2026 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

High-Level Synthesis Meets FPGA Prototyping in the Cloud

Sivasankar Palaniappan, Siemens EDA
Mark Azadpour, AWS

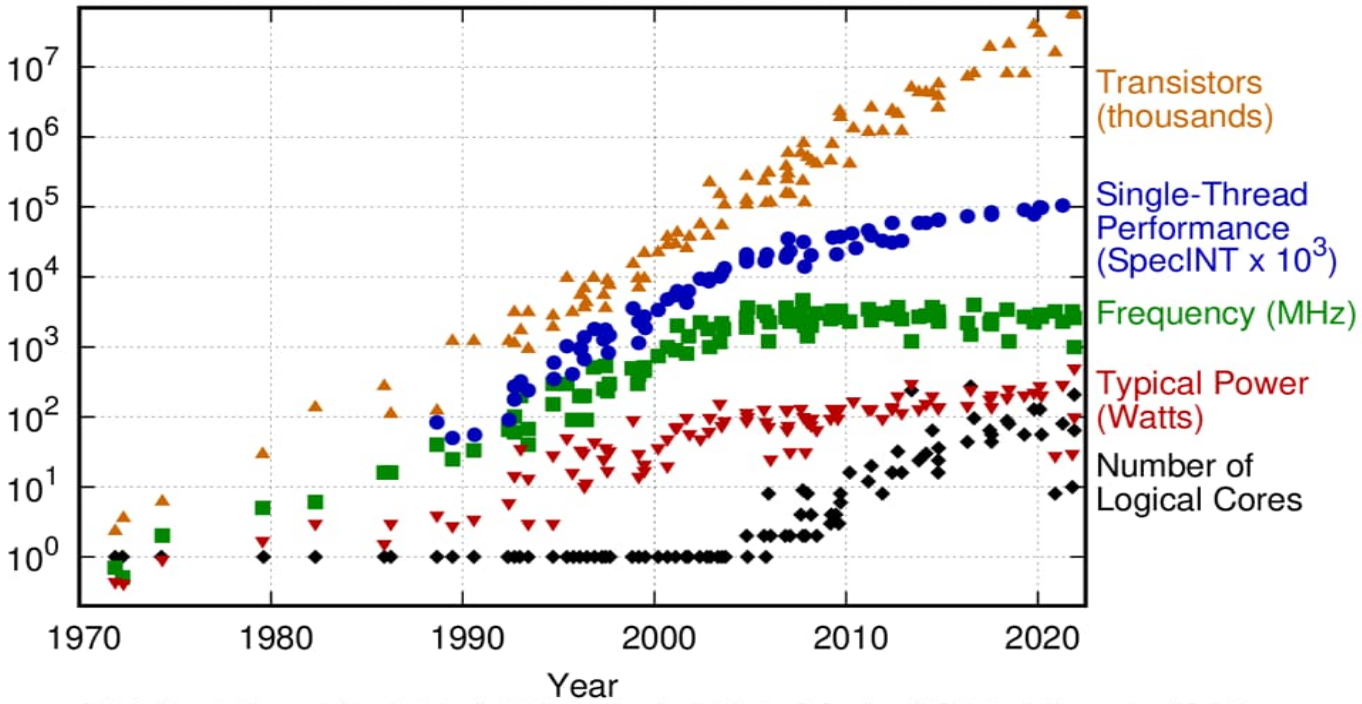
SIEMENS

aws

Agenda

- High-Level Synthesized Accelerators
- HLS Verification Flow
- Accelerating HLS Verification
- AWS F2 FPGA Instances
- Design Example
- Q and A

General Purpose Processors are Limited



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

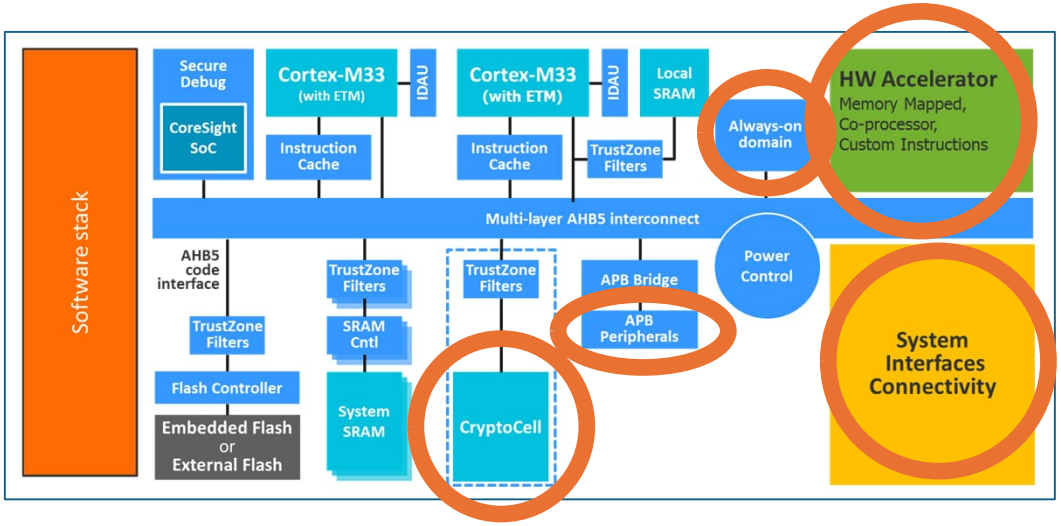
Moore's law continues unabated

But single thread performance for **general-purpose** compute has stalled and will not increase

Adding cores is not effective as software is single threaded

Accelerators are Key to Modern SoCs

- Delivers competitive advantage
  - Higher performance
  - Lower power
  - Use where IP is not sufficient
- For large computational demands
  - DSP, AI, 6G, encryption,...
- Every design needs acceleration
  - ISP - High bandwidth pixel pipes and computer vision
  - WiFi/5G/Optical MODEMs
  - Video CODECs
  - AI/ML - Fully custom silicon at the edge



Building an Accelerator

Traditional Hardware Description Languages

- Manual re-interpretation of algorithm by non-domain experts
- Low productivity, dealing with wires and registers
- No ability to explore design alternatives
- No early PPA metrics

#DIFFICULT

High-Level Synthesis

- Algorithms usually start as C/C++ or a procedural language
- Design faster, explore architectural alternatives
- Early metrics for performance, area, power
- Excellent verification support

#done-early!

HLS Enables Accelerator Development

H.265 & VP9 Video encode/decode

"Micro-architectural exploration simply opens up a design to improvements that would not even be considered in a traditional RTL flow."

