FPGA Inference Accelerator for Small Neural Networks:
A Framework for Basys 3

Dylan Joaquin Sandall, June 2024. Cal Poly San Luis Obispo, EE509

Abstract

With the rapid rise in artificial intelligence, the demand for efficient hardware accelerators has surged. While GPUs are the dominant choice, FPGAs present unique advantages in terms of reconfigurability, which can lead to higher efficiency and parallelism. This research aims to develop a software framework for generating HDL designs that achieve high-throughput, low-latency, and power-efficient inference for small neural networks on FPGAs. By leveraging the highly configurable nature of FPGAs, this study explores the implementation of neural network architectures directly in hardware, optimizing core functions such as weight multiplication. The research draws on foundational studies and recent advancements in FPGA-based neural network acceleration, examining challenges like numerical precision, quantization, and the use of specialized hardware blocks. Experimental results demonstrate the potential of FPGAs to outperform traditional methods in specific scenarios, highlighting the importance of tailored hardware design for neural network inference. This work sets the stage for further optimization and application of FPGAs in AI, potentially extending to larger and more complex neural network models.

Problem Statement

With the recent rise in AI, hardware accelerators are in high demand. GPUs are the most common approach, but field programmable gate arrays (FPGAs) offer some unique advantages that may allow for higher efficiency and parallelism. The biggest advantage is great reconfigurability, which allows for certain functions such as weight multiplication to be implemented in hardware rather than software. An ideal FPGA accelerator for neural networks would be able to take into account the architecture of the network, building each unique operation into a hardware configuration or, assuming there is sufficient space on-chip, entire layers as hardware.

Introduction

For anyone unfamiliar, FPGAs are composed of versatile sets of logic blocks, containing registers, LUTs, integer adders, and multiplexers - all the necessary building blocks for digital

circuits. These logic blocks are interconnected by configurable traces known as FPGA fabric, which allow multiple logic blocks to work together to form larger structures within the FPGA. FPGAs are sometimes augmented with the inclusion of DSP blocks or block RAM, more specialized blocks that can perform particular functions that would otherwise be impractical to create using exclusively logic blocks. By writing Hardware Description Language (HDL), a designer can create any number of digital circuits for a particular task, allowing for highly parallel and incredibly fast computation. This is in comparison to CPUs, GPUs, or ASICs, which have a static hardware configuration and are made to do useful things by instructing the hardware with the use of software instructions. While the fabric of an FPGA introduces additional overhead compared to a static hardware architecture, there are many advantages such as over the air bug fixes, little to no instructional overhead, and supreme versatility.

The primary goal of this research is to implement a software framework for creating HDL designs that perform high-throughput, low latency, and power efficient inference for small neural networks. The choice to focus on exclusively inference is not due to any fundamental constraints of FPGAs, it is due to limited time and the availability of existing software tools for training neural networks.

FPGAs as Neural Network Accelerators

Workloads for neural networks are highly parallel and often consist of layers with similar core architecture, but varied configurations of these core building blocks. Inference and training times must be kept to a minimum, and in embedded or mobile applications, power usage is critical. For these reasons, FPGAs have the potential to be faster and more efficient than existing methods. Because the workload for a particular network architecture is static, it can be mapped to hardware and ran in a particular configuration. This would allow an FPGA to run any neural network that can meet it's constraints entirely in hardware, rather than as a set of software instructions that make use of parallel hardware for particular portions of the computation. In the following section, we will cover previous attempts to implement an FPGA neural network, including common pitfalls and the inherent limitations of both neural networks and FPGAs. As overall performance and power efficiency are the goals, optimization of neural network architecture and clever use of hardware design will be the crux of making this work.

