Quantifying the Performance and Resource Usage of HLS4ML's Implementation of the Batch Normalization Layer on FPGAs

Waiz Khan

A thesis

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2024

Committee:

Scott Hauck

Shih-Chieh Hsu

Program Authorized to Offer Degree:

Department of Electrical and Computer Engineering

University of Washington

Abstract

Quantifying the Performance and Resource Usage of HLS4ML's Implementation of the Batch Normalization Layer on FPGAs

Waiz Khan

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

Field-Programmable Gate Arrays (FPGAs) are a powerful platform for developing hardware implementations of machine learning algorithms. Building these models is time-consuming and requires expertise in hardware design and writing code in Hardware Description Language (HDL). High-level synthesis (HLS) offers a method for developing hardware that does not require the specialized knowledge of FPGAs and HDL, but comes at the cost of not being able to modify the design to take advantage of the resources available. To evaluate models developed with HLS, we used the open-source Python library HLS4ML, which can produce low latency HLS machine learning models. In this thesis, we explore the application of high-level synthesis for machine learning, specifically the batch normalization layer, seeking to evaluate the quality, resource usage, and performance of the models produced using this technique. Our research indicates that HLS designs are efficient but not entirely accurate, whereas the optimized handwritten designs are very accurate, but require more resources.

Table of Contents

1. Introduction

In recent years, the applications of machine learning have expanded significantly, as well as the machine learning models themselves. These models have more trainable parameters, process larger data sets, and call for faster processing speeds and specialized hardware. For processing data that goes beyond the capabilities of the conventional CPU, a potentially desirable solution is to use Field Programmable Gate Arrays (FPGAs). These devices allow developers to design gate-level hardware that can take advantage of the FPGA's parallel processing capabilities as well as its ability to be reprogrammed with updated and optimized designs. The performance gains derived from a low-level design require knowledge of Hardware Description Language (HDL), which may not be in the skill set of the average software engineer. To bridge this gap, High-Level Synthesis (HLS) tools are employed to convert code written in high-level programming languages into the HDL code that is required to program an FPGA. These tools provide a simple method of developing HDL code, and decrease development times. To specifically convert machine learning models developed in Python to HDL, the High-Level Synthesis for Machine Learning (HLS4ML) [1] library provides the necessary tools. This is an open-source project that converts machine learning models into inference models that can be programmed onto an FPGA.

2. FPGA Background

FPGAs are semiconductor devices consisting of a matrix of configurable logic blocks connected using reprogrammable interconnects. These logic blocks consist of various smaller

components, usually including Look Up Tables, Registers, Multiplexers, and Carry Chains [2]. When a design is flashed to the FPGA, the interconnects are reprogrammed to link all the required resources to create the desired logic. Aside from their reprogrammability, another benefit of FPGAs is their ability to execute parallel processing tasks with low latency, which makes them an attractive option for machine learning applications. Running inference models on FPGAs allows for predictions to be made quickly from inputs, especially from live activities, such as controlling autonomous machines and vehicles. Machine learning models can become increasingly complex, and therefore the logic required to implement them would follow suit. To understand how efficient a design is, it must be evaluated in two aspects: resource utilization and performance. The FPGA targeted in this thesis is the Xilinx Virtex VC709.

2.1 Resource Utilization

Resource utilization measures the number of each type of resource in the FPGA that is used. This metric will be evaluated in raw quantity used, as well as a percent of the available resources on the board. The resources we will evaluate are:

- **Look Up Tables (LUTs):** LUTs are used to compute Boolean functions. The Virtex VC709 has 6-input LUTs, which means it takes six 1-bit inputs, and produces one 1-bit output. Each LUT can handle a total of 2^6 combinations of inputs and has a predetermined output stored per input pattern. For processing larger inputs, multiple LUTs are linked together.
- **Flip-Flops (FFs):** FFs are binary elements that store data between clock cycles. They are used to synchronize data transfer between different logic blocks and implement sequential logic.

- **Digital Signal Processors (DSPs):** DSPs are logic blocks used for mathematical computations including addition and multiplication. Each DSP in the Virtex VC709 contains a pre-adder, a 25 x 18-bit multiplier, and an accumulator.
- **Block RAM (BRAMs):** BRAMs are memory blocks used to hold large amounts of data. Each BRAM unit in the Virtex VC709 can hold 36 KB of data.

LUTs	FFs	DSPs	BRAMs
433,200	866,400	3,600	1,470

Table 2.1: Total Resources Available on the Virtex VC709 [3]

2.2 Performance

Along with resource utilization, the other aspect of a design is its performance, or the speed of the design in hardware. All performance metrics are measured at the fastest clock speed at which the design can run, and will be discussed in terms of the number of clock cycles as well as the actual time taken (in nanoseconds). The performance metrics we will evaluate are:

- **Latency:** This is the time interval between the first input being passed into the system and the system producing its first output. In other words, the time taken for one input to be processed by the system. This metric is essentially the response time of the system.
- **Initiation Interval (II):** This is the time interval between successive outputs from the system. In other words, after it produces one output, the time needed by the system to produce the next output. This metric is inversely proportional to the throughput of the system (data processed per unit of time).

3. Neural Network Background

Neural Networks are systems designed to generate outputs based on training data, and in essence, replicate a human brain that learns from experience. These neural networks are composed of smaller, interconnected units called neurons, which share information with other neurons when they "fire". In a neural network, a neuron will process and pass forward inputs based on the neuron's specific trained weights, biases, and activation function. Each input to a neuron has a specifically trained weight it is multiplied with, and each neuron has a bias value that is added to the sum of all the weights multiplied with the inputs. The activation function is used to map the result to a specified range and determine the output of the neuron. Neurons "fire" if the linear combination of the inputs multiplied by their respective weights exceeds the threshold of the neuron's activation function. During training, these weights and biases are adjusted using a method called backpropagation, which compares the results from the current state with what the result should be at the output, and adjusts the weights and biases in order to bring the two closer. The process is repeated multiple times to fine-tune these values. For more complex models, more layers are added to achieve a greater granularity in the results, as well as to allow the model to "learn" more features of the data.

3.1 Dense Layer

The Dense layer, also known as the "Fully Connected" layer, is made up of a bank of neurons such that every input to the layer is an input to every neuron in the layer [4]. To mathematically represent a dense layer with n inputs and j neurons, the following equation can be used:

$$y_j = \left(\sum_{i=0}^{n} (w_{i,j} * x_i) \right) + b_j \quad \textbf{(3.1)}$$

Where x_i is the ith input value, $w_{i,j}$ is the weight for input i in neuron j, b_j is the bias for neuron j, and y_j is the output of neuron j.

