Evaluating the Quality of HLS4ML's Basic Neural Network Implementations on FPGAs

Caroline Johnson

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2023

Committee:

Scott Hauck
Shih-Chieh Hsu

Program Authorized to Offer Degree:

Department of Electrical and Computer Engineering

University of Washington

Abstract

Evaluating the Quality of HLS4ML's Basic Neural Network Implementations on FPGAs

Caroline Johnson

Chair of the Supervisory Committee:
Scott Hauck
Department of Electrical and Computer Engineering

Field-programmable gate arrays (FPGAs) offer an attractive platform for machine learning models; however, creating these models is a time-consuming process that demands specialized knowledge. High-level synthesis (HLS) offers the potential to streamline this hardware development, but with the trade-off of not being able to hand tune the design to take advantage of the specific resources and performance capabilities available. The popular open-source Python library HLS4ML offers low latency HLS machine learning models. However, it is unclear how much quality reduction is caused by adopting an HLS design flow. In this thesis we create carefully optimized SystemVerilog versions of identical HLS4ML designs, allowing us to evaluate the quality of the designs produced by HLS4ML. Our research reveals that HLS designs are highly competitive with hand-optimization techniques for basic layers with standard parameters, especially in terms of the capabilities of the Vivado HLS tools. However, when advanced parameters like reuse factor and stride are altered, our hand-written design is able to achieve a 2x-3x increase in performance over HLS4ML's. Through this evaluation we were able to identify weaknesses in the existing tools and develop workarounds to help provide significant quality improvements.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to start out by thanking my two advisors, Scott Hauck and Shih-Chieh Hsu. I'm deeply grateful for the support Scott has offered me over the past three years. His guidance, along with the help from Shih-Chieh, have been invaluable.

I am also very thankful to the members of ACME labs and the HLS4ML community. Special appreciation goes out to Waiz Khan, Matthew Bavier, Oleh Kondratyuk, Trinh Nguyen, Stephany Ayala-Cerna, Aidan Short, Jan Silva, Anatoliy Martynyuk, and Geoff Jones for their collaboration on these benchmarks as well as the ongoing work on new benchmarks. Further, thank you to Yilin Shen for your help with my many computer issues. Additionally, the HLS4ML community helped our team overcome many barriers in the design process. Vladimir Loncar was a big help in exploring new solutions for our third benchmark discussed here. Finally, thank you to all in the A3D3 community for the support and continued motivation for the work. This research was funded in part by National Science Foundation (NSF) grants No. 1934360 and 2117997.

I would like to thank my parents, my brothers, and my dog, Dublin, for their continued support throughout my education. Having such a solid support system made all the difference over the past years. Finally, thank you to the rest of my friends and family for helping me through classes at UW and supporting me through my academic journey.

Section 1. INTRODUCTION

In the midst of the machine learning revolution, machine learning models are rapidly expanding their parameter sizes, demanding faster processing speeds and specialized hardware beyond the capabilities of conventional CPUs. Field Programmable Gate Arrays (FPGAs) offer this increase in performance and compatibility. FPGA's parallel processing techniques allow for optimized matrix multiplications and advanced computations required of the machine learning models while preserving the flexible hardware required by these constantly changing algorithms. However, programming FPGAs requires expertise beyond that of a typical software engineer. To facilitate broader FPGA usage, High-Level Synthesis (HLS) tools are employed. HLS tools take higher-level programming that the traditional software engineer would be very familiar with and can automatically produce the lower-level code that is required for programming FPGAs. For specific machine learning applications, the open-source Python library High-Level Synthesis for Machine Learning (HLS4ML) [1] allows for models written in Python to be converted to the hardware description language (HDL) that can be used to program an FPGA.

Section 2. FPGA BACKGROUND

FPGAs are reprogrammable integrated circuits that can be configured for a user's specific purpose. FPGAs are very attractive for machine learning models due to their parallel processing abilities and low latency. This parallel processing is critical for the matrix multiplication required in neural networks. Additionally, their low latency allows for the quick decision-making that is required of many machine learning applications, such as robots and self-driving cars. However, in order to take advantage of the benefits of using FPGAs, the FPGAs must be properly programmed.

This requires knowledge of the different resources that are available as well as an understanding of the different performance metrics. We will now go over the general resources of FPGAs as well as some performance metrics.

2.1 Resources

When uploading a design onto an FPGA, that design utilizes resources within the FPGA itself. The board targeted in this thesis is the Xilinx Virtex 709, device XC7VX690T. The resources of interest here are:

- **Digital Signal Processing Blocks (DSPs)** - DSP blocks are utilized for complex math operations. In our case, DSPs are used to implement multiplication. Each DSP in the Virtex 709 contains a pre-adder, a 25x18 bit multiplier, an adder, and an accumulator.
- **Look-Up Tables (LUTs)** – LUTs are utilized to represent a Boolean function. The LUTs on the Virtex 709 are 6-input LUTs. This means it takes 6 1-bit inputs, that can represent up to 2^6 combinations and produces 1 1-bit output. Each input combination has a corresponding output value stored in the LUT, which allows LUTs to store complex Boolean functions.
- **Flip-Flops (FFs)** - FFs are utilized to achieve sequential logic. They represent how much data we are holding onto each clock cycle.
- **Block Random-Access Memories (BRAMs)** - BRAMs are dedicated memory blocks that can be assigned to hold large pieces of data. The Virtex 709 offers 36 KB BRAMs. A 36 KB BRAM means the memory unit can hold up to 36 KB of data.
- **Shift-Register LUTs (SRLs)** - SRLs are specific LUTs that are utilized as shift-registers. For our purposes, we were able to utilize SRLs as FIFOs, rather than needing to use FFs. In resource usage calculations, SRLs are added to the total number of LUTs used.

Table 1.1. displays the total number of each resource available on the Virtex 709 [2].

Table 1.1. Virtex 709 Available Resources [2]

LUTs	DSPs	36 kB BRAM	FFs
693,120	3,600	1,470	866,400

This resource overview of the Virtex 709 shows that there is not an equal amount of all resources available. Therefore, to account for the different costs of using the different resources, all resource usage will be presented as a percent of that resource used on the Virtex 709.

2.2 PERFORMANCE

Not only are the resources that a design takes up important, but another important aspect is how quickly the design is able to run. The following two performance metrics will be discussed for each model:

- **Latency -** How long does it take to get the first output?
- **Initiation Interval (II) -** How long, from one input to the next input, does it take to reach the same part of the model again? For example, from input Y to input Y+1, how long does it take to reach the first output for that given input?

Latency is important because it is essentially response time – how long from when data is received, until an answer is obtained. II is inversely proportional to throughput, the amount of data that can be processed within a given amount of time. Both will be discussed in terms of the number of clock cycles and the time taken (in nanoseconds) based on the fastest clock period at which the design can run.

Section 3. NEURAL NETWORK BACKGROUND

Neural networks were created to model the function of the neurons in a brain. In a simplified overview, if an individual neuron receives enough information from other neurons, then that neuron will fire to share that information. Neurons learn to anticipate when they will fire to decrease the delay from receiving the information to the firing. Neural networks quickly became of interest in machine learning due to a neuron's ability "to perform distributed computation, to tolerate noisy inputs, and to learn..." [3]. Although there are a variety of other machine learning systems utilized to perform similar computation, neural networks are one of the most popular and effective types.

In a mathematical version of the neuron and its connections, we see multiple inputs coming into a neuron, and one (or many) output(s) coming out. The neuron itself will fire if the linear combination of its inputs exceeds some threshold. The linear combination scales each input by a given weight, representing how much an input affects the output. The threshold is determined by an activation function, which can be looked at as determining the frequency of the neuron firing. An example of a simple neural network is shown below.

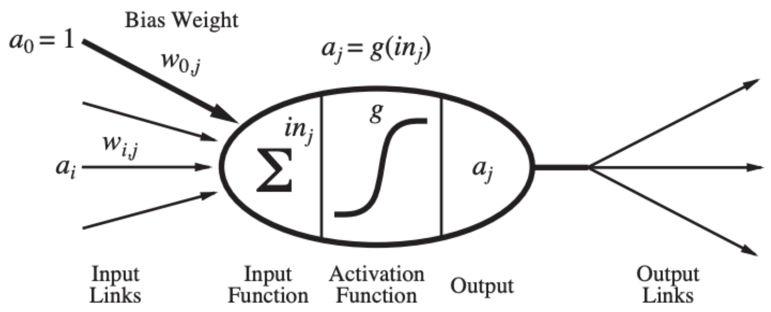


Figure 3.1. Example Neural Network Computation [3].

This graphic can be explained by the following equation:

$$x_j = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right) \tag{3.1}$$

Where a_i is the input, $w_{i,j}$ is the current weight being utilized, $g()$ is the activation function, and x_j is the output. In the layers we will be looking at, besides weights, each output will have an individual bias added. This changes the equation to be:

$$x_j = g\left(\left(\sum_{i=0}^{n} w_{i,j} a_i\right) + b_j\right) \tag{3.2}$$

where b_j is the current bias. Like with the weights, a bias is utilized to adjust the output based on the linear combination and the expected output [3].

In machine learning, a neural network is commonly utilized when one knows example inputs and outputs but wants to be able to predict the output for any set of inputs. To accomplish this functionality, the model needs to be trained. Training consists of obtaining weights and biases that allow the model to give as close to the expected output as possible. The model is run for many iterations, each time comparing the output of the model to the expected, or true, output. Depending on how close the output value is to the expected value, individual weights and biases are tuned so that on the next iteration, the output will hopefully be even closer. This approach is called back-propagation. Once the model is trained, the optimized weights and biases can then be used on new inputs, which is known as inference.

For complex neural networks, additional layers are added to increase the accuracy of the model. A key attribute of neural networks is that they are generally acyclic graphs, or graphs where the output of one layer connect to the input of the next layer such that no cycles, or infinite loops, are formed. If all the outputs of a layer connect to all the inputs of the next layer, that layer is said to be fully connected. Each graph will contain an input and output layer, and may contain hidden layers, ones that do not connect directly to either the input or the output.

We will now go over some common neural network layers and activation functions.

3.1 Common Neural Network Layers

3.1.1 *Convolution Layer*

A convolution layer is the core layer of a Convolutional Neural Network (CNN). CNNs are a specific implementation of a neural network commonly used for computer vision applications. A convolution layer will take a large input matrix, say of size 8x8, that represents an image. The first step of this layer will be to add any padding around the matrix if needed. The padding can be added to only some of the sides of the matrix or to all the sides. Padding is added so that information on the borders of the input matrix is not lost in the computation. Then, the filters of the layer move across the width and the height of the matrix computing the dot product of the filter with the matrix at each location. A filter is used to detect a feature of the image and is smaller than the input matrix, say of size 3x3. This computation produces a 2D map that represents each filter at every location of the image. It is common to use multiple filters to detect multiple features. An example of a common filter used, filtering for vertical edge detection, is shown below.

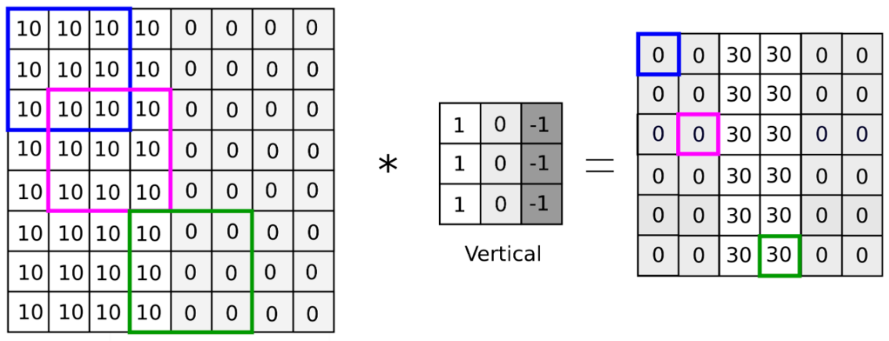


Figure 3.2. Example CNN Edge Detection Computation [4].

The 8x8 matrix on the left represents the input image and the 3x3 matrix next to it is the filter. The 6x6 matrix on the right is the output matrix and was computed by having the input image be convolved with the filter. The vertical edge detection filter is highlighted in three different locations along the input image. Compared to the pink or blue square, the green square on the input image highlights a location where there is a distinct change in input matrix vertically. Therefore, the value of the output for the green square is higher than the pink or blue squares. This shows how a filter can be utilized to detect a feature in an input image.

One of the parameters we can adjust for in a convolutional layer is the stride of the filter. The example above utilized a stride of 1. This means the filter shifts by one horizontally and vertically as it moves across the matrix. By increasing the stride, the spatial dimensions of the output matrix are reduced which leads to a smaller output matrix. An example output of a grayscale 4x4 input image that is convolved with an averaging filter and a stride of 2 is shown below. The averaging filter is of size 2x2 and takes all the pixels at each location and divides their sum by 4. With a stride of 2, there are no overlapping filter locations as the filter slides across the matrix.

