Image Classification on FPGAs
ECE 285

Anshul Devnani
ECE
A59010615

Abstract

Machine learning algorithms have been widely adopted in various fields such as computer vision, NLP, speech recognition, etc. One of the key challenges of deploying such algorithms is the need for efficient hardware acceleration to meet performance requirements. FPGAs have emerged as a promising platform for accelerating machine learning algorithms due to their flexibility, parallel processing capabilities, low latency and power consumption. In this work, I map multiple machine learning models onto the PYNQ-Z2 FPGA using the HLS4ML python library and compare resource consumption and power between each model as well as compare latency between the same model running on the FGPA Hardware and a Raspberry PI CPU.

1 Introduction

With IoT devices and computer vision algorithms becoming increasingly popular, there are many applications that can benefit from integrating the two technologies in domains such as smart surveillance, autonomous vehicles, healthcare etc. The main goal is to have quick inference on the edge device. However two main problems exist:

- Computer vision algorithms are computationally expensive and power hungry.
- The majority of CPU's implemented on IoT devices are not ideal to run these computationally expensive algorithms as the latency is too large and takes a huge amount of power.

In order to mitigate these two issues, there has been an extensive amount of research that deals with how the expensive computation of machine learning models can be offloaded to FPGAs to solve the issue of latency and power consumption.

The FPGA used in this project is the Zynq®-7000 Artix™-7 FPGA which sits on board the PYNQ-Z2 development board.

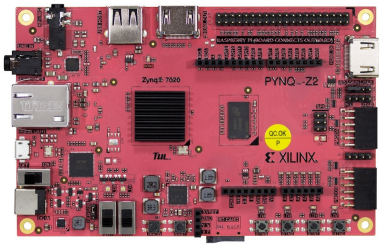

Figure 1: PYNQ-Z2 board

The main problem with this FPGA is the amount is the storage capacity, image classification require an abundant amount of space if you want to achieve a decent accuracy. Therefore, low power / embedded

machine learning techniques such as network pruning and weight quantization are explored in order to mitigate this problem.

In order to implement algorithms to run on a FPGA, some sort of Hardware Description Language (HDL) must be used. However, writing machine learning algorithms in a HDL is complex and time consuming. Therefore, this project relies heavily on the use of the HLS4ML python library. The HLS4ML python library is a Python package for machine learning inference in FPGAs, it allows us to write machine learning algoritms in pure python using popular frameworks such as Tensorflow/Keras or Pytorch. HLS4ML compiles our trained models into high level synthesis code ie. C code and then into HDL using popular synthesis tools such as Xilinx Vivado or Vitis HLS. In this project, the machine learning framework used is a combination of Tensorflow, Keras, and Quantized Keras (qkeras) and the HLS tool used is Vivado 2019.2.

Subsequent sections will dive into more details about the experiment results. In summary, we train 5 models, making minor tweaks to each model, a measure/compare FPGA resource consumption, power usage and compare latency/throughput of results on FPGA hardware and on a Raspberry Pi CPU.

2 Related Work

There are many research examples of mapping of machine learning algorithms to FPGAs. Unfortunately I could not find any papers specifically relating to mapping FPGAs onto the PYNQ-Z2 board using the HLS4ML package, however there are a couple presentation PDFs of researchers doing the same thing. These PDFs are mentioned in the References section. They are similar to my work in the way that they both aim to map image classification algorithms onto the PYNQ Z2 but do not dive into the specifics (ie. code) in order to achieve this.

3 Method

The method taken to implement this project can be summarized by the figure below.

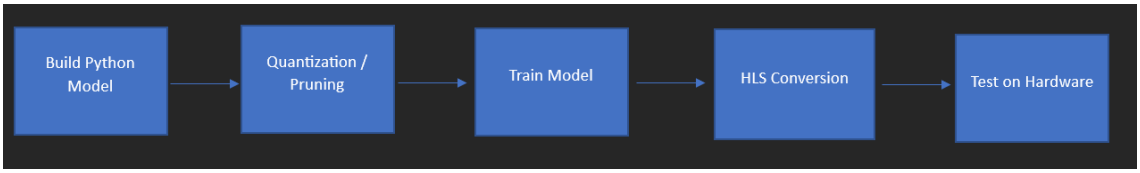

Figure 2: Project Method

All steps shown above are covered in the demo video however more details for each step is shown below.

