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

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

University of Washington, Dept. of Electrical and Computer Engineering
[†] University of Washington, Dept. of Physics

Abstract

High-level synthesis (HLS) offers the promise of simpler and easier hardware development, but at a cost. In this paper we consider the application of high-level synthesis to machine learning applications, seeking to quantify the resource and performance costs of this technique within the widely used HLS4ML framework. By creating carefully optimized SystemVerilog versions of identical HLS4ML designs, we demonstrate that the HLS designs are very competitive with hand-optimization techniques. We also identify weaknesses in the existing tools, and develop work-arounds to help provide significant quality improvements.

Introduction

High-Level Synthesis (HLS) tools are based on a compelling premise: a designer expresses the intent of their computation in a higher-level language, such as C or C++, and automatic software flows translate these intentions into hardware realizations. However, this generally means giving up hand-tuning the implementation, thus eliminating opportunities for area, power, and performance improvements from such hand-tuning. Thus, the viability of HLS tools depends on a tradeoff: are the ease-of-use benefits worth the resulting implementation quality impacts?

It has been previously shown [Xu10] that early HLS provides a significantly faster development time at the cost of the quality of the overall design. Studies comparing HLS to custom designs reported that the development time is about 2x - 4.4x faster using HLS as opposed to doing a custom hardware design [Xu10, Cornu11, Homsirikamol14, Andre18]. However, in using HLS there was a quality loss of roughly 2x in comparison to a custom design due to factors like longer clock periods, significantly higher resource utilization, etc. [Xu10, Pelcat16, Andre18]. In our experience current generation HLS tools retain this development time advantage; in this paper we seek to quantify the quality losses due to these tools within the machine learning domain.

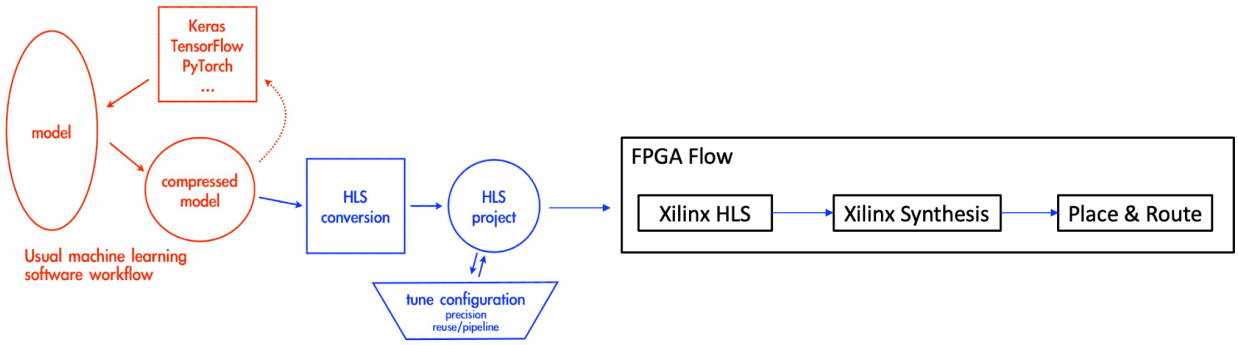


Figure A: Overview of the HLS4ML flow. Standard Verilog designs only go through Synthesis and Place & Route.

We focus on hardware implementations of machine learning algorithms within the HLS4ML framework [Duarte18]. HLS4ML is a widely used high-level synthesis based machine learning flow, that automatically translates from ML-specific design languages such as Keras, PyTorch, and TensorFlow to FPGA-based implementations via the vendors' HLS flows. In this way data scientists and other ML users can produce high-performance hardware implementations automatically for their computations, requiring little or no FPGA-specific expertise.

HLS4ML has been used for a wide range of applications, particularly for low-latency high energy physics applications. There have been compelling results reported for HLS4ML applications including:

- Bottom quark jet detection (B-tagging) within the CMS experiment at the Large Hadron Collider [CMS22].
- Multi-FPGA partitioning and inference for ResNet50 [Tarfdar21].
- Recurrent neural network processing, in GRU and LSTM, for jet identification and the Quickdraw datasets [Khoda22].

To measure the costs of an HLS-based implementation strategy, we have developed custom Verilog-based implementations of two HLS4ML applications: a Keras1_Layer model trained on the UCI Human Activity Recognition dataset [Anguita13], and a Keras_conv2d trained on the MNIST handwritten digit database. These are simple machine learning models, which can be carefully hand-optimized for an efficient implementation, and represent core building blocks of larger deep learning models. In this work our group, which has several decades of experience in creating efficient FPGA-based hardware, carefully crafted custom implementations of each of these systems, as well as automated flows to import the specific trained weight files from the ML models into our hand-optimized designs. We have also verified that the hls4ml designs (HLS) and our hand-optimized System Verilog (SV) implementations are bit-accurate. Note that while many implementations of neural networks will trade hardware complexity with computational accuracy (i.e. reduce the floating-point software implementation to 16-bit fixed-point integers, at

a loss of 2% in recognition accuracy), in this paper we are comparing HLS and SV flows, and thus implementations perform exactly the same computation, using exactly the same numeric format, and producing exactly the same accuracy. In fact, we will use bitwidth as an independent variable, comparing the two versions at different bitwidths to help demonstrate some tradeoffs in the implementation flows.

In this work we compare the HLS and SV designs in both resource usage and performance. We target a Xilinx Virtex 709 FPGA. For performance, we consider both clock period (and thus throughput, since the initiation interval is the same in the two designs), as well as latency of the overall computation. For resource usage, we need to consider the usage of generic fabric LUTs and DFFs, as well as hard elements including DSPs and BRAMs. These will be presented as the fraction of the available resources used. Note that there is a tradeoff in resource usage, where for example a given multiplication could either be performed in the DSPs, or instead mapped to LUTs. To create an overall resource usage metric, we will use the "max resource usage", which is the maximum resource utilization amongst all of the used resource classes. Thus, if a design uses 10% of the LUTs, 15% of the DFFs, and 25% of the DSPs, its max resource usage is 25%. Intuitively (and ignoring that FPGA tools never successfully map a 100% resource utilization design), this means that we could only fit 4 copies of the design into the FPGA, or could only fit into an FPGA ¼ the size, since the DSPs would otherwise run out.

We do not have specific numbers on the relative development time of the HLS and SV versions, though the order of magnitude improvement in HLS development time found in [Xu10] is likely a significant under-estimate for HLS4ML. As an unfair comparison, our SV implementation of our 1-layer model represents roughly 1680 lines of synthesizable code, while the corresponding HLS4ML is only 210 lines of code. We view this comparison as unfair since we had to develop the entire SV design from scratch, yet the library elements embedded in the HLS4ML system itself are not counted.

