Resource and Performance Improvements of Optimized Convolutional Neural Networks for FPGA Implementations of Automatic Modulation Recognition

Joshua A Rothe
Whiting School of Engineering
Johns Hopkins University
Baltimore, MD, United States
jrothe1@alumni.jh.edu

Haya Shajaiah
Whiting School of Engineering
Johns Hopkins University
Baltimore, MD, United States
hshajai1@jhu.edu

Abstract—**Automatic Modulation Recognition (AMR), commonly found in software defined radios, relies on speed and accuracy to effectively interpret the modulation type of incoming signals. Convolutional Neural Networks (CNNs) are growing in popularity over traditional algorithms due to their excellent performance with classification-type problems – but these are typically resource intensive, and Radio Frequency (RF) receivers are typically part of a larger system that can benefit from less resources being tied to this classification task. Field Programmable Gate Array (FPGA) implementations provide better performance than CPU and GPU implementations in most cases, and the added benefit of these functions being off the processor allows the entire system to perform better. Since resource utilization is such a critical component of effective CNN implementation, and it is often inversely tied to performance, the effectiveness of various hardware optimization techniques as well as their effects on performance should be considered by the designer. This work evaluates the tradeoffs of various model optimizations and how they affect both implementation and performance, synthesizing the models using Xilinx's quantization-aware FINN library. In this work, a methodology is presented for the generation of I and Q signals for CNN model training and evaluation, and the performance and hardware utilization benchmarks are evaluated to determine both design considerations and effective approaches for optimizing these models for real-world implementation.**

Keywords—Field Programmable Gate Array (FPGA), Convolutional Neural Network (CNN), Quantization, Structured Pruning, Automatic Modulation Detection/Recognition (AMD/AMR), Rectified Linear Unit (ReLU)

I. Introduction

FPGA implementations of convolutional neural networks have been shown to be much faster and consume less power than CPU or GPU implementations. For applications such as automatic modulation detection, both speed and power usage are crucial to the performance of the system. A radio receiver must quickly decode an incoming signal, and it must do so reliably in remote areas where a small battery might be the primary power source. Since FPGA fabric is limited, hardware utilization is an important consideration for these functions, so a high-performing and resource-optimized CNN is an excellent choice for classification tasks such as AMR.

The work done in [1] used low-precision math and quantization, combined with a residual unit-based scheme and iterative pruning, to cut hardware utilization by 40%. It also achieved 527k classifications per second with a 7.5 μs latency. Pruning and quantization were also combined in [2], greatly shrinking the model size with no loss of accuracy. The smaller model size allowed Han et. al. to move the model from off-chip DRAM to on-chip SRAM cache memory, improving processing speed by 3-4x per layer. ReLU and Max Pooling layers were combined in [3] to reduce parameter size and give accurate results despite smaller model sizes, noting that a larger model does not always improve performance. In [4], C. Zhang et. al. found that, in FPGA implementations of CNNs, there could be as much as a 90% performance difference between two models of similar resource utilization; thus, the optimal implementation was non-trivial and not a direct function of model depth. And lastly, in [5], D. Góez et. al. explores the implementation of deep neural networks for automatic modulation recognition on FPGAs, using a one-dimensional CNN and quantizing the weights using the Brevitas library. The paper states that they would like to evaluate their methods in future work by incorporating both pruning and different model architectures. This research addresses both, continuing the work done in [6] and evaluates the tradeoffs of different optimization techniques for 1D CNN implementations on FPGAs.

This research uses the FINN library for synthesis onto FPGA fabric. The FINN library is an open-source tool developed by Xilinx for use with their FPGA hardware, and in [7] they demonstrated that the tool achieved the fastest classifications to date for CNNs implemented using its tool. The tool utilized time-multiplexing or folding, in which hardware was reused for various layers by storing the weights in memory and loading these weights depending on what layer the data was currently passing through. This approach allowed the designer to set a target throughput benchmark, and then the synthesizer would unfold the architecture as necessary (if hardware allowed) to reach this throughput value. In [8], the tool was improved, allowing the library to now take advantage of quantization more heavily for optimization. Float values were acknowledged as too large, often carrying redundant data. By supporting arbitrary precision on weights as well as input and output activation, it allowed designers to use quantization to optimize their models with significant improvements on FPGAs.