Existing Methods/Lit Review

2006 FPGA Implementations of Neural Networks

This book, published in 2006, serves as a collection of important research papers regarding FPGA implementations of neural networks in the years preceding it's publishing. As both FPGAs and neural networks have been extensively researched, it made sense to start with earlier research papers, and build up to more modern developments. The book serves as a great starting point for anyone interested in neural network theory or FPGA design, as it goes into detail about the fundamentals of neural networks, the backpropagation algorithm, and activation functions as well as detailing multiple implementations of neural networks for FPGA. Chapter 1 discusses these topics, as well as the tradeoffs of FPGAs as a whole, including

reconfigurability and power loss due to such reconfigurability. Chapter 2 is details the tradeoffs between integers, fixed point, and floating point numbers, each of which are stored and calculated differently in hardware. When dealing with the training of a neural network, it is important to consider that derivatives and gradients are used for computing the next best set of weights, and eventually converging on the “best” parameters for a network. When quantizing or rounding numbers in order to improve performance in hardware, this loss of precision can pose a problem for training, exacerbating the vanishing gradient problem. The rest of the book covers specific optimization techniques and implementations from research papers, for both training and inferring neural networks on FPGAs. These implementations include training on an FPGA using the backpropagation algorithm, best match association problems, self-organizing feature maps, and fully and partially connected multilayer perceptron networks.

2017 A Survey of FPGA-Based Neural Network Inference Accelerator

This paper serves as a summarization of recent efforts to accelerate inference with FPGAs. As convolutional and recurrent neural networks become effective at solving problems such as image and speech recognition, the need to accelerate them becomes larger. This paper primarily focuses on these types of networks, but the optimization methods and constraints of FPGAs are mostly the same as other network architectures. Mentioned as one of the benefits of FPGAs when compared to GPUs, ASICs, and CPUs, FPGAs can often implement equally accurate neural networks with significantly less hardware. Because of this, the power used by a good FPGA design is often less than the alternatives, even with the additional overhead caused by the reconfigurability-enabling hardware. The biggest limiter in FPGA design is the existing software stack. HDL languages have been mostly stagnant compared to the rapidly evolving software space, and existing NN development software like Tensorflow or Caffe does not support FPGA acceleration. The other large limitation is the memory requirements of modern large network models, which often require 100-1000MB of parameters. The largest FPGAs at the time of this paper’s publication cap out at 50MB of SRAM on-chip. This requires the use of off-chip memory such as DDR SDRAM, which limits throughput, increases latency, and consumes more power. There are also some calculations which are required in small part for either communication with the host or initialization of the network, which do not lend themselves well of FPGA, as they take a large part of the chip area of an FPGA without contributing much to the parallel operation during inference. This is often mitigated with the use of a traditional CPU to augment the operations of the FPGA itself, with some chips even integrating both units into a single chip.

The primary optimization methods discussed in this paper are linear quantization, non-linear quantization, and weight reduction. All optimization methods serve to reduce the total data size of the parameters of a network, with quantization reducing the size per parameter, and weight reduction reducing the number of parameters. Linear quantization makes use of either integer or fixed point numbers rather than the typical floating point representation - this is for two reasons. The first is to reduce the complexity of hardware which is required at runtime to perform arithmetic operations on the model. This reduces area, energy use, and the total number of transistors a given unit of data must travel through, which contributes to overall throughput and latency. The second reason to use linear quantization, as mentioned previously, is to reduce the overall size of a given parameter. Nonlinear quantization is more complex, and

requires close integration with the training of the model to decide which values within a particular neuron are the most important, and assigns these values to a LUT. By eliminating values which are close enough to others, and not optimizing out the values which must have their precision in full resolution, a model's weights can be compressed to a much smaller size, and in some cases this LUT can be integrated with a neuron's activation function LUT. Quantization is an essential part of FPGA computations, and understanding the set of possible inputs and outputs as well as the tradeoffs between error and system accuracy allows for the best possible performance.

Weight reduction is a method used by well-optimized models for both FPGA accelerators and CPU/GPU/ASIC accelerators. It relies on either low rank matrix representations of the parameters of a model, or pruning. Low rank matrix representations allow for a large matrix to be compressed to two smaller matrices, reducing the data size with no loss of precision, and only a small runtime performance hit. Recent models such as QLoRA have made use of this technique to great success, as well as the paper being discussed. Pruning makes note of the parameters which are near 0, and if they are deemed unnecessary during training, they are removed entirely. This is similar to quantization in that the model loses some accuracy, but is a more aggressive optimization strategy, and can be more effective for reducing the size of a model in some scenarios.

