Building Zynq Accelerators with Vivado High Level Synthesis

Stephen Neuendorffer and Fernando Martinez-Vallina

FPGA 2013 Tutorial - Feb 11, 2013
Schedule

- Motivation for Zynq and HLS (5 min)
- Zynq Overview (45 min)
- HLS training (the condensed version) (1.5 hours)
- Zynq Systems with HLS (45 min)
Motivation

- ASICs *are* being displaced by programmable platforms
  - Packaging, verification costs dominate
  - FPGA/ASSP process advantage over commodity ASIC process
  - Full-/semi-custom design vs. standard cell ASIC

- Lots of competing programmable platforms
  - CPU+GPGPU
  - CPU+DSP+hard accelerators (e.g. OMAP, Davinci, etc.)
  - Multicore
  - FPGAs

- From FPGAs to “All Programmable Devices”
  - ‘Small' devices are very capable with increasing integration
  - 'Big‘ devices are getting REALLY big.
Xilinx Technology Evolution

Programmable Logic Devices
Enables Programmable “Logic”

All Programmable Devices
Enables Programmable “Systems Integration”
Zynq-7000 Family Highlights

Complete ARM®-based Processing System
- Dual ARM Cortex™-A9 MPCore™, processor centric
- Integrated memory controllers & peripherals
- Fully autonomous to the Programmable Logic

Tightly Integrated Programmable Logic
- Used to extend Processing System
- High performance ARM AXI interfaces
- Scalable density and performance

Flexible Array of I/O
- Wide range of external multi-standard I/O
- High performance integrated serial transceivers
- Analog-to-Digital Converter inputs
# Zynq-7000 AP SoC Applications Mapping

<table>
<thead>
<tr>
<th>CLUSTER</th>
<th>MARKET</th>
<th>KEY APPLICATIONS</th>
<th>Z-7010</th>
<th>Z-7020</th>
<th>Z-7030</th>
<th>Z-7045</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Auto</td>
<td>Driver Assistance, Driver Info, Infotainment</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Consumer</td>
<td>Business-class Multi-function Printers</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>ISM</td>
<td>IP &amp; Smart Cameras</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Medical Diagnostics, Monitoring and Therapy</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Medical Imaging</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Broadcast</td>
<td>Prosumer / Studio Cameras, Transcoders</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>A&amp;D</td>
<td>Video / Night Vision Equipment</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A&amp;D</td>
<td>Milcomms, Cockpit &amp; Instrumentation</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Comms</td>
<td>Wireless</td>
<td>LTE Radio, Baseband, Enterprise Femto</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Wired</td>
<td>Routers, Switches, Multiplexers, Edge Cards</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>ISM</td>
<td>Motor Control and Programmable Logic Controller (PLC)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>A&amp;D</td>
<td>Missiles, Smart Munitions</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Bridging</td>
<td>Broadcast</td>
<td>EdgeQAMs, Routers, Switchers, Encoders / Decoders</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>ISM</td>
<td>Industrial Networking</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

<10X>
Zynq-7000 Embedded Processing Platform

Processor core complex
- Two ARM® Cortex™-A9 with NEON™ extensions
- Floating Point support
- Up to 1 GHz operation
- L2 Cache – 512KB Unified
- On-Chip Memory of 256KB
- Integrated Memory Controllers
- Run full Linux

State-of-the-art programmable logic
- 28K-235K logic cells
- High bandwidth AMBA interconnect
- ACP port - cache coherency for additional soft processors

How to Leverage the Compute Power of the Fabric?
Systems in FPGA: 3 independent pieces

- Interface IP blocks (HDMI, Memory Controller)
  - Everything within 1 or 2 cycles of IO, "glue logic"
  - Timing accurate
  - Structural RTL + constraints, Spice, IBIS models

- Core IP (microblaze, NOC)
  - Cycle accurate
  - Structural or synthesizable RTL

- Application-specific IP
  - Differentiation/added value
  - High level throughput/latency constraints
  - Synthesizable RTL or Algorithmic spec
High Level Synthesis

Generating Application-Specific IP from Algorithmic C specification

– Focus on Macro-architecture exploration… leave microarchitecture to tool

A few key problems

– Extracting lots of parallelism
  • Statically scheduled Instruction-level parallelism (in loops)
  • Dynamically controlled task-level parallelism (between loops)
– Analyzing pointer aliases
  • Most arrays map into BRAM, rather than global address space
– Understanding performance
  • Good timing models for FPGA synthesis
  • Interval/Latency analysis
All Programmable SOC Approach
Vivado High-Level Synthesis

Accelerates Algorithmic C to Co-Processing Accelerator Integration
Complete ARM-based Processing System

**Processor Core Complex**
- Dual ARM Cortex-A9 MPCore with NEON™ extensions
- Single / Double Precision Floating Point support
- Up to 1 GHz operation

**High BW Memory**
- Internal
  - L1 Cache – 32KB/32KB (per Core)
  - L2 Cache – 512KB Unified
- On-Chip Memory of 256KB
- Integrated Memory Controllers (DDR3, DDR2, LPDDR2, 2xQSPI, NOR, NAND Flash)

**Integrated Memory Mapped Peripherals**
- 2x USB 2.0 (OTG) w/DMA
- 2x Tri-mode Gigabit Ethernet w/DMA
- 2x SD/SDIO w/DMA
- 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 32b GPIO

**AMBA Open Standard Interconnect**
- High bandwidth interconnect between Processing System and Programmable Logic
- ACP port for enhanced hardware acceleration and cache coherency for additional soft processors

Processing System Ready to Program
Powerful Application Processor at Heart

The Application Processor Unit (APU)

- Dual ARM Cortex-A9 MPCore with NEON extensions
  - Up to 1 GHz operation (7030 & 7045)
  - Multi-issue (up to 4), Out-of-order, Speculative
  - Separate 32KB Instruction and Data Caches with Parity

- Snoop Control Unit
  - L1 Cache Snoop Control
    - Snoop filtering monitors cache traffic
    - Accelerator Coherency Port

- Level 2 Cache and Controller
  - Shared 512 KB Cache with parity
  - Lockable

- On-Chip Memory (OCM)
  - Dual-ported 256KB
  - Low-latency CPU access
  - Accessible by DMAs, Programmable Logic, etc.
Processing System External Memories

Built-in Controllers and dedicated DDR Pins

- **DDR controller**
  - DDR3 @ up to DDR1333
  - DDR2 @ up to DDR800
  - LPDDR2 @ up to DDR800
  - 16 bit or 32 bit wide; ECC on 16 bit
  - 73 dedicated DDR pins

- **Non-volatile memory (processor boot and FPGA configuration)**
  - NAND flash Controller (8 or 16 bit w/ ECC)
  - NOR flash/SRAM Controller (8 bit)
  - Quad SPI (QSPI) Controller
**Zynq OS Boot process**

- **Multi-stage boot process**
  - Stage 0: Runs from ROM
    - loads FSBL from boot device to OCM
  - Stage 1 (FSBL): Runs from OCM
    - loads Uboot from boot device to DDRx memory
    - Initiates PS boot and PL configuration
  - Stage 2 (e.g. Uboot): runs from DDR
    - loads Linux kernel, initial ramdisk, and device tree from any location
    - May access FPGA
  - OS boot (e.g. Linux): runs from DDR