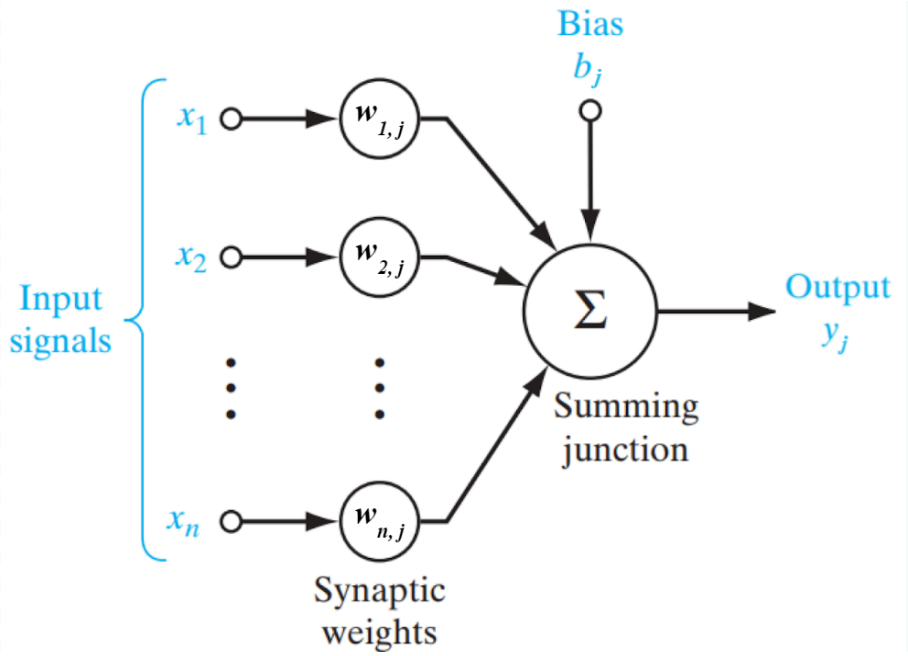

Figure 3.1: A Visual Representation for a Neural Network Computation [5]

This process can be described in matrix notation:

$$Y = XW + B \quad \textbf{(3.2)}$$

For example, let's take a layer that has 8 inputs, 4 outputs, and consists of 4 neurons. The Input vector X would be of size 1x8, the Weights matrix W would be of size 8x4, and the Bias vector B would be of size 1x4. This example would be as follows:

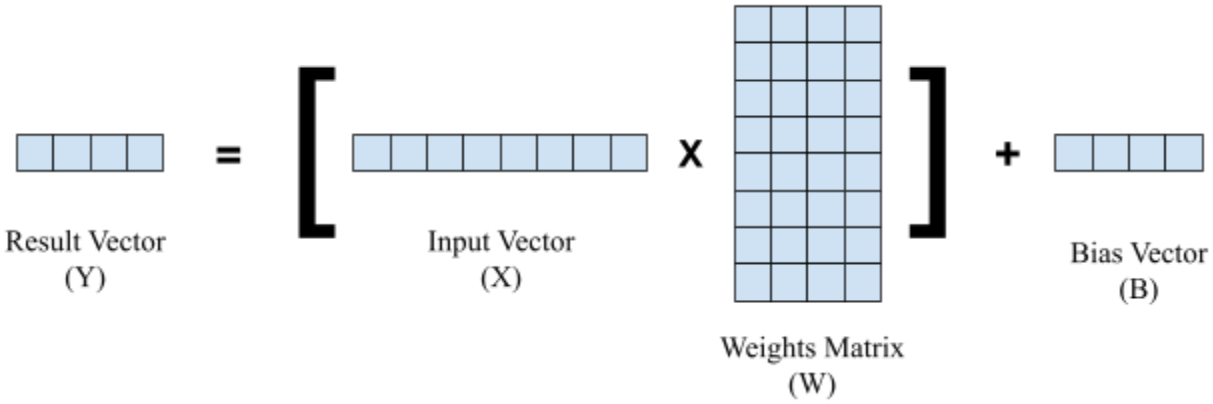

Figure 3.2: Example of Dense Layer Computation

Through a simple matrix multiplication and addition, we would get a result vector Y of size 1x4. This is a basic implementation of a fully connected layer.

3.2 Batch Normalization Layer

The Batch Normalization layer standardizes and normalizes the outputs of a layer in order to speed up training and reduce overfitting. Mathematically, batch normalization transforms the input values such that the resultant values have a mean of 0 and a standard deviation of 1. When a model is trained, the weights for intermediate layers (such as the dense layer) start at random values, and are updated as the model "learns" [6]. For some initializations, these values may cause some outputs to be abnormally small or large. This can result in instability in the training process and lead to the network not "learning" useful weights during the training period. Another important aspect of batch normalization is that it removes bias due to different input features being of different scales. For example, let's take a model with two input features: the age and yearly salary of a person. Generally, the salary of any given individual is many factors of 10 larger than their age. This can cause certain inputs, weights, and neurons to

have a larger impact on the overall output than others, and the model will take longer to train because of this skewed distribution.

The weights in Machine Learning models are updated during training using a process known as Gradient Descent [7]. Gradient Descent is an optimization algorithm that will find the values of a function's parameters that minimize the cost function. In machine learning applications, the cost function that is minimized is the loss. Loss is a measure of how inaccurate a prediction is for one input, and to develop an accurate model that provides accurate predictions, the loss function must be minimized. Gradient Descent works by evaluating the cost function of the parameter, determining its derivative, and adjusting the parameter such that the result of the cost function is moved closer to the local minima [7]. This process is repeated with each training run of a model, also known as an epoch. The rate at which the parameter is adjusted is known as the "learning rate", and it can be adjusted to improve training time. Without normalized inputs, training takes longer because the initial weights can start far from the local minimums on the cost function due to the inputs being of different scales.

There are two main methods for normalizing data. All training data can be normalized before it is passed into the model, and the model can train on the normalized data to begin with. This is a common practice and, in most cases, significantly improves training time. However, it is hard to determine how the weights are behaving inside the model during training. Every layer in the model is effectively an input layer: layer N's outputs are layer N+1's inputs. If the inputs to one specific layer change drastically, the model may run into the problem of unstable gradients [8]. To compensate for this, a batch normalization layer can be added.

During training, the technique works by taking batches of data, calculating their mean and standard deviation, standardizing each data point in the batch using those parameters, and

finally scaling and shifting the values to normalize them accurately. The Batch Normalization process [9] for one batch of *n* values during training is as follows:

$$\mu_B = \frac{\sum_{i=0}^{n} (h_i)}{n} \quad \textbf{(3.3)}$$

Step 1: Calculate the Batch Mean (μ_B) by summing *n* input values, and then divide by *n*

$$\sigma^2 = \frac{\sum_{i=0}^{n} (h_i - \mu_B)^2}{n} \quad \textbf{(3.4)}$$

Step 2: Calculate the Batch Variance (σ^2) by summing the square of the difference of each input less the batch mean, and then divide by *n*

$$\hat{x}_i = \frac{h_i - \mu_B}{\sqrt{\sigma^2 + \varepsilon}} \quad \textbf{(3.5)}$$

Step 3: Calculate the Normalized input value ($\hat{x}_i$) by subtracting the batch mean from the input value, and then dividing by the Batch Variance. Epsilon (ε) is a small constant value added to avoid a divide-by-zero error in the case the batch variance is 0.

$$\hat{y}_i = \gamma * \hat{x}_i + \beta \quad \textbf{(3.6)}$$

Step 4: Calculate the adjusted result value using gamma (γ) to scale the normalized input value, and beta (β) to shift. These parameters are learned values: γ is the weight, and β is the bias. Note how this equation is of the same structure as that of the dense layer.

However, the Batch normalization layer works differently in inference than it does in training [10]. During inference, the layer uses the final moving mean and variance that were

observed during training. The mathematical process for calculating the normalized value during inference is also slightly different, with fewer calculations. The batch mean and variance are not calculated, and the normalized input value equation uses the moving mean (μ_{mov}) and moving variance (σ^2_{mov}) from the training data. The moving mean is defined as:

$$\mu_{mov} = (\mu_{mov} * momentum) + (\mu_B * (1 - momentum)) \quad \textbf{(3.7)}$$

Where μ_B is the batch mean, and the *momentum* is a value in the range (0, 1) which controls how quickly the moving mean adjusts to the current batch mean. The moving variance is defined in the same way:

$$\sigma^2_{mov} = (\sigma^2_{mov} * momentum) + (\sigma^2 * (1 - momentum)) \quad \textbf{(3.8)}$$

Where σ^2 is the batch variance, and the *momentum* is a value in the range (0, 1) which controls how quickly the moving variance adjusts to the current batch variance.

