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

Caroline Johnson, Scott Hauck, Shih-Chieh Hsu[1], Waiz Khan, Matthew Bavier, Oleh Kondratyuk, Trinh Nguyen, Stephany Ayala-Cerna, Aidan Short, Jan Silva, Anatoliy Martynyuk, and Geoff Jones
Department of Electrical and Computer Engineering [1]Department of Physics
University of Washington
{cjj9, hauck, schsu, wkhan, mbavier , olehkond , nguyetri , sayala , jansilva, shortap}@uw.edu, anatoliym2@gmail.com, geoffhjones@msn.com

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 HLS to machine learning applications, seeking to quantify the resource and performance costs of this technique within Vivado HLS and 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.

1 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 does not allow for hand-tuning the implementation, thus eliminating opportunities for area, power, and performance improvements. Therefore, 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 that HLS provides a significantly faster development time at the cost of the quality of the overall design [Xu10]. 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 [Andre18, Cornu11, Homsirikamol14, Xu10]. However, this resulted in a quality loss of roughly 2x due to factors like longer clock periods, significantly higher resource utilization, etc. [Andre18, Pelcat16, Xu10]. In this paper we seek to quantify the quality losses in modern HLS tools within the machine learning (ML) domain.

We focus on hardware implementations of ML algorithms within the HLS4ML framework [Duarte18]. HLS4ML is a widely used HLS based ML flow, that automatically translates from MLspecific design languages such as Keras, PyTorch, and TensorFlow to FPGA-based implementations via the vendors' HLS flows, in this case Vivado HLS. This allows for data scientists and other ML users, with little or no FPGA-specific expertise, to produce high-performance hardware implementations. HLS4ML has been used for a wide range of applications, including high-energy physics [CMS21, CMS22, Guglielmo20, Hartmann20, Khoda22], astronomy [Jwa22], quantum computing [Xu22], image processing and recognition [Aarrestad21, Ghielmetti22, Guglielmo20, Tarfdar21], superconducting magnets [Hoang21, John21], food contamination [Ricci21], and ultra-low power inference [Borras22].

To measure the tradeoffs 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 model 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.

We have also verified that the HLS4ML designs (HLS) and our SystemVerilog (SV) implementations are bit-accurate. Note that 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). However, in this paper we are comparing HLS and SV flows; thus the implementations perform exactly the same computation using exactly the same numeric format and produce exactly the same accuracy. In fact, we will use bitwidth as an independent variable, comparing the two versions at various bitwidths to help demonstrate some tradeoffs in the implementation flows.

We compare HLS and SV designs in resource usage and performance. For performance, we consider both clock period (and thus throughput, since the initiation intervals are 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. We will present all resources as the overall percentage of the chip's resources used. Finally, 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 generally 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 numbers on the relative development time of the HLS and SV versions, though HLS4ML development is likely even faster than the order of magnitude improvement in HLS development time found in [Xu10]. 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 as unfair since we had to develop the entire SV design from scratch, yet the library elements embedded in HLS4ML itself are not counted.

2 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.

3 Our Two Benchmarks

In order to understand the effectiveness of HLS4ML, and ultimately the Vivado HLS tool, we used two simple neural network models as our benchmarks – a one layer dense neural network and a two-dimensional convolutional neural network.

The one layer neural network is a simple dense neural network. It sequentially processes inputs through four layers: dense, ReLU, dense, and Sigmoid (Figure 1). Each dense layer performs matrix multiplication on an input vector and kernel matrix. The output 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.

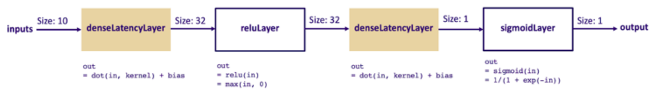

inputs
Size: 10
denseLatencyLayer
out = dot(in, kernel) + bias
Size: 32
reluLayer
out = relu(in) = max(in, 0)
Size: 32
denseLatencyLayer
out = dot(in, kernel) + bias
Size: 1
sigmoidLayer
out = sigmoid(in) = 1/(1 + exp(-in))
Size: 1
output

Figure 1. The One Layer design.