- **Supports ‘secure boot’ chain of trust**
Typical Linux Boot from SD card

Typical boot image (contents of BOOT.BIN)

```
the_ROM_image:  
{               
    [bootloader]zynq_fsbl.elf
    system.bit
    u-boot.elf
}
```

Typical SD card contents

```
zyng> ls
devicetree.dtb
BOOT.bin
ramdisk8M.image.gz
uImage
```
Comprehensive set of Built-in Peripherals

*Enabling a wide set of IO functions*

- Two USB 2.0 OTG/Device/Host
- Two Tri-Mode GigE (10/100/1000)
- Two SD/SDIO interfaces
- Two CAN 2.0B, SPI, I2C, UART
- Four GPIO 32bit Blocks
- Multiplexed Input/Output (MIO)
  - Multiplexed output of peripheral and static memories
  - Two I/O Banks: each selectable - 1.8V, 2.5V or 3.3V
- Extended MIO
  - Enables use of Select IO with PS peripherals
  - FPGA must be configured before using EMIO connections
  - EMIO connections use FPGA routing

![Diagram showing connections between various peripherals and MIO]

- Static Memory Controllers
  - 2x SPI
  - 2x I2C
  - 2x CAN
  - 2x UART
  - GPIO
  - 2x SD/SDIO with DMA
  - 2x USB with DMA
  - 2x GigE with DMA

Extended MIO
### Multiplexed I/O (MIO) Pinout

<table>
<thead>
<tr>
<th>IP</th>
<th>MIO</th>
<th>Extendable MIO in Programmable Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>QSPI</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>NOR/SRAM</td>
<td>Yes, Phy off chip</td>
<td>No</td>
</tr>
<tr>
<td>NAND</td>
<td>Yes – 50MHz</td>
<td>Yes – 25MHz</td>
</tr>
<tr>
<td>USB: 0,1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDIO: 0,1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPI: 0,1</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I2C: 0,1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAN: 0,1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GigE: 0,1</td>
<td>RGMII v2.0 (HSTL)</td>
<td>Supports GMII, RGMII v2.0 (HSTL), RGMII v1.3 (LVC莫斯), RMII, MII, SGMII with wrapper in Programmable Logic</td>
</tr>
<tr>
<td></td>
<td>Phy off chip</td>
<td></td>
</tr>
</tbody>
</table>
| UART: 0,1   | Simple UART: Only 2 pins (Tx & Rx) | Full UART (Tx, Rx, DTR, DCD, DSR, RI, RTS & CTS) either require:  
|             |                       | • 2 Processing System pins (Rx & Tx) through MIO + 6 additional Programmable Logic pins  
|             |                       | • 8 Programmable Logic pins          |
Clock Generator Block Diagram
Clocking the PL

Other features:

- associated reset (FCLKRSTn)
- software clock counter
- clock pause trigger (FCLKCLKTRIGxN)
Interrupts

- 16 peripheral interrupts from PL to PS
  - Used for accelerators and peripherals in PL
- 4 processor-specific interrupts from PL to PS
- 28 interrupts from PS peripherals to PL
  - PS peripherals can be serviced from Microblaze in fabric
### AXI is Part of AMBA: Advanced Microcontroller Bus Architecture

<table>
<thead>
<tr>
<th>Interface</th>
<th>Features</th>
<th>SimilarTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemoryMap/Full</td>
<td>Traditional address/data burst (single address,multiple data)</td>
<td>PLBv46,PCI</td>
</tr>
<tr>
<td>Streaming</td>
<td>Dataonly,burst</td>
<td>LocalLink/DSP interfaces/FIFO/FSL</td>
</tr>
<tr>
<td>Lite</td>
<td>Traditional address/data—no burst (single address,multiple data)</td>
<td>PLBv46single OPB</td>
</tr>
</tbody>
</table>

AXI is Part of AMBA: Advanced Microcontroller Bus Architecture.
AXI Interface: Streaming

 AXI Streams are fully handshaked
  - Data is transferred when source asserts VALID and destination asserts READY

 ‘Information’ includes DATA and other side channel signals
  - STRB
  - KEEP
  - LAST
  - ID
  - DEST
  - USER

 Most of these are optional
Memory mapped interfaces consist of 5 streams

- Read Address
- Read Data
- Write Address
- Write Data
- Write Acknowledge

Burst length limited to 256

Data width limited to 256 bits for Xilinx IP

AXI Lite is a subset

- no bursts
- 32 bit data width only
AXI Interconnect IP in PS

- Uses AXI4 Memory Mapped Interfaces
  - Automatic width conversion
  - Automatic AXI3/AXI4 Lite protocol conversion
  - Automatic clock-domain crossing

- Configurable sparse crossbar or shared bus
- Optional buffering fifos
- Optional timing isolation registers
Centralized arbitration with parallel data
– Arbitration optimized for 3+ data beats per burst

Buffering allows address pipelining
– However, Masters and Slaves have practical limits on pipelining
– Described using Master ISSUING and Slave ACCEPTANCE parameters
– Arbitration uses these parameters to limit head-of-line blocking
AXI based accelerators

HLS accelerators will combine lots of AXI interfaces
Zynq AXI Interfaces

**HP**
- 4 x 64 bit Slave interfaces
  - Optimized for high bandwidth access from PL to external memory

**GP**
- 2 x 32 bit Slave interfaces
  - Optimized for access from PL to PS peripherals
- 2 x 32 bit Master interfaces
  - Optimized for access from processors to PL registers

**ACP**
- 1 x 64 bit Slave interface
  - Optimized for access from PL to processor caches
GP Port Summary

- GP ports are designed for maximum flexibility
- Allow register access from PS to PL or PL to PS

- Good for Synchronization
- Prefer ACP or HP port for data transport
HP Port Summary

- HP ports are designed for maximum bandwidth access to external memory and OCM
- When combined can saturate external memory and OCM bandwidth
  - HP Ports: $4 \times 64 \text{ bits} \times 150 \text{ MHz} \times 2 = 9.6 \text{ GByte/sec}$
  - external DDR: $1 \times 32 \text{ bits} \times 1066 \text{ MHz} \times 2 = 4.3 \text{ GByte/sec}$
  - OCM: $64 \text{ bits} \times 222 \text{ MHz} \times 2 = 3.5 \text{ GByte/sec}$
- Optimized for large burst lengths and many outstanding transactions
- Large data buffers to amortize access latency
- Efficient upsizing/downsizing for 32 bit accesses
ACP Port Summary

ACP allows limited support for Hardware Coherency
- Allows a PL accelerator to access cache of the Cortex-A9 processors
- PL has access to through the same path as CPUs
  - including caches, OCM, DDR, and peripherals
- Access is low latency (assuming data is in processor cache)
  - no switches in path

ACP does not allow full coherency
- PL is not notified of changes in processor caches (different from ACE)
- Use “event bus” or register write of PL register for synchronization

ACP is compromise between bandwidth and latency
- Optimized for cache line length transfers
- Low latency for L1/L2 hits
- Minimal buffering to hide external memory latency
- One shared 64 bit interface, limit of 8 masters
ACP Details