Video and Image 3D processing I.P

"Smaller code base to maintain"

"This allows us to explore different options without having to manually make all the code changes"

H.264/VP9/AV1 Video encode/decode

"With our flow (HLS), we were able try numerous architectures and algorithms to find optimal quality-silicon area trade-offs"

Image Sensor Processing

"Power optimizations automatically provide significant reductions"

"Development time and cost is heavily reduced with no compromise in results and design quality"

High-Level Synthesis

```
361  void copy_to_regs(hw_cat_type *dst, index_type dst_offset, raw_memory_line *src, index_type src_offset, index_type size)
362  {
363      // read out of internal memories to an array of registers
364      // *should* make it easy for catapult to pipeline access to internal memories
365
366      index_type count;
367      index_type n;
368
369      static const index_type stride = STRIDE;
370
371      count = 0;
372      while (count < size) {
373          n = stride;
374          if ((size - count) < stride) n = size - count; // mis-aligned at the end of transfer
375          read_line(dst, dst_offset, src, src_offset, n);
376          count += n;
377          src_offset += n;
378          dst_offset += n;
379      }
380  }
```

C/C++ or SystemC

Automated path from C/C++ or SystemC into technology optimized synthesizable RTL

High-Level Synthesis

```
1   module timer (
2         input           clock,
3         input           resetn,
4         input  [1:0]    trans,
5         input  [29:0]   address,
6         input  [3:0]    bl,
7         input           we,
8         input           ce,
9         input  [31:0]   write_data,
10        output [31:0]   read_data,
11        output [1:0]    resp,
12        output          ready
13    );
14
15    reg [31:0]  timer_value;
16    reg [31:0]  rd_reg;
17
18    //reg         ready_out = 1'b1;
19    reg         resp_out  = 2'b00;
20
21    ready_gen #(1) rg (clock, resetn, ce, trans, ready_out);
22
23    assign ready          = ready_out;
24    assign resp           = resp_out;
25    assign read_data      = rd_reg;
26
27    always @(posedge clock or negedge resetn) begin
28      if (resetn == 1'b0) begin
29        timer_value <= 32'h00000000;
30      end else begin
31        timer_value <= timer_value + 32'h00000001;
32      end
33    end
34
35    always @(posedge clock or negedge resetn) begin
36      if (resetn == 1'b0) begin
37        rd_reg <= 32'h00000000;
38      end else begin
39        //if (trans[1] & ce) begin
40        if (ce & !we) begin
41          rd_reg <= timer_value;
42        end
43      end
44    end
45
46  endmodule
```

Synthesizable RTL

Make Informed Design Decisions

These are difficult or impossible to determine in a traditional Verilog design flow

Early Design Metrics

Performance
- HLS schedules design, gives latency and throughput
- C++ model can be connected to TLM, ISS, or other abstract simulations

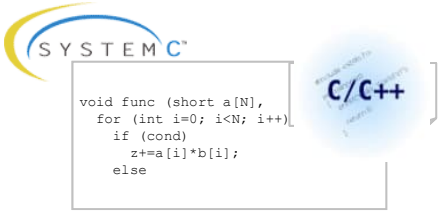
Area
- HLS provides pre-RTL synthesis estimates for area
- Feeds into downstream synthesis tools

Power
- HLS provides pre-RTL synthesis estimates for power
- Has links to downstream power analysis tools

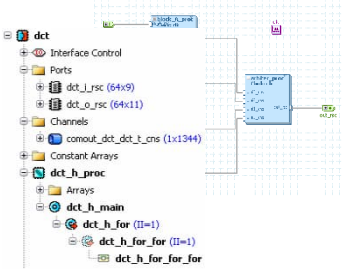
How HLS Works

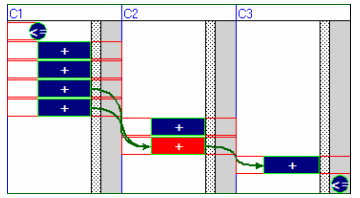
- Parses C/C++ source code and creates control/data flow graph
- Creates a set of state machines and logic to implement the same function
- Determines possible parallelism based on data dependencies
- Creates a synthesizable RTL implementation
  - Based on user directives unroll loops and pipeline computations
  - Based on target technology (ASIC library/FPGA device)
  - Based on target clock frequency

High-Level Synthesis Features



- User architectural control
  - Parallelism, Throughput, Area, Latency (loop unrolling & pipelining)
  - Memories vs Registers (resource allocation)
- Exploration and implementation by applying constraints
  - Not by changing the source code
- Bit accurate data types and operators
- Automatic arithmetic optimizations and bit-width trimming
- Multi-objective process-aware scheduling for FPGA and ASIC
  - Area/latency/blend driven datapath scheduling
  - Eliminates RTL technology penalty of I.P. reuse

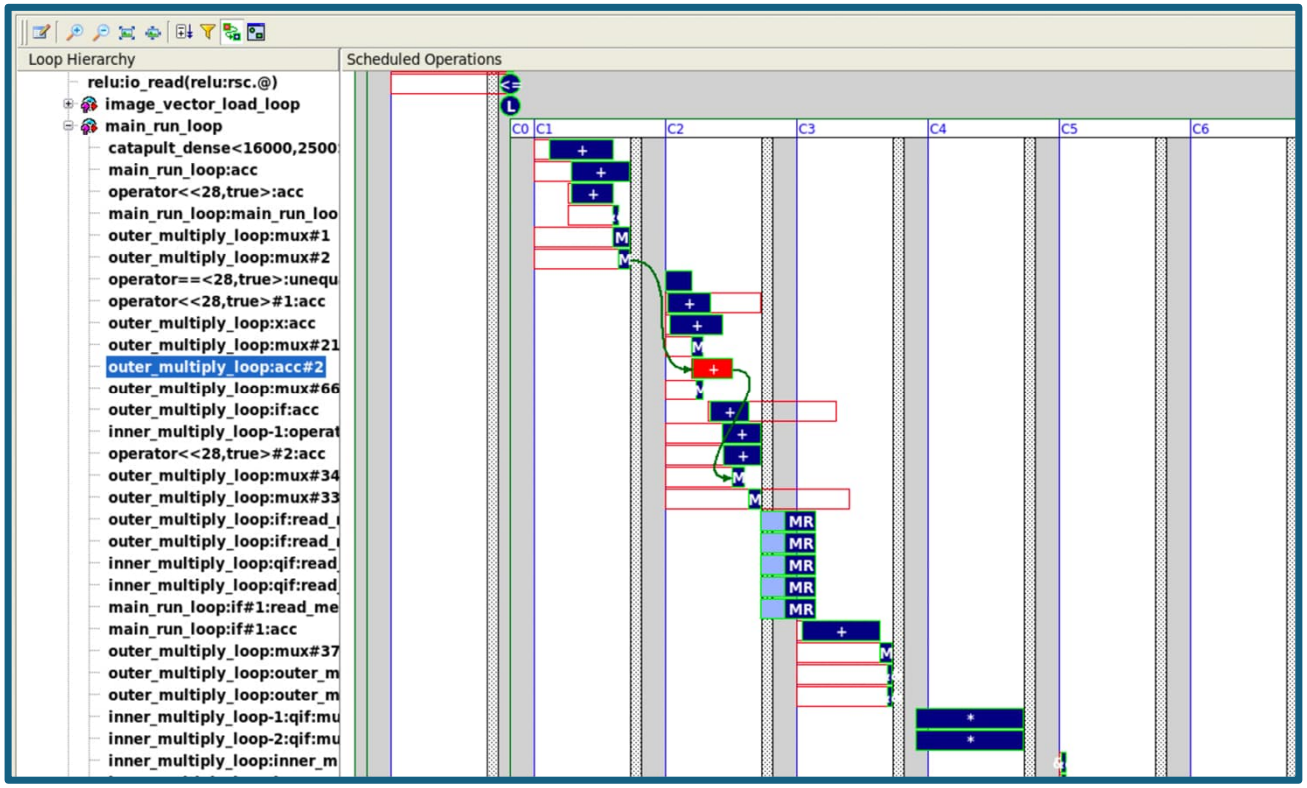




High-Level Synthesis Benefits

- Faster design
  - Typically, RTL design phase is 2X faster for novice users, up to 10X for experts
  - Project start to tape-out can be 4X faster
- Faster verification
  - Algorithm is verified at the abstract level
  - Formal and dynamic verification prove equivalence between C++ and RTL
- Easy technology retargeting, retiming
  - RTL can be mapped to new technology library or clock frequency
  - Simple transition between FPGA and ASIC
- No manual re-interpretation of the algorithm
  - Which is time consuming and error prone