Combining equations 3.5 and 3.6 results in one linear operation, which gives us the following equation, where h_i is the input, and all other values on the right-hand side are constants:

$$\hat{y}_i = \gamma \left(\frac{h_i - \mu_{mov}}{\sqrt{\sigma^2_{mov} + \varepsilon}} \right) + \beta \quad \textbf{(3.9)}$$

This is the complete equation used for the inference model for the batch normalization layer.

4. HLS4ML Background

HLS4ML (High-Level Synthesis for Machine Learning) is a tool used to convert machine learning models written in Python into HDL code that can be uploaded to an FPGA. The core part of HLS4ML is HLS (High-Level Synthesis), which can take programs written in high-level programming languages, such as C++, and convert them into HDL that can be used to program an FPGA. The HLS4ML library [1] takes this conversion one step further, and allows the process to start from Python. It converts Python machine learning models into C++ code written for HLS. This is then fed into Vivado HLS, which in turn outputs the HDL.

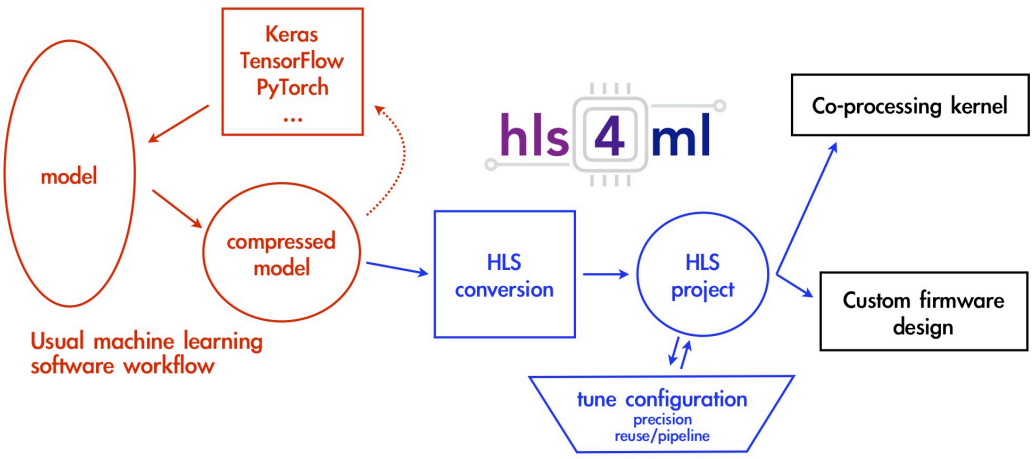

Figure 4.1: A Visual Representation of the HLS4ML Flow [1]

The process starts with a Python machine learning model developed using TensorFlow or Pytorch [1]. The user can specify all the layers they want in the model, and then train the model on their desired data. For this process, we are examining inference models, so the models are already trained and their weights and biases were already tuned. The next step is to convert the trained model into an HLS model, which is done with the HLS4ML library. The conversion from Python to HDL begins with the configurations of the build being set using the config dictionary.

With this, the reuse factor and bitwidth of the model can be set [11]. The last step is to call the hls_model.build() command, which starts the complete conversion process. The tool will produce three sets of files: the intermediate C++ files, the output files in Verilog, as well as in VHDL, along with the log files, the config files, and the weights and biases. HLS4ML also produces a resource usage and performance summary, but we have found that these results produced during the C synthesis stage are not always accurate or representative of the true resource utilization of the model. To thoroughly evaluate the model, the HDL is taken and a Vivado project is created with it. This allows us to specify the target device and clock speed, and run synthesis and implementation to produce a full set of reports. The synthesized design report is used to evaluate resource usage, and the timing report produced is used to evaluate performance.

5. Developing the Batch Normalization Layer

The batch normalization layer in inference takes the outputs of a previous layer and performs the batch normalization operation (Equation 5) on each input, using the learned parameters that were set during training. The two main learned parameters are gamma (γ) which is the scaling (weight) parameter, and beta (β), which is the shifting (bias) parameter. In addition to these two, the moving mean (μ_{mov}) and moving variance (σ^2_{mov}) from the training data are also needed for the inference model. For each batch normalization layer, these values will be constants. This leads to an interesting development: all of the values on the right-hand side of Equation 5 are constants except the input. The most complex part of the equation is the

inverse square root, which is an expensive computation. To alleviate the system of having to perform this computation, the moving variance is used as the input to a table that contains all possible values of the inverse square root operation for all possible values of the moving variance at the specified bitwidth. The output of the table is simply multiplied by the $(h_i - \mu_B)$ term. The resultant value is then scaled and shifted using gamma and beta. For this part, the default parameters were used to simplify testing. By default, gamma is set to 1, so the value is not scaled, and beta is set to 0, so the value is not shifted.

5.1 Fixed Point Numbers

Commonly in hardware designs, decimal numbers used for computation are represented in binary in one of two ways: floating-point, or fixed-point. Floating-point numbers offer greater precision than fixed-point numbers of the same bitwidth because the position of the decimal point can vary based on the number being represented, but comes at the cost of complexity and more resource usage. Fixed-point binary numbers have a set position of the decimal point, any bits below this point represent fractional bits, and any bits above the decimal point represent integer bits. Fixed-point numbers with higher bitwidths are more precise, but have a larger hardware cost. On the other hand, fixed-point numbers with smaller bitwidths are less precise, but have reduced hardware costs. For this thesis, the objective was to scale the bitwidth of the inputs to the batch normalization layer and examine how the resource usage and performance scales.

A perk of fixed point numbers is that basic arithmetic with these numbers works exactly the same as it does with the basic binary representation of whole numbers. This means that all

fixed-point numbers in this layer are also 2's complement numbers. To simplify the layer, the bitwidth is split 50/50 between integer and fractional bits. For example, an 8-bit fixed point number will have 4 fractional bits, and 4 integer bits, with the most significant integer bit being the 2's complement sign bit. For a fixed-point number, the decimal value is evaluated by evaluating the integer bits with positive powers of 2 starting from 0 and increasing towards the left, and evaluating the fractional bits with negative powers of 2 starting from -1 and decreasing towards the right. For example, the fixed point number 0110 1100 would evaluate to 6.75.

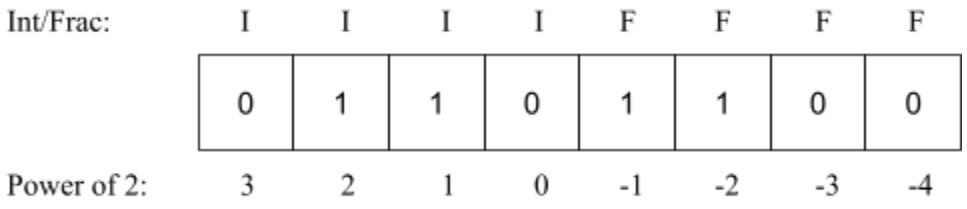

Figure 5.1: A Visual Representation of a Fixed Point Number. The Number 0110 1100 is Evaluated as: $2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

All of the mathematical computations done in the hand-written version of the batch normalization layer were done in fixed point. To avert the inverse square computation, a C program was developed to generate tables for complex computations.

5.2 Table Generator

The table generator program evaluates an equation for all possible fixed-point inputs for a specific bitwidth using floating-point numbers, and then converts the results back to fixed-point. In this case, the equation was:

$$\frac{1}{\sqrt{\sigma_{mov}^2 + \varepsilon}} \tag{5.1}$$

The moving variance is a constant value input to the batch norm layer, and epsilon (ε) is a small stabilizing value (0.00001001) used to avoid dividing by 0. The table generator takes a bitwidth *n*, and evaluates the equation for all 2^n -1 possible fixed-point binary values for the given bitwidth. The fixed-point value is first converted to a floating-point value, the equation is evaluated, and the result is converted to a fixed-point binary number. All results are formatted in hexadecimal and sequentially written to a .dat file. The conversions have the potential to lose precision due to the conversions between fixed- and floating-point formats. In order to evaluate the effectiveness of this table generation program, the percent error for every fixed-point result was calculated. The results indicate that as bitwidths increase, the average error shows an exponential decay. At low bitwidths, the error is high. This is compounded by the fact that fixed-point values of small bitwidths already have limited precision. At high bitwidths, the average error is close to zero. The results are shown in the graph below.