Background

The Verilog code produced by HLS, as well as our SV implementation, were both run on Vivado 2020.1. We target the Xilinx Virtex 709 FPGA. The SV version is written in SystemVerilog, and compiled with Vivado synthesis, place and route. The HLS version starts as a Python YAML file with additional files for configuration, weights and biases. This gets converted by HLS4ML into C++ code before being built by Vivado HLS into Verilog code. Using Vivado, the generated Verilog is then synthesized, placed and routed in the same manner as the SV version, allowing us to compare reports one-to-one.

Our Two Benchmarks

In order to understand the effectiveness of HLS4ML, we used two simple neural network models as our benchmarks – a one layer neural network (KERAS-1D) and a two-dimensional convolution neural network (KERAS-CONV-2D).

The one layer neural network, or one layer model, is a simple dense neural network. This benchmark sequentially processes inputs through four layers: dense, ReLU, dense, and

Sigmoid (Figure B). Each dense layer performs matrix multiplication on an input vector and kernel matrix. The output of these layers is then passed through activation functions such as ReLU and Sigmoid. The ReLU activation layer outputs element-wise `relu(x)= max(x, 0)`. The Sigmoid processes each value in its input and implements `sigmoid(x)= 1 / (1 + exp(-x))`.The one layer implementation is fully unrolled, with an initiation interval of 1. This allows for a new computation to be started every clock cycle.

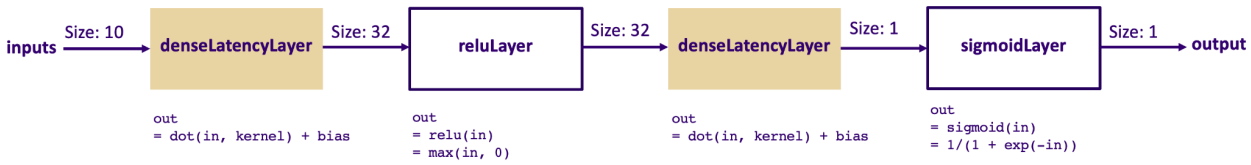


Figure B. The One Layer design.

The second benchmark is a 2D convolutional neural network (CNN), or CNN model. This algorithm is made up of 4 layers: convolution, ReLU, dense, and Softmax. The first layer, a convolution layer, utilizes two 3x3 filters being convolved over an 8x8 matrix, thus resulting in `8x8x2 = 128` outputs. Convolution is typically used in image processing and data recognition due to its ability to go over a large matrix and analyze the pixels based on the filters provided. From the convolution layer we go to the ReLU layer. The dense latency layer then takes the ReLU output and splits it into 10 groups, representing 10 possible classifications. Finally, the neural network ends with a Softmax, which converts the classifications into class probabilities, such that they sum to 1. This is done by taking the exponentiation of each input and dividing it by the sum of all of the values, $output[i] = \frac{e^i}{\Sigma e^j}$.

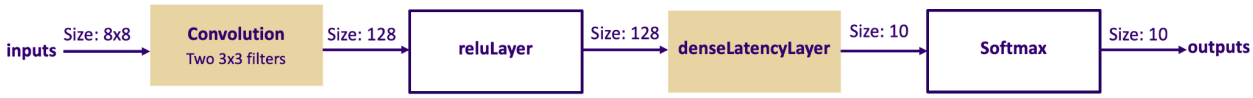
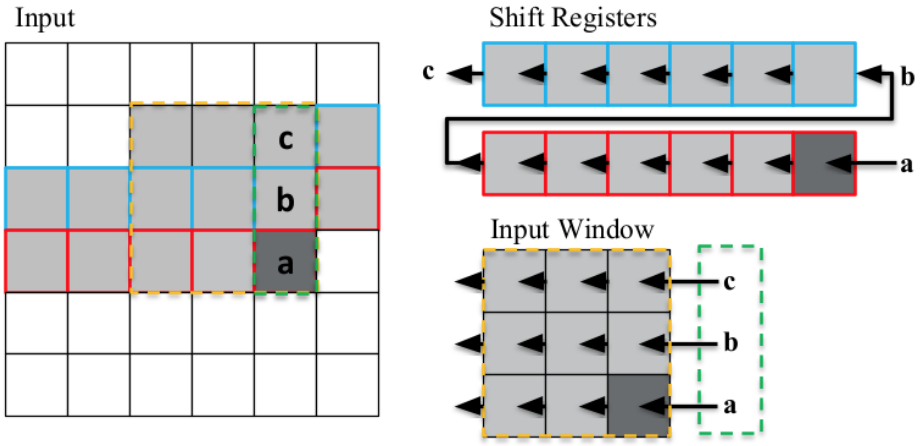


Figure C. The CNN design.

For the CNN, both HLS and SV do not read in the whole input matrix at once; instead the data is streamed in. This iterative approach makes it so one input of a matrix is read in each clock cycle. For an 8x8 matrix, the convolution layer itself takes 64 clock cycles in total. This will be referred to as the streamed method and it allows for a much more efficient implementation as the overall resources for the inputs are significantly lower, as well as creating a pipelined convolution layer [Lin21]. The convolution is done in chunks and shift registers will hold onto the data as it passes through the rows of the kernel, as shown below.



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.**

Figure D: Line buffering for the 2D convolution [Lin21].

These two benchmarks contain basic layers that are used in most machine learning algorithms. By seeing how HLS is able to perform with these layers, we are able to evaluate its performance.

Bitwidth and Fixed Point Numbers

In machine learning (like many algorithms) there is a tradeoff between the amount of resources used, and the accuracy of the resulting computation. Since we are targeting FPGAs, we will avoid floating-point computations by converting all of the operations to fixed-point representations. However, a designer can choose to use very wide values, which have the greatest accuracy but at a significant hardware cost, or very small bitwidths, which reduce hardware costs but often with an accuracy loss. For our testing, we explored a wide range of bit widths to see how the resources scaled.

All numbers in the benchmarks utilize fixed point 2's complement numbers. We will refer to all of our bit widths in terms of how many total bits are used. For simplicity, we set the number of fractional bits to half the total bitwidth, rounded up. For a 16-bit fixed point number with 8 fractional bits (and thus 7 integer bits plus a 2's complement sign bit) the decimal value would be evaluated as the sum of the bits with the following powers of 2:

	S	I	I	I	I	I	I	I	F	F	F	F	F	F	F	F
Power of 2:	<sign>	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8

Verilog Library

Our SV implementation is highly pipelined to support high throughput and low latency computations. The primary layers in the SV implementations contain matrix multiplications. Depending on the bitwidth, these expensive computations can heavily stress Xilinx's multiplication support. There are also non-linear activation functions: ReLU, SoftMax, and Sigmoid. ReLU is a simple comparison operation, converting any negative value to zero. For

SoftMax and Sigmoid, which are complex operations involving division and exponentiation, we use table lookups. These tables are read in and stored in LUTs or BRAMs.

