Vivado High-Level Synthesis and the SDx tools

Daniele Bagni
DSP / ML Specialist (EMEA Lead)
Agenda

> Introduction
> Vivado High Level Synthesis
> SDx: SDAccel
> SDx: SDSoC
> References
Agenda

> Introduction

> Vivado High Level Synthesis

> SDx: SDAccel

> SDx: SDSoC

> References
Customizable Architectures – The FPGA Advantage

**CPUs and GPUs**
- Fixed instruction set and rigid memory hierarchy
- Developer adapts the program to the architecture

**FPGAs**
- Flexible, fully customizable architecture
- Developer adapts the architecture to the program
FPGAs – The Ultimate Parallel Processing Device

> No predefined instruction set or underlying architecture

> Developer customizes the architecture to his needs
   » Custom datapaths
   » Custom bit-width
   » Custom memory hierarchies

> Excels at all types of parallelism
   » Deeply pipelined (e.g. Video codecs)
   » Bit manipulations (e.g. AES, SHA)
   » Wide datapath (e.g. DNN)
   » Custom memory hierarchy (e.g. Data analytics)

> Adapts to evolving algorithms and workload needs
Custom Architectures – The Key to Acceleration

- Custom datatypes, parallel and pipelined
- User-defined bitwidths

- Custom dataflow pipelines
- Multiple stages executing simultaneously
- Streaming programming model

- Custom memory architectures
- Double-buffers, FIFOs, Shift-registers
FPGAs – High-Performance and Versatility

Speech Recognition

Database Analytics

Pattern Matching
Shreyas G Singapura et al. "FPGA Based Accelerator for Pattern Matching in YARA Framework." CENG 2015. [Link]

Genomic Analysis
Edico Genome. "DRAGEN Genome Pipeline." Last accessed April 6, 2017. [Link]
<table>
<thead>
<tr>
<th>Application</th>
<th>Performance Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Initial Margin Model Calculation</td>
<td>10x - 50x</td>
</tr>
<tr>
<td>Real-Time Risk Dashboard</td>
<td>89x</td>
</tr>
<tr>
<td>High Performance Monte Carlo Option Pricing Simulation</td>
<td>42x - 540x</td>
</tr>
</tbody>
</table>

For more information, visit the respective links:
- High Performance Monte Carlo Option Pricing Simulation: [https://github.com/KitAway/FinancialModels_AmazonF1](https://github.com/KitAway/FinancialModels_AmazonF1)
Alveo Accelerator Cards

Available Now

Available Now

Coming Soon
FPGA Accelerator Cards That Fit Your Performance Needs

> **Alveo U200**
  - 18.6 Peak INT8 TOPs
  - 77GB/s DDR Memory Bandwidth, 64GB (4 DDR)
  - 31TB/s Internal SRAM Bandwidth
  - 892,000 LUTs

> **Alveo U250**
  - 33.3 Peak INT8 TOPs
  - 77GB/s DDR Memory Bandwidth, 64GB (4 DDR)
  - 38TB/s Internal SRAM Bandwidth
  - 1,341,000 LUTs

> **Alveo U280**
  - 24.5 Peak INT8 TOPs
  - 460GB/s HBM2 Memory Bandwidth, 8GB (2 HBM)
  - 38GB/s DDR Memory Bandwidth, 32GB (2 DDR)
  - 30TB/s Internal SRAM Bandwidth
  - 1,079,000 LUTs

> Deploy in the Cloud or On-Premises

© Copyright 2019 Xilinx
Active and Passive Cooling

**Active** Cooling – for desktop PC / Workstations

**Passive** Cooling – for servers/data center
Agenda

> Introduction
> Vivado High Level Synthesis
> SDx: SDAccel
> SDx: SDSoC
> References
Compute Acceleration Options

Note: Software programmable devices often paired with FPGA for connectivity and co-processing
Vivado High-Level Synthesis: from C source to RTL IP

**High-Level Synthesis from C/C++ to RTL**
- C code for control
- Algorithms

```c
case ARP_QUERY: // arpState FSM
{
    queryResult = arpTable.compare(inputIP);
    if (queryResult.valid == 1)
```

