NSF #2117997

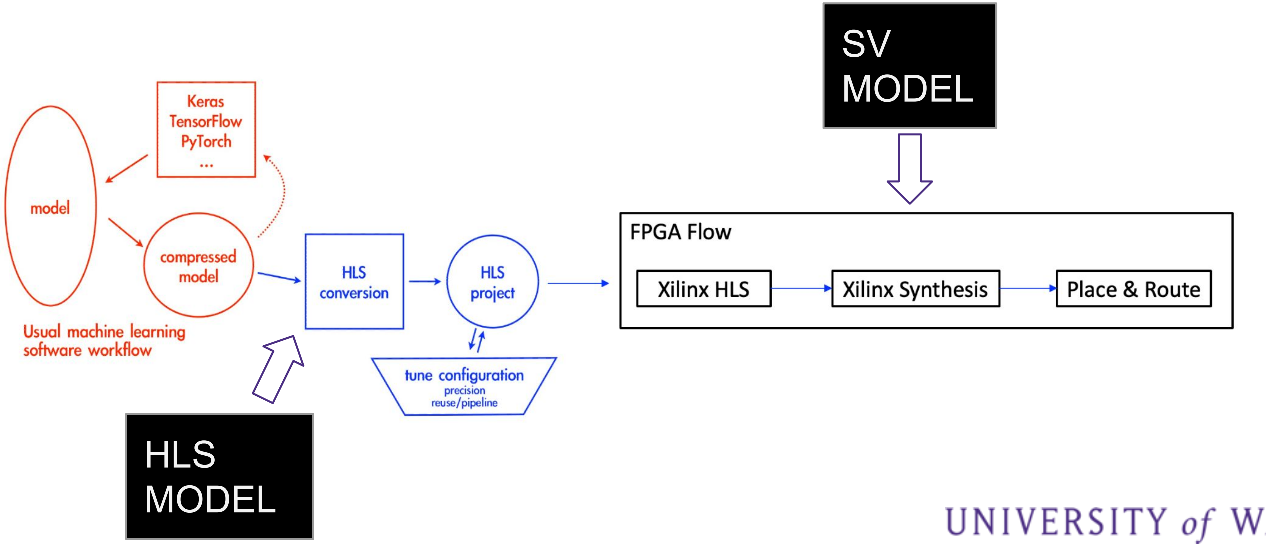
Quantifying the Efficiency of High-Level Synthesis for Machine Learning Inference

Caroline Johnson, Scott Hauck, Shih-Chieh Hsu, Waiz Khan, Matthew Bavier, Oleh Kondratyuk, Trinh Nguyen, Stephany Ayala-Cerna, Anatoliy Martynyuk, Aidan Short, Jan Silva, and Geoff Jones

UNIVERSITY of WASHINGTON

Our Process

- Goal: quantify the losses/gains from using the HLS4ML platform
- Compare resources and performance



SV MODEL

FPGA Flow

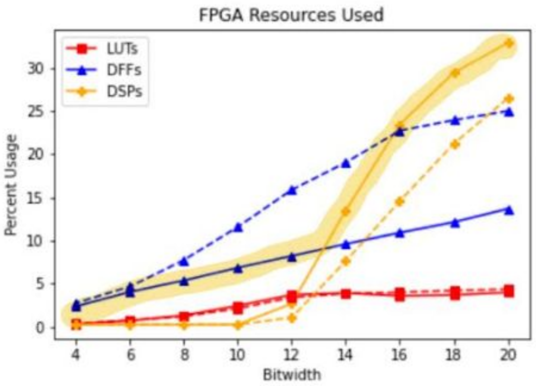
Xilinx HLS → Xilinx Synthesis → Place & Route

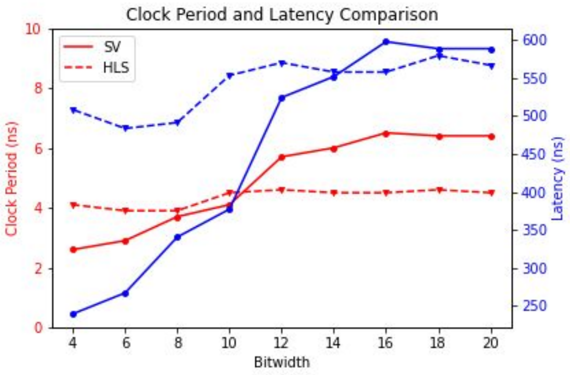
HLS MODEL

Our Analysis

- Results are in terms of max resource usage
- HLS results are dashed, SV results are solid

