INTRODUCTION TO FPGAS
FPGA Enabled Performance and Agility

Workload Optimization: ensure Xeon cores serve their highest value processing

Efficient Performance: improve performance/watt

Real-Time: high bandwidth connectivity and low-latency parallel processing

Developer Advantage: code re-use across Intel FPGA data center products

FPGAs enhance CPU-based processing by accelerating algorithms and minimizing bottlenecks
FPGAs Provide Flexibility to Control the Data path

Inline Data Flow Processing
- Machine learning
- Object detection and recognition
- Advanced driver assistance system (ADAS)
- Gesture recognition
- Face detection

Compute Acceleration/Offload
- Workload agnostic compute
- FPGAaaS
- Virtualization

Storage Acceleration
- Machine learning
- Cryptography
- Compression
- Indexing
FPGA Architecture

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

Used to create **Custom Hardware!**
FPGA Architecture: Basic Elements

Configured to perform any 1-bit operation:
AND, OR, NOT, ADD, SUB
FPGA Architecture: Flexible Interconnect

Basic Elements are surrounded with a flexible interconnect
Wider custom operations are implemented by configuring and interconnecting Basic Elements.
FPGA Architecture: Custom Operations Using Basic Elements

Wider custom operations are implemented by configuring and interconnecting Basic Elements

- 16-bit add
- 32-bit sqrt
- Your custom 64-bit bit-shuffle and encode
FPGA Architecture: Memory Blocks

Can be configured and grouped using the interconnect to create various cache architectures.
FPGA Architecture: Memory Blocks

Can be configured and grouped using the interconnect to create various cache architectures.
FPGA Architecture: Floating Point Multiplier/Adder Blocks

Dedicated floating point multiply and add blocks
DSP Blocks

Thousands DSP Blocks in Modern FPGAs

- Configurable to support multiple features
  - Variable precision fixed-point multipliers
  - Adders with accumulation register
  - Internal coefficient register bank
  - Rounding
  - Pre-adder to form tap-delay line for filters
  - Single precision floating point multiplication, addition, accumulation
FPGA Architecture: Configurable Routing

Blocks are connected into a custom data-path that matches your application.
FPGA Architecture: Configurable IO

The Custom data-path can be connected directly to custom or standard IO interfaces for inline data processing.
FPGA I/Os and Interfaces

FPGAs have flexible IO features to support many IO and interface standards

- **Hardened Memory Controllers**
  - Available interfaces to off-chip memory such as HBM, HMC, DDR SDRAM, QDR SRAM, etc.

- **High-Speed Transceivers**
- **PCIe* Hard IP**
- **Phase Lock Loops**

*Other names and brands may be claimed as the property of others
Intel® FPGA Product Portfolio

Wide range of FPGA products for a wide range of applications

- Products features differ across families
  - Logic density, embedded memory, DSP blocks, transceiver speeds, IP features, process technology, etc.
Mapping a Simple Program to an FPGA

Mem[100] += 42 * Mem[101]

CPU instructions
R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]
First let’s take a look at execution on a simple CPU

Fixed and general architecture:
- General “cover-all-cases” data-paths
- Fixed data-widths
- Fixed operations
Looking at a Single Instruction

Very inefficient use of hardware!
Sequential Architecture vs. Dataflow Architecture

Sequential CPU Architecture

FPGA Dataflow Architecture

Time

load

load

store

42
Custom Data-Path on the FPGA Matches Your Algorithm!

High-level code

\[ \text{Mem}[100] += 42 \times \text{Mem}[101] \]

Custom data-path

Build exactly what you need:
Operations
Data widths
Memory size & configuration

Efficiency:
Throughput / Latency / Power
Advantages of Custom Hardware with FPGAs

- Custom hardware!
- Efficient processing
- Fine-grained parallelism
- Low power
- Flexible silicon
- Ability to reconfigure
- Fast time-to-market
- Many available I/O standards
FPGA Development and Programming Tools

Software Developer

- Intel® SoC FPGA Embedded Design Suite (EDS)
- Intel® FPGA SDK for OpenCL

Algorithm Designer

- DSP Builder for Intel® FPGAs

IP Library Developer

- Intel® HLS Compiler

HDL Designer

- Verilog

HDL

Software Developer

Hardware Developer

Intel® Quartus Prime Design Software

Verilog, VHDL and the Intel® FPGA SDK for OpenCL are currently supported by the Acceleration Stack. High Level Synthesis can be used manually by following the app note.
Traditional FPGA Design Entry

Circuits described using Hardware Description Languages (HDL) such as VHDL or Verilog

A designer must describe the behavior of the algorithm to create a low-level digital circuit

- Logic, Registers, Memories, State Machines, etc.

Design times range from several months to even years!
Traditional FPGA Design Flow

Time-Consuming Effort

- HDL
- Behavioral Simulation
- Synthesis
- Place & Route / Timing Analysis / Timing Closure
- Intel Quartus Prime
- Stratix FPGA
- Board Simulation & Test

Intel Proprietary for LRZ
Intel® Quartus® Prime Design Software

Default Operating Environment

Project Navigator

Tasks window

Tool View window

IP Catalog

Messages window
Intel® Quartus® Prime Design Software Projects

Description

- Collection of related design files & libraries
- Must have a designated top-level entity
- Target a single device
- Store settings in the software settings file (.qsf)
- Compiled netlist information stored in qdb folder in project directory

Create new projects with New Project Wizard

- Can be created using Tcl scripts
Download complete example design templates for specific development kits

Design examples include design files, device programming files, and software code as required

Install .par files and select as template in New Project Wizard
Device Selection

Choose device family & family category (transceiver options, SoC options, etc.)

Filter device list

Tcl: set_global_assignment –name FAMILY "device family name"

Tcl: set_global_assignment –name DEVICE <part_number>
Chip Planner

Graphical view of

- Layout of device resources
- Routing channels between device resources
- Global clock regions

Uses

- View placement of design logic
- View connectivity between resources used in design
- Make placement assignments
- Debugging placement-related issues
Chip Planner

**Tools** menu or toolbar

![Image of Chip Planner interface]

- **Tasks window**
- **Report window**
- **Device floorplan aka Chip View**
- **Selected Node Properties**
- **Memory block in use**
- **Unused LAB**
- **Layers Settings**
Floorplan Views

Overall device resource usage

Lower level block usage

Lowest level routing detail

Zoom in for detailed logic implementation & routing usage
Pin Planner

Interactive graphical tool for assigning pins
- Drag & drop pin assignments
- Set pin I/O standards
- Reserve future I/O locations

Default window panes
- Package View
- All Pins list
- Groups list
- Tasks window
- Report window

Assignments menu → Pin Planner, toolbar, or Tasks window
Pin Planner Window

- Toolbar
- Groups list
- Package View
- Tasks pane
- All Pins list
The Programmer

Tools menu → Programmer
State Machine Editor

Create state machines in GUI

- Manually by adding individual states, transitions, and output actions
- Automatically with State Machine Wizard (Tools menu & toolbar)

Generate state machine HDL code (required)

- VHDL
- Verilog
- SystemVerilog
Components in system use different interfaces to communicate (some standard, some non-standard)

Typical system requires significant engineering work to design custom interface logic

Integrating design blocks and intellectual property (IP) is tedious and error-prone
Automatic Interconnect Generation

Avoids error-prone integration

Saves development time with automatic logic & HDL generation

Enables you to focus on value-add blocks

Platform Designer improves productivity by automatically generating the system interconnect logic
The Platform Designer GUI

Access in Tools menu, toolbar, or Tasks window

Draggable, detachable tabs

IP Catalog

System Contents

Hierarchy

Messages
RTL FLOW

Lab 1
FPGA PROGRAMMING MODEL:

High Level Synthesis
Can Also Be Wrapped With Higher Level Flows
The Software Programmer’s View

Programmers develop in mature software environments

- Ideas can easily be expressed in languages such as ‘C’
  - Typically start with simple sequential program
  - Use parallel APIs / language extensions to exploit multi core for additional performance
- Compilation times are almost instantaneous
- Immediate feedback
- Rich debugging tools
High Level Design is the Bridge Between HW & SW

100x More Software Engineers than Hardware Engineers

Key to wide-spread adoption of FPGA in Datacenter

Debugging software is much faster than hardware

Many functions are easier to specify in software than RTL

Simulation of RTL takes thousands times longer than software

Design Exploration is much easier and faster in software

We Need to Raise the Level of Abstraction

- Similar to what assembly programmers did with C over 30 years ago
  - (Today) Abstract away FPGA Design with Higher Level Languages
  - (Today) Abstract away FPGA Hardware behind Platforms
  - (Tomorrow) Leverage Pre-Compiled Libraries as Software Services
HLS Use Model

C/C++ Code

Standard gcc/g++ Compiler

HLS Compiler

100% Makefile compatible

EXE

HDL IP

src.c

i++ <options>

lib.h

a.exe

FPGA

Intel® Quartus® Ecosystem

Intel Proprietary
for LRZ
Intel® HLS Compiler

Targets Intel® FPGAs

Command-line executable: `i++`

Builds an IP block

- To be integrated into a traditional FPGA design using FPGA tools

Leverages standard C/C++ development environment

Goal: Same performance as hand-coded RTL with 10-15% more resources
HLS Procedure

Create Component and Testbench in C/C++

Functional Verification with g++ or i++
- Use \(-march=x86-64\)
- Both compilers compatible with GDB

Compile with i++ \(-march=<FPGA fam>\) for HLS
- Generates IP
- Examine compiler generated reports
- Verify design in simulation

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

Integrate IP with rest of your FPGA system
Intel® HLS Compiler Usage and Output

Develop with C/C++:

src.c  
lib.h

i++ -march=x86-64 src.c  
a.exe|out

Run Compiler for HLS:

src.c  
lib.h

i++ -march=<fpga fam> --component func src.c  
a.exe|out

GDB-Compatible Executable

Executable which will run calls to func in simulation of synthesized IP

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

Intel Proprietary

for LRZ
HLS Procedure: x86 Emulation

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

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

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

4. **Integrate IP with rest of your FPGA system**
Simple Example Program: `i++` and `g++` flow

Example Program

```c
// test.cpp
#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

Terminal Commands and Outputs