3.1 Build Python Model

A total of 5 python models were built using Tensorflow and Qkeras. Minor changes to each model such as quantization and number of filters are made to compare the effect of resource usage and latency on hardware however the overall network structure is kept mainly the same between models. They are summarized below.

1. Model 1 (Baseline)
   (a) General Model Architecture: 3x Conv -> Relu -> Max Pool, 3 Dense Layers, Softmax Output
   (b) Number of filters per Conv Layer: 8, 8, 16
   (c) Number of neurons per Dense Layer: 42, 64, 43
   (d) Quantization: Fixed Point 12 bit
2. Model 2 (change number of conv filters)

(a) General Model Architecture: 3x Conv -> Relu -> Max Pool, 3 Dense Layers, Softmax Output
(b) Number of filters per Conv Layer: 2, 4, 8
(c) Number of neurons per Dense Layer: 42, 64, 43
(d) Quantization: Fixed Point 12 bit

3. Model 3 (change quantization)
   (a) General Model Architecture: 3x Conv -> Relu -> Max Pool, 3 Dense Layers, Softmax Output
   (b) Number of filters per Conv Layer: 8, 8, 16
   (c) Number of neurons per Dense Layer: 42, 64, 43
   (d) Quantization: Fixed Point 4 bit
4. Model 4 (reduce filters and quantization)
   (a) General Model Architecture: 3x Conv -> Relu -> Max Pool, 3 Dense Layers, Softmax Output
   (b) Number of filters per Conv Layer: 4, 4, 8
   (c) Number of neurons per Dense Layer: 42, 64, 43
   (d) Quantization: Fixed Point 6 bit
5. Model 5 (modify the network architecture slightly)
   (a) General Model Architecture: 4x Conv -> Relu , 1 Dense Layers, Softmax Output
   (b) Number of filters per Conv Layer: 4, 4, 4, 4
   (c) Number of neurons per Dense Layer: 43
   (d) Quantization: Fixed Point 6 bit

Detailed images of the network architecture are provided below. Figure 3 is actually model 1 architecture, however, the only difference between model 1 through model 4 is the number of convolution filters and/or the quantization, therefore the other 3 model architectures are omitted to save space.

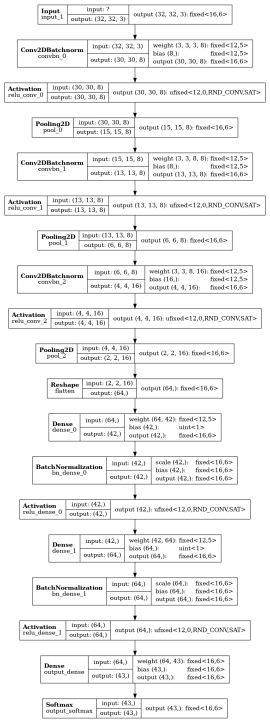

Figure 3: Model 1 - 4 Architecture

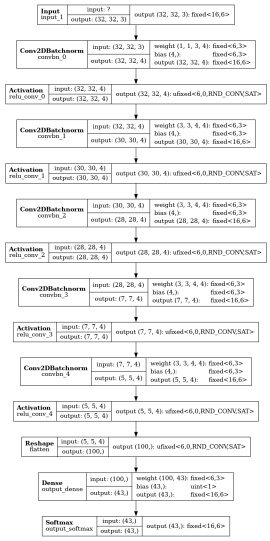

Figure 4: Model 5 architecture

3.2 Quantization/Pruning

As mentioned in earlier sections the PYNQ Z2 FPGA has limited capacity so in order to fit out models techniques such as quantization and pruning must be implemented. Quantization and pruning are two techniques commonly used in machine learning to reduce the size and complexity of models, making them more efficient in terms of memory usage, computational requirements, and deployment on resource-constrained devices.