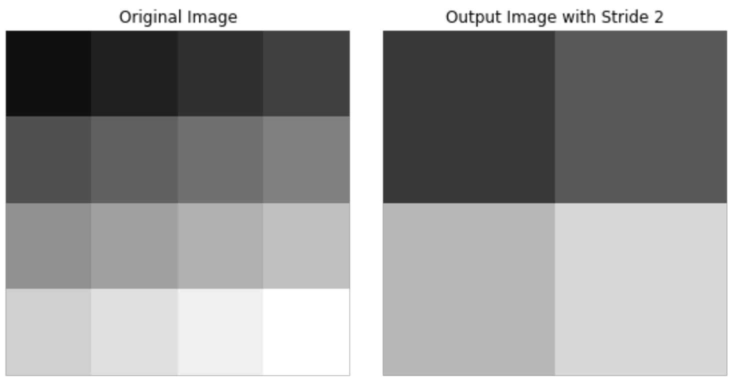


Figure 3.3. Example Convolution Output with a Stride of 2.

On the left we see the input image and on the right we see the output image as a result of the input being convolved with an averaging filter with a stride of 2. The 2x2 output image highlights the down sampling that occurs with higher stride values. For more detailed information on CNNs, please refer to [5].

3.1.2 *Dense Latency Layer*

A dense latency layer is a fully connected layer where each output of the previous layer is connected to each input of the current layer. Each output of the current layer is a dot product of the previous layer's outputs and a set of weights, with biases added on. This can be seen here:

$$output_j = \left(\sum_{i=0}^{n} \left(x_i \cdot w_{ij} \right) \right) + b_j \tag{3.3}$$

where x is the input, w is the weights matrix, and b is the bias vector. In matrix notation, this can be seen as:

$$OUTPUT = XW + B \tag{3.4}$$

Dense latency layers can be used to flatten data from multiple filters into one set of data and provide classifications that the later activation function can use to classify the output [6].

3.2 ACTIVATION FUNCTIONS

If a model were to just be made up of the layers explained above, the output would not be meaningful. The output would be some sort of linear combination of the inputs and various weights and biases. To make the output meaningful, activation functions are used. Activation functions allow a neural network to learn complex patterns as well as classify the data [3].

3.2.1 *RELU*

The Rectified Linear Unit (ReLU) computes the following equation:

$$f(x) = max(x, 0) \tag{3.5}$$

This is the simplest activation function in terms of mathematical complexity [5]. The graph of the ReLU function is shown below.

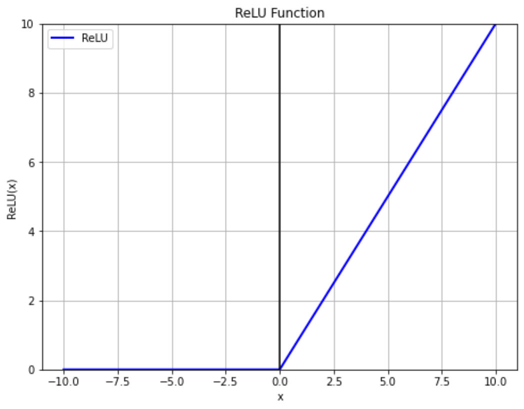


Figure 3.4. ReLU function.

3.2.2 *Sigmoid*

The Sigmoid nonlinearity unit computes the following equation:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.6}$$

This function takes any real number and squashes it in the range of 0 to 1 [5]. The graph of the Sigmoid function is shown below.

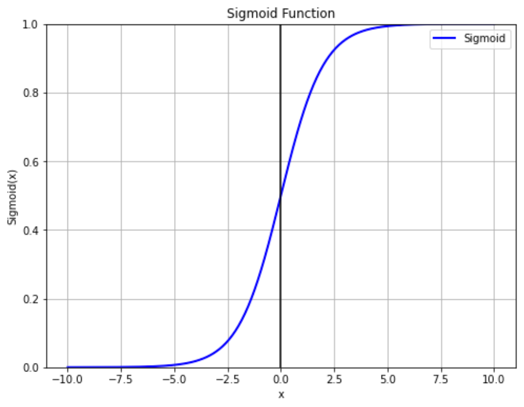


Figure 3.5. Sigmoid function.

3.2.3 *Softmax*

The Softmax nonlinearity unit converts the inputs into class probabilities, such that they sum to 1. It computes the following equation, where n is the number of points in the sample:

$$output[i] = \frac{e^{i}}{\sum_{j=0}^{n} e^{j}} \tag{3.7}$$

This function takes the exponentiation of each input and divides it by the sum of the exponentiation of all the points [5]. For example, if the input is a vector with values from -2 to 7, the values will be scaled such that the larger values, i.e. 6 and 7, will have much larger outputs. An example Softmax output for these 10 points is shown below.

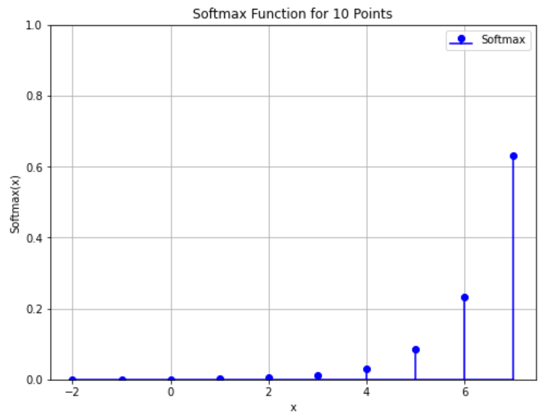


Figure 3.6. Softmax function.

Section 4. HLS & HLS4ML BACKGROUND

High-Level Synthesis (HLS) takes C++ programs and converts them into the HDL that will be used to generate the programming bitfile for the FPGA. HLS allows a programmer to not need to dig into the lower-level details of HDLs but rather write code at a higher level, in this case C++. Vivado HLS is a version of this used by Xilinx FPGAs. HLS4ML [1] takes this abstraction one step further and allows for the code to be written in an even higher-level language, Python.

HLS4ML is an open-source library that takes a Python machine learning model, whether it be in TensorFlow or Keras, and converts that model down into the C++ code used by Vivado HLS. This allows a user to simply define the layers they wish their model to contain, as well as their weights and biases, and then HLS4ML will do the rest. The user will be left with functioning code to upload to an FPGA. This process is shown below.

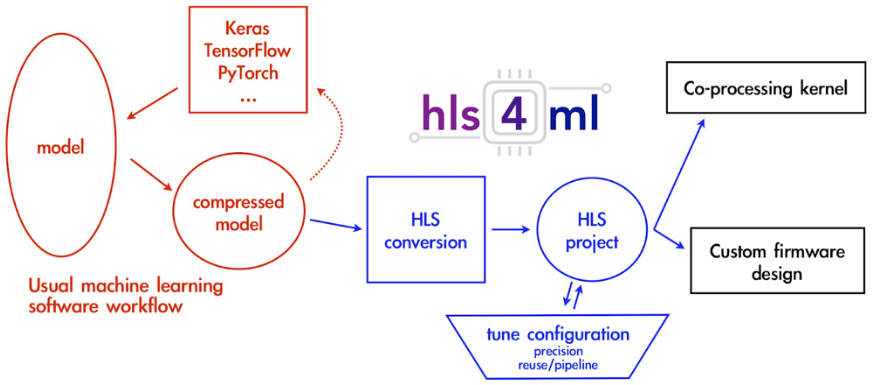


Figure 4.1. HLS4ML Workflow [1].

We will now dive deeper into this workflow.

4.1 HLS4ML Workflow & Vivado HLS

In order to utilize HL4ML, one needs to create a machine learning model with Python. One will need to create a .h5 file and a .json file that describe the model, or one can specify these constraints and the layers directly using the commands provided by the specific machine learning library used, for example with Keras. If the model has already been trained, one is then able to import the trained weights and biases. Otherwise, HLS4ML will create default weights and biases for you. In this thesis we are looking at inference models, meaning the models have already been trained and the weights and biases are already optimized.

After creating a machine learning model with Python, one will then need to convert that model into an HLS model. This is done utilizing the commands provided by HLS4ML [7]. When converting to an HLS model, there are many different configuration parameters that can be set in the `config` dictionary of the HLS model. This ranges from advanced tuning parameters like reuse factor, which will be explained in section 4.2, or even simple attributes of the model like the precision of the bitwidth, which will be explained in section 5.1.1.

Once the HLS model is created, one can then run various commands such as `hls_model.predict()`, which predicts the accuracy of the given model. The most important command is the `hls_model.build()` command. This is the command that takes the model from Python to C++ and finally to the HDL that can be used to program the FPGA. Two versions of HDL are produced: Verilog and VHDL. The target directory specified in the `config` dictionary will become populated with these Verilog and VHDL files, as well as all the C++ files utilized by Vivado HLS, some configuration data such as the weights and biases used, and a log file describing the results of the C-synthesis from Vivado HLS [8]. An example of the resource usage and performance summary given in this log file will look like the following:

```
{'EstimatedClockPeriod': '4.299',
 'BestLatency': '568',
 'WorstLatency': '569',
 'IntervalMin': '163',
 'IntervalMax': '568',
 'BRAM_18K': '467',
 'DSP48E': '1048',
 'FF': '21470',
 'LUT': '26768',
 'URAM': '0',
 'AvailableBRAM_18K': '5376',
 'AvailableDSP48E': '12288',
 'AvailableFF': '3456000',
 'AvailableLUT': '1728000',
 'AvailableURAM':'1280'}
```

Based on our experience, these C-Synthesis results are not always trustworthy. Therefore, to fully understand the model, one should take the HDL of choice that was produced and create a

Vivado project with it. Then, after specifying the clock speed, synthesis and implementation can be run to produce a full report. By creating this Vivado project, one can also run simulation on the model, the purpose of this will be discussed in section 4.4.

In addition to the Python interface, HLS4ML provides a command line interface (CLI). The CLI offers more flexibility in dividing up the build command in to separate stages. Please note that the CLI is deprecated [9].

The HDL produced by HLS is created based on the defined IO, the computation, and any pragmas in the C++ files. Pragmas control different performance and resource utilizations of the code such as pipelining, loop unrolling, array partitioning, and the II of different loops [10]. HLS4ML has predefined C++ files for each of the layers that it supports. This allows HLS4ML to take the Python machine learning model and map the layers to the corresponding C++ code. Depending on the configuration of the Python model, HLS4ML will adjust the pragmas automatically.

We will now go over some configuration aspects of the HLS4ML models.

4.2 Reuse Factor

Reuse factor is an important parameter of the HLS4ML configuration that is set for each layer of the machine learning model. Reuse factor determines how many times resources are used for each layer [1]. This can be seen most directly in the usage of multipliers, or DSPs. If the model has a reuse factor of 1, each multiplier is used only once within each inference, whereas a reuse factor of 3 has each multiplier being used 3 times. By changing the number of times a multiplier is used, the number of cycles to compute a computation increases one-to-one with the reuse factor. For a 3x3 filter, a reuse factor of 1 has each filter utilizing 9 multipliers and a new inference can start each cycle, versus a reuse factor of 3 has each filter using 3 multipliers and a new inference starting only every 3 cycles. The following graphic illustrates reuse factor in terms of this 3x3 filter.

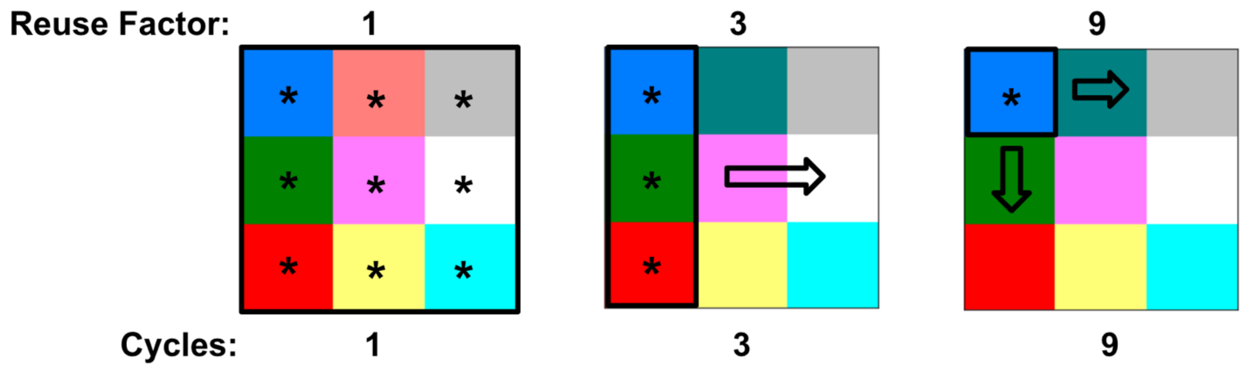


Figure 4.2. Reuse factor example.