Early Performance Estimation



Schedule of RTL implementation

- Broken into “C-Steps”
- Correlates to source

Early Area & Power Estimation

Breakdown of early area estimates

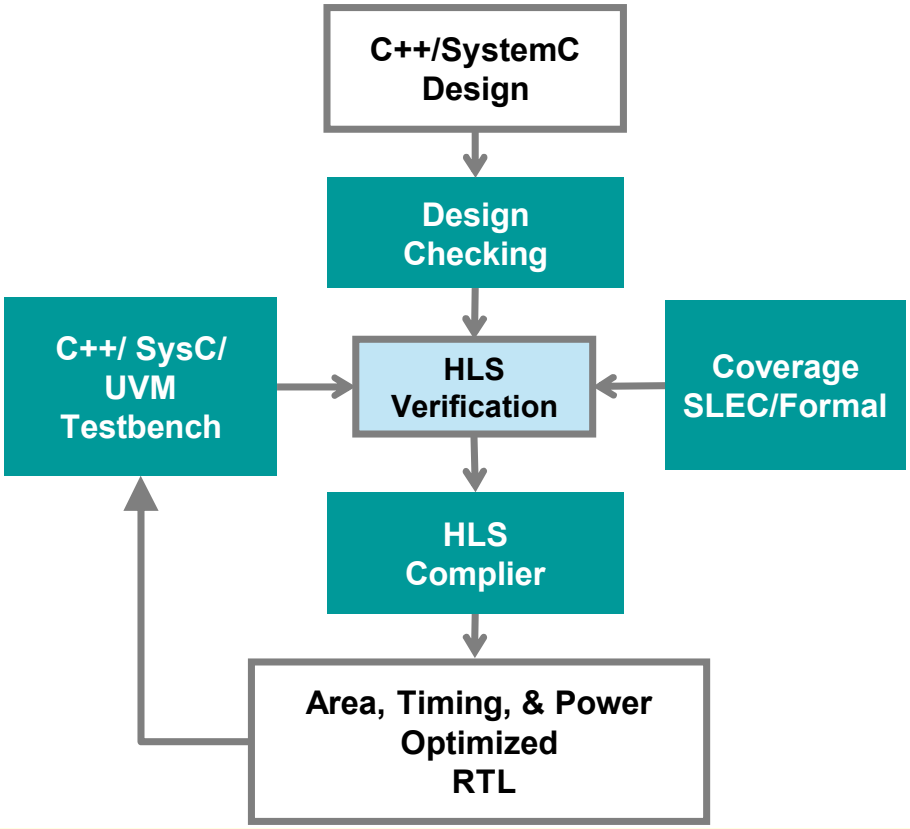
Report: Area Score

Solution	Registers	MUX	Functional	Logic	Memory	FSM Reg	FSM Comb	FSM	Datapath	Total Reg	Total Area	Rom
orig.v1 (extract)	12908.45	59.58	84813.59	891.64	0.00	10.00	0.00	10.00	98673.27	12918.45	98683.27	0.00
opt.v1 (power)	**13112.74**	**1847.10**	**84843.81**	**976.59**	**0.00**	**10.00**	**0.00**	**10.00**	**100780.24**	**13122.74**	**100790.24**	**0.00**

Breakdown of early power estimates

Solution	Total	Tot Mem	Tot Reg	Tot Comb	Tot Seq	Tot Clk	Dyn Tot	Dyn Mem	Dyn Reg	Dyn Comb	Dyn Seq	Dyn Clk	Leak Tot	Leak Mem	Leak Reg	Leak Co...	Leak Seq	Leak Clk	SA Flops	SA User...
opt.v1 (power)																				
pre_pwropt_default_Verilog	34354.20	0.00	7091.67	25190.49	0.00	2072.06	32654.50	0.00	6929.07	23656.30	0.00	2069.18	1699.68	0.00	162.60	1534.19	0.00	2.88	1.56	20.74

HLS Verification Flow



- Pre-synthesis checks
  - Code linting, design check
  - Coverage
- Post-synthesis verification
  - C++/RTL comparisons
  - Assertions and properties
  - Sequential equivalence
  - Coverage

Design Checking - linting

- Static code analysis
  - “lint” for code correctness
  - Quality of results (QofR) checking flags inefficient constructs
  - Formal property checking
- Reduces debugging needed in later design stages
- Find coding style issues that synthesize poorly

Pre-Synthesis Coverage

- Ensures pre-synthesis tests exercise all implemented behaviors
- Formal engine identifies and excludes unreachable code
- Supports HDL coverage tools and reporting, merge with RTL coverage data

