2023
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SAN JOSE, CA, USA
FEBRUARY 27-MARCH 2, 2023

Verification of Inferencing Algorithm Accelerators

Russell Klein
Petri Solanti
SIEMENS

accellera
SYSTEMS INITIATIVE

Agenda

- AI Accelerators
- High-Level Synthesis
- Bespoke Accelerator Optimization
- Verification
  - From Python to RTL
- Example Algorithm
  - Wakeword verification flow

2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SAN JOSE, CA, USA
FEBRUARY 27-MARCH 2, 2023

AI Accelerators

accellera SYSTEMS INITIATIVE

Deploying Inferencing Systems, Where and How

Where

The Cloud

A Gateway

The Edge

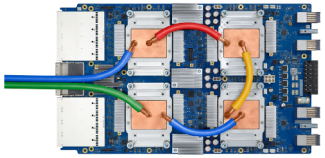
Highest performance and efficiency are achieved with specialized ASIC implementation running on the edge

Higher Latency/Lower development costs

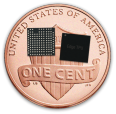
How

CPU

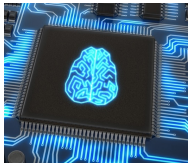
GPU



TPU/NPU



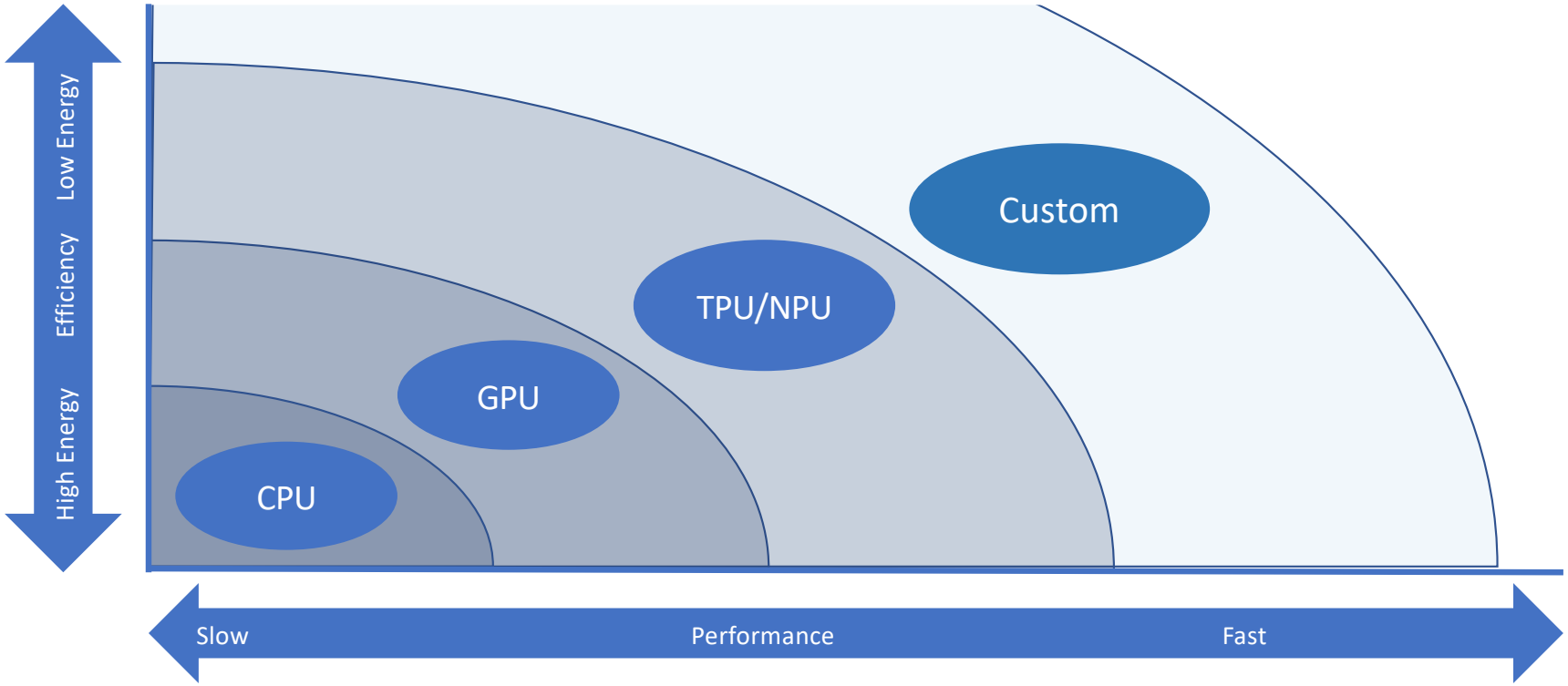
Edge TPU



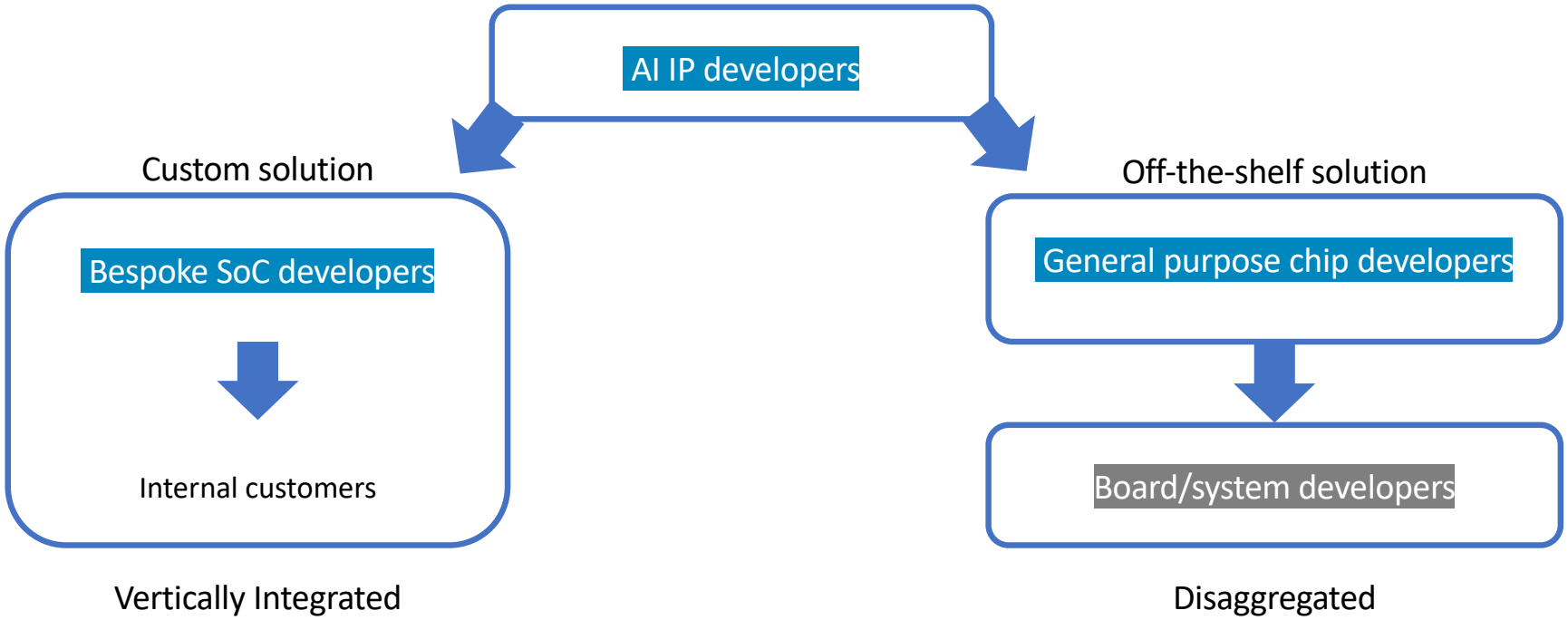
FPGA or ASIC

Higher specialization/Lower energy

Inference Execution (on-chip)



Accelerated AI SoC Eco-system



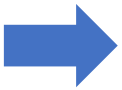
AI Accelerator Verification Challenges

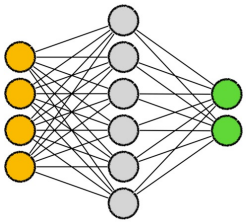
- CPU, GPU, NPU, TPU
  - Verify algorithm implementation runs on IP
  - Verify that IP is correctly integrated
  - IP is assumed to be correct from the IP provider
- Bespoke accelerator
  - Verify the algorithm runs on the accelerator
  - Verify the accelerator is correctly integrated
  - Verify the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