Quantization is the process of reducing the precision of numerical values in a model. In machine learning, most models are trained using floating-point numbers, which require more memory and computational resources compared to fixed-point or integer representations. Quantization involves converting these floating-point numbers into fixed-point with a lower bit-width.

Pruning is a technique used to remove unnecessary or redundant connections (weights) in a neural network. During training, neural networks tend to have many connections that are not contributing significantly to the overall accuracy of the model. Pruning aims to identify and remove these connections, resulting in a sparser network with fewer parameters.

In this project quantization is achieved by using Qkeras, a framework like Keras which gives us access to layers like "QConv2d", "QDense", "QConv2DBatchnorm", "QActivation", that allows us to specify the level of weight quantization. See the models.py file for the implementation details. Pruning is setup to try and achieve a sparsity of 50 percent across all models. For example, the following graph shows the distribution of weights for Model 5, it is shown that 50 percent of the weights are set to zero.

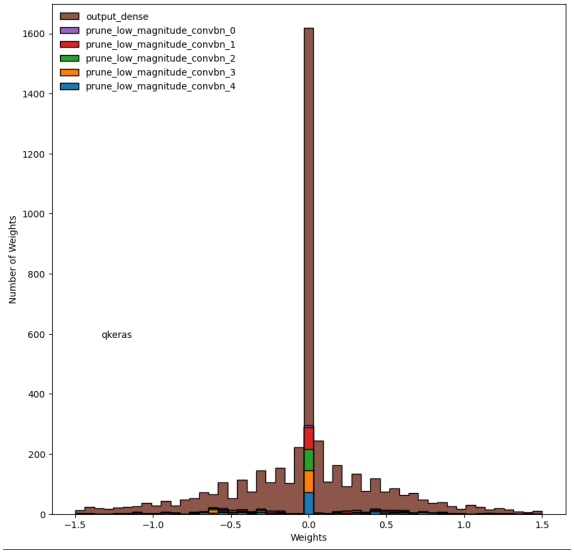

Figure 5: Weight Distribution for each layer in Model 5

See the pynq.ipynb file for implementation details for quantization and pruning.

3.3 Train Model

Now that the models are built with quantization and pruning in mind, it is time to train them. These models are trained on the German Traffic Sign Recognition Benchmark dataset (GTSRB). This dataset contains more than 50,000 total images of different German road signs belonging to more than 40 total classes. The data format for the images are png files and class label mappings are given in corresponding csv files. The images in this data set are of different sizes but are resized to 32x32 RGB images and then normalized.

I wanted to have some consistency when training each model therefore the hyperparamters are kept the same when training each model and is shown below.

1. Number of EPOCHS = 30

2. Loss function = Categorical Cross entropy
3. Optimizer = Adam
   (a) Starting LR = 3e-3
   (b) beta 1 = 0.9, beta 2 = 0.999, epsilon = 1e-7

In addition, some callbacks were implemented to help with the training process.

1. Early Stopping - if the training accuracy has not increased for 10 epochs, stop the training
2. Reduce LR on Plateau - if the val loss has not increased for 3 epochs, reduce the learning rate by the factor of .5
3. Update Pruning Step - This callback does the model pruning during training

Train and validation accuracy and loss graphs for each model is shown below.

