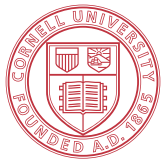
MICRO'16 Tutorial on Rapid Exploration of Accelerator-rich Architectures

Rapid Hardware Specialization with HLS: Glass Half Full?

Zhiru Zhang

School of ECE, Cornell University

October 2016

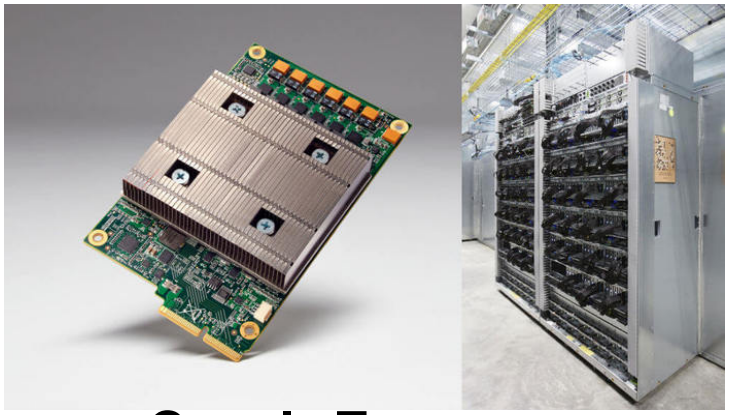


Cornell University

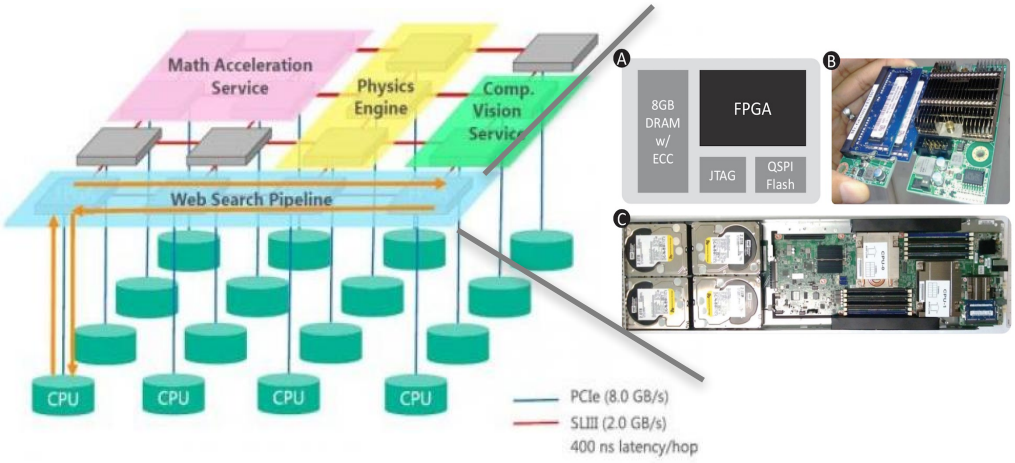
CSL

Best of Times to Work on Hardware Accelerators

- **Pressing demand** to efficiently accelerate a growing array of datacenter & embedded workloads
  - Multicore performance scaling significantly slowed
  - Pervasive hardware specialization is inevitable?



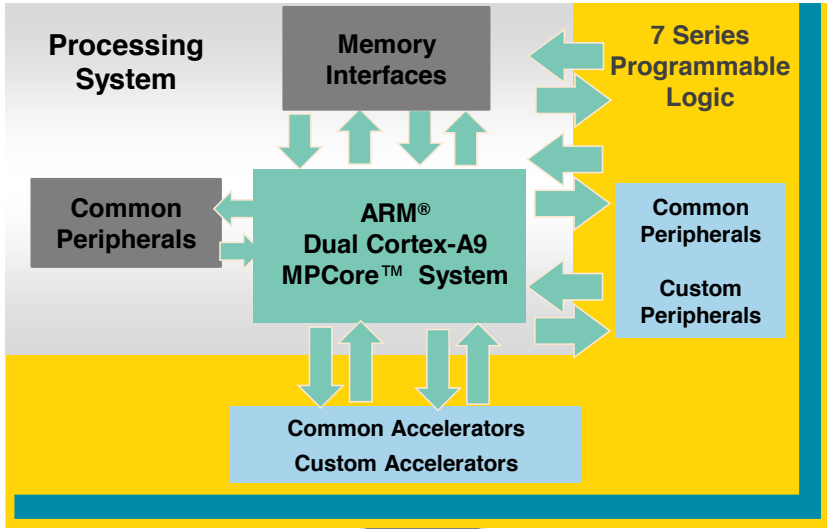
Google Tensor Processing Unit



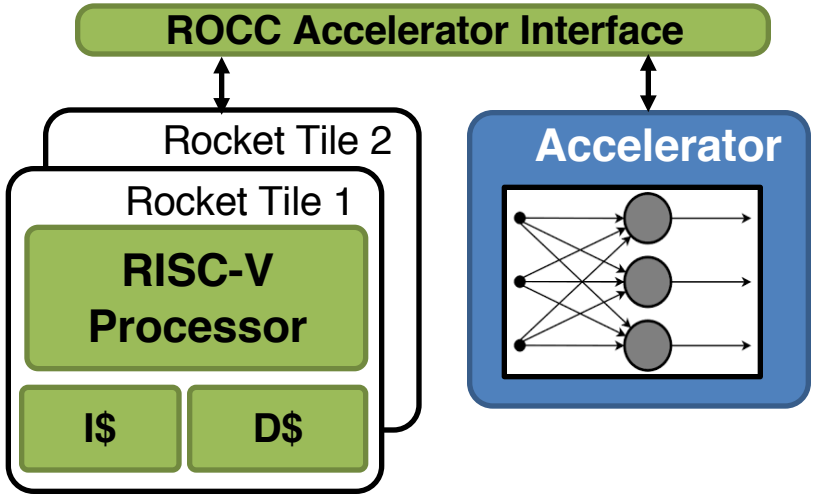
Microsoft Project Catapult

Best of Times to Work on Hardware Accelerators

- **Substantial leverage** readily available for building realistic specialized computing systems
  - Open-source hardware movement (e.g., RISC-V)
  - Open-source compiler infrastructure and architecture simulator (e.g., LLVM, gem5)
  - FPGA SoCs have come of age



