2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Verification of Inferencing Algorithm Accelerators

Russell Klein

Petri Solanti

Siemens EDA

accellera SYSTEMS INITIATIVE

Agenda

- AI Accelerators
- High Level Synthesis
- Bespoke Accelerator Optimization
  - Neural Network Architecture
  - Quantization
  - Data Movement
- Verification
  - From Python to RTL

2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

MUNICH, GERMANY
DECEMBER 6 - 7, 2022

AI Accelerators

accellera SYSTEMS INITIATIVE

Deploying inferencing systems, where and how

Where

The Cloud

A Gateway

The Edge

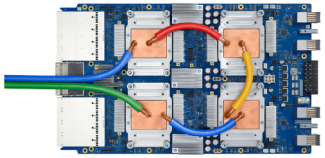
Highest performance and efficiency are achieved with specialized ASIC implementation running on the edge

Higher Latency/Lower development costs

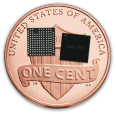
How

CPU

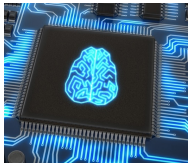
GPU



TPU/NPU



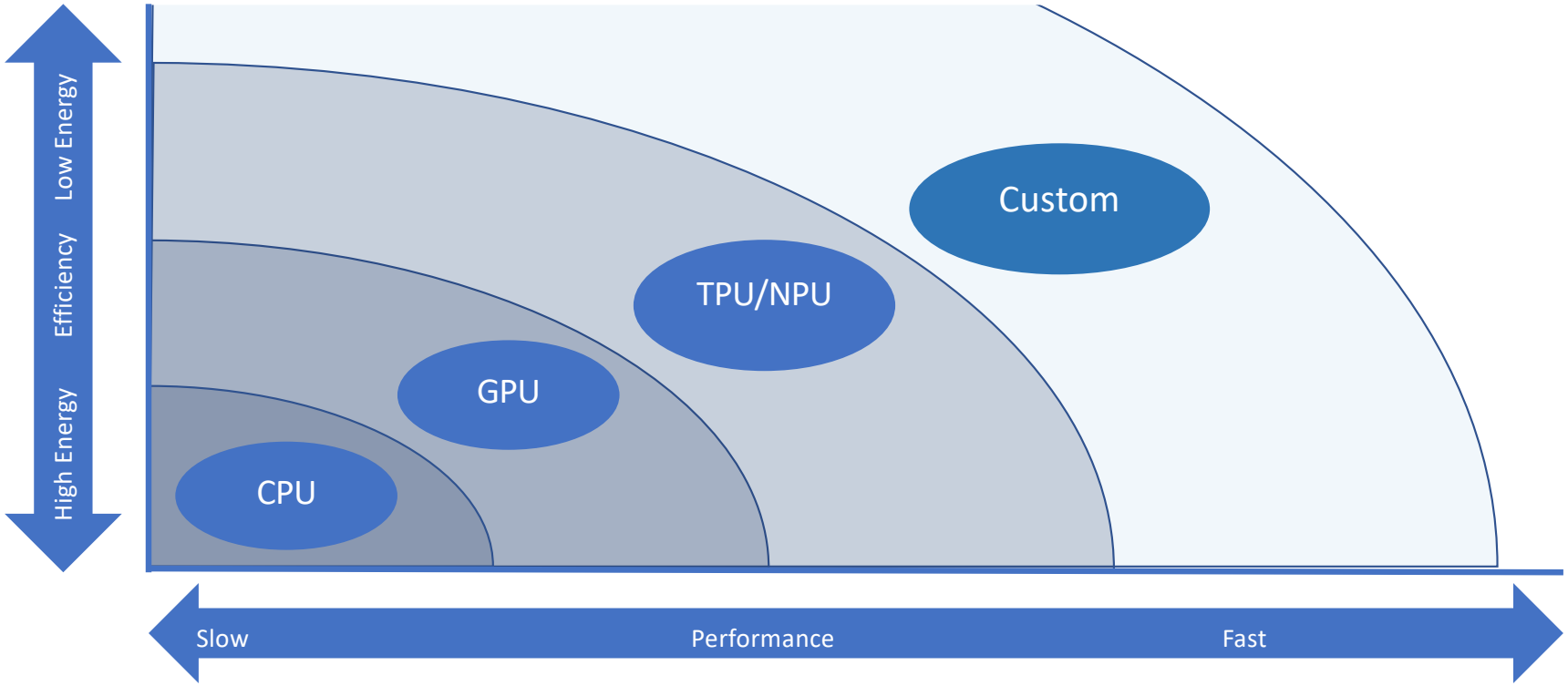
Edge TPU



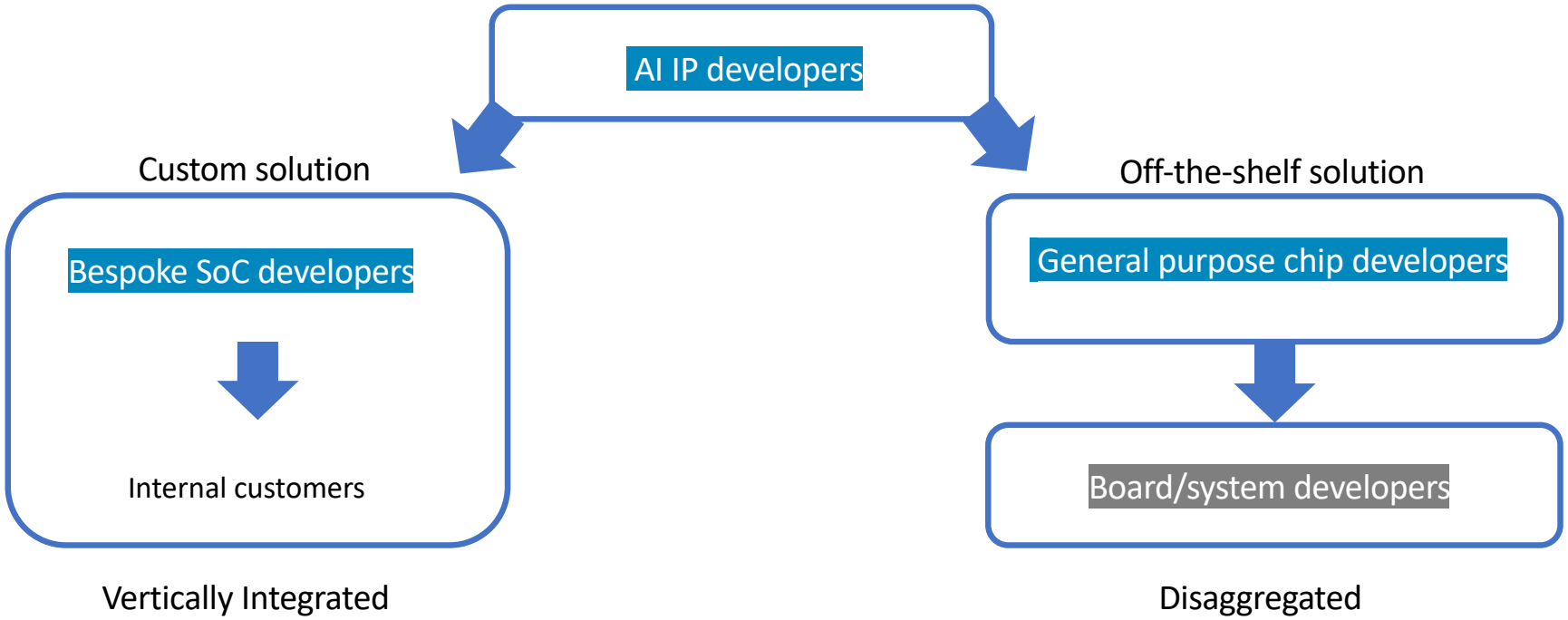
FPGA or ASIC

Higher specialization/Lower energy

Inference Execution (on-chip)



Accelerated AI SoC Eco-system



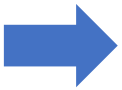
AI Accelerator Verification Challenges

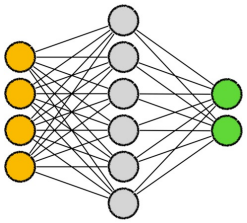
- CPU, GPU, NPU, TPU
  - Verify algorithm implementation runs on IP
  - Verify that IP is correctly integrated
  - IP is assumed to be correct from the IP provider
- Bespoke accelerator
  - Verify the algorithm runs on the accelerator
  - Verify the accelerator is correctly integrated
  - Verify the accelerator functions correctly