2018 FP-BNN: Binarized neural network on FPGA

Deep neural networks, which make use of a much larger number of neurons, can be extremely difficult to both train and perform inference on due to the vast number of operations and parameters necessary. This paper discusses a particularly intense quantization method that reduces most or all layers of the network to binary representations, with neurons either being entirely on or off during inference. This reduces the effective bit widths to 1, which also allows for incredibly simple multiplication and addition. Weights and biases of each neuron are constrained to either 1 or 0, and activation functions are entirely unnecessary due to the simplicity. The implemented networks in this paper are an MLP trained on the MNIST dataset, CIFAR10 CNN, and the AlexNet CNN. Models were tested on CPU, GPU, and a Stratix V FPGA. Each of these networks were able to be sped up significantly with varying losses in accuracy. The most effective model for binary quantization was the MNIST MLP, with an accuracy of 98.24% compared to the original accuracy of 98.7%. It's model size was reduced by 30x, and it's performance was 20x faster than that of a comparable GPU. Even for the least effective binarization, AlexNet, it's performance, energy efficiency, and size were greatly improved. These results show promise for both accelerating on FPGAs, as well as for reducing model complexity as a whole. The biggest limitation for this type of quantization is that it is not very effective for networks with very small numbers of neurons. The other flaw with this type of optimization is that it takes much longer to train a model when heavily optimizing, as it requires many more epochs and functional checks beyond just verifying that the model is accurate when using fixed point calculation.

2020 A survey of FPGA-based accelerators for convolutional neural networks

This paper discusses various methods for optimization of CNNs on FPGA. Starting with the network itself, there are a couple optimizations discussed, including fusing layers, binarization, and pruning. Fusing layers relies on the idea that some layers do not necessarily need to be done in order, and that the combination of two layers allows any accelerator to avoid data transfers. As some layers perform the same convolutional filters on slightly offset locations of the original input data, these layers can be combined to make use of the similarities in the computations rather than performing the same computation twice. Binarization and pruning, as discussed earlier, can allow for further model compression and optimization. Pipelining, loop unrolling, and prefetching are all techniques that were initially developed for use with CPUs, and have made their way into most digital architectures. Pipelining makes use of the fact that not all sections of a given hardware design are in use at a single time for a single operation. Similar to a washer and dryer unit, where an operator can wash one load of laundry while another is drying, pipelining allows for increasing throughput without increasing area usage. Loop unrolling relies on the fact that many computations need not be done in order, and allow for massive parallelism by splitting a single group of computations across multiple units within an accelerator. This comes at the cost of increased area usage, but throughput and latency can be reduced linearly with the area usage. Prefetching allows for no time to be wasted waiting for data transfers, instead preparing the data in advance for use by a computational unit. One example of this technique is double buffering, which allows for off-chip memory to be read and buffered during inference, such that the memory transfer can be done in parallel to the processing of the previous layer. This requires more FPGA area for the buffer, but can minimize the impacts of memory transfers and allow for much larger models to be accelerated.

(2016) Going Deeper with Embedded FPGA Platform for Convolutional Neural Network

This paper expands on previous research with FPGA acceleration, as well as CNN theory. A big takeaway from this paper is that for CNNs the largest impact on space complexity (the amount of data required to store a model) comes from the fully connected layers, which require large numbers of parameters. On the other hand, time complexity (the amount of time it takes to compute the result) is primarily effected by the convolutional layers, which share parameters, but take much more time to compute. For this reason, it might be important to optimize these layers differently. This paper also highlights the importance of differently quantizing the layers of a network, as precision is not equally needed across layers. As shown in Table 3 of the paper, model accuracy is greatly effected by the quantization choices, particularly in the edge cases. By making use of dynamic precision with fixed point numbers, they were able to achieve the best results, often within a percent or two of the floating point models. Because they used a Zynq FPGA and CPU combination chip, they elected to stick to 8 or 16 bit quantization levels, as these make the most sense when working with a static architecture CPU.