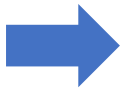
From Python to Hardware

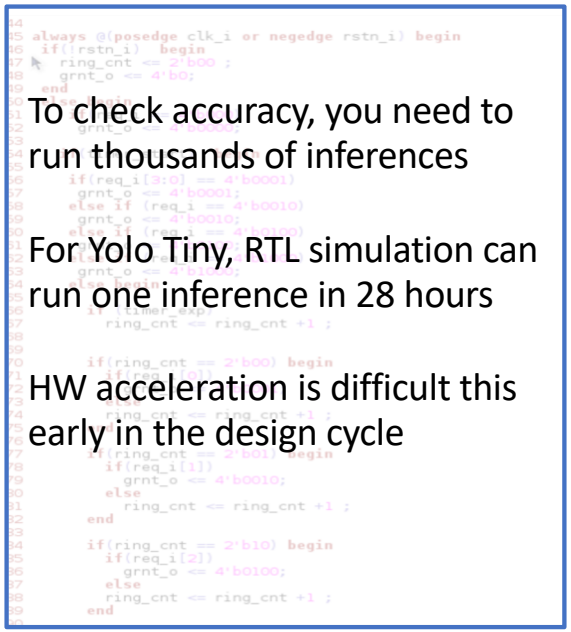
Machine Learning Framework

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

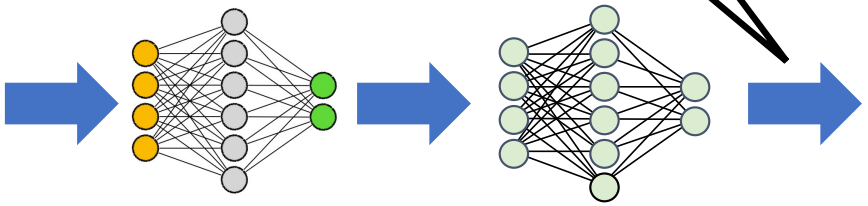
Python

Verilog

A Better Path

Machine Learning Framework

Iterate to find optimal NN architecture



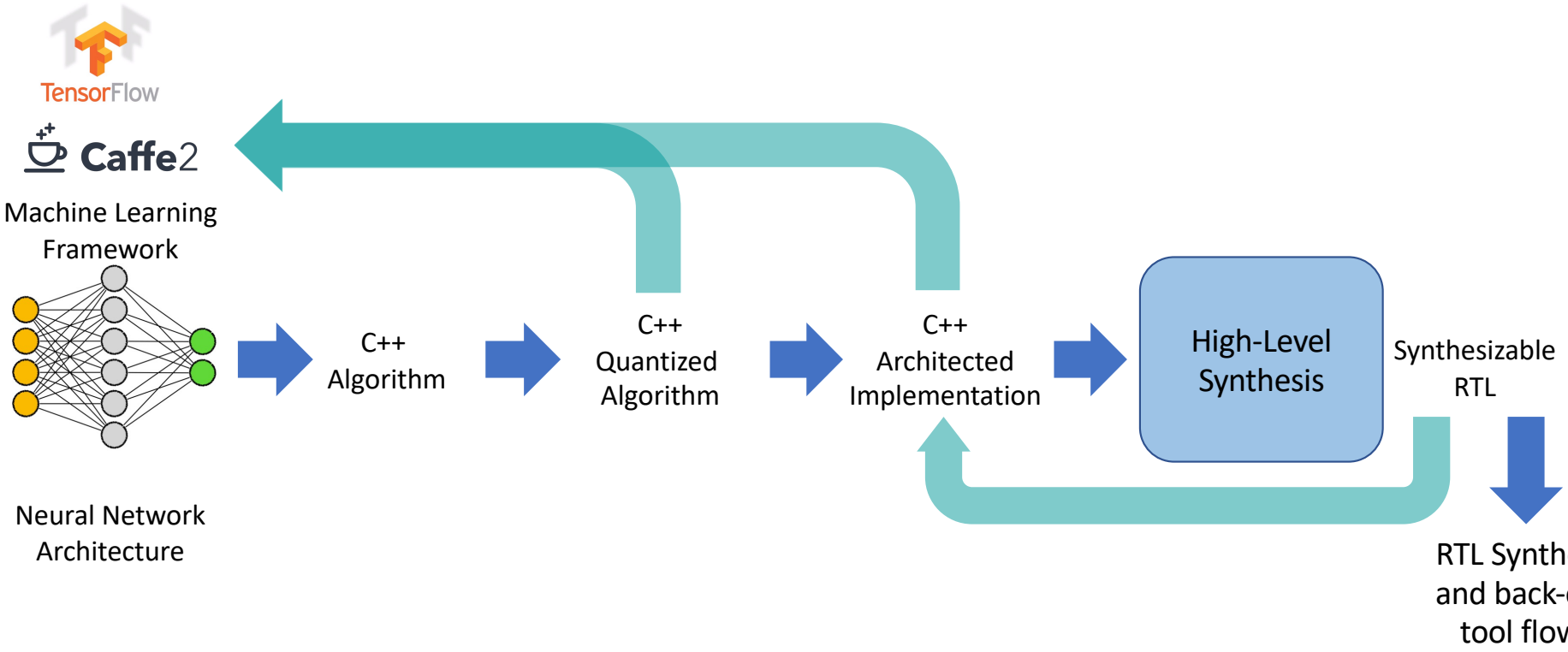
Python

C++

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

Verilog

HLS AI Design Flow



2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SAN JOSE, CA, USA
FEBRUARY 27-MARCH 2, 2023

High-Level Synthesis

accellera SYSTEMS INITIATIVE

What is 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

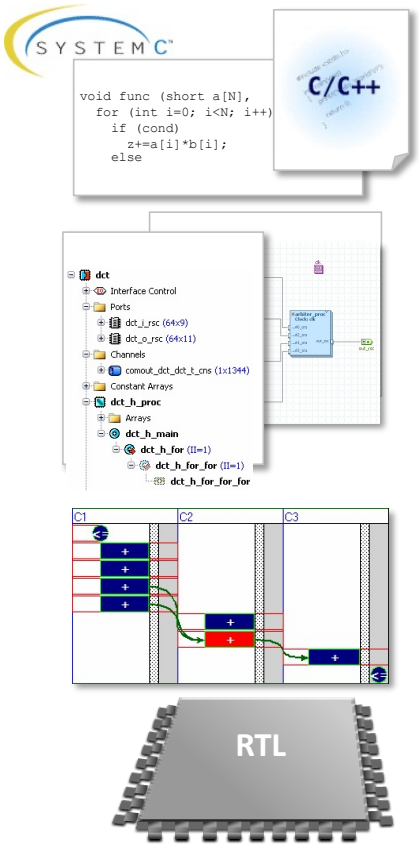
High-Level Synthesis

```
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

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



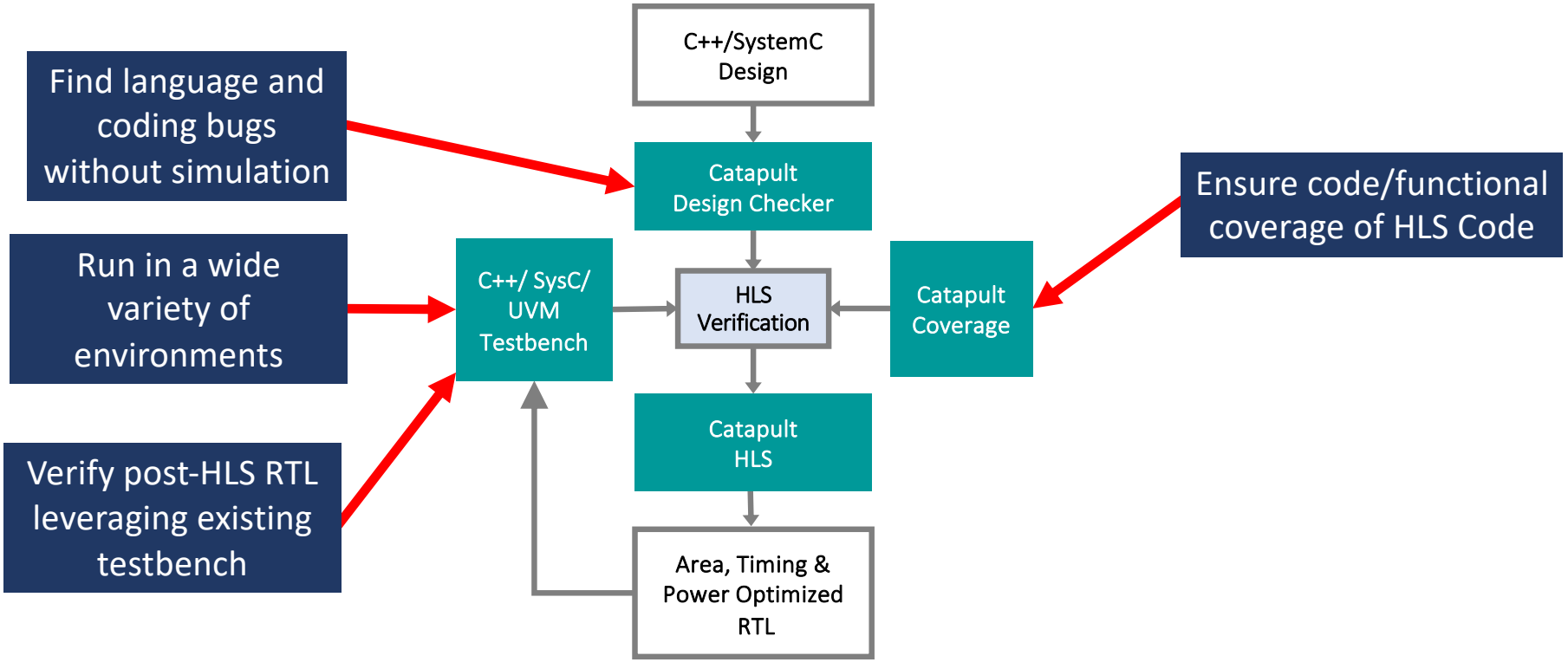
High-Level Synthesis Benefits

- 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

Verification in HLS Flow



2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SAN JOSE, CA, USA
FEBRUARY 27-MARCH 2, 2023

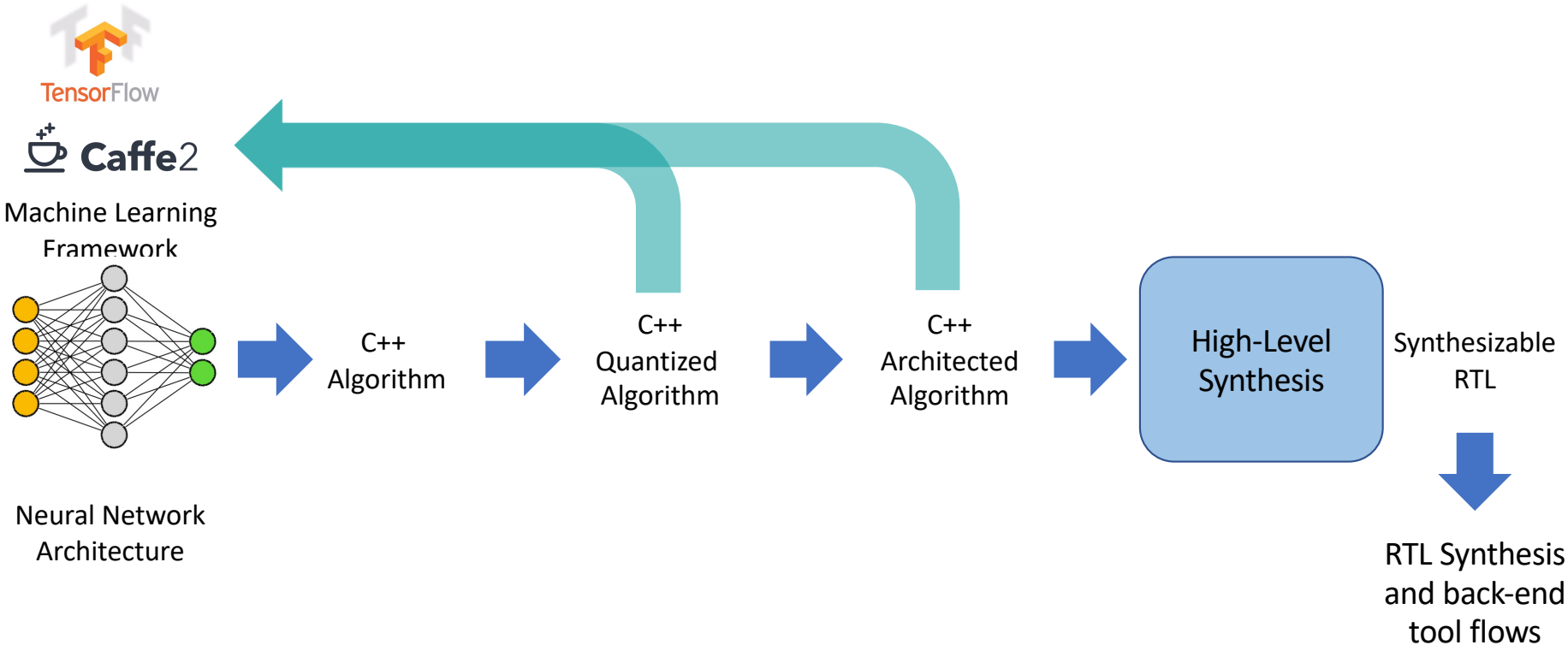
Accelerator Optimization

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

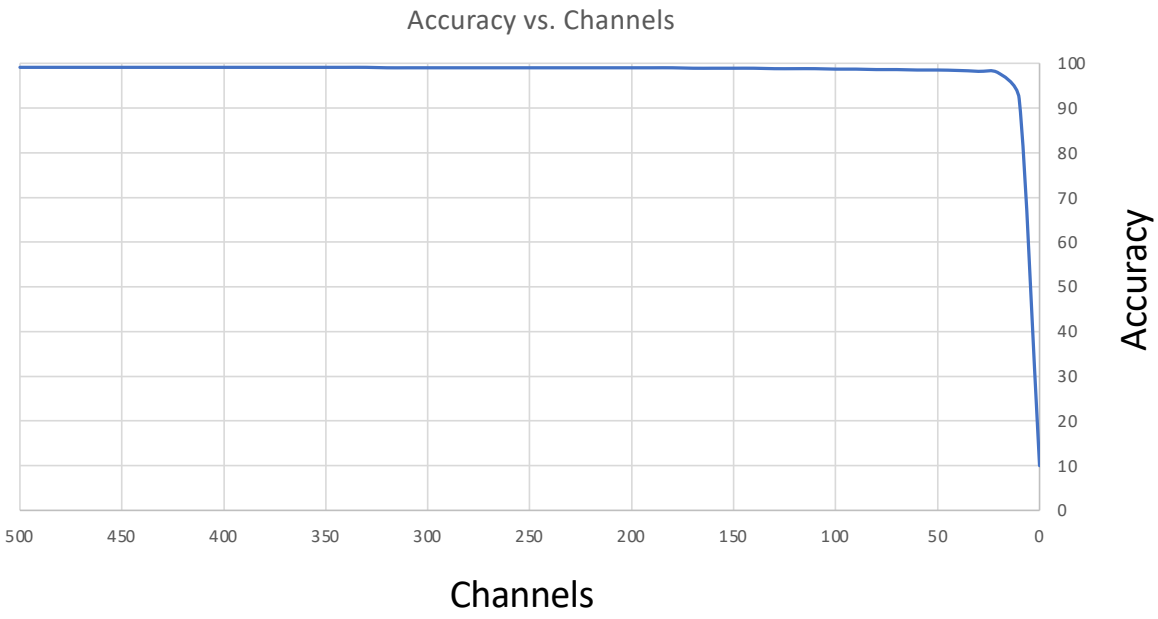
HLS AI Design Flow



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

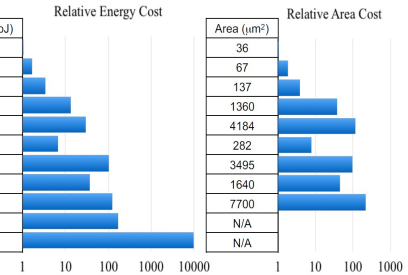
```
MAC operations:             537,410
Number of parameters:       145,977
Minimum data transfer:      150,728 words