Xilinx Zynq SoC

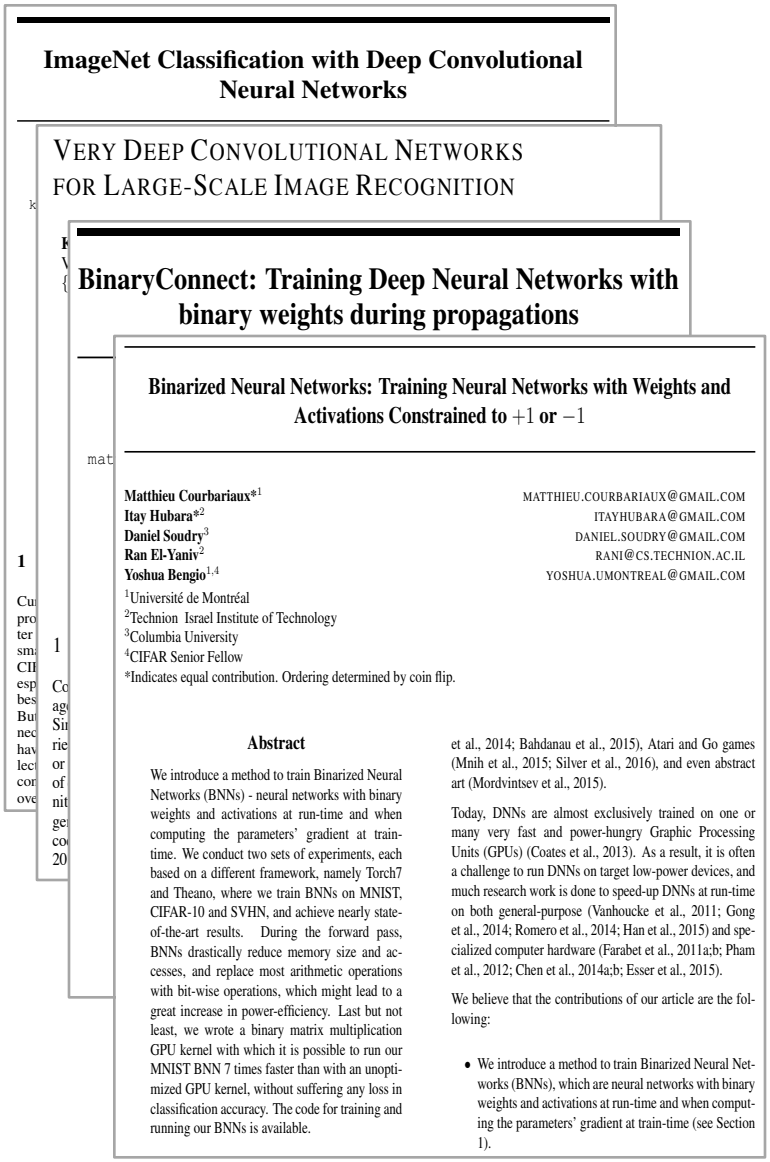


RISC-V + Accelerator

"Worst of Times" Also?

Is RTL design method still viable?

- **Target** of specialization is constantly moving, and **moving fast**!
  - In particular, deep learning algorithms are changing every few days
  - Tooling and software frameworks are also rapidly evolving



Motivation for High-Level Synthesis (HLS)

```
module dut(rst, clk, q);
  input rst;
  input clk;
  output q;
  reg [7:0] c;

  always @ (posedge clk)
  begin
    if (rst == 1b'1) begin
      c <= 8'b00000000;
  end
  else begin
    c <= c + 1;
  end

  assign q = c;
endmodule
```

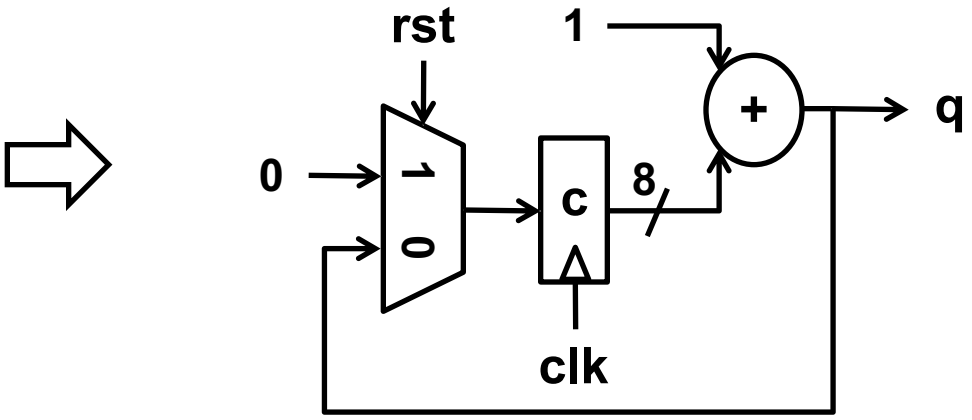
RTL Verilog

vs.

```
uint8 dut() {
  static uint8 c;
  c+=1;
}
```

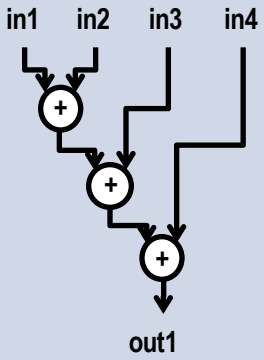
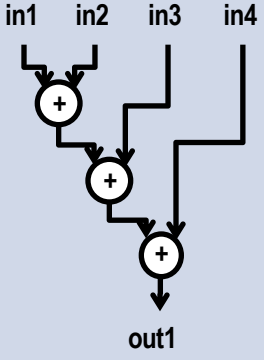
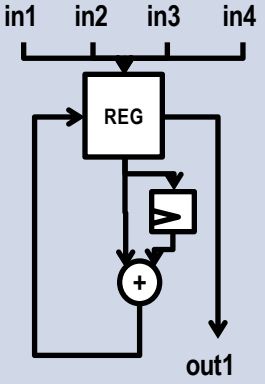
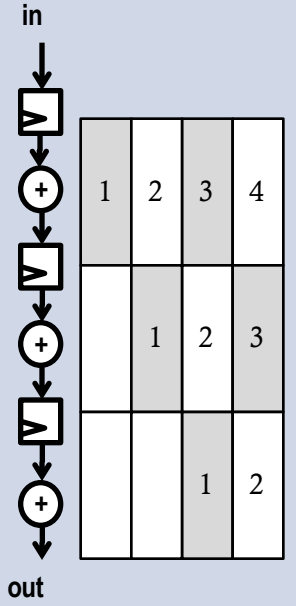
Untimed C code

HLS



An 8-bit counter

HLS: From Untimed to Timed

Control-Data Flow Graph	Latency	Area	Throughput
			
Untimed	Combinational	Sequential	Pipelined

HLS Interface Synthesis

- A GCD example with handshaking interface
  - HLS greatly simplifies interface design

C code

```
void GCD ( msg& req,
                msg& resp ) {
  short a = req.msg_a;
  short b = req.msg_b;
  while ( a != b ) {
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  resp.msg = a;
}
```

Manual RTL (partial)

Declaration

```
module GcdUnitRTL
(
  input  wire [   0:0] clk,
  input  wire [  31:0] req_msg,
  output wire [   0:0] req_rdy,
  input  wire [   0:0] req_val,
  input  wire [   0:0] reset,
  output wire [  15:0] resp_msg,
  input  wire [   0:0] resp_rdy,
  output wire [   0:0] resp_val
);
```