HLS4ML is generally targeted to inference applications that use fixed weights, which are pretrained and compiled into the implementation itself. While it is possible to have the designs support loading of new weight matrices, this is typically not done, since the implementations can be significantly optimized and simplified by using fixed constants.

To support fixed weights in our hand-coded versions, we created a Python converter that takes Keras weight files in plain text and converts them to SystemVerilog constants. We also pass constants between hardware modules via SystemVerilog parameters, which allows the compilation tools to perform constant folding and related optimizations.

Multiplier Packing

Because neural network algorithms are so multiplication-intensive, efficient use of the DSP resources is important to achieving a good implementation. Since the DSPs built into Virtex 7 FPGAs support 25x18 multiplication, it is straightforward to handle a single multiplication for up to 18 bit numbers within a single DSP, though for smaller bitwidths they will use that DSP inefficiently.

DSP packing [Fu17] is a mechanism to more efficiently support low bitwidth neural network computations within an FPGA's DSP resources. Most neural network layers have multiple outputs corresponding to different neurons, and inputs to the layers are multiplied by several different constant weights, with each weight leading to a different layer output. Since the DSP supports 25x18 multiplication, for small bit widths (<= 8), it is possible to combine weights via the DSP pre-adder, with one of the weights shifted up to higher, normally unused bit positions. By combining two weights via ports A and D (see Figure E), their sum can be multiplied by the input on port B and then separated on the output P.

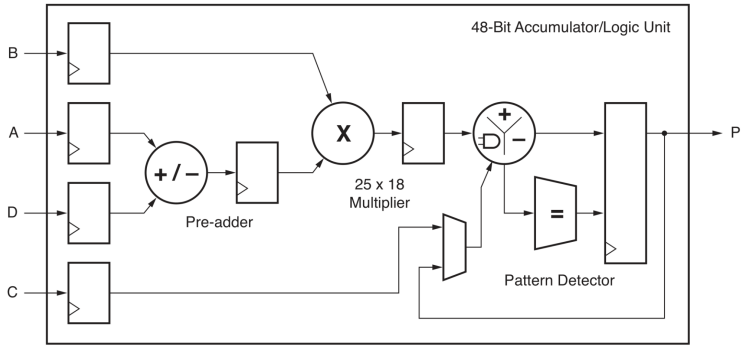


Figure E: Xilinx DSP Slice [Xilinx18]

For an 8-bit example, the DxB result will be 16-bits, so in order for the AxB to not overlap on the output, we must right-shift the A weight by at least 16 bits:

```
S[XXXXXXXX]00000000000000000000 (Weight A) 1111100000000000000000000 (-16) << 16
SSSSSSSSSSSSSSSSSSSS[XXXXXXXX] (Weight D) 11111111111111111100101 0 (-54)
                                      Sum: 11110111111111111111001010
```

When multiplied by an input (via the B port), a conditional correction term (decimal 1 shifted left 16 bits) is applied whenever the signs of the Port D weight and Port B input differ, since the negative result in the lower bit term (bits 15:0) steals a “1” from the upper bit term (bits 31:16). If we take our example weight and multiply by an input of positive 46, this can be shown:

```
                  1111011111111111100101 0
                                  *00101110
...1111111111010001111111011001001100
             (-737)          (-2484)
  CORRECT:   (-736)          (-2484)
```

This correction term is applied conditionally via the C port of the DSP slice. The full condition is: $C = \left(w_{D,sign} \oplus I_{B,sign}\right) * \left(w_D \neq 0\right) * \left(I_B \neq 0\right)$ due to the fact that multiplication by zero does not exhibit the same stealing behavior since a negative multiplied by a zero remains a zero and does not cause the lower bit term to become negative.

If the bitwidth is reduced to 5 bits or less, the port size permits multiplying 3 weights by a single input, and requires two conditional bits in the correction term.

Applying DSP Packing

Initially, we explored how DSP packing can help reduce resources within the SV implementation of the one layer benchmark. For small precision implementations (8 bits and under), we can pack the ports of the DSP slices. This requires some hardware overhead to perform corrections on the final result due to interactions that occur in the multiplication process. These results are shown in Figure F.

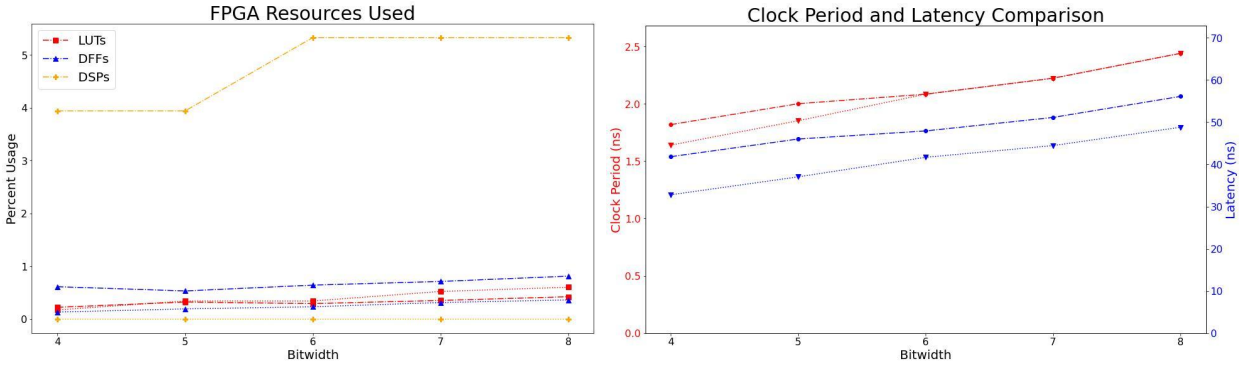


Figure F: Packed vs. unpacked multiplication. Resource type is indicated by symbol and line color. Dashdotted lines are packed, dotted are unpacked.

Latency is consistently better for the unpacked version, and clock speed is either better, or equal, in the unpacked version. For resource usage, observe that we can only pack relatively small multiplications (8 bits or less) together into a DSP block, and these multiplications can instead be implemented in LUTs relatively easily. Thus, forcing low bitwidth multiplications into DSPs is not a good tradeoff. However, if there was a design that was truly LUT limited, and both DFFs and DSPs can be considered "free", then there is a benefit for packing at bitwidths 7 and 8, where packing reduces the LUT cost to 0.68x and 0.70x respectively. However, unless there are spare DSPs and the other resource and latency hits are acceptable, it is best to leave DSP packing disabled. For the rest of this paper we will not use multiplier packing.