(2016) ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network

This research paper discusses a project, in which the goal was to create both an FPGA based accelerator and a CNN in tandem, in order to optimize both the network architecture and hardware architecture together. It was implemented on a Xilinx Zynq FPGA, and made use of HLS tools. HLS tools have not been mentioned in the previous papers, and their primary purpose is to simplify the process of hardware design by abstracting further than HDL, and make use of tools that are common in higher level programming languages like C. This enables the hardware developer to focus on higher level abstraction, rather than implementation details. In order to select a CNN to start development from, the following characteristics were used to compare differing architectures: Computational Complexity, Regularity, Purely Convolutional, and Accuracy. The idea was that for an initial implementation, complexity should be kept low, regularity allows for a consistent hardware layer architecture, purely convolutional layers do not include fully connected layers (which are highly space-consuming), and of course a network should be accurate. A custom tool, Netscope, was written to visualize the complexity of various networks built in the Caffe software. At the start of their implementation details, they discussed the choice of data types, and mentioned the limitations of floating point numbers. However, they eventually decided to use single precision floating point numbers to retain compatibility with Caffe models, at the cost of performance and FPGA area. This paper also mentions the option of using an FFT to perform convolution, rather than the intuitive sliding layer approach - they ultimately did not implement it, but the option is an intriguing one, as it has the potential for large speedups. Detailed discussion of the use of HLS tools has shown that there is a unique learning curve, as added layers of abstraction can cause confusion to the programmer - albeit with large potential for decreased development time. Their suggested workflow includes making use of object oriented features, in a similar way that one would when writing HDL. By splitting discrete computational modules into their own blocks rather than writing the entire algorithm as a for loop and expecting HLS to figure it out, HLS performed much better in the task of compilation to a hardware design. Looking at their results, their FPGA utilization shows that DSP cores and LUTs are the largest limiting factor for FPGA resources, with flip flops being abundant in comparison. Their results are lackluster, with a very high inference time, but their proposed methods for increasing performance are promising, with a projected increase of 60 times over their final results with minimal effort, including increased clock speed, minimal bugfixes, and a switch to fixed point representations. As for power consumption, they were also held back by bugs and floating point format, but with the same proposed methods as before, they projected that power consumption per inference could be improved drastically.

Implementation

For this implementation, a simple software stack is used for describing a model, training a model, applying optimization features, and exporting the model description to existing HDL code that will implement the model. Tensorflow and Keras are used for the initial training and verification of the model, as well as providing a best-case output for the quantized network. Once a model is described and trained, Python scripts are used to export the model information to the HDL design, storing quantization levels, weights, biases, and activation functions as a set of LUTs or in Block RAM depending on the size of the quantized model. Network size is limited to ~800 neurons on a Basys 3 development board when using signed Q4.6 numbers for all model information. For higher precision, more neurons can be implemented. There are no hard limits on the quantization levels, unlike a GPU or ASIC acceleration solution which will typically only have support for FP32 or FP64 arithmetic and weights. This allows for maximum quantization for a given performance target. Xilinx Vivado was used for all synthesis and implementation, mapping the HDL design to the FPGA logic blocks.

The choice of Tensorflow for model training was due to it's support for quantization aware training, in which each successive change to the model weights and biases quantizes them to a given accuracy, unlike MATLAB, which only supports this feature within it's Deep Learning toolbox. Unfortunately, Tensorflow's Model Optimization features are designed for software solutions or embedded applications, in which inference is limited to 8 bit, 16 bit, or 32 bit integer / fixed point math, or alternatively 16 bit, 32 bit, and 64 bit floating point math. This loses out on one of the big benefits of FPGAs, and that is the completely configurable bitwidth of each parameter. Because of this, quantization aware training was not used. An ideal implementation would be able to target a particular FPGA and error margin, and calculate the best bitwidth and weights for each layer of the network, given the area and error constraints. By tailoring training to the hardware, the best overall result can be achieved. This also goes further than training, as neural network architecture is highly variable in it's parallelization and tolerance to low precision arithmetic. The perfect implementation is a function of all of it's parts - algorithm, software, and hardware.