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

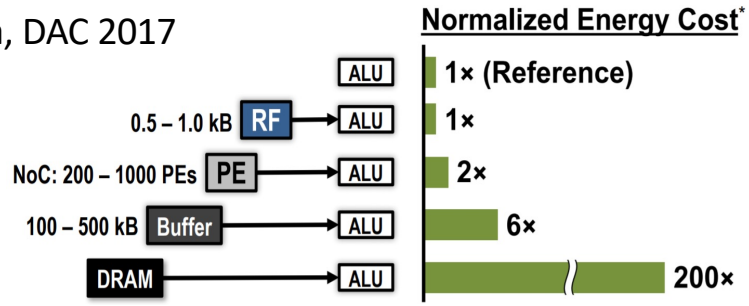
Cost of Operations

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



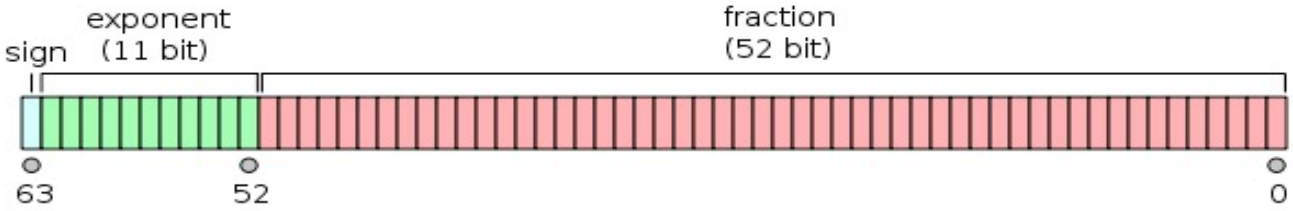
Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014

Source: Nvidia, DAC 2017

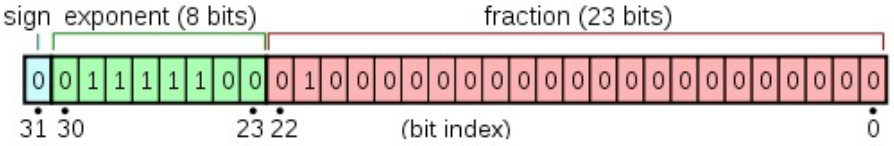


Fixed Point Representation

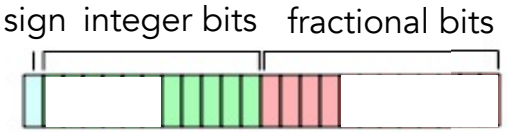
64-bit Float



32-bit Float



Fixed Point

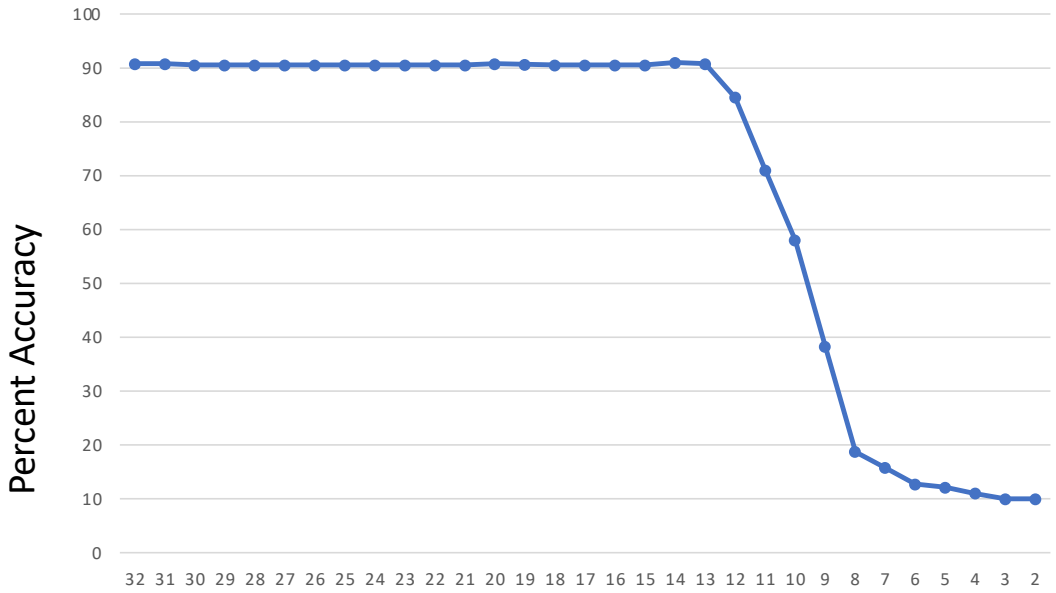


Any size you want

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



Wake word Algorithm

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

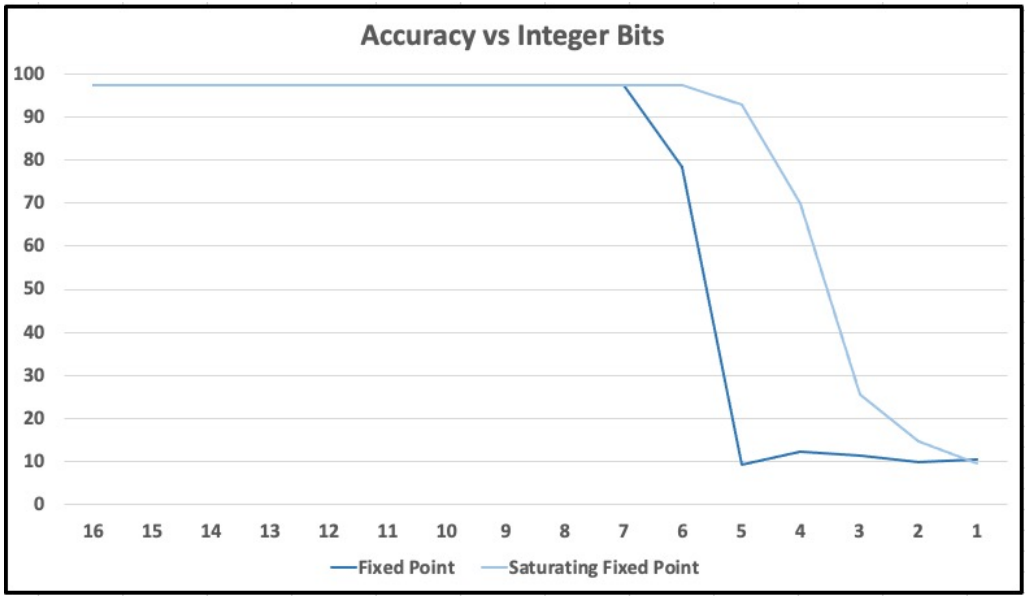
32 bit floating point accuracy is 98.05

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

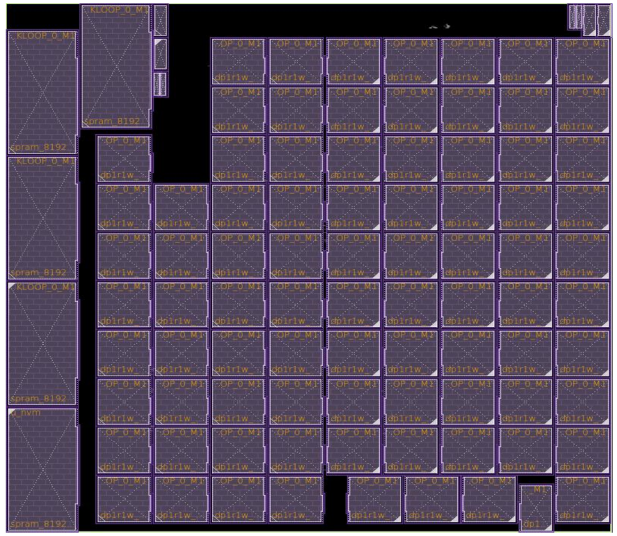
- Movement and storage of weights and features impacts performance and power
- Reducing numeric representation has a linear effect on storage costs
- For data movement, fully packing the bus with data is optimal
  - Buses are typically sized based on powers of two
  - For example, 16-bit representation 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 Layout

Accelerator Optimization

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

2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SAN JOSE, CA, USA
FEBRUARY 27-MARCH 2, 2023

