Benchmarking HLS4ML Designs with Batch Normalization

Donovan Clay

An honors thesis submitted in partial fulfillment of the
departmental honors requirements for the degree of

Bachelor of Science

University of Washington

2025

Reading Committee:

Scott Hauck

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Benchmarking HLS4ML Designs with Batch Normalization

Donovan Clay

This thesis explores the implementation and optimization of neural network models on Field-Programmable Gate Arrays (FPGAs) using the HLS4ML library, addressing the increasing energy consumption and inference time of complex deep learning models. The core objective is to benchmark HLS4ML's performance against handmade designs to identify and explore opportunities for optimization.

The methodology involved novel hardware implementations for key neural network layers. Specifically, the Batch Normalization layer and Softmax operation. The benchmark model, a jet tagging neural network designed for high-energy physics experiments at the Large Hadron Collider, includes a Batch Normalization layer as a distinguishing feature from other benchmarks of HLS4ML. All designs were evaluated on a Xilinx Alveo U250 board.

Results indicate that the HLS4ML-generated design generally has better resource utilization. In terms of timing, the handmade design achieved a faster clock frequency but exhibited higher total latency, although both designs achieved an Initiation Interval (II) of 1. A per-module analysis revealed that dense layers were the primary consumers of FFs and DSPs, while ReLU layers significantly contributed to LUT utilization.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grant No. OAC-2117997.

I acknowledge the Fast Machine Learning collective as an open community of multi-domain experts and collaborators. This community and Scott Hauck, in particular, were important for the development of this project.

DEDICATION

1 Introduction

The power of neural networks has been demonstrated across an increasingly diverse range of applications, revolutionizing fields from computer vision to natural language processing [Rostam et al., 2024; Russakovsky et al., 2015]. As these neural network models continue to grow in complexity and size, however, the associated computational demands lead to significant increases in energy consumption and inference time. Implementing machine learning models directly in specialized hardware, such as Field-Programmable Gate Arrays (FPGAs), offers a compelling solution to these challenges by leveraging their inherent parallelism and energy efficiency. The HLS4ML library automates the crucial, yet often complex, process of converting high-level machine learning Python code into optimized hardware designs (bitfiles) for FPGA programming, thereby simplifying a task that would otherwise require extensive hardware design expertise [FastML Team, 2025; Duarte et al., 2018]. To further unlock the potential of FPGA-accelerated machine learning, a comprehensive benchmarking of HLS4ML's performance is crucial for identifying bottlenecks and devising strategies to optimize both the library and the generated hardware, ultimately pushing the boundaries of real-time machine learning inference.

2 Background

2.1 How Can Neural Networks Be Implemented on FPGAs?

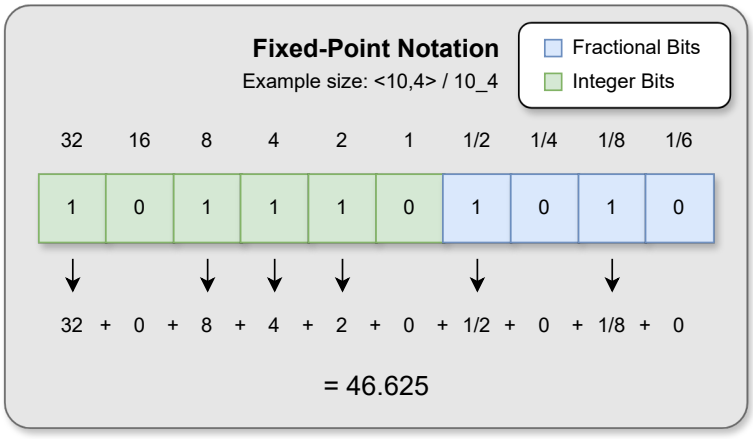

Figure 1: Fixed-Point Notation

Fixed-point Notation Fixed-point notation is a method for representing real numbers using a fixed number of digits after the radix point (decimal point). Unlike floating-point notation, where the radix point can "float," its position is predetermined and consistent. This makes fixed-point arithmetic simpler and often more efficient to implement in hardware, such as FPGAs, as it avoids the complex logic required for floating-point operations.