Initial Results - One Layer model

In Figure G we present comparisons of the SV and HLS implementations of the one layer model. The SV versions are solid lines, and the HLS versions are dashed lines, with color indicating resource type.

While the graphs are somewhat "twitchy" (strange irregularities that, we will show later in the paper, actually expose interesting features of the toolchain), the story is fairly consistent – the HLS designs outperform the optimized SV on almost all metrics. While for bitwidths <= 18 the SV version has a higher clock rate, by an average of 1.46x, the two versions have roughly the same clock rate for >=20 bits. The SV versions have a 1.7x worse latency overall, use roughly 1.76x more flipflops and 1.08x more DSPs, and use between 1x to 10x the number of LUTs (excluding the 26 bitwidth). In terms of maximum resource usage, the area is dominated by fabric resources for up to 10 bits, and DSPs at most higher bitwidths.

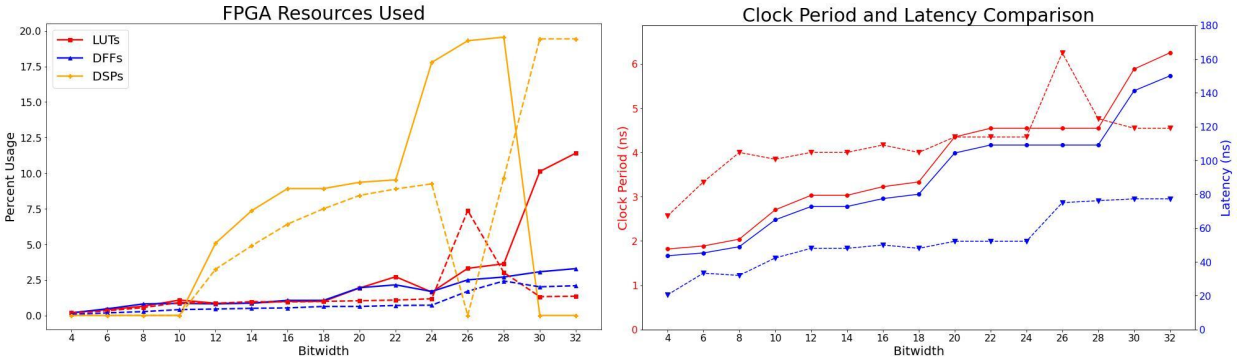


Figure G: Initial results for SV (solid) and HLS (dashed) implementations of the one layer model.

Although we carefully optimized the SV implementation, using standard hardware optimization techniques, heavy pipelining, constant folding, and neural network-specific DSP optimizations, the automated tools are able to produce better solutions on almost all metrics. While this could be written off as "well, those guys just don't know what they're doing", as we will show in the next few sections the differences are explainable (including many of the "twitchy" values), and show that there are optimizations available in the HLS flow that allow them to shine in comparison to best-practice Verilog hardware design.

Pipelining and Performance

The primary goal of the initial SV version was to be functionally identical to the HLS one layer model we were evaluating, with as high a throughput as possible. Thus, we essentially pipelined everything. For example, in the initial version the adder tree for the accumulate operation would add two terms per pipelining stage in a tree structure. By experimentation we found that increasing this to four terms per pipelining stage did not have any negative effect on clock speed, while reducing the number of pipelining stages, total LUT logic, and DFFs utilized. Furthermore, we experimented with inferring 1 pipeline stage and 3 pipeline stage DSP configurations, though we ultimately stuck with our original 3-stage configuration since we found a single cycle DSP was limiting our clock speed with little benefit except for DFF savings.

The Missing DSPs

In the graph of DSP Utilization in Figure G, notice that for bitwidths 12 to 22 the HLS versions consistently use a few less DSPs than the SV versions. This is somewhat strange, since both versions implement multiplication by simply using their source languages' "*" operator, and multiplications of those widths should fit easily into a single DSP block.

A hint of what is happening can be found by noticing that, as the bitwidth goes down from 22 to 12 bits, the number of DSPs used also decreases, even though the number of multiplications done by the algorithm stays constant. The difference is the use of shift-add arithmetic in LUTs for constant multiplications with "easy" constants. For example, consider calculating `output = input*6`. We could use a DSP to implement this, or instead simply compute `output = (input<<2)+(input<<1)`. Since constant shifts are essentially free in an FPGA, the replacement of a DSP with an adder is a smart optimization. As the bitwidth of the constants goes down, more of the constants are "easy", and more multiplications are converted to LUT-based shift-adds.

However, if the two source-codes are just using the "*" operator, and relying on the Xilinx mapping flow to convert to shift-adds where appropriate, why is the HLS version using fewer DSPs than the SV version? The answer is that both the Xilinx HLS and the Xilinx Vivado synthesis tools (see figure A) perform this transformation, and the HLS tool has a more powerful optimizer.

To test this, we created simple Vivado-HLS and SystemVerilog designs that just performed a single constant multiplication, and compiled them through the two toolflows. The constants were set to a range of values, both obvious "easy" constants, as well as constants chosen from the 1-layer model that the HLS tool is compiling into shift-adds, while the SV designs do not. Through this we found that the HLS tool appears to convert to shift-add any constant multiplication where the constant is in the form "+-(input<<c1)+-(input<<c2)". For the standard Xilinx Vivado flow, the conversion appears to be limited to "+-(input<<c1)+-(input<<c2)" where c1 and c2 are 3 or less, as well as "+-(input<<c)" for any c. This difference, where the HLS tool has a more powerful conversion optimization than the main Vivado tools, was quite surprising to us, since this type of constant folding and multiplier conversion has been a core technique for hand designs to FPGAs for decades.

```
#include "ap_fixed.h"


typedef ap_fixed<20,10> dval_t;
typedef ap_fixed<20,10> dw_t;

dval_t const_mult(dval_t input_val) {

    dw_t weight = 4.015625;
    return input_val * weight;
}
```

HLS

WEIGHT	LUTS	FFs	DSPs
20'h01010	18	0	0

```
module use_dsp #(
    parameter signed WEIGHT = 20'h01010,
    parameter int    WIDTH  = 20,
    parameter int    NFRAC  = 10
) (
    input  logic signed [WIDTH-1:0] in,
    input  logic                    clk,
    output logic signed [WIDTH-1:0] out
);

    // tmp output
    logic signed [WIDTH+NFRAC-1:0]  out_tmp;

    assign out_tmp = $signed(in) * $signed(WEIGHT[WIDTH-1:0]);
    assign out     = out_tmp[WIDTH+NFRAC-1:NFRAC];

endmodule
```

SystemVerilog

WEIGHT	LUTS	FFs	DSPs
20'h01010	0	0	1

Figure H: Equivalent HLS and SV code that gets implemented in LUTs and DSPs respectively.