- Must access complete cache lines (32 bytes)
  - LENGTH = 3 (i.e. 4 data beats)
  - SIZE = 3 (i.e. transfer 8 bytes per data beat)
  - STRB = 0xFF (i.e. all data will be read/written)
  - Proper address alignment
    - Incremental burst with 32 byte alignment
    - Wrapped burst with 8 byte alignment

- USER[0] = 1 and CACHE[0] = 1 to hit in cache

Accelerator Architecture (With Bus Slave)

Pro: Simple System Architecture
Con: Limited communication bandwidth

Zynq Processing System
Common Peripherals
ARM® Dual Cortex-A9 MPCore™ System
Memory Interfaces
HDMI Output
AXI4 interconnect
Acc. 1
Acc. 2
Programmable Logic
Bus Slave Accelerator Communication

Write to Accelerator
– processor writes to uncached memory location

Read from Accelerator
– processor reads from uncached memory location
Architecture (With DMA)

Pro: High Bandwidth Communication
Con: Complicated System Architecture, High Latency
AXI DMA-based Accelerator Communication

**Write to Accelerator**
- processor allocates buffer
- processor allocates scatter-gather list
- processor initializes scatter-gather list with physically continuous segments
- processor writes data into buffer
- processor flushes cache for buffer
- processor pushes scatter-gather list to DMA register

**Read from Accelerator**
- processor allocates buffer
- processor allocates scatter-gather list
- processor initializes scatter-gather list with physically continuous segments
- processor pushes scatter-gather list to DMA register
- processor waits for DMA complete
- processor invalidates cache for buffer
- processor reads data from buffer
Architecture (With Coherent DMA)

Pro: Low latency, high-bandwidth communication
Con: Complicated system architecture, Limited to data that fits in caches
Coherent AXI DMA-based Accelerator Communication

Write to Accelerator
- processor allocates buffer
- processor allocates scatter-gather list
- processor initializes scatter-gather list with physically continuous segments
- processor writes data into buffer
- processor flushes cache for buffer
- processor pushes scatter-gather list to DMA register

Read from Accelerator
- processor allocates buffer
- processor allocates scatter-gather list
- processor initializes scatter-gather list with physically continuous segments
- processor pushes scatter-gather list to DMA register
- processor waits for DMA complete
- processor invalidates cache for buffer
- processor reads data from buffer
How Does HLS Work?
How Does HLS Work?

- Overview of HLS
- HLS Coding and Design Capture
- Default Behaviors
- Performance Optimization
- Area Optimization
- Interface Definition
Overview of HLS
HLS Premise:
1 C Code – Multiple HW Implementations

One body of code:
Many hardware outcomes

The same hardware is used for each iteration of the loop:
- Small area
- Long latency
- Low throughput

Before going into details, let’s look under the hood ....

Different hardware is used for each iteration of the loop:
- Higher area
- Short latency
- Better throughput

Different iterations are executed concurrently:
- Higher area
- Short latency
- Best throughput

Default Design
Unrolled Loop Design
Pipelined Design
Attributes of a Program for Synthesis

functions
- Functions define hierarchy and control regions

function parameters:
- Define the RTL I/O Ports

Types:
- Data types define bitwidth requirements
- HLS optimizes bitwidth except for function parameters

Loops:
- Define iterative execution regions that can share HW resources

Arrays:
- Main way of defining memory and data storage

Operators:
- Implementations optimized for performance
- Automatically shared where possible to reduce area
Control Defined by a Program

```c
void fir(
    data_t *y,
    coef_t c[4],
    data_t x
 ){
static data_t shift_reg[4];
acc_t acc;
int i;
acc=0;
loop: for (i=3;i>=0;i--){
    if (i==0) {
        acc+=x*c[0];
        shift_reg[0]=x;
    } else {
        shift_reg[i]=shift_reg[i-1];
        acc+=shift_reg[i]*c[i];
    }
}  
*y=acc;
}
```

From any C code example ..

The loops in the C code correlated to states of behavior

This behavior is extracted into a hardware state machine

Finite State Machine (FSM) states

Function Start

For-Loop Start

For-Loop End

Function End
Combining Control and Operations

The diagram illustrates the process of combining control and operations in a system. It shows a C code example, the operations extracted from the code, the control behavior represented as a finite state machine (FSM) states diagram, and the control and datapath behavior, including a unified control dataflow.

From any C code example...

Operations are extracted...

The control is known...

A unified control dataflow behavior is created...
Optimizing and Sizing Program Operations

```c
void fir(
    data_t *y,
    coef_t c[4],
    data_t x
){
    static data_t shift_reg[4];
    acc_t acc;
    int i;
    acc=0;
    loop: for (i=3;i>=0;i--){
        if (i==0) {
            acc+=x*c[0];
        shift_reg[0]=x;
        } else {
            shift_reg[i]=shift_reg[i-1];
        acc+=shift_reg[i]*c[i];
        }
    }
    *y=acc;
}
```

**Code**

- **Operations**
  - RDx
  - RDC
  - >=
  - -
  - ==
  - +
  - *
- **Types**

**Standard C Types**
- long long (64-bit)
- short (16-bit)
- unsigned types
- int (32-bit)
- char (8-bit)
- float (32-bit)
- double (64-bit)

For floats and doubles, there must be an FP core in the library binding can map to; else cannot be synthesized.

**Arbitrary Precision Types**
- C: (u)int types (1-1024)
- C++: (u)int types (1-1024)
- SystemC: sc (u)int types (1-1024)
- sc_fixed types

Can be used to define any variable to be a specific bit width (e.g., 17-bit, 47-bit, etc.)

From any C code example

Operations are extracted...

The C types define the size of the hardware used: handled automatically.

© Copyright 2013 Xilinx
Program Functions and RTL Modules

- Functions are by default converted into RTL modules
- Functions define hierarchy in RTL
- Functions at the same hierarchical level can be shared like any operator to reduce resource consumption
  - Performance requirements control the level possible sharing

Source Code

```c
void A() { ...body A...}
void B() { ...body B...}
void C() {
    B();
}
void D() {
    B();
}
void foo_top() {
    A(...);
    C(...);
    D(...)
}
```

RTL Hierarchy

Each function/block can be shared like any other component (add, sub, etc) provided it is not in use at the same time
Completing a Design – I/O Port Creation

» Function parameters define data I/O ports and default protocols
  – Pointers → AXI4-Master interface
  – Scalars → AXI4-Lite interface or raw wires
  – Arrays → AXI4-Lite or AXI4 stream interface

» Protocol in generated HW are controlled through user directives

```c
void foo_top(int *in1, int *in2, int *out1) {
    *out1 = *in1 + *in2;
}
```

![Diagram of Datapath and Synthesis process](image)
A complete C validation and verification environment
- Vivado HLS supports complete bit-accurate validation of the C model
- Vivado HLS provides a productive C-RTL co-simulation verification solution

Vivado HLS supports C, C++, and SystemC
- Functions can be written in any version of C
- Wide support for coding constructs in all three variants of C

Modeling with bit accuracy
- Supports arbitrary precision types for all input languages
- Allows the exact bit widths to be modeled and synthesized