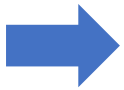
From Python to Hardware

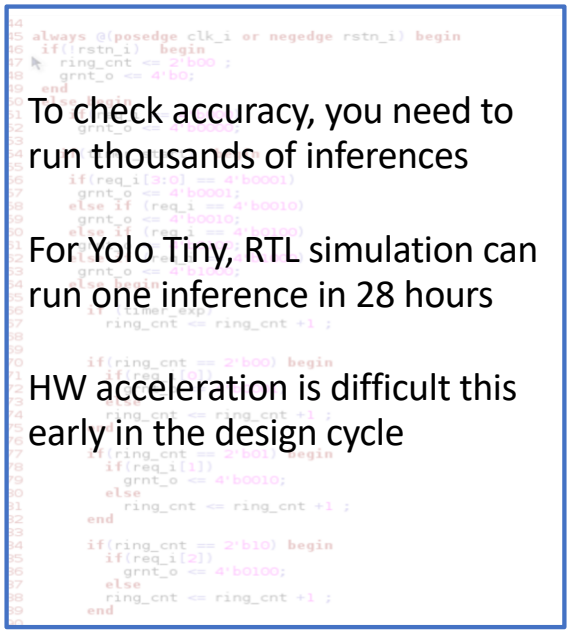
Machine Learning Framework

Iterate to find optimal NN architecture









To check accuracy, you need to run thousands of inferences

For Yolo Tiny, RTL simulation can run one inference in 28 hours

HW acceleration is difficult this early in the design cycle

Python

Verilog

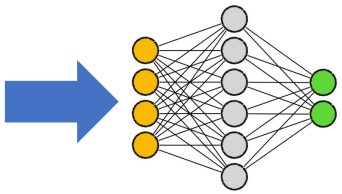
A Better Path

TensorFlow

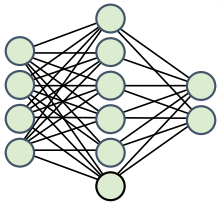
Caffe2

Machine Learning Framework

Iterate to find optimal NN architecture



Python



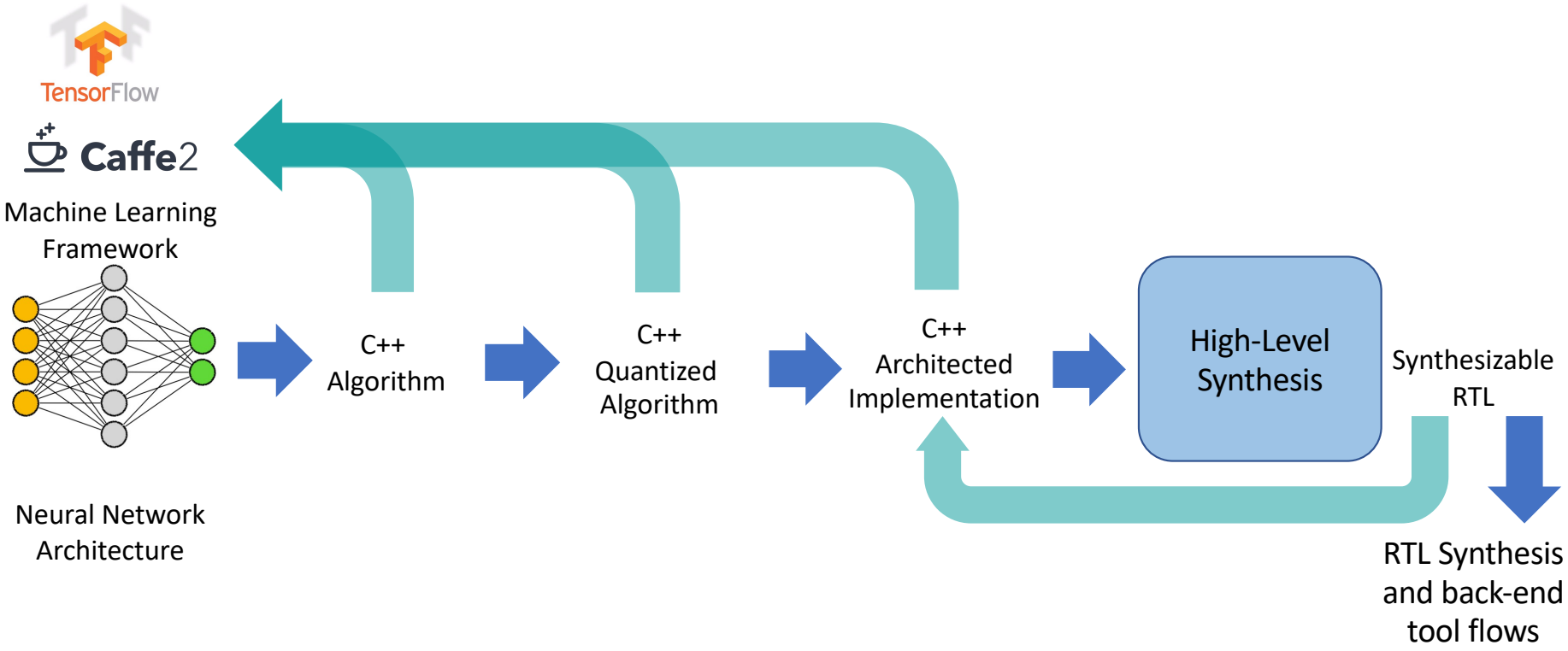
C++

Automated conversion with HLS

With this flow, proving equivalency between C++ and Verilog is much faster and easier

Verilog

HLS AI Design Flow



2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

MUNICH, GERMANY
DECEMBER 6 - 7, 2022

High-Level Synthesis

accellera SYSTEMS INITIATIVE

What is High-Level Synthesis?

```
void copy_to_regs(hw_cat_type *dst, index_type dst_offset, raw_memory_line *src, index_type src_offset, index_type size)
{
    // read out of internal memories to an array of registers
    // *should* make it easy for catapult to pipeline access to internal memories

    index_type count;
    index_type n;

    static const index_type stride = STRIDE;

    count = 0;
    while (count < size) {
        n = stride;
        if ((size - count) < stride) n = size - count; // mis-aligned at the end of transfer
        read_line(dst, dst_offset, src, src_offset, n);
        count += n;
        src_offset += n;
        dst_offset += n;
    }
}
```

C/C++ or SystemC

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

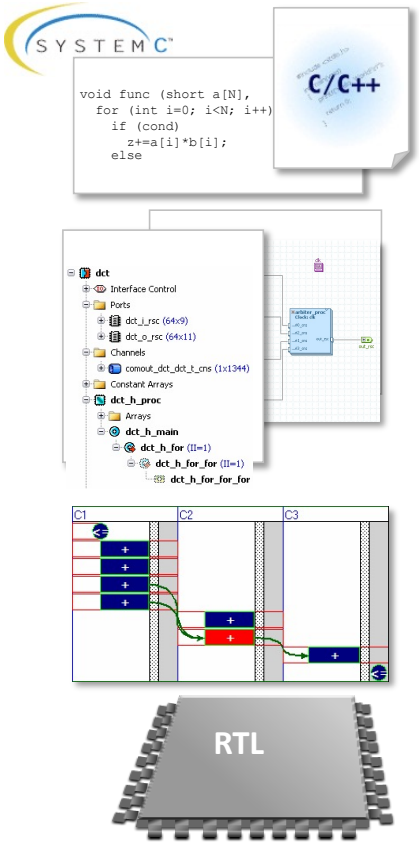
High-Level Synthesis

```
module timer (
    input          clock,
    input          resetn,
    input  [1:0]   trans,
    input  [29:0]  address,
    input  [3:0]   bl,
    input          we,
    input          ce,
    input  [31:0]  write_data,
    output [31:0]  read_data,
    output [1:0]   resp,
    output         ready
  );

  reg [31:0]  timer_value;
  reg [31:0]  rd_reg;

  //reg         ready_out = 1'b1;
  reg         resp_out  = 2'b00;

  ready_gen #(1) rg (clock, resetn, ce, trans, ready_out);

  assign ready          = ready_out;
  assign resp           = resp_out;
  assign read_data      = rd_reg;

  always @(posedge clock or negedge resetn) begin
    if (resetn == 1'b0) begin
      timer_value <= 32'h00000000;
    end else begin
      timer_value <= timer_value + 32'h00000001;
    end
  end

  always @(posedge clock or negedge resetn) begin
    if (resetn == 1'b0) begin
      rd_reg <= 32'h00000000;
    end else begin
      //if (trans[1] & ce) begin
      if (ce & !we) begin
        rd_reg <= timer_value;
      end
    end
  end

endmodule
```

Synthesizable RTL

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
- Automatic arithmetic optimizations and bit-width trimming
  - Bit-accurate types enable mathematical accuracy to propagate to outputs
- Multi-objective process-aware scheduling for both 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 10X for experienced users
  - Project start to tape-out can be 4X faster
- Faster verification
  - Algorithm is verified at the abstract level
  - Formal and dynamic verification can be used to prove equivalence between C++ and HDL
- Easy technology retargeting, retiming
  - RTL can be mapped to new technology library or clock frequency by re-synthesizing
  - Simple transition between FPGA and ASIC implementation

How does High-Level Synthesis Work?