```
$ g++ test.cpp
$ ./a.out
Hello world
$

$ i++ test.cpp
$ ./a.out
Hello world
$
```

Using the default `–march=x86-64`
g++ Compatibility

Intel HLS Compiler is command line compatible with g++

- Similar command-line flags, x86 behavior, and compilation flow
- Changing “g++” to “i++” should just work
  - g++ <flags> <src>
  - i++ <flags> <src>
- x86 behavior should match g++
  - Except for integer promotion (discussed later)
- No source modifications required (for x86 mode)
- Support for GNU Makefiles
### i++ Options: g++ Compatible Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h</td>
<td>Display help information</td>
</tr>
<tr>
<td>-o &lt;name&gt;</td>
<td>Specify a non-default output name</td>
</tr>
<tr>
<td>-c</td>
<td>Instructs compiler generate the object files and not the executable</td>
</tr>
<tr>
<td>-march=&lt;arch&gt;</td>
<td>Compile for architecture x86-64 (Default) or &lt;FPGA Family&gt;</td>
</tr>
<tr>
<td>-v</td>
<td>Verbose mode</td>
</tr>
<tr>
<td>-g</td>
<td>Generate debug information (default)</td>
</tr>
<tr>
<td>-g0</td>
<td>Do not generate debug information</td>
</tr>
<tr>
<td>-I&lt;dir&gt;</td>
<td>Add to include path</td>
</tr>
<tr>
<td>-D&lt;macro&gt;[=&lt;val&gt;]</td>
<td>Define &lt;macro&gt; with &lt;val&gt; or 1</td>
</tr>
<tr>
<td>-l&lt;library&gt;</td>
<td>Library search directory and library name when linking</td>
</tr>
</tbody>
</table>

**Example:** `i++ -march=x86-64 myfile.cpp -o myexe`
## i++ Options: FPGA Related Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--component &lt;components&gt;</td>
<td>Specify a comma-separated list of function names to be synthesizes to RTL</td>
</tr>
<tr>
<td>--clock &lt;clock_spec&gt;</td>
<td>Optimizes the RTL for the specified clock frequency or period</td>
</tr>
<tr>
<td>-ghdl</td>
<td>Enable full debug visibility and logging of all signals when verification executable is run</td>
</tr>
<tr>
<td>--quartus-compile</td>
<td>Compiles the resulting HDL files using the Intel® Quartus® Prime software</td>
</tr>
<tr>
<td>--simulator &lt;simulator&gt;</td>
<td>Specify the simulator used for verification, “none” to skip testbench generation</td>
</tr>
<tr>
<td>--x86-only</td>
<td>Only create the executable for testbench, no RTL or cosim support</td>
</tr>
<tr>
<td>--fpga-only</td>
<td>Create FPGA component project, RTL and cosim support, no testbench binary</td>
</tr>
</tbody>
</table>

Example:  
```
i++ -march=<fpga fam> --component mycomp --clock 400Mhz myfile.cpp
```

There are many other optimization options available please see the *Intel HLS Compiler Reference Manual*.
The Default Interfaces

```cpp
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
Example Makefile

FILE := myapp
DEVICE := Arria10

all:
gpp: $(FILE).cpp
    g++ $(GCFLAGS) $(FILE).cpp -o $(FILE).out
emu: $(FILE).cpp
    i++ $(GCFLAGS) $(FILE).cpp -o $(FILE)_emu.out
fpga: $(FILE).cpp
    i++ $(GCFLAGS) $(FILE).cpp -o $(FILE)_fpga.out -march=$(DEVICE)
x86 Debugging Tools

printf/cout
gdb
Valgrind

Develop with C/C++:

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

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

```
a.exe|out
```
Using `printf()`

Requires “HLS/stdio.h”

- Maps to `<stdio.h>` when appropriate

Can be included in the testbench or the component

- Used with no limitations in the x86 emulation flow

`printf` statements inside the `component` ignored for HDL generation

- Ignored in the cosimulation flow with an HDL simulator
Using printf(): Example

Example Program

```c
// test.cpp
#include "HLS/stdio.h"

void say_hello() {
    printf("Hello from the component\n");
}

int main() {
    printf("Hello from the testbench\n");
say_hello();
    return 0;
}
```

Terminal Commands and output

```
$ i++ test.cpp
$ ./a.out
Hello from the testbench
Hello from the component
$

$ i++ test.cpp -march=Arria10 \
    --component say_hello
$ ./a.out
Hello from the testbench
$
```
Debugging Using gdb

i++ integrates well with GNU gdb

- Debug data is generated by default
  - Unlike g++, -g enabled by default, use -g0 to turn off debug data

-march=x86-64 flow:

- Can step through any part of the code (including the component)

-march=<fpga family> flow:

- Can step through testbench code

- gdb does not see the component side execution (that runs in an HDL simulator)
gdb Example

Example Program

```c
// test.cpp
#include "HLS/hls.h"
#include "HLS/stdio.h"

component void say_hello() {
    printf("Hello from the component\n");
}

int main() {
    printf("Hello from the testbench\n");
say_hello();
    return 0;
}
```

Terminal Commands and output

```
$ i++ test.cpp -march=x86-64 -o test-x86
$ gdb ./test-x86
………………………………………………………………
<GDB Command Prompt>
(gdb)
```

```
$ i++ test.cpp -march=Arria10 -o test-fpga
$ gdb ./test-fpga
………………………………………………………………
<GDB Command Prompt>
(gdb)
```
Debugging with Valgrind

“Valgrind is an instrumentation framework for building dynamic analysis tools.”

- Valgrind tools can detect:
  - Memory leaks
  - Invalid pointer uses
  - Use of uninitialized values
  - Mismatched use of malloc/new vs free/delete
  - Doubly freed memory
- Use to debug component and testbench in the x86 emulation flow
Simple Valgrind Example

Example Program:

```c
#include "hls/stdio.h"
#include <stdlib.h>

int bin_count (int *bins, int a) {
    return ++bins[a];
}

int main() {
    int *bins = (int *) malloc(16 * sizeof(int));
    srand(0);
    for (int i = 0; i < 256; i++) {
        int x = rand();
        int res = bin_count(bins, x);
        printf("Count val: %d\n", res);
    }
    return 0;
}
```

Terminal Commands and output:

```
$ i++ test.cpp
$ ./a.out
Segmentation Fault
$ valgrind --leak-check=full --show-reachable=yes ./a.out

==9744== Invalid read of size 4
==9744==    at 0x4006B3: bin_count(int*, int) (test.cpp:5)
==9744==    by 0x400723: main (test.cpp:13)
==9744==  Address 0x1b31075dc is not stack'd, malloc'd or (recently) free'd
==9744== Process terminating with default action of signal 11 (SIGSEGV)
==9744== Access not within mapped region at address 0x1b31075dc
==9744==    at 0x4006B3: bin_count(int*, int) (test.cpp:5)
==9744==    by 0x400723: main (test.cpp:13)
==9744== 64 bytes in 1 blocks are still reachable in loss record 1 of 1
==9744== Process terminating with default action of signal 11 (SIGSEGV)
==9744== 64 bytes in 1 blocks are still reachable in loss record 1 of 1
```

Example Program:

```c
// test.cpp
#include "hls/stdio.h"
#include <stdio.h>
#include <stdlib.h>

int bin_count (int *bins, int a) {
    return ++bins[a];
}

int main() {
    int *bins = (int *) malloc(16 * sizeof(int));
    srand(0);
    for (int i = 0; i < 256; i++) {
        int x = rand();
        int res = bin_count(bins, x);
        printf("Count val: %d\n", res);
    }
    return 0;
}
```

```
$ i++ test.cpp
$ ./a.out
Segmentation Fault
$ valgrind --leak-check=full --show-reachable=yes ./a.out

==9744== Invalid read of size 4
==9744==    at 0x4006B3: bin_count(int*, int) (test.cpp:5)
==9744==    by 0x400723: main (test.cpp:13)
==9744==  Address 0x1b31075dc is not stack'd, malloc'd or (recently) free'd
==9744== Process terminating with default action of signal 11 (SIGSEGV)
==9744== Access not within mapped region at address 0x1b31075dc
==9744==    at 0x4006B3: bin_count(int*, int) (test.cpp:5)
==9744==    by 0x400723: main (test.cpp:13)
==9744== 64 bytes in 1 blocks are still reachable in loss record 1 of 1
==9744== Process terminating with default action of signal 11 (SIGSEGV)
==9744== 64 bytes in 1 blocks are still reachable in loss record 1 of 1
```

Segmentation fault
Valgrind: Segmentation Fault Fixed

```c
int bin_count (int *bins, int a) {
    return ++bins[a % 16];
}

int main() {
    int *bins = (int *) malloc(16 * sizeof(int));
    srand(0);
    for (int i = 0; i < 256; i++) {
        int x = rand();
        int res = bin_count(bins, x);
        printf("Count val: \d\n", res);
    }
    free (bins);
    return 0;
}
```
HLS Procedure: Cosimulation

1. Create Component and Testbench in C/C++

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
Example Component/Testbench Source

```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()`
Translation from C function API to HDL module

All component functions are synthesized to HDL

- Each synthesized component is an independent HDL module

Component functions can be declared:

- Using component keyword in source
- Specifying “--component <component_name>” in the command-line
Cosimulation

Combines x86 testbench with RTL simulation

HDL code for the component runs in an RTL Simulator

- Verilog
- RTL testbench automatically created from software main() and everything else called from main runs on x86 as the testbench

Communication using SystemVerilog Direct Programming Interface (DPI)

- Allows C/C++ to interface SystemVerilog
- Inter-process communication (IPC) library used to pass testbench input data to RTL simulator, and returns the data back to the x86 testbench
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.
Default Simulation Behavior

Function calls to the simulator are sequential by default

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

cOMPONENT int acc (int a, int b) {
    return a+b;
}

int main() {
    int x1, x2, x3;
    x1=acc(1, 2);
    x2=acc(3, 4);
    x3=acc(5, 6);
    ...
}
```
Streaming Simulation Behavior

Use enqueue function calls to stream data into the component

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

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

int main() {
    int x1, x2, x3;
    altera_hls_enqueue(&x1, &acc, 1, 2);
    altera_hls_enqueue(&x2, &acc, 3, 4);
    altera_hls_enqueue(&x3, &acc, 5, 6);
    altera_hls_component_run_all("acc");
    ...
}
Viewing Component Waveforms

- Compile design with `i++ -ghdl flag`
  - Enable full visibility and logging of all HDL signals in simulation
- After cosimulation execution, waveform available at `a.prj/verification/vsim.wlf`
- Examine with the ModelSim GUI:
  - `vsim a.prj/verification/vsim.wlf`
Viewing Waveforms in Modelsim

Locate Component

Add Signals to Waveform
Cosimulation Design Process

Compile and verify on x86
- Iterate on the algorithm
- Functional verification
- Debugging using gdb/valgrind

Compile for FPGA
- Examine the FPGA reports
- Iterate on the architecture of the design
- Use the reports as feedback on what the bottlenecks are

Simulate using Modelsim
- Test functionality
- Test latency and performance (through verification stats)
Main HTML Report

The Intel® HLS Compiler automatically generates HTML report that analyzes various aspects of your function including area, loop structure, memory usage, and system data flow

- Located at a.prj/reports/report.html
HTML Report: Summary

Overall compile statistics

- FPGA Resource Utilization
- Compile Warnings
- Quartus® fitter results
  - Available after Quartus compilation
- etc.
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

Diagram:

- Loop
  - 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
- Loop must not have dependency from iteration to iteration

![Loop Unrolling Diagram]
Loop Pipelining

Loop pipelining: Launch loop iterations as soon as dependency is resolved

- Initiation interval (II): launch frequency (in cycles) of a new loop iteration
  - II=1 is optimally pipelined
    - No dependency or dependencies can be resolved in 1 cycle
Loop analysis shows how loops are implemented
  – Ability to correlate with source code

<table>
<thead>
<tr>
<th>Loops analysis</th>
<th>Pipelined</th>
<th>II</th>
<th>Bottleneck</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>whiletrue.entry</td>
<td>yes</td>
<td>1</td>
<td>n/a</td>
<td>Serial line: Memory dependency</td>
</tr>
<tr>
<td>for.body (example.cpp:14)</td>
<td>yes</td>
<td>1</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>for.cond13.preheader</td>
<td>yes</td>
<td>2</td>
<td>II</td>
<td>Memory dependency</td>
</tr>
<tr>
<td>Fully unrolled loop</td>
<td>yes</td>
<td>n/a</td>
<td>n/a</td>
<td>Unrolled by #pragma unroll</td>
</tr>
<tr>
<td>for.body34 (example.cpp:25)</td>
<td>yes</td>
<td>1</td>
<td>n/a</td>
<td></td>
</tr>
</tbody>
</table>

Compiler-added loop, not in the code, implicit infinitely loop allowing the component to run continuously in pipelined fashion