4.3 Streaming the Input

With large inputs, it is impossible for an FPGA to read in all the inputs and do the internal processing in parallel since it will run out of resources. This is due to the limitations of available IO as well as internal resources. Therefore, HLS4ML uses the idea of streaming the inputs such that one data point is brought in and out at a time. For an input matrix of 64 16-bit numbers, it would take 64 cycles to read in the input. This is because one 16-bit number is brought in each cycle. To configure a model to utilize this streaming technique, one needs to set the `io_type` to `io_stream`, versus the default `io_parallel` in the HLS4ML configuration. When creating HLS4ML models that contain a CNN, the model will fail to build if the `io_type` is not set to `io_stream`.

4.4 Testing An HLS4ML Design

To validate functionality of the HLS model, it is critical to ensure that the outputs match the expected values for any input. This also means verifying the functionality of each internal layer by checking each layer's output. Although Vivado HLS provides C-simulation and co-simulation, for testing that the HDL output matches the output of the C++ code, it is best to create a Vivado project for the outputted HDL and write a testbench in the HDL. This is because we have observed issues with the produced HDL output not matching expected values while the C-simulation and co-simulation produce the expected values. Therefore, to ensure valid HDL functionality, the HDL should be tested directly.

Section 5. OUR BENCHMARKS.

Using the HLS or HLS4ML workflow previously explained allows a user to turn a complicated machine learning model into HDL easily and quickly. However, this generally means giving up hand-tuning the implementation, and thus eliminating opportunities for power, performance, and area improvements. 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 that early HLS provides a significantly faster development time at the cost of the quality of the overall design [11]. 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 [11, 12, 13, 14]. 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, and significantly higher resource utilization [11, 14]. Therefore, in this thesis we seek to quantify the quality losses due to these tools within the machine learning domain.

To measure the costs of an HLS-based implementation strategy, we have developed custom Verilog-based implementations of three benchmarks: a simple one-layer model, a simple 2D CNN, and a slightly more complex 2D CNN. These are simple machine learning models that can be carefully hand-optimized and represent core building blocks of larger deep learning models. We carefully crafted custom implementations of each of these benchmarks, as well as automated flows to import the specific trained weights and biases files from the machine learning models into our hand-optimized designs. Further, we ensured our designs are bit-accurate to the HLS designs through thorough testing.

In this thesis we compare the HSL4ML (HLS) and our hand-optimized (SV) designs in both resource usage and performance. Resource usage will be presented as the fraction of the available resources used on the Virtex 709. This is to account for the tradeoffs between using different resources. For example, a given multiplication could either be performed in the DSPs, or instead mapped to LUTs. For performance, we consider clock period, latency, and II.

We will now discuss some further background critical to understanding the benchmarks.

5.1 BACKGROUND

5.1.1 *Bitwidth and Fixed-Point Numbers*

In machine learning (like many algorithms) there is a tradeoff between the number of resources used, and the accuracy of the resulting computation. An example of this is the number format used for a computation. For decimal numbers, there are two common ways to represent them in binary: fixed-point and floating-point. Fixed-point means that there is a set location of the decimal and all bits above this point represent the integer bits and all bits below represent the fractional bits. Floating-point means the decimal point location can vary to represent a larger set of values. On FPGAs, floating-point computations introduce additional resource overhead and complexity with very little added benefit to the overall accuracy. We avoided floating-point computations by converting all the operations to fixed-point representations.

Additionally, a designer can choose to use very high bitwidths, 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 the first two benchmarks, we chose to vary bitwidth to see how the resource usage and performance scaled. For the final benchmark, we stuck to a bitwidth of 16 as we had a different varying parameter, reuse factor.

All fixed-point numbers in the benchmarks are 2's complement numbers. We will refer to all 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
```

5.1.2 *Verilog Library*

Our SV implementations are highly pipelined to support high throughput and low latency computations. This is the same goal as advertised by HLS4ML. The primary layers in the SV implementations contain matrix multiplications. Depending on the bitwidth, these expensive computations can heavily stress the number of DSPs available. Besides the matrix multiplication, the other layers implemented are the activation layers discussed previously. ReLU is a simple comparison operation, converting any negative value to zero, which is done in basic logic mapped to LUTs. 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 depending on the place and route results determined by the tools for that specific implementation.

Since we are utilizing fixed weights and biases, which are pretrained and compiled into the implementation itself, we created a Python converter that takes Keras weight and bias 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. Additionally, all our code is parameterized for bitwidth, in terms of fractional and integer bits.

5.1.3 *Multiplication Packing*

Because neural network algorithms are so multiplication-intensive, efficient use of the DSPs is important to achieving an efficient implementation. Since the DSPs built into Virtex 709 support 25x18 bit 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 [15] is a mechanism to make low bitwidth neural network computations more efficient within an FPGA's DSP resources. For multiplication-heavy dense latency and convolutional layers, inputs to the layers are multiplied by several different constant weights. Since the DSPs support 25x18 bit 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. The DSP block utilized by the Virtex 709 is shown below.

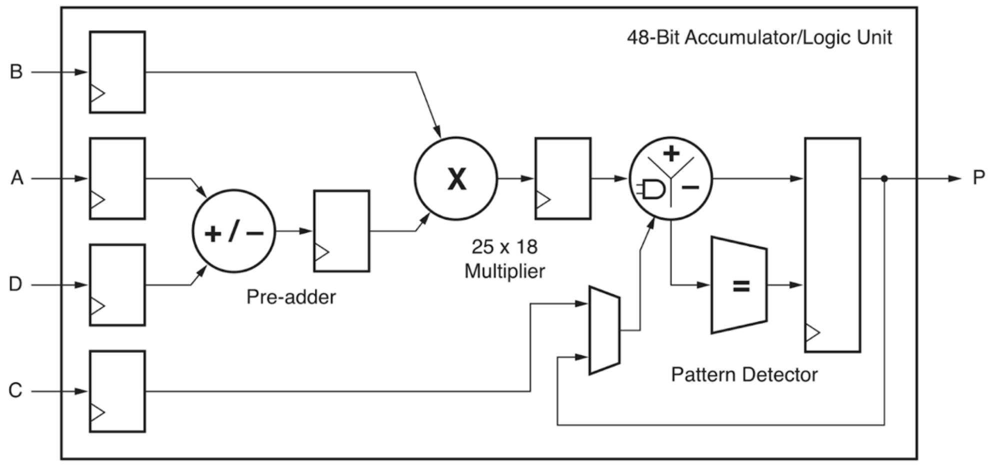


Figure 5.1. Virtex 709 DSP Block [2].

The DSP slice shown in Figure 5.1 has the inputs to the DSP slice on the left, and the output on the right. By utilizing the pre-adder, we can combine two weights via ports A and D. The weight on port D must be left-shifted so that it does not interfere with the input from port A on the output port P. For an 8-bit example, multiplying an 8-bit weight by an 8-bit input will result in 16-bits. Therefore, to make it so the resulting outputs of the combined weights via the pre-adder do not overlap, we must left-shift the weight on port D by at least 16 bits. The sum of the weights on port A and port D can then be multiplied by the input on port B. The resulting output on port P can then be separated so that we have the two individual multiplications of `weight1 * input` and `weight2 * input`.

If the input differs in sign from the weight that has been left-shifted, a conditional correction term must be applied. An example of this is shown below.