State transition

```
always @ (*) begin
  if ((curr_state__0 == STATE_IDLE))
    if (req_val)
      next_state__0 = STATE_CALC;

  if ((curr_state__0 == STATE_CALC))
    if ((!is_a_lt_b&&is_b_zero))
      next_state__0 = STATE_DONE;

  if ((curr_state__0 == STATE_DONE))
    if ((resp_val&&resp_rdy))
      next_state__0 = STATE_IDLE;
end
```

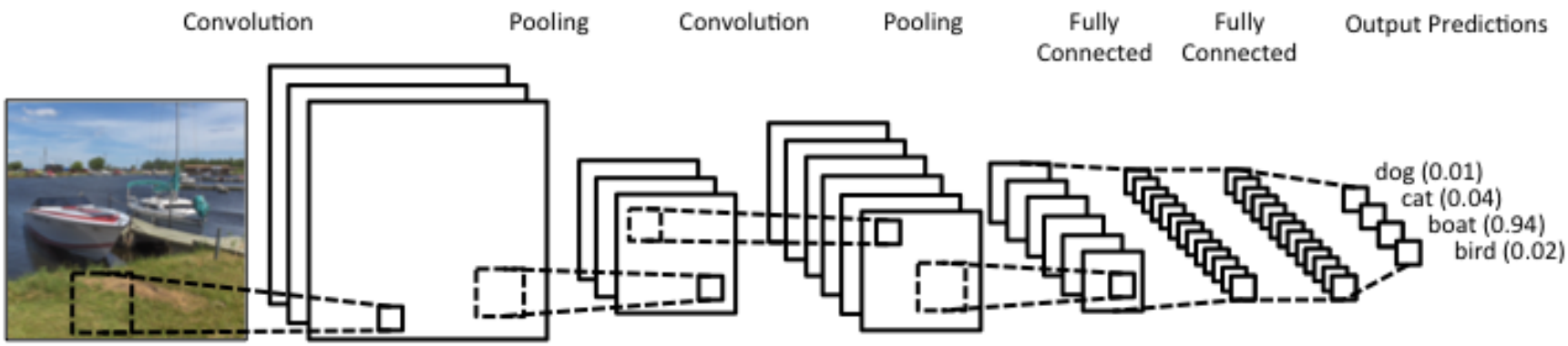
Output logic

```
always @ (*) begin
  if ((current_state__1 == STATE_IDLE))
    req_rdy = 1;   resp_val = 0;
    a_mux_sel = A_MUX_SEL_IN;   b_mux_sel = B_MUX_SEL_IN;
    a_reg_en = 1; b_reg_en = 1;

    if ((current_state__1 == STATE_CALC))
      do_swap = is_a_lt_b;   do_sub = ~is_b_zero;
      req_rdy = 0; resp_val = 0;
      a_mux_sel = do_swap ? A_MUX_SEL_B : A_MUX_SEL_SUB;
      a_reg_en = 1; b_reg_en = do_swap;
      b_mux_sel = B_MUX_SEL_A;

    else
      if ((current_state    1 == STATE_DONE))
        req_rdy = 0;   resp_val = 1;
        a_mux_sel = A_MUX_SEL_X;   b_mux_sel = B_MUX_SEL_X;
        a_reg_en = 0;   b_reg_en = 0;
end
```

Case Study: Convolutional Neural Networks (CNNs)

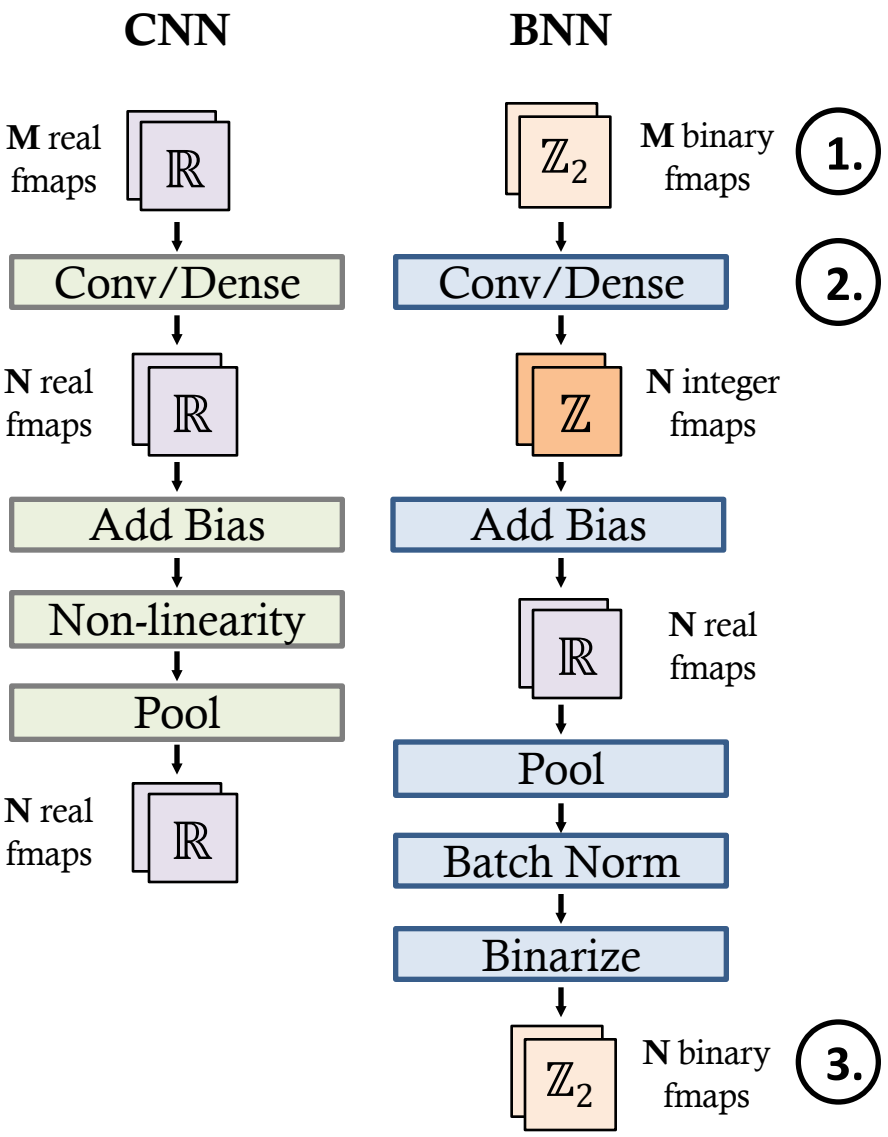


- Typical CNN architecture:
  - **Convolutional** (conv) layers in the front
  - **Fully connected** (dense) layers in the back
  - **Pooling** layers reduce the size of intermediate images (feature maps)

- CNNs have enormous computational and memory requirements
  - **VGG-19 Network** (ImageNet ILSVRC2014): 140 million floating-point (FP) parameters and 15 billion FP operations to classify one image [1]
  - 90+% of the computation is in the conv layers, 90+% of the model size is in the conv and dense weights (~**528 MB**)

[1] K. Simonyan and A. Zisserman. **Very Deep Convolutional Networks for Large-Scale Image Recognition**. arXiv:1409.15568, Apr 2015.