- HLS automatically meets timing based on the user-specified clock constraints.
- HLS understands the timing and area of the target technology and uses this to insert registers when needed.
  - Using the right HLS target library is very important!
- HLS closes on timing using:
  - Data flow graph analysis
  - Resource allocation
  - Scheduling
  - Resource sharing and timing analysis

High-Level Synthesis: Bit-Accuracy

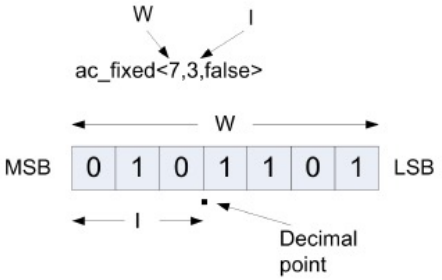
- Different datatypes used in HLS tools:
  - Algorithmic C (AC) data types
    - Faster in simulation
  - System C data types
    - Slower than AC datatype in simulation
  - Arbitrary Precision (AP) data types
- Needed to model true hardware behaviour
  - Bit-accuracy simulated in source
  - Provides a path for automated bit-for-bit comparison of C++ and RTL
- Fixed point types include optional rounding and saturation modes.

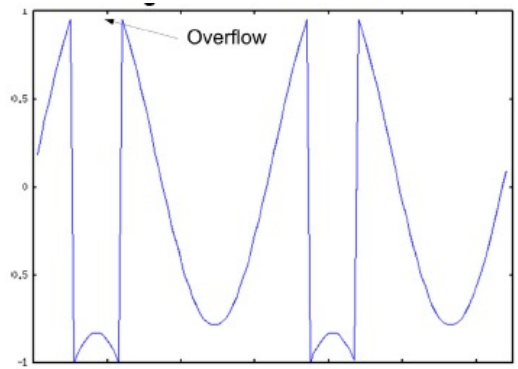
Bit-Accurate AC Data Types

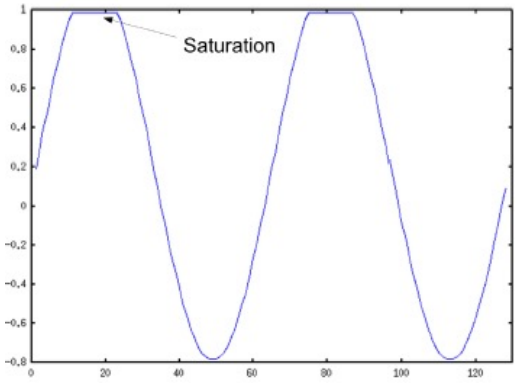
- Allows designers to model a signed or unsigned bit vector representing
  - Arbitrary length integer: ac_int
  - Arbitrary length fixed-point: ac_fixed
  - Arbitrary length floating-point: ac_float, ac_std_float
- Saturation and overflow behavior like in RTL
- Decimal or integer numerical values – no need for scaling nor conversion

The Algorithmic C fixed point data types are declared as:

```
ac_fixed<W,I,S> x;
```







AC Types Example – Integer and Fixed point

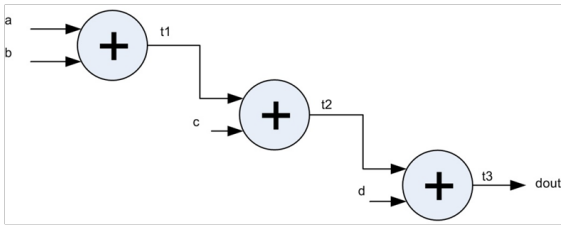
- 10 bit total, 1 integer bit, signed
  - -1.0 to 0.99:
- Accumulator for 3 bits headroom
  - No round/saturate
- Simple unsigned “int” 3-bit representation

```
#include <ac_fixed.h>
#define HEADROOM 3
typedef ac_fixed<10,1,true> coeff_type;
typedef ac_fixed<12,12,true,AC_RND_INF,AC_SAT> data_type;
typedef ac_fixed<22 + HEADROOM, 12 + 1 + HEADROOM, true> acc_type;
typedef ac_int<3,false> mode;
```

Data Flow Graph Analysis

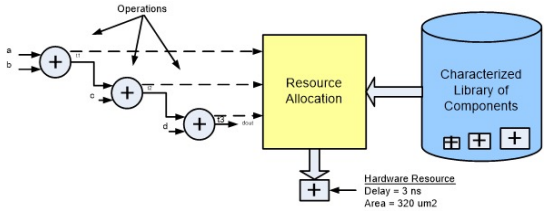
- HLS analyzes the data dependencies between the various steps in the algorithm
  - Analysis leads to Data Flow Graph (DFG) description
  - Each node of the DFG represents an operation defined in the C++ code
    - For this example, all operations use the "add" operator
  - Connections between nodes represent data dependencies and indicate the order of operations

```
void accumulate(int a, int b,
                int c, int d,
                int &dout){
   int t1,t2;
   t1   = a  + b;
   t2   = t1 + c;
   dout = t2 + d;
}
```



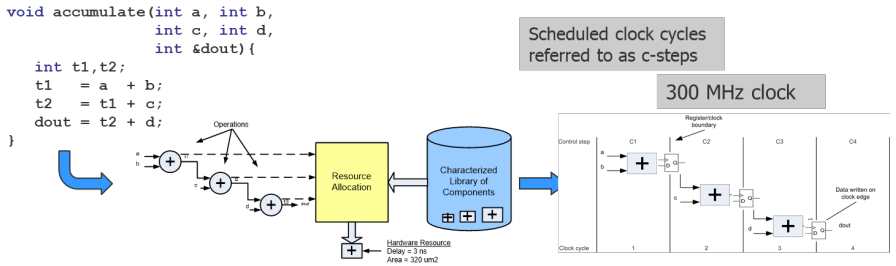
Resource Allocation

- During DFG analysis each operation is mapped onto a hardware resource which is then used during scheduling.
- Resources corresponding to a physical implementation of the operator hardware
  - Implementation is annotated with both timing and area information which is used during scheduling
  - Operations may have multiple hardware resource implementations that each have different area/delay/latency trade-offs
- Resources are selected from a technology specific library



Scheduling

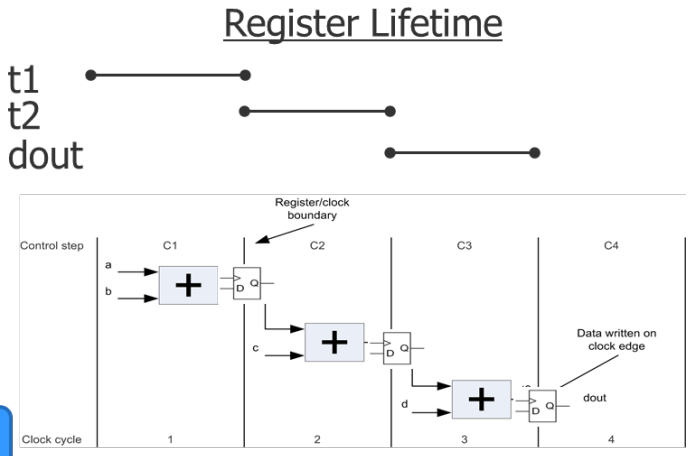
- HLS adds "time" to the design during the process known as "scheduling“
- Scheduling takes the operations described in the DFG and decides when (in which clock cycle) they are performed
  - Has the effect of adding registers when needed to meet timing
  - Similar to what RTL designers would call pipelining, by which they mean inserting registers to reduce combinational delays
- Scheduling automatically shares resources

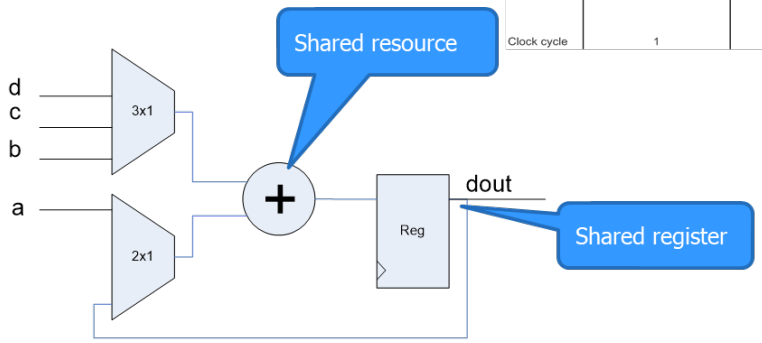


Resource and Register Sharing

- HLS shares resources
  - Via design constraints and automated analysis
  - Explicit mutual exclusivity in the code
- HLS shares registers
  - Via lifetime analysis

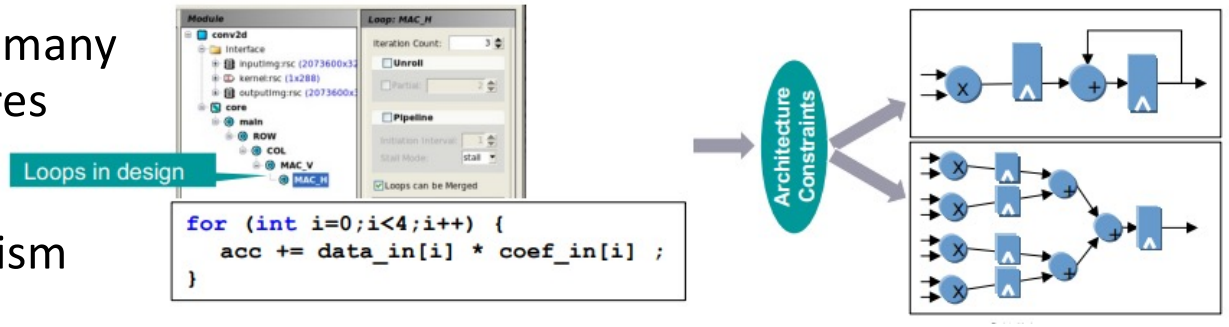
```
void accumulate(int a, int b,
                int c, int d,
                int &dout){
   int t1,t2;
   t1   = a  + b;
   t2   = t1 + c;
   dout = t2 + d;
}
```