Floating-point support
- Support for the use of float and double in the code

Pointers and streaming-based applications
Two steps to verifying the design

- Pre-synthesis: C validation
- Post-synthesis: RTL verification

C validation
- Fast and free verification on any Operating System
- Prove algorithm correctness before RTL generation

RTL Verification
- RTL Co-Simulation against the original program testbench
Coding Restrictions

▸ Data Types
  – Forward declared data types
  – Recursive type definitions

▸ Pointers
  – General casting between user defined data types
  – Pointers to dynamically allocated memory regions

▸ System Calls
  – Dynamic memory allocation – must be replaced with static allocation
  – Standard I/O and file I/O – automatically ignored by the compiler
  – System calls
    • i.e. time(), sleep()

▸ Recursive functions that are not compile time bounded

▸ STL lib calls
  – Not supported due to dynamic memory allocation
  – Have compile time unbounded recursion
C and C++ standard types are supported but limit the hardware
- 8-bit, 16-bit, 32-bit boundaries

Real hardware implementations use a wide range of bitwidths
- Tailored to reduce hardware resources
- Minimum precision to keep algorithm correctness

HLS provides bit-accurate types in C and C++
- SystemC and HLS types supported to simulate hardware datapaths in C/C++
Algorithm Modeling with Arbitrary Types

- Code using native C types

```c
int foo_top(int a, int b, int c)
{
    int sum, mult;
    sum=a+b;
    mult=sum^c;
    return mult;
}
```

- Code using HLS types
  - Software model matches hardware implementation
  - C++ types can be compiled with both gcc and Visual Studio

```c
int17 foo_top(int8 a, int8 b, int8 c)
{
    int9 sum;
    int17 mult;
    sum=a+b;
    mult=sum^c;
    return mult;
}
```
Default Behavior
HLS begins by extracting a functional model of a C expression

```c
int a, b, c, x;
int y = a * x + b + c
```
HLS accounts for target frequency and device characteristics to determine minimum required pipelining.

Circuit will close timing but is not yet the optimal implementation.

```c
int a, b, c, x;
int y = a * x + b + c
```
Datapath Synthesis - Optimization

- Automatic expression balancing for latency reduction
- Automatic restructuring to optimize use of FPGA fabric resources

```c
int a, b, c, x;
int y = a * x + b + c
```
Restructuring from previous stage leads to optimized implementations using DSP48

```c
int a, b, c, x;
int y = a * x + b + c
```
After a datapath is generated, loop control logic is added

```c
int a, b, c, x, y;
for(int i = 0; i < 20; i++) {
    x = get(); y = a*x + b + c; send(y);
}
```
C arrays can be implemented as BRAMs or LUT-RAMs

Default implementation depends on the depth and bitsize of the original C array

```c
int a, b, c, x, y;
for(int i = 0; i < 20; i++) {
    x = in[i]; y = a*x + b + c; out[i] = y;
}
```
Completing the Design – Interface Synthesis

- Function parameters become system level interfaces after HLS synthesis

```c
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;
    }
}
```
Performance Optimization

Arrays and Pointers
Arrays and Memory Bottlenecks

- Arrays are the basic construct to express memory to HLS

- Default number of memory ports defined by
  - Number of usages in the algorithm
  - Target throughput

- HLS default memory model assumes 2-port BRAMs

- Arrays can be reshaped and partitioned to remove bottlenecks
  - Changes to array layout do not require changes to the original code

```c
void foo_top (...) {
  ...
  for (i = 2; i < N; i++)
    mem[i] = mem[i-1] + mem[i-2];
}
```

Even with a dual-port RAM, all reads and writes cannot be performed in one cycle.
Array Optimization - Dimensions

Examples: C array and RTL implementation

my_array[10][6][4] → partition dimension 3 →
my_array_0[10][6]
my_array_1[10][6]
my_array_2[10][6]
my_array_3[10][6]

my_array[10][6][4] → partition dimension 1
my_array_4[6][4]
my_array_5[6][4]
my_array_6[6][4]
my_array_7[6][4]
my_array_8[6][4]
my_array_9[6][4]

my_array[10][6][4] → partition dimension 0 → 10x6x4 = 240 individual registers
Array Optimization - Partitioning

Partitioning splits arrays into independent memory banks in RTL.

Arrays can be partitioned on any dimension:
- Multi-dimension arrays can be partitioned multiple times.
- Dimension 0 applies a partitioning command to all dimensions.

Multiple memories allow greater parallel access.

Divided into blocks:
- N-1/factor elements
- One word at a time (like "dealing cards")
- Break a RAM into registers (no "factor" supported)
Array Optimization - Reshaping

- Reshaping combines
  - Array entries into wider bitwidth containers
  - Different arrays into a single physical memory

- New RTL level memories are automatically generated without changes to the C algorithm

```
array1[N]
0 1 ... N-1

block
```

```
N/2 ... N-2 N-1
0 1 ... (N/2-1)
```

```
cyclic
```

```
1 ... N-3 N-1
0 2 ... N-2
```

```
complete
```

```
N-1
N-2
...
1
0
```

Similar to converting a RAM into a very wide register: great access, high throughput

Read modify write is not allowed: read the whole word or write the whole word
Structs can contain any mix of arrays and scalar values

Structs are automatically partitioned into individual elements
- Each struct variable becomes a separate port or data bus
- Independent control logic for each struct member

```c
typedef struct{
    unsigned char A;
    unsigned char B[4];
    unsigned char C;
} my_data;

void foo(my_data *a)
```

* Assumes no array partitioning
Array Optimization – Structs and Data Packing

Data packing creates a single wide bus for all struct members

Bus Structure

– First element in the struct becomes the LSB
– Last element in the struct becomes the MSB
– Arrays are partitioned completely

```c
typedef struct{
    unsigned char A;
    unsigned char B[4];
    unsigned char C;
} my_data;

void foo(my_data *a)
```
Array Optimization - Initialization

Example array initialization

```c
int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};
```

- Implies coeff is initialized at the start of each function call
- Every function call has an overhead in writing the contents of the coeff BRAM

Using static keyword moves initialization to bitstream
- Values of coeff are part of the FPGA configuration bitstream
- No function initialization overhead

```c
static int coeff[8] = {4, -2, 8, -4, 10, 14, 10, -4, 8, -2, 4};
```
**Pointer Optimization – Access Mode**

**Standard mode**
- Each access results in a bus transaction
- Read and write operations can be mapped into a single transaction

**Burst mode**
- Uses the C memcpy method
- Requires a local array inside the HLS block
  - Stores data for the burst write transaction
  - Stores data from the burst read transaction
Pointer Optimization – Multiple Access

```c
void fifo(
    int *d_o,
    int *d_i
) {
    static int acc = 0;
    int cnt;
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
}
```

C compilers will optimize multiple accesses on the same (non-volatile) pointer into a single access: AutoESL matches this behavior.

```c
void fifo(
    volatile int *d_o,
    volatile int *d_i
) {
    static int acc = 0;
    int cnt;
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
}
```
Performance Optimization

Loops
Loops - Classification

- Only perfect and semi-perfect loops are automatically optimized

Perfect loops
- Computation expressed only in the inner most loop
- No initializations between loop statements
- Loop bounds are constant