Binarized Neural Networks (BNNs)



Key Differences

1. Input fmaps to conv/dense layers are binarized (-1 or +1)
2. Conv/dense weights are binarized (-1 or +1)
3. Output feature maps are binarized (after some other steps)

Benefits for Hardware

- Binarized weights greatly reduce total model size
- Conv/Dense FP ops replaced with binary logic ops
- Complex non-linearities (tanh, sigmoid) replaced with Sign function

[2] M. Courbariaux et al., BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations. NIPS 2015
[3] M. Courbariaux et al. Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or −1, arXiv 2016.

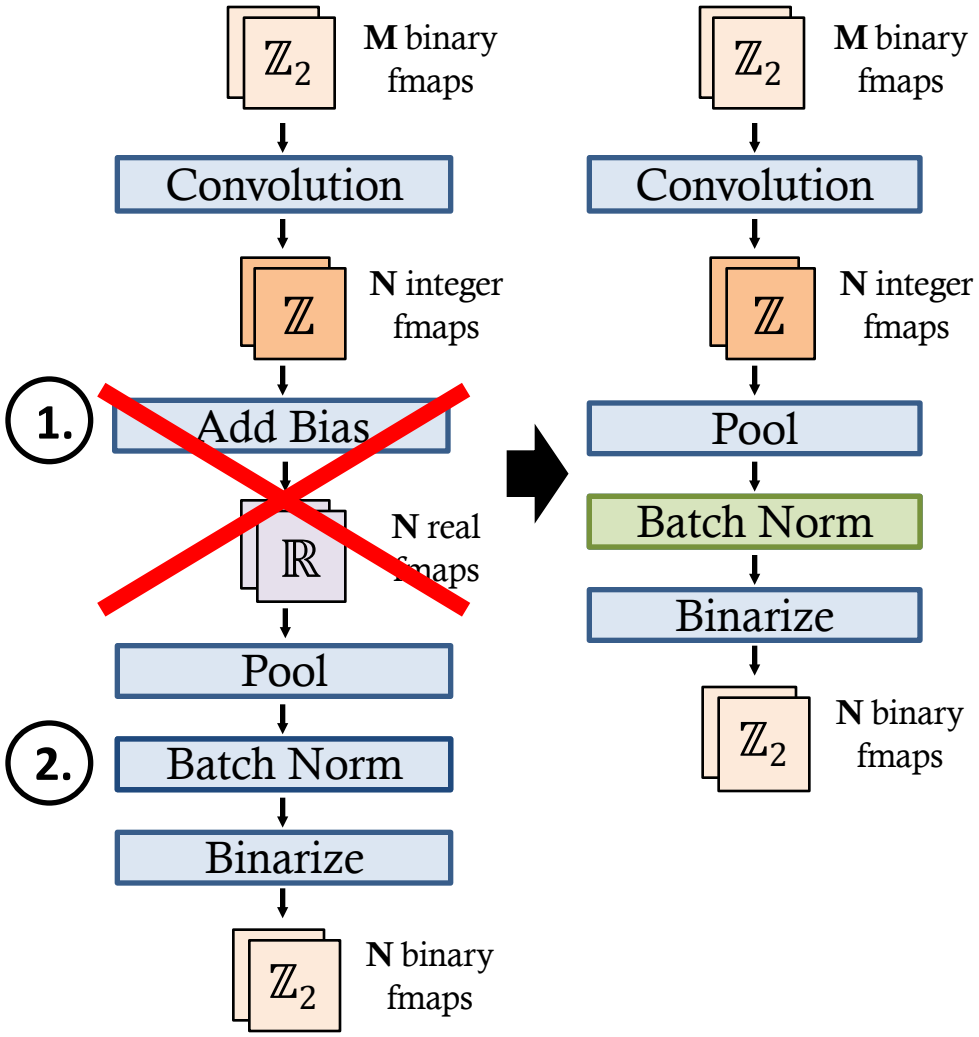
Cifar10 BNN Architecture

- Target dataset
  - CIFAR-10: 60000 32x32 color images
  - 10 classes consisting of animals and vehicles
- Architecture
  - 6 conv, 3 dense, 3 max pooling layers
  - First conv layer's input is floating-point image
- Software model
  - Python code from [3] available on Github

Layer	Input Fmaps	Output Fmaps	Output Dim	Output Bits	Weight Bits
Conv1	3	128	32	131K	4480
Conv2	128	128	32	131K	148K
Pool	128	128	16	33K	
Conv3	128	256	16	66K	297K
Conv4	256	256	16	66K	593K
Pool	256	256	8	16K	
Conv5	256	512	8	33K	1.2M
Conv6	512	512	8	33K	2.4M
Pool	512	512	4	8192	
FC1	8192	1024	1	1024	8.4M
FC2	1024	1024	1	1024	1.0M
FC3	1024	10	1	10	10K
Total					14.2M
Conv					4.59M
FC					9.46M

[2] M. Courbariaux et al., BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations. NIPS 2015
[3] M. Courbariaux et al. Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or −1, arXiv 2016.

Hardware-Optimized BNN Model



Changes

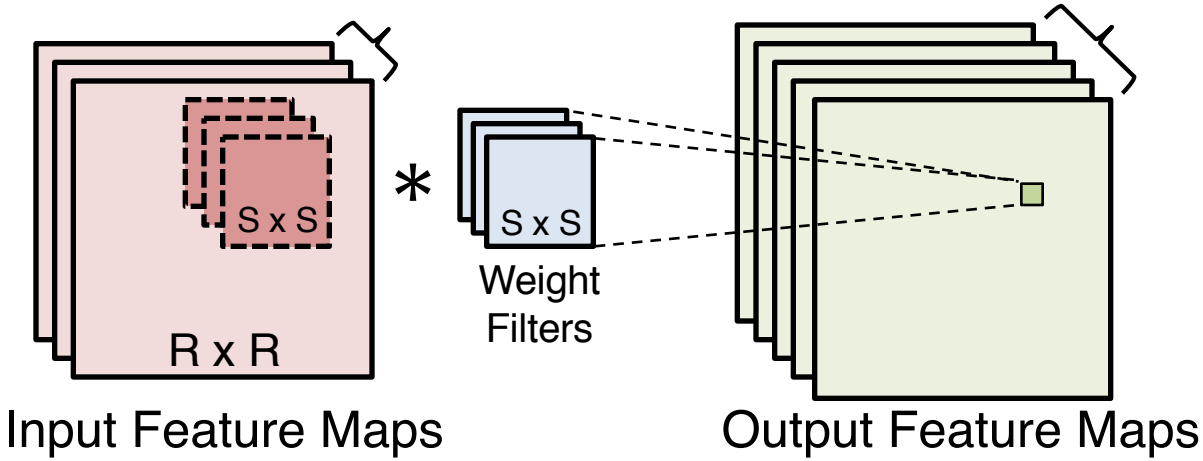
1. Removed biases (they had little effect on accuracy)
2. Simplified batch norm calculation
3. Quantized input image and batch norm parameters