**C / RTL Verification**
- C Simulation
- RTL/C Cosimulation
- Automatic RTL Testbench

**Interface / IP Export**
- AXI4-Master and Axi-Stream
- FIFO
- Many other interface (eg valid driven)

- Most of the flow is C/C++ based
- Faster iterations = Higher productivity than RTL flow

VHLS Accelerates C Code to RTL IP Creation
Vivado HLS System IP Integration Flow

C-based IP Creation

- C, C++, SystemC
- Libraries
  - Arbitrary Precision
  - Video
  - Math
  - Linear algebra
  - LogiCore IP
  - FFT, FIR, DDS
  - DSP

VHDL or Verilog

System Integration

- IP Catalog
- Vivado IP Integrator
- Vivado RTL

- IP Export from Vivado HLS
- Correct-by-construction interface handoff
- Designer assistance for interconnect in IPI
- No RTL knowledge required

Vivado HLS Integrates into System Flows

© Copyright 2019 Xilinx
The HLS Productivity Boost…

- More iterations in RTL
- Exploration is difficult
- Coding style might not yield best results

- Fast iterations…
- Easy exploration
- RTL is verified and optimized for device

Quick Algorithmic Convergence
- Change designs via typedef or templates – in seconds
- Fast simulation and synthesis
- Automatic generation of RTL test bench

Optimized RTL Output
- Code is architecture-aware
- Cooperation with RTL group

Interfaces
- Included in generated RTL
-Pragma based
- Defined within Vivado HLS

C to hand-coded quality RTL
- In weeks not months…

Accelerated verification
- Over 100X over RTL

Ideal for algorithmic designs
- Excels at math (floating / fixed point)
- Video, DSP…

Faster C-based Acceleration Verification
C Validation and RTL Verification

> Two steps to verifying the design
  >> Pre-synthesis: C validation
     - Validate the algorithm is correct
  >> Post-synthesis: RTL verification
     - Verify the RTL is correct

> C validation
  >> A HUGE advantage to using HLS
     - Fast, free verification
     - Validating the algorithm is correct before synthesis

> RTL verification
  >> Vivado HLS tool can co-simulate the RTL with the original testbench
Use a Productive Test Bench

> Ideal Productive test bench
  >> Should be self checking
    – RTL verification will re-use the C test bench
  >> If the test bench is self-checking
    – Allows RTL Verification to be run automatically
    – No requirement to re-check the results again
  >> Use a margin when floating-point math operations are used

```c
int main () {
    // Test function
    // Compare results
    int ret = system("diff --brief -w test_data/output.dat test_data/output.golden.dat");
    if (ret != 0) {
        printf("Test failed !!!\n", ret); return 1;
    } else {
        printf("Test passed \n", ret); return 0;
    }
}
```
The test bench should return 0 if the results are correct, else return some other value
Uses HLS Reports, Analysis and Debug

The HLS Report is generated after Synthesis: Performance, Utilization and Interfaces.

The Debug Perspective provides a full C Debugger: Breakpoints, Stepping, Variable values.

The Analysis Perspective shows in what cycle operations are scheduled: Right-click to cross-reference the C code.
Compare Solutions

> Create Multiple Solutions
  >> Create a base line solution
  >> Create a New Solution
    - Add New Directives
  >> Create additional solution
    - Copy the directives which helped
Vivado HLS Optimizations

> **Directives**

>> Instruct Vivado HLS on the design optimization

>> 20+ pragmas -- (*) half are commonly used for most designs

   - ALLOCATION
   - ARRAY_MAP
   - ARRAY_PARTITION *
   - ARRAY_RESHAPE *
   - DATA_PACK *
   - DATAFLOW *
   - DEPENDENCE *
   - EXPRESSION_BALANCE
   - FUNCTION_INSTANTIATE
   - INLINE
   - INTERFACE *
   - LATENCY
   - LOOP_FLATTEN
   - LOOP_MERGE
   - LOOP_TRIPCOUNT *
   - OCCURRENCE
   - PIPELINE *
   - PROTOCOL
   - RESET
   - RESOURCE *
   - STREAM *
   - UNROLL *

> **Configurations**

>> Set design wide defaults

   - Config Array Partition
   - Config Bind
   - Config Compile
   - Config Dataflow
   - Config Interface
   - Config RTL
   - Config Schedule
Simple example design – showing C-synthesis & interface synthesis

> C function arguments become RTL interface ports

```
f(int in[20], int out[20]) {
    int a, b, c, x, y;
    for(int i = 0; i < 20; i++) {
        x = in[i];
        y = a*x + b + c; out[i] = y;
    }
}
```

*Control logic – FSM*

**The State Machine Automatically Adapts to the Design Interface**
Design Decisions

User Driven Decisions...