Semi-perfect loops
- Computation expressed only in the inner most loop
- No initializations between loop statements
- Loop bounds can be variable

Other types of loops
- User needs to convert the loop into perfect or semi-perfect loop
Loop Default Behavior

- Each loop iteration runs in the same HW state
- Each loop iteration runs on the same HW resources

void foo_top (...) {
  ...
  Add: for (i=3;i>=0;i--) {
    b = a[i] + b;
  ...
}

Loops require labels if they are to be referenced by Tcl directives (GUI will auto-add labels)
Loops and Latency

Loops enforce a minimum execution latency

Incrementing the loop counter always consumes 1 clock cycle

Regardless of loop body, example will always take at least 4 clock cycles
Loops – Unrolling to Reduce Latency

```c
void foo_top (...) {
    ...
    Add: for (i=3;i>=0;i--) {
        b = a[i] + b;
    ...
}
```

Select loop “Add” in the directives pane, right-click & Insert Directive then select unroll

Vivado HLS Directive Editor
- Type: Directive
- Destination: Source File
- Directive: UNROLL
- Options:
  - skip exit check: 
  - factor (optional):
  - region:

Options explained on next slide

Unrolled loops are likely to result in more hardware resources and higher area

Unrolled loops allow greater option & exploration

Option 1: 3 2 1 0
Option 2: 3 2 1 0
Option 3: 3 2 1 0
Loops – Partial Unrolling

- HLS can unroll any loop by a factor
- Example shows unrolling by a factor of 2
  - If N is not known, HLS inserts an exit check to maintain algorithm correctness
  - If N is known and fully divisible by the unrolling factor
    - Exit check is removed

Add: for(int i = 0; i < N; i++) {
  a[i] = b[i] + c[i];
}

Add: for(int i = 0; i < N; i += 2) {
  a[i] = b[i] + c[i];
  if (i+1 >= N) break;
  a[i+1] = b[i+1] + c[i+1];
}

Effective code after compiler transformation

for(int i = 0; i < N; i += 2) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
}

An extra adder for N/2 cycles trade-off
Loops - Flattening

- Perfect and semi-perfect loops are automatically flattened
  - Flattening eliminates state transitions between loop hierarchy levels
  - A loop state transition (counter increment) takes 1 clock cycle

- Automatic flattening can be turned off
Loops - Merging

- Loop merging reduces control regions in the generated RTL.
- Does not require code changes as long as:
  - All loops have either constant or variable bounds but not both.
  - Loop body code always generates the same result regardless of how many times it is run.

  - i.e. \( A = B \) is always the same - \( A = A + 1 \) depends on the loop iteration count.

```
void foo_top (...) {
    ...
    L1: for (i=3;i>=0;i--) {
        [loop body l1]
    }
    L2: for (i=3;i>=0;i--) {
        L3: for (j=3;j>=0;j--) {
            [loop body l3]
        }
    }
    L4: for (i=3;i>=0;i--) {
        [loop body l4]
    }
    ...
}
```

```
void foo_top (...) {
    ...
    L123: for (i=16;i>=0;i--) {
        if (cond1) {
            [Loop body l1]
            [loop body l3]
            if (cond4) {
                [Loop body l4]
            }
        }
    }
    ...
}
```
Loops – Merging Example

- Loop merging eliminates redundant computation
  - Reduces latency
  - Reduces resources

Code implemented in RTL by HLS after merging

```
MyRegion:
#pragma AP merge loop
for (i = 0; i < N; ++i)
    A[i] = B[i] + 1;
for (i = 0; i < N; ++i)
    C[i] = A[i] / 2;
}
```

```
for (i = 0; i < N; ++i) {
    A[i] = B[i] + 1;
    C[i] = A[i] / 2;
}
```

Effective code after compiler transformation

```
for (i = 0; i < N; ++i)
    C[i] = (B[i] + 1) / 2;
```

Removes A[i], any address logic, and any potential memory accesses
Loops - Pipelining

Without Pipelining

Loop iterations run sequentially
Throughput = 3 clock cycles
Latency = 3 cycles
Loop Latency = 6 cycles

With Pipelining

Loop iterations run in parallel
Throughput = 1 clock cycle
Latency = 3 cycles per iteration
Loop Latency = 4 cycles for entire loop
Loops - Initiation Interval (II)

- The number of clock cycles between start of new loop body.

- II=1: one loop body per clock cycle
  - a ‘fully pipelined’ datapath for the loop body

- II=2: one loop body every 2 clock cycles
  - Allows for resource sharing of operators.
Loops – Hierarchy and II

- II is expressed by the PIPELINE directive
  - Default value for PIPELINE = 1

- Can be applied to any level of a loop hierarchy
  - Forces unrolling of any loop below the location of PIPELINE directive
  - Increases parallelism and resources in a generated implementation
  - Should be applied at a level that matches the input data rate of the design
Loops – II and Feedback

- Loop feedback is expressed as a dependence between iteration j to iteration j+1
- Type of dependence can limit pipelining
- If a dependence limits pipelining, HLS automatically relaxes the constraint

User requested II = 1
HLS generates II = 2 design due to dependence
Loops – II and Resource Contention

- HLS can instantiate all required resources to satisfy an II target within the boundaries of a generated module.

- External ports can cause resource contention and are not automatically replicated.
  - This type of contention can only be resolved by the user.

- Memory m is a top level port.
  - HLS assumes only 1 port is available to function foo.
  - Multiple read operations push II from 1 to 2.
Loops – Pipeline Behavior

- HLS pipelines by default stall if the next input is not available
  - For a loop, the next iteration doesn’t start if the input data is not ready
  - Stall affects all iterations currently being processed

- Default stall can be avoided with the flush option
  - Flushing the pipeline allows iterations to finish execution regardless of the state of the next iteration
Loops - Dataflow

- Dataflow is the parallel execution of multiple loops within a function
- Loops to run in parallel communicate through arrays

Arrays are changed to FIFOs to allow concurrent execution of Loop_1 and Loop_2

```c
int a[N], b[N], c[N];

Loop_1: for (i=0;i<=N-1;i++) {
    b[i] = a[i] + in1;
}

Loop_2: for (i=0;i<=N-1;i++) {
    c[i] = b[i] * in2;
}
```
Performance Optimization

Functions
Functions

For designs with multiple functions

Default Scheduling

Functions can also be dataflowed like in the case of loops
Area Optimization
Functions and RTL Hierarchy

Source Code

```c
void A() { ..body A.. }
void B() { ..body B.. }
void C() {
    B();
}
void D() {
    B();
}
void foo_top() {
    A(...);
    C(...);
    D(...)
}
```

RTL Hierarchy

foo_top

- A
- B
- C
- D

Functions can be inlined – the hierarchy was removed and the function dissolved into the surrounding function
Function Inlining

No Inlining

Inlining allows optimization to be performed across function hierarchies

Like RTL ungrouping, too much inlining can create a lot of logic and slow runtime
Function Inlining and Allocation

**Easy to Share**

```c
void foo() {
}
void foo_top() {
    foo(...);
    foo(...);
}
```

**Cannot be Shared**

```c
void dummy1() {
    foo();
}
void dummy2() {
    foo();
}
void foo_top() {
    dummy1(...);
    dummy2(...);
}
```