Accuracy Impact

BNN Model	Test Error
Claimed in paper [2]	11.40%
Python out-of-the-box [2]	11.58%
C++ optimized model	11.34%
Accelerator	11.34%

[2] M. Courbariaux et al., BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations. NIPS 2015
[3] M. Courbariaux et al. Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or −1, arXiv 2016.

BNN Accelerator Design



Challenges of BNN Acceleration

1. How to exploit many sources of available parallelism

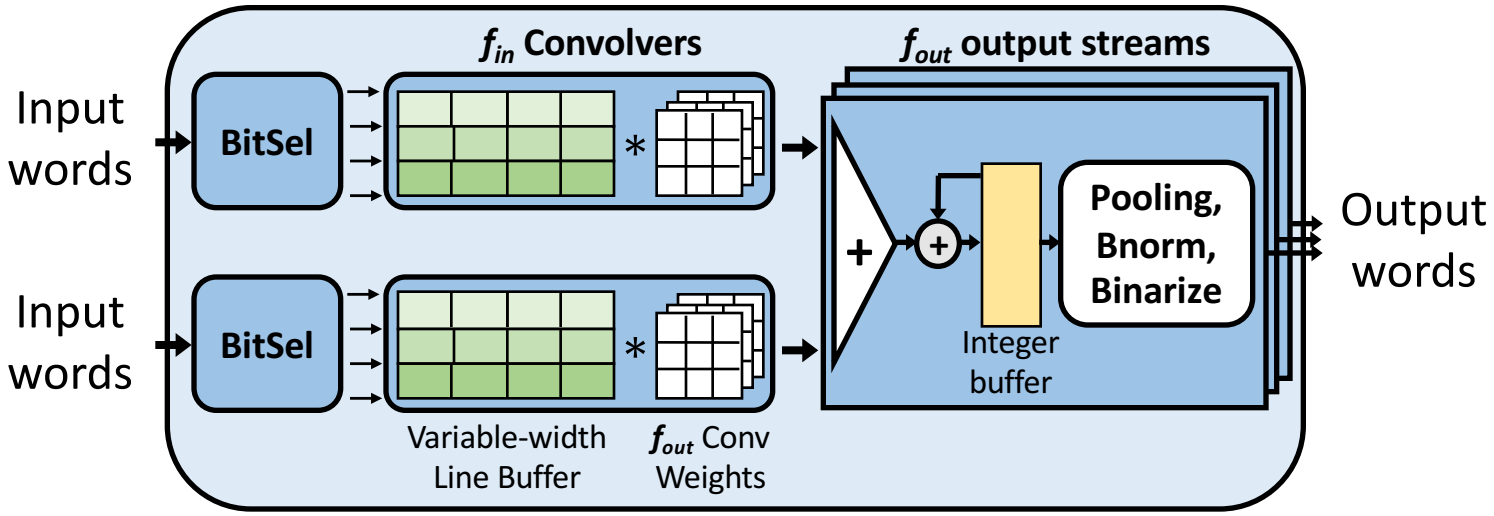
   (1) across input maps (2) across output maps

   (3) within a map (4) within a filter

2. Resource-efficient design uses a single module for all layers, and must handle different-sized input feature maps

3. Binarized data requires parallelism at the sub-word level (unique to BNN)

BNN Accelerator Architecture



Challenge	Our Solution
Many diverse sources of parallelism	Flexible architecture with parameterized numbers of input and output streams (f_{in}, f_{out})
Design must handle different-sized input maps	Novel hardware structures **Bitsel** and **variable-width line buffer** ensure pipeline is fully utilized regardless of input map width
Design must exploit sub-word level parallelization	Primary convolution pipeline processes data word-by-word, not pixel-by-pixel

HLS Code in C++

- **User writes and tests in a productive high-level language**
  - Architectural exploration through adding tool directives
  - CPU-FPGA interface automatically synthesized
  - Significant reduction in verification time
    - Full BNN would take days to simulate at RTL level
    - C++ execution finishes in seconds
- **Design effort**
  - Entire software-hardware implementation and testing performed by one student in 10 weeks

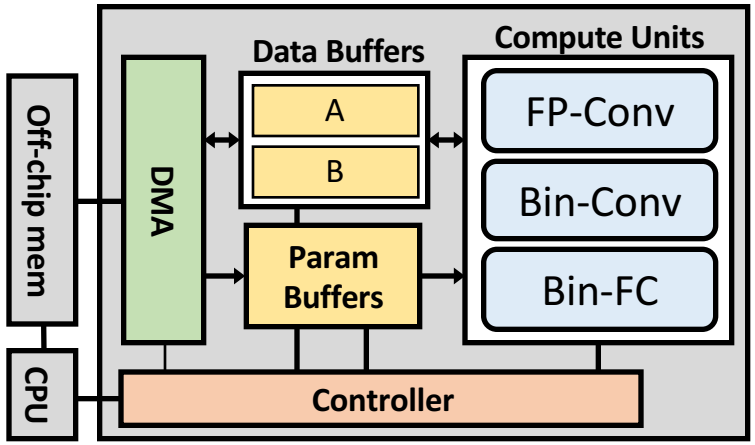
```
1  VariableLineBuffer linebuf;
2  ConvWeights wts;
3  IntegerBuffer outbuf;
4
5  for (i = 0; i < n_input_words; i++) {
6    #pragma HLS pipeline
7
8    // read input word, update linebuffer
9    WordType word = input_data[i];
10   BitSel(linebuf, word, input_width);
11
12   // update the weights each time we
13   // begin to process a new fmap
14   if (i % words_per_fmap == 0)
15     wts = weights[i / words_per_fmap];
16
17   // perform conv across linebuffer
18   for (c = 0; c < LINE_BUF_COLS; c++) {
19   #pragma HLS unroll
20     outbuf[i % words_per_fmap][c] +=
21       conv(c, linebuf, wts);
22   }
23 }
```

HLS pseudocode for part of Bin-Conv unit

FPGA Implementation Results

1. **Target platforms**
   - **CPU**: Intel Xeon E5-2640 multicore processor
   - **GPU**: NVIDIA Tesla K40 GPU
   - **mGPU**: Jetson TK1 embedded GPU board
   - **FPGA:** ZedBoard with Xilinx Zynq-7000