In the previous work [6], CNN models were generated using various architecture sizes, trained, and performance was evaluated on GPU using Pytorch. Here, the CNN architectures were modified to allow for a linear reduction in layer counts and pruning rates for a more consistent analysis, and weights were initialized from a consistent seed value to make performance and training more consistent across the different models. Separate signal datapoints were generated and used to evaluate the model, while adjusting the training parameters until model accuracy reaches at least 99%. Finally, FPGA performance and resource utilization was measured by synthesizing the model using the Xilinx FINN library. This data was then used to evaluate how quantization, pruning, and model depth affected CNN performance and resource utilization on FPGAs.

II. Methodology

A. Dataset Generation

Dataset generation was achieved in the same manner as in [6], by creating a siggen function that created labeled 'chunks' of I and Q pairs, with noise values added to simulate real-world conditions. Datatypes D_m were generated – amplitude shift keying modulation types (4ASK, 8ASK), phase shift keying (BPSK, QPSK, 8PSK, 16PSK), and quadrature amplitude modulation (8QAM, 16QAM, and 32QAM). The noise value (0.01, a dimensionless ratio) was chosen for both Additive White Gaussian Noise (AWGN) N_a and phase noise N_p. This noise value allowed for visible distortion on the modulation types while still allowing the modulation type to be discernable from the plots. An example of one such plot is shown in Fig. 1, for the 16QAM modulation type.

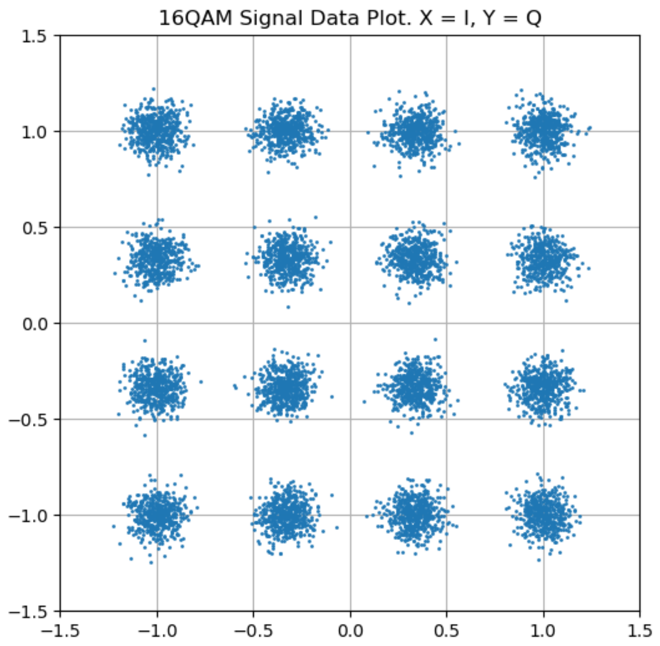

Figure 1. Generated I and Q Values for 16QAM.

This signal generator function was used to create a set of V_d quantity I and Q pairs, grouped into size c_s chunks. The data values were organized this way since the model would need to see a certain number of random datapoints to determine modulation type, as one normalized value by itself could be one of several modulation types. The output of this signal generator was a data frame D_f consisting of labeled data values of type D_m, each data values containing c_s I and Q pairs that represent a sampled and pre-processed signal. Algorithm 1 outlines the signal generation process executed after each set of I and Q values is generated, taken from the previous work in [6].

```
**Algorithm I:** CNN Dataset Generation
**Inputs:** Noise values $N_a$, $N_p$, chunk size $c_s$, quantity $V_d$.
**Output:** A data frame $D_f$ consisting of $V_d/c_s$ labeled,
            chunked datasets.
1.  Initialize qty $D_m$ list of siggen functions.
2.  $i = 0$.
3.  **while** $i \leq D_m$ **do**:
4.      Generate $V_d$ quantity I, Q pairs and shuffle.
5.      Create an empty list $L_c$.
6.      $n = 0$.
7.      **for** $j \leftarrow j + c_s$ **do**:
8.          Append qty $c_s$ I and Q pairs to entry $n$ in $L_c$.
9.          Label entry $n$ with associated label $D_m$.
10.         $n = n + 1$.
11.     **end for**
12.     $i = i + 1$.
13. **end for**
14. Combine $L_{c_i}$ for $i = 1$ thru $D_m$.
15. Convert $L_{c_i}$ into dataframe $D_f$.
16. Return $D_f$.
```