The second benchmark is a 2D convolutional neural network (CNN). This algorithm is made up of 4 layers: convolution, ReLU, dense, and Softmax (Figure 2). The convolution layer utilizes two 3x3 filters being convolved over an 8x8 matrix, thus resulting in $8x8x\,2 = 128$ 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. Here, the dense latency layer 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, $output[i] = \frac{e^i}{\sum e^i}$.

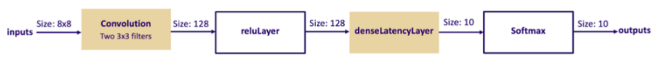


Figure 2. The CNN Design.

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 such that 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.

3.1 Bitwidth and Fixed Point Numbers

All numbers in the benchmarks utilize fixed point 2's complement numbers. We will refer to all of our bitwidths 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

3.2 Verilog Library

Our SV implementation is highly pipelined to support high throughput and low latency computations. The primary layers in our benchmarks 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 applications that use fixed weights, which are pretrained and compiled into the implementation itself. While designs may 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, we created a Python converter that takes Keras weight files in plain text and converts them to SystemVerilog constants. We pass these constants between hardware modules via SystemVerilog parameters, which allows the compilation tools to perform constant folding and related optimizations.

3.3 Multiplication Packing

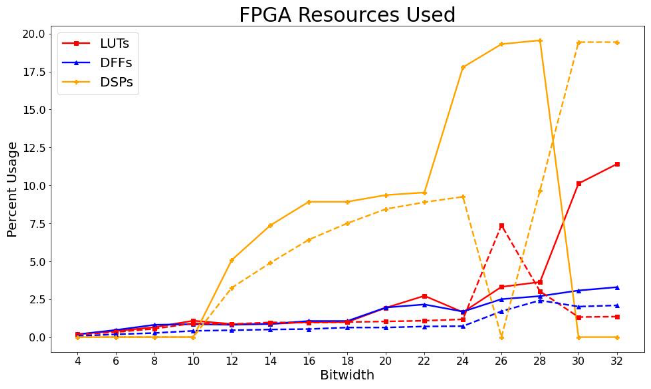
The performance and resource usage of neural networks is highly reliant upon how well the matrix multiplication is done. Since the DSPs built into Virtex 7 FPGAs support 25x18 multiplication, it is straightforward to handle a single multiplication for up to a bitwidth of 18 within a single DSP; for smaller bitwidths, they will use that DSP inefficiently. Therefore we initially explored DSP packing [Fu17], which is a mechanism to more efficiently support low bitwidth neural network computations. For small bit widths (<= 8), it is possible to combine weights via the DSP preadder, with one of the weights shifted up to higher, normally unused bit positions. After applying this DSP packing to our one layer model, we found that this technique did not help. The performance did not change significantly; for resource usage, since we can only pack relatively small bitwidths together, it did not help since these multiplications can instead be easily implemented in LUTs. Thus, forcing low bitwidth multiplications into DSPs is not a good tradeoff. If there were to be a design that had spare DSPs then this might be useful, but not in the case of these neural networks that stress the DSP usage.

4 Results

We will now go through our results and the different hand-optimization techniques we utilized on both the one layer and CNN model, as well as what we learned from the results.

4.1 Initial Results - One Layer model

In Figure 3 we compare SV and HLS implementations of the one layer model.



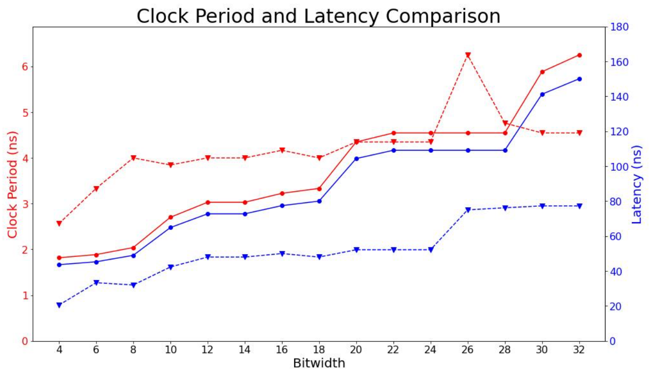


Figure 3. Initial results for SV (solid) and HLS (dashed) versions of the one-layer model.