HLS Optimizations for Area and Performance

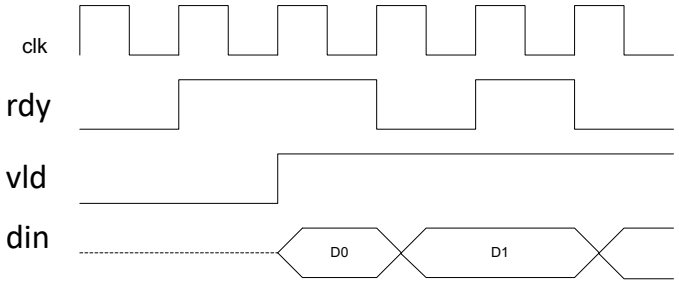
- Loop optimizations
  - provides a way to explore many possible micro-architectures
- Loop Unrolling
  - Represents space/parallelism
- Pipelining
  - Represents time/throughput
- Automatic merging



Interface Synthesis

- Adding an interface protocol to an untimed C++ design is known as “Interface Synthesis”
- C++ source code does not specify the protocol
- Interface synthesis allows the protocol to be defined using the HLS tool

```
void simple (int din, int &dout)
{
  dout = din;
}
```



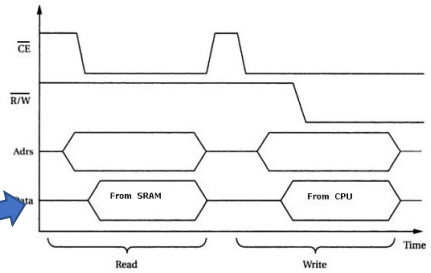
Memory Interface Synthesis

- Automatically mapped to ASIC or FPGA memories/registers
- User control over memory mapping
- Arrays on the design interface can be synthesized as memory interfaces
  - Address, data, control

```
void simple_function(… ,int data[1024]){
  int mem[1024];
  <function body>
}
```

Memory interface protocol

Instantiated memory wrapper



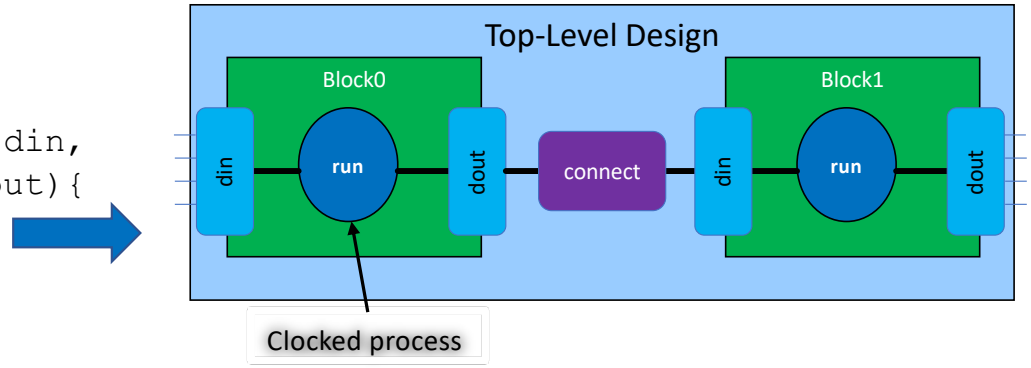
simple_function RTL

mem wrapper

Designing Concurrent Clocked Hierarchies

- Multi-block Design
  - Top-down or bottom-up synthesis
  - User specifies design blocks
- Design blocks/processes run in parallel
  - High throughput

```
void CCS_BLOCK(run)(ac_channel<uint4 > &din,
                    ac_channel<uint20 > &dout){
   inst0.run(din,connect);
   inst1.run(connect,dout);
 }
```



2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

MUNICH, GERMANY
DECEMBER 6 - 7, 2022

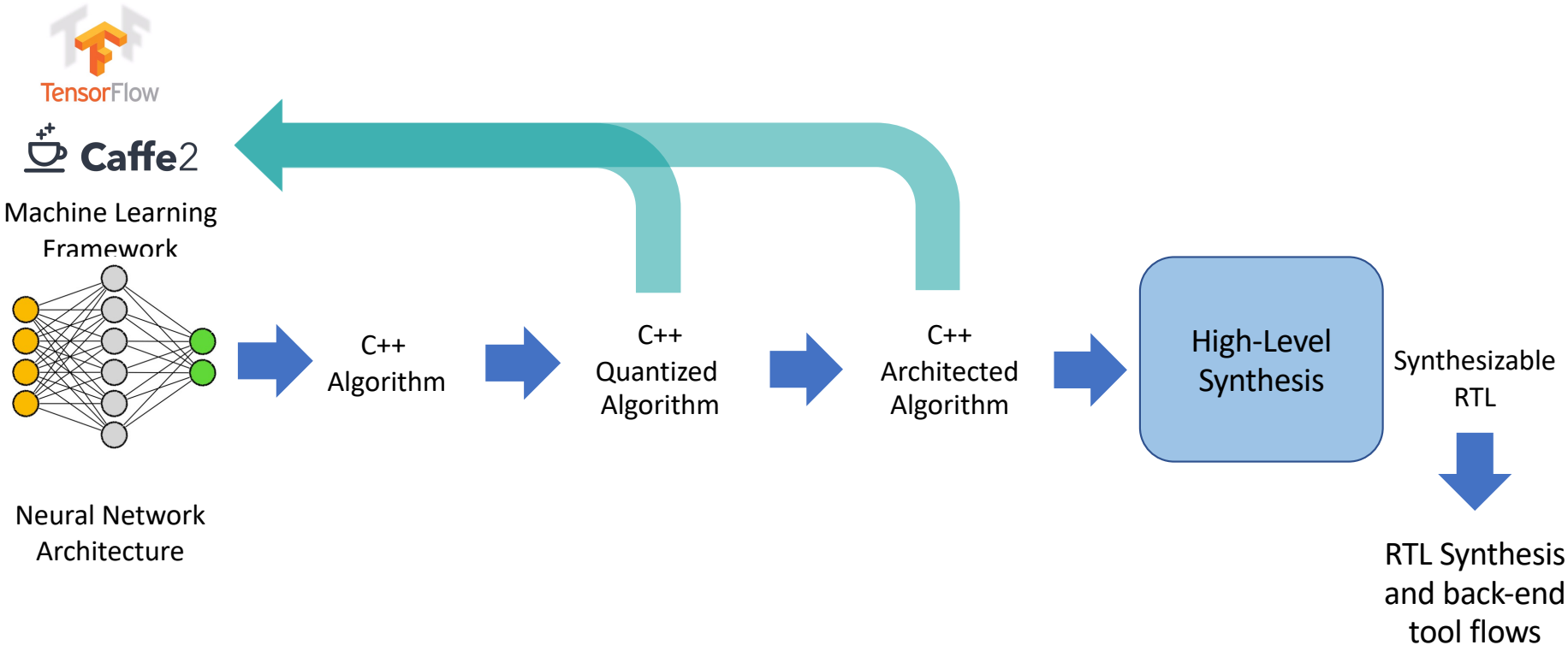
Accelerator Optimization

accellera SYSTEMS INITIATIVE

Accelerator Optimization

- Neural Network Architecture
  - Modifying layers and channels
- Quantization
  - Changing the representation of numbers
- Data Movement, Storage
  - Alter data caching and access patterns

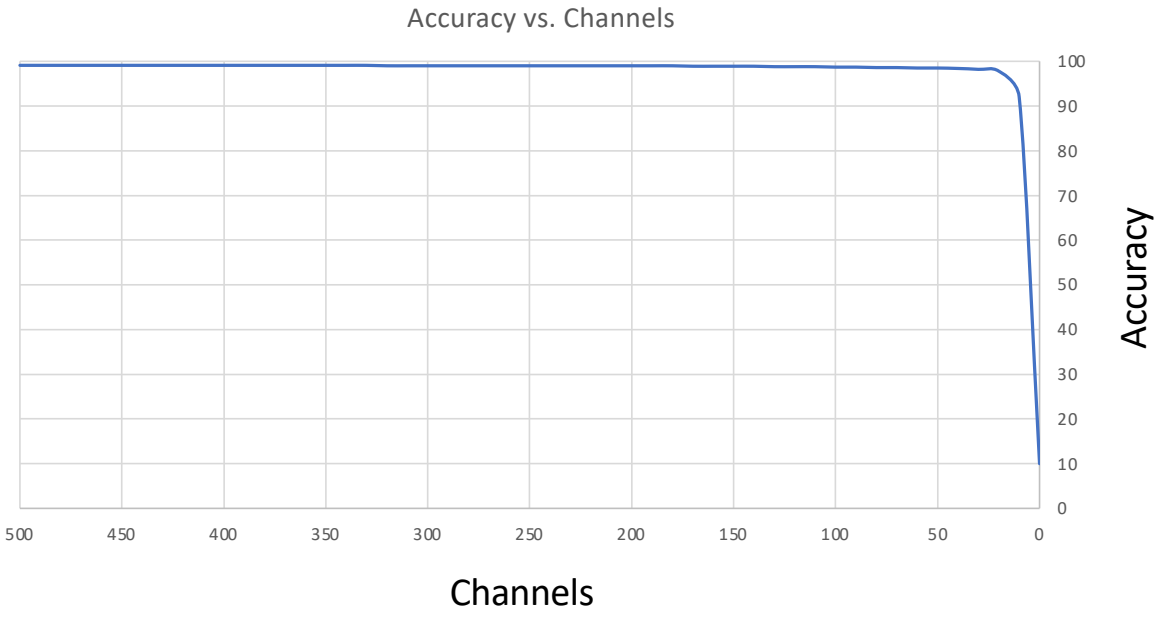
HLS AI Design Flow



Neural Network Architecture

- Most Neural Networks are architected for accuracy on servers
- Reducing the number of layers and channels in each layer
  - Small impact on accuracy (<1%)
  - Large impact on performance and efficiency (>90%)