**Controlling Sharing**

```c
set_directive_allocation -limit1 -type function foo_top foo
set_directive_inlin dummy1
set_directive_inlin dummy2
```

- **foo_top**
  - foo

- **foo_top**
  - `dummy1`
  - `dummy2`

- **foo_top**
  - foo

One RTL block is reused for both instances of function `foo`

Function `foo` is not within the immediate scope of `foo_top`

Inlining brings `foo` into function `foo_top` where it can be shared
Array Mapping - Horizontal

- Combine multiple C arrays into 1 deeper memory
- Default is to concatenate arrays one after the other
  - User can introduce an offset to account for a system address map if the combined memories are top level ports

Longer array (horizontal expansion) with more elements

Optionally, apply an offset
Offset of M+1 from the start
Combine multiple C arrays into 1 wider memory

Arrays use the same ordering as structs for packing

– First array represents the LSB bits of the wider memory
Interface Definition
Default Ports

➢ Clock
  – One clock per C/C++ design
  – Multiple clocks possible for SystemC designs

➢ Reset
  – Applies to FSM and variables initialized in the C algorithm

➢ Clock Enable
  – Optional port
  – One clock enable per design
  – Attached to all modules within an HLS generated design
Function Parameters

- **Function parameters**
  - Data ports for RTL I/O

- **Function return**
  - 1 per HLS design
  - Valid at the end of the C function call

- **Pointers**
  - Can be implemented as both input and output
  - Transformed into separate ports for each direction

```c
#include "adders.h"
int adders(int in1, int in2, int *sum) {
    int temp;
    *sum = in1 + in2 + *sum;
    temp = in1 + in2;
    return temp;
}
```
Function Parameters - Arrays

➤ RAM ports
  – Default port for an array
  – Assumes only 1 port connected to the HLS block
  – Automatic generation of address and data ports

➤ FIFO ports
  – Example of streaming I/O
  – Assumes array is accessed in sequential order
By default all HLS generated designs have a master control interface:

- **ap_start**
  - Starts the RTL module, same as starting a function call in C

- **ap_idle**
  - RTL module is idle

- **ap_done**
  - RTL module completed a function call
  - The data in the ap_return port is valid

- **ap_ready (not shown)**
  - Only generated for designs with top level function pipelining
  - Allows a processor to launch a new function call
I/O Data Transfer Protocols

- **Port I/O protocol**
  - Selected by the user to integrate the HLS generated block into a larger design
  - Control the sequencing of data on a per interface basis

- **Allows mapping to AXI and HLS provided protocols**
  - Interface synthesis in C and C++ designs

- **User can define their own interface protocol**
  - SystemC designs natively express all port interfaces
Connecting to Standard Buses

[Diagram showing connections between RTL Design, AXI4 Lite Slave Interface (slv0), AXI4 Lite Slave Interface (slv1), and Exported Design.]
Let’s try it out!

16x16 Matrix Multiply
Basic AutoESL training in one slide

- Pick good places to pipeline.
  - `#pragma HLS pipeline`

- Partition memories if needed.
  - `#pragma HLS ARRAY_PARTITION variable=? complete dim=?`

- Watch for recurrences
  - Might need to rewrite code or pick a different algorithm

- Use reduced-bitwidth operations.
  - `ap_int<>`, `ap_uint<>`, `ap_fixed<>`
Basic Matrix Multiply

```c
void mm(int in_a[A_ROWS][A_COLS],
        int in_b[A_COLS][B_COLS],
        int out_c[A_ROWS][B_COLS])
{
    // matrix multiplication of a A*B matrix
    a_row_loop: for (int i = 0; i < A_ROWS; i++) {
        b_col_loop: for (int j = 0; j < B_COLS; j++) {
            int sum_mult = 0;
            a_col_loop: for (int k = 0; k < A_COLS; k++) {
                sum_mult += in_a[i][k] * in_b[k][j];
            }
            out_c[i][j] = sum_mult;
        }
    }
}
```

DSP48s: 3
Latency: 25121 clocks
void mm_pipelined(int in_a[A_ROWS][A_COLS],
                 int in_b[A_COLS][B_COLS],
                 int out_c[A_ROWS][B_COLS])
{
    int sum_mult;

    // matrix multiplication of a A*B matrix
    a_row_loop: for (int i = 0; i < A_ROWS; i++) {
        b_col_loop: for (int j = 0; j < B_COLS; j++) {
            sum_mult = 0;
            a_col_loop: for (int k = 0; k < A_COLS; k++) {
                #pragma HLS pipeline
                sum_mult += in_a[i][k] * in_b[k][j];
            }
            out_c[i][j] = sum_mult;
        }
    }
}

DSP48s: 3
Latency: 6154 clocks
Loop II: 1
Parallel Dot-Product Matrix Multiply

```c
void mm_parallel_dot_product(int in_a[A_ROWS][A_COLS],
    int in_b[A_COLS][B_COLS],
    int out_c[A_ROWS][B_COLS])
{
    #pragma HLS ARRAY_PARTITION DIM=2 VARIABLE=in_a complete
    #pragma HLS ARRAY_PARTITION DIM=1 VARIABLE=in_b complete
    int sum_mult;

    // matrix multiplication of a A*B matrix
    a_row_loop: for (int i = 0; i < A_ROWS; i++) {
        b_col_loop: for (int j = 0; j < B_COLS; j++) {
            #pragma HLS pipeline
            sum_mult = 0;
            a_col_loop: for (int k = 0; k < A_COLS; k++) {
                sum_mult += in_a[i][k] * in_b[k][j];
            }
            out_c[i][j] = sum_mult;
        }
    }
}
```

DSP48s: 48
Latency: 263 clocks
Loop II: 1
void mm_pipelined_float(float in_a[A_ROWS][A_COLS],
                       float in_b[A_COLS][B_COLS],
                       float out_c[A_ROWS][B_COLS])
{
    float sum_mult;

    // matrix multiplication of a A*B matrix
    a_row_loop: for (int i = 0; i < A_ROWS; i++) {
        b_col_loop: for (int j = 0; j < B_COLS; j++) {
            sum_mult = 0.0;
            a_col_loop: for (int k = 0; k < A_COLS; k++) {
                #pragma HLS pipeline
                sum_mult += in_a[i][k] * in_b[k][j];
            }
            out_c[i][j] = sum_mult;
        }
    }
}

DSP48s: 1
Latency: 18453 clocks
Loop II: 4
void mm_pipelined_float_interchanged(float in_a[A_ROWS][A_COLS],
                        float in_b[A_COLS][B_COLS],
                        float out_c[A_ROWS][B_COLS])
{
    float sum_mult[B_COLS];

    // matrix multiplication of a A*B matrix
    a_row_loop: for (int i = 0; i < A_ROWS; i++) {
        a_col_loop: for (int k = 0; k < A_COLS; k++) {
            b_col_loop: for (int j = 0; j < B_COLS; j++) {
                #pragma HLS pipeline
                float last = (k==0) ? 0.0 : sum_mult[j];
                float result = last + in_a[i][k] * in_b[k][j];
                sum_mult[j] = result;
                if(k == (A_COLS-1)) out_c[i][j] = result;
            }
        }
    }
}