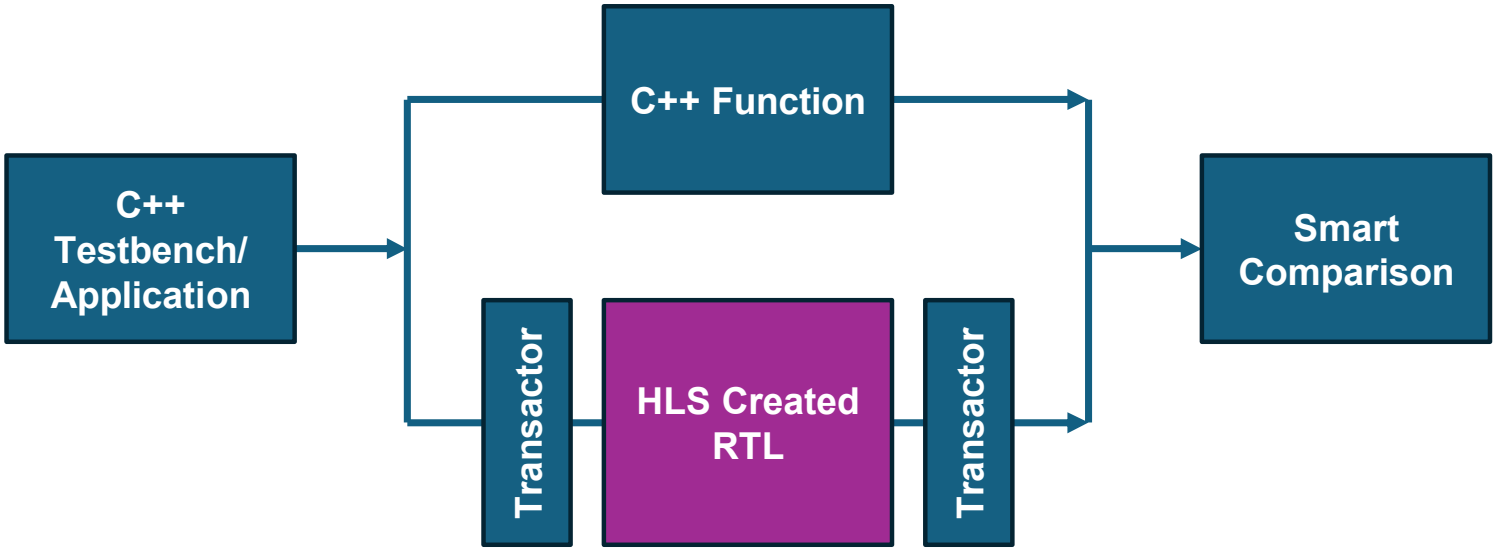
Coverage Database

Design hierarchy	Coverage%	Statement%	Branch%	Expression%
...(\accelerator::accelerator#void(sc_core::sc_module_name))	72.03%	92.18%	81.14%	100.00%
...terface::bus_interface#void(sc_core::sc_module_name))	91.99%	91.97%	84.00%	
...inst :(\conv2d::conv2d#void(sc_core::sc_module_name))	70.84%	91.17%	80.66%	100.00%
...ction=(abstraction_t)2u]#void(sc_core::sc_in<bool>&))	100.00%	100.00%		
...ction=(abstraction_t)2u]#void(sc_core::sc_in<bool>&))	100.00%	100.00%		
...AM_model<AUTO>::mem<DTYPE,692224U,TLM>&))	100.00%	100.00%		
...AM_model<AUTO>::mem<DTYPE,692224U,TLM>&))	100.00%	100.00%		
...core::conv2d_core#void(sc_core::sc_module_name))	66.94%	85.96%	77.47%	100.00%
...buffer::mem_buffer#void(sc_core::sc_module_name))	84.20%	96.20%	89.74%	
...[THREAD] :(\mem_buffer::run_mem_buffer#void())	84.08%	95.83%	89.74%	
...n_tsource_abstraction=(abstraction_t)2u]#void())	100.00%	100.00%		
...ed<16,7>;intSIZE=3]#void(Marshaller<144U>&))	100.00%	100.00%	100.00%	
...on=(abstraction_t)2u]#void(unsignedint,DTYPE))	67.85%	85.71%	50.00%	
...traction_t)2u]#void(ac_int<20,false>,DTYPE))	100.00%	100.00%		
...eam&(std::ostream&,constarray_t<DTYPE,3>&))	100.00%	100.00%	100.00%	
ram1 :(\ram::ram#void(sc_core::sc_module_name))	81.53%	79.59%	65.00%	100.00%
...ve_r_process[THREAD] :(\ram::slave_r_process#void())	88.38%	81.81%	83.33%	100.00%
...e_w_process[THREAD] :(\ram::slave_w_process#void())	70.00%	60.00%	50.00%	100.00%

HLS Formal Verification

- Assertion and cover properties are passed from HLS source to created RTL
  - Used in downstream verification
- Sequential equivalency formal check proves correctness of RTL
  - Produces counter examples for mis-matches
  - Usually cannot prove the entire module, but reduces verification space

C++ to RTL Comparison



Leverages common testbench or application to exercise RTL

C++ Testbench Challenges

- C++ testbenches (and accompanying applications) typically exercise large and realistic stimulus data sets
  - These can easily overwhelm logic simulation when the accelerator is modeled at RTL
- Impractical to scale down datasets for logic simulation
  - Large manual effort, error prone
  - Complete verification of algorithms may require large data sets
    - Video/signal processing, AI, performance/throughput, stress tests

C++ Testbench Challenges: Example

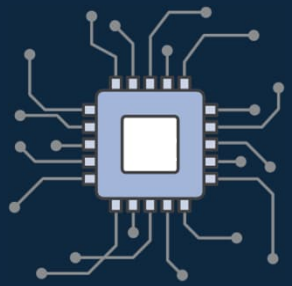
- Video compression algorithm
- Need to verify compression ratios and performance across a variety of realistic video content
  - RTL verification engineer: checks state machines and decision points...
    - Team spends weeks developing test suites
  - C++ developer: grabs 100 hours of video off YouTube and lets them run over a weekend
    - Writes a python script to orchestrate testing, is done in 2 days
    - But: has 10,800,000 4K frames to compress and decompress (~100 trillion pixels)
    - At RTL in logic simulation >100,000 CPU hours

Accelerating HLS Verification

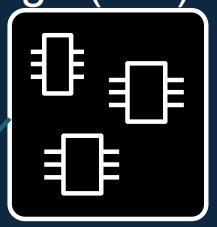
- Hardware acceleration moves RTL evaluation from serial compute to parallel execution in programmable logic
- Scales to large designs
- Performance depends on design size
  - 1 FPGA = 100s MHz
  - 10 FPGAs = 10s MHz (FPGA prototyping systems)
  - 100-1000 FPGAs = single digit MHz (Emulation systems)

Building FPGA Applications for the Cloud

Amazon Machine Image (AMI)



Amazon FPGA Image (AFI)



F2 instance can have any number of AFIs

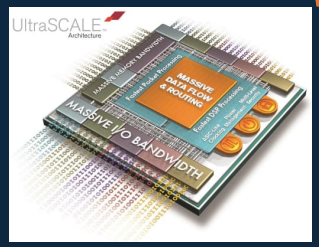
AFI can be loaded into the FPGA in seconds