While the graphs appear twitchy, with strange irregularities that expose interesting features of the toolchains which we will discuss later in the paper, the story is fairly consistent and contradicts our initial assumptions. The HLS designs outperform the SV designs on almost all metrics. For bitwidths <= 18 the SV version has a higher clock rate, by an average of 1.46x, while the two versions have roughly the same clock rate for >=20 bits. The SV version has a 1.7x worse latency overall, uses roughly 1.76x more DFFs and 1.08x more DSPs, and uses 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.

Although we carefully optimized the SV implementation, using standard hardware optimization techniques, heavy pipelining, constant folding, and neural network-specific DSP optimizations, the HLS tools are able to produce better solutions on almost all metrics. These differences are explainable (including many of the twitchy values), and there are surprising optimizations in the Vivado HLS flow that allow it to shine in comparison to best-practice Verilog hardware design.

4.2 The Missing DSPs

In the graph of DSP utilization in Figure 3, 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 at those widths should fit easily into a single DSP block.

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 simple 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 Vivado HLS and the Xilinx Vivado synthesis tools perform shift-add transformations, and the HLS tool has a more powerful optimizer.

To test this, we created simple Vivado-HLS and SystemVerilog designs that performed a single constant multiplication, and compiled them through the two toolflows. Through this we found that the HLS tool appears to convert to shift-add any constant multiplication where the constant is the sum of two powers of two, 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 any single power of two "+-(input<<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.

4.3 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 $((input << c1) + input * remainder)$, where c1 is the integer which most reduces the magnitude of the remainder 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 remainder term is non-zero, the tool decides that the multiplication should not be done via shift-add, and instead should use a DSP. For example, DEPTH = 3 can support "+-(input<<c1)+-(input<<c2)+-(input<<c3)". DEPTH = 1 will use shift-add only for powers of 2, and DEPTH = 0 turns off the optimization completely.

A comparison of the fraction of chip resource usage for various DEPTH settings is given in Figure 4. There is a steady decrease in the use of DSPs, and a steady increase in the LUT usage, as the DEPTH increases. DEPTH = 2 is what HLS is using, but it is clear that a higher DEPTH gives better results.

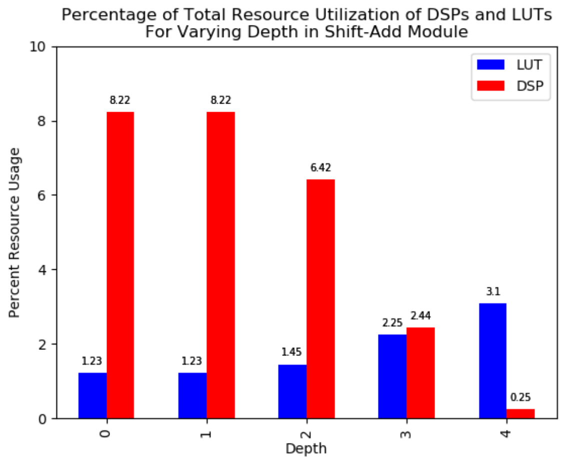


Figure 4. Fraction of FPGA resources used vs. DEPTH parameter for constant multiply. Computed for an overall bitwidth of 16, on the one-layer benchmark.

4.4 Improved One Layer Results 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=2, and tweaking of the pipelining of the design to better use DFFs. After these changes, we began to see the SV design's resource utilization converge with the HLS design (Figure 5).