```
              1111011111111111111001010
                              *00101110
...111111111101000111111111011001001100
            (-737)          (-2484)
CORRECT:    (-736)          (-2484)
```

Figure 5.2. DSP packing correction example.

The two individual weights are bolded and the resulting output for the two separate multiplications are shown below for both the computed as well as the correct value. For the weight that was left-shifted, we see that the computed output is -737 but the correct output is -736. This is because the negative result in the lower bit term, bits 15:0, steals a “1” from the upper bit term, bits 31:16. To fix this, a decimal 1 shifted left 16 bits (the conditional correction term) is applied. This correction term is applied conditionally via the C port of the DSP slice. The full condition is:

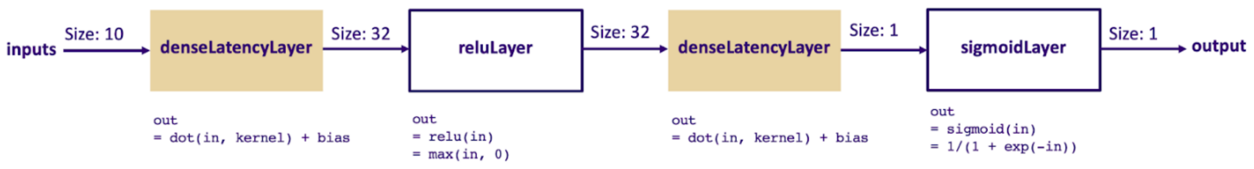
$$C = \left(weight_{PortD,sign} \oplus input_{PortB,\ sign}\right) \cdot \left(weight_{PortD} \neq 0\right) \cdot \left(input_{PortB} \neq 0\right) \tag{5.1}$$

The correction term checks for a difference in sign between the weight on port D and the input, as well as whether either of those values are zero. This caveat is required since multiplication by zero does not exhibit the same behavior since a negative value multiplied by zero remains zero.

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

5.2 Benchmark 1: One-Layer Model

Our first benchmark, the one-layer model, is a very simple neural network that sequentially processes inputs through four layers: dense latency, ReLU, dense latency, and Sigmoid, as shown below.



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

Figure 5.3. One-layer model structure.

The model takes in 10 inputs every clock cycle and outputs 1 value per these 10 inputs. The one-layer implementation is fully unrolled, with an II of 1. This allows for a new computation to start every clock cycle.

The one-layer model was trained on the UCI Human Activity Recognition dataset which contains sensor data of humans performing various activities [16]. We will now go over the comparison of our SV implementation to the HLS design.

5.2.1 *Applying Multiplier Packing*

Initially, when creating our custom implementation, we explored how multiplier packing can help reduce resource usage of the one-layer benchmark. As explained in section 5.1.3, for numbers 8 bits and under, we can combine multiple multiplications into one DSP. With this DSP reduction, there will be a hardware overhead cost to perform the conditional corrections described. The results for applying multiplication packing for bitwidths 4-8 for the one-layer model are shown below.

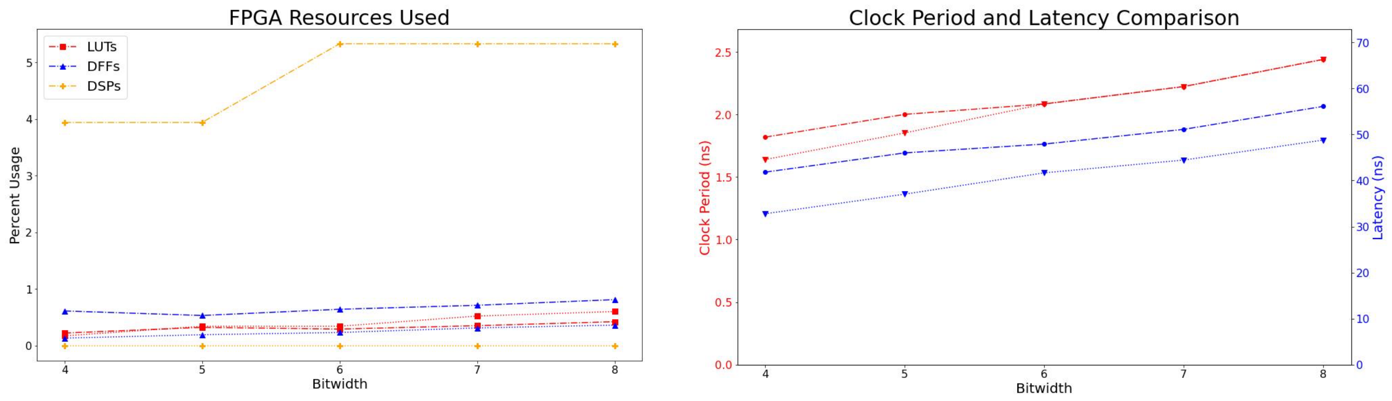


Figure 5.4. Resource and performance comparison for DSP packing.

The solid lines are the SV design without packing, while the dashed lines are the SV design with packing. On the left we see the resource usage results and, on the right, we see the performance results. Both graphs have bitwidth as the x-axis, ranging from 4 to 8. The resource usage is broken down into LUTs, FFs, and DSPs. As stated previously, we have normalized the resource usage to the total amount of resources available on the Virtex 709 to account for the different cost of using different resources. Each resource curve is important as we care about what the overall design is using; however, the limiting resource of the design, or the resource curve with the highest percent usage, is most important since that is what limits the overall size of the design. The performance graph has clock period on the left vertical axis and latency on the right vertical axis, both in nanoseconds. For both graphs, having lower curves is better. Lower curves mean fewer resources and faster computations.

There is a clear difference in resource usage that occurs when using DSP packing versus when not, as shown in the graph on the left of Figure 5.4. When using DSP packing, there is an increase in DSPs and decrease in LUTs. This increase is so significant that with DSP packing the limiting resource is DSPs while without DSP packing the limiting resource is LUTs. This is counterintuitive. One would think if you are able to combine multiple multiplications into one DSP versus using one DSP for each multiplication, there would be a decrease in DSP usage. However, since the only bitwidths that DSP packing can be used for are bitwidths 8 or less, these multiplications are small enough and simple enough that they can be easily implemented in LUTs. When using DSP packing, we are forcing the multiplications to be done in DSPs. In terms of performance, as can be seen in the graph on the right of Figure 5.4, latency is consistently better, and clock speed is either better, or equal, when not using DSP packing. Since our benchmarks rely significantly on matrix multiplication which will commonly make the limiting resource of the benchmark DSPs, forcing low bitwidth multiplications into DSPs is not a good tradeoff. Therefore, for the rest of this paper we will not use DSP packing. However, if there was a design that was LUT limited and DSPs were readily available, then using DSP packing would be a viable option since DSP packing reduces the overall LUT usage.

5.2.2 *Initial Results*

After creating a hand-written version of the one-layer model that utilizes best design practices and does not utilize DSP packing, we compared our resource usage and performance results to the HLS design. This comparison is shown below.

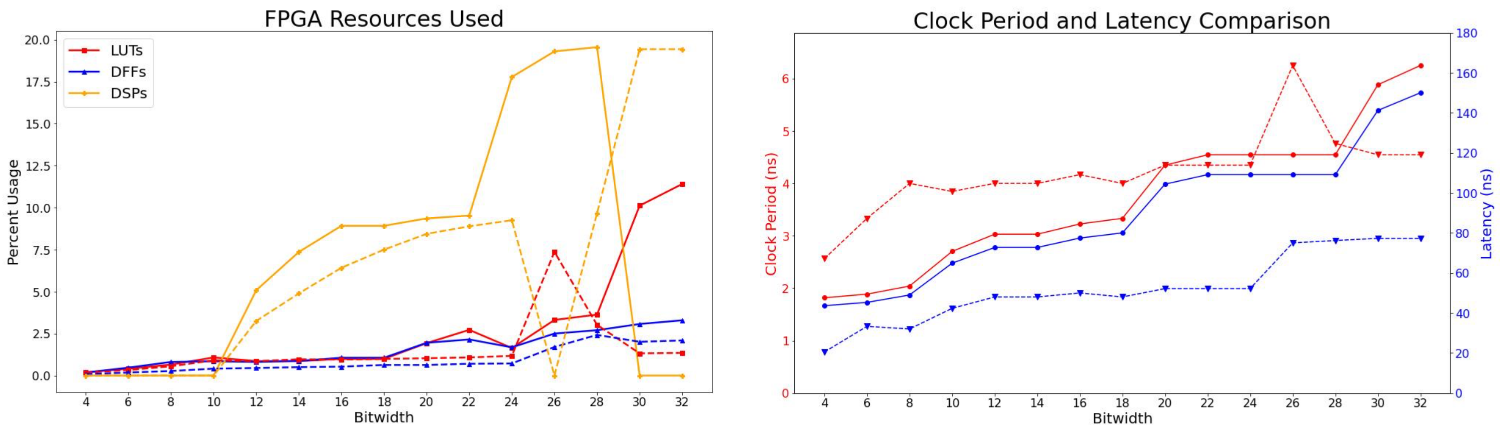


Figure 5.5. Initial resource and performance comparison for the one-layer model.

The solid lines are the SV design, while the dashed lines are the HLS design. Like with the above multiplier packing graphs, on the left we see the resource usage results and, on the right, we see the performance results. Here the x-axis is still bitwidths, but the range has expanded to bitwidths 4 to 32.

While the graphs contain some interesting irregularities, the story is consistent. The HLS designs outperforms the optimized SV design on almost all metrics. While for bitwidths less than 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 bitwidths greater than 20 bits. The SV version has a 1.7x slower latency overall, uses roughly 1.76x more FFs and 1.08x more DSPs, and between 1x to 10x the number of LUTs (excluding the anomalous 26 bitwidth). In terms of maximum resource usage, for both designs, the area is dominated by LUTs and FFs for up to bitwidth 10, and DSPs at all other 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 produce better solutions on almost all metrics. These differences are explainable and point out surprising optimizations in the Vivado HLS flow that allow it to shine in comparison to best-practice Verilog hardware design. To make it so that our SV design was able to outperform the HLS design, we had to dig into these differences. We will now explain this exploration and the results.

5.2.3 *Pipelining and Performance Optimizations*

After analyzing the initial results for the one-layer model and seeing that our SV design has a significantly worse latency than the HLS design, we learned we needed to optimize our pipeline stages. In our initial design, we thought to obtain optimal performance we should pipeline all operations. For example, the adder tree for the accumulate operation would add two terms per pipelining stage. 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 and the number of total LUTs and FFs utilized. Furthermore, we experimented with 1 pipeline stage and 3 pipeline stage DSP configurations. We ended up sticking with our original 3 stage configuration since we discovered a 1 stage pipeline would limit our clock speed with only a slight decrease in FF usage and not a significant latency improvement. After making these changes, we were able to see our performance converge closer to HLS's. These results will be shown is section 5.2.6.

5.2.4 *Resource Usage Optimizations - The Missing DSPs*

In order to understand how HLS was able to use significantly fewer resources than our design, we chose to explore the difference in DSP usage between the HLS and SV design. In the resource graph of Figure 5.4, notice that for bitwidths 12 to 22 the HLS version consistently uses a few less DSPs than the SV version. This is somewhat strange. Both versions implement multiplication by simply using their source languages' "*" operator, and multiplications of these bitwidths should easily fit into a single DSP block. Additionally notice that as the bitwidth goes down from 22 to 12 bits, the number of DSPs used also decreases. However, the number of multiplications done by the algorithm stays constant as bitwidth varies.

For a multiplication to be done in LUTs, it is broken down into several partial products of shifted "ones", or shift-adds. For example, consider calculating `output = input*6`. We could use a DSP to implement this, or instead we could 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. However, this is only viable for "easy" constants, or what the tools can optimize for. As bitwidth decreases, more constants are "easy", and more multiplications can be converted to these 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 Vivado synthesis tools perform this transformation, but the HLS tool has a more powerful optimizer.

To test this, we created simple Vivado HLS and Vivado designs that perform a single constant multiplication and compiled these designs through the two tools. To understand the limit of these shift-adds, the constants were set to a range of values. We tested both obvious “easy” constants, as well as constants chosen from the one-layer model that the HLS tool is compiling into shift-adds while the SV design does not.

Through this we found that the HLS tool appears to convert any constant multiplication to shift-adds where the constant is in the form $\pm(input \ll c1) + \pm(input \ll c2)$, for any c1 or c2. For the standard Vivado flow, the conversion appears to be limited to $\pm(input \ll c1) + \pm(input \ll c2)$ where c1 and c2 must be 3 or less, as well as $\pm(input \ll c)$ for any c. For example, in our tests we found that the constant `20'h01010` is converted to LUTs by Vivado HLS but is implemented in DSPs by Vivado. Even though there are only two “ones” to be converted to shift-adds, since the “ones” are located at bits 2 and 4, this is not converted to LUTs by Vivado. This difference, where the HLS tool has a more powerful conversion optimizer 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 FPGA hand design for decades.

5.2.5 *SystemVerilog Shift-Add*

To improve our hand-written version’s DSP usage, we developed our own constant multiplication module that can optimize constant multiplications further than Vivado HLS. This unit takes in an input, plus a constant weight as a parameter, and automatically determines whether to use shift-add and have the multiplication be done in LUTs, or a standard “*” operation which is converted by Vivado to a DSP. The module, via SystemVerilog functions and generate statements, iteratively applies the conversion:

$$input \cdot weight = \pm(input << c) + input \cdot (weight \pm 2^{c}) \tag{5.2}$$

where 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 $(weight \pm 2^c)$ term is non-zero, the tool decides that the multiplication should not be done via shift-add, and instead uses a DSP.

DEPTH is a tuning factor. For DEPTH = 2, we can support $\pm(input \ll c1) + \pm(input \ll c2)$ for any c1 and c2, which is the exact optimization done by Vivado HLS. While DEPTH = 3 can support $\pm(input \ll c1) + \pm(input \ll c2) + \pm(input \ll c3)$ for any c1, c2, and c3, DEPTH = 1 only converts pure powers of two, and DEPTH = 0 turns off the optimization completely. The value of DEPTH should be chosen based on a comparison of the LUT and DSP availability on the targeted FPGA. It is important to note that a change in DEPTH has no impact on the amount of FFs used. Therefore, we performed a comparison of the LUT versus DSP usage for various DEPTH values for the one-layer design on the Virtex 709. The results of this, normalized to the total number of each resource available, is shown below.

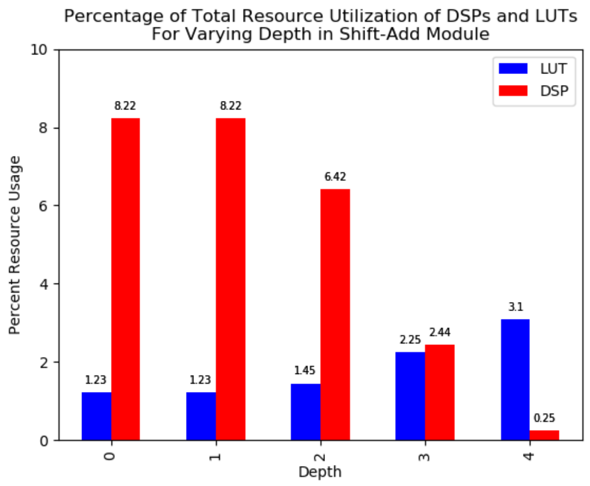


Figure 5.7. Comparison of LUT usage versus DSP usage for the Virtex 709 for the one-layer design at a bitwidth of 16.

Since a DEPTH = 1 shows no benefit versus DEPTH = 0, this means that none of the constants are powers of two. For DEPTH values above 1, there is a steady decrease in the use of DSPs, shown in red, and a steady increase in the LUT usage, shown in blue. As can be seen, DEPTH = 3 is the best match to the resource mix found in the Virtex 709.

5.2.6 *Improved SV with Manual Shift-Add*

In order to improve our hand-written design compared to the HLS design, we implemented the performance and resource usage changes described in sections 5.2.3 and 5.2.5. This includes tweaking of the pipelining of the design to better use FFs and an added manual shift-add module with DEPTH=2, to match HLS's DSP usage. After these changes, we began to see the SV design's resource utilization and performance converge with that of the HLS design. This is shown below.

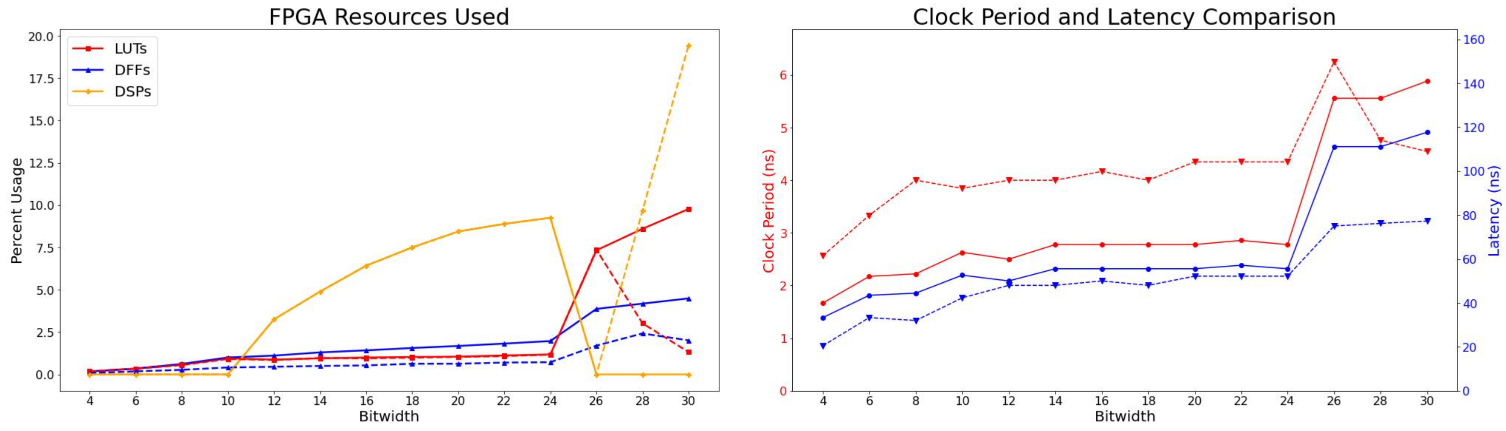


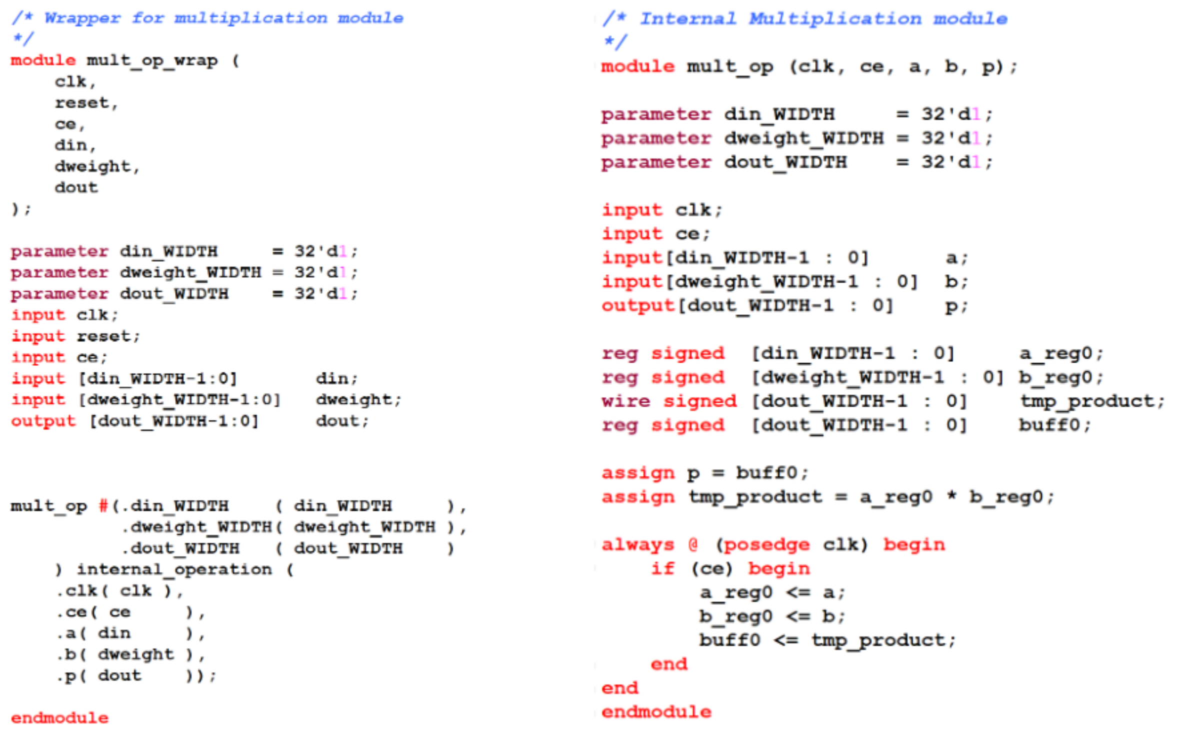
Figure 5.8. SV vs HLS one-layer design with a shift-add DEPTH = 2.

The SV design is the solid lines while the HLS design is the dashed lines. As expected, by including the manual shift-add routine with a DEPTH = 2, for bitwidths of 24 or less, the DSP usage is identical between the two implementations. For these bitwidths the SV version uses on average 2.55x more FFs, and 1.03x more LUTs; the clock speed is 0.65x lower, while the latency is 1.17x worse. With the added optimizations, the SV design still uses overall more resources, but the DSP usage matches HLS's, and performance has improved. However, the DSP usage between the two designs transitions at bitwidths above 24. At this bitwidth, multiplications stop fitting into single DSPs within the Virtex 709 which has DSPs of size 25x18. We will consider these high bitwidth cases below.

5.2.7 *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. As seen in Figure 5.8, 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.

The difference? The HLS code uses the multiplication subroutine shown below.