- Micro-architecture
  - Parallelism / Unrolling / Dataflow (via pragmas)
  - Data type optimization (e.g. floating versus fixed)
- Interfaces
- Coding style
  - Remove throughput bottlenecks
- Performance
  - Latency and Fmax (via constraints)
- Storage architecture
  - Memories (including partitioning, remapping)

HLS Tool Driven Decisions...

- State machine
  - Structure, encoding
- Register allocation
  - Pipeline registers
- Scheduling
  - Memory and Interface I/O
  - Functional operations
- Throughput Optimizations
  - Automatic code transformations
Accelerating Design Productivity with HLx & SDx
*(how Vivado HLS integrates inside the Xilinx tools offering)*

> **Separate platform design from differentiated logic**
  >> Let application designers focus on the differentiated logic
  >> SDAccel and SDSoS (aka SDx) provide base platform

> **Spend less time on the standard connectivity**
  >> IPI: configure & generate a platform on a custom board
  >> SDx makes all the AXI interconnect for you

> **Spend more time on the differentiated logic**
  >> HLS: enabling core technology: C/C++/OpenCL synthesis
  >> HLx: IP design (HLS/SysGen) + connectivity platform (IP Integrator)
  >> SDx: VHLS is the compiler used under the hood
Get Started with the HLS Examples

> **Built-In Examples**
  >> Design Examples
    - Linear Algebra, Fixed-point Square Root, AXI interfaces, etc.
  >> Coding Examples
    - Floating point, C++, Arrays as FIFOs, etc.
  >> Copy them to get Started

> **Documented Examples**
  >> Overview of each is described in the User Guide UG 902
Learn More about Parallel Programming for FPGAs

> Parallel Programming for FPGAs

> Open-source book

> Teaches HW and SW developers how to efficiently program FPGAs using high-level synthesis (HLS)

> Freely available from: http://hlsbook.ucsd.edu
Agenda

- Introduction
- Vivado High Level Synthesis
- SDx: SDAccel
- SDx: SDSoC
- References
Compute Workloads onto FPGAs

SDAccel Design Environment for Easier Development

SDAccel Design Environment for Easier Development
SDAccel is a development environment facilitating the creation of FPGA-accelerated applications
Focus on the application, not on the implementation details

- No need to develop cards, HW interfaces or SW stacks
- Acceleration platforms provides HW and SW APIs to integrate your code
How FPGA Acceleration Works

CPU handles the rest

FPGA handles compute-intensive, deeply pipelined, hardware-accelerated operations
Initializing the Application

> When the application starts, the FPGA device only contains the “Shell”
  >> The Shell will be managing the communications with the host
Initializing the Application

> When the application starts, the FPGA device only contains the “Shell”
  >> The Shell will be managing the communications with the host

> First, the host initializes the runtime

![Diagram showing the relationship between x86 CPU, Memory, Runtime Library, Shell, and Custom Logic Region]
Initializing the Application

> When the application starts, the FPGA device only contains the “Shell”
  >> The Shell will be managing the communications with the host

> First, the host initializes the runtime

> Then programs the device with the desired FPGA binary
Allocating Buffers and Migrating Data to the Device

- Host allocates input and output buffers in the device
  - Buffers are used to transfer data from the CPU to the FPGA and back

![Diagram of data transfer between CPU and FPGA](image_url)
Allocating Buffers and Migrating Data to the Device

- Host allocates input and output buffers in the device
  - Buffers are used to transfer data from the CPU to the FPGA and back

- Host migrates data to be processed by the accelerator to the buffer FPGA
Running the Accelerators

> Host schedules execution of the desired kernel

> The Runtime is responsible for starting the kernel at the right moment

> Kernel reads data from the input buffer, processes it and writes results in the output buffer previously allocated
Running the Accelerators

- Host schedules execution of the desired kernel
- The Runtime is responsible for starting the kernel at the right moment
- Kernel reads data from the input buffer, processes it and writes results in the output buffer previously allocated
- After the kernel finishes processing the data, it notifies the host
Flow Overview