Verification Challenges

accellera SYSTEMS INITIATIVE

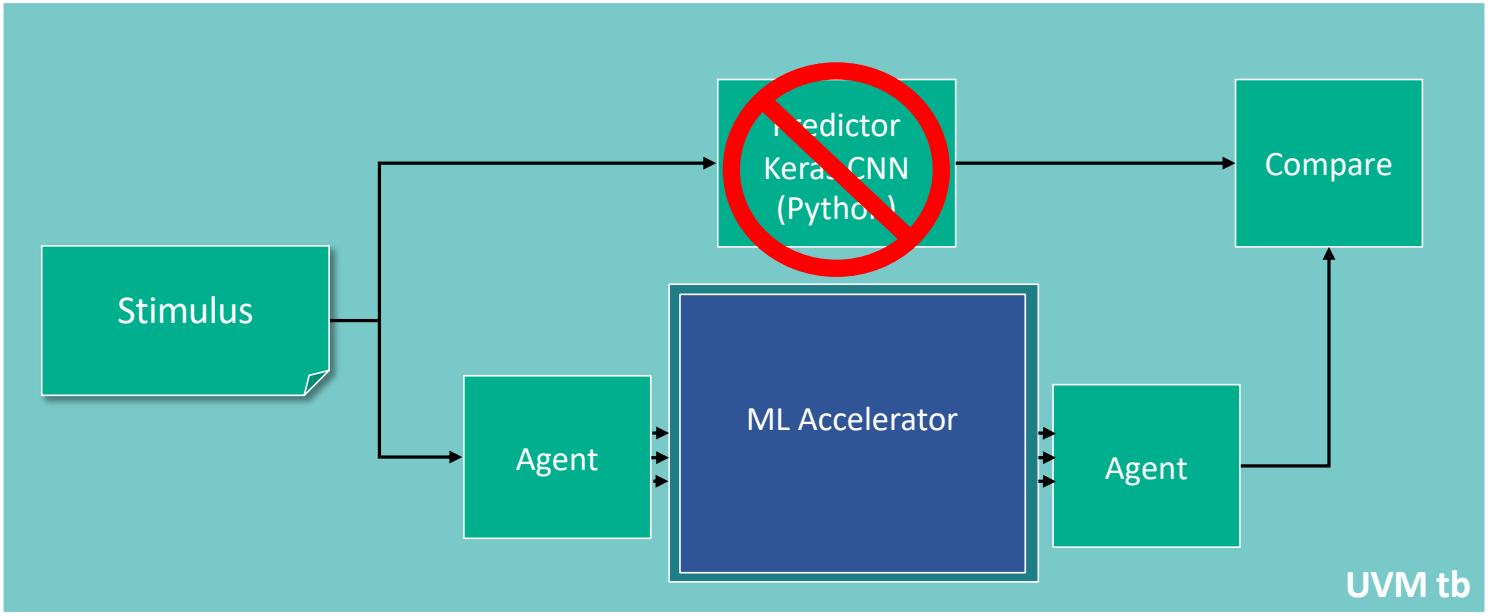
Verification of Inferencing Systems

- 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

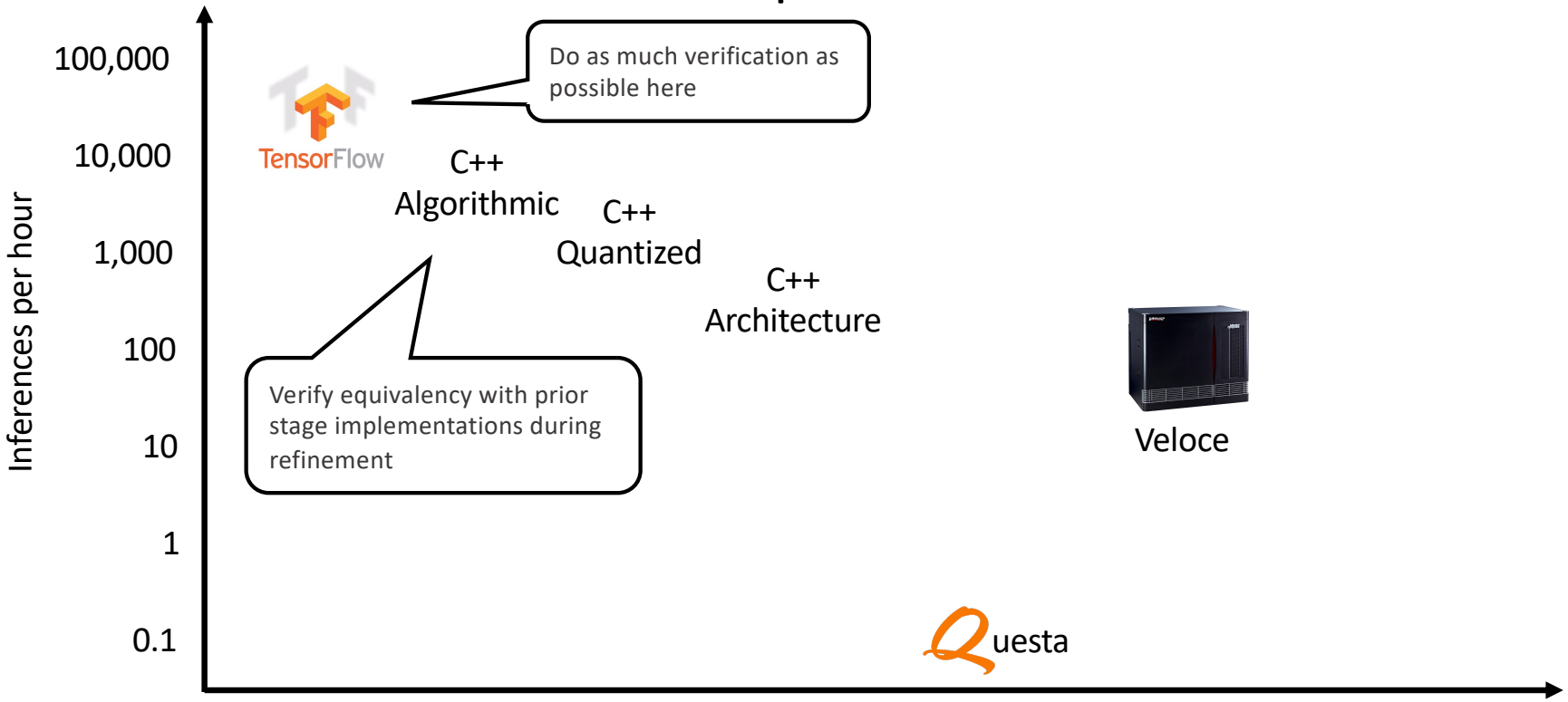
Traditional UVM Flow



Traditional UVM Flow

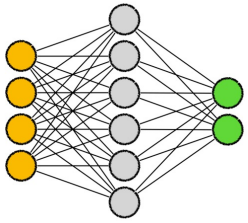
- Verilog implements modified CNN
  - Changes in layers/channels (these can be implemented in the predictor)
  - Changes in numeric representation
    - Float vs. fixed
    - Bit widths
    - Saturation/rounding
- Cannot directly compare outputs
- If there is a problem, debug is very hard

Verification Landscape



Prove Equivalence at Each Step

Machine Learning Framework



Neural Network Architecture

==

C++ Algorithm

==

C++ Quantized Algorithm

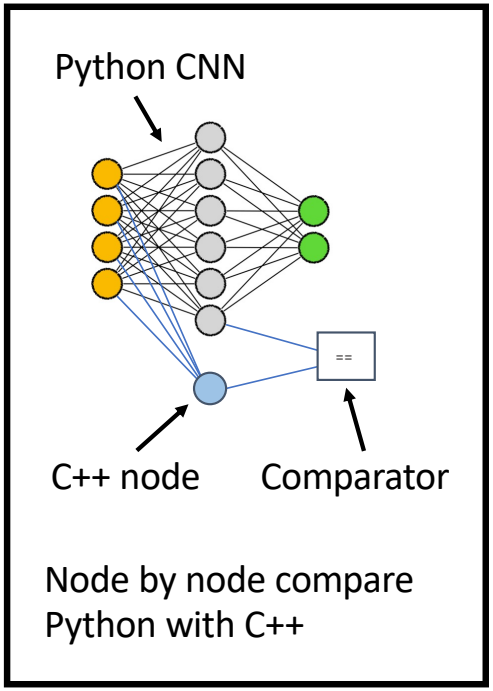
==

C++ Architected Implementation

High-Level Synthesis

Synthesizable RTL

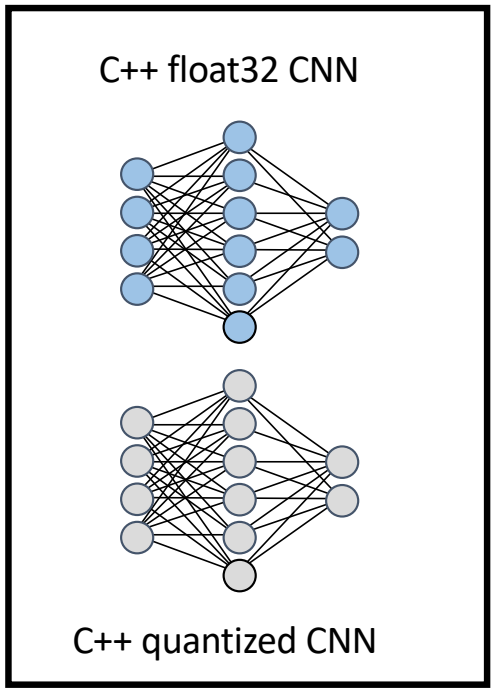
Python to C++ Consistency



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

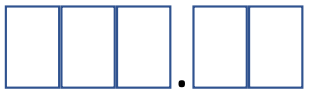


- Run C++ node in parallel with the quantized node
- Quantized implementation should be identical to C++ algorithmic except for data types
  - Verify/debug one thing at a time
- Nodes use different types
  - Float vs. fixed point, reduced bit-width (ac data types)
- Differences will exist, and may be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

C++ to Quantized Model Consistency

- Quantization errors
  - Change in value from the original floating-point number
- Saturation errors
  - Values that exceed the range of the representation will saturate
- Accumulated errors
  - As operations are performed on quantized numbers, the quantization errors can compound
  - Rounding (as opposed to truncating) can reduce this

C++ to Quantized Model Consistency



- Ranges from 111.11 (-4.0) to 011.11 (3.75)
- Fractional precision is .25

Floating point math: 1.377 + 1.377 + 1.377 + 1.377 = 5.508

Fixed point math: 1.377 => 001.10, which is 1.5
001.10 + 001.10 + 001.10 + 001.10 = 110.00 (or 6) error from floating-point is 0.492
maximum possible error is ½ of fractional precision * number of operations

The signed fixed-point interpretation of 110.00 is -2
Using saturating math, this result would be 3.75