```
/* 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 5.9. HLS magical multiplication subroutine.

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 pipeline 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.

When we include the HLS magical multiplication subroutine into our SV implementation, we get the results shown below.

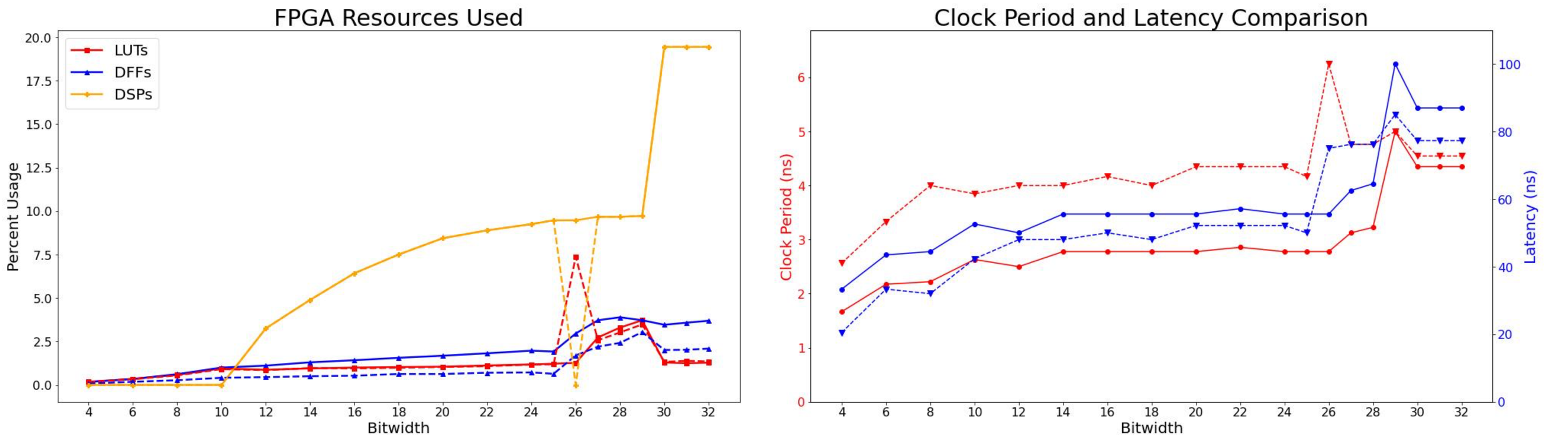


Figure 5.10. SV vs HLS one-layer design with a shift-add DEPTH = 2.

Notice that for DSP usage, the SV design and the HLS design have identical results. 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 2 DSP logic at bitwidths above 25, it fails to catch the bitwidth 26 case.

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 FFs; the clock speed is 0.69x lower and latency is 1.11x slower. Using only slightly more resources, the SV version can obtain significantly faster performance. For both designs, at a bitwidth of 10 or less, the limiting resource for both systems is either LUTs or FFs, while after a bitwidth of 10, the Vivado tools begin to infer DSP slices for multiplication, and DSPs become the dominant limiting resource.

5.2.8 *Final Optimized Version*

To optimize our design even further than the HLS version, and since DSPs are the limiting resource for bitwidths greater than 10, we decided to tune the DEPTH parameter in the shift-add module based on the limiting resource at each bitwidth. The results are shown below.

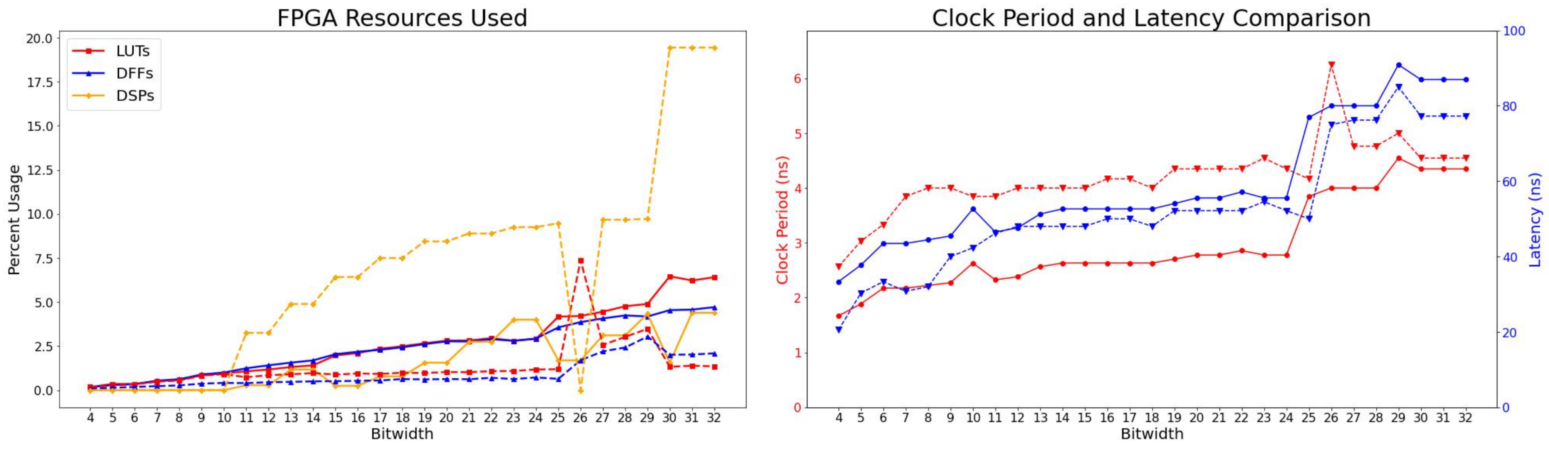


Figure 5.11. Per-bitwidth DEPTH tuning.

The SV design, shown with the solid curves, uses a DEPTH = 2 for bitwidths less than 11, DEPTH = 3 for bitwidths 11-14, DEPTH = 4 for bitwidths 15-24, DEPTH = 5 for 25-29, and DEPTH = 6 for 30-32. The HLS design, shown with the dashed curves, inherently uses a DEPTH = 2 for all bitwidths. On average, the SV version uses 0.23x the number of DSPs, 1.97x LUTs and 2.69x FFs; clock speed is 0.67x lower and latency is 1.12x worse. The DSP to LUT tradeoff is due to the multiplications now being done almost entirely in LUTs. However, our SV version is still only using a maximum of 6.4% of the total LUTs available. Further, even though we are only using around 4% of the total FFs, it is significantly more than that used by the HLS version. However, this increase in FFs allows us to have a higher clock speed than HLS's due to more pipelining.

When comparing the HLS design to our optimized design, 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 the depth of the shift-add module, our SV version can shift multiplications between LUTs and DSPs to whichever is more abundant.

So far, we have presented a comparison of hand-written designs to an HLS4ML generated design for a simple neural network. Although our initial implementation of the SV model, which corresponds to best practice FPGA hand design, compared 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? We will now extend our findings to a slightly larger model.

5.3 Benchmark 2: Basic 2D CNN

The second benchmark is a 2D CNN model that was trained on the MNIST handwritten digit dataset, a popular dataset used for image recognition tasks [16]. This model is made up of 4 layers as shown below.

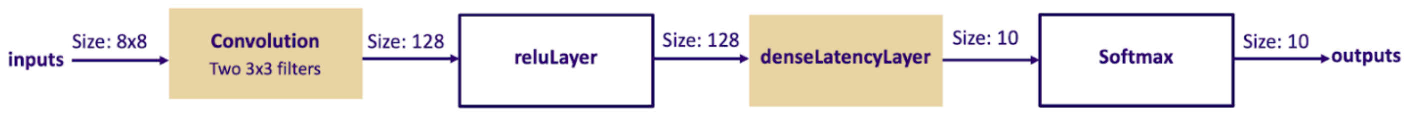
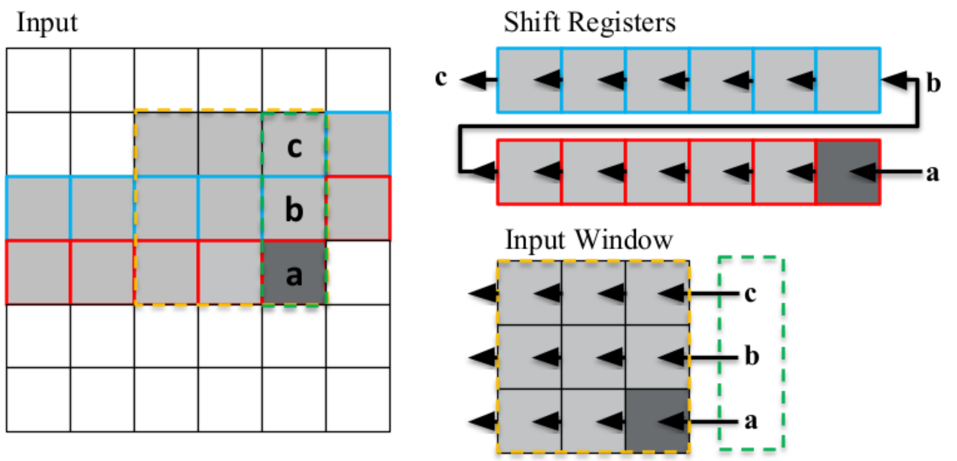


Figure 5.12. Basic 2D CNN layers.

The first layer, a convolution layer, convolves two 3x3 filters over the 8x8 input matrix that is padded on all sides. 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.

Although only slightly larger than the one-layer model, this design requires significant internal storage and staging due to the convolution layer. If not done properly, the FPGA will run out of resources or will require a suboptimal amount of clock cycles to process the data.