Dashed - HLS
Solid - SV

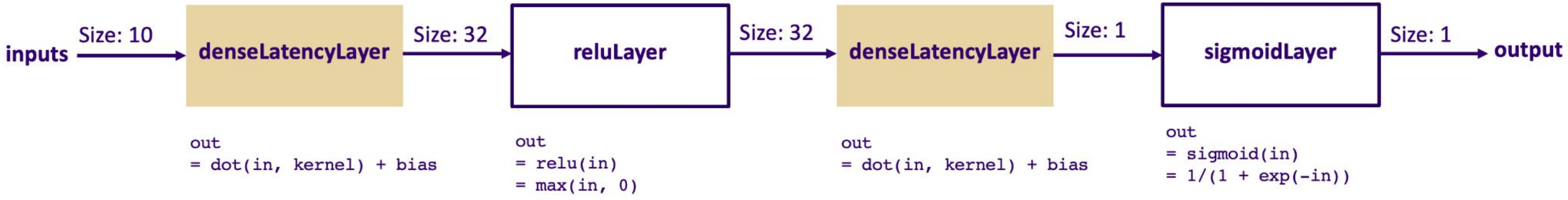

FPGA Resources Used


Clock Period and Latency Comparison

Resources: Ax more, Latency: Bx worse, Clock Period: Cx better

Benchmark 1

One-Layer Model



Initial Approach

- Heavy pipelining
- Constant folding, II = 1
- Neural-network specific DSP Optimizations

Overall goal:
Match HLS4ML's accuracy with better performance and resource usage

Multiplier Packing into DSPs

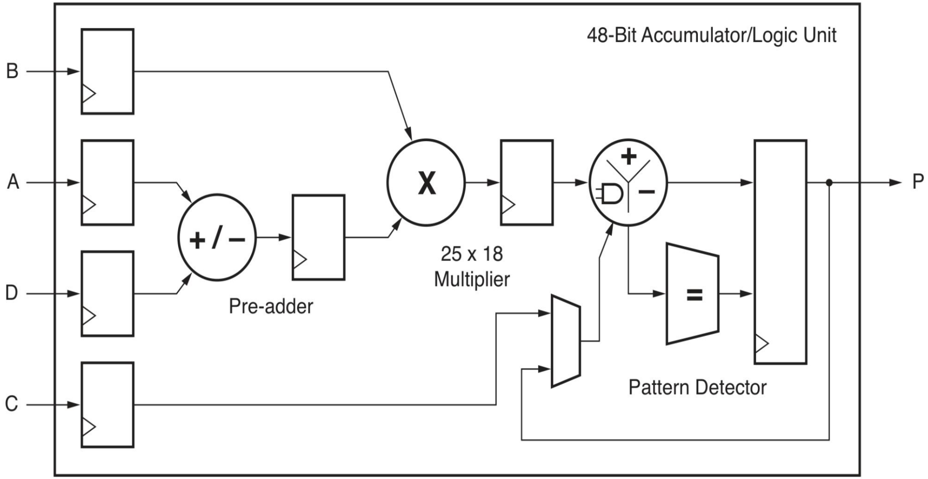
Virtex 7 supports 25x18 bit multiplication