2. **Performance Comparison**
   - FPGA is 14x faster than embedded GPU
   - ~6x higher performance/watt over server-class GPU
   - Projected ASIC performance is 10x higher than FPGA

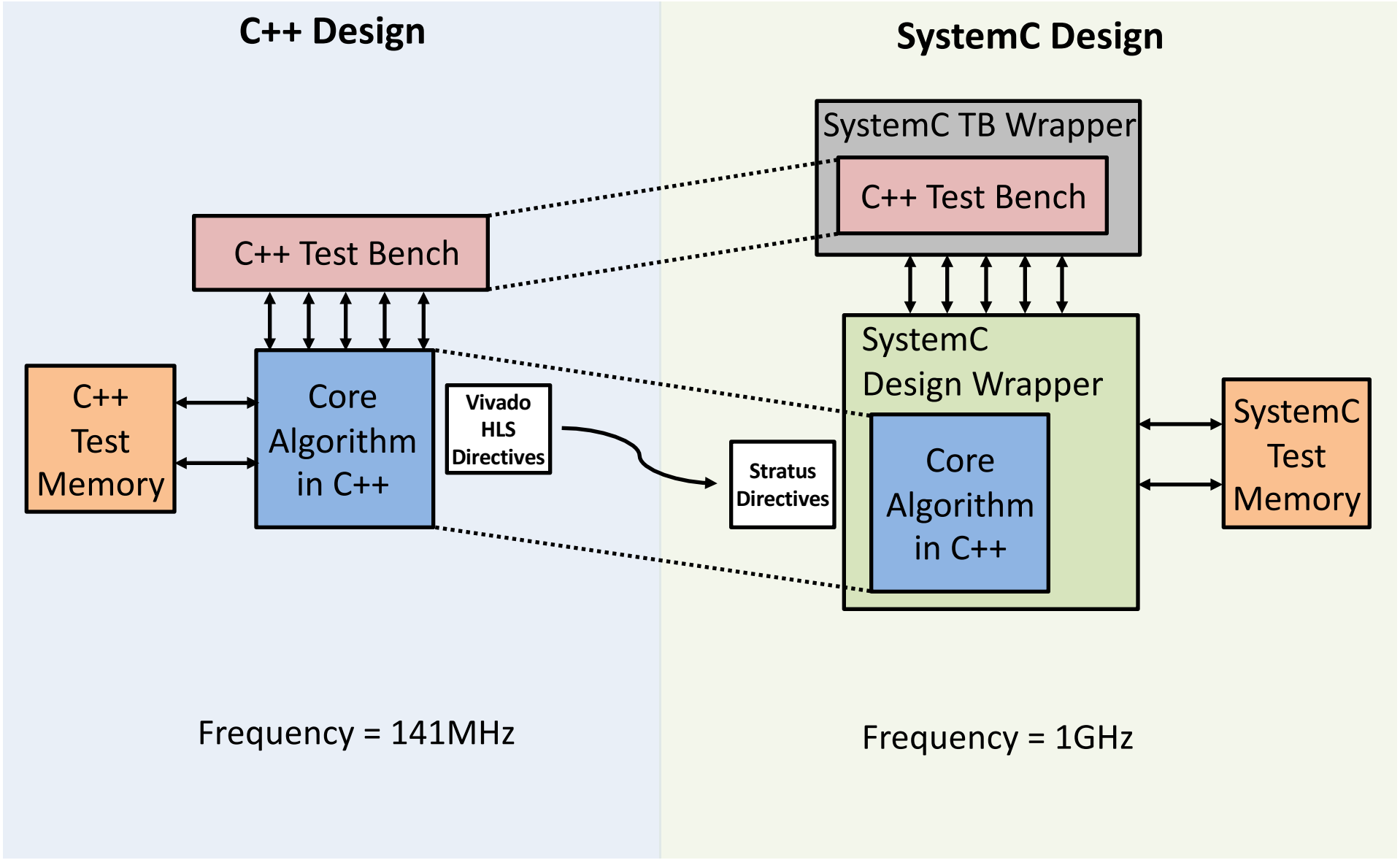


	Execution time per image (ms)			
	CPU	**GPU**	**mGPU**	**FPGA**
Conv1	0.68	0.01	–	0.11
Conv2-5	13.2	0.68	–	4.22
FC1-3	0.92	0.04	–	2.03
Total	14.8	0.73	90	6.36
Speedup	2.3x	0.11x	14.2x	1x
Power (Watt)	95*	235*	3.6	4.7
imgs/sec/Watt	0.71	5.83	3.09	33.5