Pipelined loop, II=1

Pipelined loop, II=2 due to memory dependency

Fully unrolled loop, due to user 

#pragma unroll
HTML Report: Area Analysis

View detailed estimated resource consumption by system or source line

- Analyze data control overhead
- View memory implementation
- Shows resource usage
  - ALUTs
  - FFs
  - RAMs
  - DSPs
- Identifies inefficient uses
HTML Report: Component Viewer

Displays abstracted netlist of the HW implementation

- View data flow pipeline
  - See loads and stores
  - Interfaces including stream reads and writes
  - Memory structure
  - Loop structure
  - Possible performance bottlenecks
    - Unpipelined loops are colored light red
    - Stallable points are red

Mouse over node to see tooltip and details.
Correlates with source code.
HTML Report: Memory Viewer

Displays local memory implementation and accesses

- Visualize memory architecture
  - Banks, widths, replication, etc

- Visualize load-store units (LSUs)
  - Stall-free?
  - Arbitration
  - Red indicates stallable

Mouse over node to see tooltip and details.
Correlates with source code.
HTML Report: Verification Statistics

Reports execution statics from testbench execution, available after component is simulated (testbench executable ran)

- Number and type of component invocation
- Latency of component
- Dynamic Initiation interval of Component
- Data rates of streams

Measurements based on latest execution of testbench
HLS Procedure: Integration

1. **Create Component and Testbench in C/C++**

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**
Quartus® Generated QoR Metrics for IP

Use Intel® Quartus® Prime software to generate quality-of-result reports

- i++ creates the Quartus project in a.prj/quartus

- To generate QoR data (final resource utilization, fmax)
  - Run `quartus_sh --flow compile quartus_compile`
  - Or use `i++ --quartus-compile opt`

- Report part of the HTML report
  - a.prj/reports/report.html
  - Summary page
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
HDL Instantiation

Add Components to Intel® Quartus Project

- `<component>.qsys` to Standard Edition
- `<component>.ip` to Pro Edition

Instantiate component module in your design:

- Use template

```
a.prj/components/<component>/component_inst.v
```
Platform Designer System Integration Tool

- Catalog of available IP
  - Interface protocols
  - Memory
  - DSP
  - Embedded
  - Bridges
  - PLL
  - Custom Components
  - Custom Systems

Accelerate development

Automate integration tasks

Connect custom IP and systems

Simplify integration

- IP 1
- Custom 1
- IP 2
- IP 3
- Custom 2

HDL
Platform Designer Integration

Platform Designer component generated for each component:

- For PD Standard – `a.prj/components/<component>/<component>.qsys`
- For Platform Designer – `a.prj/components/<component>/<component>.ip`

In Platform Designer, instantiate component from the IP Catalog in the HLS project directory

- Add IP directory to IP Catalog Search Locations
  - May use `a.prj/components/**/*`
- Can be stitched with other user IP or Intel® Quartus® IP with compatible interfaces

See tutorials under tutorials/usability
Platform Designer HLS Component Example

Example

- Cascaded low-pass filter and high-pass filter
HLS-Backed Components

- Generic component can be used in place of actual IP core
- Choose Implementation Type: HLS

- Specify HLS source files
- Compile Component
- Run Cosim
- Display HTML report
FPGA PROGRAMMING MODEL:

OpenCL
Intel FPGA SDK for OpenCL™ Flow

A system level view:

Kernel compiler:
- Optimized pipelines from C/C++

Board support package: (created by hardware developer)
- Timing closure, pinouts, periphery planning – we’ve got it covered

System integrator: (Quartus runs behind the scenes)
- Optimized I/O interconnects
OpenCL

Hardware Agnostic Compute Language
Invented by Apple

- 2008 Specification donated to Khronos Group
- Now managed by Intel

OpenCL C and C++

What does OpenCL™ give us?
- Industry standard programming model
- Functional portability across platforms
- Well thought out specification
Heterogeneous Platform Model

OpenCL Platform Model

Host Memory

Host

Global Memory

Device

Example Platform

x86

PCIe
OpenCL Use Model: Abstracting the FPGA away

Host Code

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

OpenCL Accelerator Code

```c
__kernel __global void sum
(__global float *a,
 __global float *b,
 __global float *y)
{
    int gid = get_global_id(0);
    y[gid] = a[gid] + b[gid];
}
```
OpenCL Host Program

Pure software written in standard C/C++ languages

Communicates with the accelerator devices via an API which abstracts the communication between the host processor and the kernels

```c
main()
{
    read_data_from_file( ... );
    manipulate_data( ... );
    clEnqueueWriteBuffer( ... );
    clEnqueueNDRange(..., sum, ...);
    clEnqueueReadBuffer( ... );
    display_result ( ... );
}
```
OpenCL Kernels

Kernel: Data-parallel function

- Defines many parallel threads
- Each thread has an identifier specified by "get_global_id"
- Contains keyword extensions to specify parallelism and memory hierarchy

Executed by an OpenCL device

- CPU, GPU, FPGA

Code portable NOT performance portable

- Between FPGAs it is!

```c
__kernel void sum(
    __global float *a,
    __global float *b,
    __global float *answer)
{
    int xid = get_global_id(0);
    result[xid] = a[xid] + b[xid];
}
```
Software Engineer’s View of an OpenCL System

Device contains compute engines that run the kernel
Host talks to global memory through OpenCL routines
Global memory is large, fast, and likes to burst
Local memory is small, fast, and supports random access
FPGA OpenCL Architecture

- Modest external memory bandwidth
- Extremely high internal memory bandwidth
- Highly customizable compute cores
Start with a Reference Platform (1/2)

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Network Enabled</th>
<th>High Performance Computing (HPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Latency</td>
<td>Compute Power/ Memory Bandwidth</td>
<td></td>
</tr>
</tbody>
</table>

### Architecture

<table>
<thead>
<tr>
<th>Global Memory</th>
<th>2x10GbE (MAC/UQE)</th>
<th>Large amount of DDR</th>
</tr>
</thead>
</table>

```mermaid
diagram flow
   start:Start with a Reference Platform
   network
   lowLatency
   highPerformance
   architecture
   globalMemory
   IO Channels
   end
```

Intel Proprietary for LRZ
Start with a Reference Platform (2/2)

Host and accelerator in same package: SoC
Development Flow using SDK

1. Modify kernel.cl
2. x86 Emulator (sec)
3. Optimization Report (sec)
4. Profiler (hours)

DONE!

Functional Bugs?
Memory Dependencies?

Hardware performance Not met?
Compiling Kernel

Run the Altera Offline Compiler in command prompt

- `aoc --board <board> <Kernel.cl>`
- Run `aoc --list-boards` to see all available boards

AOC performs system integration to generate the kernel hardware system and the Quartus Prime software to compile the design

```
/mydesigns/matrixMult$ aoc matrixMul.cl
aoc: Selected target board bittware_s5pciehq

+--------------------------------------------------------------------+
| ; Estimated Resource Usage Summary                                 |
| ; Resource + Usage                                                |
| +------------------------------------------------------------------|
| ; Logic utilization ; 52%                                         |
| ; Dedicated logic registers ; 23%                                 |
| ; Memory blocks ; 31%                                             |
| ; DSP blocks ; 54%                                                |
+--------------------------------------------------------------------+
```
Executing the kernel: clCreateProgramWithBinary

```c
const char** fp = fopen("file.aocx","rb");
  fseek(fp,0,SEEK_END);
  lengths[0] = ftell(fp);
  binaries[0] = (unsigned char*)malloc(sizeof(unsigned char)*lengths[0]);
  rewind(fp);
  fread(binaries[0],lengths[0],1,fp);
  fclose(fp);
```

clCreateProgramWithBinary

Program (exe)

const char**

clBuildProgram

clCreateProgram

cl_getDevices

cl_getPlatforms

cl_getContext

clCreateKernel

clEnqueueNDRangeKernel

cl_context

cl_device

cl_platform

cl_device

cl_context

clCreateCommandQueue

clCreateProgram

exe

Kernel (src)

Kernel (src)

cl_program

cl_kernel

clEnqueueNDRangeKernel

Offline Compiler

CL File

≈ OpenCL “Program”

≈ Bitstream

Intel Proprietary
for LRZ
Development Flow using SDK

1. Modify kernel.cl
2. x86 Emulator (sec)
3. Optimization Report (sec)
4. Profiler (hours)

DONE!

Hardware performance Not met?

Functional Bugs?

Memory Dependencies?
Emulator – The Flow

Generate emulation aocx

Run host program with emulator aocx

- Host compile does not change
- set CL_CONTEXT_EMULATOR_DEVICE_ALTERA=<number_of_boards>

```
c:\opencl\aoc -march=emulator conv.cl
c:\opencl\dir
host.exe conv.cl conv.aocx
```

```
c:\opencl\host.exe
running...
done!
```
Printf

Can use printf within kernel on FPGA
- Adds some memory traffic overhead

In the emulator, printf runs on IA
- Useful for fast debug iterations
Development Flow using SDK

1. Modify kernel.cl
2. x86 Emulator (sec)
3. Optimization Report (sec)
4. Profiler (hours)

DONE!

Hardware performance Not met?

Functional Bugs?

Memory Dependencies?
Optimization Report

Intel FPGA SDK for OpenCL provides a static report to identify performance bottlenecks when writing single-threaded kernels.

Use –c to stop after generating the reports

- `aoc -c <kernel.cl>`
- Report is in: `<kernel>/reports/report.html`
<table>
<thead>
<tr>
<th>Pipelined</th>
<th>II</th>
<th>Bottleneck</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>1</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>li</td>
<td>li</td>
<td>Memory dependency and...</td>
</tr>
<tr>
<td>Yes</td>
<td>1</td>
<td>n/a</td>
<td></td>
</tr>
</tbody>
</table>

**Details**

**Block3:**

II bottleneck due to memory dependency between:
- Load Operation (acc.cl:12)
- Store Operation (acc.cl:13)

Largest critical path contributor(s):
- 31%: Load Operation (acc.cl:12)
- 27%: Load Operation (acc.cl:13)
- 18%: Hardened Floating Point Multiply-Add Operation (acc.cl:12)
- 14%: Store Operation (acc.cl:12)
Development Flow using SDK

1. Modify kernel.cl
2. x86 Emulator (sec)
3. Optimization Report (sec)
4. Profiler (hours)

DONE!

Hardware performance Not met?

Functional Bugs?
Memory Dependencies?
Profiler – the flow

1. Generate program bitstream with profiling enabled

```c
kernel void convolution(
    global int * filter_coef,
    global int * input_image,
    global int * output_image
) {
    int grid = get_group_id(0);
}
```

2. Run host program with instrumented aocx

```
c:\\opencl>dir
host.exe  conv.aocx
```

```
c:\\opencl>host.exe
running...
done!
```

```
c:\\opencl>dir
host.exe  conv.aocx  profile.mon
```

3. Run the profiler GUI:

```bash
aocl report <aocx> <profile.mon>
```
Dynamic Profiler

Intel FPGA SDK for OpenCL enables users to get runtime information about their kernel performance.

Bottlenecks, bandwidth, saturation, pipeline occupancy.