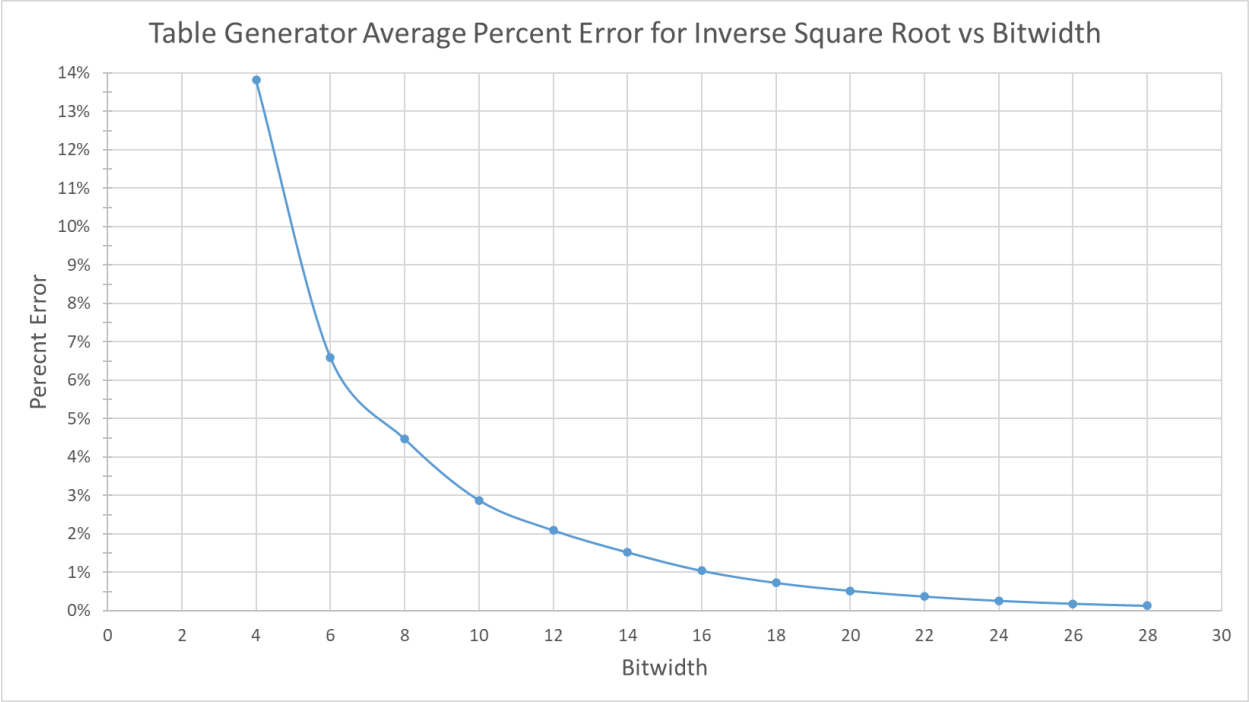

Figure 5.2: Table Generator Average Percent Error

5.3 Handwritten Verilog Model

The Verilog model of the batch normalization layer is parameterized to take in different bitwidths, as well as a different number of inputs. For this thesis, the number of inputs was kept constant at 16, while the bitwidth was varied. The layer takes 16 n-bit inputs, and produces 16 n-bit outputs. The layer also takes the beta, gamma, moving mean, and moving variance as parameters. The model also reads in the appropriate bitwidth inverse square root table's data file, which is used to select the respective inverse square root multiplier based on the moving variance. The computation is performed in two steps, each step consisting of one multiply and one add operation. The computations are as follows:

$$S1[i] = (h_i - \mu_B) * (InvSqrt\ Table\ Value) \quad \textbf{(5.2)}$$

$$Out[i] = (\gamma * S1[i]) + \beta \quad \textbf{(5.3)}$$

Equation 7.1 is a modified implementation of Equation 3.3, where the $\frac{1}{\sqrt{\sigma_{mov}^2 + \varepsilon}}$ term comes from the inverse square root table, and works like a constant multiplier. The next step takes the normalized value and scales and shifts it according to the learned parameters of the layer.

5.4 HLS4ML Verilog Model

The batch normalization layer generated by HLS4ML was created using a simple model that contains one input layer, containing 16 inputs, followed by a batch normalization layer. The model shown below was developed for a bitwidth of 24.

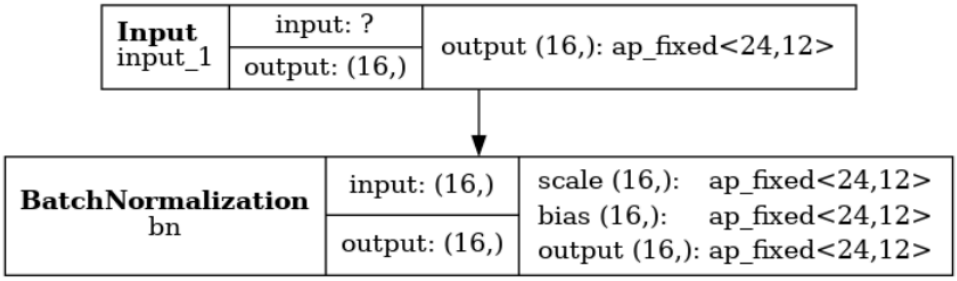

Figure 5.3: HLS4ML Model Plot for a Batch Normalization Layer

This model was generated with bitwidths varying from 4 bits to 22 bits, and was passed through the HLS flow described in part 5. The model produces two main Verilog files: a top-level module, and a normalization module.

6. Results

6.1 Initial Results

Once the handwritten and HLS4ML-generated models were created, they were evaluated for resource usage and performance. In the following graphs, the dashed lines are the handwritten design, and the solid lines are the HLS4ML designs.

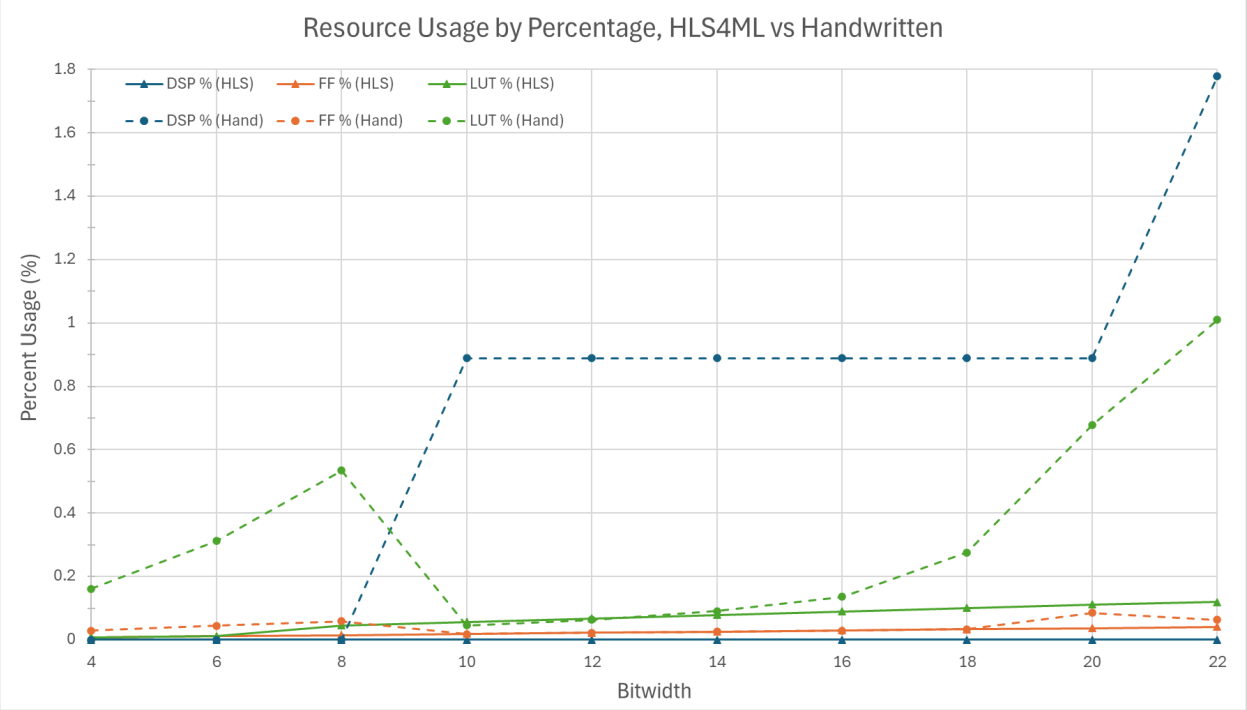

Figure 6.1: Resource Utilization Comparison Between the HLS4ML and Handwritten Layers

The results above show resource usage as the percent of the total available of each type of resource. Starting with the handwritten model, it can be seen that at small bitwidths (4, 6, and 8 bits), the computations are done without any DSPs, and the layer relies entirely on LUTs and FFs to process the logical operations. At a bitwidth of 10 bits, the model starts to use DSPs. This is

correlated to the drop in LUTs and FFs used, due to the multiplications being computed with the DSPs. From 12 bits up to 20 bits, the results show the number of DSPs used remaining constant, but the LUT and FF usage increases as the bitwidth increases. At a bitwidth of 22, it can be seen that the DSP usage doubles, and the FF usage drops slightly. However, the LUT usage continues to increase. This can likely be attributed to the fact that at larger bitwidths, the inverse square root table grows exponentially, so the number of LUTs required to store all the values increases.

Moving onto the HLS4ML-generated model, it can be seen that this model overall uses significantly fewer resources than the handwritten model. The model does not use any DSPs for processing, and only relies on the LUTs and FFs. This was an interesting result, because even though batch normalization is effectively a linear operation, HLS4ML seems to be implementing it with very few resources. The Flip-Flop usage was relatively similar between the two models, but the LUT and DSP usage is significantly different at certain bitwidths. Next, the performance was evaluated. The results are shown below.

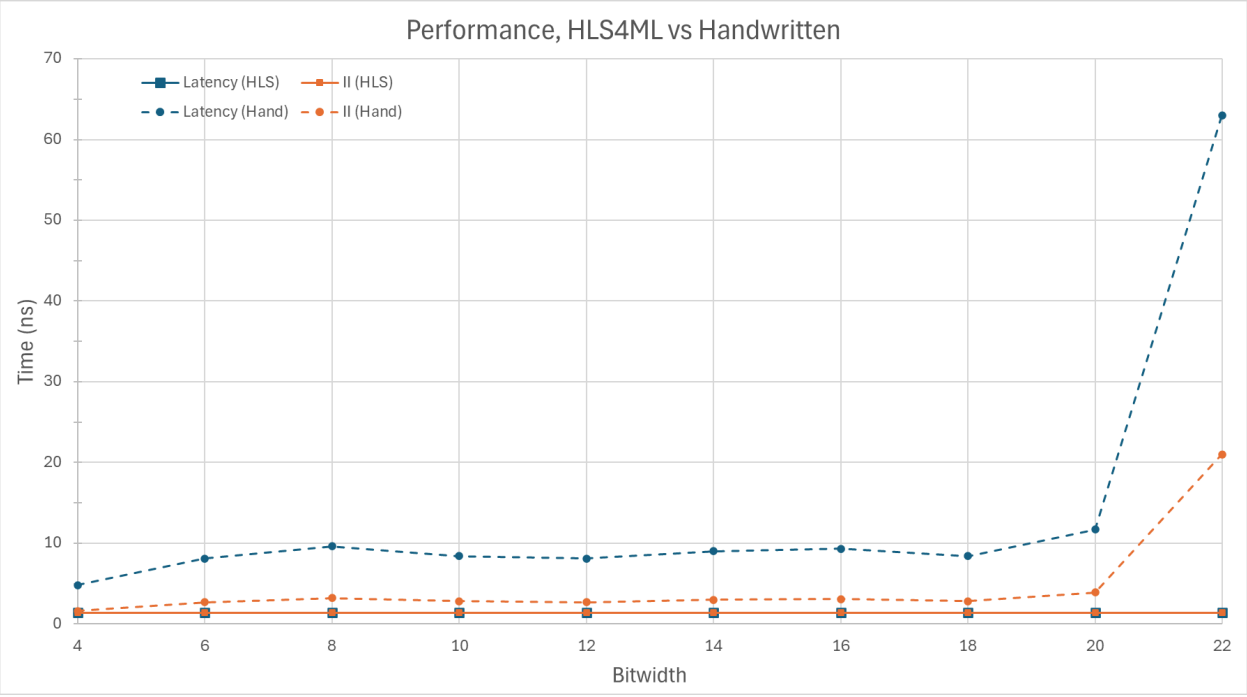

Figure 6.2: Performance comparison between the HLS4ML and handwritten layers

The handwritten version had 3 clock cycles of latency, and had an initiation interval of 1 clock cycle regardless of bitwidth. Changing the bitwidth did impact the fastest clock speed the model could run at. The handwritten model had an average clock period of around 2.9 ns, an average latency of 8.6 ns, and an average initiation interval of 2.9 ns, excluding the 22-bit test. At 22 bits, the size of the model increases significantly and the jump in clock period is potentially linked to the doubling of DSPs used at a bitwidth of 22.

The HLS4ML version had a consistent 1 clock cycle of latency, as well as an initiation interval of 1 clock cycle, regardless of bitwidth. Changing the bitwidth had no impact on the fastest clock speed the model could run at, and had a smallest clock period of 1.41 ns. This results in the latency and initiation interval always being 1.41 ns. HLS4ML's approximation implementation of the batch normalization algorithm not only saves resources, but also improves performance. The algorithm does not require any heavy computations, which allows the clock period to be small. The HLS4ML version achieves 2x the performance of the handwritten version, and can run at nearly double the clock frequency.

To understand how the HLS4ML implementation was able to achieve such low resource usage, the HLS4ML implementation was thoroughly examined. First, both versions were tested using a testbench and given the same input, a series of numbers [0, 7]. The handwritten version was given a moving mean of 3.5, and a moving variance value of 5.25. These values were chosen since they are the mean and variance of the input set of data. Mathematically, the handwritten batch normalization layer computes the correct values for each input. This can be seen in the graph below: the output values are distributed with a mean of 0, and a standard

deviation of 1. The HLS4ML version produces different results, and with a similar distribution but in a very different range. Inputs are represented in fixed-point hexadecimal, and output values are represented in fixed-point converted to decimal.

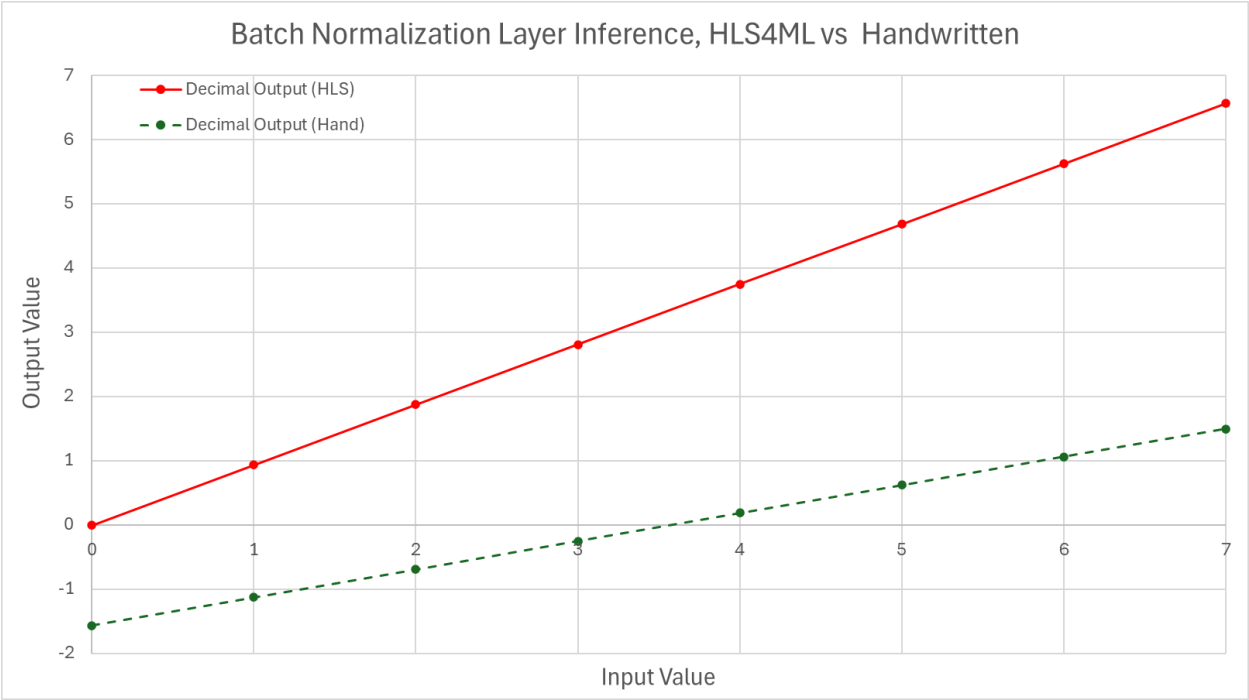

Figure 6.3: Inference Outputs of the HLS4ML and Handwritten Versions

Upon inspecting the HLS4ML code itself, it was found that the process the HLS4ML implementation uses is significantly different from the standard batch normalization algorithm. The HLS4ML algorithm does the following, with X being the input value:

$$Output = ((X << (WIDTH/2)) - X) >> (WIDTH/2) \quad \textbf{(6.1)}$$

This mathematical shortcut provides an approximation for the normalization results. The algorithm used here simply scales all input values such that they become closer to 0, rather than actually performing the batch normalization process. The HLS4ML implementation takes the input value, shifts it left by half of the bitwidth, and then subtracts the original input value from

this shifted value. It is then right-shifted back by the same amount in order to fit in the same original bitwidth. The HLS4ML implementation requires nearly 25% of the resources used by the handwritten implementation, but it comes at the cost of accuracy, and the results produced can show upwards of 333% error.

The initial results showed a large difference in resource usage and performance, and with the handwritten design using more resources and computing slower than the HLS design, we looked to optimize the handwritten design.

6.2 Optimized Design

The first, and biggest cause of ballooning resource usage in the handwritten design is the storing of the inverse square root table. The solution to this is to not store the table at all, and to take the value that the table produces as a parameter. This parameter effectively replaces the moving variance parameter. This serves to reduce the quantity of LUTs used in the design. The next step taken was to simplify the computation the design performs. Going back to equation 3.9, it was found that all of the values on the right-hand side of the equation were constants, except for the input value (h_i). This means the equation can be simplified into the linear slope-intercept form. In this form, the resulting equation is:

$$\widehat{y}_i = m * x + b \tag{6.2}$$

Where the input x is the input data value h_i, and the two constants, m and b, are the following:

$$m = \frac{\gamma}{\sqrt{\sigma_{mov}^2 + \varepsilon}} \tag{6.3}$$

$$b = -\left(\frac{\gamma * \mu_{mov}}{\sqrt{\sigma_{mov}^2 + \varepsilon}} + \beta\right) \quad \textbf{(6.4)}$$

When the layer is initialized, the two constant values are computed once, then the batch normalization computation is performed on the input data. Since the revised computation only requires one multiply and one add, it only takes one cycle, which should improve the latency. The optimized version is bit-accurate to the original, and produces the same outputs.

6.3 Optimized Results

This new optimized model was evaluated for resource usage and performance. In the following graphs, the solid lines are the initial handwritten design, and the dashed lines are the optimized handwritten designs (marked with (Opt) in the legend).

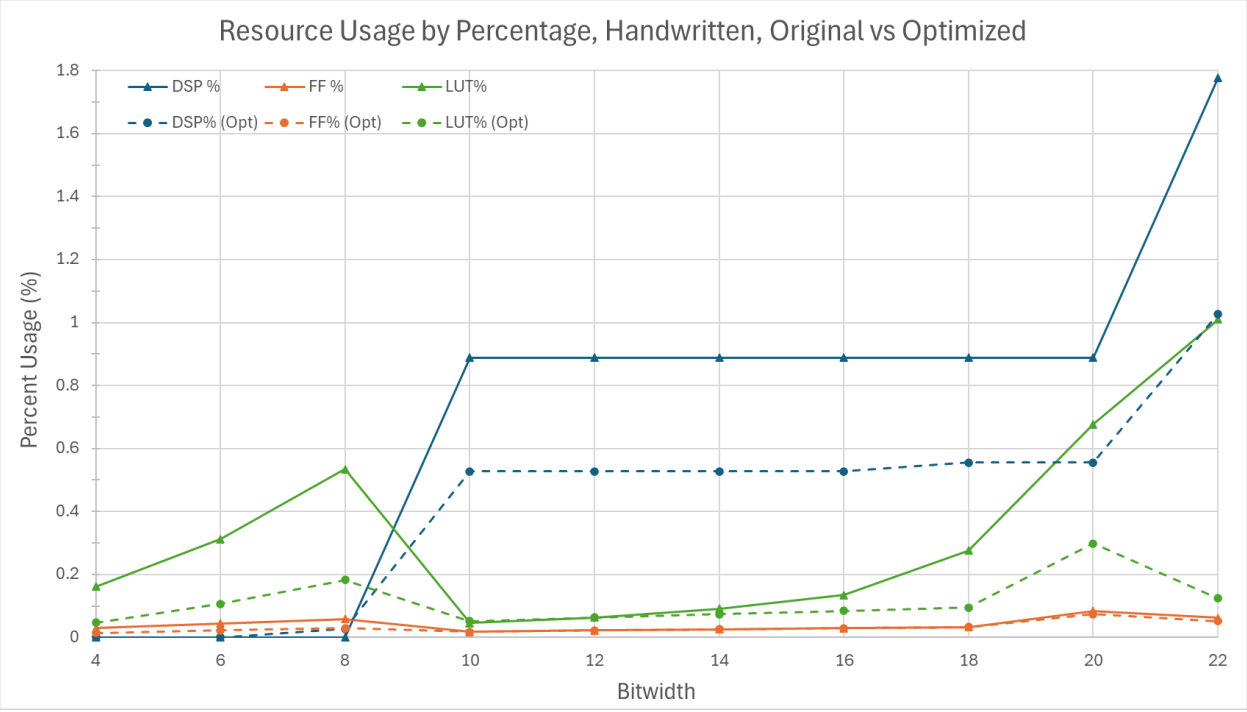

Figure 6.4: Resource Utilization Comparison Between the Initial and Optimized Implementation

The results show an overall decrease in total resource usage in the optimized handwritten layer in comparison to the original implementation. The DSP usage in the optimized model is similar for smaller bitwidths (4 to 8 bits), and for all other bitwidths, the optimized model uses close to 40% fewer DSPs. The FF usage is nearly identical in both implementations. Reducing the computation and eliminating storing the inverse square root tables reduces the LUT usage for most of the tested bitwidths in the optimized implementation, up to 90% at the highest bitwidth of 22. Next, let's evaluate the performance of the optimized design.

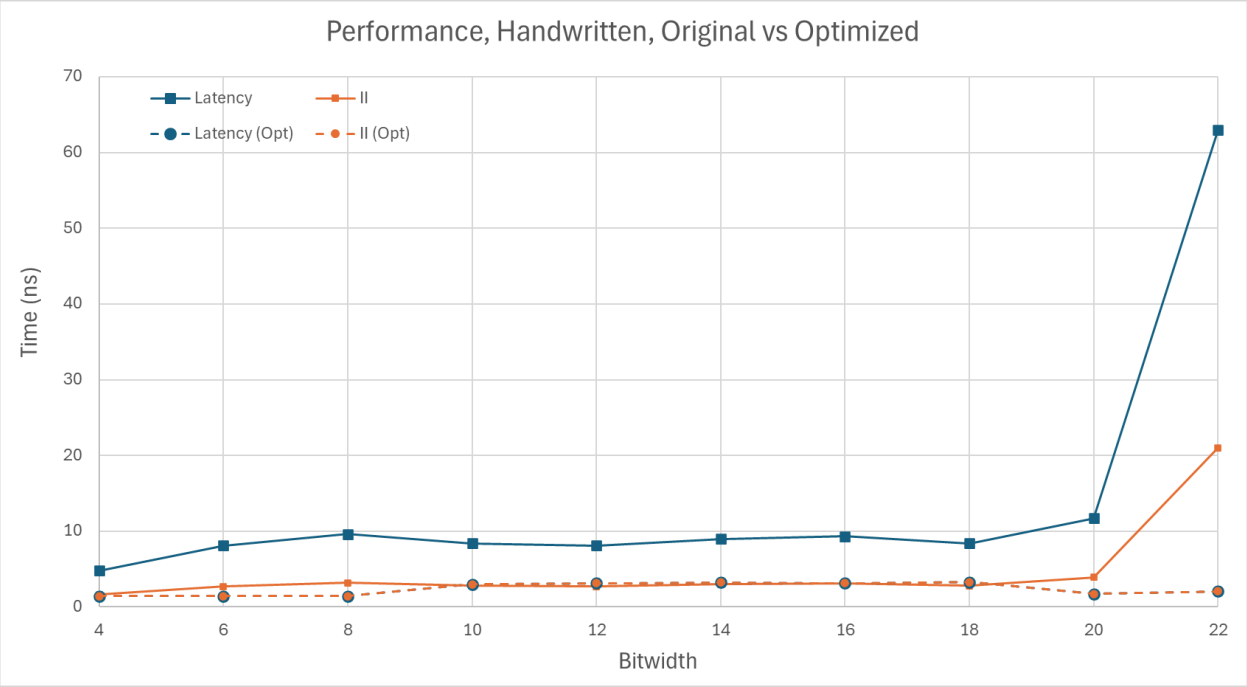

Figure 6.5: Performance Comparison Between the Initial and Optimized Implementations

The optimized implementation had a latency of one clock cycle, as well as an initiation interval of one clock cycle, whereas the original implementation had a latency of three clock cycles, and an initiation interval of one clock cycle. The improvements also resulted in the optimized design being able to be run at a faster clock speed. The II remained similar between the two, but the latency was close to 3x lower in the optimized design at most bitwidths.

Having shown that the optimized version uses fewer resources and has better performance, we will now compare the optimized handwritten design to the HLS4ML design. In the following graphs, the solid lines are the HLS4ML design, and the dashed lines are the optimized handwritten designs (marked with (Hand, Opt) in the legend).

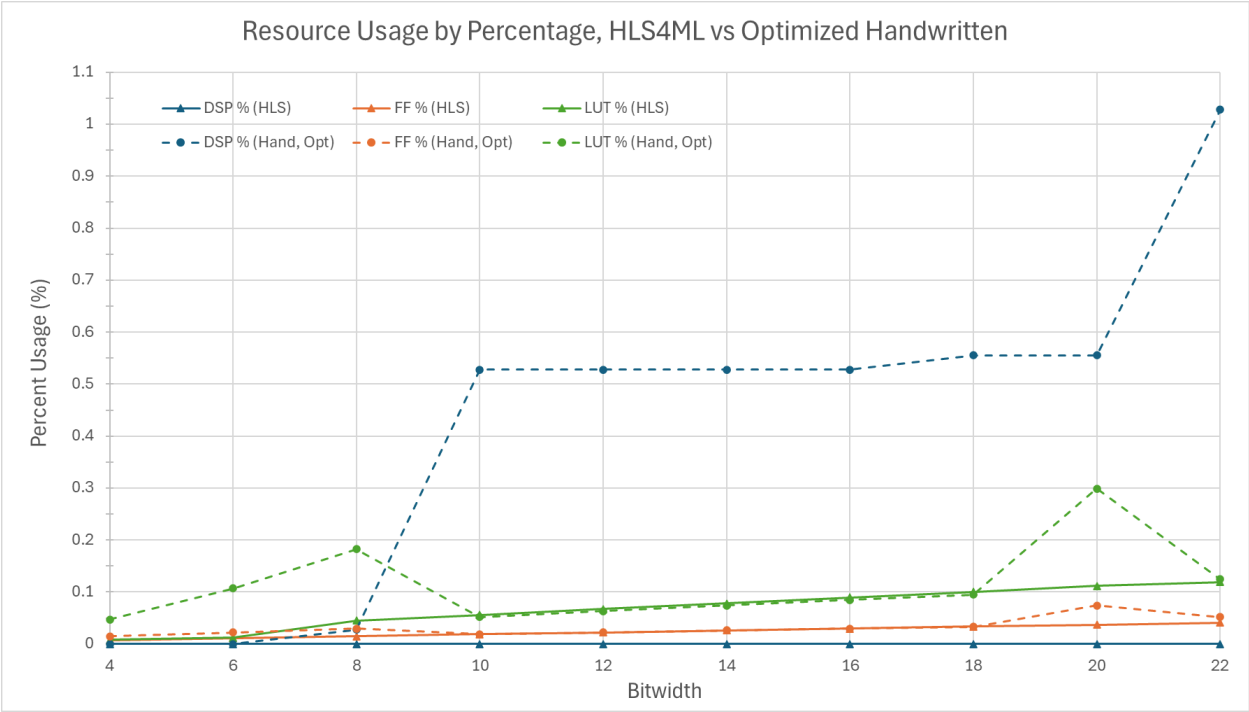

Figure 6.6: Resource Utilization Comparison Between the HLS and Optimized Implementations

The results above indicate a more comparable design in terms of overall resource usage. The FF and LUT usage between the two designs is nearly identical, with the exception of a small spike in usage at a bitwidth of 20. Unlike the original, the LUT usage does not exponentially increase with the bitwidths. Similar to the original, with the handwritten model, it can be seen that at small bitwidths (4 to 8 bits), the computations are done without any DSPs, using only LUTs and FFs, which explains the early spike. At a bitwidth of 10 bits, the handwritten model starts to use DSPs, which appears to cause a decrease in the LUT and FF usage.