For this model, the data is streamed in one data piece per clock cycle. Therefore, to stream in the whole 8x8 image, it takes 64 clock cycles. This is done in the manner explained in section 4.3. This allows for an efficient implementation as the overall resources for the inputs are lowered due to a decrease in the overall IO pins used. Further, a Line Buffer approach is utilized to efficiently process the data as it is streamed in. This Line Buffer approach is shown below for a 6x6 input image.



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 5.13. CNN Line Buffer approach [17].

The Line Buffer approach creates a pipelined convolution layer. Rather than storing the whole input image as it is streamed in, the only storage used is a 3x3 matrix, labeled as the input window in Figure 5.13, representing the data currently being multiplied by the filter, and two shift registers. Each cycle, the data in the 3x3 matrix is shifted to accept the new data piece, input *a* in Figure 5.13. Note that the data in the bottom two rows of the 3x3 matrix will be reused as the image is processed. For the data pieces that will be used again, they are held in shift registers [17]. The Line Buffer approach allows for efficient resource usage and performance. HLS4ML utilizes this approach and since it is efficient for this convolution layer, we implemented the same approach in our SV design. Finally, like the first benchmark, we define this model to have an II of 1. This means the II will be 64 cycles since it takes 64 cycles to read in one input image.

5.3.1 *Initial Results*

Using the same design process as used for the one-layer model, we initially set out with the goal of matching HLS4ML's accuracy while outperforming in terms of resource usage and performance. To do so, we utilized the Line Buffer approach as explained above and pipelined all computations. Additionally, just like with the one-layer model, we wanted our initial results to highlight the differences between HLS4ML, and the HLS tools, versus hand-written designs. Therefore, for our SV design, we did not use the previously described shift-add module or other optimizations discovered through the one-layer model. The initial comparison of the HLS to SV design for this basic CNN is shown below.

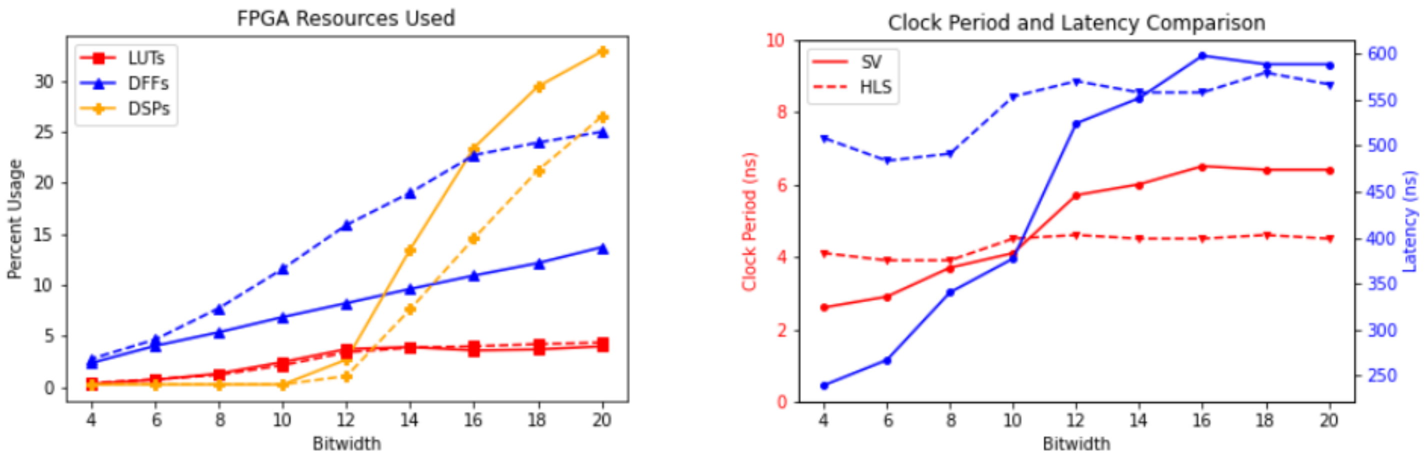


Figure 5.14. Initial basic CNN results.

Note that the HLS4ML configuration used had a clock period constraint of at most 5 nanoseconds. At bitwidths above 20, that constraint is not met. Therefore, our x-axis is now limited to bitwidths of 4 to 20. The vertical axes have stayed the same as in the one-layer design where on the left we see the percent of the total resources used for DSPs, FFs, and LUTs, and on the right we see clock period on the left vertical axis and latency on the right vertical axis. The SV design is represented by the solid lines and the HLS design is represented by the dashed lines.

As shown in the resource graph on the left of Figure 5.14, initially the SV design is using 0.94x the number of overall resources than the HLS design. Since we are not using the shift-add module, the SV design is using 1.38x more DSPs. However, the reduction in overall resource usage comes from the SV design using 0.55x less FFs than the HLS design. The fewer FFs used corresponds to the HLS design having better performance than our design due to more efficient pipelining. The SV design has a 0.84x lower latency than the HLS design, but the SV design period is 1.13x worse than the HLS design.

These initial results are consistent with the initial results of the one-layer model. The hand-written design is using overall more resources (ignoring FFs) and is performing computations at a slower rate than the HLS design. Therefore, using the same approach as the one-layer design, we looked to optimize our design in comparison to the HLS design.

5.3.2 *Optimized Results*

To optimize our results, we looked at both resource usage and performance. Just like with the one-layer design, we needed to decrease our overall number of DSPs used. To do so, we implemented the shift-add module as described for the one-layer model. For the shift-add module, this was implemented with a DEPTH = 3 as this DEPTH has the best tradeoff of LUTs vs DSPs for the Virtex 709 as shown in Figure 5.7. For performance, we transitioned to a 4-stage adder tree pipeline, versus our original 2-stage adder tree pipeline. Our optimized results are shown below.

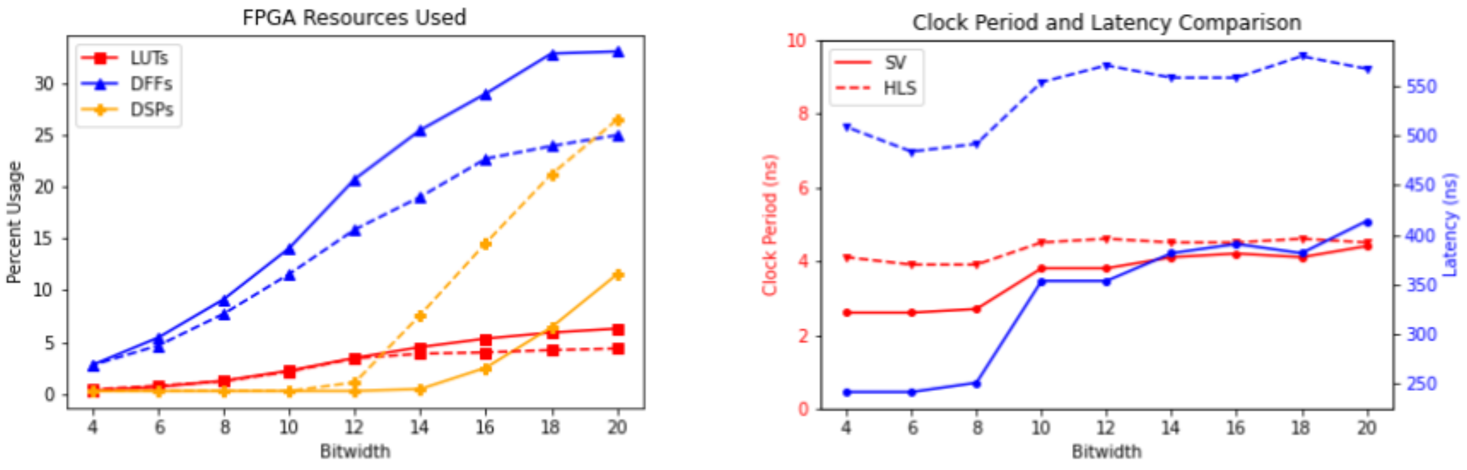


Figure 5.15. Optimized basic CNN results.

Based on the comparison shown in Figure 5.15, using the described optimizations, the SV design is now comparable to the HLS design. We are outperforming in terms of DSP usage and latency but are using significantly more FFs. Our overall resource usage grew to 1.25x that of the HLS design. On average the SV design used 0.58x the DSPs and achieved a 0.82x lower clock period than the HLS design. Our latency continued to be better than HLS's: initially the SV design was 0.84x lower, and in the second implementation it became 0.62x lower. However, initially we were using 0.55x the number of FFs used in the HLS design, now the SV design is using 1.29x the number of FFs used in the HLS design. However, it is this increase in FFs that allowed our SV design to outperform the HLS design in terms of latency and clock period through adding more pipeline stages.

Overall, after going through the same optimizations as followed for the one-layer model, we were able to produce a more efficient implementation than the HLS design. However, the models are very close in terms of efficiency considering the tradeoffs between FFs and performance. We do see that by being able to use a DEPTH of 3 for the shift-add module, versus the DEPTH of 2 that HLS uses, we are able to significantly reduce the number of DSPs used.

5.4 Benchmark 3: Complex 2D CNN

After analyzing the prior two benchmarks, we wanted to see how HLS4ML designs can compete with hand-designs when using more advanced parameters in the configuration of the design. Specifically, we wanted to look at how HLS4ML designs do with adjusted reuse factors as well as stride of the convolution layer. For our third and final benchmark, we implemented the encoder portion of the Econ-T Autoencoder [18]. An autoencoder is an encoder and decoder in series and is a common structure used in machine learning algorithms. The Econ-T autoencoder was trained on the data produced by the Compact Muon Solenoid (CMS) Endcap Calorimeter at CERN. This experiment looks at the energy of particles moving perpendicular to the path otherwise followed by the particles [19].

The layers of this model are shown below.

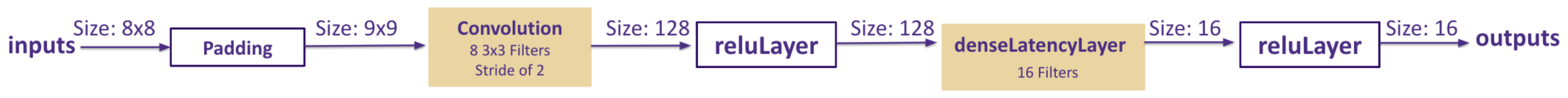


Figure 5.16. Complex CNN layers.

The input size is the same as the previous model, an 8x8 matrix, but has padding only on the top and the bottom, resulting in a 9x9 matrix. This 9x9 matrix is then put into a convolution layer with a stride of 2 and reuse factor of either 3 or 9. The convolution layer has 8 filters each processing the input image. The outputs of the convolution layer then go through a ReLU, a dense latency layer with 16 filters, and finally another ReLU. The model results in 16 outputs per the given 8x8 input image.

One important thing to note is that the previously discussed shift-add module cannot be used when the reuse factor of a layer is greater than 1. With a reuse factor greater than 1, the input to the multiplication is no longer just an input and a constant value. It instead becomes an input being multiplied by a variety of constants, depending on what cycle of the computation we are on. A mux is needed to determine the constant that will be used for each of these specific multiplications. Therefore, the optimizations for this benchmark look quite different from the prior two.

We will now go over our two implementations (a reuse of 3 and a reuse of 9), and the corresponding optimizations and results.

5.4.1 *Reuse of 3*

Since this convolution layer now utilizes a stride of 2, the previously used Line Buffer approach is no longer the best solution. When computing the convolution of the 9x9 image (the 8x8 input image with padding), one 3x3 filter will have 16 outputs. This means that the dot product of the filter and the input image is computed 16 times. In comparison to the stride of 1 example in the basic CNN with padding on all sides, that dot product was computed 64 times.

The 16 locations of the filter are shown below.

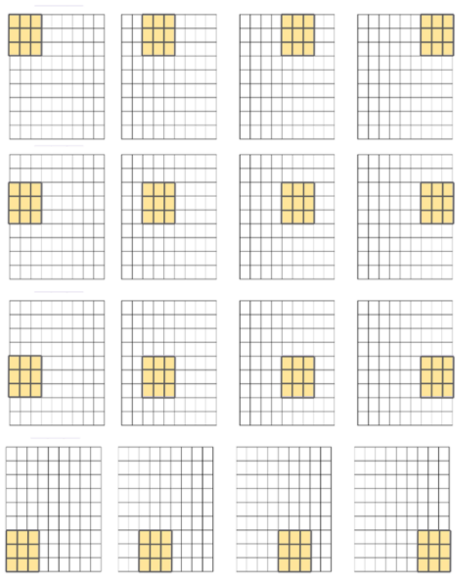


Figure 5.17. 16 filter locations of a 9x9 matrix with a 3x3 filter and stride of 2.

Since the filter runs at 16 locations, in a model with a reuse of 1, this would only take 16 cycles. Because our implementation has a reuse of 3, we give each filter location 3 cycles to compute its output. The first filter location being broken down into a reuse of 3 is shown below.