C/C++ with OpenCL API

C++ bindings supported

RTL, C/C++ or OpenCL C

Build Target Selection

Compiler x86: gcc, g++

Host Application Executable (.exe)

Compiler xocc

FPGA Binary (.xclbin)
Flow Overview

SDAccel can be run using **Makefile, GUI**

Recommended Flow

**Phase #1**
- SW Emulation
- HW Emulation

**Phase #2**
- System Run
Eclipse IDE Based Environment

SDx Assistant

All project tasks in one place

A single environment to Manage, Compile, Debug and Run your application (if a board installed locally)
Debugging

> GDB – GUI (IDE) and Command Line  
  >> Host: SW/HW Emulation, System  
  >> Kernel: SW Emulation, HW Emulation (limited)

> printf support in Kernels (OpenCL)  
  >> SW / HW Emulation, System Run

```c
...  
__kernel
void K_VADD(__global int* A, __global int* B, __global int* R)
{
  #ifdef KERNEL_DEBUG
    printf("\n__Kernel-Debug__: Number of Vector Elements: %d\n\n", N);
  #endif
  ...

HOST-Info: .................................................................
HOST-Info: (Step 5) Set Kernel Arguments and Execute Kernel
HOST-Info: .................................................................
HOST-Info: Setting Kernel arguments ...
HOST-Info: Executing Kernel ...

__Kernel-Debug__: Number of Vector Elements: 256

HOST-Info: Kernel Execution Completed
```
Debug Checks in HW Emulation

> Checks
  >> Uninitialized memory read by kernel
  >> Out of Bounds array access
  >> Use enable_oob = true
  >> Control via sdaccel.ini or GUI

Run Configurations ...

// Vector Addition
__kernel
void K_VADD (__global int* a, ... * b, ... *c, ...) {
  for(int i = 0; i < length; i++)
    c[i] = a[i] + b[i+1];
}

Error: Result mismatch

Emulation Console

i CRITICAL_WARNING: [SDx-EM 12] Out of bound access while reading from argument "K_VADD:B" at offset 0x200

CRITICAL_WARNING: [SDx-EM 10] Argument "K_VADD:B" is trying to access un-initialized memory while reading at offset 0x200.

© Copyright 2019 Xilinx
Profiling: Helps to identify performance bottlenecks

**Profile Summary**

- Execution times and invocation counts
- Memory access and data transfer rates
- Stall Information

**Application Timeline**

- Time stamped events for both host and device

---

### Top Data Transfers: Kernels and Global Memory

<table>
<thead>
<tr>
<th>Device</th>
<th>Compute Unit</th>
<th>Number Of Transfers</th>
<th>Average Bytes per Transfer</th>
<th>Transfer Efficiency (%)</th>
<th>Total Data Transfer (MB)</th>
<th>Total Write (MB)</th>
<th>Total Read (MB)</th>
<th>Total Transfer Rate (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xilinx_u1052s_dynamic_5_1-0</td>
<td>KS_1</td>
<td>240</td>
<td>70,769</td>
<td>90.23</td>
<td>0.029</td>
<td>0.084</td>
<td>0.015</td>
<td>2440.1500</td>
</tr>
<tr>
<td>xilinx_u1052s_dynamic_5_1-0</td>
<td>KS_1</td>
<td>256</td>
<td>70,769</td>
<td>89.23</td>
<td>0.023</td>
<td>0.084</td>
<td>0.015</td>
<td>2400.1500</td>
</tr>
<tr>
<td>xilinx_u1052s_dynamic_5_1-0</td>
<td>KSU_kernel_1</td>
<td>132</td>
<td>93,091</td>
<td>82.75</td>
<td>0.012</td>
<td>0.084</td>
<td>0.005</td>
<td>1910.9300</td>
</tr>
<tr>
<td>xilinx_u1052s_dynamic_5_1-0</td>
<td>KSU_kernel_1</td>
<td>8</td>
<td>4694,000</td>
<td>100.00</td>
<td>0.013</td>
<td>0.018</td>
<td>0.015</td>
<td>76506.8000</td>
</tr>
</tbody>
</table>