In Algorithm 1, each modulation type D_m had its own function used for generating its respective values. Once the data values are combined into chunks of size c_s, the individual labels for each datapoint are removed and only one label for the entire chunk is appended. Once the data is generated, it is then shuffled (while keeping the datapoints within the same chunk) so that the model would not learn from the pattern that the signal generation algorithm creates. Fig. 2 provides a visual representation of this process.

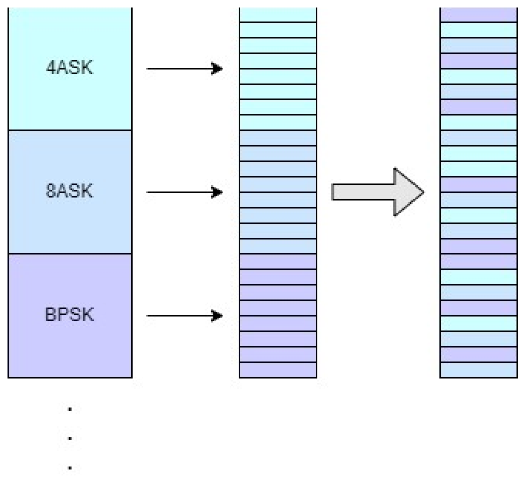

Figure 2. Data Generation Visualization.

The process illustrated in Fig. 2 depicts the methodology for both the training and evaluation datasets. A separate dataset was generated for evaluation to ensure the model was not simply memorizing the training data, and to avoid overfitting.

B. *Model Creation*

Next, the labels were encoded, and these chunks were added into a data loader that could then load the data into a Pytorch model. The models created for this work were named "VGGlike" due to their architecture resembling the model presented by Simonyan et. al. in [9]. The first part of the model (convolutional etc.) dealt with feature extraction, and the second part (the fully connected layers) dealt with feature classification. The model architecture was made more linear than in [6] for easy comparison between model sizes. The architecture was as follows:

- A convolutional layer, to extract features from the input data, producing an activation map with various weighted values which are adjusted during training based on calculated performance.
- A Rectified Linear Unit (ReLU) layer, used to introduce non-linearity and avoid the vanishing gradient problem (where gradients become very small on deeper layers, making it difficult for their weights to change during training).
- A Max Pooling layer to reduce the special size of the output of the layer, capturing the most notable features and adding robustness with respect to noise by reducing the influence of unimportant variations in input data.
- After a final convolutional layer, several fully connected layers (equal to convolutional layer count) exist to classify the input based on the extracted features.

For each of the models, the Brevitas library was used to quantize the layers as well as the inputs and outputs at various bit counts. Pruning was applied after the model was generated, with the weights being pruned at 10%, 20%, and 30%. Unpruned models were also produced for comparison.

Since the FINN library in its current iteration requires exporting the models in QONNX format, even the non-quantized models were quantized (at 32 bits) and used only in training benchmarks. This differs from the process used in [6], where non-quantized Pytorch layers were used for non-quantized models where possible. The QONNX format is a dialect of standard ONNX and was developed to be more friendly with both Brevitas and the FINN library, as discussed in [10]. Since the FINN library officially supports the QONNX format for quantization at 8 bits and lower, this was the common framework used for all models during synthesis.