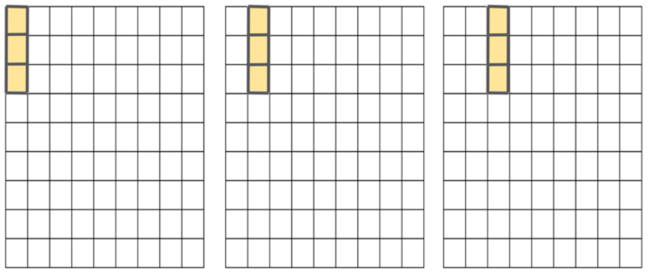


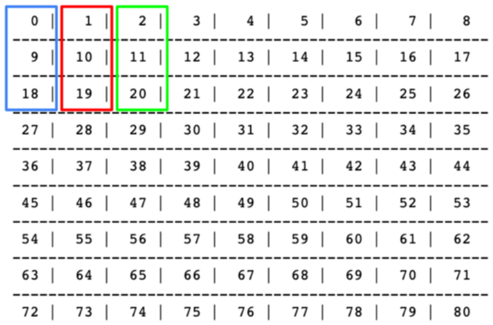
Figure 5.18. First filter location with a reuse factor of 3.

With each filter location being broken up as such, we would see the 16 locations now each taking 3 cycles each. In total, this convolution computation would require 48 cycles.

However, since the data is being streamed in, the computation is not able to start right away. As can be seen in the first filter location of Figure 5.18, the first output relies upon data in the first three rows. Therefore, there needs to be some initial delay in the processing to wait for this data to be filled in. Additionally, hiccups in the processing will occur with the vertical jumps down by a stride of 2. Regardless, since it takes 64 cycles to load the data in, and the processing will take at least 48 cycles, careful overlapping of communication and computation can at best handle one complete inference every 64 cycles.

Because of this design limitation, we set out with the initial goal of designing our system to be able to accept a new image every 64 cycles. Therefore, the goal II of the design is 64 cycles. To accomplish this, we dug further into how we can utilize the clear pattern that the filter locations follow, as shown in Figure 5.17. For the first four locations, the filter moves horizontally by two. Then after moving down vertically by two, this repeats until the whole image has been processed. The filter locations following this cyclical pattern implied to us that the addresses that the filter is reading from also follow this same cyclical pattern. Therefore, a simple implementation would be to store the input data that is being streamed in into a large array that can represent the whole input image. We can then read from this large array at the addresses of the current filter location.

This large matrix implementation is not as simple when being brought into hardware realizations. This large array will be implemented as a BRAM. Individual BRAMs only have one read and one write port. Therefore, since our reuse of 3 implementation requires reading from 3 values of the array each cycle, we need 3 identical BRAMs. This allows us on each cycle to be able to read from the correct addresses. We broke down each filter computation as shown below.



0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

Figure 5.19. First filter location broken up by reuse factor of 3.

The first cycle is represented by the blue rectangle, the second cycle by the red rectangle, and the third cycle by the green rectangle. These three rectangles represent the first filter location. Addresses 0, 1, and 2, would be read from BRAM1, addresses 9, 10, and 11 would be read from BRAM2, and addresses 18, 19, and 20 would be read from BRAM3. The read addresses for each BRAM as well as the write addresses for where the input data should be written to are controlled by two FSMs: an FSM for reading and an FSM for writing. By being able to use a BRAM to store the data and two FSMs to control the processing of the data, we were able to minimize the logic needed for the implementation of this complex CNN.

After successfully implementing our FSM and BRAM based complex CNN, we then compared our resource usage and performance results to the HLS4ML implementation. The resource usage and performance comparison are shown below.

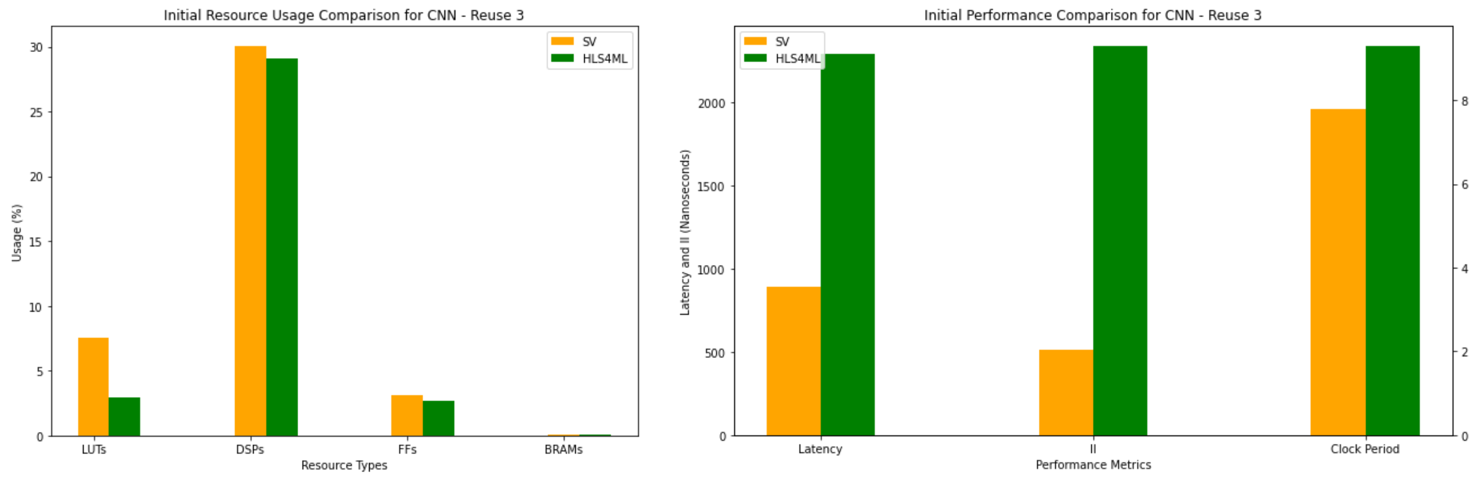


Figure 5.20. Complex CNN initial resource usage with a reuse of 3.

Note that since we are now focusing on how reuse factor changes the efficiency of the design, unlike the first two benchmarks, we are now only looking at bitwidth 16. These graphs show the same resource and performance comparisons as the prior two benchmarks, but with the x-axis being the resource types and performance metrics used. The left vertical axis of the performance graph is the scale for latency and II, while the right vertical axis is the scale for clock period. The SV implementation, shown in orange, is using 1.6x more LUTs, 1.03x more DSPs, 1.16x more FFs, and 3x more BRAMs. This significant increase in BRAM usage is due to the SV design utilizing 3 BRAMs to store the input matrix. However, the SV design is only using 0.10% of the total BRAMs available. In terms of performance, the SV version has a 0.83x lower clock period, 0.39x lower latency, and a 0.26x lower II. The II of the SV design is 66 cycles, very close to our goal II of 64 cycles, and the II of the HLS design is 252 cycles. Although the resource usage of the SV design is slightly higher than the HLS design, the SV design is significantly faster than the HLS design in all regards.

Where is this performance difference coming from? Just like with the one-layer design, we dug into the HLS4ML implementation to understand the differences. The HLS4ML design starts out with a padding layer. This layer takes in the input data piece and stores it to a BRAM. However, unlike the SV design, the BRAM is only used for debugging purposes and the data is streamed out directly from this padding layer to the convolution layer. The convolution layer is the same as it was for the Basic CNN implementation except the parameters for reuse factor and stride have changed to 3 and 2 respectively. This means that the HLS design is still utilizing the Line Buffer approach and is looping through the 9x9 matrix and stalling for 3 cycles at each location. This takes the design 81 x 3 = 243 cycles. Rather than optimizing for the simplicity of the stride of 2 design, HLS4ML is trying to utilize the same CNN implementation for all strides.

Even though we see this comparison as a strong win over HLS4ML, we realized our design is not as good as it could be. The convolution itself is quite efficient, but the dense latency layer is stalled until the convolution finishes. Then the dense latency layer utilizes 2048 DSPs over 3 cycles to compute the final output. In order to optimize for fewer DSP usage in the dense latency layer, we explored how the dense latency computation can overlap with the convolution layer.

5.4.2 *Pipelined Dense Latency Layer*

To have the dense latency computation be done during the convolution layer, we created a pipelined dense latency layer. Because the convolution layer in this benchmark has 8 different filters processing the same input image, every 3 cycles, corresponding to the reuse of 3, the convolution layer outputs 8 different values. This makes it so in the end the convolution layer has 16 x 8 = 128 different outputs. Then, the dense latency layer has 16 different filters that each process the 128 outputs of the convolution layer. For each filter the following is computed:

$$output_j = \sum_{i=0}^{128} \left(convolutionOutput[i] \cdot \mathrm{weight}_j[i] \right) + \mathrm{bias}_j \tag{5.3}$$

Clearly, each dense latency layer output is a running sum of the convolution output being multiplied by the proper weight, with a final bias term added at the end. Therefore, instead of waiting for the convolution layer to be done and sending all 128 outputs to the dense latency layer, we can instead send the values along as they are calculated. This gives the dense latency layer many clock cycles to process this early-arriving data.

This implementation can be improved further to account for the reuse of 3. Because the convolution output does not change for 3 cycles, the 8 convolution outputs can be multiplied by their corresponding weight values over 3 cycles instead of doing 8 multiplications in one cycle. We do 3 multiplications in the first cycle, 3 multiplications in the second cycle, and finally the last 2 multiplications in the third cycle. This repeats for each new set of convolution outputs, or every 3 cycles. After computing this running sum 16 times, we end up with the same summation as the previous implementation. However, with this implementation, each filter in the dense latency layer is only ever using 3 DSPs and on the cycle after the convolution ends, the dense latency layer also ends. Note that this can be adjusted for any reuse factor. The DSPs used per cycle will be adjusted based on how many cycles there are between the convolution outputs.

After implementing the pipelined dense latency layer into the SV design, we again compared our results to the HLS design. When doing this comparison, we also tried increasing the reuse factor of the dense latency layer within HLS4ML to see if the tools could infer a similar performance enhancement to our pipelined dense latency layer. The comparison is shown below.

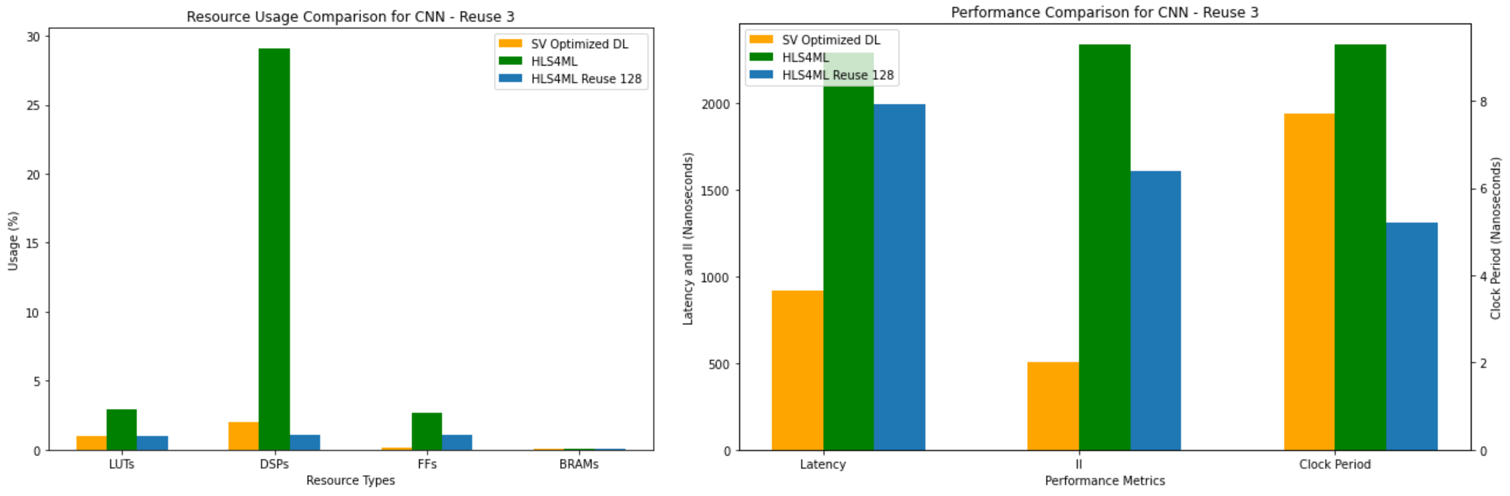


Figure 5.21. Complex CNN resource usage with a reuse of 3.

The orange SV bars show the SV implementation with the pipelined dense latency layer (Optimized DL), the green HLS4ML bars show the original HLS design, and the blue bars show the HLS design with the dense latency reuse factor increased to 128. A reuse factor of 128 is the closest value to the pipelined dense latency implementation that the HLS design could be set to.