---

### Top Kernel Execution

<table>
<thead>
<tr>
<th>Kernel Instance Address</th>
<th>Kernel</th>
<th>Context ID</th>
<th>Command Queue ID</th>
<th>Device</th>
<th>Start Time (ms)</th>
<th>Duration (ms)</th>
<th>Global Work Size</th>
<th>Local Work Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80000000</td>
<td>KA</td>
<td>0</td>
<td>0</td>
<td>xilinx_u1052s_dynamic_5_1-0</td>
<td>0.027</td>
<td>0.012</td>
<td>1</td>
<td>1:1</td>
</tr>
<tr>
<td>0x80000000</td>
<td>KD</td>
<td>0</td>
<td>0</td>
<td>xilinx_u1052s_dynamic_5_1-0</td>
<td>0.039</td>
<td>0.011</td>
<td>1</td>
<td>1:1</td>
</tr>
<tr>
<td>0x18000000</td>
<td>KCAK</td>
<td>0</td>
<td>0</td>
<td>xilinx_u1052s_dynamic_5_1-0</td>
<td>0.050</td>
<td>0.010</td>
<td>1</td>
<td>1:1</td>
</tr>
<tr>
<td>0x18000000</td>
<td>KVCAK</td>
<td>0</td>
<td>0</td>
<td>xilinx_u1052s_dynamic_5_1-0</td>
<td>0.029</td>
<td>0.006</td>
<td>1</td>
<td>1:1</td>
</tr>
</tbody>
</table>

---

### Top Memory Writes: Host and Device Global Memory

<table>
<thead>
<tr>
<th>Buffer Address</th>
<th>Context ID</th>
<th>Command Queue ID</th>
<th>Start Time (ms)</th>
<th>Duration (ms)</th>
<th>Buffer Size (kB)</th>
<th>Writing Rate (kB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
<td>16,394</td>
<td>N/A</td>
</tr>
<tr>
<td>0x00000000</td>
<td>0</td>
<td>0</td>
<td>12636,400</td>
<td>N/A</td>
<td>16,394</td>
<td>N/A</td>
</tr>
<tr>
<td>0x00000000</td>
<td>0</td>
<td>0</td>
<td>12636,400</td>
<td>N/A</td>
<td>16,394</td>
<td>N/A</td>
</tr>
</tbody>
</table>
SDAccel Supports and Guides the Optimization Process

I want to achieve...

> Start with the end in mind → conceptualize desired results

> Use visualization and guidance tools → confirm and converge
SDAccel Design Guidance

> Expert system built-in the tool
  >> Analysis of build results and emulation runs
  >> Explicit and actionable hints: How to improve the design

<table>
<thead>
<tr>
<th>Name</th>
<th>Threshold</th>
<th>Actual</th>
<th>Details</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emulation SW (6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Emulation HW (61)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prj_4_Pipes-Default</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Host Data Transfer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resource Usage</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel Data Transfer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDR_BANK_WRITE_TRANSFER_UTIL [1]</td>
<td>&gt; 5.000</td>
<td></td>
<td></td>
<td>DDR bank 1 write utilization was 2.714% on ConEx. Improve kernel memory.</td>
</tr>
<tr>
<td>DDR_BANK_WRITE_TRANSFER_UTIL #1</td>
<td>&gt; 5.000</td>
<td>2.714</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KERNEL_WRITE_TRANSFER_UTIL [2]</td>
<td>&gt; 5.000</td>
<td></td>
<td></td>
<td>Kernel write utilization on port KCalc_1/m. Improve kernel data path and/or memory write efficiency. Click here</td>
</tr>
<tr>
<td>KERNEL_WRITE_TRANSFER_UTIL #1</td>
<td>&gt; 5.000</td>
<td>2.714</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KERNEL_WRITE_TRANSFER_UTIL #2</td>
<td>&gt; 5.000</td>
<td>64.647</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KERNEL_WRITE_TRANSFER_SIZE [2]</td>
<td>&gt; 0.512</td>
<td></td>
<td></td>
<td>Kernel average write size on port KCalc_1/m.</td>
</tr>
<tr>
<td>KERNEL_WRITE_TRANSFER_SIZE #1</td>
<td>&gt; 0.512</td>
<td>4.096</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KERNEL_WRITE_TRANSFER_SIZE #2</td>
<td>&gt; 0.512</td>
<td>1.024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KERNEL_READ_TRANSFER_AMOUNT_MIN [4]</td>
<td>&gt; 0.250</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KERNEL_READ_TRANSFER_AMOUNT_MIN #1</td>
<td>&gt; 0.250</td>
<td>0.000</td>
<td>Total kernel read of 0.000000 MB on xilinx_u2000_g</td>
<td>Ensure compute units read data written by host. Click here</td>
</tr>
<tr>
<td>KERNEL_READ_TRANSFER_AMOUNT_MIN #2</td>
<td>&gt; 0.250</td>
<td>0.250</td>
<td>Total kernel read of 0.016384 MB on xilinx_u2000_g</td>
<td></td>
</tr>
</tbody>
</table>