Launch F2 instance and load AFI

CPU Application

AMI

PCIe



DDR Controllers

DDR-4 Attached Memory

AWS F2 FPGA Instances

AMD server running Ubuntu Linux with PCI card with up to 8 FPGAs

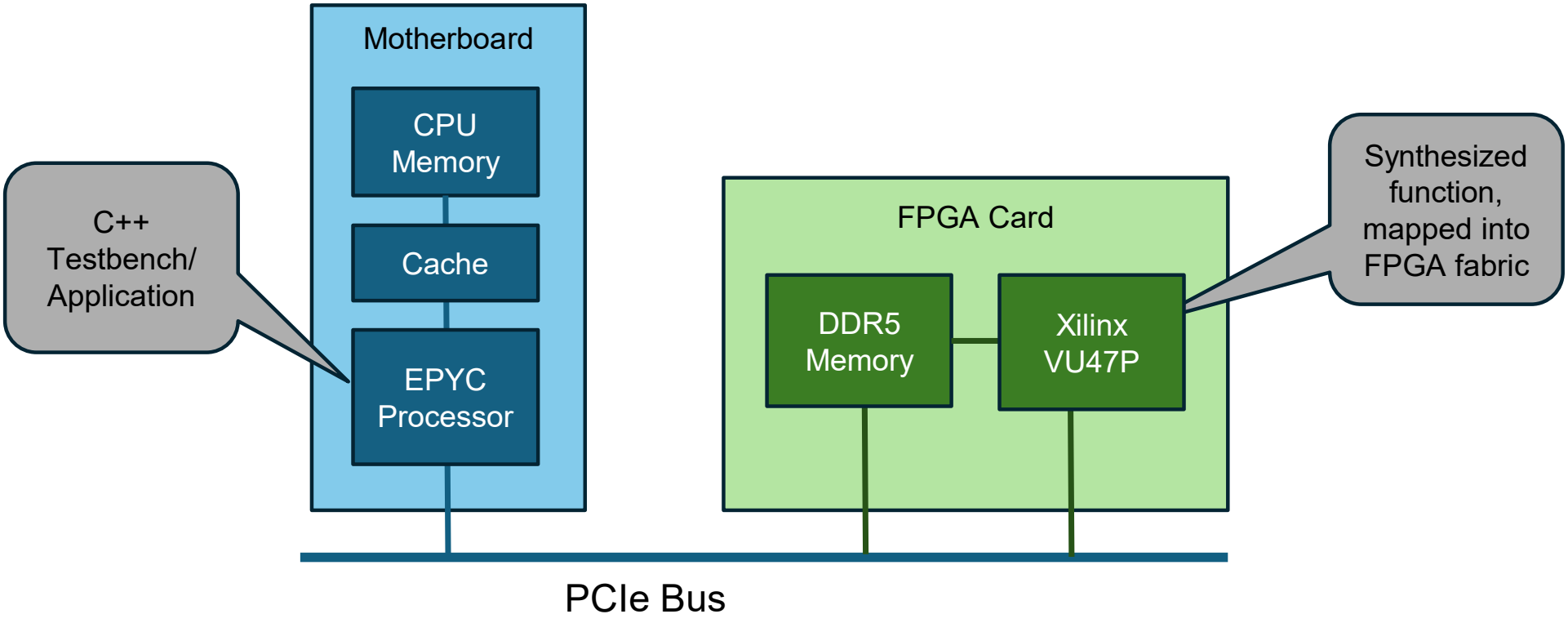
AMD(Xilinx) Virtex UltraScale+ HBM VU47P FPGAs, 3.9 million LUTs

Instance	FPGA	vCPU	Memory (GiB)	NVMe SSD (GiB)	Networking
f2.6xlarge	1	24	256	950	10 Gbps
f2.12xlarge	2	48	512	1,900	25 Gbps
f2.48xlarge	8	192	2,048	7,600	100 Gbps

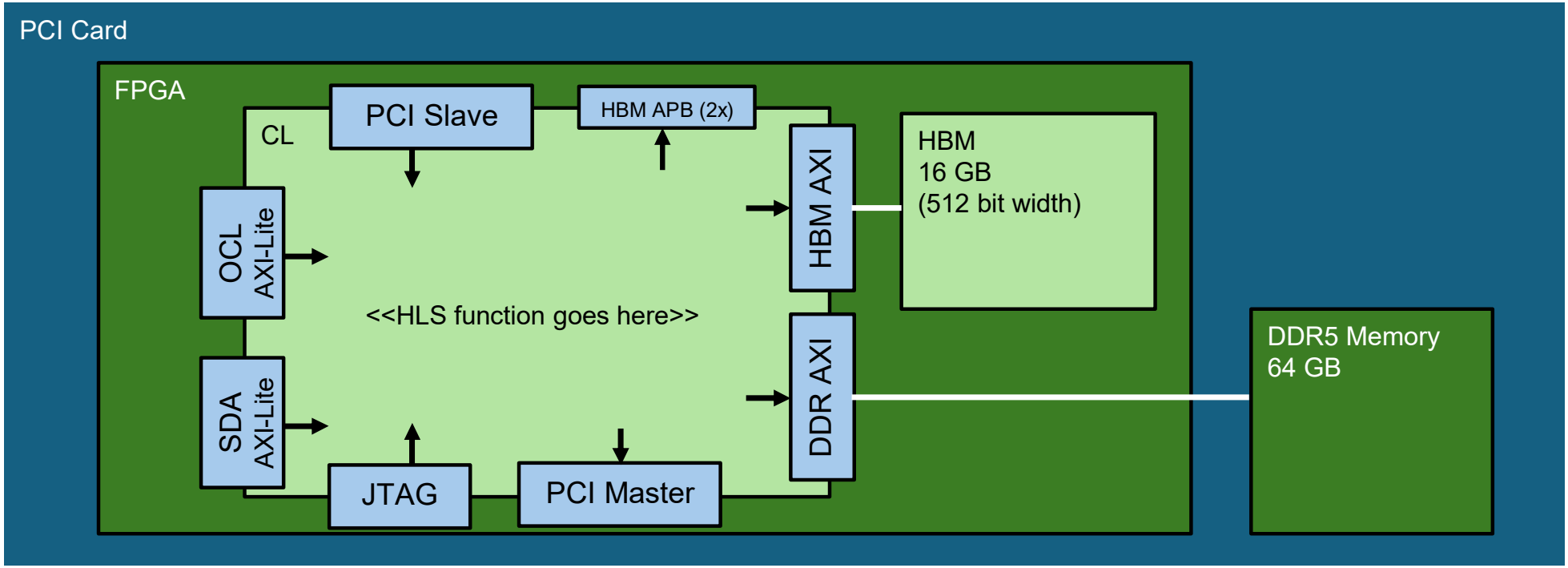
3rd generation AMD EPYC processors (code named Milan) running up to 3.6 GHz

24 vCPU:1 FPGA ▲ from 8 vCPU:1 FPGA
100 Gbps Networking ▲ from 25 Gbps
16 Gb:1 vCPU ▲ from 15.25 Gb:1 vCPU