C. *Model Training*

Training the model involved defining a Cross Entropy Loss function to evaluate the training process, and an Adam optimizer for optimizing the model as it learned. The cross-entropy loss was pulled from the Pytorch library, and is defined as:

$$L = -\sum_{c=1}^{C} y_c \log(p_c) \tag{1}$$

In equation 1, y represents the binary indication (1 or 0, true or false) if the class label c is the correct classification. The letter p represents the predicted probability of the class label c for the current observation. Values for learning rate and weight decay (for L2 regularization, to prevent overfitting) were tweaked to try and allow the model to converge on an acceptable loss value, which was also tweaked as needed between 0.003 for larger models up to .025 for smaller, heavily pruned models. The training stopped once this loss value was reached, and the model's performance was evaluated. This was repeated as necessary for each model with varying layers, quantization factors, and pruning rates. For models where pruning was applied, the model implemented structured pruning on the convolutional and fully connected layers, and then was retrained after pruning to optimize for the lower quantity of weights that were now being utilized on the model.

D. *Model Evaluation*

Evaluation was done by generating a new dataset using the same methods as training dataset generation, and then evaluating correct versus incorrect predictions. Models with less than 99% accuracy had their learning parameters tweaked until the desired accuracy could be achieved. The model sizes were selectively shrunk until they were too small to attain good accuracy values. The model sizes that failed to meet the desired accuracy levels for all combinations of quantization and pruning rates were removed, and that is how the bottom floor for model size was chosen for this work.

Model size was defined by layer count (fully connected, convolutional, ReLU, and max pooling) and noted in the model names (for example, VGGLike_5f_5c_4re_4mp). The upper bound of the model's accuracy value is obvious, and can be verified using the normal approximation method (discussed in [11]) to be 100%. Several models achieved 100% accuracy when tested against the evaluation dataset, at 16,384 samples.

The lower bound of the model's accuracy (and therefore, worst possible performance) is more relevant for performance metrics, and the Clopper-Pearson method is used for its calculation. This method accounts for the discrete nature of the sample size and gives a conservative value that is safe for estimated worst-case accuracy [12]. The value for A_a is the value of the accuracy of the worst-scoring model, at approximately 99.57%.

$$B_l = \frac{1}{1+\frac{n-A_a+1}{A_a} f_{\frac{\propto}{2},2(n-A_a+1),2A_a}} \tag{2}$$

In equation 2, the significance level $\propto$ is 0.05, a commonly used standard value in statistical analysis, as shown in [11]. The sample size, n, is 16,384 (the number of batches the evaluation dataset is tested against). The overall lower bound thus works out to be approximately 99.45%. With a median value of 99.73%, this calculated accuracy denotes a very reliable system; since real-world accuracy values of 97.1% [13] for modulation classification are considered high, this research accepts this methodology and dataset size as sufficient.

E. *Synthesis*

The models were exported in QONNX format, and then loaded into the FINN library using a docker script provided by the forked repository. A Jupyter notebook was used to load, process, and synthesize each model, where benchmarks were generated post-synthesis by Xilinx Vivado and evaluated. These benchmarks were then used for utilization and performance analysis.

III. Experiments and Results

Performance on GPU using CUDA, prior to synthesis, showed no improvements aside from reduction in models size, which was to be expected as the model generation and processes were optimized for hardware (now using Brevitas' layers instead of Pytorch). One notable benchmark was training speed; smaller models were notably harder to train down to an acceptable loss threshold, with quantization typically increasing this difficulty further across the smaller models.

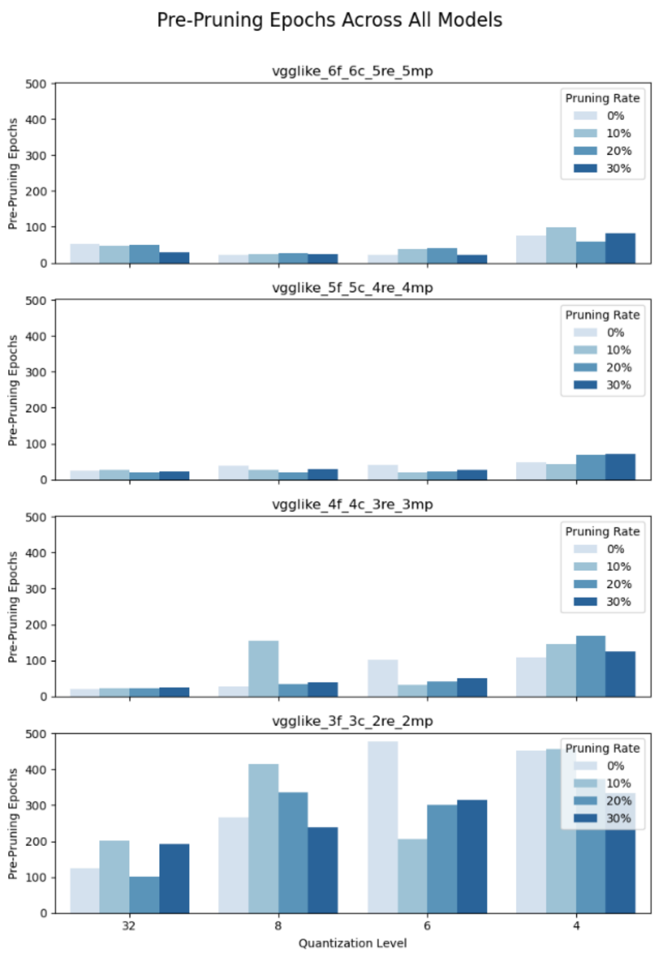

Figure 4. Pre-Pruning Epoch Counts During Training.

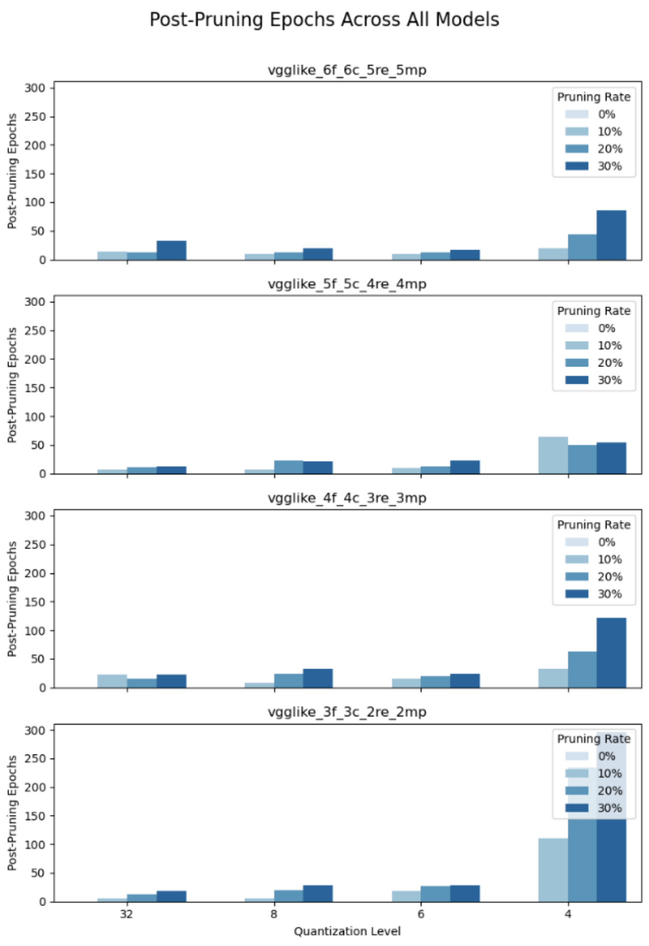

Figure 5. Post-Pruning Epoch Counts During Training.

Fig. 4 shows the number of epochs needed to converge on a proper model size, with quantization and model size both affecting how quickly the models converged below the loss threshold. In Fig 5., the epoch count for post-pruning training is shown. Quantization once again has a significant effect once 4 bits is reached, and the level of pruning also increases the training length linearly regardless of quantization. This demonstrates the difficulty of training more optimized models, though all were able to reach an acceptable accuracy level with the correct parameters applied.

When imported using the FINN library, the goal of the synthesizer is to meet timing and throughput, and then optimize for hardware utilization. If hardware is limited, the synthesizer will implement time-multiplexing (as discussed previously); but if performance requirements are too strict, the design may fail to meet them.