DSP48s: 1
BRAM: 1
Latency: 4105 clocks
Loop II: 1
#include <ap_int.h>

```c
void mm_18_parallel_dot_product(ap_int<18> in_a[A_ROWS][A_COLS],
                                 ap_int<18> in_b[A_COLS][B_COLS],
                                 ap_int<18> out_c[A_ROWS][B_COLS])
{
    #pragma HLS ARRAY_PARTITION DIM=2 VARIABLE=in_a complete
    #pragma HLS ARRAY_PARTITION DIM=1 VARIABLE=in_b complete
    ap_int<18> sum_mult;

    // matrix multiplication of a A*B matrix
    a_row_loop: for (int i = 0; i < A_ROWS; i++) {
        b_col_loop: for (int j = 0; j < B_COLS; j++) {
            #pragma HLS pipeline
            sum_mult = 0;
            a_col_loop: for (int k = 0; k < A_COLS; k++) {
                sum_mult += in_a[i][k] * in_b[k][j];
            }
            out_c[i][j] = sum_mult;
        }
    }
}
```

DSP48s: 16
Latency: 260 clocks
Loop II: 1
Zynq Accelerated Applications
X-Ray Tomography Scanning

\[ \text{sinoSize} = 256 \times \sqrt{2} \]

\[ \text{sinoNum} = \text{angles} = 256 \]

256 x 367

Video
Backprojection Algorithm structure

Partitioning (recursive case 1)

Angular Downsampling (recursive case 2)

Back propagation (base case)
Application characteristics

- Total dataset will not easily fit in blockram or cache
  - $256 \times 367 \times 32\text{bit} = 275\ \text{KByte}$
  - Recursive case 1 is 2x input size
  - Recursive case 2 is same as input size

- Downsampling reduces overall operations
  - Each downsampling stage reduces operations by factor 2.

- Partitioning and Downsampling improves memory locality
  - Output data sets are smaller

- Partitioning and Downsampling partitions data sets
  - Output data sets can be processed in parallel
Backprojection Application

- Open Source Linux-based Application
  - Compiles directly on ARM/Zynq
  - Single-precision floating point

- Lots of things that are not synthesizable
  - Memory allocation
  - File I/O
  - Recursion
Code Structure

```c
bp(sino, size, tau, img)
    if(size < limit) {
        direct(sino, size, tau, img); // Base case
    } else foreach quadrant {
        newSino = allocSino(newSinoSize, newNumSino);
        if(condition) { // With downsampling
            newSinoForNextIter(newSino, sino, newSinoSize);
        } else { // No downsampling
            newSinoForNextIter2(newSino, sino, newSinoSize);
        }
        subImage = getTile(img, quadrant);
        bp(newSino, newSize, tau, subImage); // Recursion
case
    freeSino(newSino);
}
```
Code Structure: HLS limitations

- Pointers must point to statically allocated structures
- Pointers to pointers must be inlined

```c
typedef struct{
    int size;
    myFloat **pixel; // [size][size]
} image;

typedef struct{
    int num;       // number of angles
    int size;      // length of each filtered sinograms
    myFloat T;
    myFloat **sino; // [size][size];
    myFloat *sine;  // [size];
    myFloat *cosine; // [size];
} sinograms;
```
Acceleration Approach

- Two functions with HLS-generated accelerators
  - ap_newSinoForNextIter
    - Decomposes a sinogram into smaller sinogram tiles, with angular downsampling
  - ap_direct
    - Computes result for a tile of the output image from a sinogram tile.

- Most code runs on ARM
  - Memory allocation
  - File I/O
  - Sinogram Decomposition without angular downsampling

- Pipelined Coherent DMA
  - Use good tile processing order for data locality
Code Transformations

- **ap_direct: 1 hour**
  - Introduce statically allocated buffers to resolve pointers.

- **ap_newSinoForNextIter: 4 hours**
  - Introduce statically allocated buffers to resolve pointers.
  - Stripe-based processing of large data set (loop refactoring)

![Diagram showing code transformations](image)
Architecture (With Coherent DMA)

- Zynq Processing System
- ARM® Dual Cortex-A9 MPCore™ System
- Common Peripherals
- Memory Interfaces
- HDMI Output
- Programmable Logic
  - AXI4 interconnect
  - AXI_DMA
  - Acc. 1
  - Acc. 2

© Copyright 2013 Xilinx
Board + video

Left side of the screen: SW running, ~ 1.5 fps

Right side of the screen: SW+ HW running, ~ 10 fps

HMDI Out
Intelligent Vision Applications for FPGAs

- HD Surveillance
- Driver Assistance
- Video Conferencing
- Machine Vision
- A&D UAV
- Office-class MFP
Lane Detection – Algorithm Overview

- Analyze a video frame to detect road lane markings

**Lane Detection**
- RGB to Gray Conversion
- Image Histogram Equalization
- Edge Detection
- Hough Transform

Lane Detection
Application characteristics

- Total dataset will not easily fit in blockram or cache
  - $1920 \times 1080 \times 32$ bit = $\sim 8$ MB

- Predictable access patterns and algorithms with high spatial locality
  - line buffers and frame buffers

- Applications are heterogeneous
  - Pixel processing (good for FPGA)
  - Frame-level processing (good for processor)
Acceleration Approach

- **Pixel processing accelerators with deep dataflow pipelines**
  - Video Function library corresponding to OpenCV functions
  - Extract features from pixels

- **Frame rate processing runs on ARM**
  - UI
  - Feature matching
  - Decision Making

- Pipelined High performance DMA for video

- Features through general purpose interfaces
OpenCV Code

One image input, one image output
– Processed by chain of functions sequentially

```c
...  
IplImage* src=cvLoadImage("test_1080p.bmp");  
IplImage* dst=cvCreateImage(cvGetSize(src),  
    src->depth, src->nChannels);  

cvSobel(src, dst, 1, 0);  
cvSubS(dst, cvScalar(100,100,100), src);  
cvScale(src, dst, 2, 0);  
cvErode(dst, src);  
cvDilate(src, dst);  

cvSaveImage("result_1080p.bmp", dst);  
cvReleaseImage(&src);  
cvReleaseImage(&dst);  
...  
```

*test_opencv.cpp*
Accelerated with Vivado HLS video library

Top level function extracted for HW acceleration

```c
#include "hls_video.h"  // header file of HLS video library
#include "hls_opencv.h"  // header file of OpenCV I/O

// typedef video library core structures
typedef hls::AXI_Base<32>                      AXI_PIXEL;
typedef hls::stream<AXI_PIXEL>                 AXI_STREAM;
typedef hls::Scalar<3, uchar>                  RGB_PIXEL;
typedef hls::Mat<1080, 1920, HLS_8UC3>        RGB_IMAGE;

void top(AXI_STREAM& src_axi, AXI_STREAM& dst_axi, int rows, int cols);
```

top.h

```c
#include "top.h"
...
IplImage* src=cvLoadImage("test_1080p.bmp");
IplImage* dst=cvCreateImage(cvGetSize(src),
    src->depth, src->nChannels);

AXI_STREAM src_axi, dst_axi;
IplImage2AXIvideo(src, src_axi);

top(src_axi, dst_axi, src->height, src->width);

AXIvideo2IplImage(dst_axi, dst);

cvSaveImage("result_1080p.bmp", dst);
cvReleaseImage(&src);
cvReleaseImage(&dst);
```
test.cpp
#include "top.h"
#include "ap_interfaces.h"

void top(AXI_STREAM& src_axi, AXI_STREAM& dst_axi, int rows, int cols){
  //Create AXI streaming interfaces for the core
  #pragma HLS RESOURCE core=AXIS variable=src_axi metadata="-bus_bundle INPUT_STREAM"
  #pragma HLS RESOURCE core=AXIS variable=dst_axi metadata="-bus_bundle OUTPUT_STREAM"