As illustrated in Figure 1, a fixed-point number is typically represented by a total number of bits, which are divided into two parts: integer bits (to the left of the implied radix point) and fractional bits (to the right). The integer bits represent the whole number part, while the fractional bits represent the precision or the decimal part of the number. For this paper, a <T, F> or T_F notation signifies T total bits and F fractional bits. This fixed structure provides a predictable range and precision, which is crucial for resource-constrained environments like FPGAs where exact control over bit-width and computational complexity is desired for optimal performance and resource utilization in applications like neural networks.

2.2 The Benchmark Neural Network

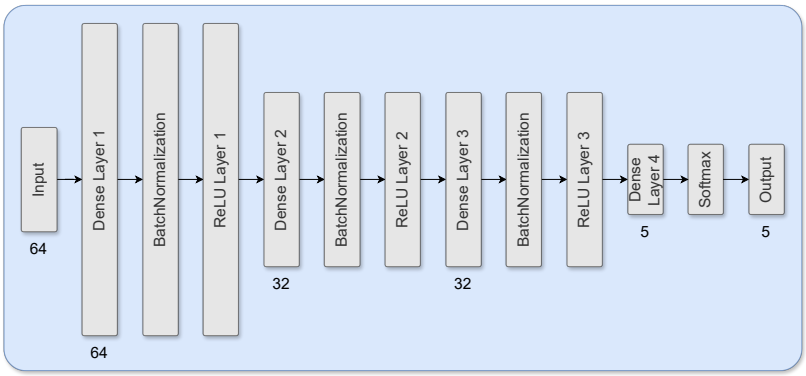

Figure 2: Jet Tagging Benchmark Neural Network

One benchmark neural network model is used for this thesis to evaluate the effectiveness of HLS4ML. This model is designed for classifying jets of particles during high-energy physics experiments at the Large Hadron Collider [Ngadiuba et al., 2020]. The project specifically utilizes the `hls4ml_lhc_jets_hlf` dataset, which comprises high-level features extracted from simulated jet events. Such models used designed for real-time scientific experiments are a common choice for

benchmarking HLS4ML applications due to the stringent demands for ultra-low latency and high throughput inherent in experimental data processing.

The architecture of the benchmark model, as depicted in Figure 2, consists of a series of dense layers interspersed with activation functions. Previous works by Johnson [2023] and Khan [2024] have explored various neural network architectures for similar benchmarking purposes, demonstrating the adaptability of HLS4ML to different model complexities.

A key distinguishing feature of the benchmark model used in this thesis, compared to some earlier HLS4ML benchmarks, is the inclusion of a Batch Normalization (BN) layer. Batch Normalization layers, typically applied during training to stabilize and accelerate the learning process, present unique challenges and opportunities for hardware implementation via HLS. The exploration of this specific architecture allows for a more comprehensive evaluation of HLS4ML's capability to handle contemporary neural network design patterns that incorporate such layers.

2.3 Metrics

In this thesis, I will compare HLS4ML's performance against the performance of "handmade" Verilog designs. Metrics are crucial for comparing the efficiency and effectiveness of different design choices, particularly concerning resource utilization and timing performance.

2.3.1 Resource Utilization

Field-Programmable Gate Arrays (FPGAs) are reconfigurable integrated circuits that offer a flexible platform for implementing custom digital logic. To achieve this flexibility, FPGAs are populated with various types of programmable resources, each serving a specific function. Understanding these resources is crucial for optimizing designs, especially when deploying complex algorithms like those found in machine learning.

Flip-Flops Flip-flops (FFs) are a basic building block of digital circuits in FPGAs. They are used to store a single bit of data and update their state synchronously with a clock signal. FFs are essential for implementing state machines, registers, and any circuit requiring memory elements.

Look-Up Tables Look-Up tables (LUTs) are another fundamental building block in FPGAs. LUTs are used to implement combinational logic, mapping input combinations to output values. LUTs are a resource implemented in the combinational logic blocks (CLBs) [AMD, 2025] which are useful for creating a variety of Boolean functions.

Digital Signal Processors Digital Signal Processors (DSPs) are specialized modules that are optimized for high-performance arithmetic. In this project's context, they are primarily used for multiplication operations.

RAM Block RAM (BRAM) refers to blocks of synchronous static random-access memory (SRAM) integrated into FPGAs. These memory blocks provide high-bandwidth, on-chip storage for data.

2.3.2 Timing Constraints

Timing constraints are fundamental specifications that define the acceptable temporal behavior of a digital circuit on an FPGA. For any digital circuit, meeting these constraints is critical for the circuit to run as expected. For neural network implementations, meeting these constraints is crucial for achieving real-time inference and high throughput. Key timing metrics include the **clock frequency/period**, which sets the fundamental speed at which the circuit operates. Beyond this, two critical performance indicators are **latency** and **Initiation Interval (II)**.

Latency refers to the total number of clock cycles (or the absolute time) it takes for a single input to propagate through the entire circuit and produce its corresponding valid output. It represents the "start-to-finish" time for a single inference, making it critical for applications requiring immediate responses.

The Initiation Interval (II), conversely, defines the number of clock cycles that must pass before a new input can be accepted by the circuit. A smaller II (ideally 1) signifies higher throughput, allowing for the continuous processing of data streams.

2.4 Benchmark FPGA Board

The benchmark neural network model is evaluated on the Xilinx Alveo U250 board. The Alveo U250 is a high-performance, reconfigurable accelerator card designed for data center and cloud deployments, making it well-suited for machine learning inference tasks. It features a large Xilinx Virtex UltraScale+ FPGA, with an abundance of logic resources (LUTs, FFs), numerous Digital Signal Processing (DSP) blocks for high-throughput arithmetic operations, and substantial on-chip Block RAM (BRAM) for data storage. The selection of the Alveo U250 for this thesis provides a robust platform to demonstrate the performance and resource efficiency capabilities of HLS4ML in a realistic, high-end hardware environment.

Table 1 shows the resources available on the Alveo U250 board.

FFs	LUTs	DSPs	BRAMs
3,456,000	1,728,000	12,288	2,688

Table 1: Total Resources Available on Alveo U250

3 Methods

Implementing this benchmark model required new implementations of two neural network layers, namely BatchNormalization and Softmax, for efficient resource utilization on FPGAs. BatchNormalization, in particular, has been widely adopted in deep learning models due to its proven ability to improve training stability and accelerate convergence [Ioffe and Szegedy, 2015], leading to better overall model performance. This section details the methodologies developed to adapt these layers for hardware synthesis, highlighting the optimizations made for efficient FPGA implementation.

3.1 BatchNormalization Implementation

The BatchNormalization layer is initialized with parameters ϵ and learns parameters $\mu, \sigma, \gamma, \beta$. Once a model is trained, these parameters don't change. For a trained model, the BatchNormalization layer performs this operation to an input vector x:

$$\text{BatchNormalization}(x) = \gamma \odot \frac{(x - \mu)}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

Here, $\odot$, division, and the square root operate element-wise. This transformation can be shown to be a linear transformation of x in terms of a scale factor w and an offset factor b such that

$$\text{BatchNormalization}(x) = wx + b$$

where

$$w = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}$$

$$b = \beta - \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \cdot \mu$$

Combining Dense and BatchNormalization layers During Inference

The Dense layer computes

$$y = W_{\text{dense}}x + b_{\text{dense}}$$

where x is the input vector, W_{dense} is the weight matrix, and b_{dense} is the bias vector.

Both the Dense layer and the BatchNormalization layers behave as linear transformations during inference. Therefore, we can eliminate the BatchNormalization layer by adjusting the Dense layer's weights to incorporate the linear operation. A combined Dense-BatchNormalization layer would compute:

$$\begin{aligned} z &= \gamma \odot \frac{(W_{\text{dense}}x + b_{\text{dense}}) - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \\ &= \gamma \odot \frac{W_{\text{dense}}x}{\sqrt{\sigma^2 + \epsilon}} + \frac{(W_{\text{dense}}b_{\text{dense}}) - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \end{aligned}$$

Therefore, we can define the new weight matrix W_{new} and bias vector b_{new} as

$$W_{\text{new}} = \gamma \odot \frac{W_{\text{dense}}}{\sqrt{\sigma^2 + \epsilon}}$$

$$b_{\text{new}} = \frac{(W_{\text{dense}}b_{\text{dense}}) - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

and replace the Dense-BatchNormalization sequence with a single Dense layer defined by W_{new} and b_{new}:

$$z = W_{\text{new}}x + b_{\text{new}}$$

These parameters can be precomputed and used during the synthesis of the model. This optimization improves both resource utilization and timing constraints metrics without any cost.

3.2 Softmax Implementation

The Softmax operation computes the following operation $\sigma(x)$ for an input vector $x \in \mathbb{R}^d$:

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{d} e^{x_j}}$$

The exponential and division operations are approximated using look-up tables, which are indexed by the fixed-point representation of the input operand.

Figure 3 illustrates the accuracy of the hardware-implemented exponential function at 16_10 bitwidth, approximated via a look-up table, against the true exponential function. This comparison highlights how well the fixed-point, LUT-based approach mimics the ideal mathematical function, demonstrating the trade-off between hardware resource efficiency and numerical precision.

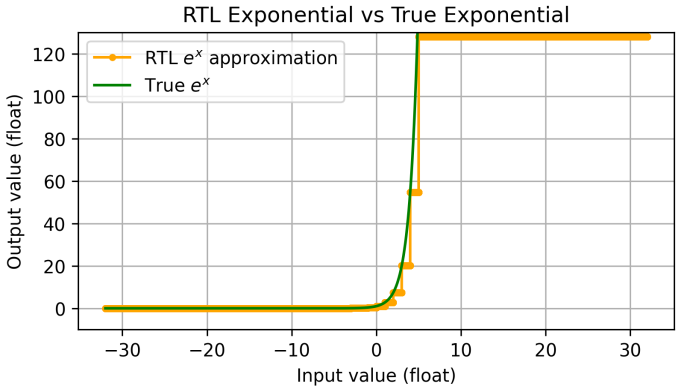

Figure 3: Comparison of Exponential Functions

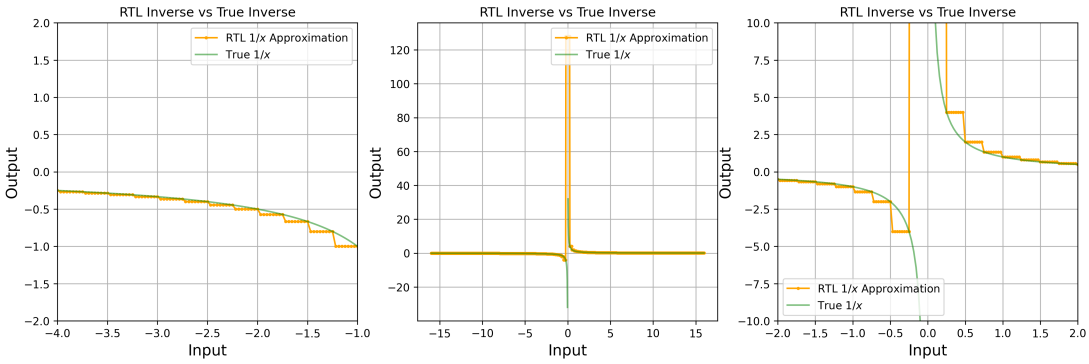

Figure 4: Comparison of Inverse Functions

Figure 4 presents a similar comparison for the inverse operation, a crucial component in the softmax division. It visualizes the behavior of the inverse function as implemented in hardware using a look-up table versus its ideal mathematical counterpart, showcasing the fidelity of the fixed-point approximation for division.

4 Results

The benchmark neural network model was implemented and evaluated using a fixed-point representation with a <16,10> bitwidth, signifying 16 total bits with 10 fractional bits. Prior work showed significant resource and performance optimizations could be achieved by carefully selecting the depth of a shift-add module [Johnson, 2023], which approximates multiplication operations. Building upon

this, the handmade implementation presented in this thesis utilized a Shift-Add depth of 4, which I found to be the most effective for maximizing efficiency.

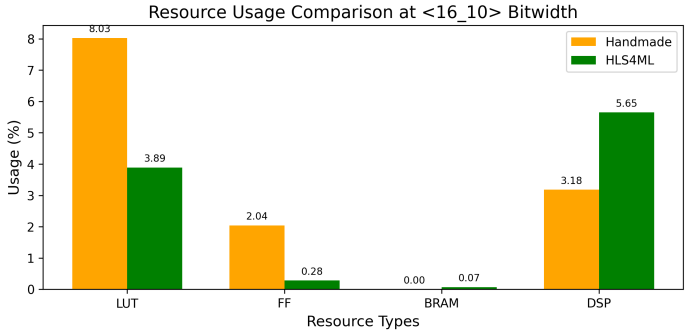

Figure 5: Benchmark Model Resource Utilization Comparison

Resource Utilization Metrics Figure 5 provides a side-by-side comparison of the resource utilization percentage for the benchmark neural network model when implemented using a handmade design versus an HLS4ML-generated design. The handmade implementation generally consumes more LUTs and FFs, while the HLS4ML design utilizes a larger percentage of DSPs and a small amount of BRAM.

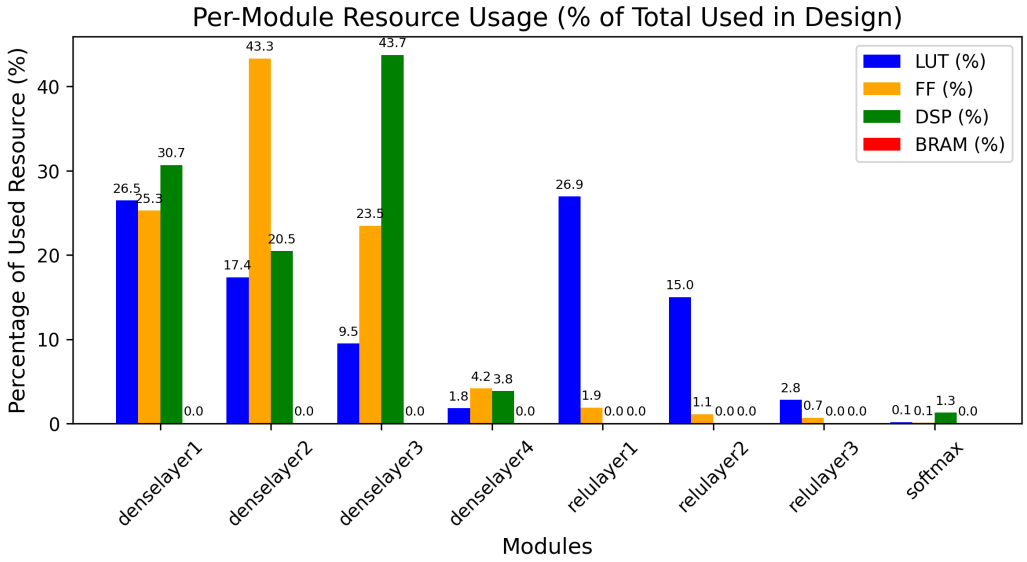

Figure 6: Percent of Total Resource Utilization (per module)

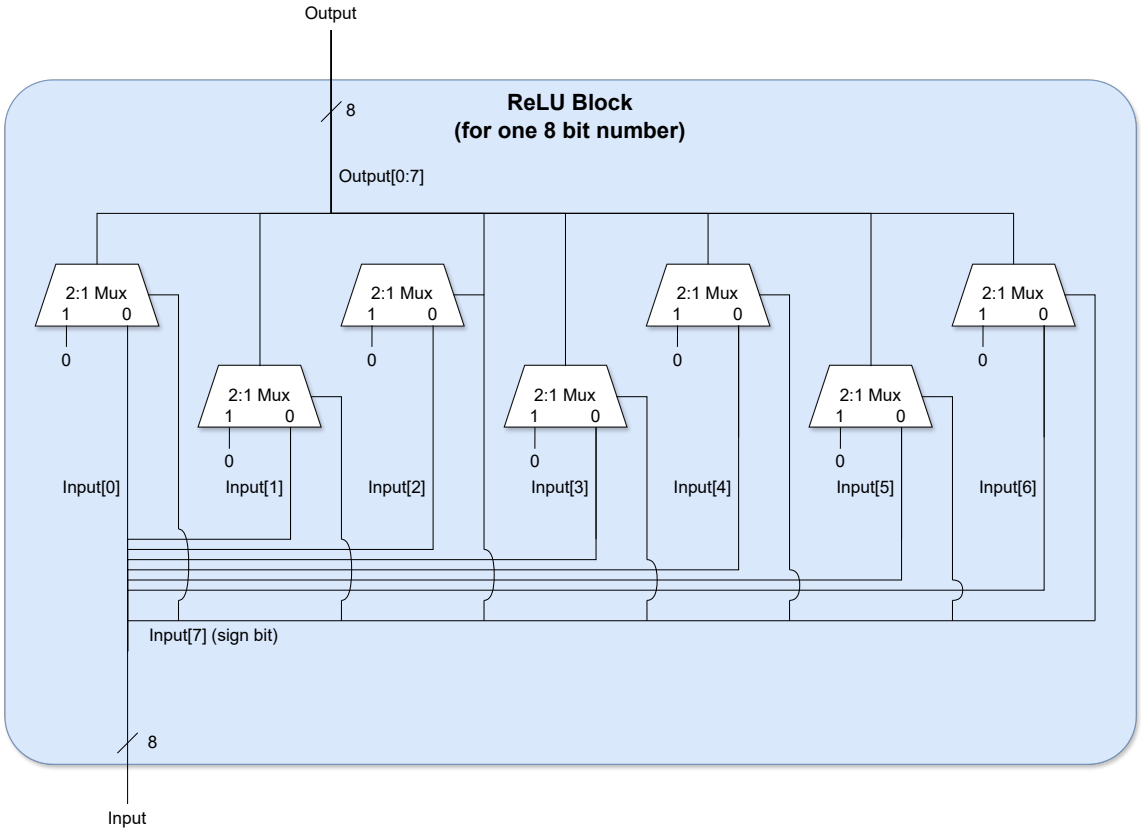

Figure 7: ReLU Implementation

Figure 6 shows the resource usage by individual modules within the handmade design. For each layer (e.g., denselayer1, relulayer1, softmax), it shows the percentage contribution of LUTs, FFs, DSPs, and BRAMs to the total resources consumed by the entire design. The dense-layer modules are the primary consumers of FFs and DSPs, while relulayer modules contribute significantly to LUT utilization, providing insight into the resource bottleneck within the architecture.

I was surprised to find the LUT usage of ReLU so high. ReLU is a simple operation that only uses LUTs to compute $\max(0, x)$. Looking at the synthesized HLS4ML modules, HLS4ML only instantiates one ReLU module. This might contribute to the discrepancy between resource utilization. Figure 7 shows the expected implementation of the ReLU operation for an 8 bit number. As the figure demonstrates, $n-1$ MUXes are required to implement an n-bit number. A basic design would use one LUT for the implementation of a 2-to-1 MUX. The model for the first ReLU layer has 64 16-bit inputs, which means it should be possible for the first ReLU layer to be implemented

with $64 \cdot (16 - 1) = 960$ LUTs. Comparing against the LUT usage for ReLU Layer 1 in Table 6, the real design's usage is 37380, whereas the expected usage is 960.

Design	Max Clock Frequency	Latency	Initiation Interval
Handmade	206.23 MHz	368.5ns	1
HLS4ML	164.50 Mhz	79.03ns	1

Table 2: Comparison of Timing Metrics

Timing Metrics Table 2 shows the timing results of both designs. Compared to HLS4ML, the handmade design can run at a faster clock frequency, but has higher total latency. Both designs have an II of 1.

5 Conclusion

Building upon the insights gained from this thesis, several promising avenues for future research emerge, particularly concerning the optimization of neural network layers on FPGAs.

One key area for future investigation is the optimal utilization of Look-Up Tables (LUTs) for ReLU activation functions. As said in section 4, HLS4ML only instantiates one ReLU module. Investigation into how HLS4ML computes ReLU for 3 activation layers might prove insightful for learning about optimizations HLS4ML and HLS are doing.

This thesis shows that another critical area for optimization is still the dense layer. The resource-intense multiplication and summation still uses more resources in the handmade designs compared to HLS4ML.

Furthermore, given the rapid advancements in deep learning research, exploring the hardware implementation of additional and emerging activation layers presents a significant area for future work. The landscape of neural network activation functions is constantly evolving, with new layers introduced in research papers frequently to address specific challenges or improve model performance. Investigating how these diverse activation functions (e.g., ELU, SELU, Swish, GeLU) can be efficiently mapped onto FPGA resources, what approximation techniques are most suitable, and how HLS4ML can be extended to support them, would be valuable. This research would contribute to broadening the applicability and efficiency of FPGA-accelerated neural networks to a wider range of contemporary deep learning architectures.

BIBLIOGRAPHY

AMD. UltraScale Architecture Configurable Logic Block User Guide AMD Technical Information Portal. https://docs.amd.com/r/en-US/ug574-ultrascale-clb/CLB-Resources, 2025. URL https://docs.amd.com/r/en-US/ug574-ultrascale-clb/CLB-Resources.

Javier Duarte et al. Fast inference of deep neural networks in FPGAs for particle physics. *JINST*, 13(07):P07027, 2018. doi: 10.1088/1748-0221/13/07/P07027.

FastML Team. fastmachinelearning/hls4ml, 2025. URL https://github.com/fastmachinelearning/hls4ml.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015. URL https://arxiv.org/abs/1502.03167.

Caroline Johnson. Evaluating the Quality of HLS4ML's Basic Neural Network Implementations on FPGAs. 2023.

Waiz Khan. Quantifying the Performance and Resource Usage of HLS4ML's Implementation of the Batch Normalization Layer on FPGAs. 2024.

Jennifer Ngadiuba, Vladimir Loncar, Maurizio Pierini, Sioni Summers, Giuseppe Di Guglielmo, Javier Duarte, Philip Harris, Dylan Rankin, Sergo Jindariani, Mia Liu, Kevin Pedro, Nhan Tran, Edward Kreinar, Sheila Sagear, Zhenbin Wu, and Duc Hoang. Compressing deep neural networks on fpgas to binary and ternary precision with ¡tt¿hls4ml¡/tt¿. *Machine Learning: Science and Technology*, 2(1):015001, December 2020. ISSN 2632-2153. doi: 10.1088/2632-2153/aba042. URL http://dx.doi.org/10.1088/2632-2153/aba042.

Zhyar Rzgar K. Rostam, Sándor Szénási, and Gábor Kertész. Achieving peak performance for large language models: A systematic review. *IEEE Access*, 12:96017–96050, 2024. ISSN 2169-3536. doi: 10.1109/access.2024.3424945. URL http://dx.doi.org/10.1109/ACCESS.2024.3424945.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge, 2015. URL https://arxiv.org/abs/1409.0575.

Appendix

Proof of BatchNormalization Linear Transformation

$$
\begin{aligned}
y &= \frac{\gamma(x-\mu)}{\sqrt{\sigma^2+\epsilon}} + \beta \\
&= \frac{\gamma x}{\sqrt{\sigma^2+\epsilon}} - \frac{\gamma\mu}{\sqrt{\sigma^2+\epsilon}} + \beta \\
&= \frac{\gamma}{\sqrt{\sigma^2+\epsilon}} x + \left(\beta - \frac{\gamma\mu}{\sqrt{\sigma^2+\epsilon}}\right) \\
&= wx + b \qquad\qquad \text{where } w = \frac{\gamma}{\sqrt{\sigma^2+\epsilon}} \text{ and } b = \beta - \frac{\gamma\mu}{\sqrt{\sigma^2+\epsilon}}
\end{aligned}
$$

Resource Utilization

Resource	Utilization	Available	Utilization %
LUT	138742	1728000	8.03
FF	70570	3456000	2.04
BRAM	0	2688	0.00
DSP	391	12288	3.18

Table 3: Handmade Resource Utilization

Resource	Utilization	Available	Utilization %
LUT	67147	1728000	3.89
FF	9712	3456000	0.28
BRAM	2	2688	0.07
DSP	694	12288	5.65

Table 4: HLS4ML Resource Utilization Tables

Module	LUT	FF	BRAM	DSP
denselayer1	36736	17828	0	120
denselayer2	24087	30580	0	80
denselayer3	13167	16555	0	171
denselayer4	2506	2941	0	15
relulayer1	37380	1336	0	0
relulayer2	20810	788	0	0
relulayer3	3882	480	0	0
softmax	206	59	0	5

Table 5: Handmade Per-Module Resource Utilization