Impact of Channel Count on Accuracy



Based on MNIST LeNet
Dense layer has 500 channels

Reducing Network Size Example

Original MNIST network

```
MAC operations:          12,353,000
Number of parameters:     4,915,080
Minimum data transfer:    4,941,854 words

Accuracy:     98.75%
```

Optimized MNIST network

```
MAC operations:             537,410
Number of parameters:       145,977
Minimum data transfer:      150,728 words

Accuracy:     98.46%
```

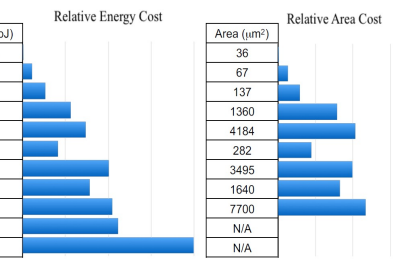
Quantization: Data Sizes and Operators

- Fixed point multipliers are about ½ the area of a floating-point multiplier
- Multipliers are proportional to the square of their inputs
- A 64-bit floating point multiplier is about 64 times larger than an 8-bit fixed point multiplier
- Data storage and movement scale linearly with size

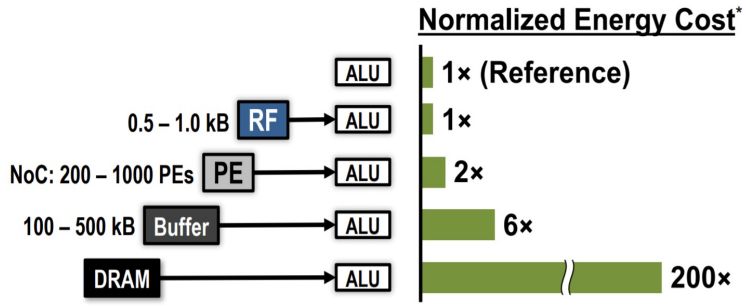
Source: Nvidia, DAC 2017

Cost of Operations

Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

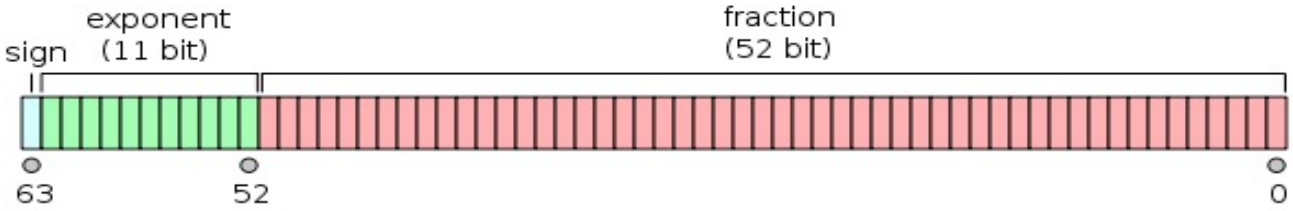


Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014

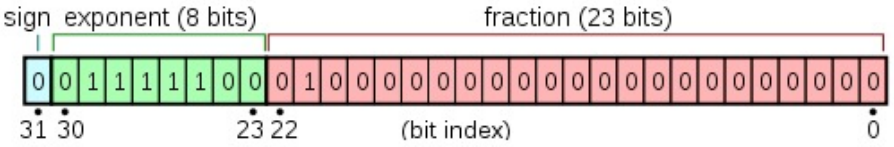


Fixed Point Representation

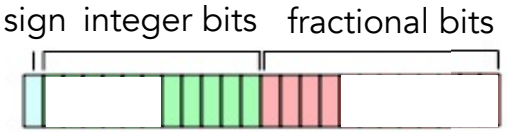
64-bit Float



32-bit Float



Fixed Point

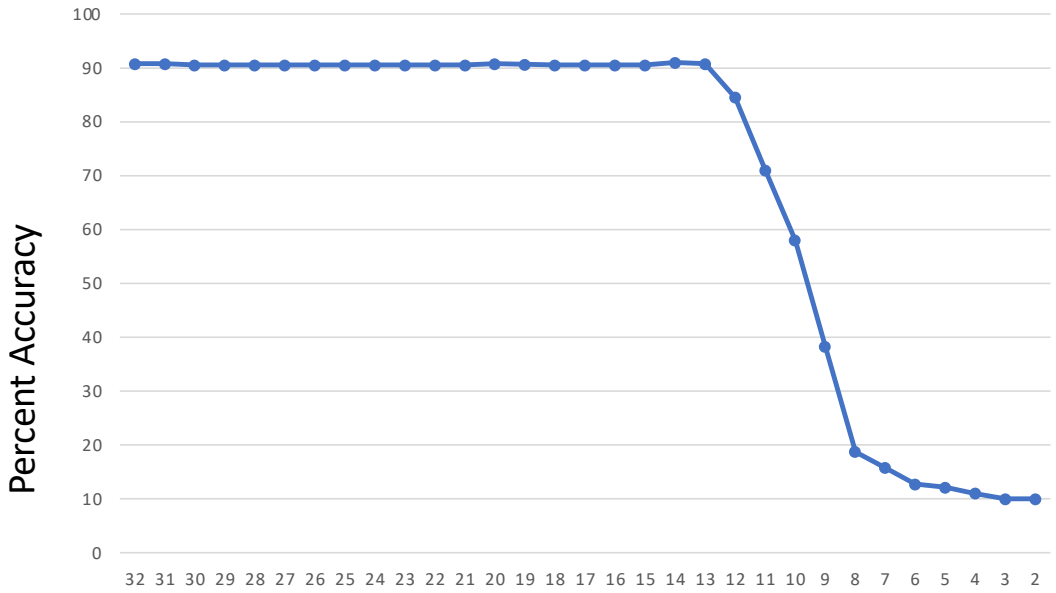


Any size you want

Quantizing Neural Networks

- Convert weights and features from floating point to fixed point
- Eliminate unused high-order bits
  - Removes constant 0 values from design
  - Many neural network values are normalized to near 0
    - May only need 4 or 5 integer bits
- Reduce fractional precision and measure impact on accuracy
  - Iterative process

Bitwidth vs Accuracy



Accuracy vs. Bit Width, Post-training Quantization

Fractional Bits \ Integer Bits	8	7	6	5	4	3	2	1	0
8	98.05	98.05	98.05	97.55	76.75	28.70	18.00	16.80	14.90
7	97.85	97.85	97.85	97.25	75.39	27.90	17.50	16.60	15.40
6	97.13	97.95	97.91	97.45	75.15	28.30	17.30	15.90	13.90
5	97.21	98.08	98.10	97.40	72.57	24.50	16.90	15.20	14.90
4	96.94	97.79	97.76	95.71	59.90	21.40	16.20	13.10	15.10
3	95.56	96.37	96.35	90.08	38.83	16.70	14.00	11.50	12.70
2	82.31	83.13	83.13	64.73	22.70	14.90	12.30	10.50	8.50
1	30.15	30.97	30.92	33.72	32.07	24.60	34.90	12.30	8.50
0	9.53	9.33	9.50	9.37	9.37	8.50	8.50	8.50	10.00