Testing Methodology and Performance Metrics

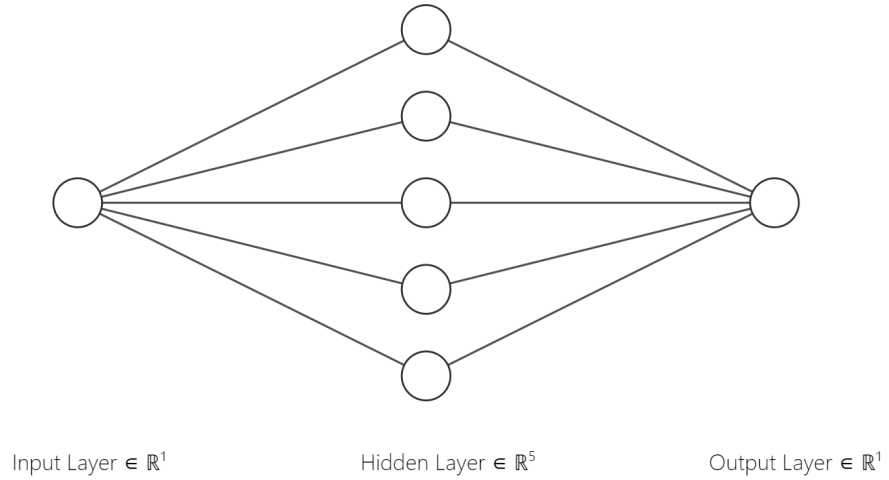

Figure 1: Test Network Architecture

Testing methodology consisted of training a simple feedforward network with a single hidden layer composed of 5 neurons [figure 1]. The model was trained on the function xe^{-x}, with 5000 equally spaced points between $x = 0$ and $x = 5$. This model was trained in Tensorflow using Keras as a backend, in 50 epochs using Nadam, a variant of Adam that makes use of Nesterov momentum. After training, the model was verified with 100 points from the input space. The eventual parameters, including weights, biases, and the sigmoid activation function are included with the project package. This model was evaluated in standard floating point format, as well as with weights and biases quantized to the nearest fixed point signed Q4.6 representation. The results of the preliminary evaluation [figure 2], show that the model should produce results that track the function well, considering that the fixed point Q4.6 output can only represent 24 values between the function's domain of 0 to 0.368. In retrospective, it is not necessary for the output resolution to be the same as the input resolution, and as such, the neural network's accuracy could be increased for the same Q4.6 input space.

Once a preliminary model was created and evaluated as a suitable test platform, the hardware implementation began. Working in SystemVerilog, a basic set of fixed point arithmetic operations were created, accounting for the fact that multiplying two fixed point numbers requires a right shift equal to the number of their fractional bits. Once addition and multiplication were verified, some Python code was used to create a hex file that would be used to instantiate the activation functions LUT in hardware. Now that we have the core blocks of a neuron, a hardware neuron could be created. A focus was put on ensuring that the code for a neuron was robust, including checks for arithmetic overflows, as well as parameterizable code that allows for any activation function or number of inputs to be used. By parameterizing the code, any number of neural network architectures could be implemented with little additional effort. Once a single neuron was built, the neural network [figure 1] was implemented, and a testbench was assembled to test the entire input space and save the results of the simulated design. Our results [figure 3] show that the hardware implementation and the preliminary evaluation match

closely in accuracy, with both outputs being slightly lower than the desired function. Given that our goal was to create a hardware architecture that accurately models a neural network, these results are expected, and a good sign.

The overall throughput and inference time of the system was evaluated with a clock period of 10ns, and with each inference only requiring 20ns to evaluate, our throughput is 50 MHz. This is a conservative value as well, as with further tuning or hardware chip binning, greater frequencies could be achieved. Power consumption for the FPGA could be measured, but a good comparison would be hard to make, as running a neural network of this size on a GPU would be completely overkill for the application. It is better to consider this as a proof of concept than as a fully fledged competitor to Nvidia or AMD.