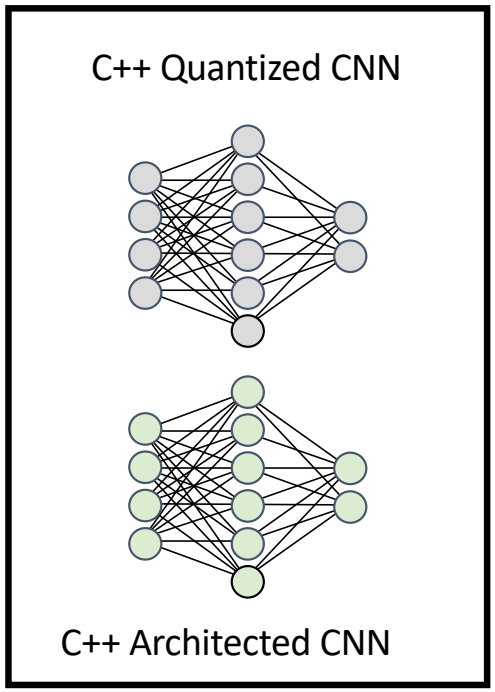
The error in a comparison with the floating-point algorithm would be 1.758, which is a correct implementation

Quantized Model Must be **Validated**

- 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

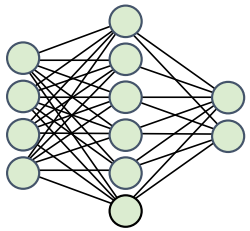


Architecture

- 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

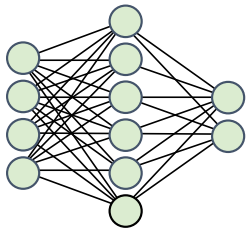
Static code analysis and synthesis checks. Find coding errors and problem constructs

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

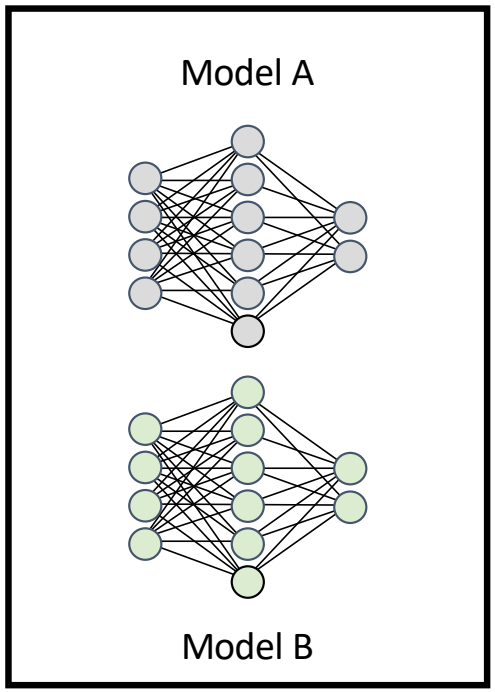
Architected C++ is used as a predictor for RTL verification