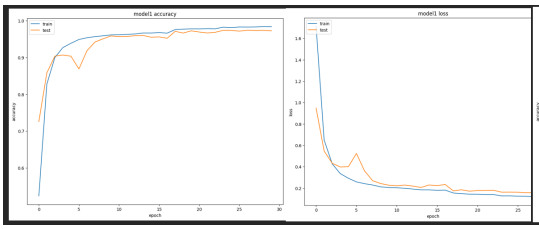

Figure 6: Model 1 Training Accuracy and Loss

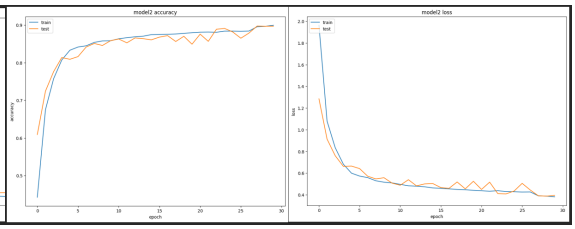

Figure 7: Model 2 Training Accuracy and Loss

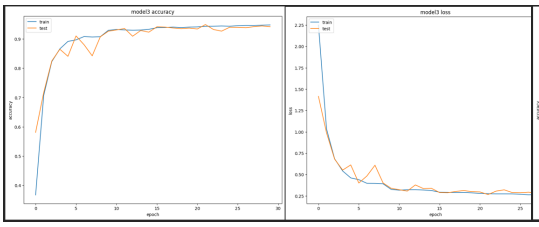

Figure 8: Model 3 Training Accuracy and Loss

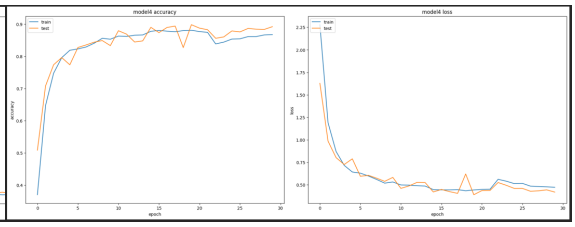

Figure 9: Model 4 Training Accuracy and Loss

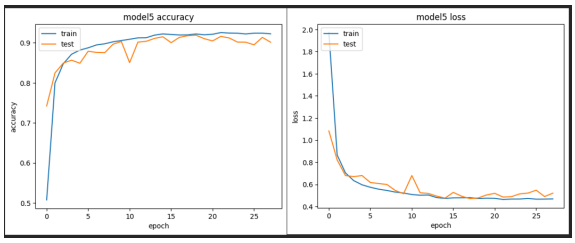

Figure 10: Model 5 Training Accuracy and Loss

3.4 HLS Conversion

Now that the models are trained, it is now time to use the HLS4ML python library to convert out pure python models to HLS code. The HLS4ML python library handles all this for us which helps immensely with prototyping time. In order to convert our python models to HLS we must specify a minimum configuration detailed below:

1. Reuse Factor - this is a parameter that determines the number of times each multiplier is used in order to compute a layer of neuron's values. A low reuse factor leads to lower latency but higher resource usage, while higher reuse factor leads to higher latency but lower resource usage. For the FGPA on board the PYNQ-Z2 it is recommend to set this to 64

2. IOType - this parameter specifies the type of data structure used for inputs, intermediate activations between layers, and outputs. This is set to "io stream" which are a more efficient/scalable mechanism to represent data that are produced and consumed in a sequential manner, ie processing image data.

3. KerasModel - this parameter must be set to the keras python model

4. Clock Period - this pareameter specifies the default clock period that will drive the FPGA, this is set to 10ns in this case.

5. XilinxPart - this parameter is set to the board number, in this case it is set to xc7z020-clg400-1

After specifying all these parameters, we can use the HLS4ML API to convert our python code to HLS code and create Vivado project files.

Before calling the HLS4ML API to actually generate the necessary files to load onto the board, (ie. perform synthesis) it is a good idea to check the accuracy of the converted model with the test dataset and compare it to the python model test accuracy. If both match, we can call the HLS4ML API to start generating a bitstream to load onto the FPGA, this will use the Vivado application to run synthesis and generate all the necessary files to load onto the board.

Figure 11 below is generated during the HLS conversion process, this is a block diagram that sets up the communication via the CPU (processing system) onboard the PYNQ-Z2 and the FPGA (programmable logic) on the PYNQ-Z2. In summary, an DMA is setup between the CPU and FPGA to stream in our test image data to the FPGA.

Figure 12 below shows the final routed and placement design for model 1. This is done during the HLS conversion process, more specifically in the implementation (after synthesis).

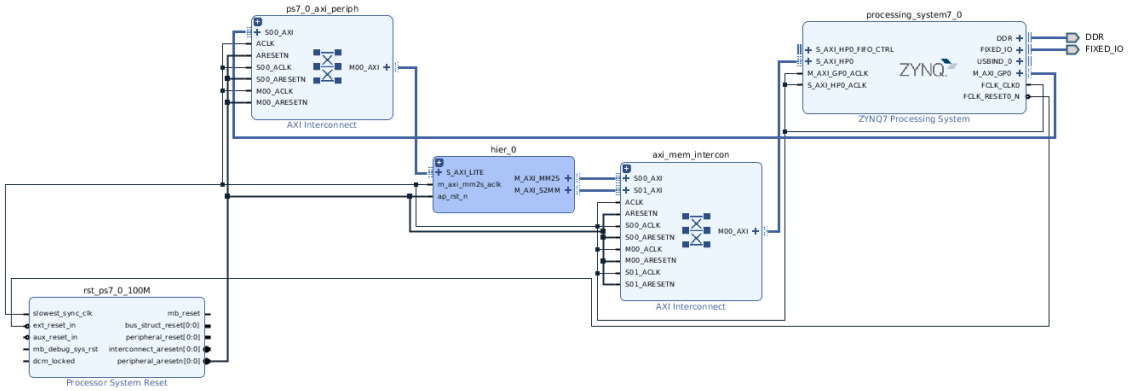

Figure 11: Block Diagram for Model 1

3.5 Test on Hardware

The last step is to actually test on hardware. Once the synthesis and implementation completes successfully, it is time to test on our target boards. All necessary files are loaded via SSH onto the pynq z2 board and run against the test set of 1500 images. Latency, throughput are measured in the process and the output numpy array is returned and loaded back onto the host machine to measure the test accuracy. This value should be very close to the value obtained in the previous section when profiling the model before actually running synthesis.

In addition to running the model on the PYNQ-Z2 board, the same model is run on the CPU of a raspberry pi. Latency and throughput is also measured in this case and compared to the FPGA case. Detailed comparison of actual values in done in the Experiment section

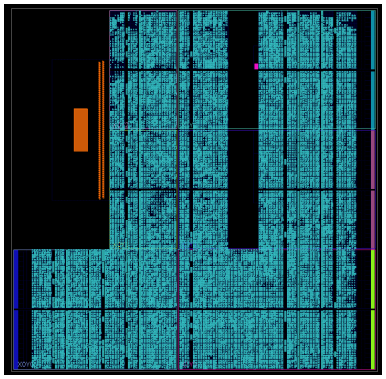

Figure 12: Final Routed and Placed Design for Model 1

4 Experiments

Each of the 5 models were trained on the same data set mentioned above: German Traffic Sign Recognition Benchmark dataset (GTSRB), with the same set of hyperparamters. The input into each model was normalized 32x32 RGB images.

The results of running each model on hardware and its resource usage, latency, and accuracy are summarized in the below table.

	Model 1	Model 2	Model 3	Model 4	Model 5
Resource Usage					
BRAM_18K	40.5	29.5	32.5	23.5	26
DSP	220	208	150	174	1
FF	50,581	41,829	40,812	38,906	25,782
LUT	38,381	26,696	32,021	27,805	19,488
Total Power FPGA (Watts)	2.054	1.949	1.871	1.864	1.627
Latency (s) - FPGA	0.574743	0.4209	0.5718	0.358	1.57
Latency (s) - CPU	3.5589352	3.209	3.591	3.293	4.117
Throughput (images/sec) - FGPA	2609.862	3562.97	2622.987	4179.413	953.223
Throughput (images/sec) - CPU	421.474	467.372	417.704	455.507	364.26
Final Test Accuracy (%) - HW	92.4	79.5	89.2	82.6	85.2
Final Test Accuracy (%) - SW	92.2	80.7	89.4	83.1	85.4

Figure 13: Results

Model 1 is the baseline model as mentioned in the previous section, this model features 3 convolution layers with filter sizes 8, 8, 16 respectively. This model also features a 12 bit quantization of weights. This model has the highest resource usage but also the highest accuracy both in pure python and in the HLS implementation. In terms of latency and throughput, it can classify 1500 images in .57 seconds on the FPGA and 3.55 seconds on the Raspberry Pi CPU which features a Cortex-A72 SoC. These latency values lead to a throughput of 2609.862 images / sec on the FPGA and 421.47 images / sec on the CPU. It is very clear that the latency and throughput measured on the FPGA is far better than running on the CPU in all cases.