Even though the synthesizer optimizes for utilization at set clock speed, performance improvements can still be seen on hardware. One such example is the number of clock cycles needed to perform the classification, which would reduce from 131 cycles for 8-bit quantization, down to 127 cycles for 6-bit quantization, and then 123 cycles for 4-bit quantization. Notably, model architecture size had no effect on cycle count, nor did pruning.

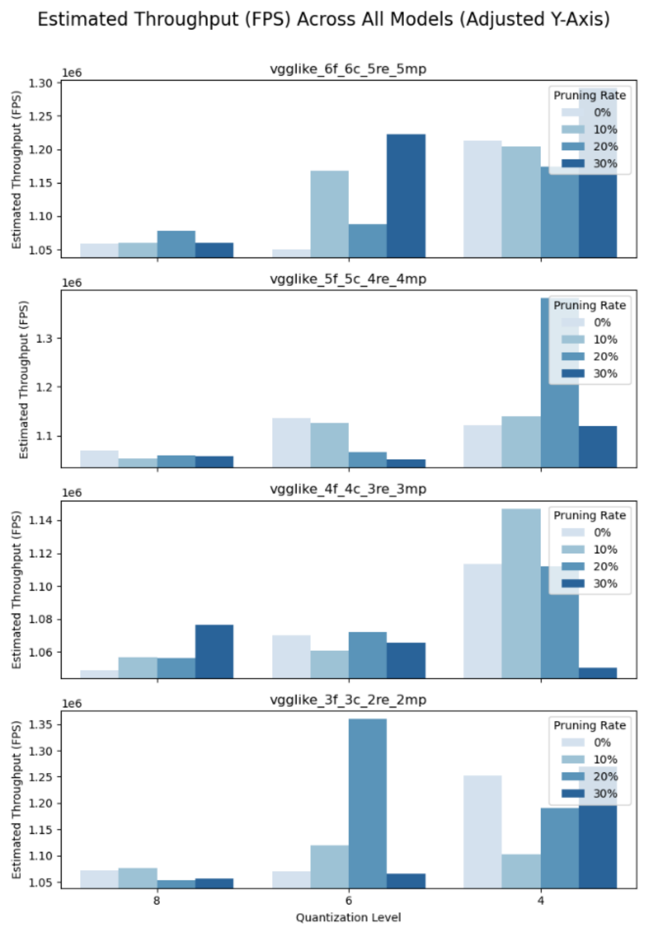

Figure 6. Estimated Throughput Across All Models, Adjusted Y-Axis.

While pruning is not shown to affect cycle count performance, there is another metric where some differences can be seen, such as in Fig. 6 above. How well pre- and post-training goes during pruning can be inconsistent depending on how the weights settle as the model iterates over its training set. In Fig. 6 above, the Y-axis is adjusted to show the differences in quantization and pruning. For the largest model in particular, aggressive pruning (30%) met with notable gains in throughput. The benefits of pruning on the smaller models with more aggressive quantization were variable, and the synthesizer may have taken potential performance gains away to conserve resources during utilization. Thus, utilization becomes an important benchmark to examine for the various model implementations.

In Fig. 7 and 8, hardware utilization is shown for the smallest model only (since all model sizes were the same for these metrics). The synthesizer appears to be unable to recognize the effects of pruning on memory utilization until quantization is down to the 4-bit level, in which it drops linearly. This is not surprising, as FINN's current iteration optimizes for quantization gains. Notably, the synthesizer moved its memory utilization onto the on-chip LUTs instead of the off-chip BRAM when it was able to do so. These would give considerable gains in power use and are therefore an important consideration for the designer.