RTL Coverage

Determine remaining verification effectiveness through RTL coverage metrics

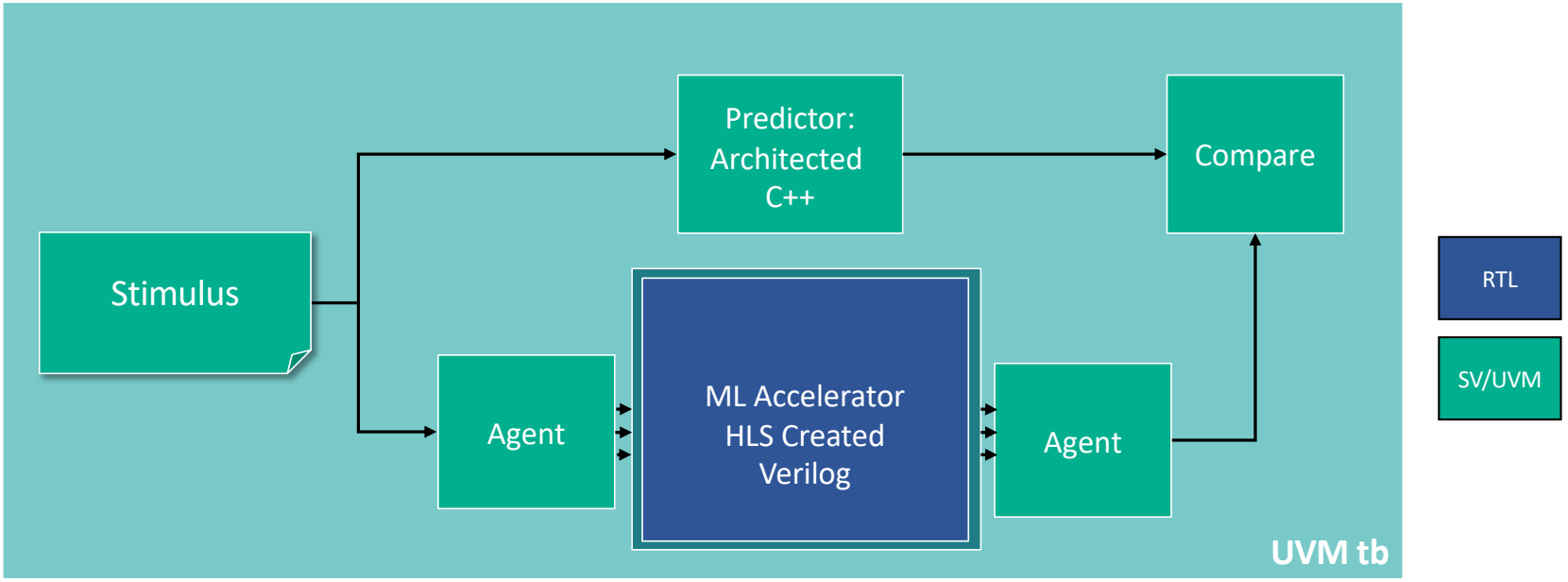
```
always @(posedge clk_i or negedge rstn_i) begin
 if(!rstn_i)  begin
   ring_cnt <= 2'b00 ;
   grnt_o <= 4'b0;
 end
 else begin
    if(req_i == 4'b000)
     grnt_o <= 4'b0000;

  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

2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SAN JOSE, CA, USA
FEBRUARY 27-MARCH 2, 2023

Wakeword Example

accellera SYSTEMS INITIATIVE

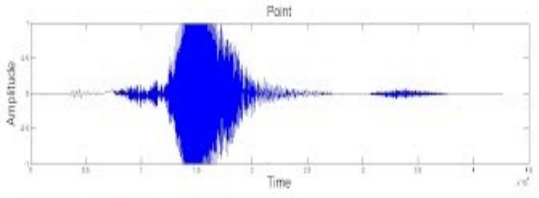
Wakeword Algorithm

- Monitors audio for keywords, performs some action when recognized
  - Like “Hey Google” or “Alexa”
- Runs continuously, so needs to be efficient
  - Essential when system is battery powered
- Trained with Tensorflow speech commands data set
  - http://download.tensorflow.org/data/speech_commands_v0.02.tar.gz
- From that we selected utterances from “zero” to “nine”
  - Like the MNIST digit recognition, but spoken words instead of handwritten
- Base algorithm (the inspiration) came from:
  - 'cnn-one-fstride4’ from 'Convolutional Neural Networks for Small-footprint Keyword Spotting': http://www.isca-speech.org/archive/interspeech_2015/papers/i15_1478.pdf

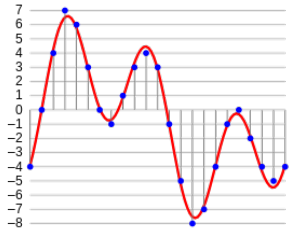
Wakeword Algorithm

- Processes one second samples of Pulse Code Modulation audio
  - 16,000 16-bit samples per second
- Algorithm is run every 50 ms.
  - Add 320 samples to rolling window of PCM audio data
- Processing must complete in 50 ms.
- Audio is preprocessed with an MFCC function
- The resulting spectrogram is used as a feature-map for a CNN

Wakeword Audio Pre-processing



Audio input

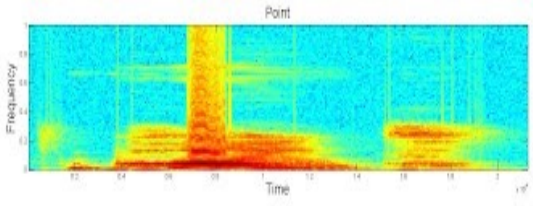


Quantization

-4	0	4	7	

Integer array
(16k x 16 bits)

MFCC()



Spectral Array

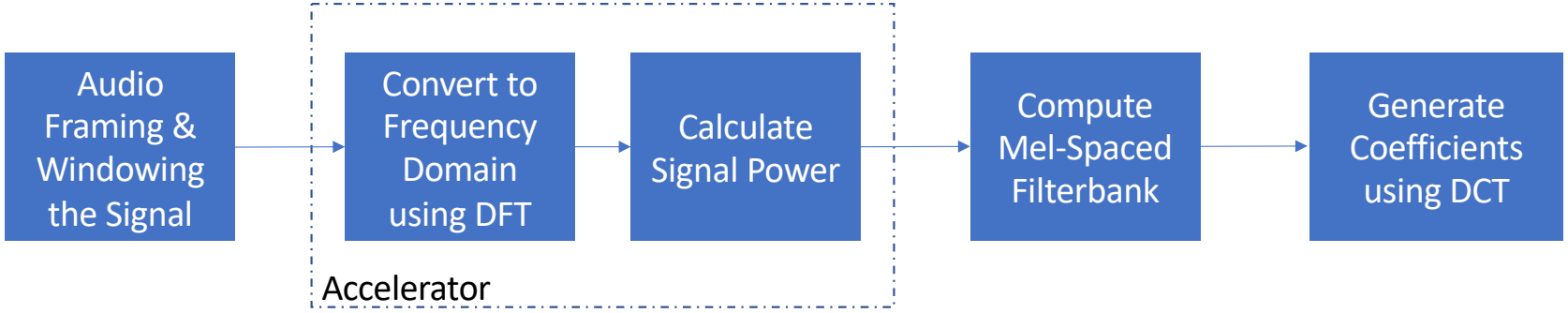
0.123	0.456	-0.872	0.567	
0.324	0.547	0.376	-0.231	
0.846	0.183	0.834	0.937	
0.625	0.737	0.746	0.827	

Float array
(101 x 20 x 32-bits)

To Neural Network
as feature map for
training and inferencing

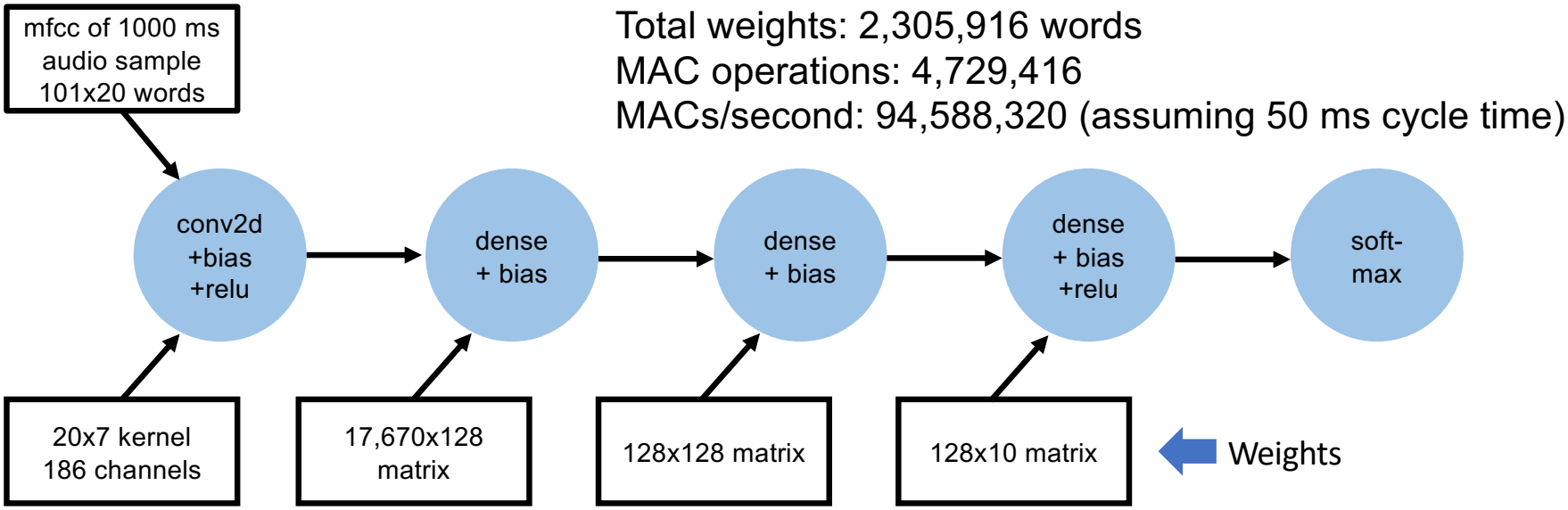
Mel Frequency Cepstral Coefficients (MFCC)

- Audio representation format commonly used for feature extraction.
- Mel scale mimics human perception of sound.
- Widely used in machine learning and speech processing techniques.
- Steps to obtain coefficients:



https://github.com/ddbourgin/numpy-ml/blob/master/numpy_ml/preprocessing/dsp.py

Wakeword Neural Network



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

Wakeword Profile

Weight		Self Weight	Symbol Name
158.00 ms	100.0%	0 s	▼wakeword (95781)
158.00 ms	100.0%	0 s	▼Main Thread 0x1af28c6
154.00 ms	97.4%	0 s	▼start libdyld.dylib
154.00 ms	97.4%	0 s	▼main wakeword
154.00 ms	97.4%	0 s	▼test_wakeword() wakeword
80.00 ms	50.6%	0 s	▼sw_inference(float*, float*, float*) wakeword
57.00 ms	36.0%	0 s	▼sw_auto_infer(float*, int, float*) wakeword