SystemVerilog Shift-Add

In order to improve the SV version's DSP usage, we developed our own constant multiplication module. This unit takes in an input, plus a constant weight as a parameter, and automatically determines whether to use shift-add or a standard "*" operation (which is then converted by Vivado to a DSP). The module, via SystemVerilog functions and generate statements, iteratively converts "input*weight" to "$\pm\ (input\ <<\ c)\ +\ input\ *\ (weight\ \pm\ 2^c)$", where "$\pm\ 2^c$" is the power of two which most reduces the magnitude of the weight towards zero. This transformation is applied iteratively, up to DEPTH times, where DEPTH is a configurable constant. If after DEPTH applications of the rule, the remaining "(weight-2^const)" term is non-zero, the tool decides that the multiplication should not be done via shift-add, and instead should use a DSP.

DEPTH is a tuning factor. For DEPTH = 2, we can support "+-(input<<c1)+-(input<<c2)", while DEPTH = 3 can support "+-(input<<c1)+-(input<<c2)+-(input<<c3)". DEPTH = 1 only allows powers of two, and DEPTH = 0 turns off the optimization completely.

A comparison of the resource usage for various DEPTH settings is given in Figure I. As can be seen, none of the constants are pure powers of two, so DEPTH = 1 has no benefit. For DEPTH above 1, there is a steady decrease in the use of DSPs, though a steady increase in the LUT usage. However, it is important to realize that the costs of a DSP and a LUT are very different. In Figure J, we report resource usage as a fraction of chip resources. As can be seen, DEPTH = 3 is the best match to the resource mix found in our chosen FPGA. A change in DEPTH has no impact on the amount of DFFs.

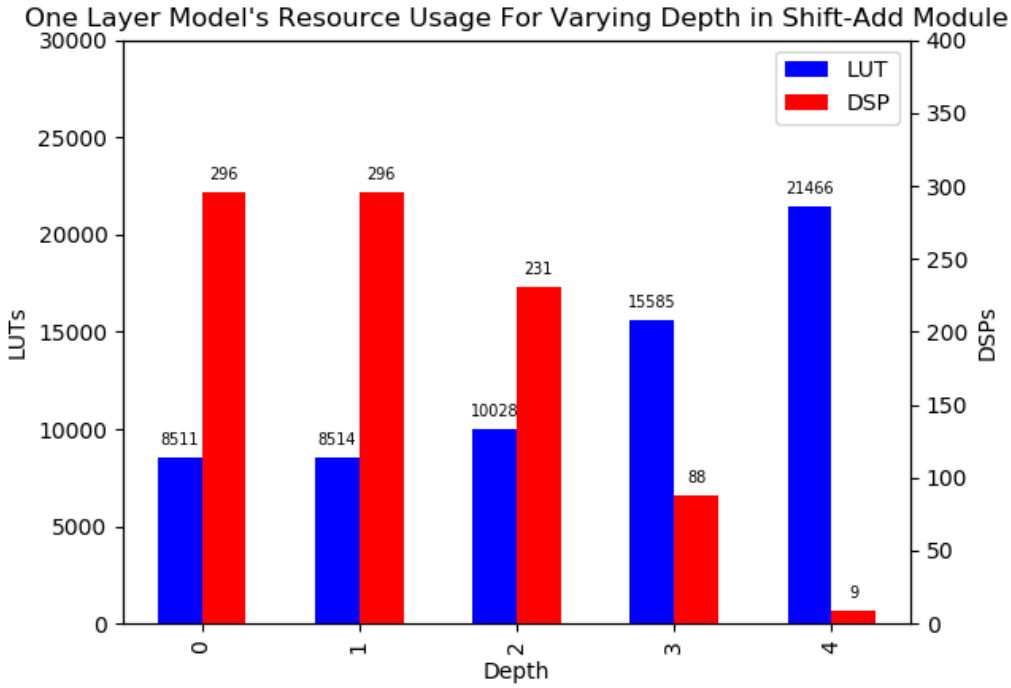


Figure I: Resource usage vs. DEPTH parameter for constant multiply. Computed for an overall bitwidth of 16, on the 1-layer benchmark.

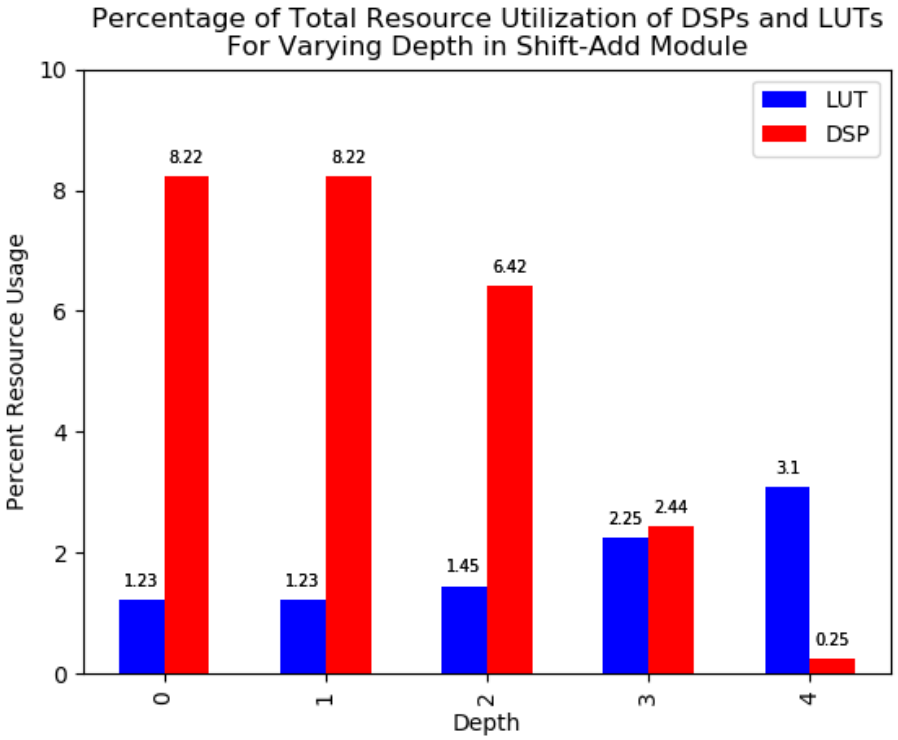


Figure J: Fraction of FPGA resources used vs. DEPTH parameter for constant multiply. Computed for an overall bitwidth of 16, on the 1-layer benchmark.

Improved SV With Manual Shift-Add

As discussed previously, based on intuition gained from the HLS version, we mimicked those optimizations in the SV version. This included a manual shift-add routine with DEPTH=3, and tweaking of the pipelining of the design to better use flipflops throughout the pipeline. After these changes, we began to see the SV design's resource utilization converge with the HLS design (Figure K).

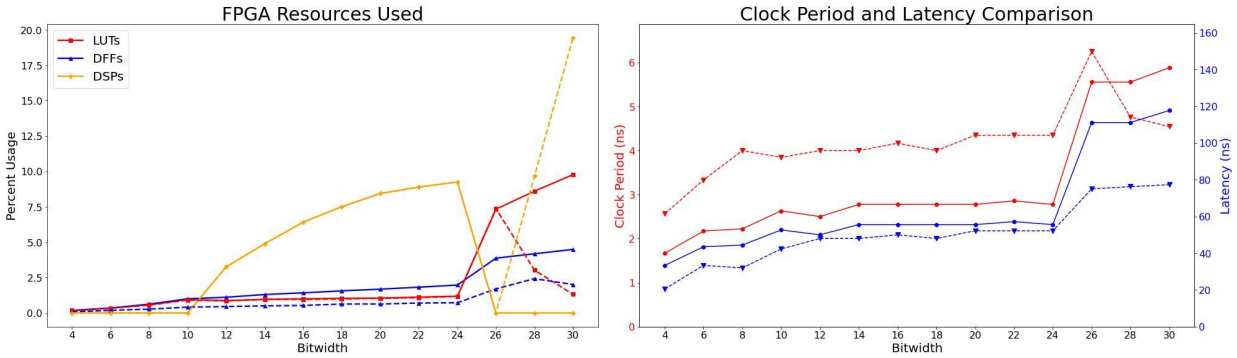


Figure K: 1-layer model results after applying manual shift-add calculations.

As can be seen from the graphs, there is a transition in behavior at bitwidths above 24, when the multiplications stop fitting into single DSPs within our target FPGA; we will consider these high bitwidth cases below. For bitwidths of 24 or less, the DSP usage is identical between the HLS and SV implementations. For these cases the SV version uses on average 2.55x more flipflops, and 1.03x more LUTs; the clock speed is 1.54x faster, while the latency is 1.17x worse.

Multi-DSP Support for High Bitwidths

While the previous optimizations took care of bitwidths of 24 or less, the results are quite different for bitwidths above this limit. Given the fact that the DSPs support 25x18 multiplications, this breakpoint makes some sense. However, why does the HLS version switch to using pairs of DSPs at high bitwidths, while the SV version abandons DSPs completely?

Recall that, outside of the "easy" weights that are converted to shift-adds, both the HLS and SV versions perform multiplication via the standard "*" in their respective source languages. In fact, the HLS compiler converts the C-based source code into Verilog which also uses the Verilog "*". Yet somehow the HLS version gets converted into 2 DSPs, while our SV version gets converted to LUTs.

The difference? The HLS code uses the multiplication subroutine shown in Figure L. Although it appears to be basic Verilog without any special features, this version compiles efficiently at high bitwidths, and similar versions in Verilog do not. We have experimented with different SV versions of the multiplication code, including identical pipelining stages, and have discovered

the following: if we call the HLS-supplied multiplier subroutine in our SV code, it compiles to DSPs, and if we use the "*" instead, it does not. Somehow, that HLS subroutine is recognized by Vivado, which then "does the right thing".

```
/* 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
```

Figure L: Parameterized "Magical Multiplication" Subroutine.

When we include the HLS magical multiplication subroutine into our SV implementation, we get the results shown in Figure M. Notice that now the SV and HLS results are identical in DSP usage. The only difference appears to be a bug in the HLS optimization flow – somehow when it switches from the straightforward DSP usage in bitwidths 25 or less, to the 1 DSP + LUT logic at bitwidths 27-29, it fails to catch the bitwidth 26 case. 26 bits seems to essentially "fall into the cracks" between the two optimization styles.

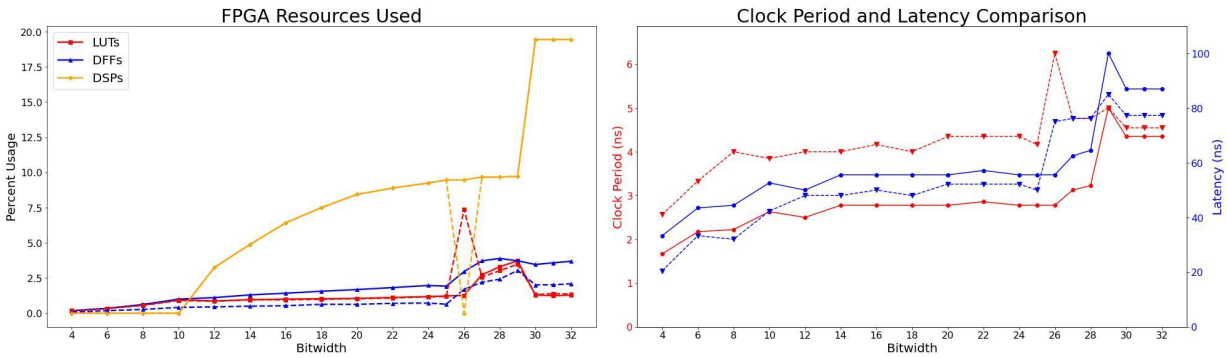


Figure M: Comparison of SV and HLS after applying manual shift-add and the magical multiplication subroutine.

Compared to the HLS system (excluding bitwidth 26), the SV version on average uses the same number of DSPs, 1.03x LUTs and 1.89x DFFs; the clock speed is 1.44x faster and latency is 1.11x worse. At a bit width of 10 or less the limiting resource for both systems is either LUTs or DFFs, while after a bit width of 10, the Xilinx tools begin to infer DSP slices for multiplication, and DSPs become the dominant limiting resource.

Up to this point, we had only tried to match the HLS DSP utilization, which corresponds to DEPTH=2 (meaning any weight that was a sum/subtraction of two powers of 2 was converted to a shift-add operation). However, since DSPs are generally the scarcest FPGA resource for this application, converting more complex weights to shift-add operations allows us to better balance between LUTs and DSPs. In Figure N we adjust the DEPTH, per bitwidth, based on maximum resource usage, and report the results.

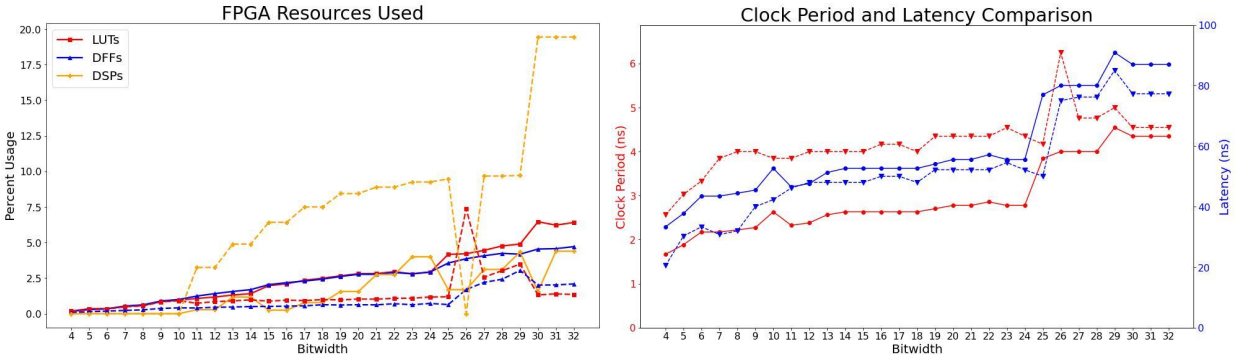


Figure N: Per-bitwidth DEPTH tuning. We use DEPTH=3 for bitwidths 11-14, DEPTH=4 for bitwidths 15-24, DEPTH=5 for 25-29, and DEPTH=6 for 30-32.

On average, the SV version uses 0.23x the number of DSPs, 1.97x LUTs and 2.69x DFFs; clock speed is 1.49x faster and latency is 1.12x worse. Since DSPs are so scarce a resource, it often made sense to convert more complex multiplications into LUTs from DSPs. As a percentage of total FPGA resources used, the greatest percentage of an FPGA resource any HLS bitwidth used was 19.44% (DSPs) while the greatest percentage of an FPGA resource any SV bitwidth used was 6.46% (LUTs). On average the SV design has 0.39x the maximum resource usage of the HLS design. By adjusting depth, SV is able to shift the implementation resources between LUTs and DSPs to whichever is more abundant.

CNN Implementation

So far, we have presented a comparison of SV and HLS for the one layer model. Although our initial implementation of the SV model (which corresponds to “best practice” FPGA hand design) compared fairly poorly to the HLS design, by learning from the HLS results we were able to reverse-engineer those optimizations into the SV version and significantly improve the results.

However, how much do these changes generalize? In this section we now apply these techniques to the CNN benchmark, which mirrors our actual development process. The CNN is a more complex neural network, with significant internal storage and staging as the convolution kernel is moved across the input array to produce the final result.

Below are our initial graphs displaying the comparison without the shift-add optimization as explained above. Note that the current HLS4ML system has a hard constraint of a clock period of at most 5ns. At bitwidths above 22, that constraint is not met. Thus, we will consider only 2-layer designs with bitwidths of 22 or lower.

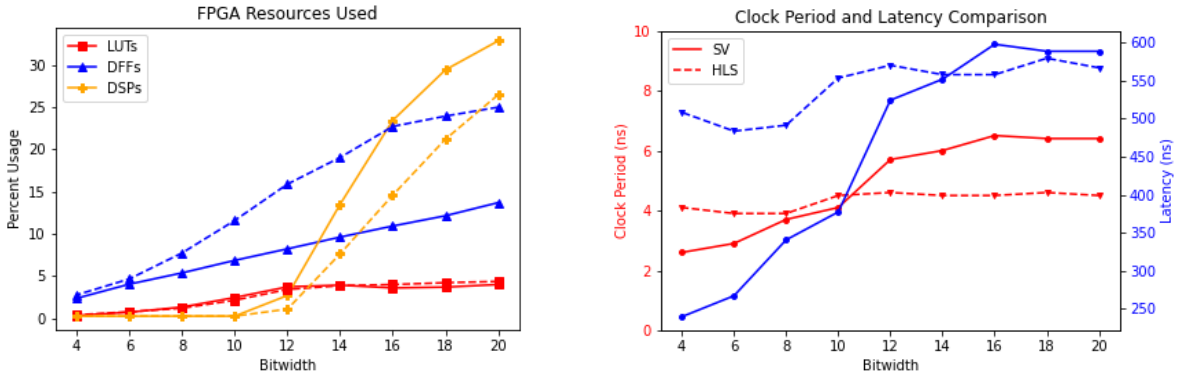


Figure O: Initial SV vs HLS results for the 2DConv benchmark.

On average the SV version is using 1.38x more DSPs, but has 1.67x better latency. The HLS version is outperforming our SV version in clock period by 1.13x. As a percentage of total FPGA resources used, the greatest percentage of an FPGA resource any HLS bitwidth used was 26.44% (DSPs) while the greatest percentage of an FPGA resource any SV bitwidth used was 32.81% (DSPs). On average the SV design has 0.94x the maximum resource usage of the HLS design.

For the first benchmark, after implementing the modifications we discovered for the first benchmark, we were able to see significant improvement in resource utilization as well as clock speed. Figure P shows the results of applying those techniques to the 2DConv benchmark.

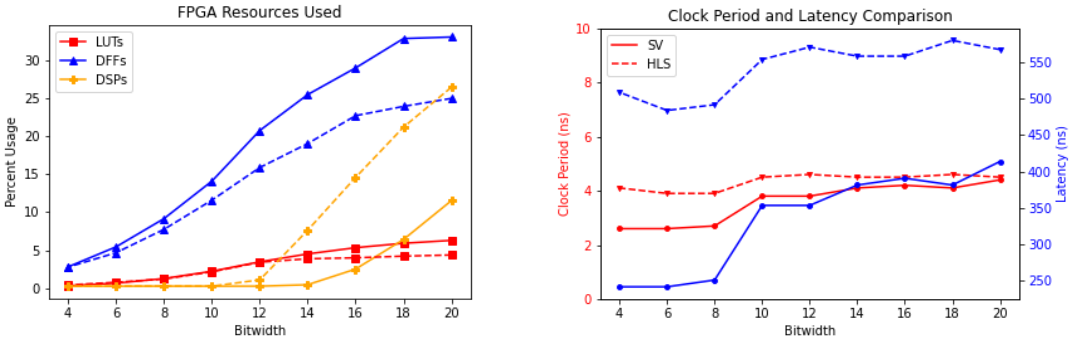


Figure P: Comparison of final SV and HLS results for the 2DConv benchmark.

Based on what we learned in the comparison of the 1Layer SV and HLS versions, we applied similar optimizations to the 2DConv benchmark, with results shown in Figure P. Our implementation becomes comparable to the HLS version in most resources, except we are outperforming in terms of DSPs and latency, but are using significantly more DFFs. Compared

to the HLS system, on average we used 0.58x the DSPs, and achieved a 1.21x faster clock period. Our latency continued to be better than HLS's: initially our design was 1.19x faster, and in the second implementation it became 1.62x times faster. However, our design becomes even more DFF-dominated, such that our max resource usage grows to 1.25x that of the HLS design.

Compared to our initial implementation, the amount of FPGA resources used increased. This is due to the shift from DSPs to FFs and LUTs. Note that all three designs make heavy use of SRL 16s to buffer data through the convolution layer; these are accounted for in the LUT usage. The HLS design uses SRL 16s to buffer data between the layers, while the SV versions use DFFs.

Overall Results

For the two benchmarks, averaged across all bitwidths, our initial hand SV designs had 1.22x-1.28x max resource usage, 1.70x-0.84x latency, and 0.77x-1.13x clock period compared to the HLS versions. After reoptimizing those designs based upon the optimizations observed in the HLS designs, our improved versions had 0.39x-1.25x max resource usage, 1.12x-0.62x latency, and 0.70x-0.83x clock period.

Table 1: Benchmark results, averaged across all bitwidths. Normalized results in parentheses, normalized to the best result amongst the three implementations.

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)

Conclusions

It is fairly common knowledge in the FPGA community that, while HLS can be an efficient way to quickly develop an implementation of a computation, hand optimization is the gold standard for achieving the highest quality implementation. After this investigation, we are not sure that is actually the case. Although we were eventually able to achieve hand-optimized results that were

in some sense equivalent or better than the HLS results, in many ways we “cheated” – we found optimizations in the HLS designs that could be imported into the SV designs, but many of them simply would not have been discovered in standard hand-optimization flows.

If HLS designs can be this competitive with hand designs, yet with SIGNIFICANTLY simpler development cycles, is this the death of hand designs? We think that viewpoint is somewhat premature. HLS4ML has the advantage of a very restricted application domain (deep learning models) which allows for many domain-specific optimizations to be embedded in the compilation flow. Also, the computations start as very highly parallel feed-forward networks, which are especially easy to implement as high-performance FPGA applications. Thus, in some ways we can view HLS4ML as something of a module generator, converting from a highly restricted domain into carefully optimized modules, albeit ones that are coded in HLS instead of SV. Whether this approach will also work in less structured, or less parallel, application domains is an open question.

However, there is a second way to look at these results - for the application domain of latency-sensitive deep learning applications, HLS can provide implementations that are very competitive with hand designs, to the point where hand-optimizing any but the most critical designs seems to be of limited utility. Hand design may not be dead everywhere, but it is quite suspect in at least this domain.

Acknowledgements
This research was funded in part by National Science Foundation (NSF) grants No. 1934360 and 2117997.

References

[Altoyan20] W. Altoyan and J. J. Alonso, "Investigating Performance Losses in High-Level Synthesis for Stencil Computations," 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2020, pp. 195-203, doi: 10.1109/FCCM48280.2020.00034.

[Andre18] Marc-André, Tétrault. 2018. “Two FPGA Case Studies Comparing High Level Synthesis and Manual HDL for HEP Applications.”

[Anguita13] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra and Jorge L. Reyes-Ortiz. A Public Domain Dataset for Human Activity Recognition Using Smartphones. 21th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2013. Bruges, Belgium 24-26 April 2013.

[Bailey15] Bailey. (2015). The advantages and limitations of high level synthesis for FPGA based image processing. Proceedings of the 9th International Conference on Distributed Smart Cameras, 134–139. https://doi.org/10.1145/2789116.2789145

[CMS22] CMS Collaboration, “Neural network-based algorithm for the identification of

bottom quarks in the CMS Phase-2 Level-1 trigger", June 2022, CMS-DP-2022-021, https://cds.cern.ch/record/2814728.

[Cornu11] Cornu, Alexandre, Steven Derrien, and Dominique Lavenier. "HLS Tools for FPGA: Faster Development with Better Performance." In Reconfigurable Computing: Architectures, Tools and Applications, 67–78, 2011. Berlin, Heidelberg: Springer Berlin Heidelberg, n.d. doi:10.1007/978-3-642-19475-7_8.

[Duarte18] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, Z. Wu, 2018. "Fast inference of deep neural networks in FPGAs for particle physics". Journal of Instrumentation 13 (2018), P07027. https://doi.org/10.1088/1748-0221/13/07/P07027 arXiv:1804.06913

[Fu17] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep Learning with INT8 Optimization on Xilinx Devices," docs.xilinx.com, Apr. 24, 2017. https://docs.xilinx.com/v/u/en-US/wp486-deep-learning-int8.

[Homsirikamol14] E. Homsirikamol and K. Gaj, "Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study," 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14), 2014, pp. 1-8, doi: 10.1109/ReConFig.2014.7032504.

[Khoda22] Khoda, Elham E and Rankin, Dylan and de Lima, Rafael Teixeira and Harris, Philip and Hauck, Scott and Hsu, Shih-Chieh and Kagan, Michael and Loncar, Vladimir and Paikara, Chaitanya and Rao, Richa and Summers, Sioni and Vernieri, Caterina and Wang, Aaron, "Ultra-low latency recurrent neural network inference on FPGAs for physics applications with hls4ml", arXiv, 2022, https://arxiv.org/abs/2207.00559.

[Lin21] Kelvin Lin. "Convolutional Layer Implementations in High-Level Synthesis for FPGAs", M.S. Thesis, University of Washington, Dept. of ECE, 2021.

[Pelcat16] M. Pelcat, C. Bourrasset, L. Maggiani and F. Berry, "Design productivity of a high level synthesis compiler versus HDL," 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016, pp. 140-147, doi: 10.1109/SAMOS.2016.7818341.

[Tarafdar21] Tarafdar, Naif and Di Guglielmo, Giuseppe and Harris, Philip C and Krupa, Jeffrey D and Loncar, Vladimir and Rankin, Dylan S and Tran, Nhan and Wu, Zhenbin and Shen, Qianfeng and Chow, Paul, "AIgean: An open framework for deploying machine learning on heterogeneous clusters", *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, Vol. 15, No. 3, Pp 1-32, 2021.

[Xilinx18] Xilinx, Inc. "7 Series DSP48E1 Slice. User Guide", 2018.. https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1

[Xu10] J. Xu, N. Subramanian, A. Alessio and S. Hauck, "Impulse C vs. VHDL for

Accelerating Tomographic Reconstruction," 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, 2010, pp. 171-174, doi: 10.1109/FCCM.2010.33.