A value we could not measure is indicated with –

Numbers with * are sourced from datasheets

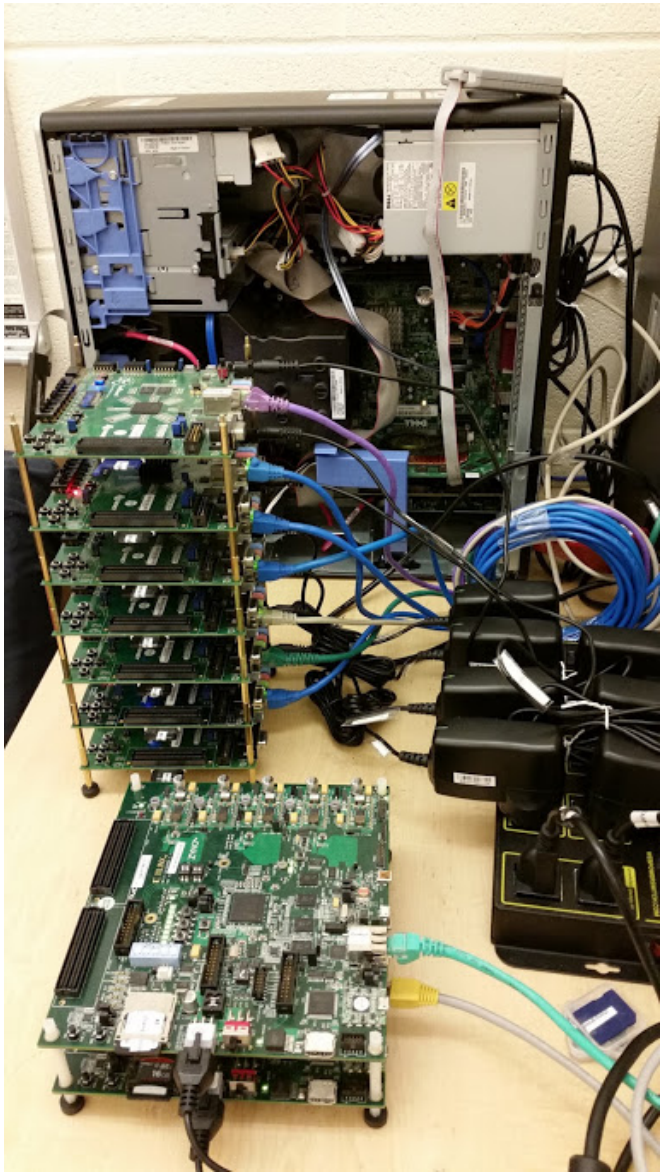
ASIC Implementation with SystemC



Supported by DARPA award HR0011-16-C-0037 under the CRAFT program

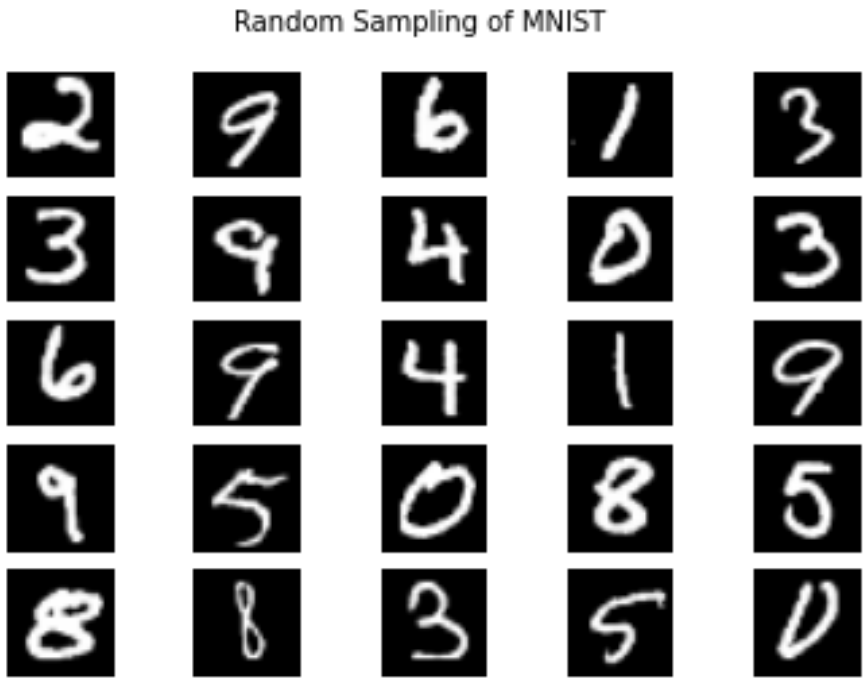
Teaching HLS for FPGAs

- Cornell ECE 5775: High-level digital design automation
  - 30-40 graduate and senior students
  - Finished multiple HLS assignments
  - Implemented a digit recognition accelerator on FPGAs



Case Study: Digit Recognition

- Use a simple machine learning algorithm to recognize handwritten digits
  - 2000 training instances per digit
  - Each training/test instance is a 7x7 bitmap after downsampling



MNIST dataset: http://yann.lecun.com/exdb/mnist/

K-Nearest-Neighbor (KNN) Implementation

```
bit4 digitrec( digit input )
{
  #include "training_data.h"
  // This array stores K minimum distances per training set
  bit6 knn_set[10][K_CONST];
  // Initialize the knn set
  for ( int i = 0; i < 10; ++i )
    for ( int k = 0; k < K_CONST; ++k )
      // Note that the max distance is 49
      knn_set[i][k] = 50;
```

Main compute loop
(10 cycles per innermost loop)

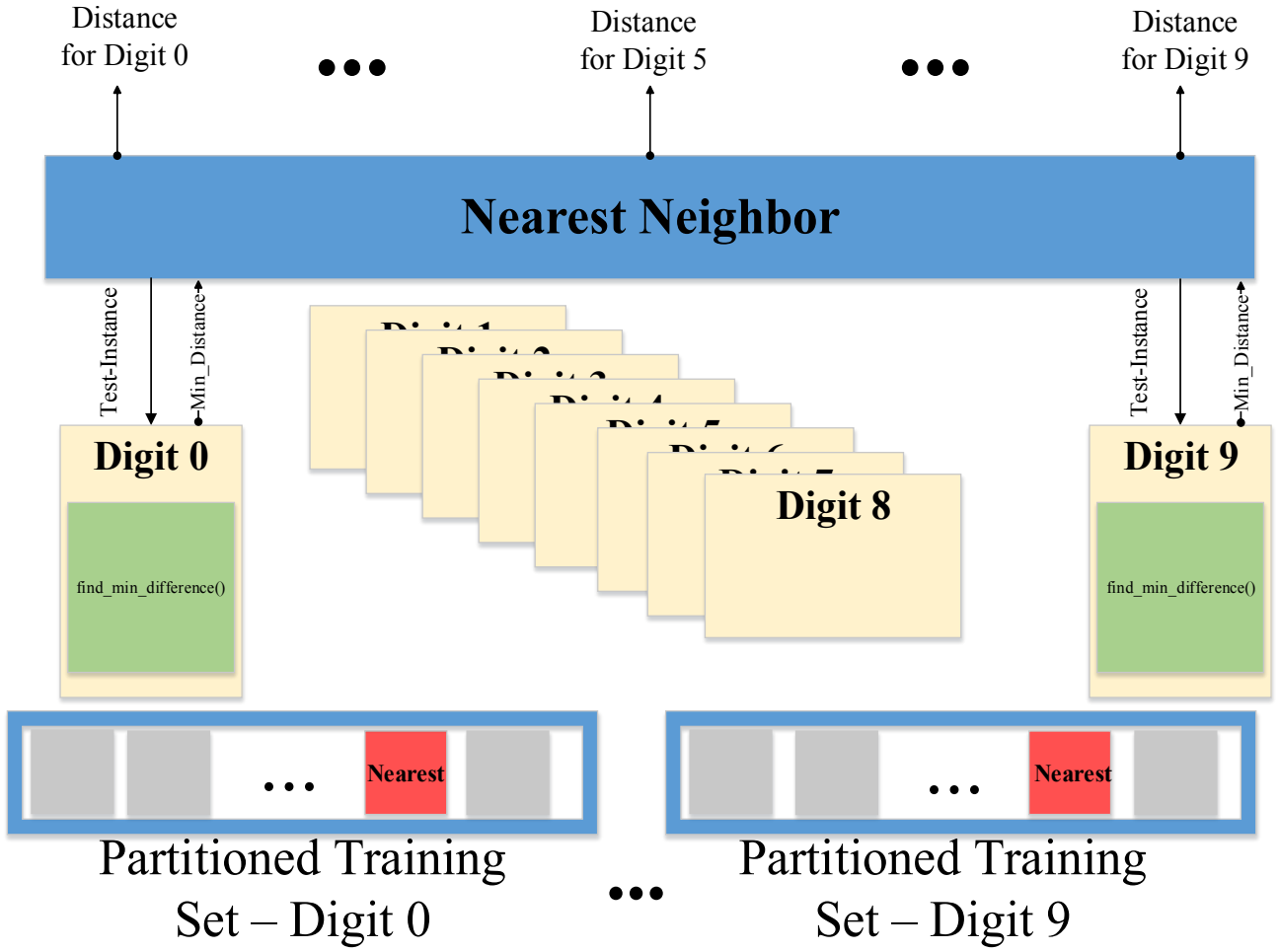
```
L2000: for ( int i = 0; i < TRAINING_SIZE; ++i ) {
  L10:   for ( int j = 0; j < 10; j++ ) {

    // Read a new instance from the training set
    digit training_instance = training_data[j * TRAINING_SIZE + i];
    // Update the KNN set
    update_knn( input, training_instance, knn_set[j] );
  }
}
```

~200K cycles by default without optimizations

Assignment: 100x Speedup!

- Students are expected to insert pragmas or modify source code to achieve a 100x speedup over unoptimized baseline



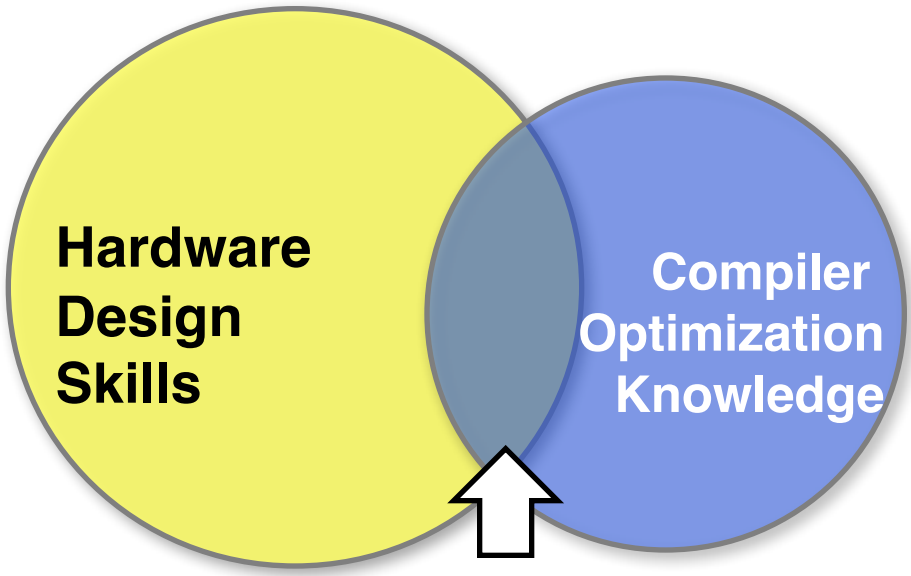
Optimization through Pragmas

- To achieve a 100x speedup over unoptimized baseline
  - Two pragmas would suffice: pipelining + array partitioning
    - Pipelining outer loop entails unrolling of the inner loop

- Only a small subset of students used the minimum set of pragmas
  - Many achieved the target speedup with pragma overuse and unnecessary code changes
  - “Wrong” combinations of pragmas led to counter-intuitive performance-area trade-offs
    - Completely partition the array hurts the performance

So Have We Solved HLS?

- Existing HLS methodologies are still not intuitive enough for non-expert users
  - Low guarantee on **out-of-the-box** quality of results (QoR)
  - **Leaky abstraction** of user directives



List of tunable options of a commercial HLS tool:

inline_calls [-all] [object_id]
pipeline_function [-latency {min [max]}] [-extra_timing_effort]
convert_to_lookup
unroll_loop [-num_pre_body integer] [-num_in_body integer] [-num_tries integer]
pipeline_loop [-init_interval num_states] [-extra_timing_effort] [-min_lat_interval num_states] [-max_lat_interval num_states] [-allow_io_reordering]
flatten_array
merge_arrays [-data | -addr] [-min_write_width integer] [-name new_array_name]
split_array -addr/data bit_index
restructure_array addr_width_delta
…

HLS Expert Users

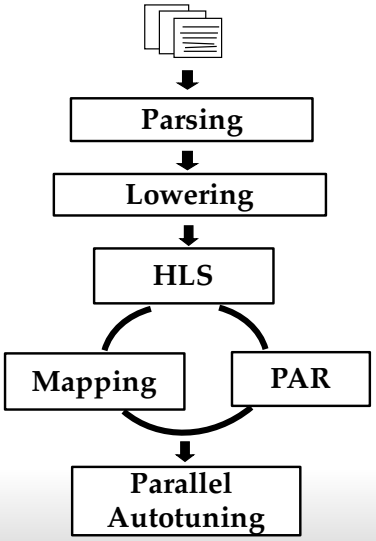
Who can figure out what annotations to provide to get the desired hardware

Our Ongoing Effort

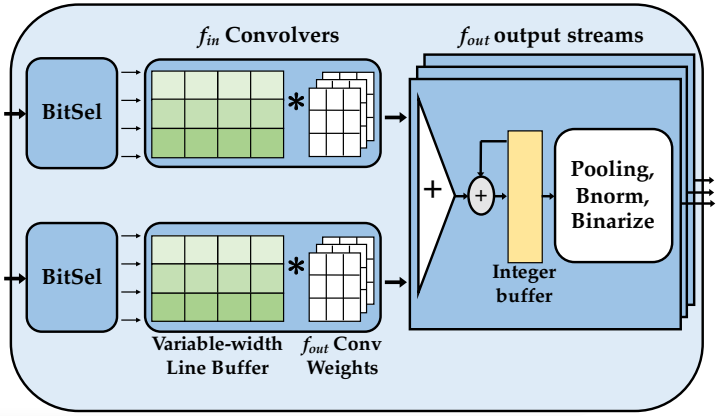
```
// algorithm
padded(x, y, j, i) = select(x > =0 &&
x < S && y >= 0 && y < S,
converted(x, y, j, i), 0);
RDom r (0, K, 0, K, 0, M);
res(x, y, j, i) += kernel(r.x, r.y, r.z, j) *
padded(x+1-r.x, y+1-r.y, r.z, i);

// schedule
padded.compute_root();
res.vectorize(x).parallel()
```

Domain-specific Specification



Cross-Layer Synthesis



Hardware Architecture

High-Level Design & Automation

- Co-design of programming language, compiler, and architecture
  - Domain-specific specifications
  - Cross-layer synthesis + parallel autotuning
  - Fine-grained, medium-grained, and coarse-grained co-processor architectures

Concluding Remarks

- HLS is here to stay as a key enabler for rapid hardware specialization
  - More turns per day than RTL-based flow

- Challenges remain to widespread use of HLS
  - Substantial room for improvement in out-of-the-box QoR and programming abstractions
  - Research opportunities abound for both architecture and DA communities!