- Bitwidths <=8 can be combined via the DSP pre-adder

input

weight 1

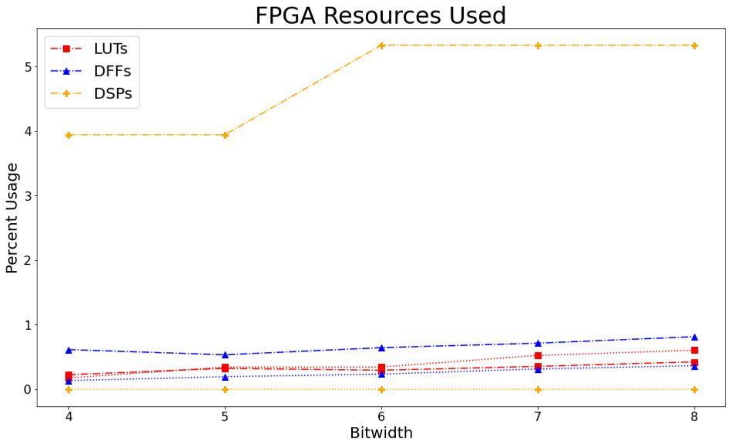
weight 2 << X

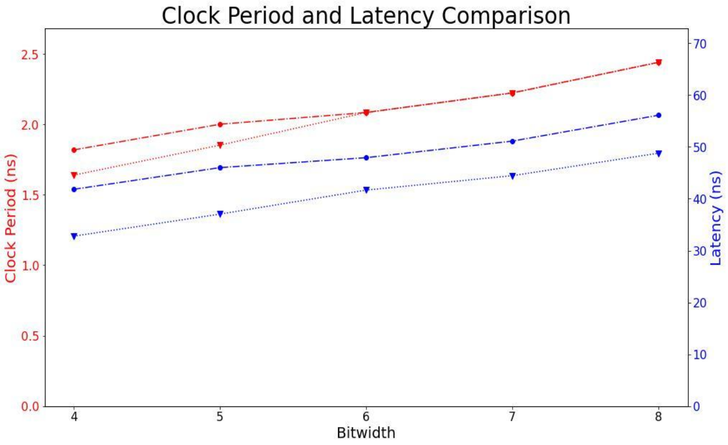


output = weight 1 * input
and weight 2 * input

Multiplier (DSP) Packing

Dashed - SV with packing
Solid - SV without packing

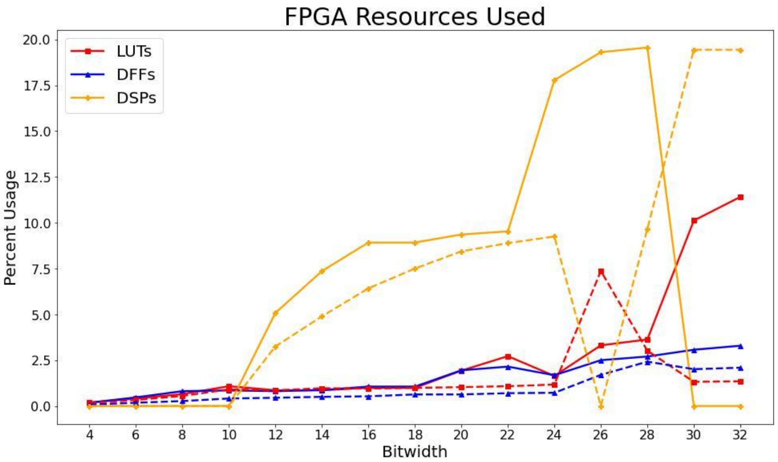


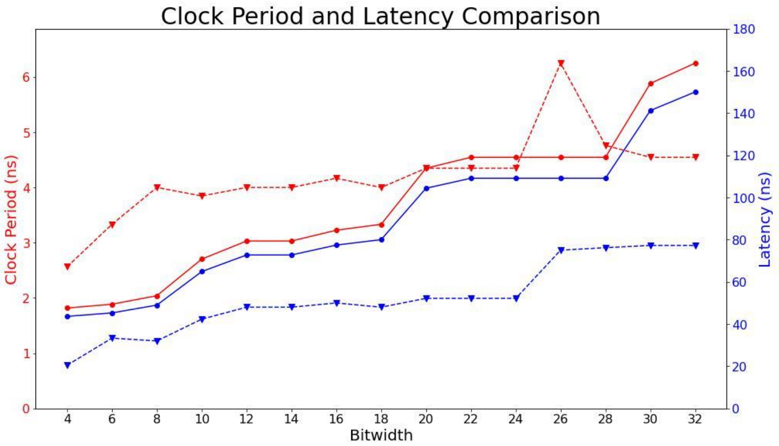


- > Besides latency, all ~ equal
  Bitwidths 7 and 8 packing reduces the LUT cost to 0.68x and 0.70x
- > Tradeoff of DSP usage not beneficial in our DSP limited design

One Layer - Initial Results

Dashed - HLS
Solid - SV





Resource: 1.28x more, Latency: 1.7x worse, Period: 1.46x better

HLS4ML is outperforming on almost all metrics.

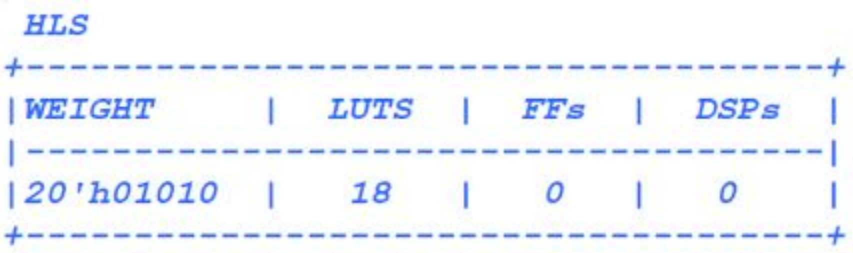
UNIVERSITY *of* WASHINGTON

Missing DSPs?

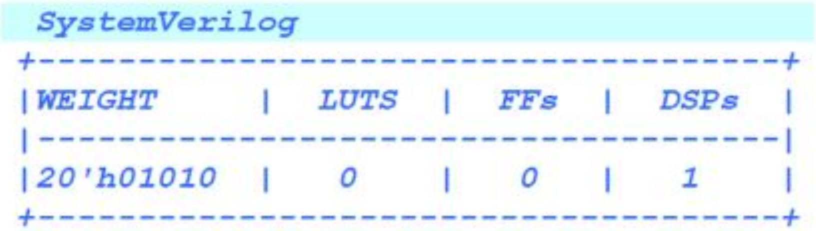
Hint from HLS4ML: DSPs decrease as bitwidth goes down

output = input*6 could instead be _output = (input<<2)+(input<<1)_

Shift-add module:



```
HLS
+-------------------------------------+
|WEIGHT      |  LUTS  |  FFs  |  DSPs  |
|-------------------------------------|
|20'h01010   |   18   |   0   |   0    |
+-------------------------------------+
```



```
SystemVerilog
+-------------------------------------+
|WEIGHT      |  LUTS  |  FFs  |  DSPs  |
|-------------------------------------|
|20'h01010   |   0    |   0   |   1    |
+-------------------------------------+
```

Shift-Add Capabilities

Vivado HLS

+-(input<<c1)+-(input<<c2)

for any c1 or c2

Vivado

+-(input<<c1)+-(input<<c2)

where c1 and c2 must be less than 3

or

+-(input<<c1)

for any c1

Shift-Add Module

Implemented a module that allows for DEPTH powers of 2 to be added

Depth = 0:

always uses DSP

Depth = 1:

input << i

Depth = 2:

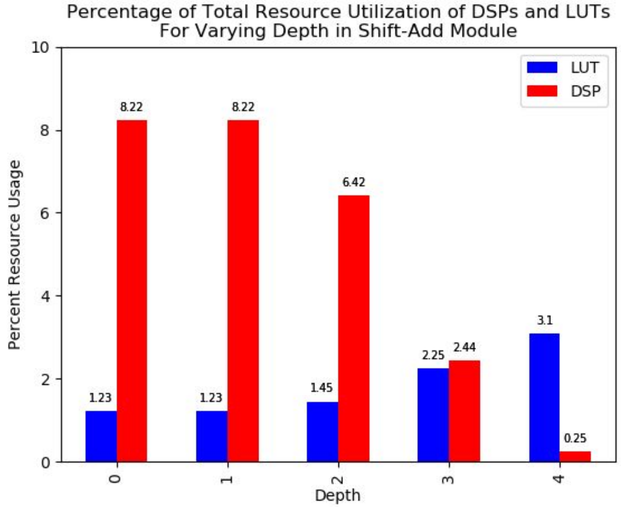
input <<i + input << j

Depth = 3:

input << i + input << j + input <<k

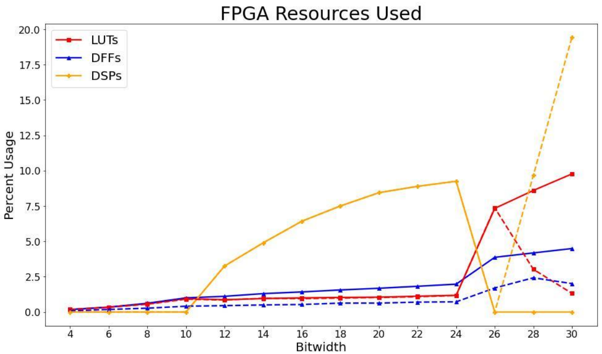
Depth = 4:

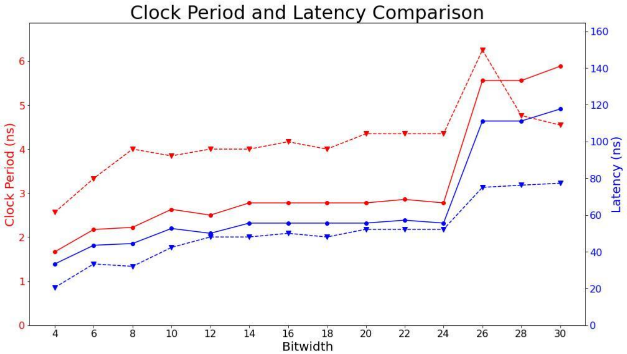
input << i + input << j + input << k + input << l



Updated One Layer Results

Dashed - HLS
Solid - SV





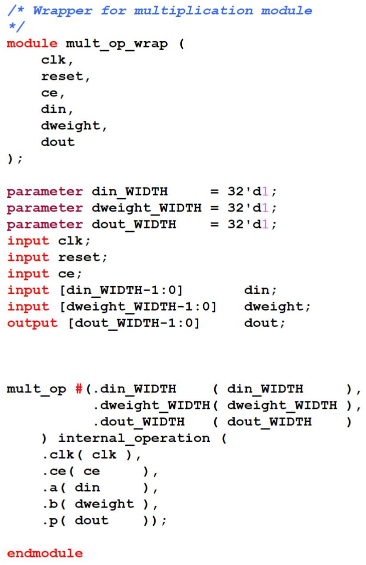
Resource: 0.97x less, Latency: 1.17x worse, Period: 1.54x better

DEPTH = 2
DSP usage identical for < 24, DSP > 24 → does not fit into 1 DSP anymore

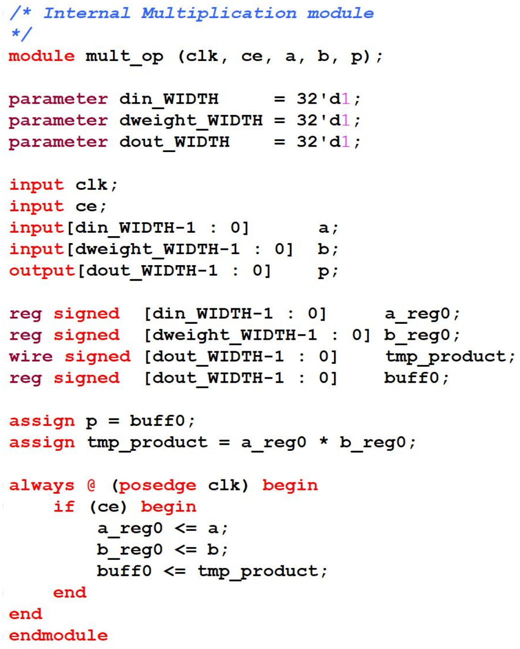
UNIVERSITY *of* WASHINGTON

DSPs > 24

HLS4ML “Magic Multiplier” Subroutine



```
/* Wrapper for multiplication module
*/
module mult_op_wrap (
    clk,
    reset,
    ce,
    din,
    dweight,
    dout
);

parameter din_WIDTH     = 32'd1;
parameter dweight_WIDTH = 32'd1;
parameter dout_WIDTH    = 32'd1;
input clk;
input reset;
input ce;
input [din_WIDTH-1:0]       din;
input [dweight_WIDTH-1:0]   dweight;
output [dout_WIDTH-1:0]     dout;



mult_op #(.din_WIDTH    ( din_WIDTH     ),
          .dweight_WIDTH( dweight_WIDTH ),
          .dout_WIDTH   ( dout_WIDTH    )
    ) internal_operation (
    .clk( clk ),
    .ce( ce     ),
    .a( din     ),
    .b( dweight ),
    .p( dout    ));

endmodule
```



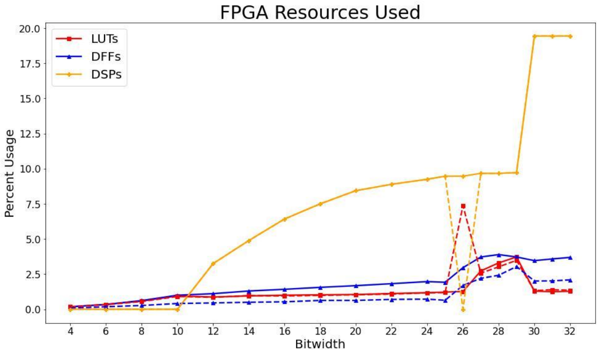
```
/* Internal Multiplication module
*/
module mult_op (clk, ce, a, b, p);

parameter din_WIDTH     = 32'd1;
parameter dweight_WIDTH = 32'd1;
parameter dout_WIDTH    = 32'd1;

input clk;
input ce;
input[din_WIDTH-1 : 0]      a;
input[dweight_WIDTH-1 : 0]  b;
output[dout_WIDTH-1 : 0]    p;

reg signed  [din_WIDTH-1 : 0]     a_reg0;
reg signed  [dweight_WIDTH-1 : 0] b_reg0;
wire signed [dout_WIDTH-1 : 0]    tmp_product;
reg signed  [dout_WIDTH-1 : 0]    buff0;

assign p = buff0;
assign tmp_product = a_reg0 * b_reg0;

always @ (posedge clk) begin
    if (ce) begin
        a_reg0 <= a;
        b_reg0 <= b;
        buff0 <= tmp_product;
    end
end
endmodule
```

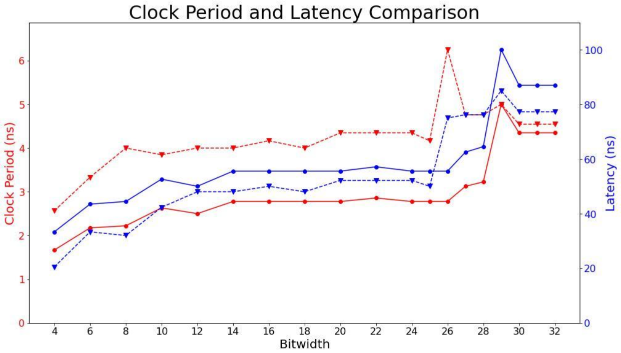
UNIVERSITY of WASHINGTON

Updated Results

HLS4ML “Magic Multiplier” Subroutine

Dashed - HLS
Solid - SV

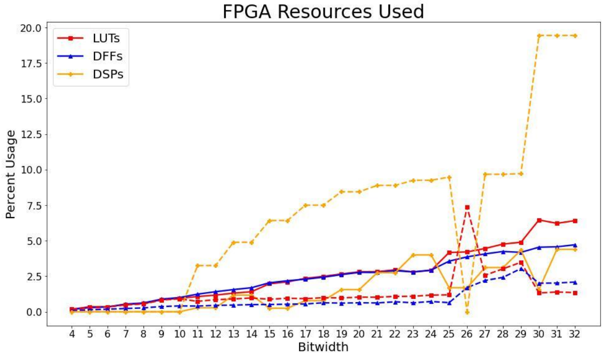


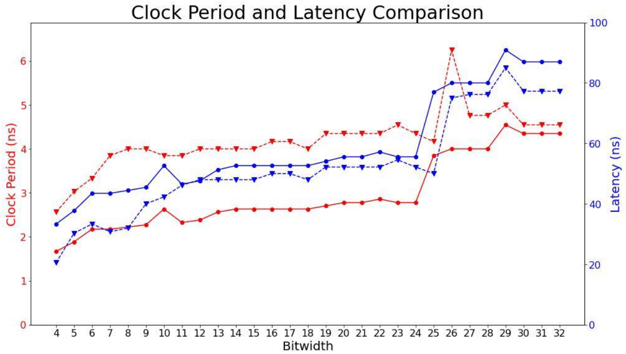


Resource: 1.03x more, Latency: 1.11x worse, Period: 1.44x better

Optimized Results

Dashed - HLS
Solid - SV





DEPTH VALUE:	2	3	4	5	6

Resource: 0.49x less, Latency: 1.12x worse, Period: 1.49x better

> **Tuning of shift-add DEPTH parameter based on optimal results per bitwidth**

Major Takeaways from One-Layer Model

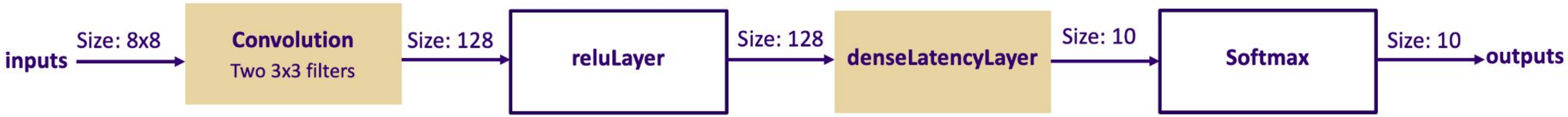
- DSP packing is not beneficial for multiplication-heavy algorithms such as these ML ones

- HLS4ML handles DSPs better than the tools normally allow for

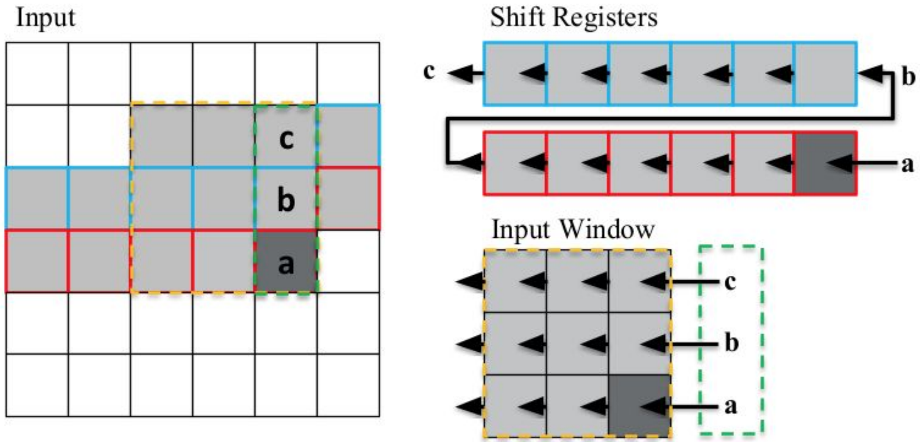
- HLS4ML multiplier subroutine allows for DSP usage at higher bitwidths

Benchmark 2

CNN Model



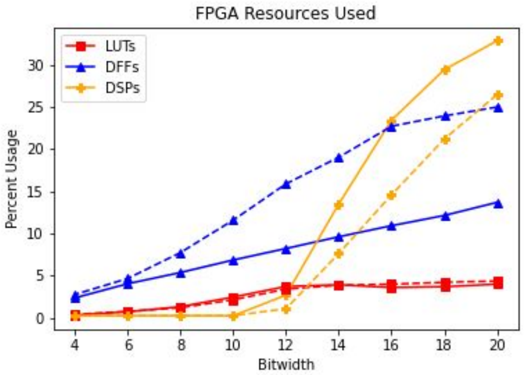
Convolution Streaming Method

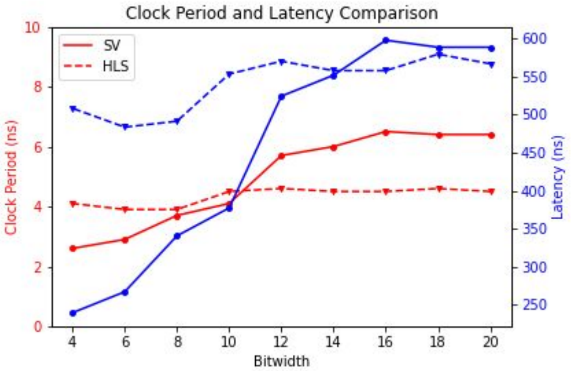


Line Buffer approach. Shift Register elements (red and blue) are shifted by one index. Input window buffer (orange) is updated with concatenation (green) of popped pixels—**b** and **c**—and input **a.**

CNN Model Initial Results

Dashed - HLS
Solid - SV



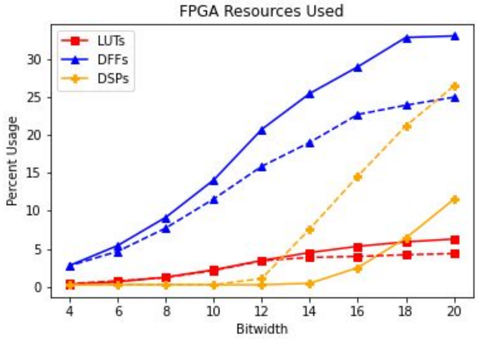


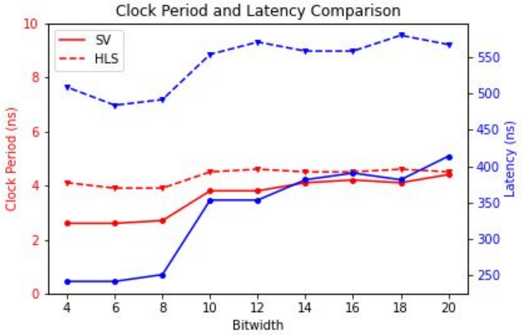
Resource: 0.82x less, Latency: 1.67x worse, Period: 1.13x worse

HLS4ml outperforming in all resources, except for DFFs

Optimized Results

Dashed - HLS
Solid - SV





Resource: 1.23x more, Latency: 1.19x better, Period: 1.21x better

DFFs become the limiting factor.
Shift-add DEPTH of 3 (now use 0.58x DSPs)

Conclusions

- HLS4ML is leveraging the power of Vivado HLS in ways that normal optimizations do not
- To achieve the same resource usage, we had to mimic HLS results

Should we ever hand code again?
Depends on the application. HLS4ML does these specific models very well, but does it scale?

Next Steps

Using our two models, build larger and more applicable models to see how our results scale.

Encoder model

Convolution with Stride of 2

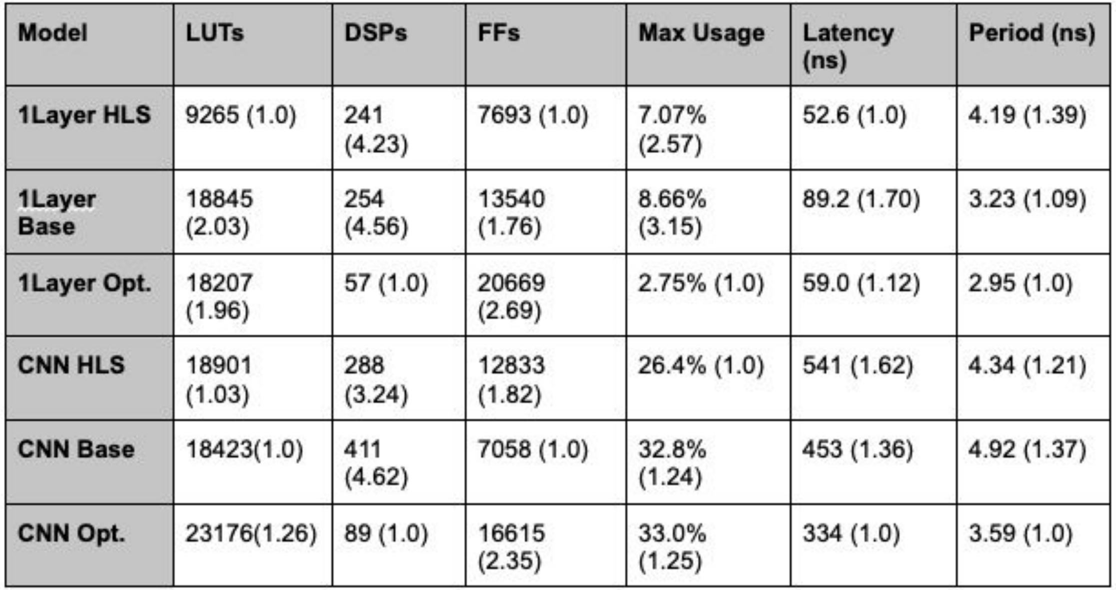
Reuse of 3 and 9

Jet Tagger

Introducing more complex layers - Batch Normalization

Questions?

Overall Results



Model	LUTs	DSPs	FFs	Max Usage	Latency (ns)	Period (ns)
1Layer HLS	9265 (1.0)	241 (4.23)	7693 (1.0)	7.07% (2.57)	52.6 (1.0)	4.19 (1.39)
1Layer Base	18845 (2.03)	254 (4.56)	13540 (1.76)	8.66% (3.15)	89.2 (1.70)	3.23 (1.09)
1Layer Opt.	18207 (1.96)	57 (1.0)	20669 (2.69)	2.75% (1.0)	59.0 (1.12)	2.95 (1.0)
CNN HLS	18901 (1.03)	288 (3.24)	12833 (1.82)	26.4% (1.0)	541 (1.62)	4.34 (1.21)
CNN Base	18423(1.0)	411 (4.62)	7058 (1.0)	32.8% (1.24)	453 (1.36)	4.92 (1.37)
CNN Opt.	23176(1.26)	89 (1.0)	16615 (2.35)	33.0% (1.25)	334 (1.0)	3.59 (1.0)