Resolution
There are a number of ways to increase this bandwidth utilization by improving the data paths of the kernels and/or the efficiency of the data transfers. Possible options include: loop unrolling, pipelining, vectorization, and maximizing memory port widths.

Not recommended:
void add( __global int* a, __global int* b, __global int* c) {
  for (int i=0; i < 256; i++) {
    c[i] = a[i] + b[i];
  }
}

Recommended:
void add( __global int* a, __global int* b, __global int* c) {
  __attribute__((vec Nehalem_pipeline_loop))
  for (int i=0; i < 256; i++) {
    c[i] = a[i] + b[i];
  }
}

SDAccel Profiling and Optimization Guide
Various Reports for Detailed Application Analysis
Flexible Platforms

> Card specific platform scales to necessary connectivity
  >> Increases user logic space
    - Memory controller IP used only if needed
    - Debug IP instantiated via compile flags
  >> Reduces power, build times when not all DDR banks are required
  >> Simplifies Timing closure

> Clocks
  >> Main - 300 MHz
  >> 2\textsuperscript{nd} - up to 500 MHz available for RTL kernels,

> Automatic Frequency scaling supported
Optimization Methodology

SDAccel Environment Profiling and Optimization Guide

UG1207

SDAccel Design Github Repository
https://github.com/Xilinx/SDAccel_Examples

- acceleration
  - basic
  - clk_freq
  - dataflow
  - debug
  - host
  - kernel_opt
  - kernel_to_gmem
  - rtl_kernel

- img
- libs
- security
- utility
- vision

Baselining function and performance
- Run application on processor
- Profile application to identify bottleneck and select functions to be accelerated
- Convert host code to use OpenCL APIs
- Convert target functions to CL or C/C++ kernels
- Run CPU Emulation Verify Function Correctness
- Run Hardware Emulation
- Analyze Kernel Compilation Reports, Profile Summary, Timeline Trace, Device HW Transactions
- Build and Run application on FPGA acceleration card
- Analyze Profile Summary, Analyze Timeline Trace
- Function/Performance baseline

Optimizing Data Movement
- Optimize data movement that maximizes utilization of PCIe link, DDR banks, on-chip memories with only data transfer code
- Run CPU Emulation Verify Function Correctness
- Run Hardware Emulation
- Analyze Kernel Compilation Reports, Profile Summary, Timeline Trace, Device HW Transactions
- Build and Run application on FPGA acceleration card
- Analyze Profile Summary, Analyze Timeline Trace
- Data Movement Optimized

Optimizing Kernel Computation
- Optimize kernels with both data movement and computation code following optimization guide
- Run CPU Emulation Verify Function Correctness
- Run Hardware Emulation
- Analyze Kernel Compilation Reports, Profile Summary, Timeline Trace, Device HW Transactions
- Build and Run application on FPGA acceleration card
- Analyze Profile Summary, Analyze Timeline Trace
- Application Optimized
SDAccel on Github

> Github repository for developers resources

> 80+ examples

- Host code
- Kernels with C++, RTL, OpenCL C
SDAccel Makes FPGA Applications Portable

- Develop once
- Build for different target platforms
- Deploy on-premise or in the cloud

VCU1525, U200, U250
F1 2x.large, 16x.large
VCU1525, U200, U250
FP1
F2, F3
Agenda

> Introduction

> Vivado High Level Synthesis

> SDx: SDAccel

> SDx: SDSoC

> References
What Is SDSoC

- ASSP-like programming experience
- System-level profiling
- Full system optimizing compiler
- Expert use model for platform developers and system architects

"With SDSoC, I was able to complete a full Zynq design in 4 days. Then, I did the same design without SDSoC and took me 3 weeks."