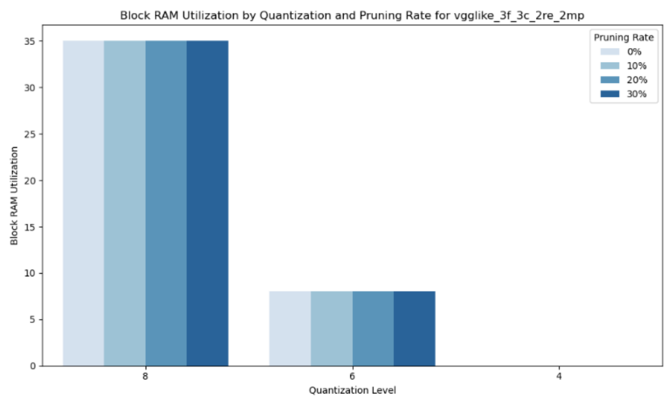

Figure 7. BRAM Utilization for VGGlike_3f_3c_2re_2mp.

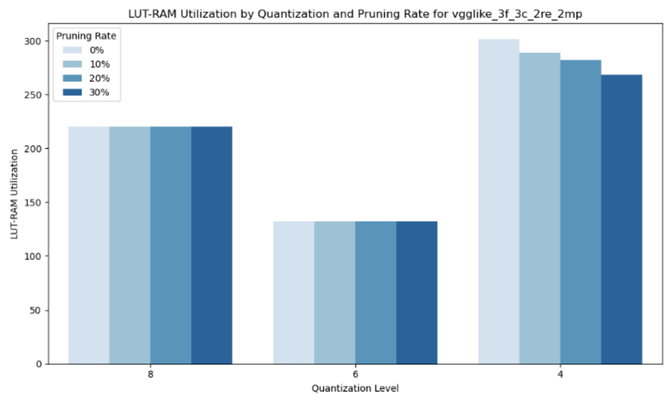

Figure 8. LUT-RAM Utilization for VGGlike_3f_3c_2re_2mp.

In an attempt to force the synthesizer to realize gains from model architecture size, the throughput and clock speed values were raised from the defaults. Maximum clock speed for the design (on FINN's default PYNQ device [14]) was around 120 MHz, and when throughput was raised, a target of 3 million FPS (up from 1 million) was achievable.

Due to hardware constraints, however, these gains were not particularly noteworthy. Memory components were noted to adjust slightly; for example, the VGGlike_6f_6c_5re_5mp model with no pruning and 8-bit quantization used 9,226 LUT blocks and 23,611 Flipflops, whereas the VGGlike_4f_4c_3re_3mp used 9,088 LUTs and 23,510 FFs. For these same models at 20% pruning and 6-bit quantization, the former utilized 6,728 LUTs and 17,301 FFs, whereas the latter utilized 6,572 LUTs and 17,207 FFs. The smallest model in all cases used slightly more FFs and LUTs than the largest model size but saw gains in throughput (overshooting the 3 million FPS and reaching around 3.06 million, whereas the largest model achieved 2.92 million at 20% pruning and 6-bit quantization).

Another comparison, 20% pruning and 8-bit quantization, provided a more consistent example of how utilization and performance trade off. This one was chosen because we do not see as much variation in this entry in Fig. 7, suggesting no quirks in training that may make any of these significant outliers.

TABLE I. 8-BIT QUANTIZATION, 20% PRUNING UTILIZATION

Model	**Metrics**				
	LUTs	***FFs***	***BRAM***	***Carry***	***Throughput***
VGGlike_6	9,212	23,582	32	1,168	2887202.762
VGGlike_5	9,267	23,728	32	1,165	3089509.262
VGGlike_4	9,058	23,469	32	1,161	3071404.000
VGGlike_3	9,131	23,578	29	1,171	3023267.063

For the smallest model architecture, the gains here seem to be primarily in BRAM utilization. Flipflop utilization trended downwards, but would sometimes trend upwards if throughput also trended upwards (likely another quirk of training for that model). Overall, model architecture gains on this smaller board were minimal due to the inability of the synthesizer to unfold the larger, deeper models.