Performance Stats

Execution Times
HIGH PERFORMANCE DATA FLOW
Execution of Threads on FPGA – Naïve Approach

Thread execution can be executed on replicated pipelines in the FPGA

```c
kernel void
add( global int* Mem ) {
    ...
    Mem[100] += 42*Mem[101];
}
```
Execution of Threads on FPGA – Naïve Approach

Thread execution can be executed on *replicated* pipelines in the FPGA

```c
kernel void
add( global int* Mem ) {
    ...
    Mem[100] += 42*Mem[101];
}
```
Execution of Threads on FPGA – Naïve Approach

Thread execution can be executed on replicated pipelines in the FPGA

- **Throughput** = 1 thread per cycle
- Area inefficient

![Diagram showing parallel threads](image)
Execution of Threads on FPGA

Better method involves taking advantage of *pipeline parallelism*

- Attempt to create a deeply pipelined implementation of kernel
- On each clock cycle, we attempt to send in new thread

```c
kernel void add( global int* Mem ) {
    ...
    Mem[100] += 42*Mem[101];
}
```
Execution of Threads on FPGA

Better method involves taking advantage of *pipeline parallelism*

- Attempt to create a deeply pipelined implementation of kernel
- On each clock cycle, we attempt to send in new thread

```c
kernel void add( global int* Mem ) {
   ...
   Mem[100] += 42*Mem[101];
}
```
Execution of Threads on FPGA

Better method involves taking advantage of pipeline parallelism

- Attempt to create a deeply pipelined implementation of kernel
- On each clock cycle, we attempt to send in new thread

```c
kernel void add( global int* Mem ) {
    ...
    Mem[100] += 42*Mem[101];
}
```
Execution of Threads on FPGA

Better method involves taking advantage of *pipeline parallelism*:

- Attempt to create a deeply pipelined implementation of kernel
- On each clock cycle, we attempt to send in new thread

```c
kernel void
add( global int* Mem ) {
    ...
    Mem[100] += 42*Mem[101];
}
```
Execution of Threads on FPGA

Better method involves taking advantage of *pipeline parallelism*

- Attempt to create a deeply pipelined implementation of kernel
- On each clock cycle, we attempt to send in new thread

```c
kernel void add( global int* Mem ) {
  ...
  Mem[100] += 42*Mem[101];
}
```
Execution of Threads on FPGA

Better method involves taking advantage of *pipeline parallelism*

- Attempt to create a deeply pipelined implementation of kernel
- On each clock cycle, we attempt to send in new thread

```c
kernel void
add( global int* Mem ) {
  ...
  Mem[100] += 42*Mem[101];
}
```
Execution of Threads on FPGA

Better method involves taking advantage of **pipeline parallelism**

- **Throughput = 1 thread per cycle**
SINGLE THREADED OPTIMIZATIONS
OpenCL on Intel FPGAs

Main assumptions made in previous OpenCL programming model

- Data level parallelism exists in the kernel program

Not all applications well suited for this assumption

- Some applications do not map well to data-parallel paradigms

These are the only workloads that GPUs support
Data-Parallel Execution

On the FPGA, we use the idea of pipeline parallelism to achieve acceleration.

```c
kernel void sum(
    global const float *a,
    global const float *b,
    global float *c)
{
    int xid = get_global_id(0);
    c[xid] = a[xid] + b[xid];
}
```

Threads can execute in an embarrassingly parallel manner.
Data-Parallel Execution - Drawbacks

Difficult to express programs which have partial dependencies during execution

Would require complicated hardware and new language semantics to describe the desired behavior

```c
kernel void sum(global const float *a, 
    global const float *b, 
    global float *c)
{
    int xid = get_global_id(0);
    c[xid] = c[xid-1] + b[xid];
}
```
Solution: Tasks and Loop-Pipelining

Allow users to express programs as a single-thread

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

Pipeline parallelism still leveraged to efficiently execute loops in Intel's FPGA OpenCL

- Parallel execution inferred by compiler
- Loop Pipelining
Loop Carried Dependencies

Loop-carried dependencies are dependencies where one iteration of the loop depends upon the results of another iteration of the loop.

```c
kernel void state_machine(ulong n)
{
    t_state_vector state = initial_state();
    for (ulong i=0; i<n; i++) {
        state = next_state( state );
        unit y = process( state );
        write_channel_altera(OUTPUT, y);
    }
}
```

The variable state in iteration 1 depends on the value from iteration 0. Similarly, iteration 2 depends on the value from iteration 1, etc.
Loop Carried Dependencies

To achieve acceleration, we can *pipeline* each iteration of a loop containing loop carried dependencies

- Analyze any dependencies between iterations
- Schedule these operations
- Launch the next iteration as soon as possible

```c
kernel void state_machine(ulong n)
{
    t_state_vector state = initial_state();
    for (ulong i=0; i<n; i++) {
        state = next_state( state );
        unit y = process( state );
        write_channel_altera(OUTPUT, y);
    }
}
```

At this point, we can launch the next iteration
Loop Pipelining Example

No Loop Pipelining

Clock Cycles

i=0

i=1

i=2

No Overlap of Iterations!

With Loop Pipelining

Clock Cycles

i=0

i=1

i=2

i=3

i=4

i=5

Finishes Faster because Iterations Are Overlapped

Looks almost like multi-threaded execution!
Parallel Threads vs. Loop Pipelining

So what’s the difference?

Loop Pipelining enables Pipeline Parallelism *AND* the communication of state information between iterations.
Image Filter

// Note that no thread identifiers anywhere in the kernel - single threaded code!
kernel void sobel ( global char * restrict data_in, global char * restrict data_out,
   unsigned iterations, int threshold ) {

   const int filterH[3][3] = { {-1,0,1}, {-2,0,2}, {-1,0,1} };  
   const int filterV[3][3] = { {-1,-2,-1}, {0,0,0}, {1,2,1} };  

   char rows[2 * WIDTH + 3]; // Pixel buffer of 2 rows and 3 extra pixels

   int count = 0;
   while (count != iterations) {
     // Each cycle, shift a new pixel into the buffer.
     // Unrolling this loop allows the compiler to infer a shift register.
     #pragma unroll
     for (int i = WIDTH * 2 + 2; i > 0; --i) { 
       rows[i] = rows[i - 1];
     }
     rows[0] = data_in[count]; // Shift image data (from DDR) into one end

     int accumH=0, accumV=0;
     for (unsigned y=0; y<TILE SIZE; y++) {
       for (unsigned x=0; x<TILE SIZE; x++) {
         unsigned int pixel = rows[y * WIDTH + x];

         accumH += pixel * filterH[y][x];
         accumV += pixel * filterV[y][x];
       }

     }

     int accum = accumH*accumH + accumV*accumV;
     char out_val = (accum > (threshold * threshold)) ? 255 : 0;
     data_out[count++] = out_val; //output pixel (to DDR)
   }
CHANNLES
Harnessing Dataflow to Reduce Memory Bandwidth
Data Movement in GPUs

Data is moved from host over PCIexpress

Instructions and data is constantly sent back and forth between host cache and memory and GPU memory

- Requires buffering larger data sets before passing to GPU to be processed
- Significant latency penalty
- Requires high memory and host bandwidth
- Requires sequential execution of kernels
Altera_Channels Extension

An FPGA has programmable routing

Can't we just send data across wires between kernels?

Advantages:

- Reduce memory bandwidth
- Lower latency through fine-grained synchronization between kernels
- Reduce complexity (wires are trivial compared to memory access)
  - Lower cost, lower area, higher performances
- Enable modular dataflow design through small kernels exchanging data
- Different workgroup sizes and degrees of parallelism in connected modules
Data Movement in FPGAs

FPGA allows for result reuse between instructions

Ingress/Egress to custom functions 100% flexible

Multiple memory banks of various types directly off FPGA

- Algorithms can be architected to minimize buffering to external memory or host memory
- Multiple optional memory banks can be used to allow simultaneous access
Example: Multi-Stage Pipeline

An algorithm may be divided into multiple kernels:

- Modular design patterns
- Partition the algorithm into kernels with different sizes and dimensions
- Algorithm may naturally split into both single-threaded and NDRange kernels

Generating random data for a Monte Carlo simulation:

```
kernel void rng(int seed) {
    int r = seed;
    while (true) {
        r = rand(r);
        write_channel_altera(RAND, r);
    }
}
```

```
kernel void sim(...) {
    int gid = get_global_id(0);
    int rnd = read_channel_altera(RAND);
    out[gid] = do_sim(data, rnd);
}
```

Single-Threaded

NDRange
Traditional Data Movement Without Channels

- DDR
- Memory Controller
- DMA
- Kernel
- PCIe
- System Memory
- HOST
- FPGA
- HOST
- DDR
- PCIe
- DMA
- Kernel
- Memory Controller
- DMA
- DDR
- System Memory
- HOST
- DDR
- FPGA
Data Movement Using Channels

- FPGA
- Host
- System Memory
- PCIe
- DMA
- DDR
- Memory Controller
- Data In
- Data Out
- Kernel
- FIFO
- DMA
Data Movement Using Host Channels

- DDR
- Memory Controller
- DMA
- Kernel
- DMA
- PCIe
- System Memory
- HOST
- FPGA

Intel Proprietary for LRZ
Kernel Replication with num_compute_units using OpenCL

- Step #1: Design an efficient kernel
- Step #2: How can we scale it up?

```c
kernel void PE() {
    ...
}
```
Kernel Replication With Intel® FPGA SDK for OpenCL

Attribute to specify 1-dim or 2-dim array of kernels

Add API to identify kernel in the array

```
__attribute__((num_compute_units(4,4)))
kernal void PE() {
    row = get_compute_id(0);
    col = get_compute_id(1);
    ...
}
```

Compile-time constants allows compiler to specialize each PE
Kernel Replication With Intel® FPGA SDK for OpenCL

Topology can be expressed with software constructs

- Channel connections specified through compute IDs

```c
channel float4 ch_PE_row[4][4];
channel float4 ch_PE_col[4][4];
channel float4 ch_PE_row_side[4];
channel float4 ch_PE_col_side[4];

__attribute__((num_compute_units(4,4)))
kerneld void PE() {
    row = get_compute_id(0);
    col = get_compute_id(1);

    float4 a,b;
    if (row==0)
        a = read_channel(ch_PE_col_side[col]);
    else
        a = read_channel(ch_PE_col[row-1][col]);
    if (col==0)
        ...
}
```
Matrix Multiply in OpenCL

Every PE / feeder is a kernel

Communication via OpenCL channels

Data-flow network model

Software control:
- Compute unit granularity
- Spatial Locality
- Interconnect topology
- Data movement
- Caching
- Banking

Performance: ~1 TFLOPs
Traditional CNN

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

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
CNN On FPGA

Want to minimize accessing external memory

Want to keep resulting data between layers on the device and between computations

Want to leverage reuse of the hardware between computations

Parallelism in the depth of the kernel window and across output features. Defer complex spatial math to random access memory.

