QUANTIZATION AND PRUNING OF CONVOLUTIONAL NEURAL NETWORKS FOR EFFICIENT FPGA IMPLEMENTATION OF DIGITAL MODULATION DETECTION FIRMWARE

by
Joshua Andrew Rothe

A thesis submitted to Johns Hopkins University in conformity with the requirements for the degree of Master of Science

Baltimore, Maryland
July 2024

Abstract

Automatic modulation detection is an important function of communications systems. Commonly found in software defined radios, it enables radio receivers to interpret multiple and potentially changing modulation types without needing manual input from the user. Due to vastly increasing performance, many modern systems are moving away from the traditional two-stage process of feature extraction and classification; instead, a neural-network based system (also known as deep learning) is being utilized with increased speed and virtually no loss in accuracy. These implementations, when placed on hardware or on a Field Programmable Gate Array (FPGA), provide the fastest performance; but until recently the barrier of entry has been the size of the neural networks and the infeasible amount of resources they would need to occupy on the FPGA fabric. This thesis explores the effects of both quantization and pruning on convolutional neural network models of various sizes while maintaining high classification accuracy for the digitally modulated signals generated. In this thesis, a framework is proposed for the generation of the signals, models, and hardware estimation to serve as a guide for efficient deep learning implementations of models intended to fit on hardware with limited resources. The results demonstrate tradeoffs and design considerations that balance performance and implementation size for engineers aiming to implement a deep learning-based automatic modulation detection scheme on FPGAs.

Primary Reader and Advisor: Haya Shajaiah

Secondary Reader: Arnab Das

Contents

List of Figures

Chapter 1

Introduction

Signal modulation detection is an essential function of many radio frequency (RF) communications systems, and fast modulation detection is essential for systems where the signal being received may have an unknown or varying modulation style. A quick and reliable modulation detection algorithm enables a system to receive multiple signals quickly, and has several useful applications in the Electronic Warfare realm where the modulation may not necessarily be known or freely given prior to signal acquisition. Traditional modulation detection algorithms have been run on the CPU side and take up both time and processing power – hence the shift to firmware modulation detection. Offloading this computational responsibility frees up the CPU for other functions, making an FPGA implementation a solid choice. However, with considerations for both FPGA utilization as well as SWaP (Size, Weight, and Power) constraints, better methods are always desirable to improve the communications capability of a system. Convolutional Neural Networks (CNNs) have been shown to outperform traditional algorithms, but implementing them on FPGAs with limited resources has been a challenge for firmware engineers. Recent breakthroughs in this topic have led to various real-world implementations of CNNs that allow for FPGA implementations that can fit on FPGA firmware in a way that is realistic for the SWaP specifications for the systems that require it. Various implementation techniques can be utilized based on the system's requirements and limitations, to provide the same functionality that traditional algorithms do. The benefits of these novel implementations include additional functionality (on resource-limited systems) as well as meeting much faster timing requirements. [1]

In this thesis, various CNN models are created and trained on normalized I and Q values that would be typically extracted from a digital, modulated radio frequency signal, with noise added to simulate real-world signal quality. The proposed models consider different numbers of layers, with a focus on decreased layer count and smaller model architecture. Separate signal datapoints are generated and used to evaluate the model, while adjusting the training parameters until model accuracy reaches 100%. Finally, FPGA resource utilization is estimated based on the size and structure of the model. With these and various other benchmarks, the trade-offs between network size, quantization, and structured pruning are evaluated.

Related Work

Several papers have previously explored methods for FPGA implementations of automatic modulation detection. S. Kumar et al. [1] explores low-precision math and quantization to solve the model size problem, using a residual unit based scheme and iterative pruning-based training mechanisms to cut hardware utilization by 40%, achieving 527k classifications per second with a 7.5 μs latency. They acknowledge that both methods are well-known to reduce complexity and storage requirements for FPGA implementations of CNNs. They also reference SqueezeNet, a modification of AlexNet that reduces parameters by 50x and implementation size by 510x [2]. AlexNet is a CNN model architecture that was noteworthy for having won the ImageNet Large Scale Visual Recognition Challenge in 2012, drastically reducing the error percentage for its task by utilizing a model with a very deep layer count – five convolutional layers, two fully connected hidden layers, and one fully connected output layer; while also using the ReLU (rectified linear unit) as its activation function.

SqueezeNet itself uses several different methods for compressing the model size. Network pruning, demonstrated on SqueezeNet by Han et al. [3], was combined with quantization to create what is referred to as Deep Compression, greatly shrinking the model with no loss of accuracy. In [4], Han et al. later goes on to create the EIE (Efficient Inference Engine) hardware accelerator, where more hardware-based solutions (such as forsaking DRAM for SRAM and using ALUs) - but these implementations were still focused on CPUs and GPUs rather than FPGAs or hardware [5].

The work in [6] on SqueezeNet was done by P. M. Gysel in the development of Ristretto. Named after a highly concentrated form of espresso, it is a model approximation framework that can which reduced the bit-width of both network parameters and outputs of resource-intense layers, as well as the need for multiplication operations and utilization. It does this by using fixed point arithmetic and representation instead of floating point, as well as fine-tuning the resulting network. This work was done with a GPU focus, but the implementation of reducing parameter size, as well as Rectified Linear Unit (ReLu) layers and max pooling layers, are relevant to reducing FPGA utilization. The paper's data on accuracy versus model size implied that a larger model does not necessarily equal a better performing model, which lends optimism to the idea that a well-performing model can be efficiently placed onto an FPGA with lower resource utilization.

C. Zhang et al. explores CNN implementation on FPGAs, producing promising results [7]. One noteworthy conclusion from the paper is that there can be as much as a 90% performance difference between two solutions of similar FPGA utilization, so finding an optimal implementation is non-trivial.

Finally, D. Góez et al. in [8] explores the specific implementation of quantization on deep neural networks for automatic modulation recognition on FPGAs. They use a one-dimensional CNN and quantize the weights individually using the Brevitas library, and then analyze performance. Their paper states that they would like to evaluate this methodology when also using different model architectures and potentially pruning. In this thesis, I address both, and evaluate the trade-offs of both when combined with quantization.

This research expands upon existing techniques mentioned in the previous papers, including quantization to reduce bit width, structured pruning to reduce the number of unnecessary weights, ReLU layers, max pooling layers, and reducing the overall size of the CNN model itself. This study investigates the combined impact of these techniques on performance while verifying model accuracy is maintained. Furthermore, an FPGA resource utilization scheme is applied to estimate the resource achieved by implementing the above techniques on the models.

This thesis begins by providing a background on key concepts such as Automatic Modulation Recognition (AMR), Neural Networks (NNs), quantization techniques for reducing bit width, and structured pruning methods for removing redundant weights in neural networks. This information is followed by the problem statement, which is then discussed at length. Next, the methodology employed to address this problem is thoroughly described, followed by an analysis of the experimental setup, and resulting outcomes and analysis. Finally, the conclusions drawn from the research are presented, along with their potential implications.

Chapter 2

Background

Automatic Modulation Recognition

Automatic Modulation Recognition (AMR) is used on the receiver side of a communication system to detect the modulation of a signal without prior configuration. Modulation refers to how a Radio Frequency (RF) signal is transmitting data – a basic RF signal, called a carrier wave, is combined with a modulating signal that contains the data to be transmitted in a specific format. There are two main categories of modulation. Analog modulation, which carries analog data such as sound, includes familiar types such as Amplitude Modulation (AM) and Frequency Modulation (FM). Digital Modulation, which carries digital data (1s and 0s), includes modulation types such as Phase-shift keying (PSK) and Quadrature Amplitude Modulation (QAM). Digital modulation also has an additional parameter of increasing symbol count, which increases the amount of data that can be sent over the same period while also making it more prone to degradation due to noise, as adding more symbols to the same constellation (when limiting your system to the same amount of power regardless of modulation) will reduce the distance between them [9].

For friendly known signals where the modulation type will generally be given freely, AMR helps to reduce overhead and thus speed up the communications cycle; for unfriendly or unknown signals, it is a necessary step before the signal can be demodulated and read [10]. Another application is Electronic Warfare (EW), as the hostile signal must be matched in modulation before it can be sufficiently jammed.

Automatic Modulation Recognition typically has two stages in a traditional system – feature extraction, and classification. Feature extraction is the method by which the system, such as a Software Defined Radio (SDR), chooses what features to pull from the signal of interest. At this stage, the system will also perform necessary signal processing and filtering to make the features as easy to extract as possible. Classification is the second stage, where the system uses the previously extracted features to determine what the modulation type of the signal is.

Modern systems are moving away from this two-stage classification system and instead into a CNN-based system, which does the feature extraction and classification at the same time, albeit with different layers in the network (convolutional layers for extraction, and fully connected layers for classification). Letting the neural network select the important features to extract and classify makes the process both simpler for the designer and more accurate, as the system is not limited to only the features the designer thinks are important but is instead free to select its own. Most of these new classification methods revolve around the implementation of trained Neural Networks (NNs) for this purpose [11].

Neural Networks

"Neural networks" (NNs) are a concept that has been around for over a century, although obviously not in its current form. The first neural network was published by Adrien-Marie Legendre in 1805, although Johann Gauss is also credited with unpublished work on this same topic around 1795 [12]. Called the method of least squares at the time, now known as linear regression; its early framework consisted of an input layer, an output layer, and real-valued weights which continuously adjust based on a set of input vectors. One-dimensional General Regression Neural Networks (GRNNs) are the clearest example of this, but of course modern

GRNNs tend to have multiple dimensions for increased complexity and performance [13]. These types of neural networks, commonly referred to as Machine Learning (ML), are some of the simpler forms that are in use today, and the most common tutorial application of these is prediction of house sale prices based on attributes it is trained on (such as square footage, number of rooms, location, and so on).

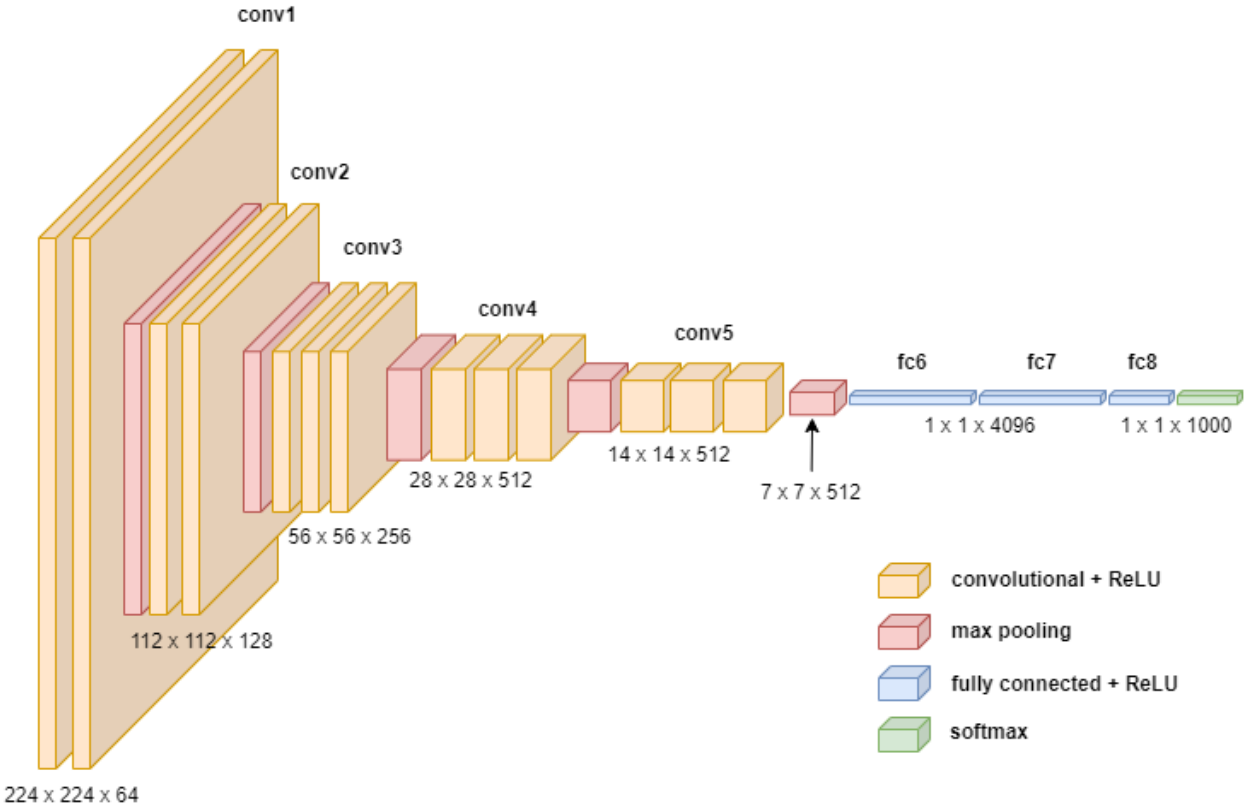


Figure 1. Visualization of a VGG-like DeepLearning Neural Network [14]

Deep Learning (DL), more academically known as Artificial Neural Networks (ANNs), is a subset of Machine Learning, differentiated by its attempt to mimic biological neural networks in the way that it trains on the provided data. Nodes communicate with other nodes (which were previously the iteratively updating weighted values), and the connections between nodes are weighted based on their successful ability to deliver a correct outcome [15]. Increasing

complexity and adding layers can sometimes, but not always, increase the performance of the neural network. The discovery that Artificial Intelligence (AI) researchers have been making recently is that bigger is not always better, and some well-tuned models of vastly smaller size can outperform models that are even orders of magnitude more complex. As a recent example, Google executives have concluded that open-source chat models are beginning to overtake corporate AI researchers' larger, more complex, and proprietary models, the most common example being OpenAI's ChatGPT [16]. There are several possible contributing factors to this phenomenon – one, smaller models can avoid overfitting to unnecessary features; two, the quality of the dataset and the tools used in the architecture of the model itself can greatly influence its performance. There is no "one size fits all" brute force model you can simply apply to everything with infinite computing power, and sometimes clever design can lead to breakthroughs (as with AlexNet). As data scientist and engineers become more skilled in their use of neural networks, this naturally progresses to more effective implementation of CNN models.

What this means for AMR NNs is that simply, the theoretical networks that were too large to fit on common hardware are now able to be scaled down without sacrificing performance. For embedded systems and systems where SWaP is a concern, this allows actual implementation to occur on hardware where it previously was not feasible. Several articles have been published just in the last two years which detail various implementations of these lightweight yet fully functional neural networks [1] [8].

Quantization

Quantization is a very common process in machine learning and neural network optimization, and the concept itself is not new – for very small or resource-constrained systems, reducing precision of numbers to free up memory space is a common practice. Quantization applies this concept to neural networks by reducing precision – on node weights and biases (parameters) within the network as well as values going into and out of the model. Many modern systems that synthesize NNs onto CPUs, GPUs, and FPGAs even have some form of quantization optimizer built in, as the practice has become standard for purpose of both size reduction as well as increasing performance – smaller numbers can easily lead to faster computation times, as less bits need to be handled in the operation.

Standard precision for most neural networks is 32-bit, but it is widely accepted that 8-bit quantization leads to no loss of accuracy – in [4], quantization from the typical value (32-bit) can drop the model down to ~17% of its original size before accuracy begins to become noticeably affected, and they start at 8-bit and work their way down to lower and lower precision. Thus, this thesis starts at 8-bit quantization, works lower to investigate how much precision can be further removed while maintaining accuracy, and analyzes how performance is affected.

Pruning

Another method used to reduce the size of CNN models is pruning, which is usually applied after training to remove unnecessary weights that may negatively impact performance without helping (or possibly even hindering) accuracy. After a model is trained, some neurons that contribute little to a model's accuracy will have a correspondingly low weight assigned to

them. By applying pruning, you remove a ratio (such as 20%) of the lowest scoring neurons from the network.

There are different types of pruning – weight pruning, neuron pruning, and structured pruning. Weight pruning removes individual weights between neurons, setting them to zero. Since the weight is not actually removed, for an FPGA implementation it is dubious whether a synthesizer will be able to take advantage of the implementation benefits. Neuron pruning is better, removing entire neurons (and the connections) resulting in a smaller network architecture. However, again, it is not guaranteed that a synthesizer can take advantage of this, either; it may need to pad the removed nodes with zeros but still leave them within the network. Structured pruning, which is the type of pruning used in this thesis, provides the best chance of benefit – it prunes entire channels and filters, which reduces computations and improves memory requirements.

Again in [4], pruning was able to compress a model down to 20% of its original size with no loss of accuracy, and down to around 8% with an accuracy loss of only 0.5%. Han et. al. performed their work on much larger models to demonstrate the effects of quantization and pruning combined, so they were able to see sizeable improvements by applying both techniques together. In this thesis, quantization and pruning are combined with reducing model size as low as possible for a relatively simple classification task; so, while the results may not be quite as drastic, the improvements should still be notable as they were in previous works.

Chapter 3

Problem Formulation

The foremost concern, when implementing a CNN onto an FPGA to classify digitally modulated signals, is reducing the size of the network while maintaining sufficient accuracy. Techniques such as pruning, quantization, and reducing model size can all be used to compress a neural network's implementation footprint without significantly affecting accuracy.

This thesis characterizes the trade-offs between performance and implementation size across different model architectures; specifically, it evaluates combinations of pruning rates (none, 20%, 30%) and quantization (none, 8-bit, 6-bit, 4-bit) across CNN models with varying layer counts. The thesis pays close attention to:

- Diminishing returns – at what point does a specific combination of these techniques yield less-than-ideal results?
- Are there optimal configurations that balance size, latency, and other benchmarks?
- Which techniques should be favored based on observed performance improvements when implemented?

The approach applied to answering these aforementioned questions is displayed in Fig. 2.

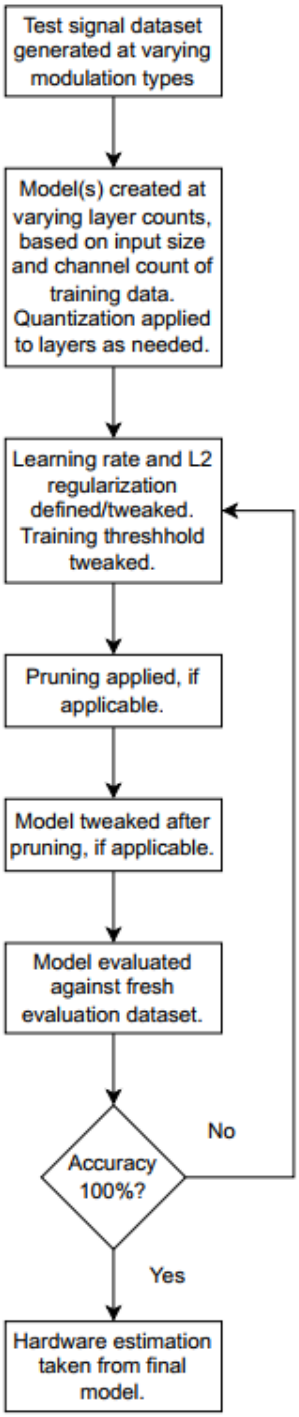


Figure 2. Workflow for Model Creation and Benchmarking.

This solution, in Fig. 2, utilizes a standard workflow for model creation, quantization, pruning, training, and evaluation. First, a test dataset algorithm is created, with labeled "chunks" of I and Q values corresponding to a specific modulation type. Multiple model architectures are

constructed with varying layer counts, which are then divided and categorized by quantization level and again by pruning rate. The models are trained with a unique training set generated by the test dataset algorithm, and learning parameters are adjusted until the desired accuracy (100%) is achieved with a separately generated evaluation dataset. Finally, the benchmarks are then analyzed to identify trends and identify trade-offs in model compression techniques.

Chapter 4

Methodology

Dataset Generation

The generated data type is a critical component of training and evaluating CNNs. Therefore, the first task was to create a signal generator that would create the various data types for the modulation recognition model. 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 model assumed normalized I and Q values with noise added to account for real-life conditions. It should be noted that if the model was trained on noisy data, larger models could maintain 100% evaluation accuracy even if the noise was turned up to a point where it was almost undiscernible on data plots. Therefore, a reasonable noise value of 0.005 (typically in Watts, but here it is 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. 3, for the 16QAM modulation type.

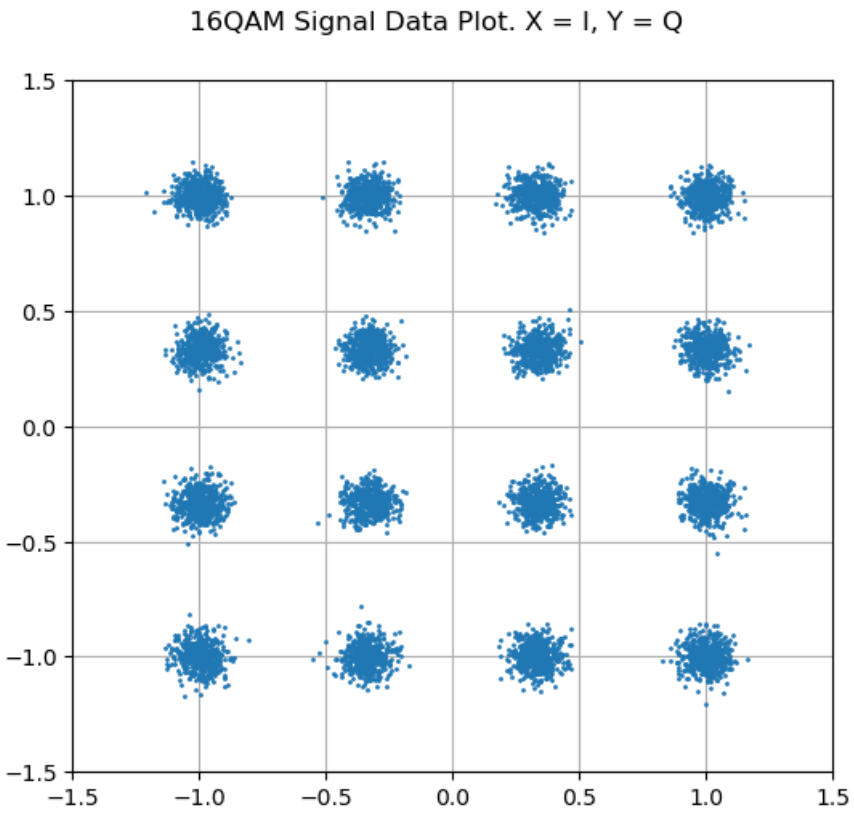


Figure 3. Generated I and Q Values for 16QAM.

This noise would be combined with the generated data to produce imperfect I and Q data (as seen in Fig. 3), more like what would be extracted from real-life signals. Additionally, for each labeled dataset item, the signal generator created a set of V_d quantity I and Q pairs, grouped into size c_s chunks. The data values were split this way since the model would need to see a certain number of random datapoints to determine modulation type (for example, a normalized I value of 1 and a Q value of 0 can be present in all but two of the modulation types listed, so this datapoint is useless by itself – and with sufficient noise added, it can be present in all of them). 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 to be executed after each set of I and Q values is generated.

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. (L_c is the list of I and Q pairs).
15. Convert L_{c_i} into dataframe D_f.
16. Return D_f.

In Algorithm 1, signal generation functions are not listed for brevity, but each modulation type D_m had its own function used for generating its respective values. Once the data values are combined together into sets of size c_s, the individual labels for each datapoint is 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 does not attempt to learn from the pattern that the signal generation algorithm creates. Fig. 4 provides a visual representation of this process.

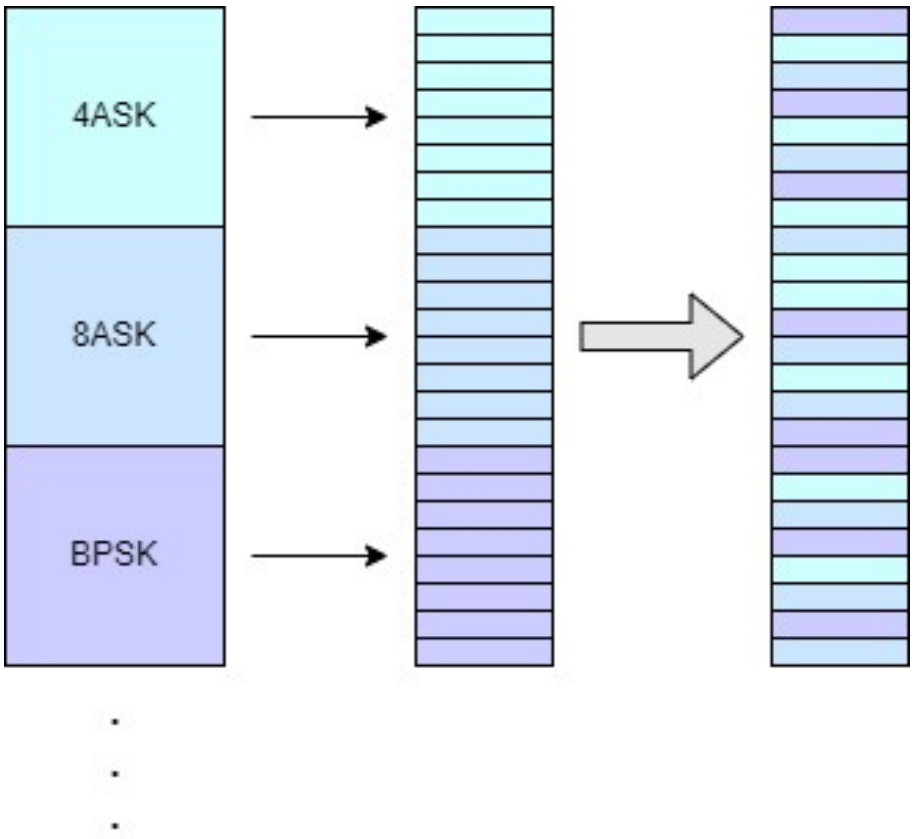


Figure 4. Data Generation Visualization.

As previously stated, a large set of randomized values of each modulation type is generated, and then these sets are split into chunks, labeled, and shuffled. This is 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.

Model Creation

Next, the labels were encoded, and these chunks were added into a custom data loader that could then load the data into a Pytorch model. The models created for this thesis were named "VGG-like" due to their architecture resembling the model presented by Simonyan et. al. in [17]. The model consists of several convolutional layers, each followed by ReLU and max pooling layers, and then

at the end having several fully connected layers (each also followed by ReLU layers). The first half (convolutional etc.) dealt with feature extraction, and the second half (the fully connected layers) dealt with feature classification. For models where quantization was applied, the Brevitas library was used to quantize the layers individually. Pruning was applied after the model was generated in various forms, with the weights being pruned at 20% and 30%.

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:

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

In the above equation, 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 0.030 and 1.000. 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 factors. For models where pruning was applied, the model was again retrained after pruning to optimize for the lower quantity of weights that were still being utilized on the model.

Model Evaluation

Evaluation was done by simply generating a new random dataset similar to training, and evaluating correct versus incorrect predictions. Models with less than 100% performance had their learning parameters tweaked until 100% accuracy could be achieved – the smallest model was dropped out once quantization began since it failed to reach this level of performance. 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) with layer counts varying from 5 down to 2 or 1. Since the training and test datasets both used 2,304 chunks of input data, a 100% accuracy result could be interpreted as a real-world accuracy between 99.84% and 100%. The upper bound of this accuracy value can be verified using the normal approximation method [18] and by plugging in the following values: a 95% confidence level C_l (a commonly used standard value in statistical analysis [19]), a critical value z of 1.96 (commonly calculated from C_l, also a common value), sample size n of 2,304, the achieved accuracy (100%) A_a, and using a normal approximation method [19] to calculate the upper bound of confidence interval B_u.

$$B_u = A_a \pm z \times \sqrt{\frac{(A_a \times (1 - A_a)}{n}}$$

From the above equation, the upper bound on the accuracy of this neural network is verified to be 100%. Since this equation would give the same result for the lower bound B_l, the Clopper-Pearson method was used instead for its calculation. This method accounts for the discrete nature of the sample size and gives a more conservative value [20].

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

The significance level, $\propto$, is simply $1 - C_l$. The overall lower bound thus works out to be approximately 99.78%. With a median value of 99.89%, this performance accuracy could be seen as reliably high performance and would likely perform even better since this lower bound estimation was so conservative. Since real-world accuracy values of 97.1% [21] for modulation classification can be considered high, the thesis accepts this methodology and dataset size as sufficient.

During evaluation, benchmark data was generated. Each model was run on its own, on the same GPU - an NVIDIA RTX 3050 Ti, using NVIDIA's Compute Unified Device Architecture (CUDA) to accelerate the model training - and values such as inference time and training throughput were stored for each model. For hardware utilization, code was generated to estimate FPGA resource utilization. Since the hardware utilization estimation is not specific to any one FPGA device, and devices will vary widely based on how the synthesizer distributes the network across the varying architectures, the estimation is more comparative and assumes the same FPGA target for the varying model types.

Hardware Utilization Estimation

Hardware utilization is difficult to estimate without synthesizing the model onto an FPGA, since FPGAs have varying resources available and vendor tools may handle the synthesis differently. Thus, the following assumptions were made:

- A Multiplier Adder Accumulator (MAC) uses one DSP block for the MAC operation plus another for the accumulator portion in both convolutional and linear layers. This is a common practice within FPGA implementation and provides an upper bound on DSP block usage [23].

- For look up tables (LUTs), 4 per bit per layer was a (possibly over-) estimation to account for additional logic on convolutional layers. For linear layers, the number of input and output channels was used, again to account for additional logic. In the max pooling layers, the number of LUTs is based off the number of input channels of each layer. This is a rough estimate of the logic complexity involved, and actual LUT usage will differ based on the synthesizer, target hardware, and optimizations used.

- Two flipflops (FFs) per bit were assumed to store the weights and intermediate results, and then one for the bias parameters for both convolutional and linear layers. Storage elements are understandably necessary within the pipeline stages of the CNN.

- Block RAM (BRAM) utilization was based on weight matrix size. For this hypothetical FPGA implementation, each BRAM is assumed to be 32 Kbits, and is measured off the linear and convolutional layers. Basing BRAM utilization on the weight matrix size is a logical approach since weights are the primary data stored in BRAMs.

- ReLU layers were left out of the hardware estimation since their impact is very minimal – they do not require multipliers or a large number of adders, and can be implemented in-place (meaning it can simply overwrite the input data with its own output data, thus not requiring any additional storage).

- Since pruning leaves zero weights behind, the FPGA being synthesized on is assumed to have a capability for sparse matrix operations, which allow it to take advantage of structured pruning, which is supported by C. Zhu et al. [22]

With these assumptions in place, Algorithm II was used to estimate FPGA utilization benchmarks. Layer counts for number of convolutional, linear, and max pooling (C_C, C_L, and C_M, respectively) as well as values for quantization bit width (B_w), non-zero weights (N_w), and non-zero biases (N_B) are used to estimate utilization of DSP, LUT, FF, and BRAM blocks on a typical FPGA (E_D, E_L, E_F, E_B).

Algorithm II: FPGA Hardware Resource Estimation
Input: Layer counts (C_C, C_L, C_M,), quantization bit width B_w, non-zero weights N_w, non-zero biases N_B, number of input and output channels for the layer C_i and C_o.
Output: Estimated utilization of FPGA resources:E_D, E_L, E_F, E_B.
1. $C_{nw} = N_w * B_w / 32$.
2. $C_{nb} = N_w * N_B / 32$.
3. $E_D = (2C_{nw} * C_C) + (2C_n * C_L)$
4. $E_L = (4C_{nw} * C_C) + (C_i C_o) + (C_{nw} * C_M * C_i)$
5. $E_F = (2C_{nw} + C_{nb}) C_C + (2C_{nw} + C_{nb}) C_L + (2C_{nw} + C_{nb}) C_M$
6. $B_B = ((C_{nw} * C_C) + (C_{nw} * C_L)) / 32{,}000$
7. Return estimates E_D, E_L, E_F, E_B.

Algorithm II is written specifically for this VGG-like model architecture, which accounts for quantization (using bit width) as well as handles the layer types (linear, convolutional, and max pooling) used. The estimation algorithm also attempts to throw out zeroed components, to give an idea of pruning's impact on hardware utilization. To this end, the coefficients C_{nw} and C_{nb} are used to take both quantization and pruning into account for both weights and biases – the zeroed

components of a layer as well as the unused bit width would be dropped by an appropriately competent compiler.

Chapter 5

Experiments and Results

Training speed was significantly slower when pruning was applied. On the largest unquantized models, the largest had a training time of 19.49 seconds, and when pruned at 30% the training time rose to 28.06 seconds. When quantized to 6 bits, the unpruned model's training took 99.97 seconds, whereas the pruned (30%) model took 133.07 seconds. In all cases, it was evident that when applying techniques to shrink the size of the model, the training process became more difficult as the model had less resources to work with.

The model's latency (the time it took for a single task to be processed) during training seemed consistent at around 2.5 ms for the larger models and 1.7ms for the smaller models, with pruning having very little effect. Quantizing the models led to an increase of roughly a factor of five, so from 2.5 ms to 15 ms, without much variation with pruning added.

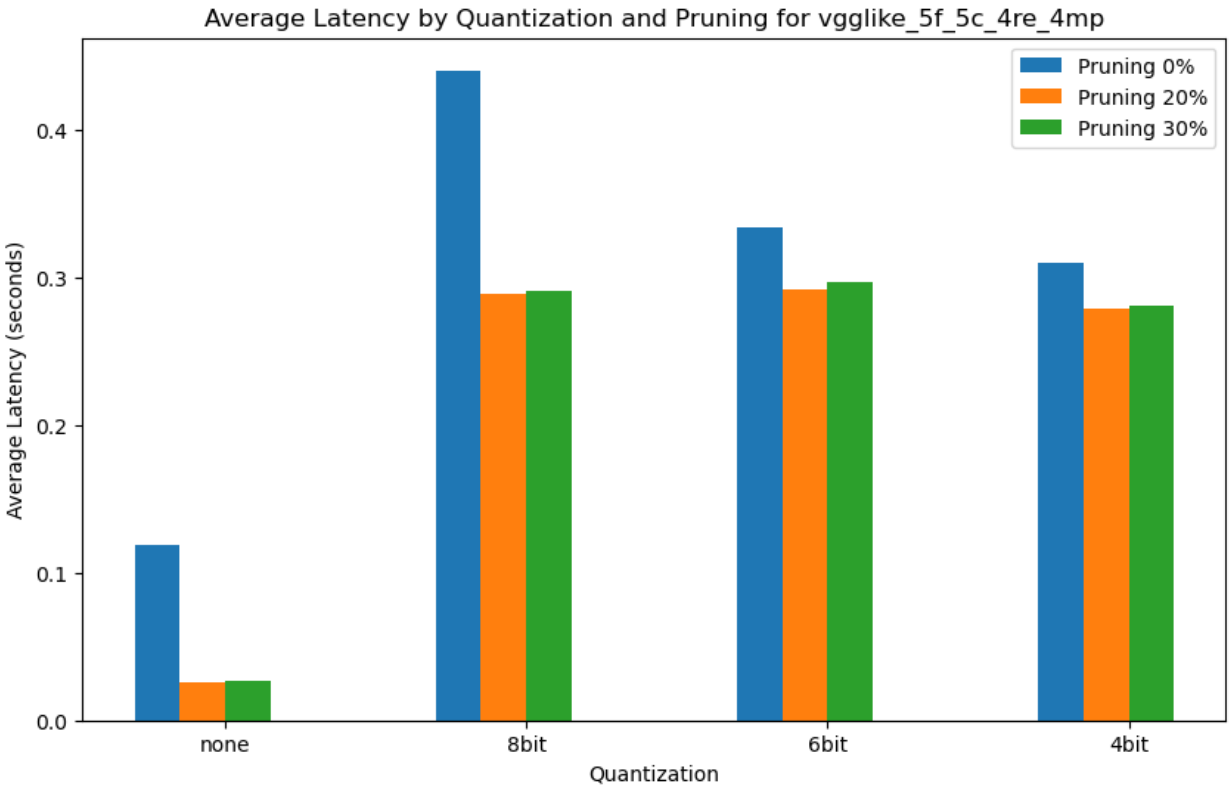


Figure 5. VGGLike_5f_5c_4re_4mp Average Latency at Various Quantization and Pruning Rates.

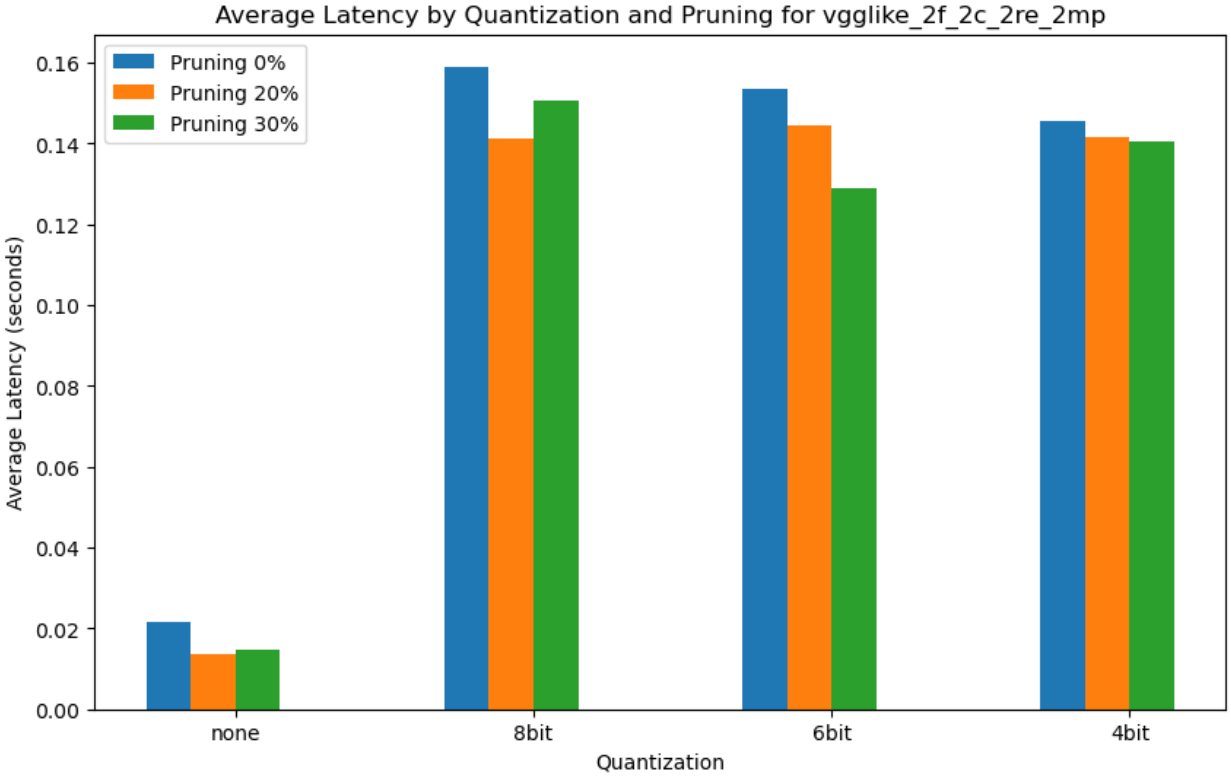


Figure 6. VGGLike_2f_2c_2re_2mp Average Latency at Various Quantization and Pruning Rates.

In Fig. 5, the average training latency for the largest model is shown, and in Fig. 6, the second smallest. The difference between the larger and smaller models is apparent, with smaller models have much lower average training latency overall. Quantization seemed to increase the latency by a factor of 4-6, denoting a reduced performance when bit count is truncated – with pruning providing a slight benefit. Note that for the model titles, the fully connected layers, convolutional layers, ReLU layers, and max pooling layers are stated in shorthand – in Fig. 6, there are two fully connected layers, two convolutional layers, two rectified linear unit layers, and two max pooling layers.

Training throughput (number of requests that can be processed per unit of time) was much higher for smaller models than larger ones, reflecting much faster performance (16,000 samples/sec for the smallest, versus around 7,500 samples/sec for the larger models). Quantization in both cases cut the throughput down by a factor of three regardless of bit size, with pruning not having much effect.

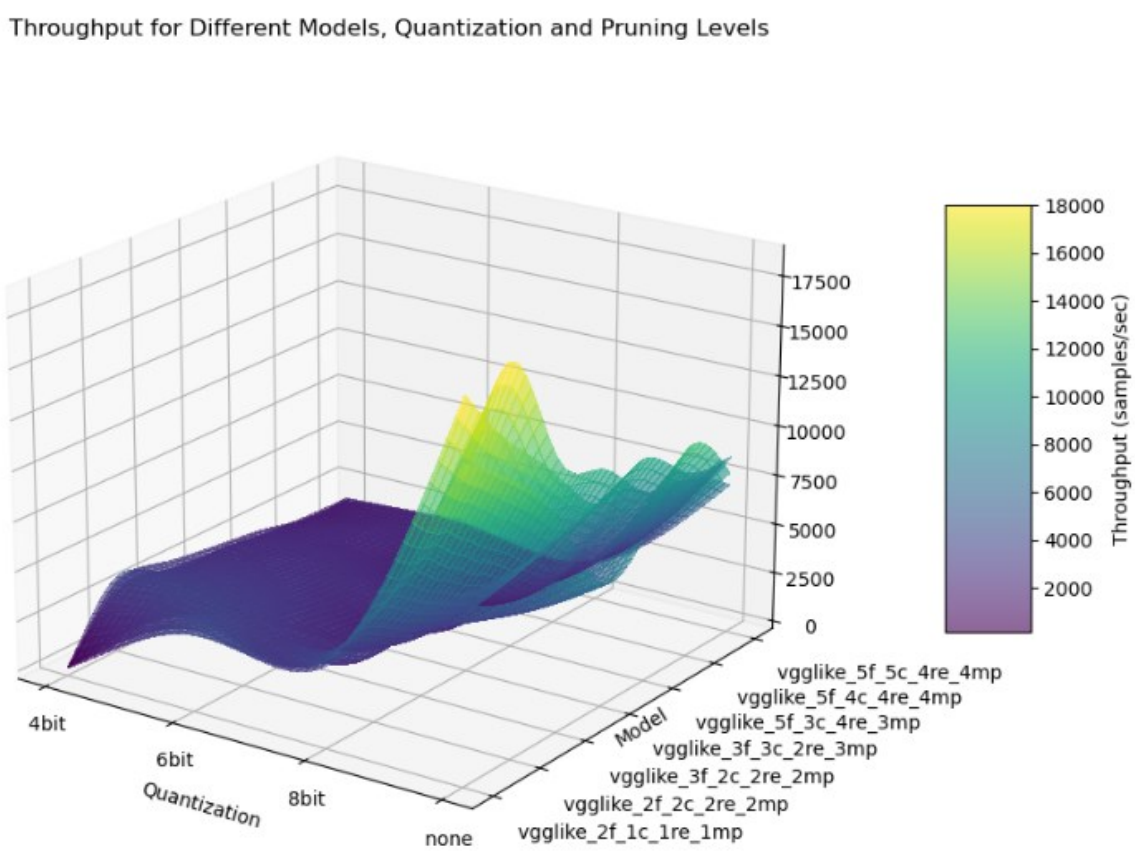


Figure 7. All Models Compared – Throughput (samples/sec) and Quantization, Sorted by Model Architecture. Each Layer is a Different Pruning Rate.

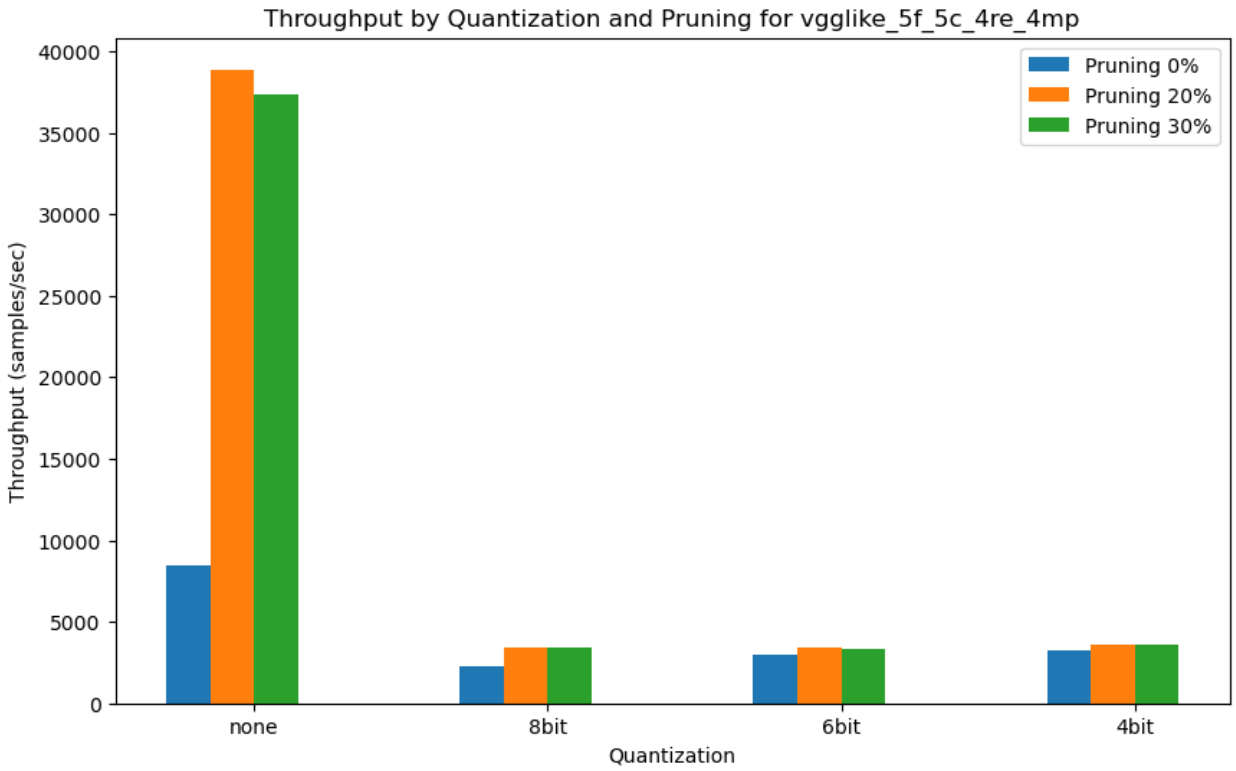


Figure 8. VGGLike_5f_5c_4re_4mp Average Throughput at Various Quantization and Pruning Rates.

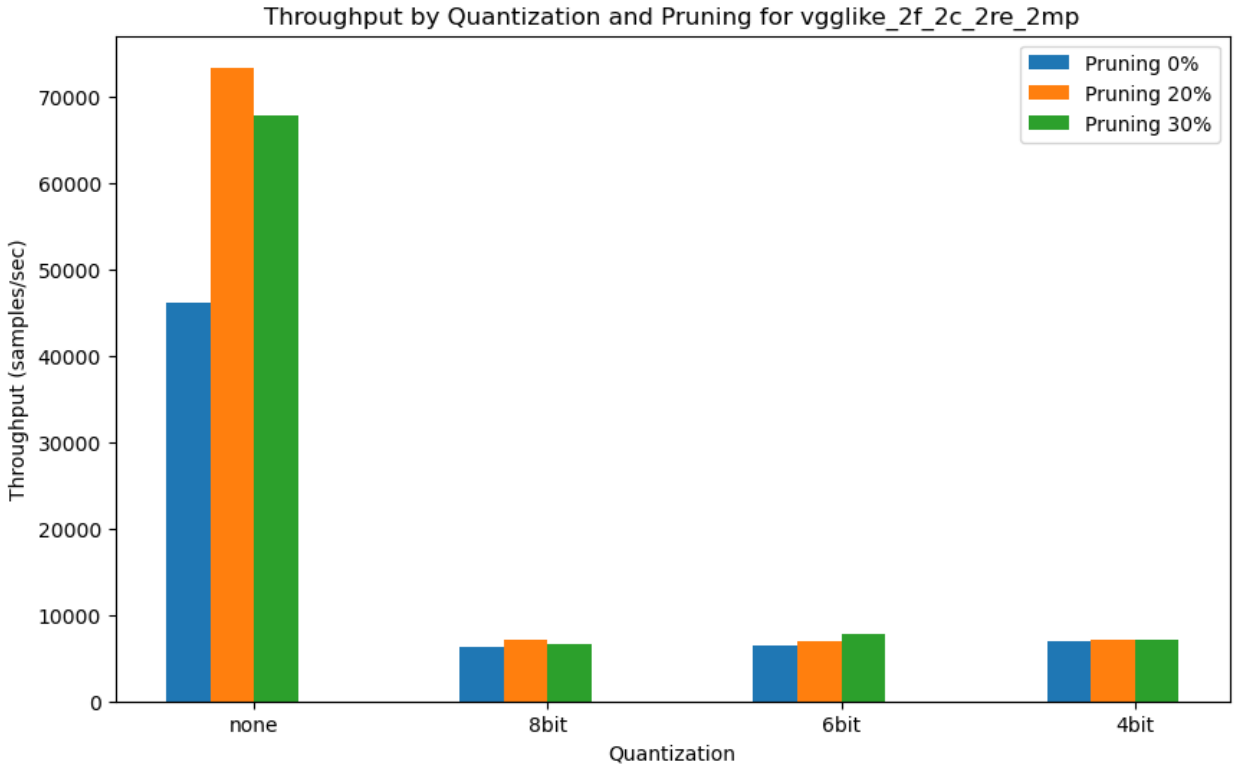


Figure 9. VGGLike_2f_2c_2re_2mp Average Throughput at Various Quantization and Pruning Rates.

In Fig. 7, throughput (the amount of samples per second that the model can process) at different model sizes versus quantization rate is shown on a 3-dimensional graph. In Fig. 8 and 9, the largest and second-smallest models are displayed individually. Throughput performance suffers once quantization is introduced, with pruning (generally) providing a slight benefit. In Fig. 7, a graph layers are close here, with more aggressively-pruned models having slightly better performance; however, in all cases, the non-quantized model has significantly higher throughput.

Inference time (the time it took for the model to make predictions on a single batch of inputs) during evaluation raised by a factor of ten when quantized by any amount, with pruning reducing the inference time linearly when applied. Smaller models had less of a growth, in proportion with their lower original inference time (0.5 to 5ms or so when quantized, with larger models increasing from around 1ms to 10ms). Inference time is one of the most important metrics when measuring a model's performance, thus highlighting the importance of quantization and pruning for both size and performance of a CNN model.

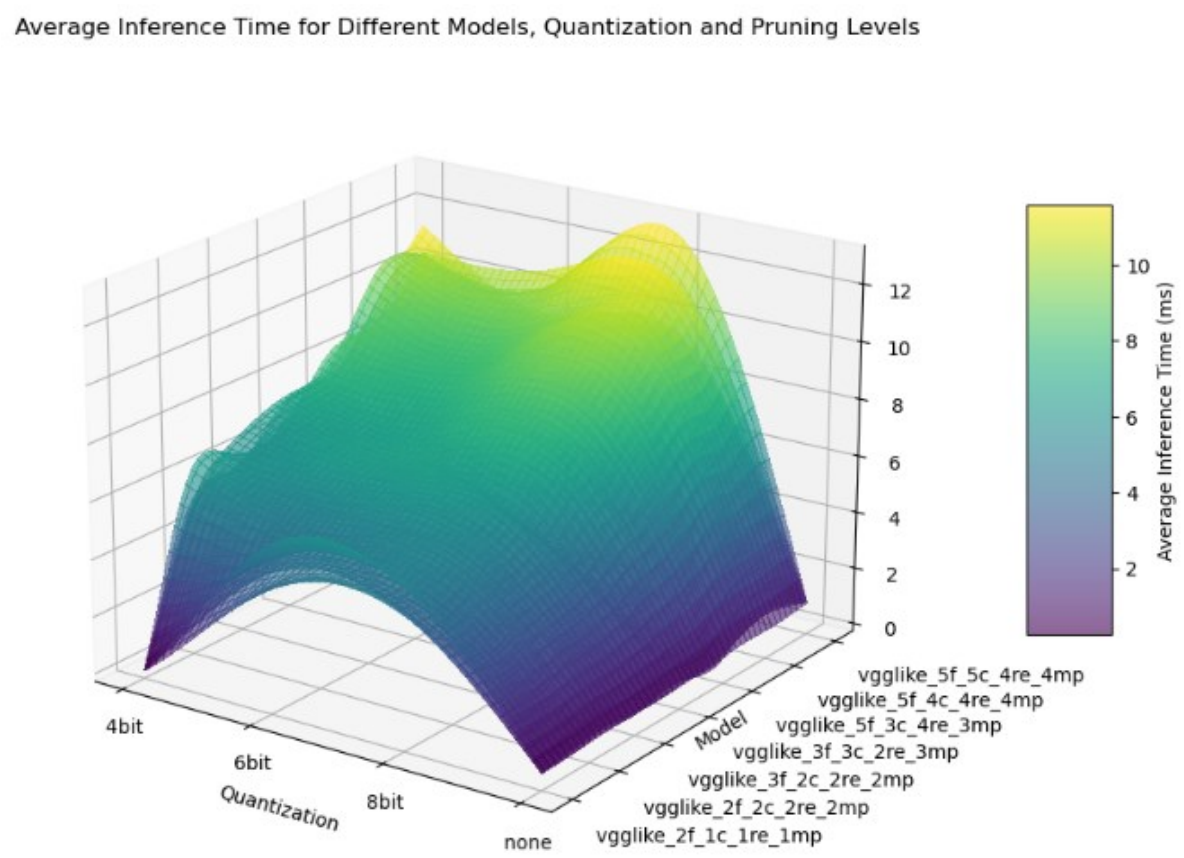


Figure 10. All Models Compared – Inference Time During Evaluation (ms) and Quantization, Sorted By Model Architecture. Each Layer Is a Different Pruning Rate.

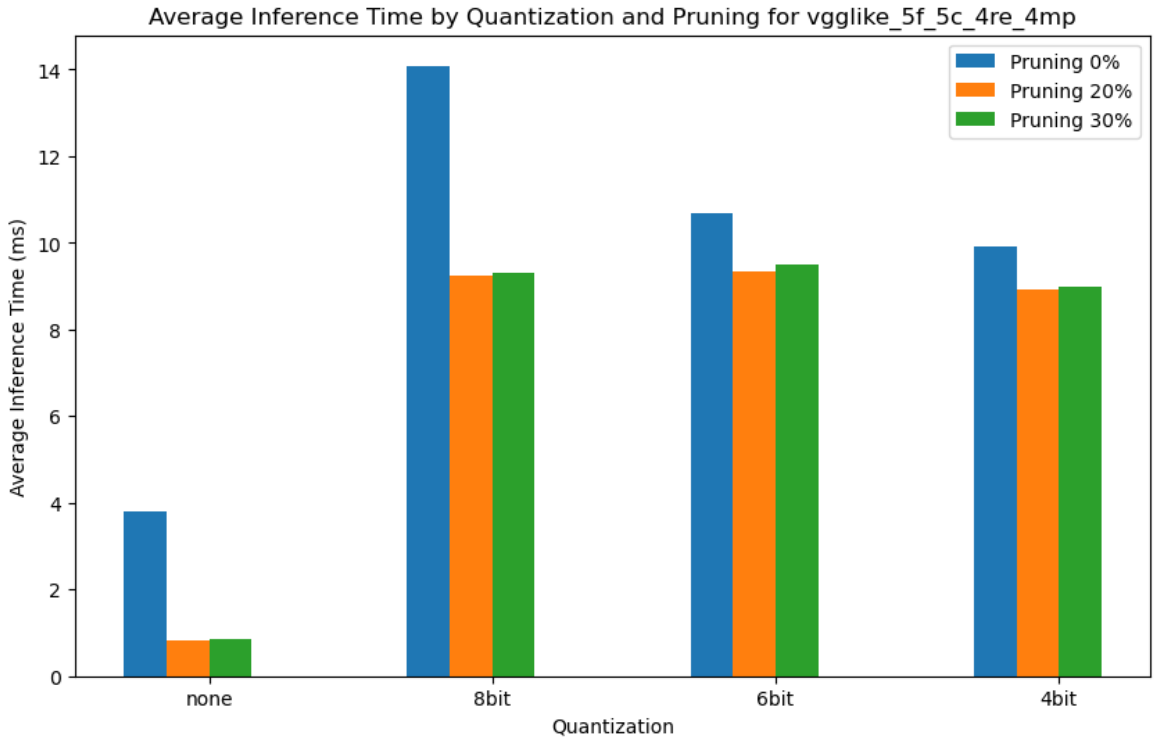


Figure 11. VGGLike_5f_5c_4re_4mp Inference Time (ms) at Various Quantization and Pruning Rates.

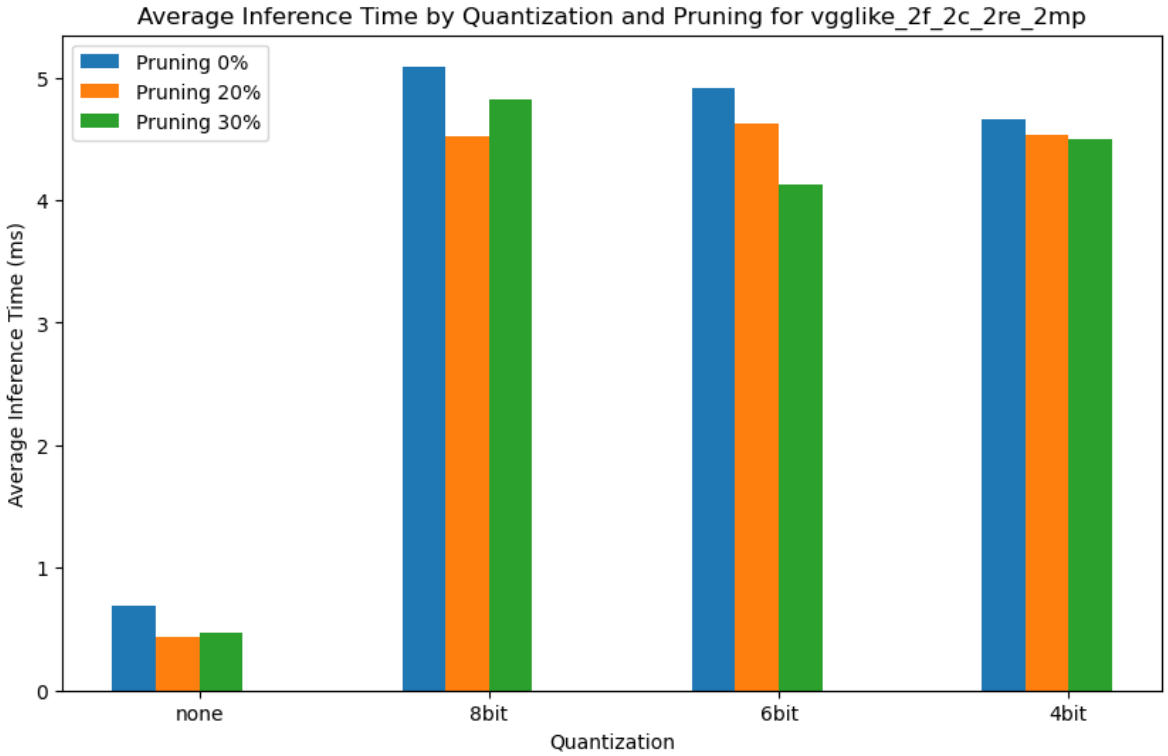


Figure 12. VGGLike_2f_2c_2re_2mp Inference Time (ms) at Various Quantization and Pruning Rates.

Fig. 10 (3-d) and Fig. 11 and 12 (largest and second-smallest model) show the average inference time during evaluation. In Fig. 11 and 12, inference time scales up rapidly with the introduction of quantization, with pruning giving a marked improvement in performance when applied. Fig. 11 shows even larger performance gains when pruning is applied, with a much less but still noticeable improvement in smaller models as shown in Fig. 12.

Hardware resource utilization estimates were heavily influenced by the size of the model, and quantization caused a cut in resource utilization by a factor of 4. Further quantization showed a linear and predictable cut in resource utilization for all models. Pruning seemed to have a linear effect on reducing resource utilization further, with a 20% prune rate corresponding to a roughly 10% drop in resource utilization. Estimating hardware utilization is very rough in this case, since it is difficult to determine that pruning will provide any reduction in resources if the synthesizer cannot handle it properly, without synthesizing the model.

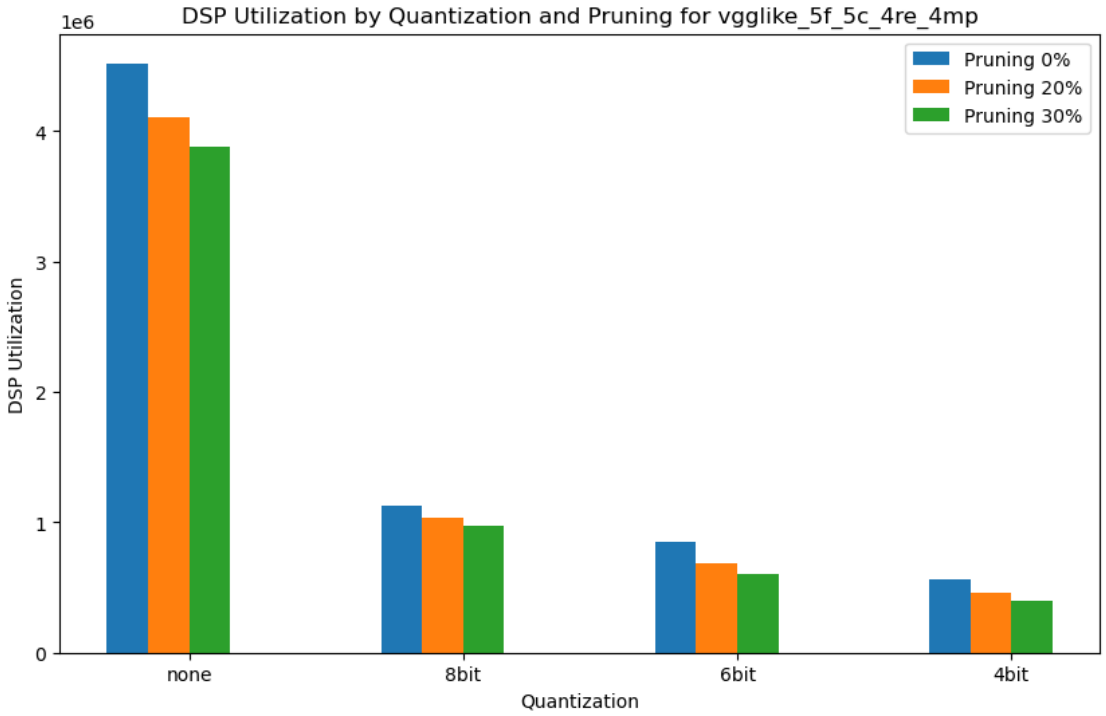


Figure 13. VGGLike_5f_5c_4re_4mp DSP Utilization at Various Quantization and Pruning Rates.

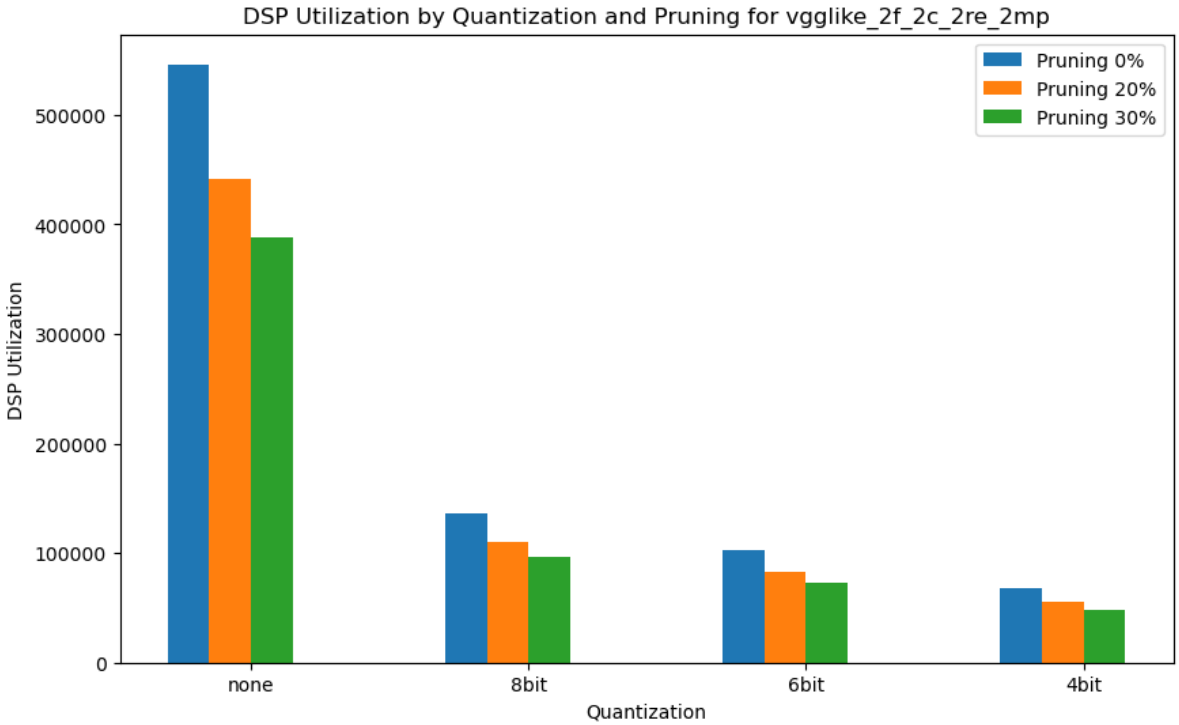


Figure 14. VGGLike_5f_5c_4re_4mp DSP Utilization at Various Quantization and Pruning Rates.

Fig. 13 and 14 show DSP utilization for the largest and second-smallest models. Hardware utilization results for flipflops, look-up tables, and block RAM were similar, predictably due to the way these resources were estimated.

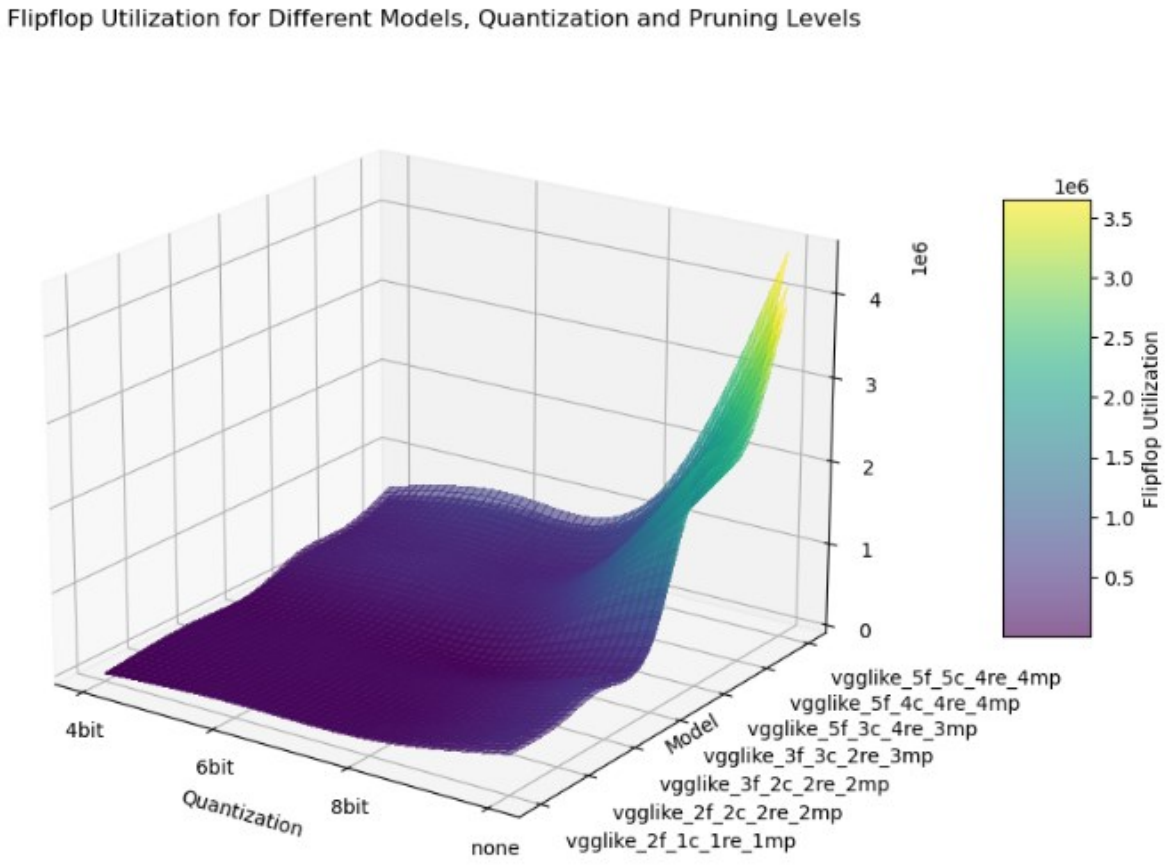


Figure 15. All Models Compared – Flipflop Utilization and Quantization, Sorted by Model Architecture.

In Fig. 15, similar estimated utilization reductions can be viewed for flipflops. This is not surprising, but does visualize the major impact quantization can have on the model's size. The visualizations for lookup tables and BRAM (Fig. 16 and 17) are similar. As with the previous 3-D graph, each layer is a different pruning rate.

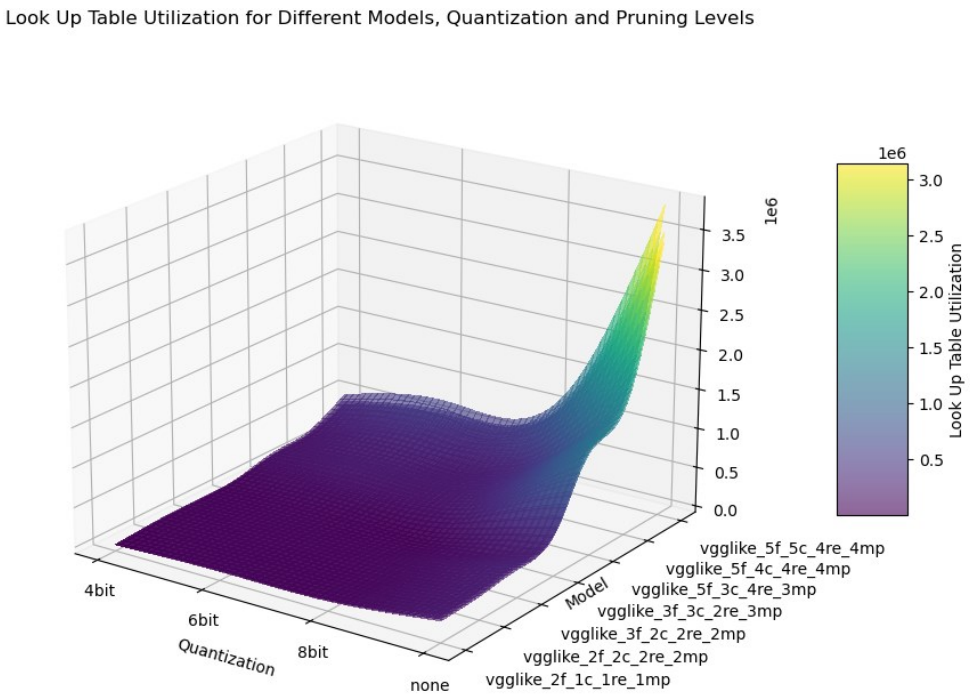


Figure 16. All Models Compared – Look-Up Table Utilization and Quantization, Sorted by Model Architecture.

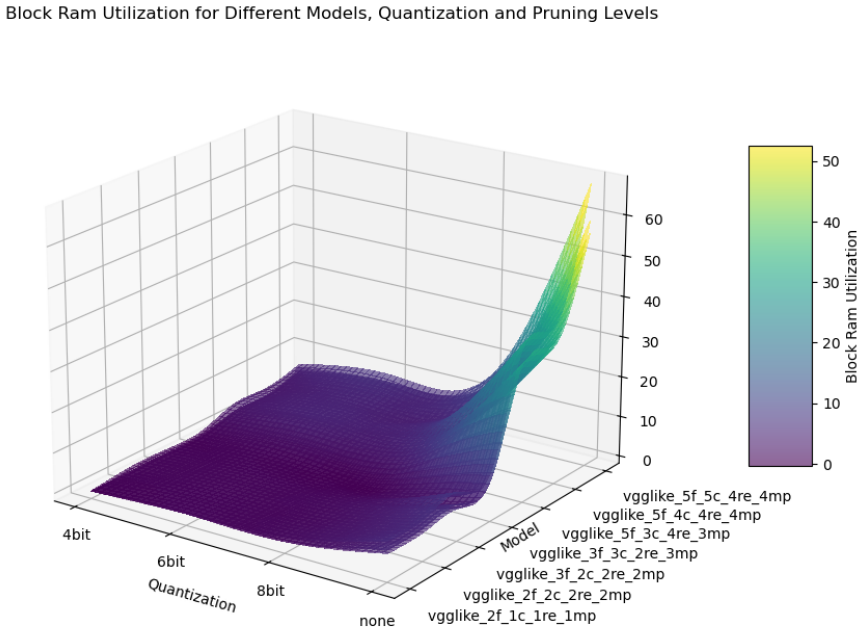


Figure 17. All Models Compared – BRAM Utilization and Quantization, Sorted by Model Architecture.

Appendices A, B and C contain tabulated data that shows benchmarking and utilizations for models with no pruning, 20% pruning, and 30% pruning, respectively. All of the numerical data used to create these visualizations is listed there, for a more detailed look on the data that was used to make these figures. Lastly, Appendices D through J hold all of the 2-D graphs for the various metrics held to the same scale, as to provide a more consistent visualization of how quantization and pruning affected the benchmarks of the different model architectures.

Chapter 6

Conclusion and Future Work

Automatic Modulation Recognition (AMR) is a crucial function in many radio receiver systems, allowing the system to classify the incoming signal and decode the data being transmitted. CNNs are a powerful tool for many classification tasks such as this one, and can be used in place of algorithms to serve this function with a high level of performance. For resource-constrained applications such as FPGAs, the size of CNNs can quickly become an issue. This paper explored the combination of various techniques to reduce the implementation size - primarily by combining quantization, pruning, and reduced model size. Quantization, the reduction in size (and precision) of weights and activation values, was one method. Pruning, the removal of vestigial or low-contribution weights and connections within a model, was another. By combining these techniques with model simplification, this thesis developed compact and efficient CNN models that maintained a high degree of accuracy and while reducing implementation size, and explored the tradeoffs while doing so.

The hardware benefits of quantization are well-established, but pruning less so, since synthesizers may not be able to take advantage of the opportunity to discard pruned weights – if the synthesizer cannot handle this, the input data will still need to pass through these zeroed layers as they are still connected. There are methodologies proposed to take advantage of pruning, such as spare matrix operations, but they are not widely available or utilized.

The benefits of pruning, while minor in comparison to quantization when viewing through the lens of resource utilization, become more apparent when the model is being trained. Being able to drop the lowest-scoring weights allows the model to converge much faster on a desired loss value when training, making it a good technique to combine with quantization as a more heavily quantized model may struggle to reach its final loss value. Even if a specific FPGA cannot fully realize the benefits of pruning for implementation size, the benefits it offers in conjunction with quantization could potentially result in even more hardware savings. Additionally, with smaller model architectures being the most optimal, it was noted that convergence during training was sometimes easier when pruning was applied, and the model was retrained. While the benefit was difficult to observe as the models tested were already small, this may have a greater benefit when scaled up to more difficult classification problems (and thus, greater overall model sizes).

Another very noticeable benefit of pruning was in throughput and inference time – the speed at which the model processes inputs. The best performance came from models that were the smallest and most highly pruned – suggesting that pruning is a “free” source of both performance and utilization improvements. It can also be used to offset the performance downsides of quantization, which may be unavoidable to the designer due to utilization requirements but should not be overlooked when trying to get the best performance possible.

While this work focused primarily on evaluating reduced-size models for modulation classification, many promising research directions remain open. Implementing Python libraries that take advantage of sparse matrix operations and can better handle discarding weights lost through structural pruning would be an excellent first step to evaluating the actual performance and implementation improvements that combining structural pruning, quantization, and model simplification can provide. There are several being worked on, but no official tool seems to exist. Examining actual implementations on FPGAs, as opposed to just estimations, would be valuable as well if combined with a synthesizer that can also take advantage of structural pruning. Measuring real-world performance of these techniques and comparing them with their software benchmarks would offer more insight on the actual implementation gains that are achieved by quantization and structured pruning. Both real-world measurements and effective sparse-matrix implementation would produce tangible benefits in efficient CNN hardware implementation.

References

[1] S. Kumar, R. