Modeling On-Chip Variations in Analog Neural Network Accelerators with an FPGA

Dinali Assylbek

Team Members:

Prof. Armin Tajalli, University of Utah (supervisor)

Michael Keyser, University of Utah (Mentoring/Advice)

Farzad Ordubadi, University of Utah (Mentoring/Advice)

Kenneth Van Gordon

Laboratory of Circuits & Systems

Electrical & Computer Engineering Department

University of Utah

Date: August 22, 2025

Abstract

This report presents the design, implementation, and evaluation of an FPGA-based emulator for studying mismatch effects in neural networks. Analog accelerators promise faster and more energy-efficient inference than CPUs or GPUs, but they are highly sensitive to process, voltage, and temperature (PVT) variations that can introduce errors. Understanding these mismatch effects is crucial for determining whether analog hardware is practical for real-world applications. A previous LCAS project developed a Python-based emulator to study this problem[1], but its slow runtime and limited scalability made it unsuitable for larger experiments. This project addressed those limitations by introducing an FPGA backend and pursuing two main objectives: exploring the scalability of mapping larger networks to FPGA hardware and running a complete mismatch experiment on a trained MNIST model. The scalability study revealed that DSPs are the primary resource bottleneck, while LUTs, FFs, and BRAM remained within safe limits, and also demonstrated how the reuse factor and precision choices influence feasibility. The mismatch experiment demonstrated that the full end-to-end pipeline can be executed on the FPGA, though results showed a performance gap with the Python baseline due to fixed-point versus floating-point differences. While execution on hardware was slower at this stage, the framework proved functional in practice and established a foundation for future improvements. Together, these results highlight both the potential and current challenges of FPGA-based emulation, laying the groundwork for more scalable, efficient, and reusable tools for testing analog-inspired neural networks under realistic variations.

[1] https://lcaswiki.ece.utah.edu/doku.php?id=sinproj:seniorproj_isaac_vaughan_2024

Problem Statement & Objectives

2.1 Why FPGA-Based Emulation Matters

The broader motivation for this work has already been laid out in detail in the previous LCAS report, which provides a comprehensive background on convolution neural networks, accelerators, and the challenges of process, voltage, and temperature (PVT) variations. Rather than restating that material here, this report will briefly summarize the key ideas and direct the reader to that document for a more complete discussion.

In short, analog neural network accelerators promise significant power and speed advantages compared to traditional GPU or CPU approaches. However, they are also highly sensitive to mismatch effects caused by process, voltage, and temperature (PVT) variations. In practice, this means that internal components can deviate slightly from their expected behaviour. For example, values might deviate slightly from their intended level, as is often observed with resistors. This is generally not an issue in digital designs but can introduce errors in analog systems.

To study these effects, the LCAS strives to develop an emulator framework that can run "mismatch experiments" either entirely in Python or with computational offloading to an FPGA. The purpose of these mismatch experiments is to simulate realistic conditions that a neural network might encounter in an analog chip, helping determine which models are most suitable for analog hardware and whether the resulting errors are acceptable or problematic.

A Python-based emulator was developed in earlier work, providing flexible experimentation with various models, datasets, and configurable sigma values to control the levels of variation. However, a purely Python-based approach proved slow and limited in scalability. To address these limitations, an FPGA backend was introduced as FPGAs can execute inference much faster while also exposing real hardware trade-offs such as DSP usage and memory utilization.

This FPGA-based emulation is important because it moves beyond purely theoretical modeling into hardware-aware validation. Beyond just speed, it allows us to evaluate which kinds of networks can realistically be supported on FPGA fabric, how resource constraints shape the design, and whether mismatch experiments can even be run at a practical speed. Thus, this work continues this direction by focusing on making the FPGA emulator more scalable and demonstrating its viability on a complete image classification model.

2.2 Project Objectives

In the previous iterations of this project, significant challenges arose when attempting to fit larger models on the FPGA board. These limitations were largely due to the restricted hardware resources of the device, particularly the limited number of DSPs. Furthermore, while partial progress was made, a complete mismatch experiment had never been fully carried out on the FPGA. This work focused on addressing these shortcomings through two main objectives.

The first was to test the scalability of the FPGA-based emulator by exploring how larger neural networks can be mapped onto the hardware. This involves examining how network size and architecture affect feasibility, as well as investigating strategies for fitting larger networks onto

the FPGA fabric. The second objective was to successfully run and validate a complete image-based neural network with a mismatch experiment. With that comes establishing an end-to-end experiment flow on the FPGA and ensuring that mismatch-induced variations could be introduced, measured, and analyzed. Beyond achieving these immediate goals, the works also aimed to lay the groundwork for a more reusable framework- developing scripts, libraries, and design practices that could support future iterations and make the emulator easier to extend to new models and experiments.

Design and Implementation

3.1 Scalability Exploration

Early attempts to fit neural networks on the FPGA encountered significant barriers. For example, a model with 500 parameters could not be placed successfully, and even with HLS4ML, it could not successfully synthesize a network with exceeding 60,000 weights. Although optimization options existed (reuse factor, precision), the early attempts lacked a consistent hardware integration template for systematic testing.

With a fixed circuit template in place, a systematic exploration of scalability was possible. The primary strategy was adjusting the reuse factor in HLS4ML, which controls how many times each multiplication is reused to compute neuron values in each layer. Lower reuse factor parallelizes computation but requires more DSP resources, while higher reuse factors save DSPs at the cost of longer latency. Precision reduction was also considered as an optimization strategy, though testing was mostly limited to larger networks.

Each candidate model (with varying layer counts, parameter sizes, and architectures) was synthesized and implemented in Vivado. Resource utilization was then collected directly from implementation reports, including:

- LUTs (Look Up Tables): 53,200 available
- LUTRAM (Look Up Tables Random Access Memory): 17,400 available
- FF (Flip-Flops):106,400 available
- BRAM (Block Random Access Memory): 6.3 MB / 160 block available
- DSP (Digital Signal Processing): 220 available
- BUFGs (Global Clock Buffer): 32 available

For each design, Vivado's reported usage was compared against the board's fabric limits to determine whether the configuration was feasible. Results were expressed as utilization percentage to make comparisons across reuse factor and precision experiments straightforward. Additionally, compilation time in HLS4ML was tracked, since previous attempts had shown that large models were prone to long synthesis times or outright failures.

To explore feasibility, custom networks were generated (not trained for a particular task) with varying parameter counts, depths, and architectures. Each network was inserted into the fixed circuit template, synthesized, and implemented. The resulting reports revealed the impact of reuse factor settings on LUT, BRAM, and DSP usage. By iteratively adjusting reuse factors and

comparing utilization, the practical upper bounds of model size that the board could support were identified.

Looking ahead, scalability may also be constrained not by fabric resources but by the SoC memory system itself. The ZedBoard provides 512 MB of DDR3 and 256 Mb of QSPI flash, which places hard limits on how many weights and how large of dataset can be loaded into memory at runtime. These considerations highlight that true scalability must account for both fabric utilization and on-chip/off-chip memory availability.

3.2 MNIST Mismatch Experiment

The goal of this project was to run a full mismatch experiment on the FPGA. For this, a custom TensorFlow model trained on MNIST was chosen. The model was intentionally designed to be small and controllable, with a fixed number of parameters, so the focus remained on completing the mismatch experiment rather than handling the complexity of a large-scale network. The architecture shown in Figure 1 consists of 500 trainable parameters, 490 kernel weights, and 10 bias weights. Each 28 x 28-pixel MNIST image first passes through an average pooling layer, then is flattened, and finally processed by a dense layer containing all trainable parameters. The output is a 10-element vector produced by a SoftMax layer where each value is a 16-bit probability. Since only the dense layer is trainable, this is effectively a one-layer model.

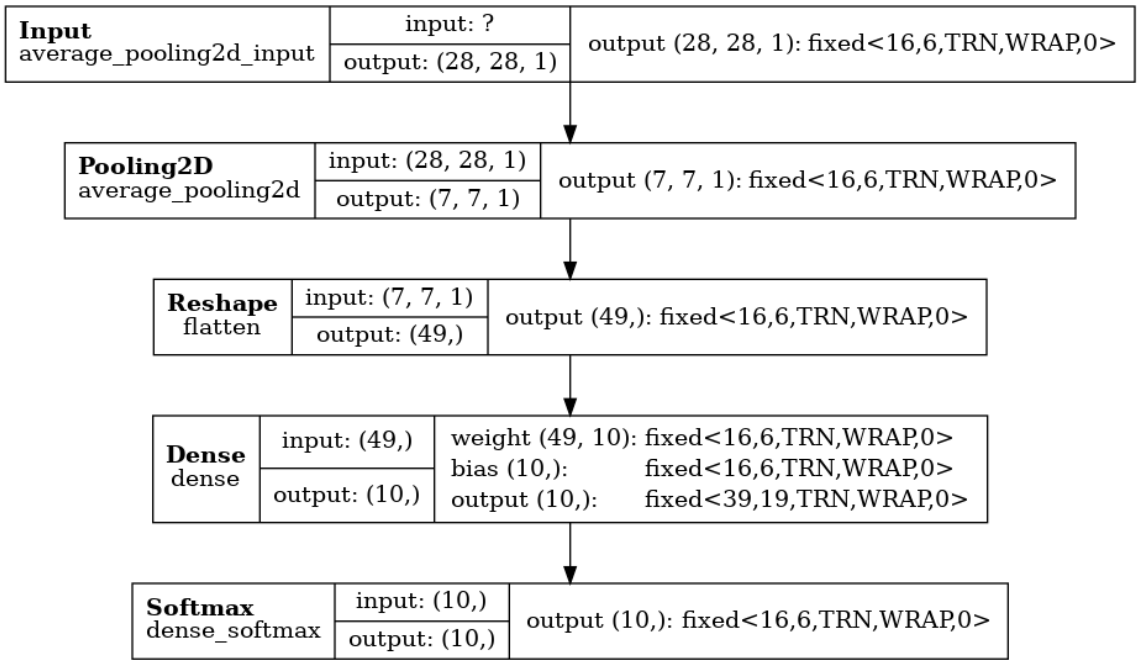

Figure 1: Architecture of the custom MNIST model with one dense layer containing 500 trainable parameters and a SoftMax output.

The model was trained in TensorFlow on the MNIST dataset and then converted into programmable logic using HLS4ML, with internal computations restricted to 16-bit precision.

The converted design was connected to the Zynq SoC and peripherals through a Vivado block design, while all host communication and control logic were implemented in C using Vitis. This setup allowed the FPGA to run inferences while the host managed data transfer and mismatch configuration.

The mismatch methodology applied to the model weights followed a simple but rigorous approach. For a given experiment, an array of σ values were defined to represent the magnitude of variation applied to the weights. Each weight was perturbed according to a Gaussian distribution with a mean and a standard deviation. The adjustments were generated using the Box-Muller transform:

$$z_0 = \sqrt{-2\ln(u_1)} * \cos(2\pi u_2), \quad w' = \mu + \sigma z_0$$

where u_1 and u_2 are uniform random numbers drawn from [0,1), and w' is the mismatched weight. This procedure ensured that each modified weight followed a normal distribution centered around its original value.

The experiment flow proceeded as follows. For each iteration, the host transmitted the dataset and the current set of weights to the FPGA. A specific σ value was selected, and the weights would be changed accordingly before being written into BRAM. The model on the FPGA then loaded the modified weights and performed inference on a chosen digit from the dataset. The prediction result was sent back to the host, which recorded the outcome. This process was repeated across different σ values, creating a systematic record of model performance under increasing levels of mismatch. A flowchart of this experiment pipeline is provided in Figure 2.

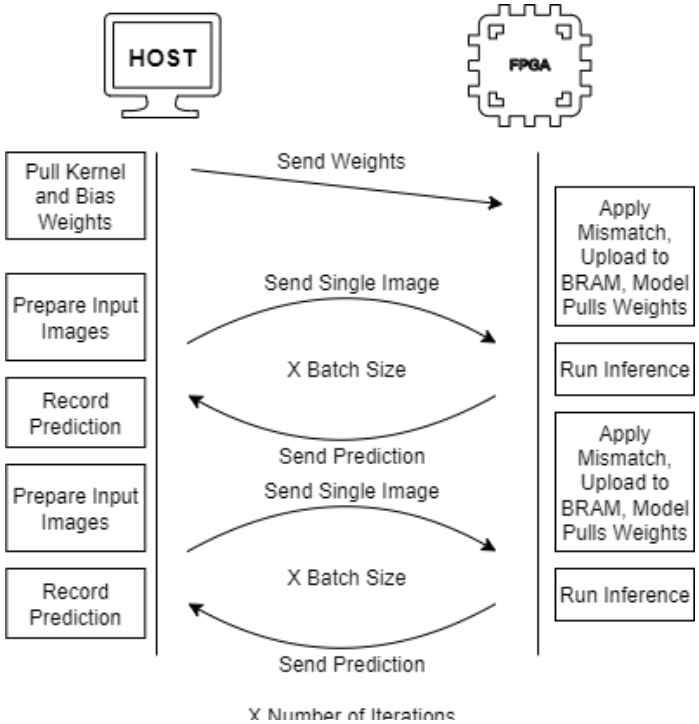

Figure 2: Flow of the mismatch experiment showing host–FPGA communication, with weights sent once and mismatches applied each iteration.

Results

4.1 Scalability Results

Although the dataset of experiments was relatively small, the results still provide useful insight into how different resources scale with the reuse factor and network size. The maximum network size successfully explored was 10,497 parameters, since larger models synthesized in HLS4ML but failed to compile in Vivado, eventually exhausting server memory. As such, these findings should be considered preliminary, pointing to general trends that warrant further investigation.

4.1.1 DSP Usage

DSP blocks were previously the most critical bottleneck in fitting models to the ZedBoard, and the results here reinforce their importance. Across all model sizes, a clear correlation was observed between reuse factor and DSP count: higher reuse factors consistently lowered DSP usage, with a steep initial drop before leveling off. For example, the smallest model dropped from 88 DSPs at reuse = 1 to just 3 at reuse = 64. Larger models showed the same downward trend, though at very high reuse factors the DSP count ticked upward again, suggesting that scheduling overhead may complicate the simple trend. Latency is assumed to increase with higher reuse factors, but this was not measured since latency was not a primary concern for this experiment. The DSP trends are summarized in Figure 3.

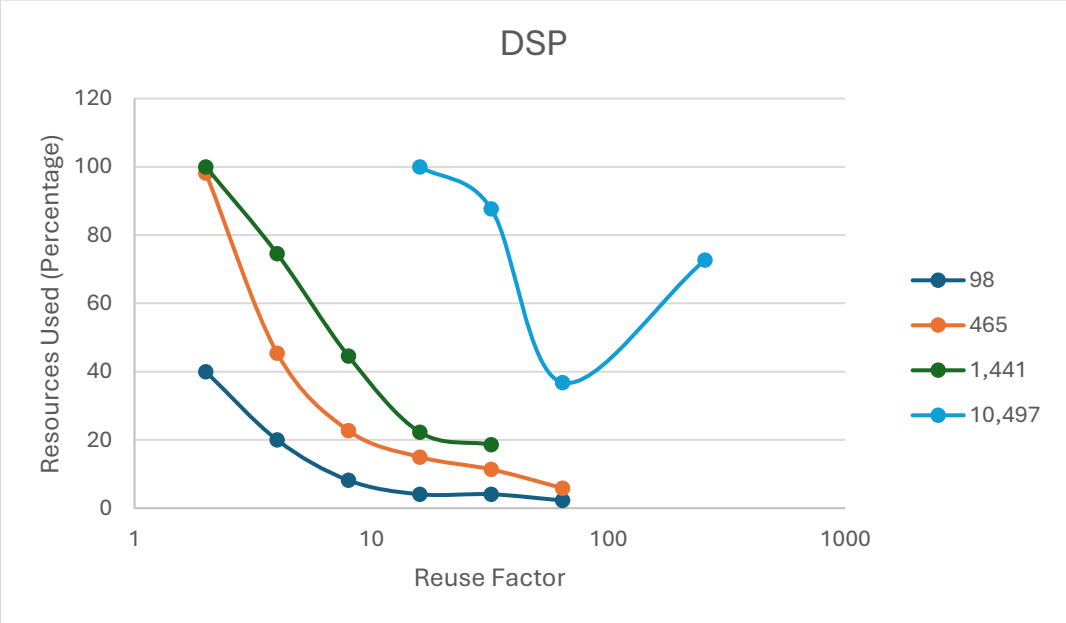

Figure 3: DSP usage across models of different sizes, showing a steep decrease with higher reuse factors before leveling off, with larger models exhibiting slight increases at extreme reuse values.

4.1.2 LUT and FF Usage

The Look-Up-Table and Flip-Flop counts followed a similar pattern across experiments, though with more variability and less predictability compared to DSPs. Resource usage fluctuated slightly with reuse factor, sometimes showing a local minimum where a particular reuse factor seemed to "fit" the architecture more efficiently. Despite this instability, all models remained well below the ZedBoard's thresholds, leaving a comfortable margin. This suggests that LUTs and FFs are not immediate limiting factors for the sizes explored. The observed behavior is displayed in Figure 4.

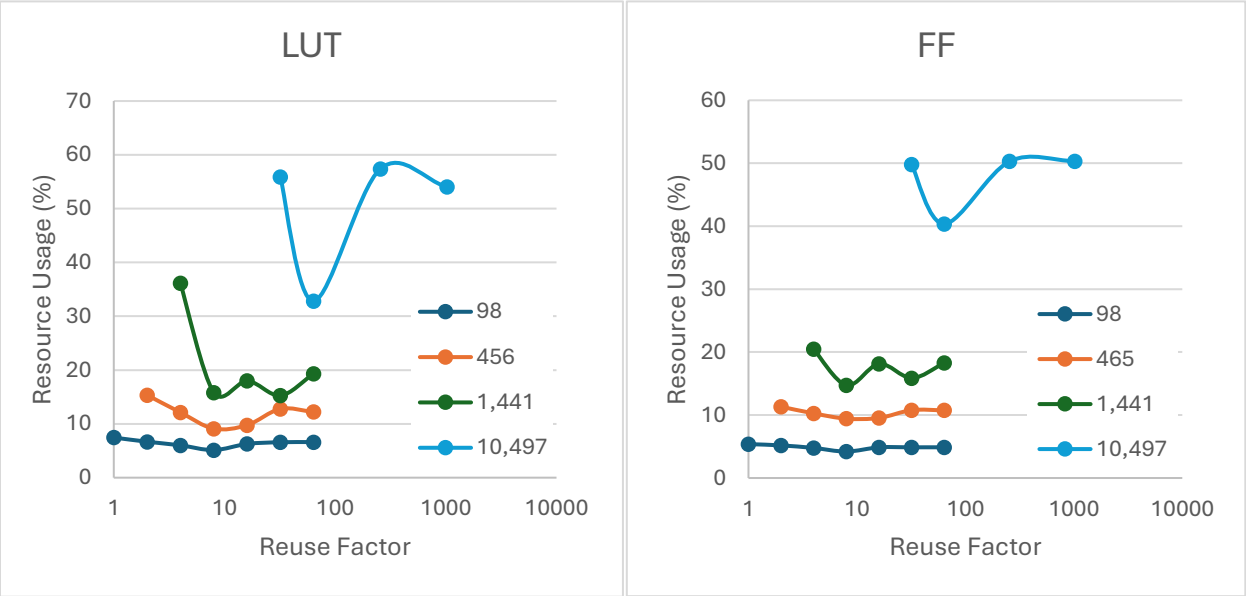

Figure 4: LUT and FF usage across different model sizes, showing generally low utilization well below device limits, with some fluctuations depending on reuse factor.

4.1.3 BRAM and LUTRAM Usage

Unlike DSPs, BRAM and LUTRAM usage remained flat across all tested reuse factors and model sizes. BRAM use stayed at 6.5 blocks, which is expected since the tested models did not inherently rely on on-chip BRAM for weights. In practice, larger models would likely increase BRAM utilization, as external BRAM would be required to store weights. However, because these experiments were performed with a fixed design template, the contribution from BRAM and LUTRAM remained stagnant. The data for these resources is presented in Figure 5.

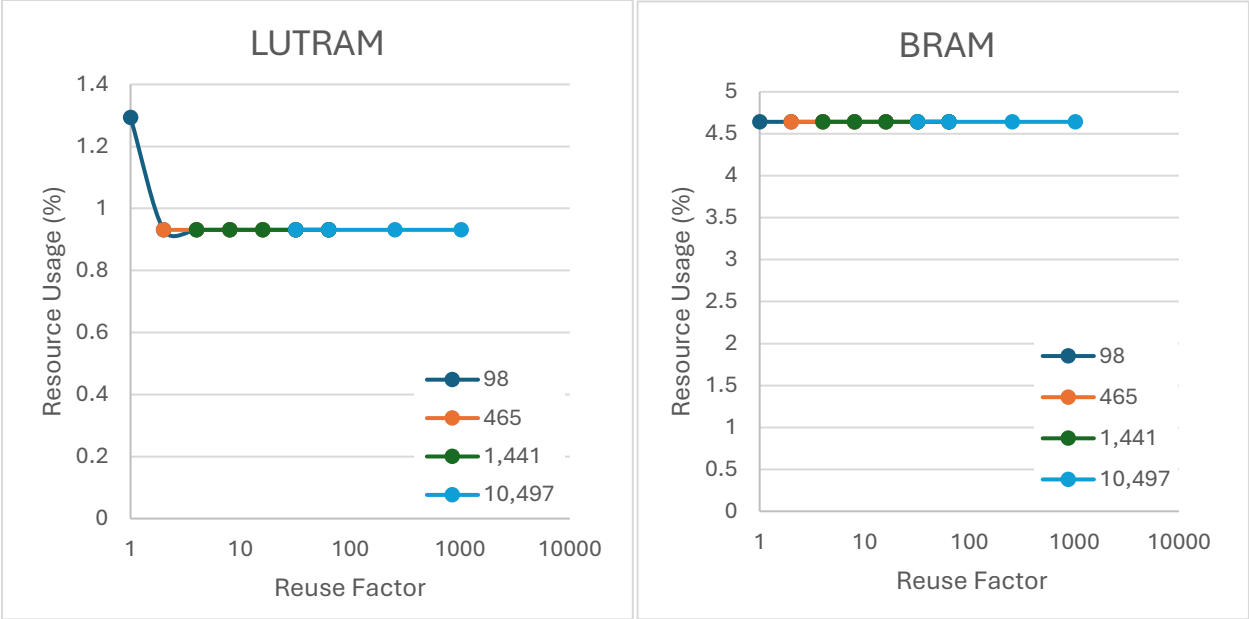

Figure 5: BRAM and LUTRAM usage across all tested models, remaining nearly constant and well below device limits regardless of reuse factor.

4.1.4 Compilation Time

Finally, compilation time was tracked, not as a resource constraint but as a practical metric. The overall trend mirrored DSP usage: as the reuse factor increased, compilation time generally decreased. For smaller models, compile times stayed consistently under one minute, but for larger models, the trend reversed at very high reuse factors, where compile times began to rise again. This suggests diminishing returns at extreme reuse values. The compilation time results are displayed in Figure 6.

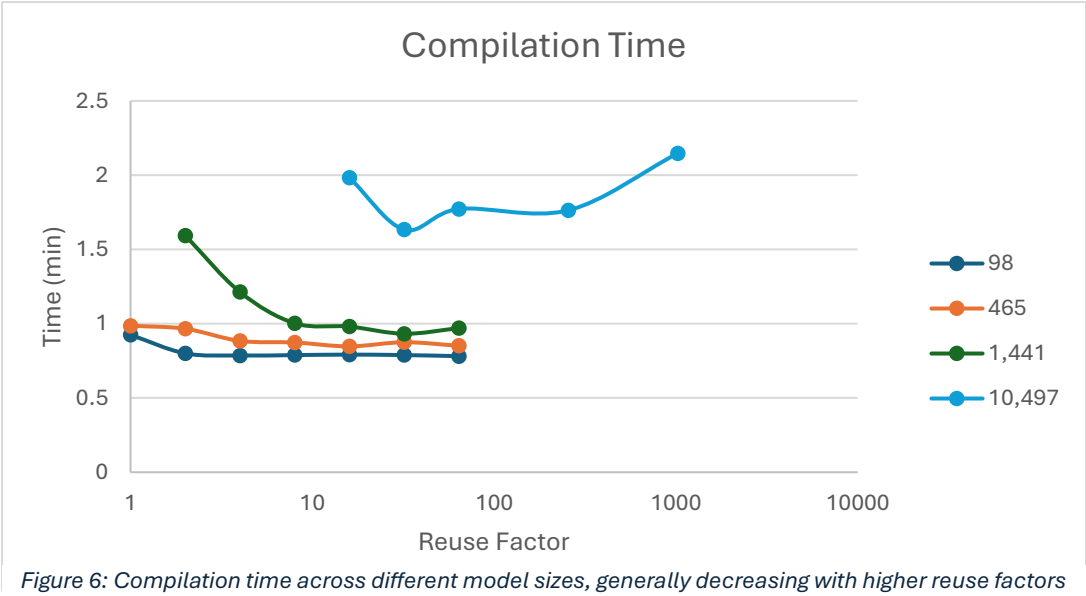

Figure 6: Compilation time across different model sizes, generally decreasing with higher reuse factors but rising again for the largest model at extreme values.

Overall, these experiments showed that DSPs are still the main limit on the ZedBoard. Using higher reuse factors helps cut down DSP use, though it comes with more latency. LUTs and FFs moved around a bit but never came close to the limits, and BRAM/LUTRAM stayed flat for the models tested. Compile time mostly went down as reused increased, but for big models and very high reuse factors it started to creep back up. Overall, reuse factor proved to be the primary parameter controlling scalability, but pushing it too far brings fewer gains and some new trade-offs.

4.2 MNIST Mismatch Results

The mismatch experiment with the MNIST model was carried out on the FPGA and compared against a Python-based emulator of the same process. In both cases, the goal was to measure baseline accuracy and then track how accuracy degraded as mismatch was applied with increasing values of σ. The Python implementation, shown in Figure 7, represents the expected outcome: the models start with a baseline testing accuracy of 88% when $\sigma = 0$, and accuracy gradually decreases as σ increases. The degradation is smooth and relatively controlled, with only minor variance across iterations.

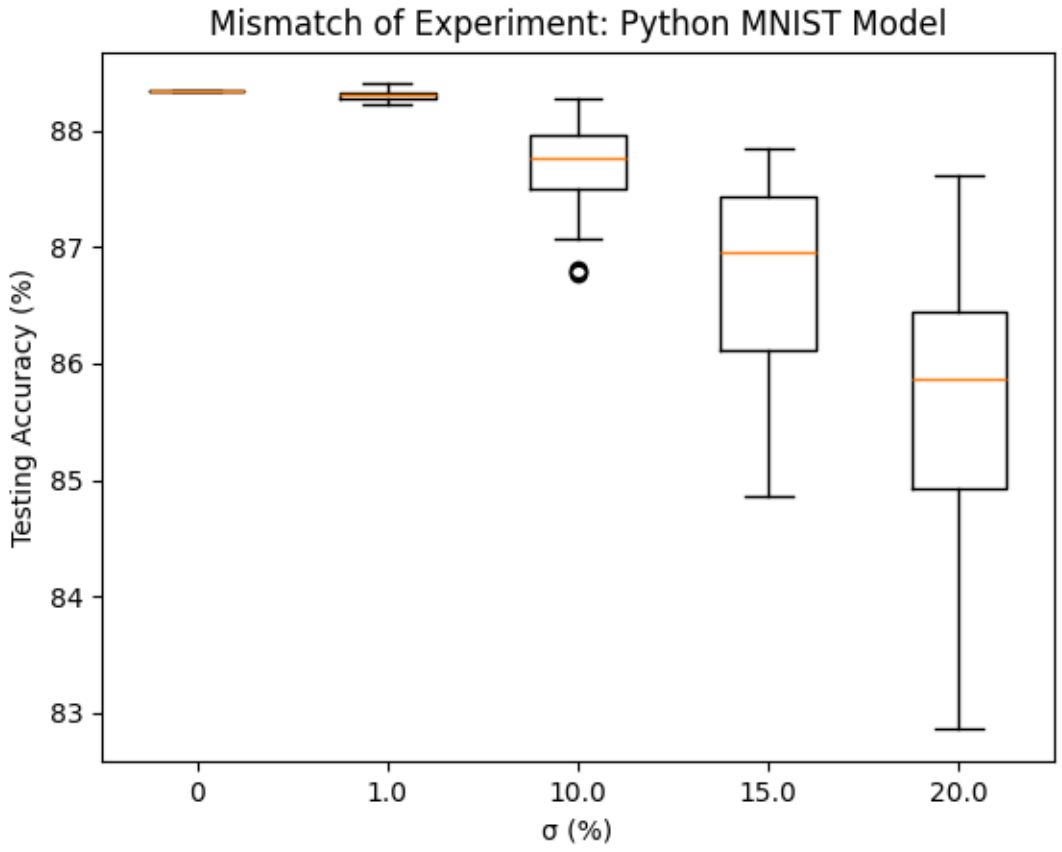

Figure 7: Python-based mismatch experiment showing smooth accuracy degradation as σ increases.

The FPGA results, shown in Figure 8, tell a different story. At $\sigma = 0$, the baseline accuracy already starts lower, 82.5%, meaning that even without any mismatch noise, there is a noticeable gap compared to the Python results. As the mismatch is increased, accuracy drops more sharply and the spread of results widens, especially at σ value of 15% and 20%. This behaviour confirms that the two systems are not perfectly aligned. A major factor here is that Python used floating-point weights while the FPGA implementation used 16-bit fixed-point weights. Ideally, the weights would have been restricted to 16 bits during training so that both platforms behaved the same, but that step was not completed. This quantization mismatch most likely explains much of the systematic difference observed between Figures 7 and 8.

For both the Python and FPGA experiments, the test setup used a batch size of 100 images, meaning that the same 100 MNIST test images were sent to the model for each iteration, and the process was repeated for 50 iterations. This approach is not fully representative of the MNIST dataset, which has a total of 70,000 images, but was chosen because the system runs slowly and time constraints limited the scale of the experiment. Even though the dataset was reduced, the trends in accuracy degradation were still clear.

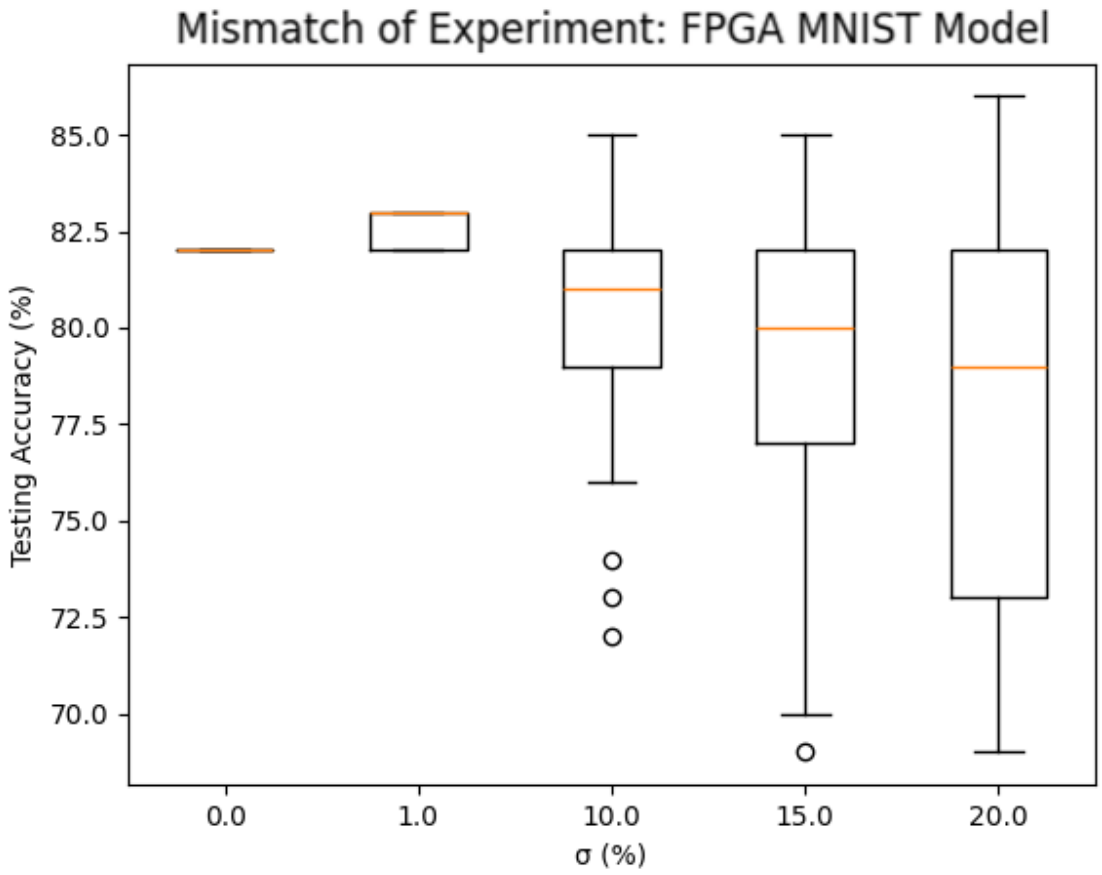

Figure 8: FPGA-based mismatch experiment showing accuracy degradation and increased variability as σ increases.

It is important to point out that the long-term goal of this project is to eventually use FPGA acceleration to speed up mismatch experiments, not slow them down. At the current stage, execution on hardware is significantly slower than Python, but this was expected given the early stage of implementation. Several ideas for improving speed and efficiency are outlined in the section titled *Future Work*. Overall, these results demonstrate that the mismatch experiment can run end-to-end on the FPGA, but that further refinement, especially fixed-point aware training and performance improvements, will be needed before the hardware results can fully match the Python baseline. Taken together, the scalability and mismatch results show both the potential and the current limitations of the FPGA approach.

Technical Challenges

A key challenge in this work was the rigidity of FPGA design tools such as Vivado. Building circuits required extremely precise configurations, and even small missteps in the settings could lead to non-functional designs. While the graphical block design environment provided access to many pre-built modules, these often came with limited documentation and little available support in the literature or user forums. As a result, components such as the DMA engine, BRAM generator, and BRAM itself required a significant amount of trial and error before their behavior and constraints were fully understood.

Another recurring difficulty was the lack of comprehensive resources and documentation of the tools used. Both the Xilinx ecosystem and the HLS4ML framework often required piecing together knowledge from scattered sources or experimenting directly on hardware to identify the correct design flow. Additionally, since synthesis and implementation processes are slow, retrying experiments or different configurations was highly time-consuming. Although this made progress slower at times, it also provided a deeper understanding of how these modules operate under the hood and how they interact within larger systems.

Conclusion

This project demonstrated both the potential and the challenges of using FPGA-based emulation for mismatch experiments. On the scalability side, this project was able to test how the larger networks map onto the ZedBoard and saw clearly how the reuse factor, precision, and DSP usage shape what is possible. Although LUTs, FFs, and BRAM were not immediate limits, DSPs quickly became the deciding factor, showing that resource balancing will be key for future designs.

On the experimental side, a full mismatch pipeline was successfully run on the FPGA, marking an important step forward. The results showed expected trends with accuracy dropping as mismatch increased, but also highlighted clear gaps compared to the Python baseline, mainly because of the fixed-point versus floating-point differences. Even though the hardware runs were slower than Python at this stage, the project proved that an end-to-end mismatch experiment can be carried out on the FPGA and that the framework works in practice, not just in theory.

Overall, this work lays the foundation for future improvements. With better fixed-point aware training, optimization for speed, and more flexible templates for hardware integration, FPGA-

based emulation can become a powerful and reusable tool for exploring how neural networks behave under real-world variations.

Future Work

Moving forward, LCAS plans to pursue the following areas of research and development to advance the analog neural network emulator:

- **Increased Automation with the Emulations Performed on the FPGA**: The current workflow for running mismatch experiments on the FPGA is highly manual and requires considerable expertise with both the hardware and software toolchains. Each new model demands custom circuitry, IP module configuration, and application-level adjustments before a full mismatch experiment can be executed. This manual process slows down development and increases the likelihood of user error. Looking ahead, the goal is to streamline and automate much of this workflow. For example, custom TCL scripts could be developed to automatically generate the required FPGA circuitry within Vivado, removing many of the repetitive setup procedures. On the software side, Python-based utilities could be created to handle dataset preparation and weight extraction directly from TensorFlow models, ensuring that they are formatted and ready to send to the FPGA. By automating these steps, the emulator would become significantly easier to use, faster to iterate on, and less dependent on deep technical knowledge of the underlying system.

- **Extending Beyond MNIST and Supporting Larger Networks:** While implementing a complete flow with the MNIST dataset marks an important milestone, the current model is far smaller and simpler than the kinds of networks typically used in real-world applications. Practical image-based neural networks, such as modern CNNs, require far more layers, parameters, and complex input-output mappings than MNIST can provide. To make the emulator a meaningful tool for research and deployment, future efforts must focus on scaling to larger models with richer datasets (e.g., CIFAR or ImageNet) and exploring how these networks can be efficiently mapped onto FPGA hardware. This will require careful consideration of resource constraints, precision trade-offs, and data transfer bottlenecks, as well as potential architectural changes to better support deep and computationally demanding models. By extending support beyond MNIST, the emulator can transition from a proof-of-concept to a platform capable of evaluating the types of neural networks that “actually do work” in state-of-the-art vision and learning tasks.

- **Improve Speed of Mismatch Experiments**: As mentioned previously, the central motivation for moving neural network emulation onto the FPGA was to achieve significant speedups compared to running mismatch experiments on the Python-based emulator. However, in its current state, the FPGA implementation is actually slower than the Python version, even before factoring in the time-consuming setup process. This gap highlights the need for substantial optimization in future iterations. One major bottleneck is communication between the FPGA and the host computer. At present, all transfers

occur over UART, which is limited to a baud rate of 115,200 – far too slow for larger models and datasets. Transitioning to Ethernet, as suggested in the previous report, would provide a massive boost in bandwidth and reduce data transfer time. Another area for improvement lies in the communication protocol itself: the current method prioritizes stability by requiring the FPGA to acknowledge each data point and weight, but this adds considerable overhead. Exploring lighter-weight protocols or batch-based acknowledgement could significantly reduce transmission latency. Finally, an idea raised earlier but not yet pursued is leveraging the FPGA's onboard SD card for dataset storage. By preloading input data locally on the FPGA, only weights, biases, and results would need to be exchanged with the host, cutting down communication traffic substantially. Together, these enhancements could transform the FPGA implementation into the faster, more efficient mismatch experiment platform it was originally envisioned to be.