We see that by utilizing the pipelined dense latency layer, the SV design uses a fraction of the total resources used by the initial HLS design and a similar number of resources used by the HLS design with a reuse factor of 128. The max resource used by the SV is design is only 2% (DSPs) while the original HLS design's max resource used is 29.11% (DSPs) and the HLS design with a reuse of 128 uses 1.11% (DSPs). Besides DSP usage, compared to the HLS design with a reuse of 128, the SV design uses 1.04x more LUTs, 0.16x less FFs, and 3x more BRAMs. Besides being able to use significantly fewer resources than the original HLS design and around the same max resource usage for the HLS design with a reuse of 128, the pipelined dense latency layer continues to allow the SV design to be significantly faster. The SV design with the pipelined dense latency layer has a 0.45x lower latency and a 0.21x lower II than the HLS design with the increased reuse factor.

Overall, we see that adding the pipelined dense latency layer allowed the SV design to continue to be significantly faster than the HLS design, while using at most 2% of the total resources available.

5.4.3 *Reuse of 9*

With a 3x3 filter, the different reuse factors that can be used are 1, 3, and 9. With a reuse factor of 9, this means that each filter location will take 9 cycles to compute. With this increase in cycle time to compute each convolution output from a reuse of 3, our pipelined dense latency layer is given more cycles to compute its running sum as well. This allows for fewer resources to be used in both the convolution and pipelined dense latency layer than in a reuse of 3.

Just like with the reuse of 3, when utilizing the pipelined dense latency layer, we increased the reuse factor of the dense latency layer in the HLS design. Here, a reuse factor of 128 directly matches the reuse factor of the pipelined dense latency layer. The comparison of the SV design to the HLS design with a reuse of 128 is shown below.

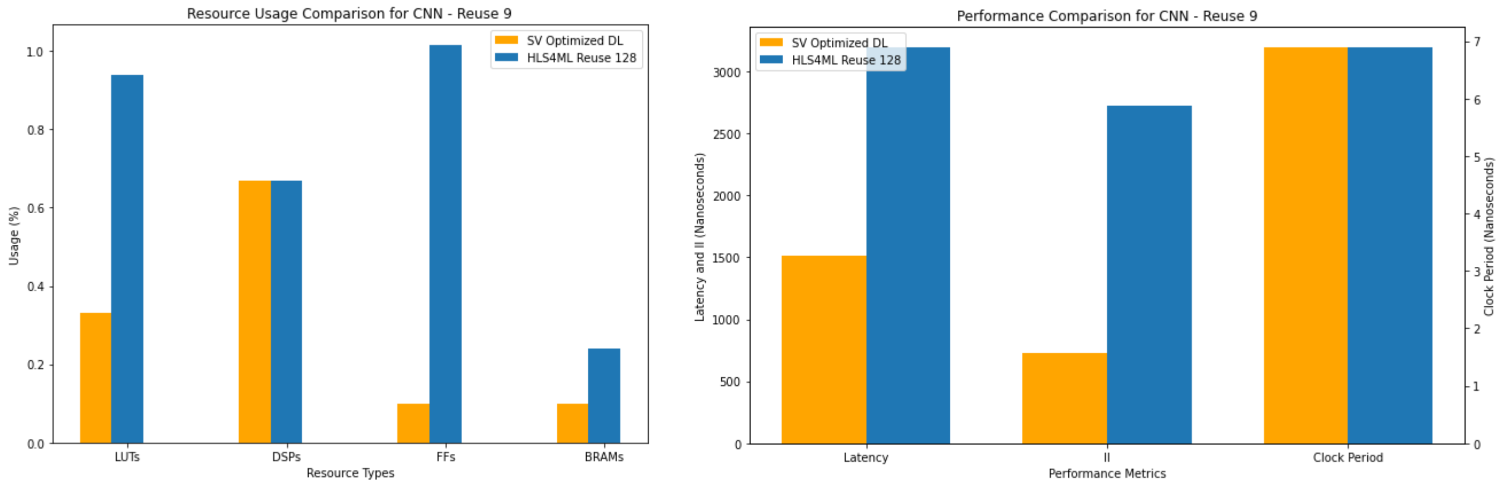


Figure 5.22. Complex CNN resource usage with a reuse of 9.

The SV design is shown in orange and the HLS design is shown in blue. In terms of resource comparison, the SV design uses 0.35x less LUTs, the same number of DSPs, 0.09x less FFs, and 0.125x less BRAMs. Further, the SV design has the same minimum clock period as the HLS design, but a 0.45x lower latency and a 0.26x lower II. These results further confirm the results we saw with a reuse of 3. By using our BRAM and FSM based implementation, avoiding the Line Buffer Implementation, and by using the pipelined dense latency layer, the SV design can outperform the HLS design in terms of all resources used and performance.

Section 6. DISCUSSION

It is common knowledge in the FPGA community that, while HLS can be an efficient way to quickly develop an implementation for a design, hand optimization is the gold standard for achieving the highest quality implementation. After analyzing the discussed three benchmarks, we are conflicted on this front. For the first two benchmarks, we were eventually able to achieve hand-optimized results that were in some sense equivalent or better than the HLS results. However, 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. Whereas in the third benchmark, we were able to outperform the HLS designs on all fronts.

For simple models, 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 Verilog. We see the benefits of this in the first two benchmarks.

However, from the third benchmark, we saw that this approach falls short of hand-designs when the model's parameters are adjusted from the default parameters. By increasing the complexity with a higher stride and reuse factor, we were able to see a drawback of HLS designs. The rigidity of the HLS code that is required for it to be automated by Vivado HLS does not allow for strong implementations of models with complex parameters. It is unclear how you could change the logic of the HLS4ML C++ code to follow the implementation of our hand-written implementation. Instantiating 3 BRAMs and utilizing two FSMs to control the reading and writing of the BRAMs is not compatible with the requirements of HLS code. When attempting to implement this in HLS, the incompatibility comes down to the requirements of the `io_stream` implementation. This requires data to be streamed out of every layer and limits the design's ability to use the BRAM in parallel.

When describing reuse factor, HLS4ML states, "Low reuse factor achieves the lowest latency and highest throughput but uses the most resources, while high reuse factor saves resources at the expense of longer latency and lower throughput" [1]. However, this does not need to be true. Increasing a reuse factor by 3 does not need to increase the number of cycles for the computation by 3. As we saw with the third benchmark, the best way to process this benchmark is significantly different than the approach used by HLS4ML. By acknowledging the adjusted parameters, making a custom design allows for significant improvement in resource usage and performance.

Section 7. CONCLUSION

While HLS tools are attractive due to their decrease in development time, this thesis showed that when dealing with complex designs, HLS tools fall short of what one can accomplish by creating a custom design. We were able to clearly see with our third benchmark how understanding the computation being done allows for hand-written designs to soar above HLS designs. With this benchmark we saw our custom design obtain a max resource usage of 2.70% in comparison to HLS4ML's max resource usage of 29.11%. Further, our design has on average a 0.47x lower latency and a 0.29x lower II.

However, HLS4ML shows significant promise with the basic layers shown in benchmarks 1 and 2. By being able to go above what the tools are normally able to do with the shift-add module and the magic multiplication subroutine, these simple layers were able to perform better than our initial hand-written designs. On average, our initial design started out with a 1.23x increase in resource usage, a 1.27x higher latency, and a 0.95x lower clock period. This wasn't a result of the HLS code, but rather a difference in the optimizations of the tools itself. After digging into the differences, we were able to obtain a 0.82x decrease in resource usage, a 0.87x lower latency, and a 0.78x lower clock period.

From these results, we see that for simple designs using standard parameters, using HLS is a great idea. One can take advantage of the more-advanced optimizations done by Vivado HLS than in Vivado. However, for more complex designs, it is best to stick to the gold standard of hand-written designs.

Section 8. NEXT STEPS

Regarding the third benchmark, HLS4ML should consider creating a custom implementation of their convolution layer to account for adjusted strides and reuse factors. Even though it is not compatible with HLS, the Python flow can import SystemVerilog directly to ensure that the custom implementation will be used. More generally, HLS4ML should implement a pipelined dense latency layer to further optimize for resource usage. Further exploration should be done to see when advanced parameters are adjusted for other layers if similar problems are seen in HLS4ML.

Section 9. MY CONTRIBUTIONS

This comparison work was originally started by a team of Matthew Bavier, Oleh Kondratyuk, Trinh Nguyen, Aidan Short, Jan Silva, Anatoliy Martynyuk, and me. This group worked together to develop the one-layer model, our first benchmark. Aidan and Matthew helped with figuring out the HLS4ML flow and worked together to find out how we were going to implement each layer. Jan and Oleh explored DSP packing. I developed the Softmax layer for this model. Trinh and Anatoliy developed the Shift-Add module. Trinh also developed our Python scripts to convert weights and biases into SystemVerilog parameters. Oleh also discovered the magical multiplication subroutine and optimized our dense latency layer. Finally, Trinh and Oleh analyzed the one-layer model in comparison to the HLS4ML design.

For the Basic CNN model, our second benchmark, I developed our hand-optimized implementation. This required implementing the Line Buffer approach in SystemVerilog and utilizing our optimizations from the One-Layer model. I analyzed this benchmark in comparison to the HLS4ML design.

For the Complex CNN model, our third benchmark, I developed our hand-optimized implementation. This required designing the FSM and BRAM based implementation and the pipelined dense latency layer. I analyzed this benchmark in comparison to the HLS4ML design.

BIBLIOGRAPHY

[1] "Welcome to hls4ml's documentation". https://fastmachinelearning.org/hls4ml/.

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

[3] Russel, Stuart J. Artificial Intellgience: a Modern Approach. Upped Saddle River, N.J. Presentice Hall, 2010.

[4] Anh H. "Anh H. Reynolds." *Anh H. Reynolds*, https://anhreynolds.com/.

[5] "CS231n Convolutional Neural Networks for Visual Recognition". https://cs231n.github.io/neural-networks-1/.

[6] "Fully Connected Layers in Convolutional Neural Networks." *IndianTechWarrior*, 20 Apr. 2021, https://indiantechwarrior.com/fully-connected-layers-in-convolutional-neural-networks/.

[7] "HLS model class," HLS Model Class - hls4ml 0.7.1 documentation, https://fastmachinelearning.org/hls4ml/api/hls-model.html (accessed Nov. 10, 2023).

[8] "ExampleScriptForLoadingData.ipynb," GitHub, https://github.com/uw-acme/HLS4ML_VS_MANUAL/blob/main/documents/Benchmarks/ExampleScriptForLoadingData.ipynb .

[9] "Command line interface (deprecated)," Command Line Interface (deprecated) - hls4ml 0.7.1 documentation, https://fastmachinelearning.org/hls4ml/command.html.

[10] "Documentation Portal - HLS Pragmas," AMD Adaptive Computing Documentation Portal, https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas .

[11] 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.

[12] 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.

[13] 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.Andre18

[14] 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.

[15] 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.

[16] Fastmachinelearning, “Tutorial notebooks for HLS4ML - Part 6 CNNs” GitHub, https://github.com/fastmachinelearning/hls4ml-tutorial/blob/main/part6_cnns.ipynb.

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

[18] FastML-Science, “Ecoder,” GitHub, https://github.com/oliviaweng/fastml-science/tree/quantized-autoencoder/sensor-data-compression#input-data.

[19] “The phase-2 upgrade of the CMS Endcap Calorimeter,” CERN Document Server, https://cds.cern.ch/record/2293646?ln=en.

APPENDIX

Hand-Written Code
All hand-written code for the 3 benchmarks can be found at: https://github.com/uw-acme/HLS4ML_VS_MANUAL/tree/main/src.

Related Presentations

Caroline Johnson, Scott Hauck, Shih-Chieh Hsu, Waiz Khan, Matthew Bavier, Oleh Kondratyuk, Trinh Nguyen, Stephany Ayala-Cerna, Aidan Short, Jan Silva, Anatoliy Martynyuk, Geoff Jones, "Quantifying the Efficiency of High-Level Synthesis for Machine Learning Inference", *Fast ML for Science Workshop, ICCAD,* 2023.

Caroline Johnson, Scott Hauck, Shih-Chieh Hsu, Waiz Khan, Matthew Bavier, Oleh Kondratyuk, Trinh Nguyen, Stephany Ayala-Cerna, Aidan Short, Jan Silva, Anatoliy Martynyuk, Geoff Jones, "Evaluating the Quality of HLS4ML's CNN Implementations on FPGA's", *WAFER: Womxn at the Forefront of ECE Research, 2023*.

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.

Caroline Johnson, Oleh Kondratyuk, Trinh Nguyen, Matthew Bavier, Anatoliy Martynyuk, Aidan Short, Jan Silva, Scott Hauck, Shih-Chieh Hsu, Geoff Jones, "Analyzing the Efficiency of High-Level Synthesis for Machine Learning Inference Versus a Lower-Level Implementation", *Mary Gates Symposium on Undergraduate Research*, 2022.

Matthew Bavier, Caroline Johnson, Oleh Kondratyuk, Trinh Nguyen, Stephay Ayala-Cerna, Anatoliy Martnyuk, Aidan Short, Jan Silva, Scott Hauck, Shih-Chieh Hsu, Geoff Jones. "Quantifying the Efficiency of High-Level Synthesis for Machine Learning Inference", 2023.