Model 2 modifies model 1 by changes the number of convolution filters from 8, 8, 16 to 2, 4, 8. Doing this reduced the resource footprint on the FPGA which is expected when reducing the number of filters. It also makes sense that our latency/throughput is quicker/greater because of the lower number of filters used meaning less computation needed. The downside of this model is the final test accuracy which is the lowest of the 5 models. This is expected because the number of filters used in a CNN helps extract fine grain details from a image which leads to better classification accuracy, so if you lower the number of filters, intuitively, a lower accuracy will be obtained.

Model 3 reverts the number of convolution filters back to that of model 1 but now changes the quantization from 12 bit to 4 bit. We see a slight reduction in total resource consumption, with the majority of resource reduction in DSPs. However, there doesn't seem to be a big effect on latency and throughout on both the FPGA and CPU. The effect on final test accuracy is also minimal with only a 3 percent reduction.

Model 4 aims to combine both a lower number of filters as shown in model 2 to achieve better latency and throughput numbers as well as minimize the amount of quantization to reduce the resource footprint as shown in model 3. The number of filters is set to 4, 4, 8 and the quantization is set to 6 bit. By doing so, the resource footprint is lower than that of model 1, 2, and 3 as well as the latency (and higher throughput). In fact, model 4 has the lowest latency and highest throughput of all the models. Model 4 suffers when it comes to the final test accuracy as its about 10 percent lower than model 1 but it expected since we lower the amount of convolution filters and quantization.

Model 5 deviates from the 4 previous models by changing the model architecture a bit. Instead of 3 convolution layers, 4 are used all with 4 filters each. Instead of 3 dense layers, only one dense layer is used. In addition, 6 bit quantization is used. Comparing the resource usage with the model 4 (as it also uses 6 bit quantization), model 5 has the lowest resource usage. This is most likely due to the fact that there is only 1 dense layer. Model 5 mainly suffers when looking at the latency and throughput on both FPGA and CPU. It takes 1.57 seconds to classify 1500 images on the FPGA and 4.117 seconds on the CPU. This is because model 5 adds another convolution layer and mainly because the Reuse Factor, as described in the earlier section, is set to 64. However the final accuracy is decent considering the resource usage at about 85 percent.

In terms of total power usage, it is clear there is a direct relationship between resource usage and total power usage. The lower the resource usage, the lower the total power consumption.

All in all, when looking at all these models, I don't think one model is "better" than the other. I think it depends on your use case. If you care more about resource usage, model 5 is better. If you care about accuracy, model 1 is better. If you care about latency, model 4 is better. The purpose of this project was not to find the absolute best image classification model for our dataset but to provide a method of mapping image classification models onto a FPGA and explore how minor tweaks to a model can effect resource usage, latency, throughput, and accuracy.

References

1. F. (2023, April 14). GitHub - fastmachinelearning/hls4ml: Machine learning on FPGAs using HLS. GitHub. https://github.com/fastmachinelearning/hls4ml
2. Varma, R. (2021, January 29). Binary Neural Networks—Future of low-cost neural networks? Medium. https://towardsdatascience.com/binary-neural-networks-future-of-low-cost-neural-networks-bcc926888f3f
3. GTSRB - German Traffic Sign Recognition Benchmark. (n.d.). GTSRB - German Traffic Sign Recognition Benchmark | Kaggle. https:///datasets/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign
4. Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P., Jahre, M., amp; Vissers, K. (2017). Finn. Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. https://doi.org/10.1145/3020078.3021744
5. Yuan, C., amp; Agaian, S. S. (2023). A comprehensive review of Binary Neural Network. Artificial Intelligence Review. https://doi.org/10.1007/s10462-023-10464-w
6. "PYNQ Tutorial." Indico, indico.cern.ch/event/985266/.
7. Fast Convolutional Neural Networks on Fpgas with Hls4ml - IOPscience, iopscience.iop.org/article/10.1088/2632-2153/ac0ea1.