Based on the utilization of this design [figure 4], it could scale up to ~800 Q4.6 neurons, and by sharing activation function LUTs, it could be increased to as many as 4000 neurons. Considering that the MNIST dataset has been evaluated with as little as 1306 neurons to a high degree of accuracy (albeit in floating point), that leaves much room for any necessary interfacing logic, and it would be a feasible extension of this project.

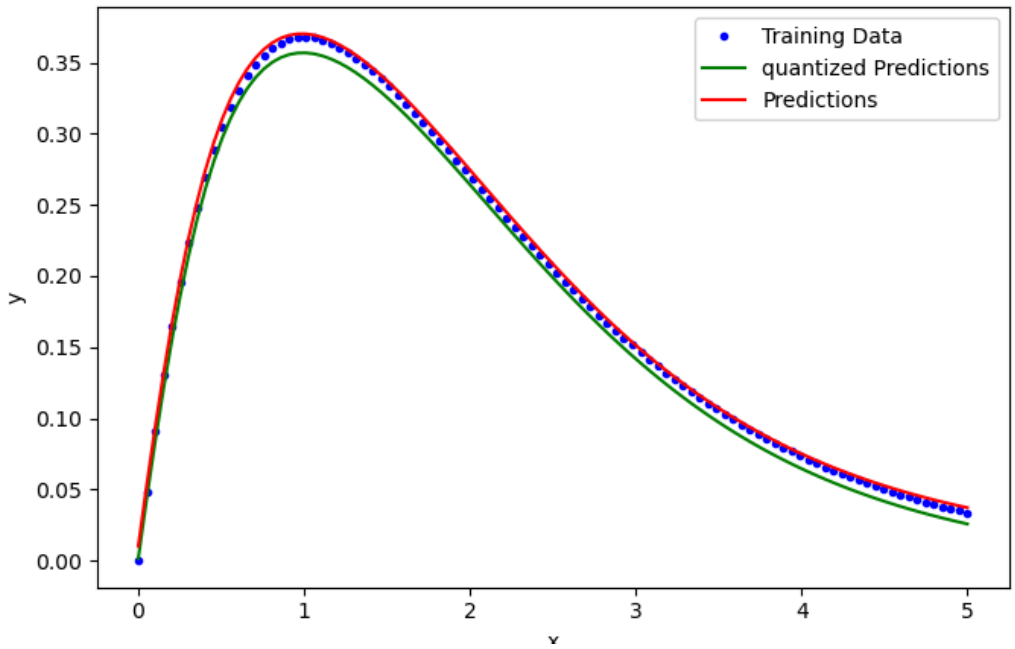

Figure 2:
Preliminary evaluation of the quantized weights and biases, performed with floating point arithmetic. Compared with the training data derived from the function, and the unquantized model.