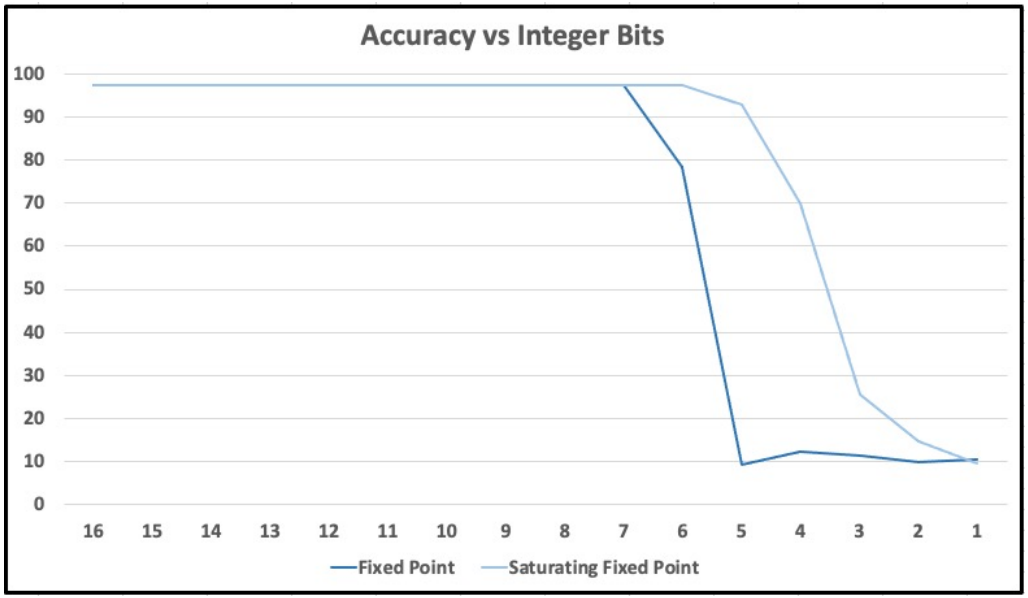
32 bit floating point accuracy is 98.05

Area/power for 32 bit floating point multiplier is ~20X more than a 10 bit fixed point multiplier

Saturating Math

- Floating point representations almost never overflow
  - 64 bit floating point represents up to 10^{308}
- Using reduced precision means overflows are more likely
  - Overflow truncation corrupts the result, and all subsequent calculations
- Saturating math stores the maximum value which can be represented when an overflow occurs
- For many neural networks when a number gets large the absolute magnitude is not important, just that the number is “large”

Saturating Math



Saturating math can reduce required representation size by 1 or 2 bits

Data Movement and Storage

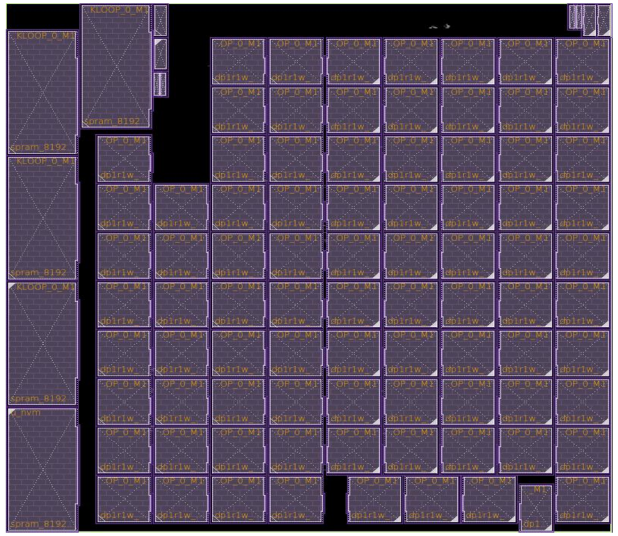
- Movement and storage of weights and features impacts performance and power
- Reducing numeric representation has a linear effect on storage costs
- For data movement, fully packing the bus with data is optimal
  - Buses are typically sized based on powers of two
  - For example, 16 bit representation is preferred to 17 bits
- While reducing the size of the representation usually negatively impacts accuracy, this can be offset by increasing layers or channels
  - This means changing the architecture of the neural network

Convolution Order of Operations

- Convolution algorithms access the input feature map and output array multiple times
- Early in the network the input data sets are typically smaller
- Later layers typically have larger input arrays
- Coordinating cache size with order of operations can optimize PPA

Caching and Buffering

- Minimizing accesses to external memory can improve performance and minimize power
- Memories tend to dominate area and power
- Data movement tends to limit performance
- If CNN data sets are too large to fit on-chip, careful data management can significantly improve design characteris



Inference Accelerator, post P&R

Accelerator Optimization

- The CNN will undergo significant modification between the ML framework and the hardware design
- This presents unique verification challenges

2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Verification Challenges

accellera SYSTEMS INITIATIVE

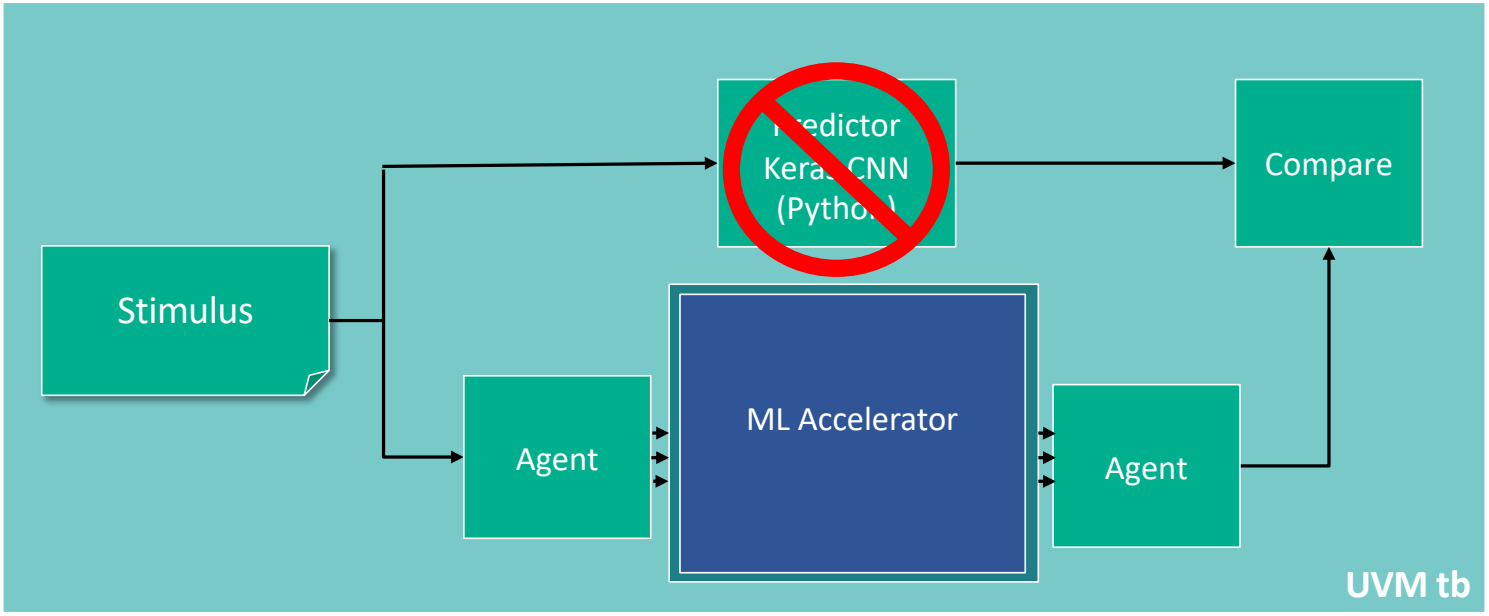
Verification of Inferencing Systems

- Need to verify:
  - Individual operators, multipliers, adders, etc.
  - Processing elements, Multiply/Accumulate (MAC) operations
  - Complete inferences
- Neural Networks are robust to failed individual operations
  - A single correct inference does not prove correctness of the implementation
  - A statistically significant number of inferences is required

Verification of Inferencing Systems

- Performance in logic simulation is prohibitively slow (28 hours for one inference in an object recognition algorithm)
  - And hardware acceleration is often not available early in the design cycle
- Verify at the abstract level and prove equivalence between representations at different design stages
  - This can be done between Python and C++, then C++ and RTL