29.00 ms	18.3%	29.00 ms	dense_sw(float*, float*, float*, float*, int, int, int, int, int) wakeword
28.00 ms	17.7%	28.00 ms	conv2d_sw(float*, float*, float*, float*, int, int, int, int, int, int, int, int) wakeword
23.00 ms	14.5%	0 s	▶load_memory(float*) wakeword
73.00 ms	46.2%	2.00 ms	▼mfcc(float*, float*) wakeword
71.00 ms	44.9%	71.00 ms	power_spectrum(double*, std::__1::complex<double>*, double*) wakeword
1.00 ms	0.6%	0 s	▶read_wavefile(char*, float*) wakeword
4.00 ms	2.5%	0 s	▶_dyld_start dyld

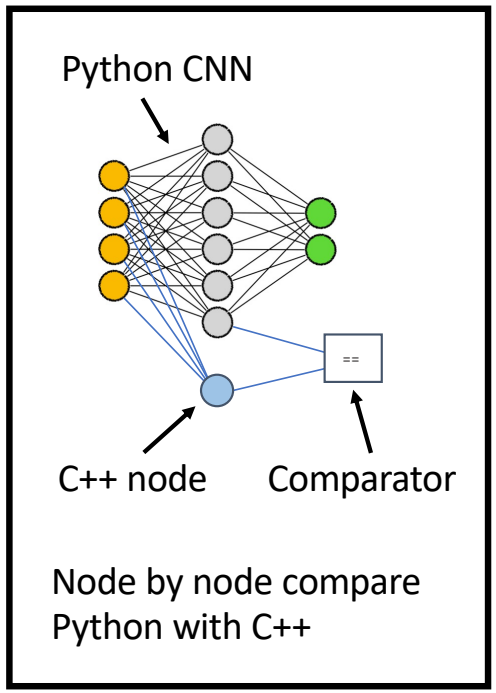
Computational Load

Function	Time - ms on RISC-V	Percentage
mfcc()	153.97	54.72
conv_2d()	59.05	20.37
dense()	61.18	21.14
Total	273.95	96.23

Profile for mfcc()

Function	Percentage
Preemphasis	0.13
Edge_padding	0.14
To_frames	0.27
Power_spectrum	96.88
Sum	0.12
Filter_energies	2.10
DCT	0.27
Cepstral_lift	0.02
Mean	0.02
Delta_mean	0.01
Cast_to_floats	0.01
Log_energy	0.00

Python to C++ Consistency



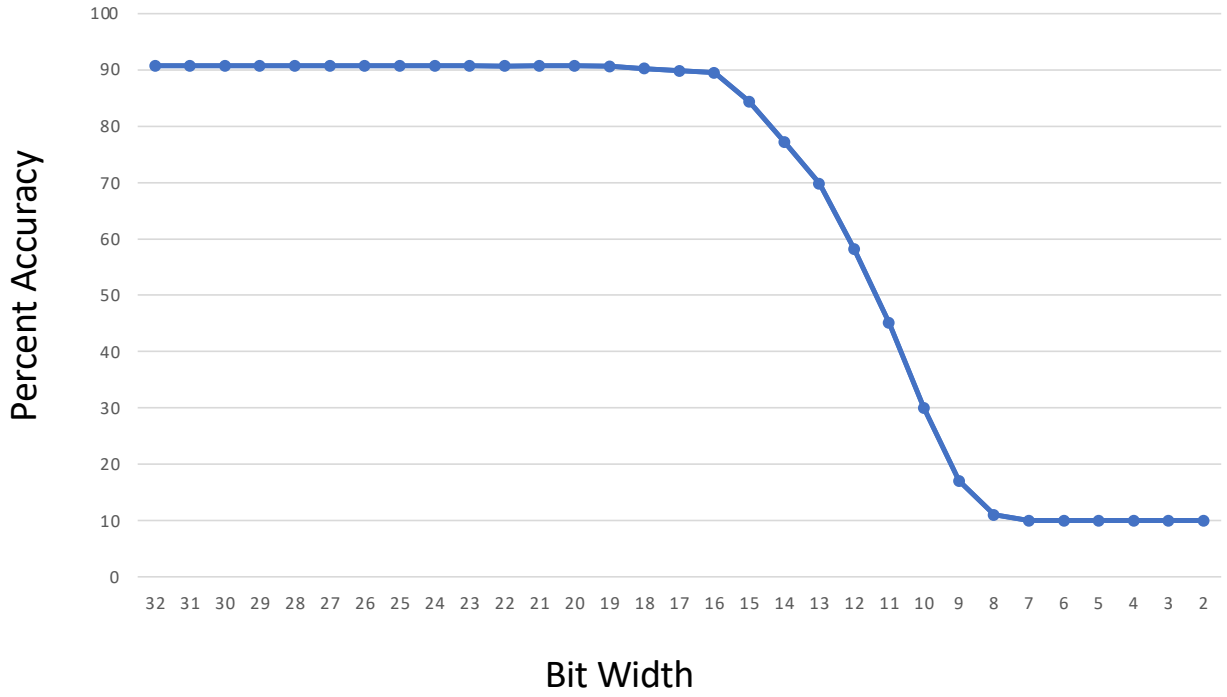
Python to C++

- Run C++ node in parallel with Python node
- Ran 80 samples and compared differences at all intermediate points
  - 10 different utterances with 8 different CNN variations
  - Run time was ~230 seconds or 3:50
- Differences were only found in bottom 4 bits of the mantissa
- Used ctypes library to link in C++ functions

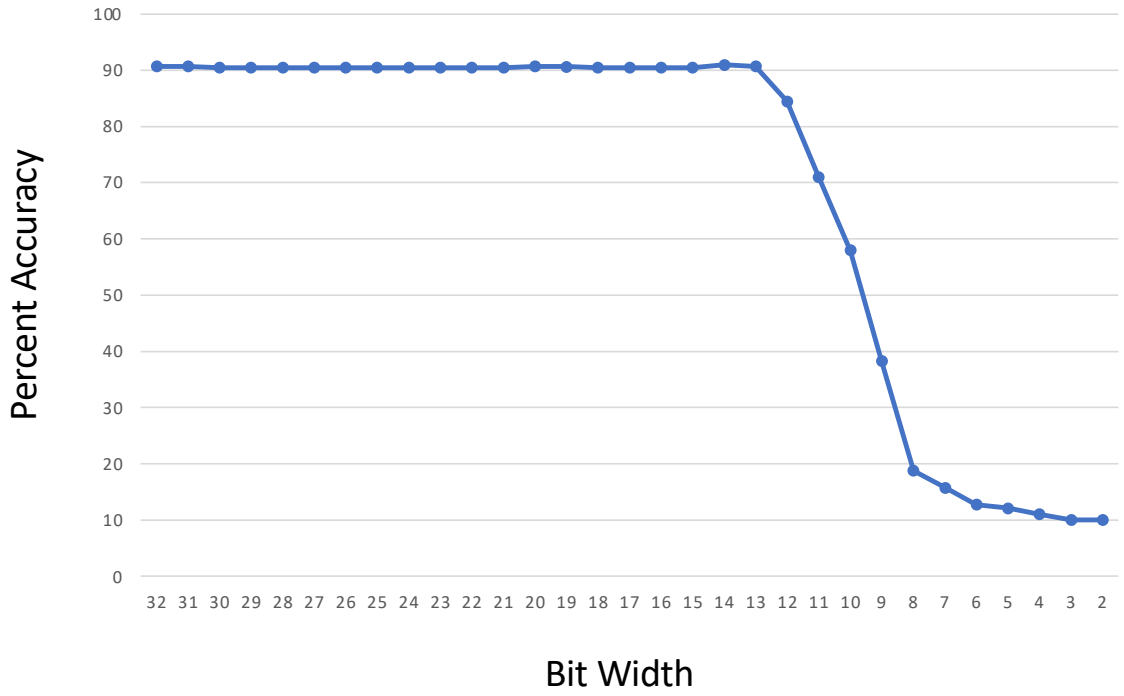
Quantization

- Performed a sweep for all fixed point values for inference and power spectrum to determine quantized accuracy
- Power spectrum needed 8-bit floating point representation
- Inference needed 16 bits for the convolution and 13 bits for dense layers
  - Settled on 16-bit fixed point

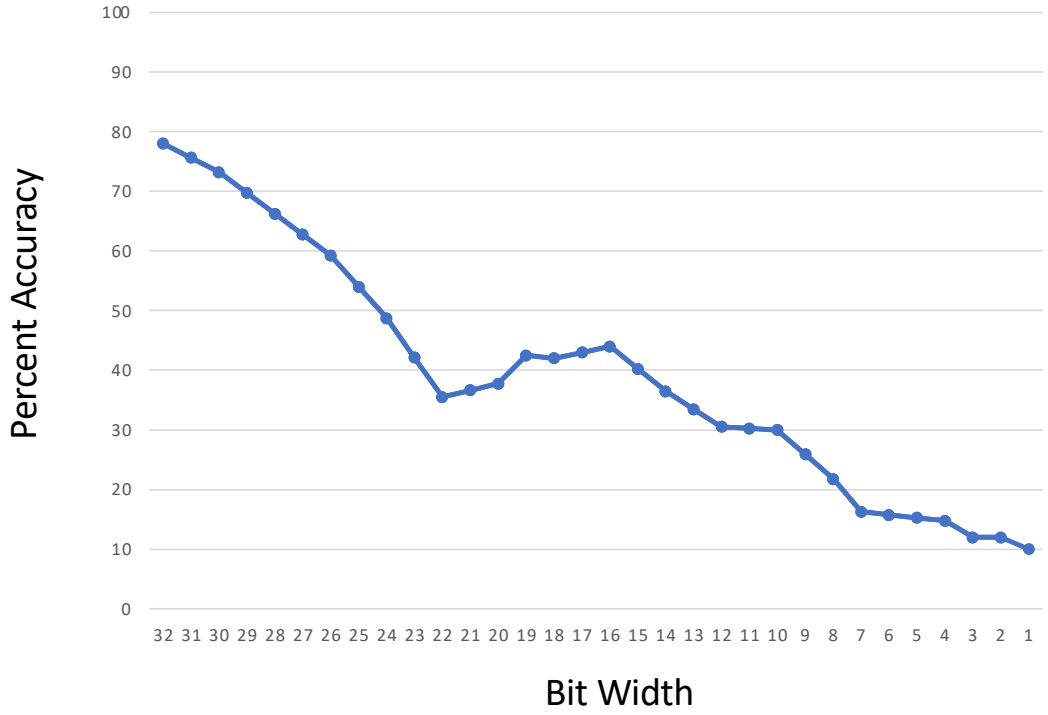
Bitwidth vs. Accuracy - Convolution



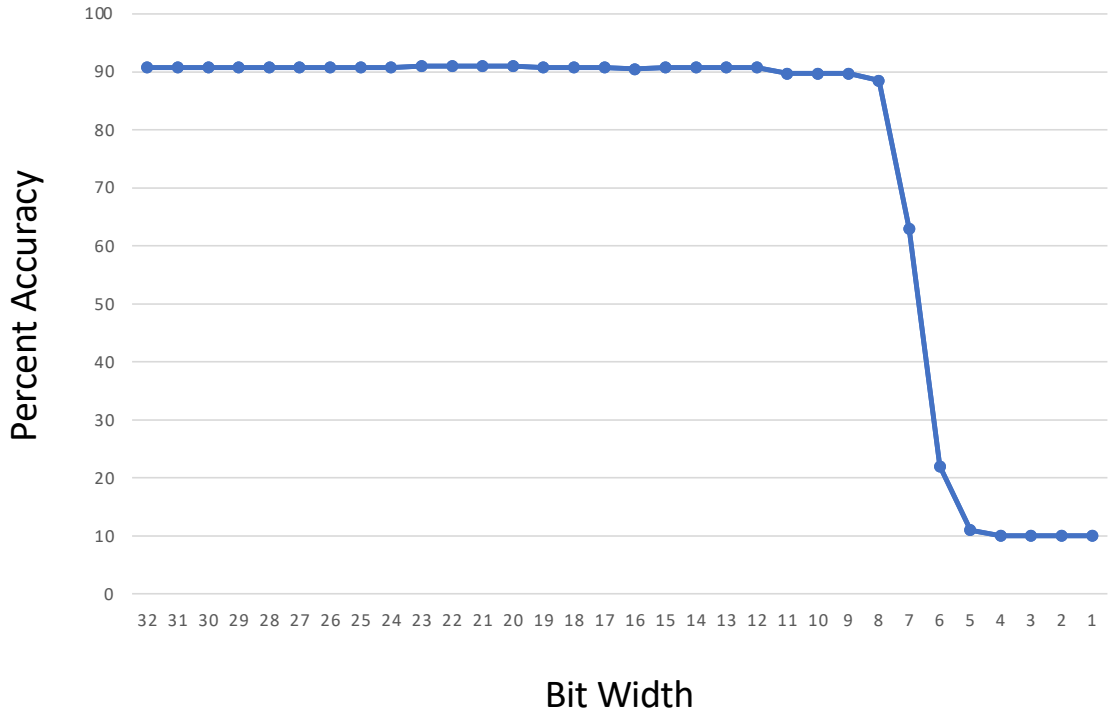
Bitwidth vs. Accuracy - Dense



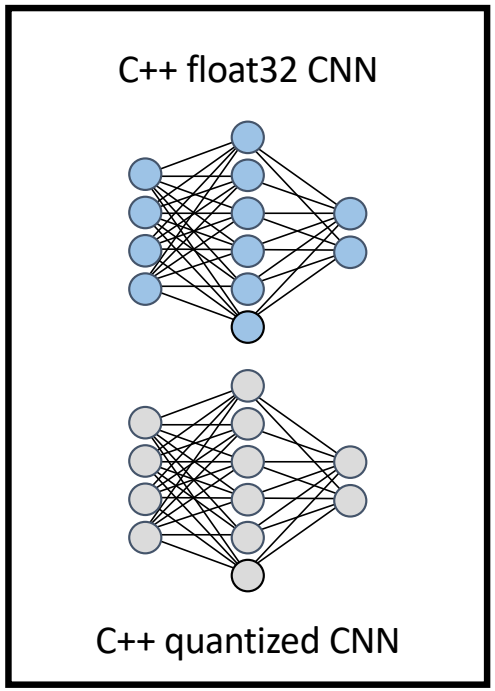
Bitwidth (Fixed Point) vs. Accuracy - PS



Bitwidth (Floating Point) vs. Accuracy - PS



C++ to Quantized Model Consistency

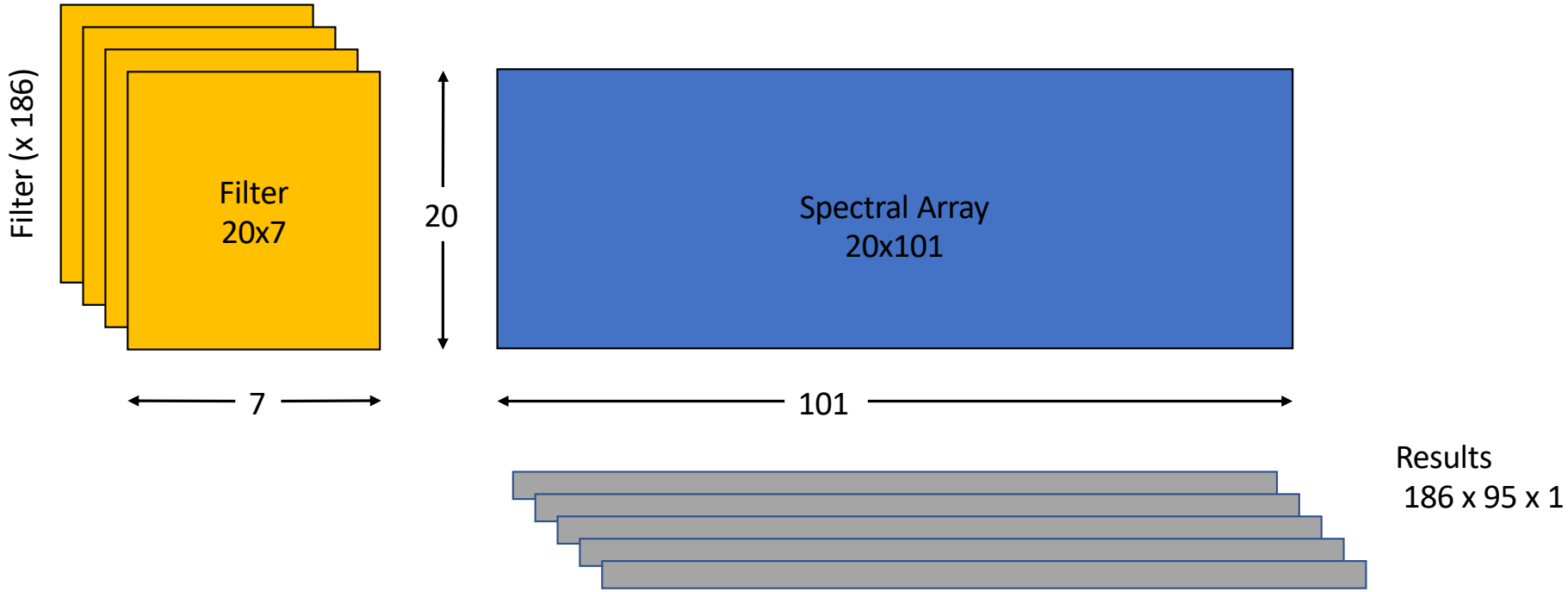


- Ran 80 samples concurrently between C++ and Quantized C++
  - Run time was 1,794 seconds (~1/2 hour)
- Measured line and branch coverage for all functions
  - Achieved 100% coverage
- Differences were large, but within expectations
  - Compared by assigning floating point representation to quantized representation
  - Results (excepting saturation) were +/- 0.00024

Architecting the Accelerators

- Data movement, buffering, loop unrolling pipelining
- When data is accessed multiple time, copy it to local memories or buffers to reduce memory traffic
- Line and frame buffers can hold portions of data being worked on
  - New data can be shifted in as older data is no longer needed
- Since data movement is often a limiting factor on performance, effective caching and buffering can significantly speed computations

Convolution – Layer 1