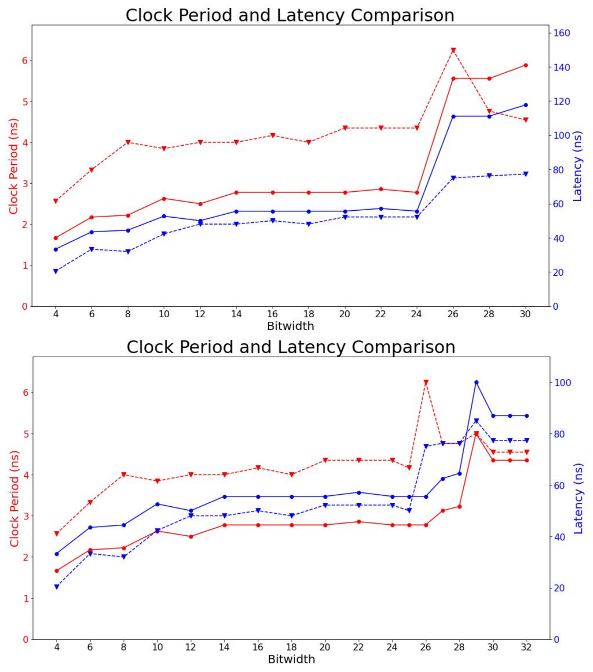


Figure 5. One-layer model results after applying optimizations.

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 DFFs, and 1.03x LUTs; the clock speed is 1.54x faster, while the latency is 1.17x worse.

4.5 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 [Khan23]. 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".

When we include this magical multiplication subroutine into our SV implementation, we get identical results to HLS as can be seen in [Khan23]. This is solely utilizing DEPTH=2. 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 6 we adjust the DEPTH, per bitwidth, based on maximum resource usage, and report the results.

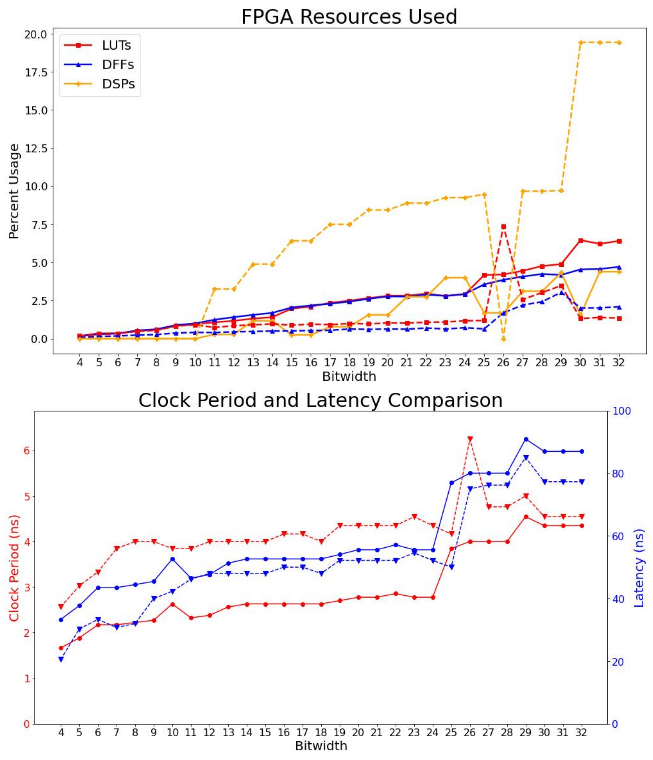


Figure 6. Comparison of SV (solid) and HLS (dashed) with perbitwidth DEPTH tuning. We use DEPTH=3 for bitwidths 11-14, DEPTH=4 for bitwidths 15-24, DEPTH=5 for 25-29, and DEPTH=6 for 30-32.

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.

5 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. 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.