Re-use hardware to compute multiple layers.
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**
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</th>
<th>Structure</th>
<th>Sign, Exponent, Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP16</td>
<td>5-bit exponent, 10-bit mantissa</td>
<td></td>
</tr>
<tr>
<td>FP11</td>
<td>5-bit exponent, 5-bit mantissa</td>
<td></td>
</tr>
<tr>
<td>FP10</td>
<td>5-bit exponent, 4-bit mantissa</td>
<td></td>
</tr>
<tr>
<td>FP9</td>
<td>5-bit exponent, 3-bit mantissa</td>
<td></td>
</tr>
<tr>
<td>FP8</td>
<td>5-bit exponent, 2-bit mantissa</td>
<td></td>
</tr>
</tbody>
</table>
FPGA PROGRAMMING MODEL:

DSP Builder Advanced Blockset
The Mathworks* Design Environment

- **Matlab***
  - High-level technical computing language
    - Simple C like language
    - Efficient with vectors and matrices
    - Built-in mathematical functions
  - Interactive environment for algorithm development
    - 2D/3D graphing tool for data visualization

- **Simulink***
  - Hierarchical block diagram design & simulation tool
  - Digital, analog/mixed signal & event driven
  - Visualize signals
  - Integrated with MATLAB*
DSP Builder for Intel® FPGAs

Enables MathWorks* Simulink for Intel FPGA design

Device optimized Simulink* DSP Blockset

- Key Features:
  - High-Level Design Exploration
  - HW-in-the-Loop verification
  - IP Generation for Intel® Quartus SW / Platform Designer
FPGA Design Flow - Traditional

**Development**
- System Level Design
- System Level Simulation
- MATLAB®/Simulink® tools
- System Engineer

**Implementation**
- HDL Coding
- DSP IP
- Precision®, Synplify® SW
- Intel® Quartus® Prime SW
- Hardware Engineer

**Verification**
- RTL Simulation
- Hardware Verification
- ModelSim® tools
- Development Kits
- Verification Engineer
FPGA Design Flow – DSP Builder for Intel® FPGAs

Development
- System Level Design
- System Level Simulation
- Algorithm-level Modeling
- MATLAB*/Simulink* tools

Implementation
- HDL Coding
- DSP IP
- Synthesis, RTL Simulation
- Precision*, Synplify* SW
  Intel® Quartus® Prime SW

Verification
- RTL Simulation
- Hardware Verification
- System-level Verification
- ModelSim* tools
  Development Kits

Single Simulink* Representation
Core Technologies

- IP (ready made) library
  - Multi-rate, multi-channel filters
  - Waveform synthesis (NCO/DDS/Mixers)
- Custom IP creation using primitive library
  - Vectorization
  - Zero latency
  - Scheduled
  - Aligned RTL generation
- System integration
  - Platform Designer
  - Processor Integration
- Automatic pipelining
- Automatic folding and resource sharing
- Multichannel designs with automatic vectorization
- Avalon® Memory-Mapped and Streaming Interfaces
- Design exploration across device families
- High-performance floating-point designs
- System-in-the-Loop accelerated simulation
Advanced Blockset - High Performance DSP IP

Over 150 device optimized DSP building blocks for Intel® FPGAs

- DSP building blocks
- Interfaces
- IP library blocks
- Primitives library blocks
  - Math and Basic blocks
- Vector and Complex data types
Build Custom FFTs from FFT Element Library

- Quickly build DSP designs using Complete FFT IP Functions from the FFT Library
- Build custom radix-$2^2$ FFTs using blocks from the FFT Element Library

<table>
<thead>
<tr>
<th>FFT IP Library</th>
<th>FFT Element Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>Pruning and Twiddle</td>
</tr>
<tr>
<td>FFT_float</td>
<td>Bit vector combine</td>
</tr>
<tr>
<td>VFFT</td>
<td>Butterfly Unit</td>
</tr>
<tr>
<td>VFFT_float</td>
<td>Choose Bits</td>
</tr>
<tr>
<td>BitReverseCoreC</td>
<td>Dual Twiddle Memory</td>
</tr>
<tr>
<td>VariableBitReverse</td>
<td>Edge Detect</td>
</tr>
<tr>
<td></td>
<td>Floating-Point Twiddle Gen</td>
</tr>
<tr>
<td></td>
<td>Crossover Switch</td>
</tr>
</tbody>
</table>
Filter and Waveform Synthesis Library

DSP Builder includes a comprehensive waveform IP library

- Automatic resource sharing based on sample rate
- Support for super sample rate architectures

<table>
<thead>
<tr>
<th>IP</th>
<th>Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td>• Half-band</td>
</tr>
<tr>
<td></td>
<td>• L-Band</td>
</tr>
<tr>
<td></td>
<td>• Symmetric</td>
</tr>
<tr>
<td></td>
<td>• Decimating</td>
</tr>
<tr>
<td></td>
<td>• Fractional Rate</td>
</tr>
<tr>
<td></td>
<td>• Interpolation</td>
</tr>
<tr>
<td></td>
<td>• Single-Rate</td>
</tr>
<tr>
<td></td>
<td>• Super Sample Rate</td>
</tr>
<tr>
<td>CIC</td>
<td>• Decimating</td>
</tr>
<tr>
<td></td>
<td>• Interpolating</td>
</tr>
<tr>
<td></td>
<td>• Super Sample Rate</td>
</tr>
<tr>
<td>Mixer</td>
<td>• Complex</td>
</tr>
<tr>
<td></td>
<td>• Real</td>
</tr>
<tr>
<td></td>
<td>• Super Sample Rate</td>
</tr>
<tr>
<td>NCO</td>
<td>• Super Sample Rate</td>
</tr>
<tr>
<td></td>
<td>• Multi-bank</td>
</tr>
</tbody>
</table>
Library is Technology Independent

- Target device using a **Device** block
- Same model generates optimized RTL for each FPGA and speed grade
Datapath Optimization for Performance

Automatic Timing Driven Synthesis of Model
- Based on specified device and clock frequency

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipelining</td>
<td>Inserts registers to improve Fmax</td>
</tr>
<tr>
<td>Algorithmic Retiming</td>
<td>Moves registers to balance pipelining</td>
</tr>
<tr>
<td>Bit Growth Management</td>
<td>Manages bit growth for fixed-point designs</td>
</tr>
<tr>
<td>Multi-rate Optimizations</td>
<td>Optimizes hardware based on sample rate</td>
</tr>
</tbody>
</table>

![Optimization Diagram]

Before Optimization

- A → B → C

After Optimization

- A → B → C

![Retiming Diagram]

- Retiming
- Bit Growth

Intel Proprietary for LRZ
Custom IP Generation

Model Primitive Features
- Vector support
- Parameterizable
- Zero latency block
- ALU folding

What to do not when to do it
ALU Design Folding Improves Area Efficiency

Optimizes hardware usage for low-throughput designs

- Arranges one of each resources in a central arithmetic logic unit (ALU) fashion
- Folding factor = clock rate / data rate
- Performed when Folding factor > 500
TDM Resource Sharing

Clock Rate = Sample Rate

Clock Rate = 2*Sample Rate

TDM_CLK
## TDM Design: Trade-Off Example

<table>
<thead>
<tr>
<th>Stratix 10</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUT4s</td>
</tr>
<tr>
<td><strong>Clock Rate = 72 MHz</strong>&lt;br&gt;Sample Rate = 72 MSPS</td>
<td>898</td>
</tr>
<tr>
<td><strong>Clock Rate = 144 MHz</strong>&lt;br&gt;Sample Rate = 72 MSPS</td>
<td>1082</td>
</tr>
<tr>
<td><strong>Clock Rate = 288 MHz</strong>&lt;br&gt;Sample Rate = 72 MSPS</td>
<td>741</td>
</tr>
<tr>
<td><strong>Clock Rate = 72 MHz</strong>&lt;br&gt;Sample Rate = 36 MSPS</td>
<td>1082</td>
</tr>
</tbody>
</table>

49-tap Symmetric Single Rate FIR Filter
2 Antenna DUC Reference Design

**ChannelFIR:**
- ChanCount = 4
- Output Sample Rate = 11.2 MSPS
- Output Period = 16
- Output Seq. = I1, I2, Q1, Q2, zeros(1, 16-4)

**Interpolate4FIR:**
- ChanCount = 4
- Output Sample Rate = 89.6 MSPS
- Output Period = 2
- ChanWireCount = ceil(4/2) = 2
- ChanCycleCount = ceil(4/2) = 2
- Output Seq. = I1, I2, Q1, Q2, zeros(1, 8-4)

**Interpolate2FIR:**
- Clock Rate = 179.2 MHz
- ChanCount = 4
- Output Sample Rate = 22.4 MSPS
- Output Period = 8
- Output Seq. = I1, I2, Q1, Q2, zeros(1, 8-4)

**NCO:**
- ChanCount = 2 (complex channel)
- Sample Rate = 89.6 MSPS
- Period = 2
- Sine Seq. = sinA1, sinA2
- Cosine Seq. = cosA1, cosA2

**ComplexMixer:**
- ChanCount = 2 (complex channel)
- Sample Rate = 89.6 MSPS
- Period = 2
- I' = I*cos − Q*sin
- Q' = I*sin + Q*cos
- Output i Seq. = I1, I2
- Output q Seq. = Q1, Q2 (Terminated)

**Deinterleaver:**
- Sample Rate = 89.6 MSPS
- Period = 2
- Input I Seq. = I1, I2
- Antenna 1 Seq. = I1,
- Antenna 2 Seq. = I2,
Changing the Design without DSP Builder

- Tedious and time consuming
- Channel Count = 8, 16, 32
- Clock Rate = 2x, 4x

Specification:
SampleRate = 11.2
ChanCount = 8
Changing the Design with DSP Builder

- Modifications done in minutes
- Design still looks the same

Specification:
SampleRate = 11.2
ChanCount = 8
## Five Designs Iterations < 1 Hour

<table>
<thead>
<tr>
<th></th>
<th>Arria® 10 6 channel</th>
<th>Arria 10 6 channel</th>
<th>Arria 10 12 channel</th>
<th>Stratix® 10 6 channel</th>
<th>Stratix 10 12 channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requested Clock (MHz)</td>
<td>250</td>
<td>450</td>
<td>450</td>
<td>450</td>
<td>450</td>
</tr>
<tr>
<td>Actual Fmax (slow model, 85C)</td>
<td>351</td>
<td>458</td>
<td>458</td>
<td>524</td>
<td>484.5</td>
</tr>
<tr>
<td>Multiplier Count (18x18)</td>
<td>10</td>
<td>6</td>
<td>10</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Logic Resources (registers)</td>
<td>686</td>
<td>465</td>
<td>818</td>
<td>1267</td>
<td>1863</td>
</tr>
<tr>
<td>Block Memory Resources (kbits)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>25.8</td>
</tr>
</tbody>
</table>
Generates Reusable IP for Platform Designer

- Platform Designer is the System Integration Environment for Intel® FPGAs
- DSP Builder designs fully compatible with Platform Designer
- Integrate with other FPGA IPs
  - Processors
  - State machines
  - Streaming interfaces
- Design reuse fully supported
Typical Design Flow

Identify system architecture, design filters and choose desired Fmax and device

Set the top level system parameters in the MATLAB® software using the ‘params’ file - number of channels, performance, etc.

Build the system using the Advanced Blockset tool

Simulate the design using Simulink® and ModelSim® tools

Target the right FPGA family and compile

As system design specs changes, edit the ‘params’ file and repeat
Design Flow – Create Model

Create a new blank model

Select New Model Wizard from DSP Builder menu
Top-level of a DSPB-AB design is a testbench
Must include Control and Signals blocks
Design Flow - Synthesizable Model

Enter the design in the subsystem

Device block marks the top level of the FPGA
Design Flow – ModelIP Blocks

Filters Library
- Single rate, multi-rate, and fractional rate FIR filters
- Decimating and interpolating cascaded integrator comb (CIC) filters

Note: Supports super-sample rate (data rate > system clock freq) interpolation by 2 filters.

Waveform Synthesis Library
- Real and complex mixer
- Numerically controlled oscillator (NCO)

Note: The NCO block supports frequency hopping (each channel can hop to different frequency from a pool of frequencies)
Design Flow – ModelPrim Blocks

ChannelIn and ChannelOut blocks to delineate the boundary of a synthesizable primitive subsystem

Add SynthesisInfo Block to control pipelining and latency and to view resource usage of the subsystem
Design Flow – Parameterize the Design

C structure like template
Runs when model is opened or simulation is run
Design Flow – Processor Interface

Drop memory and registers in the design

ModelIPs have built in memory mapped interface to control registers, coefficient registers
Design Flow - Running Simulink Simulation

Creates files in location specified by Control block

- VHDL Code
- Timing constraints file (.sdc)
- DSPB-AB subsystem Quartus® IP file
Design Flow - Documentation Generation

Get accurate resource utilization of all modules right after simulation, without place & route

DSP Builder > Resource Usage
DSP Builder > View Address Map
Design Verification

Run ModelSim block loads the design into the ModelSim simulator

RTL Simulation
Design Flow – System Integration

Add `<subsystem>_hw.tcl` directory to Qsys IP Search Path

Qsys -> Tools -> Options -> IP Search Path

Add subsystem from the Component pick list
ACCELERATION STACK FOR XEON WITH FPGA
Gap: Creating Full-Stack Accelerated Applications on FPGA is Difficult and Time Consuming. Provides standard C API to standardized FPGA interface manager.

Using FPGAs Just Got Easier.

- Orchestration / Rack Management
- SW Application
- Software Frameworks
- Libraries
- Open Programmable Acceleration Engine (OPAE)
- OS Driver

Increase Abstraction
Increase Ease of Use

Application FPGA Accelerator (Loadable Workload)

Accelerator Functions

FPGA Interface Manager (Standard I/O Interfaces)

Intel® FPGA Programmable Accelerator Card (PAC)

Pre-built Accelerator Solutions (ecosystem)

* Other names and brands may be claimed as the property of others.
OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.
ACCELERATION STACK FOR INTEL® XEON® CPU WITH FPGAS
COMPREHENSIVE ARCHITECTURE FOR DATA CENTER DEPLOYMENTS

Rack-Level Solutions
User Applications
Industry Standard Software 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))
OS & Virtualization Environment

Faster Time to Revenue
- Fully validated Intel® board
- Standardized frameworks and high-level compilers
- Partner-developed workload accelerators

Simplified Management
- Supported in VMware vSphere® 6.7 Update 1*
- Rack management and orchestration framework integration

Broad Ecosystem Support
- Upstreaming FPGA drivers to Linux® kernel
- Qualified by industry-leading server OEMs
- Partnering with IP partners, OSVs, ISVs, SIs, and VARs

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

* Demonstrated at VMWorld Las Vegas - August 28-30, 2018
Acceleration Stack Provides FPGA Orchestration in Cloud/Data Center

Public and Private Cloud/Datacenter Users

Launch workload

Software Defined Infrastructure

Orchestration Software (FPGA Enabled)

Place workload

Resource Pool

Storage

Network

Compute

Virtualized

Static/dynamic FPGA programming

Secure

End User Developed IP

Intel Developed IP

3rd party Developed IP

Workload accelerators

IP Store

Intel Proprietary
for LRZ

Workload 1

Workload 2

Workload N

Xeon VM

FPGA
Server Virtualization for the Acceleration Stack with VMware

Out-of-the-box support from VMWare for Intel Arria 10 PAC and Acceleration Stack in upcoming vSphere 6.7 U1

Server virtualization enables customers to deploy FPGA workload acceleration with lower total cost of ownership.
Migrating FPGA-Accelerated Workload with vMotion*

1. Run Application on Bare Metal
2. Implement on ESXi* Hypervisor
3. PVRDMA # connects application to remote Intel® FPGA PAC / FPGA device
4. Use vMotion* to move application from one server to another

Continuous application acceleration during vMotion – industry first demonstration

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

# – Unoptimized, proof-of-concept code. Not part of a shipping product. See supplementary slide for system configuration details.
Components of Acceleration Stack: Overview

- **Application**
- **Libraries**
- **Drivers**
- **PCIe Drivers**
  - Provided by Intel
- **Open Programmable Acceleration Engine (OPAE)**
  - Provided by Intel
- **Intel FPGA**
  - *FPGA Interface Manager* Provided by Intel
  - Qualified and Validated for volume deployment
    - Provided by OEMs
  - User, Intel, or 3rd-Party IP Plugs into AFU Slot
    - (Tuning Expert)
- **Intel Proprietary**

Developed by User (Domain Expert)
User, Intel, and 3rd Party (Tuning Expert)
PAC with Intel® Arria® 10 FPGA

- Low-profile (half-length, half height) PCIe* slot card
- 168 mm × 56 mm
- Maximum component height: 14.47 mm
- PCIe × 16 mechanical

- 128 MB Flash
- For storage of FPGA configuration
- Board Management Controller (BMC)
- Server class monitor system
- Accessed via USB or PCIe

- USB 2.0 port for board firmware update and FIM image recovery

- QSFP+ slot accepts pluggable optical modules

- Powered from PCIe+12V rail
- 70 W total board power
- 45 W FPGA power

- 2 – Banks of DDR4-2133 SDRAM, 4 GB each
- 64 bit data, 8 bit ECC
- Total 8 GB

Intel Proprietary for LRZ
PAC with Intel® Stratix® 10 FPGA

- 4x, length, full height, dual slot PCIe® slot card
- 2x QSFP+ slot accept pluggable optical modules
- Up to 100GbE each
- 128 MB Flash
- For storage of FPGA configuration
- For BMC firmware
- Board Management Controller (BMC)
- Server class monitor system
- Accessed via USB or PCIe
- USB 2.0 port for board firmware update and FIM image recovery
- 4 – Banks of DDR4-2400 SDRAM, 8 GB each
- 64 bit data, 8 bit ECC
- Total 32 GB
- Powered from PCIe+12V rail
- 225 W total board power
- 16x PCIe®
- PCIe Gen3 x16 connectivity to Intel® Xeon® host

Intel® Programmable Acceleration Card with Intel® Stratix® 10 SX FPGA

Intel® Enpirion® Power Solutions

USB Hub

Intel® Enpirion® Power Solutions

Flash

BMC
Intel® MAX® 10 FPGA
 DDR4 w/ECC
 DDR4 w/ECC
 DDR4 w/ECC
 DDR4 w/ECC

QSFP28 4x 25Gb Networking Interface

QSFP28 4x 25Gb Networking Interface
Nearly Transparent Software Application Use Model

Properties Object → Discover / search resource

Token Object → Acquire ownership of resource

Handle Object

Object model

Map AFU registers to user space → Allocate / define shared memory space → Start / stop computation on AFU and wait for result

Deallocate shared memory → Unmap MMIO → Relinquish ownership

Reconfigure AFU

Properties Object

Token Object

Handle Object
**Enumeration and Discovery**

```c
fpga_properties prop;
fpga_token token;
fpga_guid myguid; /* 0xabcdef */
fpgaGetProperties(NULL, &prop);
fpgaPropertiesSetObjectType(prop, FPGA_ACCELERATOR);
fpgaPropertiesSetGUID(prop, myguid);
fpgaEnumerate(&prop, 1, &token, 1, &n);
fpgaDestroyProperties(&prop);
```
Acquire and Release Accelerator Resource

```c
fpga_token token;
// ... enumeration ...

fpga_handle handle;

fpgaOpen(token, &handle, 0);
.
.
.

fpgaClose(handle);
```
Memory-Mapped I/O

FPGA_DEVICE

FPGA_ACCELERATOR

control register
control register
control register

SW application

fpgaMapMMIO(..., &mmio_ptr)

libopae-c

fpgaReadMMIO()
fpgaWriteMMIO()

mmio_ptr

SW application process address space (virtual)

TEXT
DATA
BSS

control register
control register
control register

Intel Proprietary for LRZ
Management and Reconfiguration

FPGA_DEVICE

FPGA_ACCELERATOR
AFU_ID: 0xbe11e5

SW application (with admin privilege)
fpgaReconfigureSlot(..., buf, len, 0)

libopae-c

Partial configuration

Storage

GBS file
xyz.gbs

GBS metadata
interface_id
afu_id
...

Intel Proprietary
for LRZ
Management and Reconfiguration

```c
fpga_handle handle;        /* handle to device */
FILE    *gbs_file;
void    *gbs_ptr;
size_t   gbs_size;

/* Read bitstream file */
gbs_ptr = malloc(gbs_size);
fwrite(gbs_ptr, 1, gbs_len, gbs_file);

/* Program GBS to FPGA */
fpgaReconfigureSlot(handle, 0, gbs_ptr, gbs_size, 0);
/* ... */
```
Where to Get AFU’s for the FPGA Accelerator

**Self-Developed**

- Higher Productivity
  - C/C++ Programming Language
    - Intel® HLS Compiler
    - Intel® FPGA SDK for OpenCL™

- Performance Optimized
  - VHDL or Verilog

**Externally-Sourced**

- Intel® Reference Designs
- Contracted Engagement
  - Ecosystem Partner

Accelerator Functional Unit (AFU)
Growing the Xeon+FPGA Ecosystem

**IP AND SOLUTIONS**
Portfolio of Accelerator Solutions developed by Intel and third-party technologists to expedite application development and deployment

**DEVELOPER COMMUNITY**
Enabling software developers access via:
- Intel Builder programs
- AI Academy
- Intel Developer Zone (IDZ)
- Rocketboards.org

**UNIVERSITIES**
Reaching over 200,000 students per year with FPGA publications, workshops and hands-on research labs
Committed to Open Source vision

**ISV PARTNERS**
Expanding the reach for system vendors with platforms and ready-to-use application workloads.
Growing List of Accelerator Solution Partners

Easing Development and Data Center Deployment of Intel FPGAs For Workload Optimization

- Data Analytics
- Finance
- Genomics
- Media Transcoding
- Cyber Security
- AI
# Intel PAC Top Solutions for Data Center Acceleration

<table>
<thead>
<tr>
<th>Category</th>
<th>Solution</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td></td>
<td>96% latency reduction</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td></td>
<td>½ TCO</td>
</tr>
<tr>
<td>Genomics</td>
<td>GATK</td>
<td>2.5X</td>
</tr>
<tr>
<td>JPEG2Lepton</td>
<td>JPEG2Webp</td>
<td>3-4X</td>
</tr>
<tr>
<td>Big Data</td>
<td>Streaming Analytics</td>
<td>5X</td>
</tr>
<tr>
<td>Financial</td>
<td>Black Scholes</td>
<td>8X</td>
</tr>
<tr>
<td>Network</td>
<td>Security/</td>
<td>3x</td>
</tr>
<tr>
<td></td>
<td>Monitoring</td>
<td></td>
</tr>
</tbody>
</table>

Intel Proprietary

for LRZ
Customer Application: Big Data Applications running on Spark/Kafka Platforms

Current solution: Run Spark/SQL on a cluster of CPUs

Challenge: For many applications in the FinServ/Genomics/Intelligence Agencies/etc. Spark performance does not meet customers SLA requirements, especially for delay sensitive streaming workloads

Solution Value Proposition

Performance - Accelerate Spark SQL/Kafka by 8x
Ease of Use – Zero Code Change
Scalability - Hardware Agnostic
Lower TCO
CASE STUDY: 5X RISK ANALYTICS PERFORMANCE INCREASE

Customer Application: Risk Management acceleration framework (financial back-testing)

Current solution: Deploy a cluster of CPUs or GPUs with complex data access

Challenge: Traditional risk management methods are compute intensive, time consuming applications -> 10+ hours for financial back-testing

Solution Value Proposition

> 5x Performance Improvement
Perform Risk and Pricing Calculations Simultaneously
Abstraction - Integrated Solution
with Apache Spark, SSD Access and FPGA Implementation
Leverage FPGA Developers and Build Your Own

HDL Programming

- **Intel® Quartus Prime Pro**
  - C
  - HDL
  - Syn. PAR
  - AFU Image
  - exe
  - SW Compiler
  - FPGA
  - Application
  - AFU
  - OPAE Software
  - FIM
  - CPU
  - AFU Simulation Environment (ASE)

OpenCL Programming

- **Intel® HLS Compiler**
  - Host
  - Kernels
  - SW Compiler
  - exe
  - FPGA
  - AFU Image
  - OpenCL Compiler
  - Application
  - OPAE Software
  - FIM
  - CPU