IV. CONCLUSION AND FUTURE WORK

For efficient FPGA implementations of CNNs that perform classification tasks such as Automatic Modulation Recognition (AMR), utilization must be carefully balanced against performance and accuracy. For the Xilinx FINN library, gains can primarily be seen by aggressively quantizing a model, as the library has been designed to take advantage of the potential improvements that come from smaller weight and input storage sizes. For the hardware designer using Xilinx FPGAs, quantization is the primary way to gain improvements with both power and resource utilization. For classification tasks with simpler features, a deep CNN such as the larger ones tested here (6 convolutional layers, 6 fully connected layers) is likely overkill, but it does not significantly affect utilization due to the FINN library's ability to employ time multiplexing. Gains of roughly 10% on BRAM utilization were seen moving to the smallest model architecture, and less than that for other benchmarks. The notable improvements came from the application of aggressive quantization (4-bit), with a drop of roughly 33% in, for example, FF utilization between 8-bit and 6-bit quantization, and another 20% between 6-bit and 4-bit quantization. As another bonus, 4-bit quantization allowed BRAM to be eschewed completely, with the storage of weighs and biases moving to the LUT-RAM on the chip – which would improve power utilization significantly, something a designer of RF receivers can make good use of.

There were several considerations that came up during this work that are promising research directions in the future. Support for larger FPGAs such as Zynq (via MakeZYNQProject) and the larger Alveo architectures (via Vitislink) exists, and wrangling these CNN implementations onto a larger FPGA would allow a designer to unfold the architectures and explore the benefits of deeper CNN models. As the FINN model continues to improve and develop, pruning (structured or otherwise) will also likely become better supported, leading to further optimizations when applied. Additionally, implementing these models onto an FPGA along with antennas, filtering, and the processing required to separate out the I and Q values could produce a real-world example of an end-to-end implementation of the models' AMR functionality. As the FINN library continues to be improved, benchmarks and design considerations will need to continuously be evaluated to make the most out of these fast-evolving technologies.

V. REFERENCES

[1] S. Kumar, R. Mahapatra and S. Anurag, "Automatic Modulation Recognition: An FPGA Implementation," *IEEE Communications Letters,* vol. 26, no. 9, pp. 2062-2066, 2022.

[2] S. Han, H. Mao and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," in *ICLR 2016*, 2016.

[3] P. M. Gysel, *Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks,* University of California Davis, 2016.

[4] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.

[5] D. Góez, P. Soto, S. Latré, N. Gaviria and M. Camelo, "A Methodology to Design Quantized Deep Neural Networks for Automatic Modulation Recognition," *Algorithms,* vol. 15, p. 441, 2022.

[6] J. A. Rothe, *Quantization and Pruning of Convolutional Neural Networks for Efficient FPGA Implementation of Digital Modulation Detection Firmware,* Johns Hopkins University, 2024.

[7] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[8] M. Blott, T. B. Preusser, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser and K. Vissers, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS),* vol. 11, no. 3, pp. 1-23, 2018.

[9] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *3rd International Conference on Learning Representations,* 2015.

[10] A. Pappalardo, Y. Umuroglu, M. Blott, J. Mitrevski, B. Hawks, N. Tran, V. Loncar, S. Summers, H. Borras, J. Muhizi, M. Trahms, S.-C. Hsu, S. Hauck and J. Duarte, "QONNX: Representing Arbitrary-Precision Quantized Neural Networks," in *4th Workshop on Accelerated Machine Learning (AccML)*, 2022.

[11] D. Altman and J. Bland, "Statistics notes: Standard deviations and standard errors," *BMJ (Clinical research ed.),* vol. 331, no. 7521, p. 903, 2005.

[12] L. Orawo, "Confidence Intervals for the Binomial Proportion: A Comparison of Four Methods," *Open Journal of Statistics,* vol. 11, pp. 806-816, 01 2021.

[13] S. Peng, H. Jiang, H. Wang, H. Alwageed, Y. Zhou, M. Sebdani and Y.-D. Yao, "Modulation Classification Based on Signal Constellation Diagrams and Deep Learning," *IEEE Transactions on Neural Networks and Learning Systems,* vol. PP, pp. 1-10, 07 2018.

[14] Diligent, "PYNQ-Z1 Board Reference Manual," 2017. [Online]. Available: https://digilent.com/reference/_media/reference/programmable-logic/pynq-z1/pynq-rm.pdf.