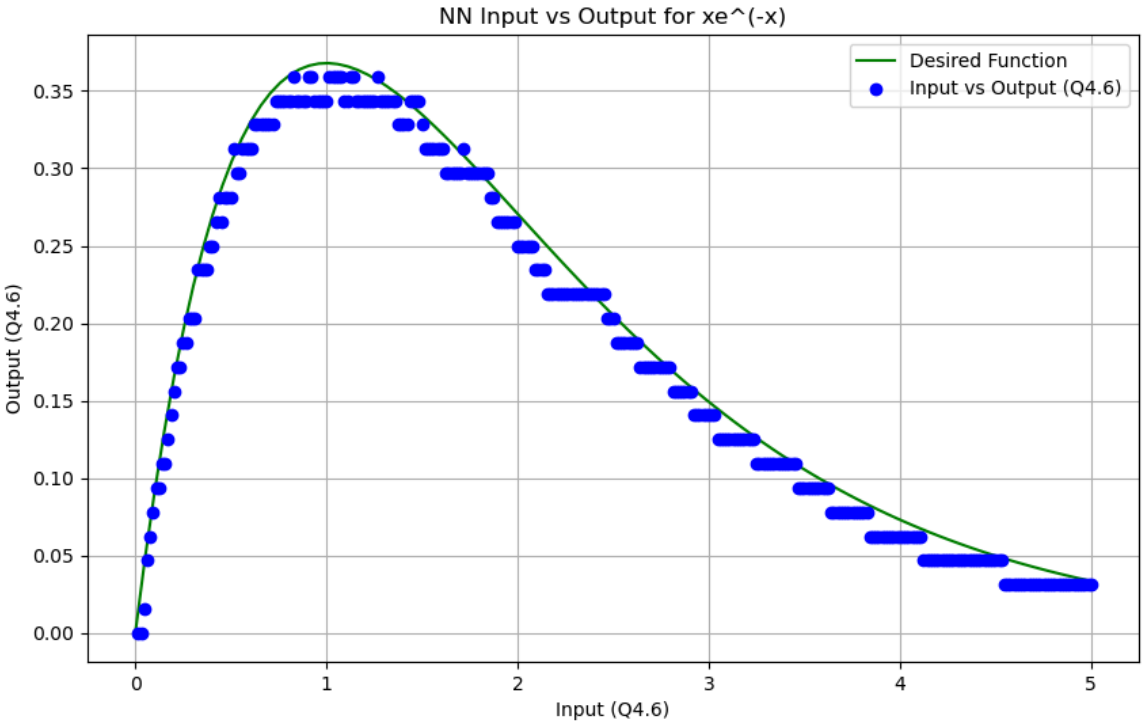

Figure 3: The hardware design's outputs, plotted with the desired function

Resource	Utilization	Available	Utilization %
LUT	111	20800	0.53
FF	10	41600	0.02
IO	41	106	38.68

Figure 4: XC7A35T-1CPG236C (Basys 3 development board) FPGA resource utilization

An Ideal Implementation

Many of the previously discussed methods for further optimization would serve this project well. I think a great place to start would be larger networks, to ensure there are no existing bugs. Beyond that, implementing a convolutional layer would be a good way to modernize the network architecture, and make it more useful. An obvious addition to convolutional layers would be binarization support, which would have a huge potential for running much larger models on the FPGA. Implementation of any large model that is not heavily quantized would necessitate the use of off-chip block ram, which would be an interesting challenge. With this, double buffering, caching, and pipelining, would make sense as these techniques all become more essential as the number of layers in a network goes up. I would like to try using HLS tools to create more complex designs, but for now I think I have much to learn about SystemVerilog before stepping to a more abstracted tool.

Conclusion

Overall, the results of the implemented design are very promising, as this forms the basis for any future FPGA projects or neural network optimization I will work on. There is much room for optimization, as the size of the implemented network is very small. With more effort and time, this could grow to be a useful tool for understanding the effects of various neural network optimization techniques as well as FPGA implementation options. Given significant investment, FPGAs and their supporting software could continue to evolve to become even more suited to neural network acceleration.

References

Gschwend, David. “ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network.” *ArXiv (Cornell University)*, 1 Jan. 2020, https://doi.org/10.48550/arxiv.2005.06892. Accessed 10 June 2024.

Guo, Kaiyuan, et al. “[DL] a Survey of FPGA-Based Neural Network Inference Accelerators.” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 12, no. 1, 28 Mar. 2019, pp. 1–26, https://doi.org/10.1145/3289185. Accessed 15 May 2023.

Liang, Shuang, et al. “FP-BNN: Binarized Neural Network on FPGA.” *Neurocomputing*, vol. 275, no. 275, Jan. 2018, pp. 1072–1086, https://doi.org/10.1016/j.neucom.2017.09.046. Accessed 17 Aug. 2020.

Mittal, Sparsh. “A Survey of FPGA-Based Accelerators for Convolutional Neural Networks.” *Neural Computing and Applications*, vol. 32, no. 4, 6 Oct. 2018, pp. 1109–1139, https://doi.org/10.1007/s00521-018-3761-1.

Omondi, Amos R, and Jagath C Rajapakse. *FPGA Implementations of Neural Networks*. Springer Science & Business Media, 4 Oct. 2006.

Qiu, Jiantao, et al. “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network.” *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 21 Feb. 2016, https://doi.org/10.1145/2847263.2847265.