Leverage FPGA Developers and Build Your Own

Intel Proprietary for LRZ
AFU Overview Flow

AF Simulation Environment (ASE) enables seamless portability to real HW

- Allows fast verification of OPAE software together with AF 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
FPGA Components of Acceleration Stack

- FPGA Interface Unit (FIU)
- Core Cache Interface (CCI)
- High Speed Serial Interface (HSSI)
- 10Gb/40Gb 100Gb**
- Accelerator Functional Unit (AFU)
- Functional Unit (AFU)
- Local Memory Interfaces
- EMIF
- EMIF
- EMIF**
- DDR4
- DDR4
- DDR4**
- DDR4**

* Could be other interfaces in the future (e.g. UPI)
** Stratix 10 PAC Card
AFU Development Flow Using OPAE SDK

AFU requests the `ccip_std_afu` top level interface classes

- $OPAE_PLATFORM_ROOT/hw/samples/hello_afu/hw/rtl/hello_afu.json

AFU RTL files implementing accelerated function

- $OPAE_PLATFORM_ROOT/hw/samples/hello_afu/hw/rtl/afu.sv

List all source files and platform configuration file

- $OPAE_PLATFORM_ROOT/hw/samples/hello_afu/hw/rtl/filelist.txt

In terminal window, enter these commands:

- cd $OPAE_PLATFORM_ROOT/hw/samples/hello_afu
- afu_sim_setup --source hw/rtl/filelist.txt build_sim
AFU Development Flow Using OPAE SDK

Compile AFU and platform simulation models and start simulation server process

- `cd build_sim`
- `make`
- `make sim`

In 2\textsuperscript{nd} terminal window compile the host application and start the client process

- `Export ASE_WORKDIR= $OPAE_PLATFORM_ROOT/hw/samples/hello_afu/build_sim/work`
- `cd $OPAE_PLATFORM_ROOT/hw/samples/hello_afu/sw`
- `make clean`
- `make USE_ASE=1`
- `./hello_afu`
AFU Simulation Environment (ASE)

Hardware software co-simulation environment for the Intel Xeon FPGA development

Uses simulator Direct Programming Interface (DPI) for HW/SW connectivity

- Not cycle accurate (used for functional correctness)
- Converts SW API to CCI transactions

Provides transactional model for the Core Cache Interface (CCI-P) protocol and memory model for the FPGA-attached local memory

Validates compliance to

- CCI-P protocol specification
- Avalon® Memory Mapped (Avalon-MM) Interface Specification
- Open Programmable Acceleration Engine
Simulation Complete

AFU Simulator Window (server)

Application SW Window (client)
AFU Development Flow Using OPAE SDK

Generate the AF build environment:

- `cd $OPAE_PLATFORM_ROOT/hw/samples/hello_afu`
- `afu_synth_setup --source hw/rtl/filelist.txt build_synth`

Generate the AF

- `cd build_synth`
- `$OPAE_PLATFORM_ROOT/bin/run.sh`
Using the Quartus GUI

Compiling the AFU uses a command line-driven PR compilation flow

- Builds PR region AF as a .gbs file to be loaded into OPAE hardware platform

Can use the Quartus GUI for the following types of work:

- Viewing compilation reports
- Interactive Timing Analysis
- Adding SignalTap instances and nodes
Acceleration Stack Demo

Lab 3
Getting Started with Acceleration

Deployment Flow

1. Buy Server w/ PAC
2. Install Server OS
3. Download & Install Deployment Package of Acceleration Stack

Development Flow

4. Download & Install Developer Package of Acceleration Stack
5. Download HLS or OpenCL (Optional)
6. Download & Install Simulator
7. Write Host Application
8. Create & Simulate Workload

Server OEM (e.g. Dell)
OS Vendor Website (e.g. CentOS, RHEL)
Intel Website
Vendor Website

Intel Proprietary for LRZ
Getting Qualified Hardware is Step 1

Now:
- Dell PowerEdge* R640, R740, R740xd, R840, R940xa

Now:
- PRIMERGY* RX2540 M4

Available soon:
- HPE ProLiant* DL360, DL380

And more coming .....
Programmable Acceleration Cards (PAC)

Intel® Arria® 10 Accelerator Card

Broadest Deployment at Lowest Power

- 40G, PCIe* Gen3 x8
- ½ length, ½ height, single-slot PCIe card
- Lowest power 66W TDP

Intel Stratix® 10 Accelerator Card

Highest Performance and Throughput

- 2x 100G, PCIe Gen3 x16
- ¾ length, full height, dual-slot PCIe card
- Up to 225 W maximum

Intel Proprietary for LRZ
INTEL® FPGA ACCELERATION HUB

A new collection of software, firmware, and tools that allows all developers to leverage the power of Intel® FPGAs.

Intel® portal for all things related to FPGA acceleration

- Acceleration Stack for Intel® Xeon® with FPGAs
- FPGA Acceleration Platforms
- Acceleration Solutions & Ecosystem
- Knowledge Center
- FPGA as a Service
- 01.org *

* 01.org is an open source community site
Follow-On Courses

- Introduction to Cloud Computing
- Introduction to High Performance Computing (HPC)
- Introduction to Apache™ Hadoop
- Introduction to Apache Spark™
- Introduction to Kafka™
- Introduction to Intel® FPGAs for Software Developers
- Introduction to the Acceleration Stack for Intel® Xeon® CPU with FPGA
- Application Development on the Acceleration Stack for Intel® Xeon® CPU with FPGAs
- Building RTL Workloads for the Acceleration Stack for Intel® Xeon® CPU with FPGAs
- OpenCL™ Development with the Acceleration Stack for Intel® Xeon® CPU with FPGA
- Intel FPGA OpenCL Trainings and HLS Trainings
Teaching Resources

University-focused content & curriculum
- Semester-long laboratory exercises for hands-on learning with solutions
- Tutorials and online workshops for self-study on key use cases
- Free library of IP common for student projects
- Example designs and sample projects

Easy-to-use, powerful software tools
- Quartus Prime CAD Environment
- ModelSim
- Intel FPGA Monitor Program for assembly & C development
- Intel® SDK for OpenCL™ Applications
- Intel OpenVINO™ toolkit (Visual Inference & Neural Network Optimization)
Teaching Resources (cont.)

Hardware designed for education
- 4 different FPGA kits with a variety of peripherals to match project needs
- Compact designs with robust shielding to provide longevity
- Reduced academic prices (range: $55-$275)
- Donations available in some circumstances

Support
- Total access to all developer resources
  - Documentation
  - Design examples
  - Support forum
  - Virtual or on-demand trainings
## DE-Series Development Boards

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE10-Standard</td>
<td>Cyclone V FPGA + SoC</td>
<td>$259</td>
</tr>
<tr>
<td>DE1-SOC</td>
<td>Cyclone V FPGA + SoC</td>
<td>$175</td>
</tr>
<tr>
<td>DE10-Nano</td>
<td>Cyclone V FPGA + SoC</td>
<td>$99</td>
</tr>
<tr>
<td>DE10-Lite</td>
<td>Max 10 FPGA</td>
<td>$55</td>
</tr>
</tbody>
</table>

Visit our [website](#) for full specs on these boards
See the full catalog of Intel FPGA boards & kits at [www.terasic.com](http://www.terasic.com)
<table>
<thead>
<tr>
<th>Feature</th>
<th>BEGINNER FPGA DEV KIT</th>
<th>FPGA+SOC ACADEMIC DEV KIT</th>
<th>FULL-FEATURED ACADEMIC DEV KIT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dev Kit</strong></td>
<td>INTEL DE10-LITE</td>
<td>INTEL DE10-NANO</td>
<td>INTEL DE10-STD</td>
</tr>
<tr>
<td>Academic Price</td>
<td>$55</td>
<td>$99</td>
<td>$175</td>
</tr>
<tr>
<td><strong>FPGA</strong></td>
<td>Max® 10</td>
<td>Cyclone® V</td>
<td>Cyclone® V</td>
</tr>
<tr>
<td>Logic Elements</td>
<td>50,000</td>
<td>110,000</td>
<td>85,000</td>
</tr>
<tr>
<td>ARM Cortex-A9 Dual-Core</td>
<td>x</td>
<td>800 MHz</td>
<td>925 MHz</td>
</tr>
<tr>
<td>System-on-Chip (SoC)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>64 MB SDRAM</td>
<td>1 GB DDR3 SDRAM (HPS)</td>
<td>1 GB DDR3 SDRAM (HPS), 64 MB SDRAM (FPGA)</td>
</tr>
<tr>
<td>PLLs</td>
<td>4</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>GPIO Count</td>
<td>500</td>
<td>469</td>
<td>469</td>
</tr>
<tr>
<td>7 Segment Displays</td>
<td>6</td>
<td>x</td>
<td>6</td>
</tr>
<tr>
<td>Switches</td>
<td>10</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>Buttons</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>LEDs</td>
<td>10</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>Clocks</td>
<td>(2x) 50 MHz</td>
<td>(3x) 50 MHz</td>
<td>(4x) 50 MHz</td>
</tr>
<tr>
<td>GPIO Count</td>
<td>40-pin header</td>
<td>(2x) 40-pin header</td>
<td>(2x) 40-pin header</td>
</tr>
<tr>
<td>Video Out</td>
<td>VGA 12-bit DAC</td>
<td>HDMI</td>
<td>VGA 24-bit DAC</td>
</tr>
<tr>
<td>ADC Channels</td>
<td>x</td>
<td>8</td>
<td>8 + programmable voltage range</td>
</tr>
<tr>
<td>Video In</td>
<td>x</td>
<td>x</td>
<td>NTSC, PAL, Multi-format</td>
</tr>
<tr>
<td>Audio In/Out</td>
<td>x</td>
<td>x</td>
<td>Line In/Out, Microphone In (24 bit Audio CODEC)</td>
</tr>
<tr>
<td>Ethernet</td>
<td>x</td>
<td>Gigabit</td>
<td>10/100/1000 Ethernet (x1)</td>
</tr>
<tr>
<td>USB OTG</td>
<td>x</td>
<td>1x USB OTG</td>
<td>2x USB 2.0 (Type A)</td>
</tr>
<tr>
<td>LCD</td>
<td>x</td>
<td></td>
<td>128x64 backlit</td>
</tr>
<tr>
<td>Micro SD Card Support</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PS/2 Mouse/Keyboard Port</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Infrared</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>HSMC Header</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Arduino Header</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
</tbody>
</table>
Undergrad Lab Exercise Suites: Digital Logic

First digital hardware course in EE, CompEng or CS curriculum
Traditionally introduced sophomore year
Offered in VHDL or Verilog

<table>
<thead>
<tr>
<th>Lab 1</th>
<th>Lab 2</th>
<th>Lab 3</th>
<th>Lab 4</th>
<th>Lab 5</th>
<th>Lab 6</th>
<th>Lab 7</th>
<th>Lab 8</th>
<th>Lab 9</th>
<th>Lab 10</th>
<th>Lab 11</th>
<th>Lab 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switches, Lights, and Multiplexers</td>
<td>Numbers and Displays</td>
<td>Latches, Flip-flops, and Registers</td>
<td>Counters</td>
<td>Timers and Real-Time Clock</td>
<td>Adders, Subtractors, and Multipliers</td>
<td>Finite State Machines</td>
<td>Memory Blocks</td>
<td>A Simple Processor</td>
<td>An Enhanced Processor</td>
<td>Implementing Algorithms in Hardware</td>
<td>Basic Digital Signal Processing</td>
</tr>
</tbody>
</table>

Intel Proprietary for LRZ
Undergrad Lab Exercise Suites: Comp Organization

- Typically second hardware course in EE, CompEng or CS curriculum
- Introduction to microprocessors & assembly language program
- Use ARM processor (on SOC kits) or NIOS II soft processor
- Intel FPGA Monitor Program for compiling & debugging assembly & C code

<table>
<thead>
<tr>
<th>Lab 1 - Using an ARM Cortex-A9 System or NIOS II System</th>
<th>Lab 5 - Using Interrupts with Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lab 2 - Using Logic Instructions with the ARM Processor</td>
<td>Lab 6 - Using C code with the ARM Processor</td>
</tr>
<tr>
<td>Lab 3 - Subroutines and Stacks</td>
<td>Lab 7 - Using Interrupts with C code</td>
</tr>
<tr>
<td>Lab 4 - Input/Output in an Embedded System</td>
<td>Lab 8 - Introduction to Graphics and Animation</td>
</tr>
</tbody>
</table>
Intel FPGA MONITOR PROGRAM

Design environment used to compile, assemble, download & debug programs for ARM* Cortex* A9 processor in Intel’s Cyclone® V SoC FPGA devices

▪ Compile programs, specified in assembly language or C, and download the resulting machine code into the hardware system
▪ Display the machine code stored in memory
▪ Run the ARM processor, either continuously or by single-stepping instructions
▪ Modify the contents of processor registers
▪ Modify the contents of memory, as well as memory-mapped registers in I/O devices
▪ Set breakpoints that stop the execution of a program at a specified address, or when certain conditions are met

Clean and simple UX

Tutorials at fpgauniversity.intel.com

Download independently or as part of University Program Installer (always free!)
Undergrad Lab Exercise Suites: Embedded Systems

Typically third hardware course in EE, CompEng or CS curriculum

Combines hardware and software

Introduction to embedded Linux

| Lab 1 - Getting Started with Linux | Lab 5 - Using ASCII Graphics for Animation |
| Lab 2 - Developing Linux Programs that Communicate with the FPGA | Lab 6 - Introduction to Graphics and Animation |
| Lab 3 - Character Device Drivers | Lab 7 - Using the ADXL345 Accelerometer |
| Lab 4 - Using Character Device Drivers | Lab 8 - Audio and an Introduction to Multithreaded Applications |
Lab Exercise Suites: Machine Learning Basics

Machine Learning on FPGAs

Senior or grad-level course in EE, CompEng, CS or data science curriculum

Teaches how to use the Intel® SDK for OpenCL™ Applications with FPGAs

Basic understanding of AI fundamentals recommended*

<table>
<thead>
<tr>
<th>Lab 1 – Introduction to OpenCL</th>
<th>Lab 5 – Neural Networks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lab 2 – Image Processing</td>
<td>Lab 6 – Using the Deep Learning Accelerator Library</td>
</tr>
<tr>
<td>Lab 3 – Lane Detection for Autonomous Driving</td>
<td>Lab 7 – Integration OpenCL Accelerators into Existing Software</td>
</tr>
<tr>
<td>Lab 4 – Linear Classifier for Handwritten Digits</td>
<td></td>
</tr>
</tbody>
</table>

*For foundational AI & Machine Learning curriculums, visit our partner program Intel AI Academy
## AI Academy Course Outline

**Runs in Cloud on Arria 10 PAC card**

**Contains Slides, Lab exercises, and recordings for each class**


<table>
<thead>
<tr>
<th>Class 1</th>
<th>Introduction to FPGAs for deep learning inferencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 2</td>
<td>Building a deep learning computer vision application w/ Acceleration</td>
</tr>
<tr>
<td></td>
<td>Lab 1 - Deploy an application on an Intel CPU using DL framework</td>
</tr>
<tr>
<td>Class 3</td>
<td>Introduction to the OpenVINO™ toolkit</td>
</tr>
<tr>
<td></td>
<td>Lab 2 - Deploy an application on an Intel CPU using the OpenVINO toolkit</td>
</tr>
<tr>
<td>Class 4</td>
<td>Introduction to the Deep Learning Accelerator Suite for Intel FPGAs</td>
</tr>
<tr>
<td></td>
<td>Lab 3 - Accelerate the application on an Intel FPGA</td>
</tr>
<tr>
<td>Class 5</td>
<td>Introduction to the Acceleration Stack for Intel Xeon CPU with FPGAs</td>
</tr>
</tbody>
</table>
In-Person Workshops

Throughout the year our technical outreach team visits universities and industry conferences around the world to conduct hands-on workshops that train professors and students on how to use Intel FPGAs for education and research.

Topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intro to FPGAs and Quartus</td>
<td>4 hrs.</td>
</tr>
<tr>
<td>Embedded Design using Nios II</td>
<td>4 hrs.</td>
</tr>
<tr>
<td>High-Speed IO</td>
<td>4 hrs.</td>
</tr>
<tr>
<td>High-level Synthesis</td>
<td>4 hrs.</td>
</tr>
<tr>
<td>Static Timing Analysis of Digital Circuits</td>
<td>4 hrs.</td>
</tr>
<tr>
<td>Machine Learning Acceleration</td>
<td>4 hrs.</td>
</tr>
<tr>
<td>Simulation &amp; Debug</td>
<td>4 hrs.</td>
</tr>
<tr>
<td>Modern Applications of FPGAs</td>
<td>1 hr.</td>
</tr>
<tr>
<td>Embedded Linux</td>
<td>4 hrs.</td>
</tr>
<tr>
<td>How to Get Hired in the Tech Industry</td>
<td>1 hr.</td>
</tr>
</tbody>
</table>

Contact us at FPGAUniversity@intel.com to inquire about scheduling a workshop
Find Materials: FPGAUniversity.INTEL.com

Educational Materials

Our educational materials include tutorials, laboratory exercises, IP cores, example computer systems and software. They are intended for use in courses on digital logic, computer organization, and embedded systems.

Available Materials:

- Tutorials
- Laboratory Exercises
- IP Cores
- Computer Systems
- Software
- External Links
Membership: FPGAUniversity.INTEL.com
Contact the University Team

Rebecca Nevin
Outreach Manager
Intel FPGA University Program
rebecca.l.nevin@intel.com

Larry Landis
Senior Manager
New User Experience Group
lawrence.landis@intel.com
How do GPUs Deal With Fine Grained Data Sharing?

Some GPU techniques involve implicit SIMT synchronization

FPGA threads aren’t warp-locked, so implicit sync doesn’t make sense

- FPGAs do exactly what you ask them to do the way you code it
## An Even Closer Look: CUDA Execution Model

<table>
<thead>
<tr>
<th></th>
<th>FERMI GF100 SM</th>
<th>FERMI GF104 SM</th>
<th>KEPLER GK104 SMX</th>
<th>KEPLER GK110 SMX</th>
<th>MAXWELL GM107 SMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
<td>5.0</td>
</tr>
<tr>
<td>Shared Memory/SM</td>
<td>48KB</td>
<td>48KB</td>
<td>48KB</td>
<td>48KB</td>
<td>64KB</td>
</tr>
<tr>
<td>32-bit Registers/SM</td>
<td>32768</td>
<td>32768</td>
<td>64K</td>
<td>64K</td>
<td>64K</td>
</tr>
<tr>
<td>Max Threads/Thread Block</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Max Thread Blocks/SM</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Max Threads/SM</td>
<td>1536</td>
<td>1536</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Threads/Warp</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Warps/SM</td>
<td>48</td>
<td>48</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Max Registers/Thread</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>255</td>
<td>255</td>
</tr>
</tbody>
</table>
FPGA Execution Model

Single Block of Data

Multiple Blocks of Data, with Multiple Instructions

All execute in parallel
Divergent Control Flow on GPU

Single instruction
- Thread-locked work items running through different branches
- Serialized
- Major performance factor

GPU uses SIMT pipeline to save area on control logic

for (i=0;i<N;i++)
if (x[i]<y[i])
foo() else bar();

CPUs offer branch prediction
Divergent Control Flow: Just Fine for FPGA

FPGA data path already has all operations in silicon

- Speculatively execute

Overlap branch condition computation

Absorb into one block

No longer any control flow

Compress the schedule
Memory Hierarchy

1. Register data: Registers in FPGA fabric
2. Private data: Registers in FPGA fabric
3. Local memory: On-chip RAMs
4. Global memory: Off-chip external memory
External Memory Dynamic Coalescing

For CPU/GPU the cache and memory controller handle

For FPGA, we create dynamic coalescing hardware matched to specific memory characteristics connected to

- Re-order memory accesses at runtime to exploit data locality
- DDR is extremely inefficient at random access
- Access with row bursts whenever possible
On-chip FPGA Memory

“Local” memory uses on-chip block RAM resources
- Very high bandwidth, 8TB/s,
- Random access in 2 cycles
- Limited capacity

The memory system is customized to your application
- Huge value proposition over fixed-architecture accelerators

Banking configuration (number of banks, width), and interconnect all customized for your kernel
- Automatically optimized to eliminate or minimize access contention

Key idea: Let the compiler minimize bank contention
- If your code is optimized for another architecture (e.g. array[tid + 1] to avoid bank collisions), undo the fixed-architecture workarounds
- Can prevent optimal structure from being inferred
FPGA Local Memory

Split memory into logical banks

- An N-bank configuration can handle N-requests per clock cycle as long as each request addresses a different bank
- Manipulate memory addresses so that parallel threads likely to access different banks – reduce collisions
Local Memory Attributes

Annotations added to local memory variables to improve throughput or reduce area

Banking control:
- numbanks
- bankwidth

Port control:
- numreadports/numwriteports
- singlepump/doublepump
numbanks(N) and bankwidth(N) memory attribute

**What does it do?**

Specifies the banking geometry for your local memory system

A bank = single independent memory system

**What is it for?**

Can be used to optimize LSU-to-memory connectivity in an effort to boost performance

Banking should be set up to maximize “stall-free” accesses
numbanks(N) and bankwidth(N) memory attribute

```c
local int lmem[8][4];
#pragma unroll
for(int i = 0; i<4; i+=2)
{
    lmem[i][x] = ...;
}
```

Not stall-free

<table>
<thead>
<tr>
<th></th>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th>0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
<td></td>
</tr>
<tr>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td></td>
</tr>
<tr>
<td>3,0</td>
<td>3,1</td>
<td>3,2</td>
<td>3,3</td>
<td></td>
</tr>
<tr>
<td>4,0</td>
<td>4,1</td>
<td>4,2</td>
<td>4,3</td>
<td></td>
</tr>
<tr>
<td>5,0</td>
<td>5,1</td>
<td>5,2</td>
<td>5,3</td>
<td></td>
</tr>
<tr>
<td>6,0</td>
<td>6,1</td>
<td>6,2</td>
<td>6,3</td>
<td></td>
</tr>
<tr>
<td>7,0</td>
<td>7,1</td>
<td>7,2</td>
<td>7,3</td>
<td></td>
</tr>
</tbody>
</table>

local int lmem[8][4]
numbanks(N) and bankwidth(N) memory attribute

```c
local int __attribute__((numbanks(8), bankwidth(16))) lmem[8][4];

#pragma unroll
for(int i = 0; i<4; i+=2)
{
  lmem[i][x & 0x3] = ...;
}
```

Mask access to tell compiler no out-of-bounds accesses
numreadports/numwriteports and singlepump/doublepump memory attribute

**What does it do?**

num<read/write>ports: specifies the number of read/write ports in the local memory system

<single/double>pump: specifies the pumping of the local memory system (1x/2x clock)

**What is it for?**

Controls the number of memory blocks used to implement the local memory system
numreadports/numwriteports and singlepump/doublepump memory attribute

local int __attribute__((singlepump,
    numreadports(3),
    numwriteports(1)))) lmem[16];

local int __attribute__((doublepump,
    numreadports(3),
    numwriteports(1)))) lmem[16];