AWS F2 FPGA Instances



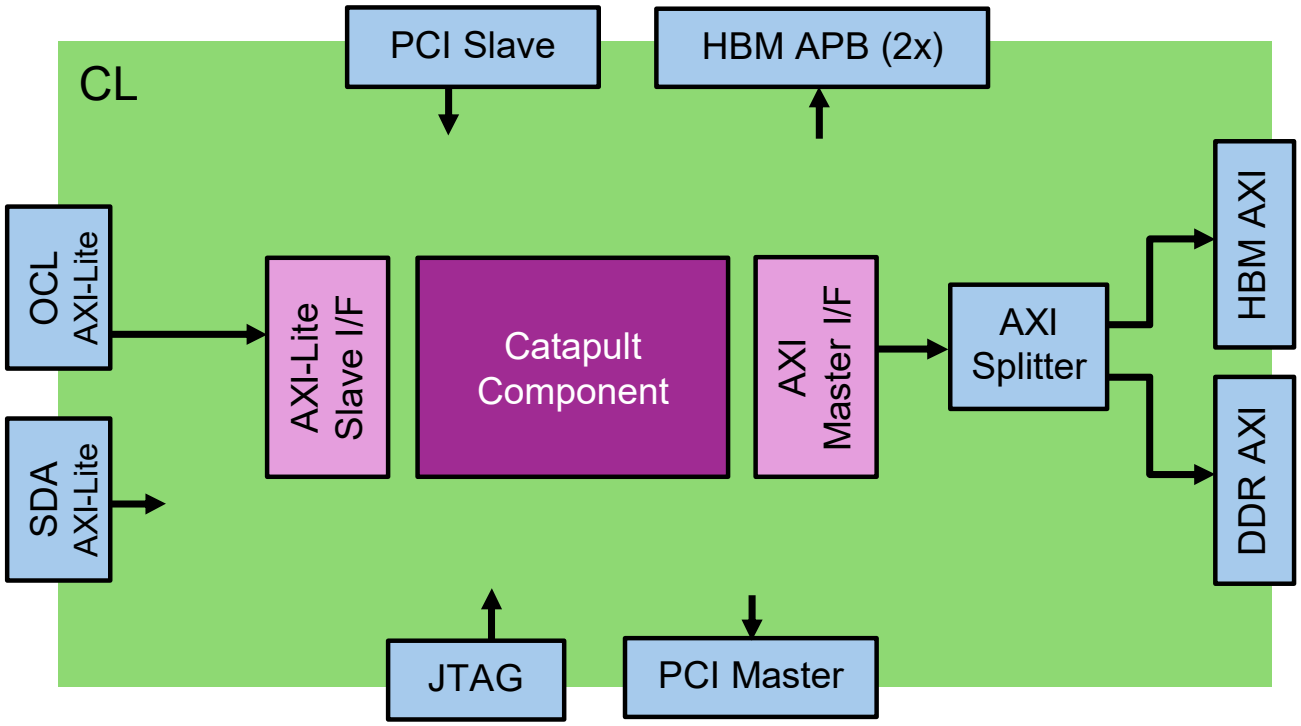
The “Shell”


PCI Card
FPGA
CL
PCI Slave
HBM APB (2x)
OCL AXI-Lite
SDA AXI-Lite
<<HLS function goes here>>
JTAG
PCI Master
HBM AXI
DDR AXI
HBM
16 GB
(512 bit width)
DDR5 Memory
64 GB

Interfaces

- Driven by EPYC processor
  - OCL AXI-Lite – register accesses
  - SDA AXI-Lite
  - CL AXI Slave
    - PCI from HOST CPU -> AXI bus -> CL
    - Enables full AXI transactions across the PCI bus from software on host CPU
  - JTAG – for debugging logic
- Driven by CL logic
  - CL AXI Master
    - CL->AXI bridge->PCI to host
  - DDR AXI
    - Full AXI to DDR5 memory (512 bits wide)

CL (Customer Logic) Connections



Catapult exposes
- Clock & reset
- AXI-Lite Slave
- AXI-Master

HLS module

HLS interface

AWS IP

HLS Accelerator Design Flow

```
main()
{
   fn_a(p1, p2);
   fn_b(p2, p3);
   fn_c(p3, p4);
}
```

Main program is run on embedded CPU (EPYC)

Profile to identify ”bottleneck” functions

- Can re-arrange computations to optimally accelerate

Create new interface function “fn_b_prime()”

- Same function signature
- Handles interface and synchronization with hardware accelerator
  - Parameters moved to CSR registers
  - Memory accesses need to be moved to shared memory

Original fn_b() is synthesized through Catapult to RTL representation

HLS Accelerator Design Flow

```
fn_b(p2, p3)
{
  // complex compute
}
```

C++

```
fn_b(p2, p3)
{
  ac_fixed<x, y> var1, var2;
}
```

Quantized algorithm

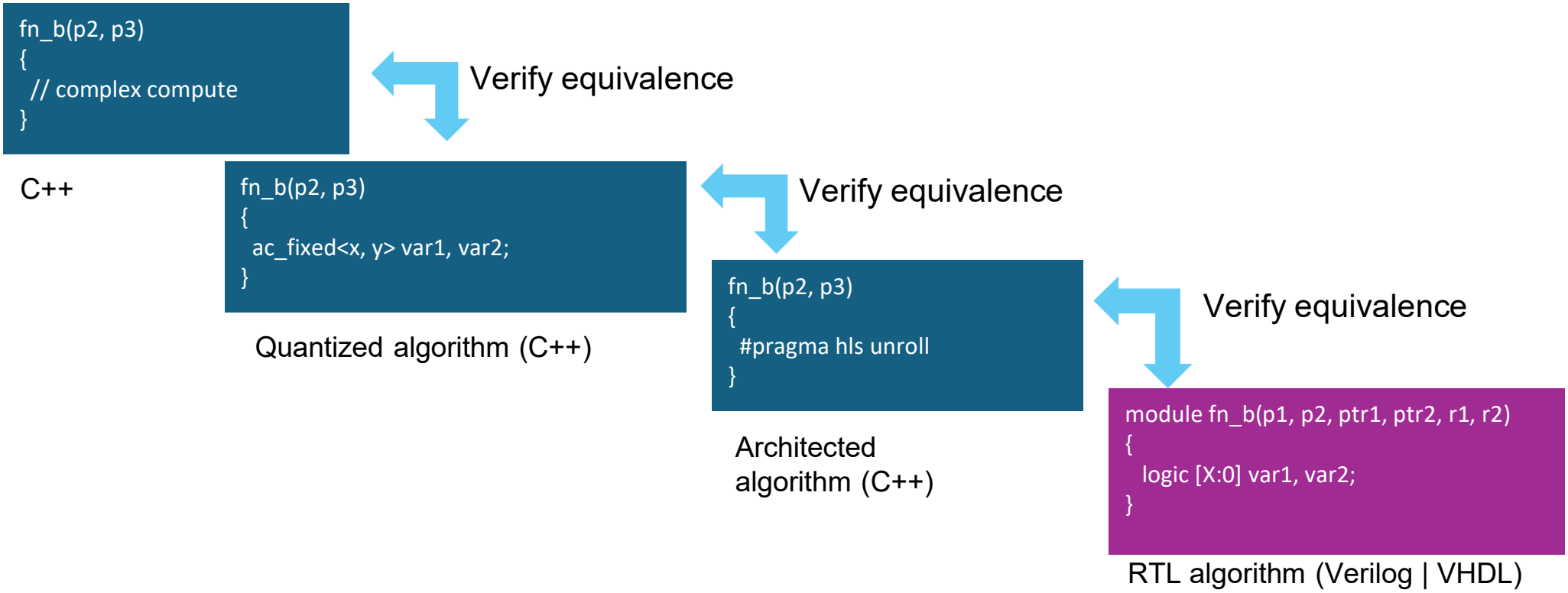
```
fn_b(p2, p3)
{
  #pragma hls unroll
}
```

Architected algorithm

```
module fn_b(p1, p2, ptr1, ptr2, r1, r2)
{
   logic [X:0] var1, var2;
}
```