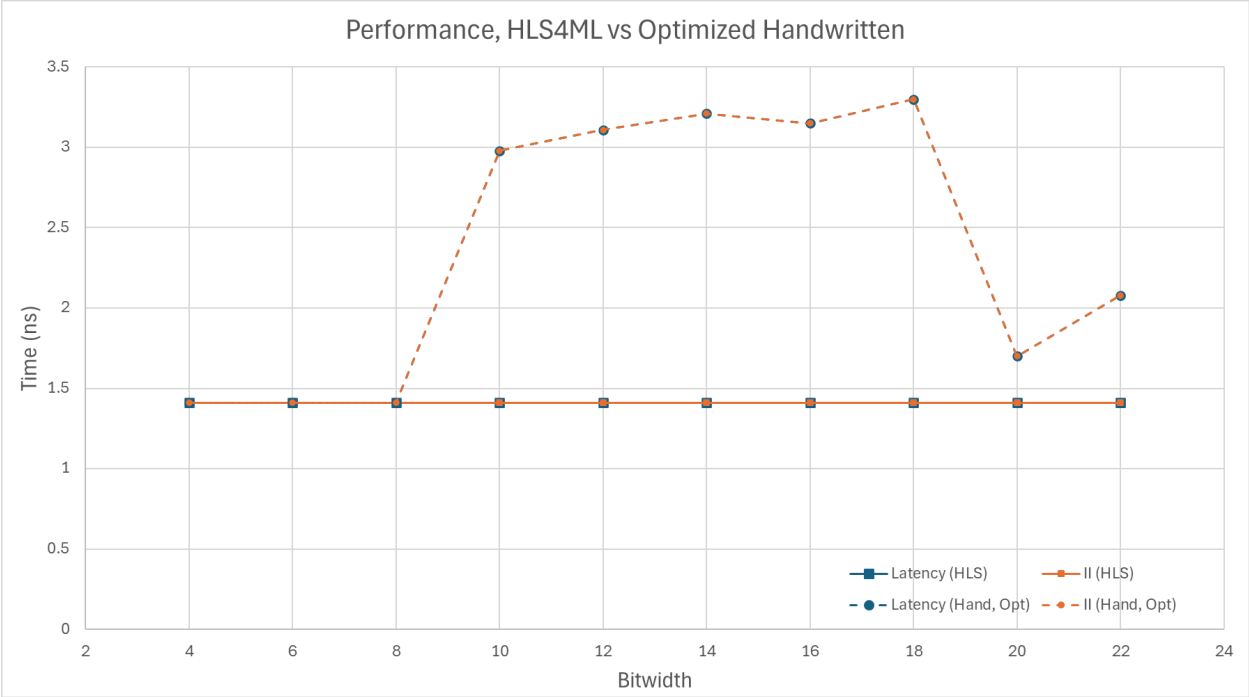

Figure 6.7: Performance Comparison Between the HLS and Optimized Implementations

The performance results also show a significant improvement in the latency of the optimized design as compared to the original. At small bitwidths (4 to 8 bits), the handwritten model matches the performance of the HLS4ML model, with the latency and II only increasing when DSPs are introduced. From bitwidths 10 to 18, the latency and II have the general trend of a slight increase. At a bitwidth of 20, the latency drops, which is potentially linked to the increase in FFs and LUTs used at this bitwidth. At a bitwidth of 22, the latency increases again, likely due to an increase in DSP usage. The improvement also is due to both designs now having a latency and II of one clock cycle. The only difference between the two is the lowest clock period the designs can run at. The HLS4ML design's smallest clock period was not dependent on the bitwidth due to it not using any DSPs, whereas the handwritten design's smallest clock period varied based on the bitwidth and resources used.

7. Conclusion

HLS tools can quickly and efficiently convert machine learning models to run on FPGAs, but at times they come at the expense of accuracy. In this thesis, it was shown that although the HLS design for the batch normalization layer was implemented with fewer resources, it was not accurate. The optimized handwritten design was implemented using slightly more resources, but produces significantly more accurate results with minimal error. HLS4ML shows promising results, and if it is implemented with the correct algorithm, it should be able to produce mathematically correct results. For designs where resources are limited and some inaccuracies can be tolerated, HLS tools can get the job done. However, for designs where achieving accuracy and precision is paramount, it is best to stick to handwritten designs.

8. Future Work

The next step to further evaluate the Batch Normalization layer is to test the layer with a complete benchmark. This will give more insights into how the layer performs when connected to other layers, as well as how accurate the final results of a model are. Further exploration should be conducted to explore the implementation of the 2-dimensional batch normalization layer to see if there are more layers where HLS4ML implements an approximation of the standard algorithm to reduce resource usage. Additionally, HLS4ML should consider developing a version of the batch normalization layer that matches the mathematical implementation used by QKeras to match the accuracy that can be achieved with a handwritten model.

Acknowledgements

First, I would like to thank my advisors, Scott Hauck and Shih-Chieh Hsu for their support over the last year. I am grateful for their guidance and counsel, which has been vital for this project. I would also like to thank Professors Rania Hussein and Sep Makhsous for sparking my interest in Hardware Design and Embedded Systems.

I would also like to thank the members of the ACME lab and HLS4ML community. A big thank you to Caroline Johnson, who took me in, showed me the ropes, and was always willing to help me, even after she graduated. Additionally, thank you to Geoff Jones, Gayathri Vadhyan, Sushree Jena, Matthew Bavier, Oleh Kondratyuk, Trinh Nguyen, Stephany Ayala-Cerna, Aidan Short, Jan Silva, Anatoliy Martynyuk for their work on previous and ongoing layers and benchmarks. This material is based upon work supported by the National Science Foundation under Grant No. PHY-2117997.

I would like to thank my parents, my brother, and my friends and family for their support throughout my time at the University of Washington. A special thank you goes out to Nusaiba Ahmed, Nathan Ford, Cory Lam, Jason Isa, Brandon Ray, Aryan Singh, and Ishan Deva for supporting me throughout my academic endeavors.

Bibliography

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

[2] AMD. "What Is an Fpga? Field Programmable Gate Array." https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[3] Xilinx, Inc. "7 Series DSP48E1 Slice. User Guide", 2018. https://docs.xilinx.com/v/u/enUS/ug479_7Series_DSP48E1

[4] Saxena, Akarsh. "Building a Simple Neural Network from Scratch." Towards Data Science, May 31, 2020. https://towardsdatascience.com/building-a-simple-neural-network-from-scratch-a5c6b2eb0c34.

[5] midflip. "How Artificial Intelligence Works." https://midflip.io/topics/how-artificial-intelligence-works.

[6] Brownlee, Jason. "A Gentle Introduction to Batch Normalization for Deep Neural Networks." Machine Learning Mastery (blog), Dec. 4, 2019. https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/.

[7] Brownlee, Jason. "Gradient Descent For Machine Learning." Machine Learning Mastery (blog), Aug. 12, 2019. https://machinelearningmastery.com/gradient-descent-for-machine-learning/.

[8] C, Bala Priya. "Build Better Deep Learning Models with Batch and Layer Normalization | Pinecone." Accessed March 13, 2024. https://www.pinecone.io/learn/batch-layer-normalization/.

[9] Saxena, Shipra. "Introduction to Batch Normalization." Analytics Vidhya (blog), Mar. 9, 2021. https://www.analyticsvidhya.com/blog/2021/03/introduction-to-batch-normalization/.

[10] Keras. "Keras Documentation: BatchNormalization Layer." https://keras.io/api/layers/normalization_layers/batch_normalization/.

[11] "Configuration — Hls4ml 0.8.1 Documentation." https://fastmachinelearning.org/hls4ml/api/configuration.html

Appendix

Hand Written Code

All code for the handwritten layers can be found at: https://github.com/uw-acme/

HLS4ML_VS_MANUAL/tree/main/src.

Related Works

Caroline Johnson, Evaluating the Quality of HLS4ML's Basic Neural Network Implementations on FPGAs, 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, "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.