Daniele B, Xilinx DSP Specialist for EMEA
Before the SDSoC Tool: Hardware/Software Partition Exploration
After the SDSoC Tool: Automatic System Generation

C/C++ to running system in hours, days
Agenda

> Introduction
> Vivado High Level Synthesis
> SDx: SDAccel
> SDx: SDSoC
> References
# My two Edge AI Tutorials

## Edge AI Tutorials

<table>
<thead>
<tr>
<th>Tutorial</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR10 Caffe Tutorial (UG1355)</td>
<td>Train, quantize, and prune custom CNNs with the CIFAR10 dataset using Caffe and the Xilinx® DNNDK tools.</td>
</tr>
<tr>
<td>Cats vs Dogs Tutorial (UG1336)</td>
<td>Train, quantize, and prune a modified AlexNet CNN with the Kaggle Cats vs Dogs dataset using Caffe and the Xilinx DNNDK tools.</td>
</tr>
</tbody>
</table>

Copyright © 2019 Xilinx

https://github.com/Xilinx/Edge-AI-Platform-Tutorials
Median Filter and Sorting Network for Video Processing with Vivado HLS

by Daniele Bagni
DSP Specialist
Xilinx, Inc.
daniele.bagni@xilinx.com

Daniele Bagni is a DSP Specialist FAE for Xilinx EMEA in Milan, Italy. After earning a degree in quantum electronics from the Politecnico of Milan, he worked for seven years at Philips Research labs in real-time digital video processing, mainly motion estimation for field-rate upconverters. For the next nine years he was project leader at STMicroelectronics' R&D labs, focusing on video coding algorithm development and optimization for VLIW architecture embedded DSP processors, while simultaneously teaching a course in multimedia information coding as an external professor at the State University of Milan. In 2006 he joined the Xilinx Milan sales office. What Daniele enjoys most about his job is providing customers with feasibility studies and facing a large variety of DSP applications and problems. In his spare time, he likes playing tennis and jogging.
Using the SDSoC IDE for System-level HW-SW Optimization on the Zynq SoC

by Daniele Bagni
DSP Specialist FAE
Xilinx, Inc.
danieleb@xilinx.com

Nick Ni
Product Manager, SDSoC
Development Environment
Xilinx, Inc.
nickn@xilinx.com
Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP

Author: Daniele Bagni, Duncan Mackay

Summary

This application note describes how to quickly implement and optimize floating-point Proportional-Integral-Derivative (PID) control algorithms specified in C/C++ code into an RTL design using Vivado HLS. You can use System Generator for DSP to easily analyze and verify the design. This enables floating-point algorithm designers to take advantage of high-performance, low cost, and power efficient Xilinx® FPGA devices.

Introduction

Floating-point algorithms are widely used in industries from analysis to control applications. Traditionally, such algorithms have been implemented on microprocessors. The primary reason for using microprocessors has been the ease with which floating-point algorithms can be captured, validated, and debugged in C/C++ code, therefore avoiding the complexity and skills required to implement them in hardware. However, implementing these algorithms on optimized and dedicated hardware provides lower cost, higher performance, and power benefits over a standard, or even optimized microprocessor.

This application note presents a new design flow enabled by the Xilinx Vivado™ Design Suite, which allows floating-point algorithms to be quickly specified in C/C++ code, optimized for performance, and implemented on Xilinx FPGA devices. This flow delivers on the cost,
Summary

This application note describes how to use Vivado® High Level Synthesis (HLS) to develop a floating-point matrix multiplication accelerator connected via an AXI4-Stream interface to the Accelerator Coherency Port (ACP) of the ARM CPU in the Zynq®-7000 All Programmable SoC (AP SoC) device.

The floating-point matrix multiplication accelerator modeled in C/C++ code can be quickly implemented and optimized into a Register Transfer Level (RTL) design using Vivado HLS. The solution is then exported as an IP core connected with an automatically-created AXI4-Stream interface to the ACP on AP SoC Processing Subsystem (PS). The connection is made through a Direct Memory Access (DMA) core in the AP SoC Programmable Logic (PL) subsystem. Vivado IP Integrator (IPI) is used to design the AP SoC PL hardware, including the matrix multiplier peripheral, the DMA engine, and an AXI timer. The Software Development Kit (SDK) is used to design the AP SoC PS software to manage the peripherals.
Summary

The Lucas-Kanade (LK) algorithm for dense optical flow estimation is a widely known and adopted technique for object detection and tracking in image processing applications. This algorithm is computationally intensive and its implementation in an FPGA is challenging from both a design and a performance perspective. This application note describes how to implement the LK algorithm with the Xilinx Vivado® High-level Synthesis (HLS) tool to achieve real-time performance in the Zynq®-7000 All Programmable (AP) SoC without image quality degradation.

A real-time demonstration on a Zynq-7000 AP SoC reference board was built with the SDSoc™ development environment's integrated tool. The design reads video data from a file and writes back the processed data to a file, instead of reading and writing frame buffers. The design was created in less than eight weeks by an engineer. This application note also serves as a tutorial demonstrating good C/C++ coding techniques for obtaining the best performance from Vivado HLS in image processing. Download the reference design files for this application note from the Xilinx website. For detailed information about the design files, see Reference Design.
My HLS XAPPs/papers

> Using the SDSoC IDE for System-level HW-SW optimization on the Zynq SoC

> Zynq-7000 All Programmable SoC Accelerator for Floating-Point Matrix Multiplication using Vivado HL

> Demystifying the Lucas-Kanade Optical Flow Algorithm with Vivado HLS

> Using Vivado HLS to Design a Median Filter and Sorting Network for video

> Vivado HLS Eases Design of Floating-Point PID Controller
My System Generator for DSP papers

> Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP

> Interpolated Lookup Tables: Simple Way to Implement a DSP Function

> Building Automotive Driver Assistance System Algorithms with Xilinx FPGA Platforms

> A real-time versatile roadway path extraction and tracking on an FPGA platform
My Papers on Digital Signal Processing

- Block Matching for Automotive Applications on Spartan-3A DSP Devices

- Implementing Downsampling FIR Filters in Xilinx FPGAs
Welcome to the online home of The Zynq Book

The Zynq Book is all about the Xilinx Zynq™-7000 All Programmable System on Chip (SoC) from Xilinx. This is the online home of The Zynq Book, designed to raise awareness of the book and host the accompanying tutorials. Thanks for finding us!

The Zynq Book is the first book about Zynq to be written in the English language. It has been produced by a team of authors from the University of Strathclyde, Glasgow, UK, with the support of Xilinx. We wanted to create an accessible, readable book that would benefit people just starting out with Zynq, and engineers already working with Zynq. We hope that it will prove a handy reference that remains on your desktop! You can find out more about the book’s contents on the About page.

Buy a Hardcopy of The Zynq Book

The book comprises 24 themed chapters, and is printed in full colour throughout (including over 150 figures). It is available internationally for a suggested price of $30 (US), £21.50 (UK), €23 (Europe), and similar prices in other countries. The Buy page provides more details about how to obtain your copy.

PDF copy The Zynq Book

You can download a free pdf copy of the Zynq Book from the Downloads area.

We hope you will enjoy The Zynq Book! If you would like to get in touch with us with questions, feedback etc., then please use the Contact form – we look forward to hearing from you!

Louise, Ross, Martin and Bob
July 2014

© Copyright 2019 Xilinx
Adaptable.
Intelligent.
Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx’s limited warranty, please refer to Xilinx’s Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx’s Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.

Automotive Applications Disclaimer

XILINX PRODUCTS ARE NOT DESIGNED OR INTENDED TO BE FAIL-SAFE, OR FOR USE IN ANY APPLICATION REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS APPLICATIONS RELATED TO: (I) THE DEPLOYMENT OF AIRBAGS, (II) CONTROL OF A VEHICLE, UNLESS THERE IS A FAIL-SAFE OR REDUNDANCY FEATURE (WHICH DOES NOT INCLUDE USE OF SOFTWARE IN THE XILINX DEVICE TO IMPLEMENT THE REDUNDANCY) AND A WARNING SIGNAL UPON FAILURE TO THE OPERATOR, OR (III) USES THAT COULD LEAD TO DEATH OR PERSONAL INJURY. CUSTOMER ASSUMES THE SOLE RISK AND LIABILITY OF ANY USE OF XILINX PRODUCTS IN SUCH APPLICATIONS.

© Copyright 2013–2016 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.