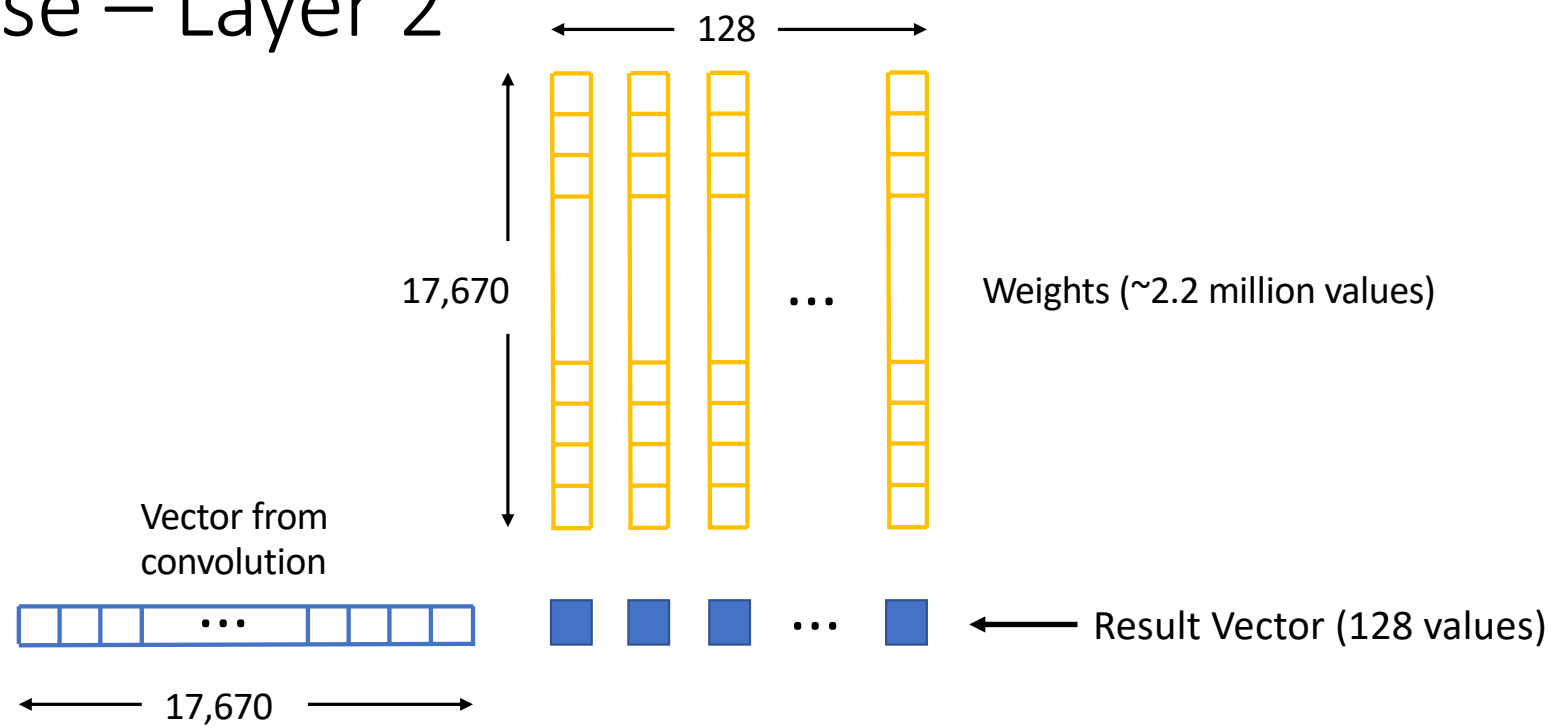
Convolution - Layer 1

- Spectral Array data elements are referenced 1,302 times each
  - The total size of the spectral array is 2020 words, small enough to cache local to the accelerator
- Each filter data element is referenced 95 times
  - The total size of the filter is 140 words
- Output lines are calculated from a single filter
  - Each filter can be read in, used to calculate an output line, then discarded
  - This requires a buffer for only one filter

Convolution - Layer 1

- Fastest architecture is 140 multipliers
  - Filters are held in registers, a portion of the spectral image in registers with the remainder in line buffers
- A more efficient architecture has 20 multipliers
  - It multiplies one column of the spectral array per clock
- Due to the high-level of data re-use, if data is held locally, this function is compute bound

Dense – Layer 2



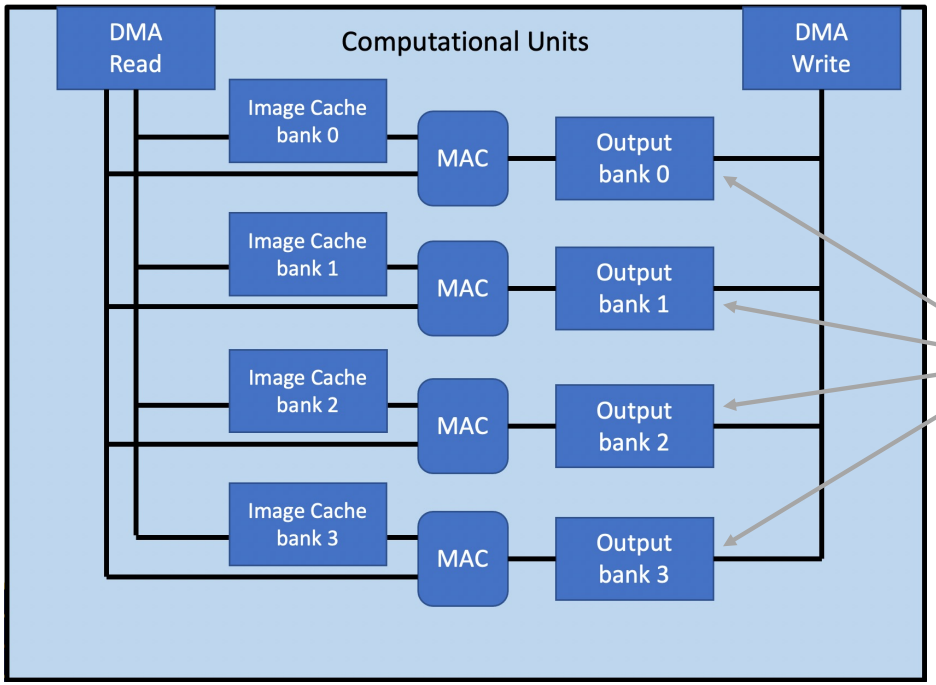
Dense – Layer 2

- Elements from the convolution are referenced 128 times
- Each weight element is referenced just once
- Output is just 128 words
  - Holding partial sums for the output lines can be done in a small memory or registers
- Each convolution element can be read in and multiplied against 128 weights
  - The result can be added to a partial sum array
  - The feature and weights can be discarded
  - This minimizes memory storage
- This layer is limited by how fast weights and features can be read in
- Weights need to be packed in memory correctly to optimize bus utilization and performance

Balancing Communication and Computation

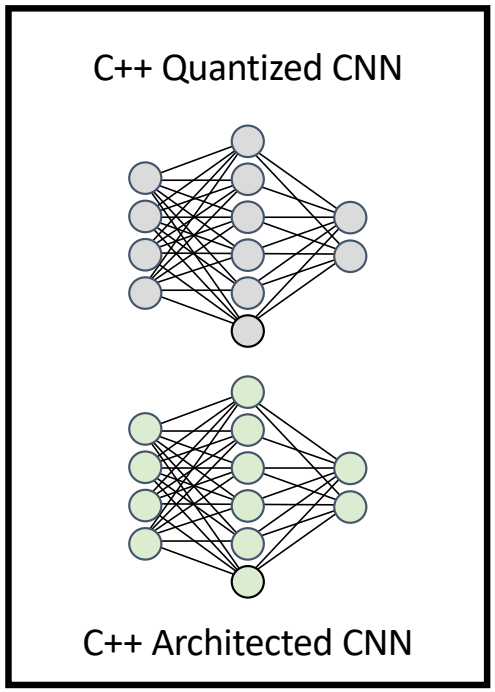
- Computations may be limited by the arrival of data
  - In this example, dense layer in CNN weights are used once
  - Any more multipliers than the number of data element delivered per clock will be wasted

Hardware Implementation – Dense Layer



Interleaved RAM Banks for image and results

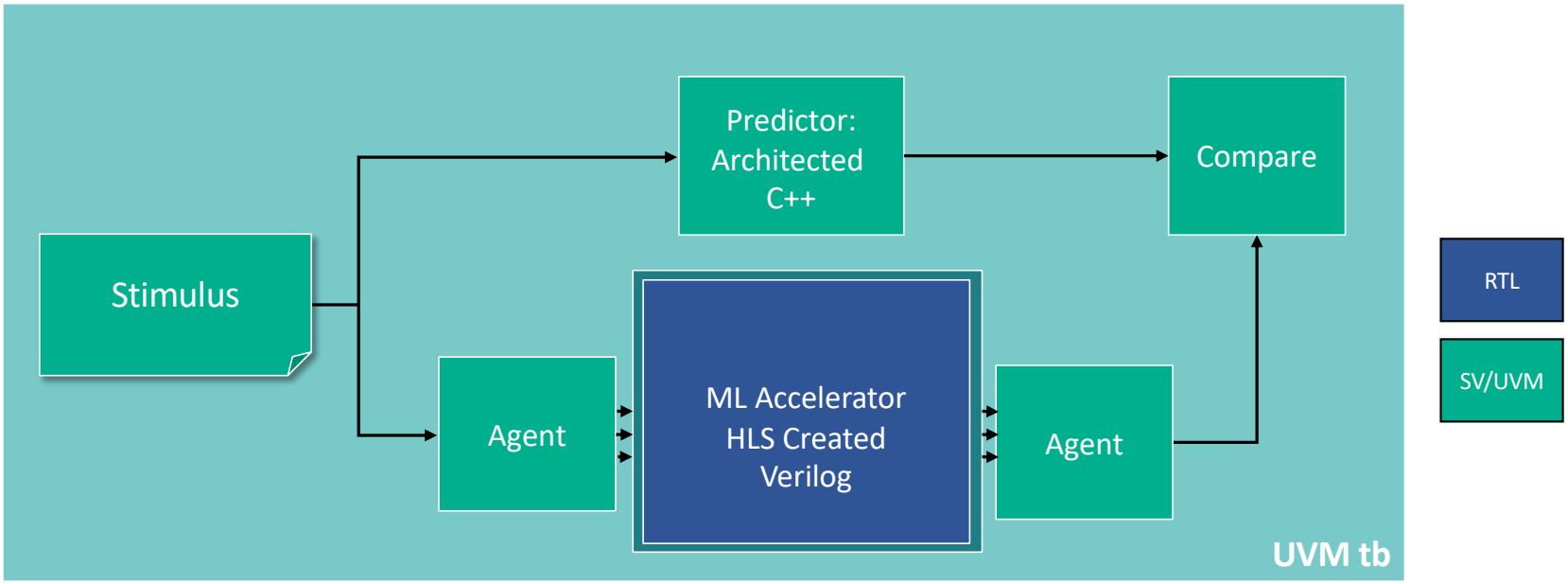
C++ Quantized to C++ Architecture Consistency



Architecture

- Ran same 80 test cases
  - 10 audio samples x 8 CNN configurations
  - Run time was 1.2 hours
- Fully covered architectural and quantized models
- Differences were limited to +/- 3 LSBs on inference and +/- 1 LSB for power spectrum

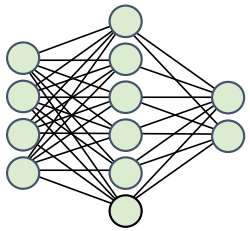
C++ to RTL comparison



With Architected C++ proved equivalent to original Python, used it as a predictor for RTL
RTL and C++ matched to the except for the LSB

C++ to RTL consistency

C++ Architected CNN



Formal

Verified the core algorithms (matmul, dense, 2d convolution)

UVM

- Architected C++ is used as a predictor for RTL verification
- Simulation covered CNN architecture variations
- 8 test cases
  - softmax was in SW
- 34.4 hours of simulation

```
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
```

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