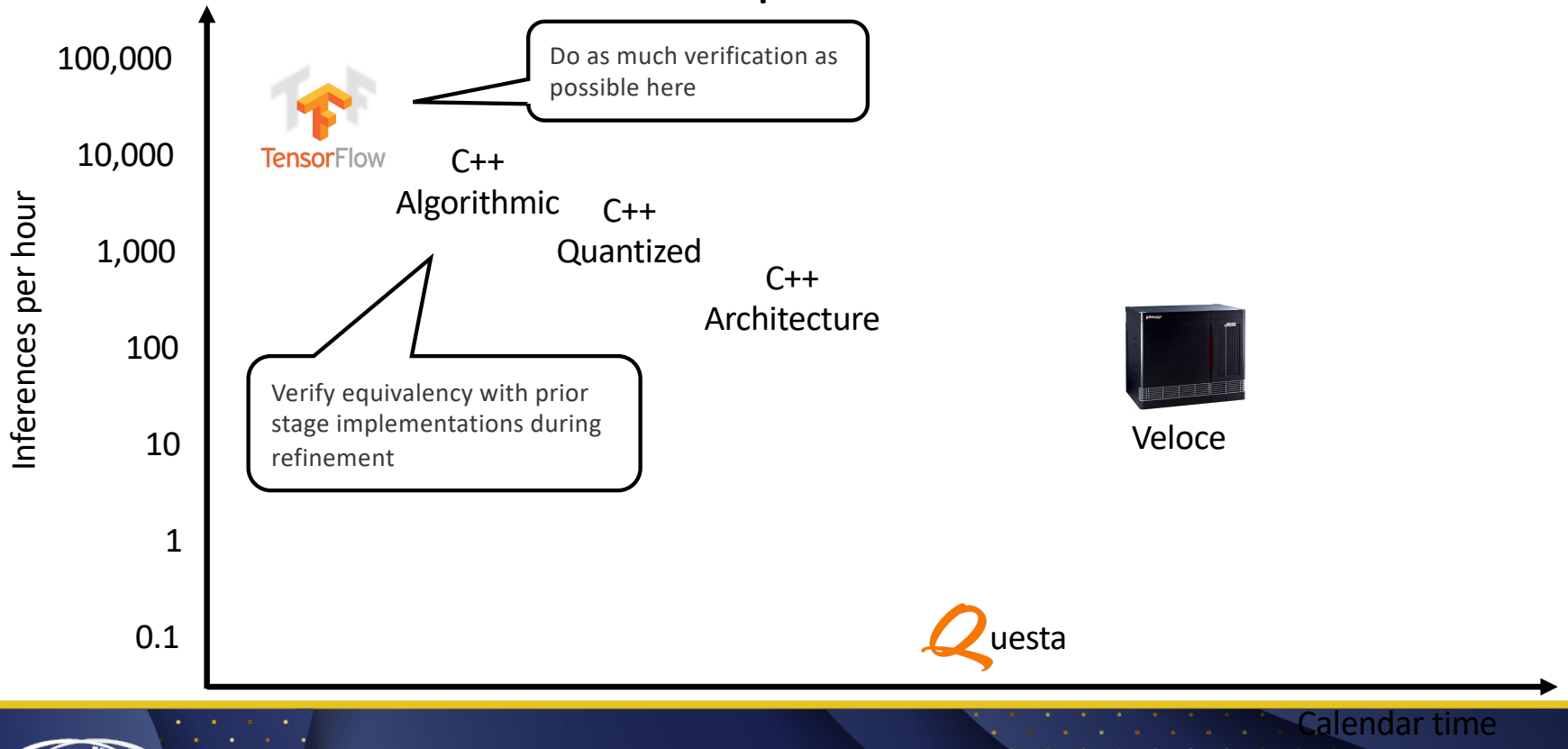
Traditional UVM Flow



Traditional UVM Flow

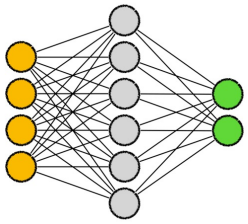
- Verilog implements modified CNN
  - Changes in layers/channels (these can be implemented in the predictor)
  - Changes in numeric representation
    - Float vs. fixed
    - Bit widths
    - Saturation/rounding
- Cannot directly compare outputs
- If there is a problem, debug is very hard

Verification landscape



Prove Equivalence at Each Step

Machine Learning Framework



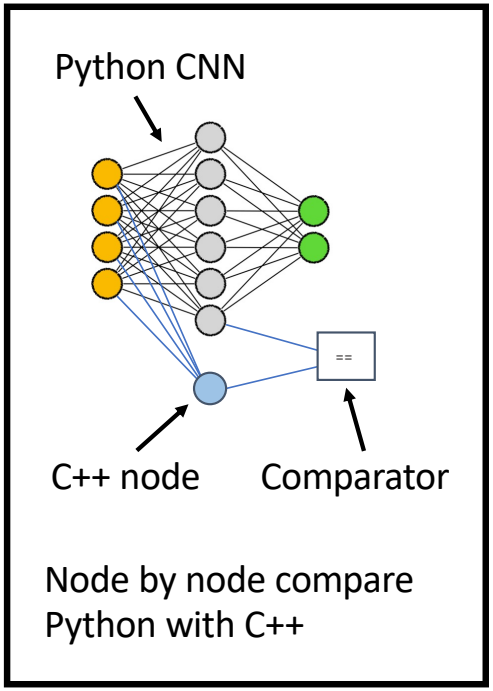
Neural Network Architecture

== C++ Algorithm == C++ Quantized Algorithm == C++ Architected Implementation

High-Level Synthesis

Synthesizable RTL

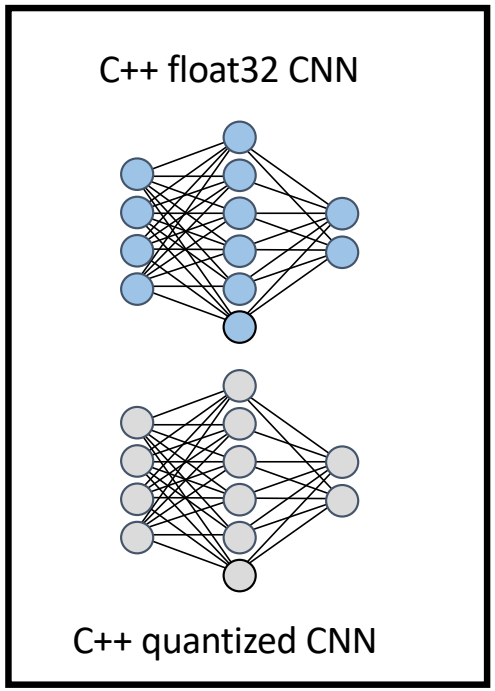
Python to C++ Consistency



Python to C++

- Run C++ node in parallel with Python node
- Both nodes use common float types
- Differences should be only order of computation rounding error
- Import C++ function into Python
  - Several ways to do this: ctypes, CFFI, PyBind11, Cython
- Repeat for subsequent nodes, then layers, then complete network

C++ to Quantized Model Consistency



Quantization

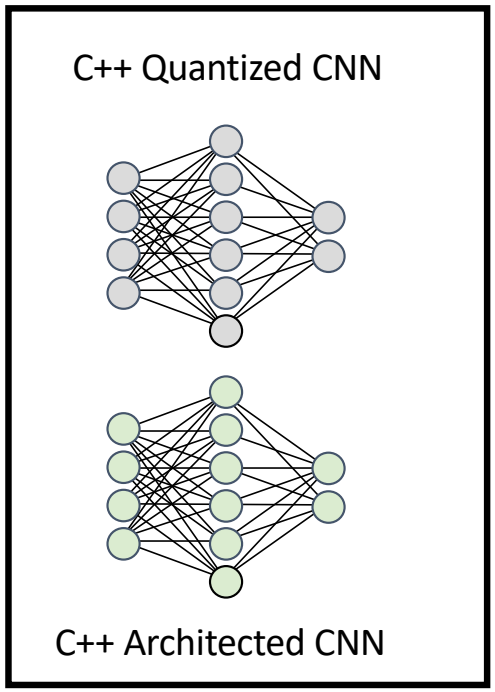
- Run C++ node in parallel with the quantized node
- Quantized implementation should be identical to C++ algorithmic except for data types
  - Verify/debug one thing at a time
- Nodes use different types
  - Float vs. fixed point, reduced bit-width (ac data types)
- Differences will exist, and may be large
  - When in range, single operations will be within rounding error
  - Outside of range will be saturated

Quantized Model Must be Verified

- Need to run large number of inferences
  - Predictions will be different from Python or C++ algorithmic model
- Determine if CNN accuracy is acceptable
  - Modify network/layers/channels as needed and repeat

- One day ML frameworks will support quantized numbers
  - Qkeras, Larq, and Hawq are examples of extensions that support quantization
  - Currently, works for TPUs, but not expressive enough for bespoke accelerator
  - Abstract model must ***exactly*** match the Verilog to be implemented

C++ Quantized to C++ Architecture Consistency

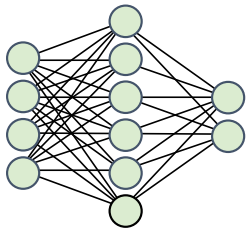


Architecture

- Run Quantized node with Architected node
- Quantized and Algorithmic nodes should differ only by order of operation rounding errors
- Nodes use same types
  - Fixed point, reduced bit-width

Verification – before HLS

C++ Architected CNN



Static Design Checks

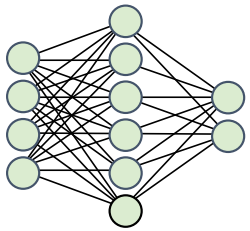
Static code analysis and synthesis checks. Find coding errors and problem constructs

Coverage Analysis

Determine completeness of test cases. Statement, branch and expression coverage as well as covergroups, coverpoints, bins and crosses

C++ to RTL consistency

C++ Architected CNN



Formal

Using formal techniques, prove as much equivalency as possible

UVM

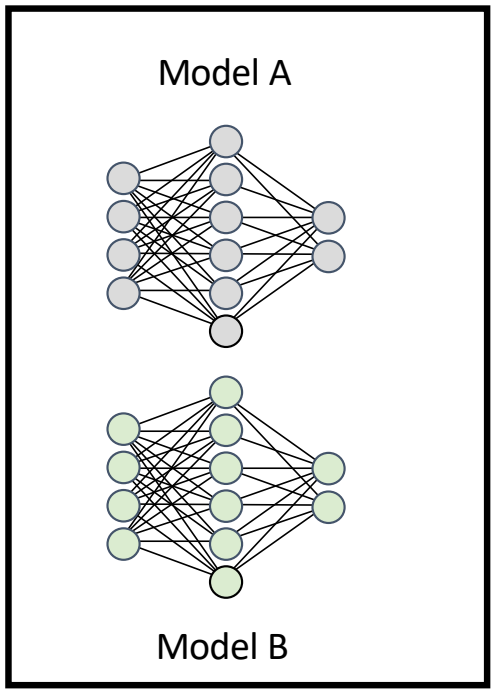
Architected C++ is used as a predictor for RTL verification