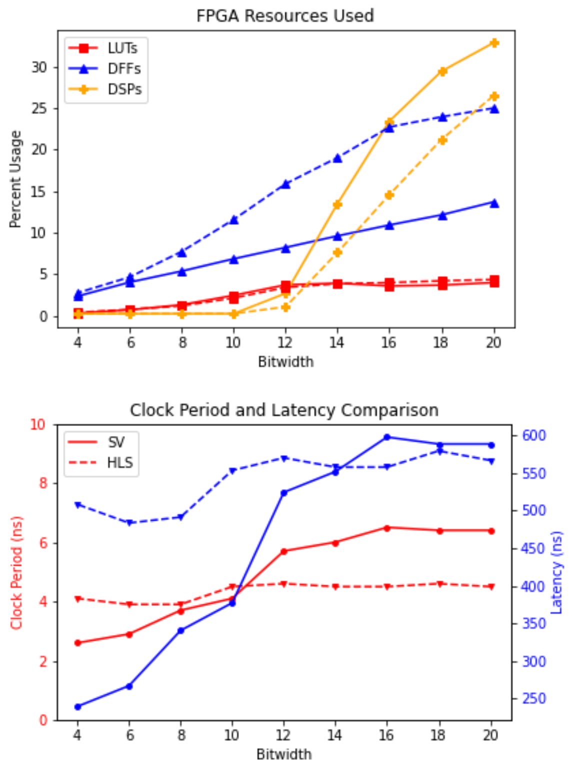


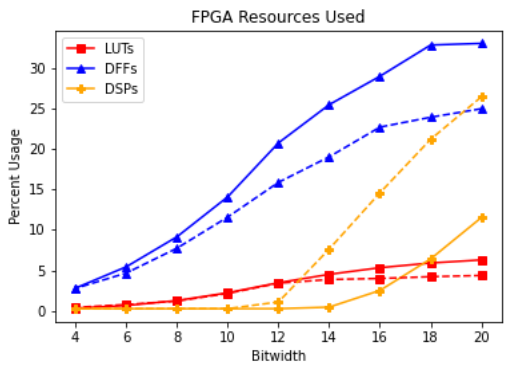
Figure 7. Initial SV (solid) vs HLS (dashed) results for the CNN benchmark.

Figure 7 has 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 20, that constraint is not met. Thus, we will consider only bitwidths of 20 or lower.

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. On average the SV design has 0.94x the maximum resource usage of the HLS design.

After implementing the modifications we discovered for the first benchmark, we were able to see significant improvement in resource utilization and performance. Figure 8 shows the results of this, including a shift-add module with a DEPTH of 3.

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. Note that both CNN designs make use of SRL 16s to buffer data through the convolution layer; these are accounted for in the LUT usage. Compared to the HLS system, on average we used 0.58x 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. This is due to the shift from DSPs to FFs and LUTs, and that the HLS design uses SRLs to buffer data between the layers, while the SV versions use DFFs.



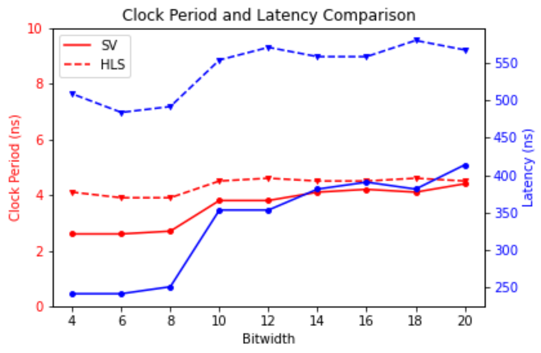


Figure 8. Comparison of final SV (solid) and HLS (dashed) results for the CNN benchmark.

7 Conclusion

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. HLS4ML's high performance and resource usage leverages the high quality results from the Vivado HLS tool. Hand design may not be dead everywhere, but it is quite suspect in at least this domain, and promotes the general use of the Vivado HLS tool.Will these results hold true with larger more complex models with varying parameters? With initial exploration of more complex designs, we see that HLS4ML has worse performance and resource usage when varying convolution reuse factor and stride length, in terms of resource usage but especially in terms of the initiation interval. It appears that the Vivado HLS tool may not be able to compete with the more advanced layers, but HLS4ML can leverage the strengths of Vivado HLS as seen in these simpler layers to make it so all of their layers are as optimal as can be [Johnson23].