  #pragma HLS RESOURCE core=AXI_SLAVE variable=rows metadata="-bus_bundle CONTROL_BUS"
  #pragma HLS RESOURCE core=AXI_SLAVE variable=cols metadata="-bus_bundle CONTROL_BUS"
  #pragma HLS RESOURCE core=AXI_SLAVE variable=return metadata="-bus_bundle CONTROL_BUS"

  RGB_IMAGE img[6];
  RGB_PIXEL pix(100,100,100);

  #pragma HLS dataflow
  hls::AXIvideo2Mat(src_axi, img[0]);
  hls::Sobel(img[0], img[1], 1, 0);
  hls::SubS(img[1], pix, img[2]);
  hls::Scale(img[2], img[3], 2, 0);
  hls::Erode(img[3], img[4]);
  hls::Dilate(img[4], img[5]);
  hls::Mat2AXIvideo(img[5], dst_axi);
}

Accelerated with Vivado HLS video library

- HW Synthesizable Block for FPGA acceleration
  - Consists of video library function and interfaces
  - Replace OpenCV function with similar function in hls namespace
## 2012.4 Beta: Video Library Function List

<table>
<thead>
<tr>
<th>OpenCV I/O</th>
<th>cvMat2hlsMat</th>
<th>lplImage2hlsMat</th>
<th>CvMat2hlsMat</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCV I/O</td>
<td>hlsMat2cvMat</td>
<td>hlsMat2lplImage</td>
<td>hlsMat2CvMat</td>
</tr>
<tr>
<td>interfaces</td>
<td>hls::AXIvideo2Mat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interfaces</td>
<td>hls::Mat2AXIvideo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Filter2D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Erode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Dilate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::MinS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::MaxS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Mul</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Avg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::AbsDiff</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::CmpS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Cmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::And</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Not</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::AddS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::AddWeighted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Mean</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::SubRS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::SubS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Sum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Reduce</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Reduce</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openCV basic function</td>
<td>hls::Scale</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For function signatures and descriptions, please refer to:
- Synthesizable functions in hls_video.h
- Interface functions in hls_opencv.h
Accelerator Architecture
Cost and power reduction by integrated solution

Performance increase by exploiting the massive compute power of multi-core processors and programmable logic
Digital Pre-Distortion Functionality

- **DPD negates PA non-linearity**
  - PAs consume massive static power
  - DPD improves PA efficiency by ~35-40%

- Increase number of coefficients (K)
  - Better linearization, higher complexity

**Estimate pre-distorter coefficients (A):**

\[
\begin{align*}
Z &= U(y) \\
A &= X \\
\end{align*}
\]

\[
U^HZ = U^HU_A
\]

\[
W = VA
\]

\[
V^{-1}W = A
\]
Application characteristics

- Complex bare metal program
  - Multiple loop nests, with no obvious bottleneck
  - Fixed and floating point
  - Complex numbers

- Use software profiling
  - Focus on VW functionality

- Initial Target: $K = 64$
  - Look for speedup with minimal hardware usage

- More Speedup $\rightarrow$ increase $K$
for (int i = 0; i < NumCoeffs; ++i) {
    #pragma HLS pipeline II=2

    W[i].real +=
        (INT64)u[i].real*tx.real
        + (INT64)u[i].imag*tx.imag;

    W[i].imag +=
        (INT64)u[i].real*tx.imag
        - (INT64)u[i].imag*tx.real;
}
Software Optimization

➤ Use the right algorithm!
  – Gains here are often easier than throwing hardware at the problem

➤ Use ARM NEON function intrinsics
  – Low-level ARM-A9 programming

![Diagram showing multiplication operations and speed-up](image)

<table>
<thead>
<tr>
<th></th>
<th>x86 2GHz</th>
<th>Zynq 800MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>0.66s</td>
<td>1.20s</td>
</tr>
<tr>
<td>Optimized</td>
<td>0.22s</td>
<td>0.54s</td>
</tr>
<tr>
<td>With NEON</td>
<td>n/a</td>
<td>0.25s</td>
</tr>
</tbody>
</table>

Target Update Time: 300ms (faster is better)

Speed-up: 5x
W[i].real += (INT64)uRow[i].real*sample.real + (INT64)uRow[i].imag*sample.imag;

// load operands
int32x2_t tr = vdup_n_s32(sample.real);
int32x2_t ti = vdup_n_s32(sample.imag);
int32x2x2_t u = vld2_s32((int32_t *)(uRow+i));
int64x2_t w = vldlq_s64((int64_t *)(W+i));

// do parallel computation
w = vmlal_s32(w,u.val[0],tr);
w = vmlal_s32(w,u.val[1],ti);

// store result
vst1q_s64((int64_t *)(W+i),w);
Acceleration Approach

- Neon intrinsics are OK, but HLS can do better
  - with minimal code modification

- Pick right partitioning between processor code and accelerator
  - Focus on efficient use of generated hardware
  - Tradeoff overall time and resources
    - time = sw + communication + hw
    - resources = communication + hw

- Efficient memory-mapped IO with fifo
  - DMA resources not justified
AXI FIFO Architecture

<table>
<thead>
<tr>
<th></th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI Infrastructure</td>
<td>~300</td>
<td>~300</td>
</tr>
<tr>
<td>Accelerator</td>
<td>2552</td>
<td>2605</td>
</tr>
</tbody>
</table>
Digital Pre-Distortion on Zynq from C/C++

- Significant speed-up for existing designs
- OR: use speedup to solve bigger problems in same amount of time
Design exploration opportunities

- Maximize resource sharing

- Insert pipelines

- Vary number of coefficients

- Unroll loops to increase performance

```
#pragma HLS allocation
\instances=mul limit=1 \operation
```

```
#pragma HLS pipeline II=1
```

```
CINT64 W[MAX_COEFFS];
```

```
#pragma HLS unroll \factor=UNROLL_FACTOR
```
Accelerator Results

VW Update Time

Not-Optimized Algorithm
Optimized algorithm

VW Accelerator Resources

BRAM
DSP
FF
LUT
Conclusion

➢ Go forth and build!

➢ Many thanks to:
  – Jan Langer
  – Baris Ozgul
  – Juanjo Noguera
  – Kees Vissers
  – Thomas Li
  – Devin Wang
  – Vinay Singh
  – Duncan Mackay
  – Dan Isaacs
  – And many others