RTL Coverage

Determine remaining verification effectiveness through RTL coverage metrics

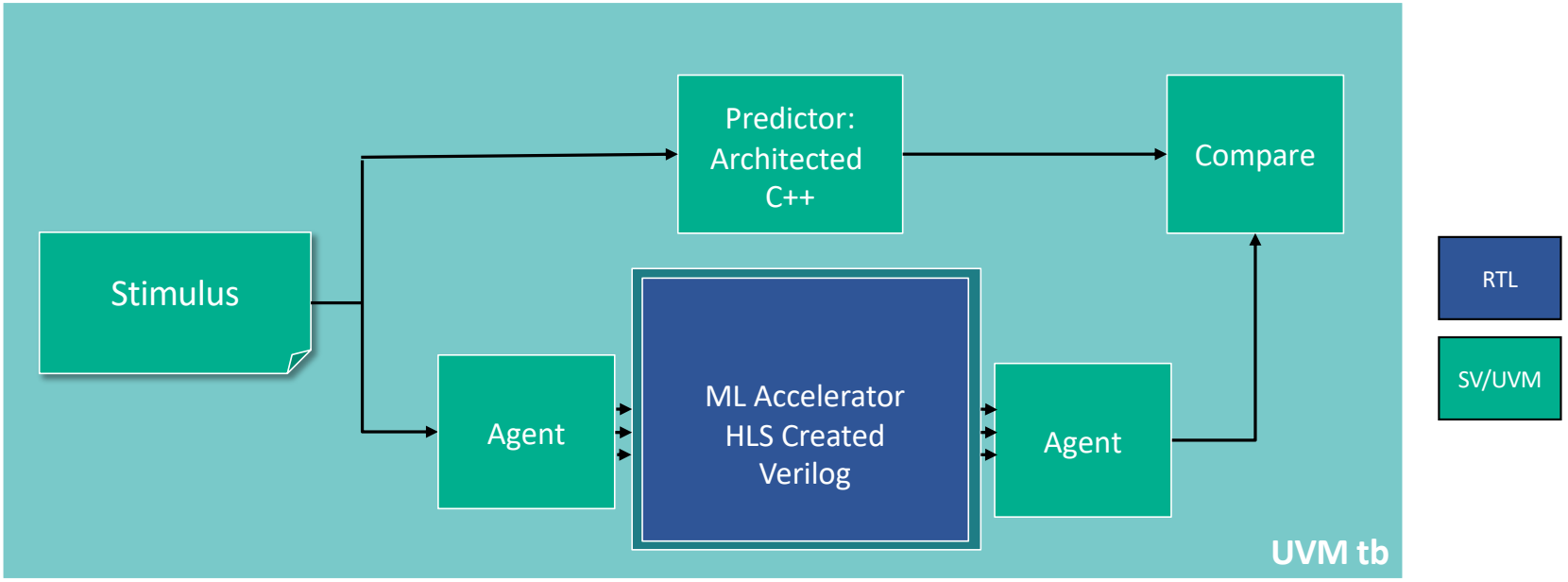
```
always @(posedge clk_i or negedge rstn_i) begin
 if(!rstn_i)  begin
   ring_cnt <= 2'b00 ;
   grnt_o <= 4'b0;
 end
 else begin
    if(req_i == 4'b000)
     grnt_o <= 4'b0000;

  if(timer_start)  begin

   if(req_i[3:0] == 4'b0001)
    grnt_o <= 4'b0001;
   else if (req_i == 4'b0010)
    grnt_o <= 4'b0010;
   else if (req_i == 4'b0100)
    grnt_o <= 4'b0100;
   else if (req_i == 4'b1000)
    grnt_o <= 4'b1000;
   else begin

     if (timer_exp)
       ring_cnt <= ring_cnt +1 ;


     if(ring_cnt == 2'b00) begin
       if(req_i[0])
         grnt_o <= 4'b0001;
       else
       ring_cnt <= ring_cnt +1 ;
     end

     if(ring_cnt == 2'b01) begin
       if(req_i[1])
         grnt_o <= 4'b0010;
       else
         ring_cnt <= ring_cnt +1 ;
     end

     if(ring_cnt == 2'b10) begin
       if(req_i[2])
         grnt_o <= 4'b0100;
       else
       ring_cnt <= ring_cnt +1 ;
     end
```

Debug – When Things Go Wrong

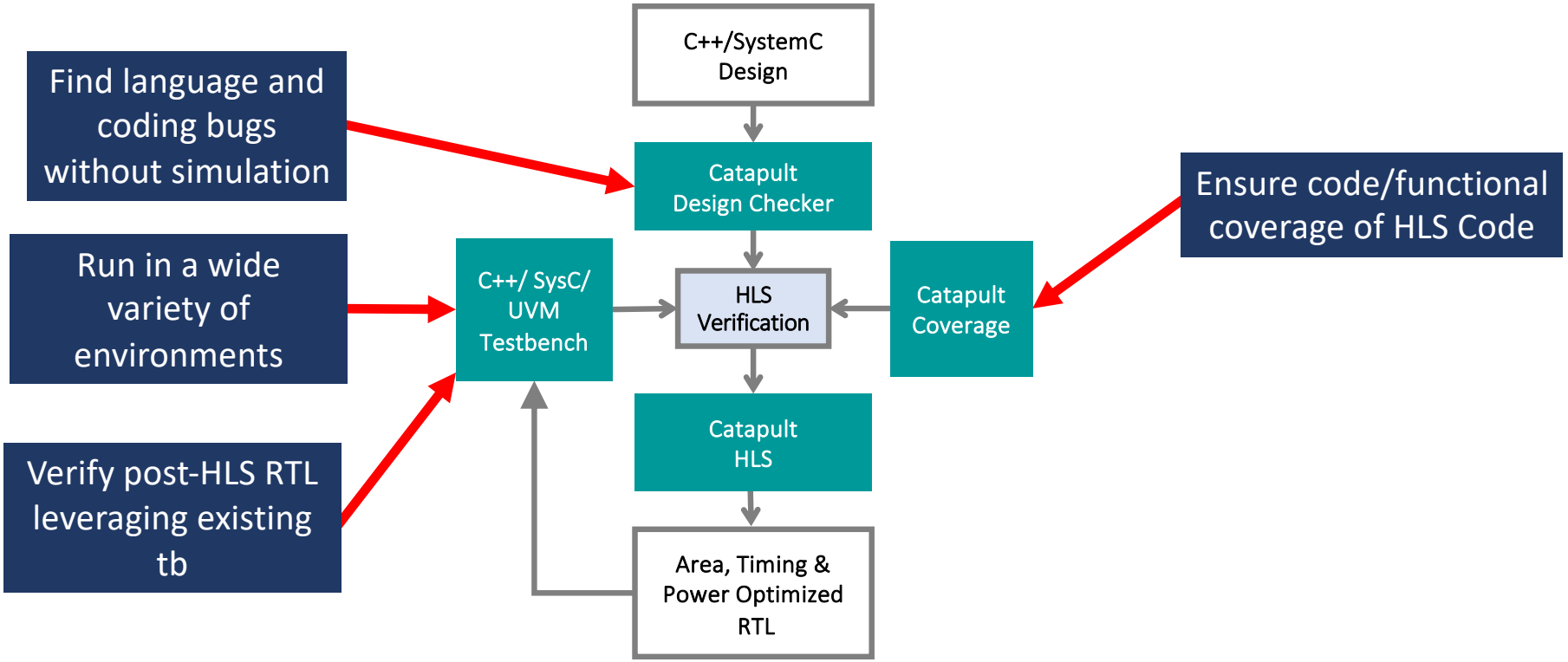


- Log all intermediate values to memory or log file
  - This includes output from each layer
- Have scripts that can compare intermediate values from different model representations
  - This identifies the first point of divergence between models
  - Immediately find layer and node where problem resides
- Intermediate values from the Python can be recorded to a file for comparison

HVL UVM Flow


Predictor:
Architected
C++
Compare
Stimulus
Agent
ML Accelerator
HLS Created
Verilog
Agent
UVM tb
RTL
SV/UVM

Verification in HLS Flow



Conclusion

- Moving from Python to RTL in a single step introduces a significant verification problem
  - Inferencing algorithms do not produce bit-level equivalency when accelerated
  - Requires many inferences to verify accuracy of implementation
  - Simulation performance is too slow, emulation or FPGA prototypes are usually not available
- High-Level Synthesis introduces an intermediate C++ model
  - Verify the algorithm at the Python level
  - Prove equivalency between subsequent model stages

Questions or Comments

?? || //

accellera
SYSTEMS INITIATIVE

Thank You

Petri Solanti, Field Applications Engineer, Petri.Solanti@Siemens.com
Russell Klein, Program Director, Russell.Klein@Siemens.com

accellera
SYSTEMS INITIATIVE