References

[Aarrestad21] Aarrestad, Thea, et al. 2021. Fast convolutional neural networks on FPGAs with hls4ml. Machine Learning: Science and Technology 2.4 (2021): 045015. arXiv:2101.05108

[Altoyan20] W. Altoyan and J. J. Alonso. 2020. 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] Tétrault Marc-André. 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. 2013. 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.

[Borras22] Borras, Hendrik, et al. 2022. Open-source FPGA-ML codesign for the MLPerf Tiny Benchmark. arXiv preprint arXiv:2206.11791 (2022).

[CMS21] CMS Collaboration. 2021. The CMS Level-1 Endcap Muon Trigger at the High-Luminosity LHC, April 2021, https://doi.org/10.22323/1.390.0890

[CMS22] CMS Collaboration. 2022. 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. 2011. 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. 2017. 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.

[Ghielmetti22] Ghielmetti, Nicol , et al. 2022. Real-time semantic segmentation on FPGAs for autonomous vehicles with hls4ml. arXiv preprint arXiv:2205.07690 (2022).

[Guglielmo20]Di Guglielmo, Giuseppe, et al. 2020. Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml. arXiv preprint arXiv:2003.06308 (2020).

[Hartmann20] F. Hartmann and A. Ball. 2021. The Phase-2 Upgrade of the CMS Level-1 Trigger, CERN Document Server. https://cds.cern.ch/record/2714892? ln=en

[Hoang21] D. Hoang et al., Intelliquench. 2021. An Adaptive Machine Learning System for Detection of Superconducting Magnet Quenches, in IEEE Transactions on Applied Superconductivity, vol. 31, no. 5, pp. 1-5, Aug. 2021, Art no. 4900805, doi: 10.1109/TASC.2021.3058229.

[Homsirikamol14] E. Homsirikamol and K. Gaj. 2014. 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.

[John21] John, Jason St, et al. 2021. Real-time artificial intelligence for accelerator control: a study at the fermilab booster. Physical Review Accelerators and Beams 24.10 (2021): 104601. https://journals.aps.org/prab/abstract/10.1103/PhysRevAccelBeams.24.104601

[Johnson23] Johnson, Caroline, et al. 2023. Evaluating the Quality of HLS4ML’s Basic Neural Network Implementations on FPGAs.

[Jwa22] Jwa, Yeon-jae, et al. 2022. Real-time Inference with 2D Convolutional Neural Networks on Field Programmable Gate Arrays for High-rate Particle Imaging Detectors. Frontiers in Artificial Intelligence 5 (2022). doi: 10.48550/arXiv.2201.05638

[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. 2022. 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. 2021. Convolutional Layer Implementations in High-Level Synthesis for FPGAs, M.S. Thesis, University of Washington, Dept. of ECE, 2021.

[Khan23] Waiz Khan, Caroline Johnson, Scott Hauck, Shih-Chieh Hsu, Geoff Jones, "Benchmarking High Level Synthesis for Machine Learning Implementations versus Hand-optimized SystemVerilog", *A3D3 High-Throughput AI Methods and Infrastructure Workshop*, 2023.

[Pappalardo22] Pappalardo, Alessandro, et al. QONNX. 2022. Representing Arbitrary-Precision Quantized Neural Networks. arXiv preprint arXiv:2206.07527 (2022).

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

[Ricci21] M. Ricci et al. 2021. Machine-Learning-Based Microwave Sensing: A Case Study for the Food Industry, in IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 11, no. 3, pp. 503-514, Sept. 2021, doi: 10.1109/ JETCAS.2021.3097699.

[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. 2021. 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. 2018. 7 Series DSP48E1 Slice. User Guide. https:// docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1

[Xu10] J. Xu, N. Subramanian, A. Alessio and S. Hauck. 2010. 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.

[Xu22] Xu, David, et al. 2022. Neural network accelerator for quantum control.
arXiv preprint arXiv:2208.02645 (2022)