RTL algorithm

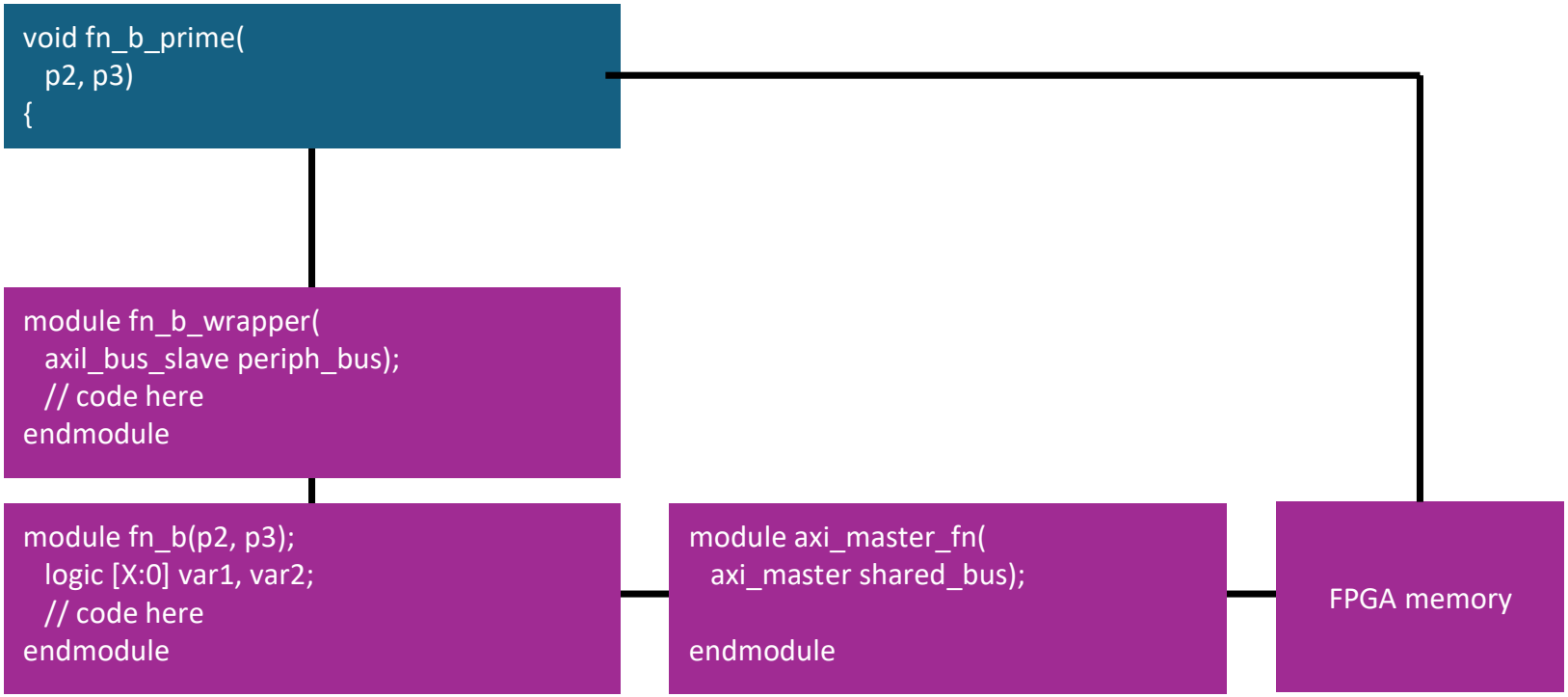
HLS Accelerator Verification



HLS Accelerator Design Flow

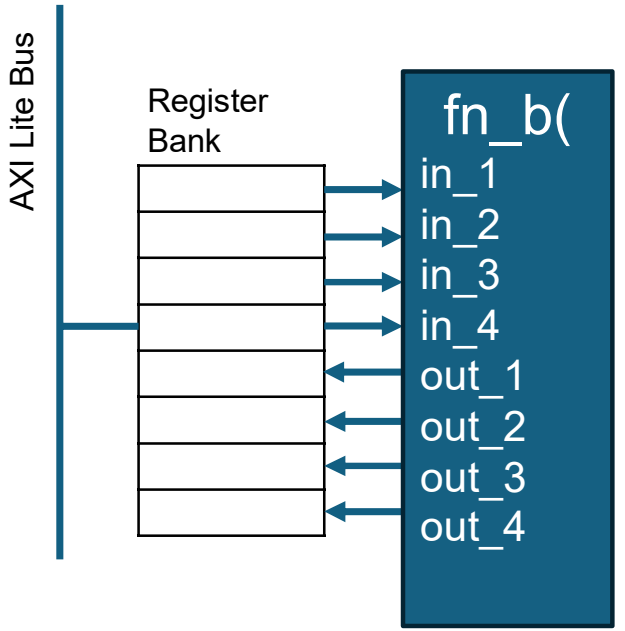
Software side
Running on host or embedded processor

Hardware side
Running on logic simulator or FPGA



Module Wrapper AXI Lite Slave

- AXI slave holds a memory mapped register bank to drive/sample inputs and outputs to the function
  - Holds basic C types or small arrays
  - Or channels for passing data streams in and out
- AXI slave is driven by pci_poke and pci_peek functions from host program on EPYC core



Module AXI Master

- Uses addresses passed in through parameters to access memory
  - Can access as single transaction or multi-beat bursts
  - Instantiate HLS IP for AXI master (SystemC or C++)
- 16 Gbytes on chip HBM
  - Accessible through 512 to 16K (512x32) bit width full AXI bus
- 64 Gbytes on board DD5 memory
  - Accessible through 512 bit wide full AXI bus

Software Access to PCIe

- Software accesses AXI Lite interface and memory through PCIe driver calls
- Wrapper function (sw-side) replaces parameter passing with calls to pci_peek and pci_poke calls

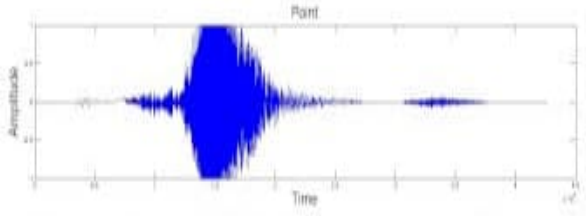
```
// set-up routines
fpga_pci_init();
fpga_mgmt_init();
fpga_pci_attach();

// data passing
fpga_pci_peek();
fpga_pci_poke();
```

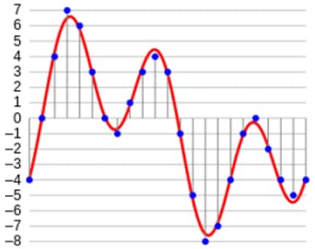
Design Example

- Keyword recognition function
  - Uses CNN for voice to text
  - Convolution and vector/matrix multiply take 97% of computational load
  - Vector/Matrix multiply function (dense layer) built into an accelerator
- Testbench drives 10,000 ten second audio samples
  - Inference is called every 50 ms (20 times per second)
  - Dense layer called 3 times per inference
  - 6 million calls to the dense layer function per test

Wakeword Algorithm



Audio input

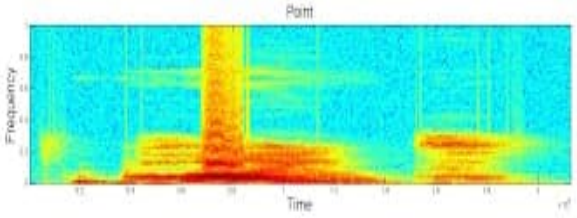


Quantization

-4	0	4	7	

Integer array
(16k x 16 bits)

MFCC()



Spectral Array

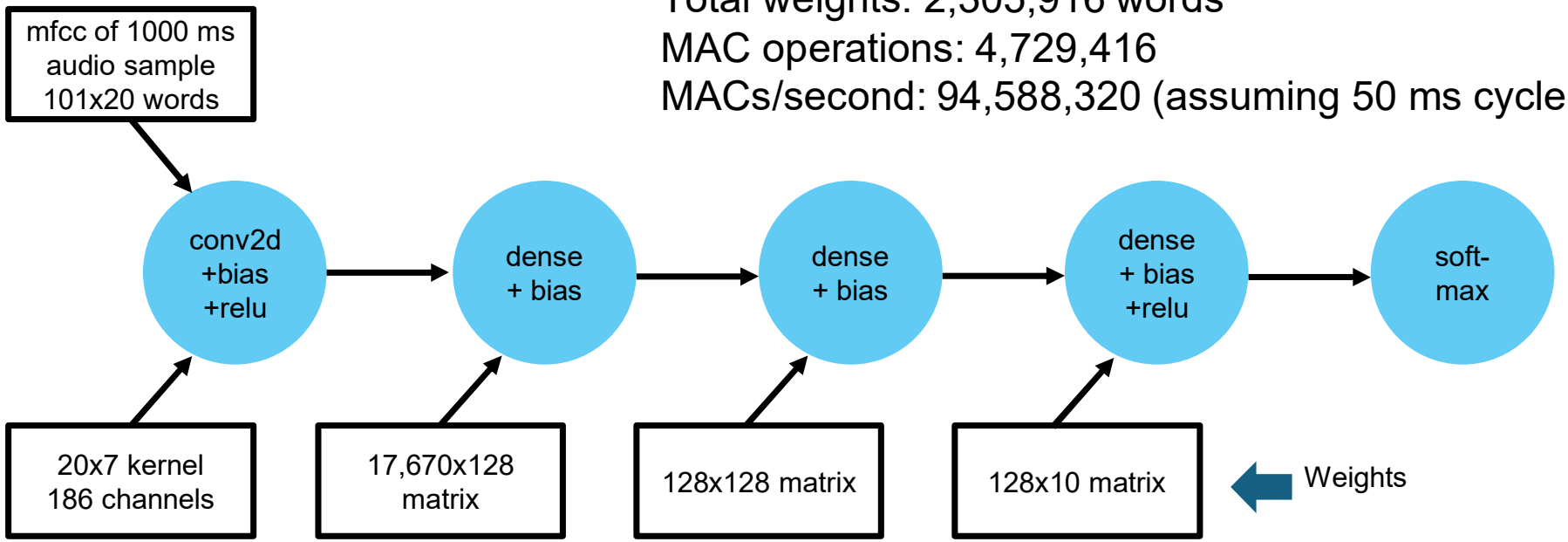
0.123	0.456	-0.872	0.567	
0.324	0.547	0.376	0.231	
0.846	0.183	0.834	0.937	
0.625	0.737	0.746	0.827	

Float array
(101 x 20 x 32-bits)

To Neural Network
as feature map for
training and inferencing

Wakeword CNN

Total weights: 2,305,916 words
MAC operations: 4,729,416
MACs/second: 94,588,320 (assuming 50 ms cycle time)



'cnn-one-fstride4' from 'Convolutional Neural Networks for Small-footprint Keyword Spotting':
http://www.isca-speech.org/archive/interspeech_2015/papers/i15_1478.pdf

Convolution Functio

```
convolution(
  float  *features,
  float  *weights,
  int    num_channels,
  int    filter_size,
  float  *result)
{
 // convolution implementation

}
```

Original C++ Code

```
convolution(
  float  *features,   float  *weights,
  int    num_channels,    int filter_size,
  float  *result)
{
  // write input params
  fpga_pci_poke(axil_handle, 0, features);
  fpga_pci_poke(axil_handle, 4, weights);
  fpga_pci_poke(axil_handle, 8, num_channels);
  fpga_pci_poke(axil_handle, 12, filter_size);

  // copy features to memory
  for(i=0;i<vec_len, i++)
    fpga_pci_poke(mem_handle, base_i, image[i]);

}
```

SW Wrap Code

Dense Function

```
dense(
  float  *features,
  float  *weights,
  int    vec_len,
  int    mat_width,
  float  *result)
{
 // vector/matrix multiply

}
```

Original C++ Code

```
dense(
  float  *features,   float  *weights,
  int      vec_len,     int    mat_width,
  float  *result)
{
  // write input params
  fpga_pci_poke(axil_handle, 0, features);
  fpga_pci_poke(axil_handle, 4, weights);
  fpga_pci_poke(axil_handle, 8, vec_len);
  fpga_pci_poke(axil_handle, 12, mat_width);

  // features are in memory from prior layer

}
```

SW Wrap Code

Results Checking

```
int check_quantized(
      feature_type *f,
      weight_type *w)
{
   feature_type cpp_results[size];
   feature type quant_results[size];
   int errors = 0;

   // run both implementations with the same inputs
   convolution(f, w, c, s, cpp_results);
   convolution_quant(f, w, c, s, quant_results);
```

```
  // compare results

   for (int i=0; i<size; i++) {
     if (!close(cpp_results[i], quant_results[i]) {
        errors++;
        // report failure
     }
    }
     return errors;
}
```

Enables simple comparison between implementations to ensure consistency

Dense Function

Metrics for 6,000,000 calls to dense layer function

	Performance	Abstraction level	Comment
Logic Simulation	~3,000,000 CPU Hours	Full RTL	Estimated time
Architected C++	17,514 seconds	Bit accurate	
Native C++ float	1,879 seconds	Algorithmic	
AWS F2 Instance	22 seconds	Full RTL	Excludes time to copy weights to HBM

FPGA Running at 250 MHz

Accelerated Verification with AWS F2

- Allows common testbench or application to drive RTL for verification
  - Drive RTL DUT with C++ through PCI transactions, HDL through DPI
  - Significantly faster verification throughput vs logic simulation
- Acceleration flow with HLS enables “RTL-in-the-loop” execution
  - Run complete applications for stimulus and results checking
  - Enables realistic compute loads and real-world data in the verification flow
- HLS speeds accelerator design flow
  - Eliminates recoding algorithm in HDL, at RTL abstraction, uses C++ directly
  - Enables optimal design through creation and evaluation of multiple candidate architectures

Questions or Comments

HLS Meets FPGA Prototyping on the Cloud

Sivasankar Palaniappan

Siemens EDA

HLS Technologist

Sivasankar.Palaniappan@siemens.com

Mark Azadpour

AWS

Business Dev. Manager HPC

azadpoup@amazon.com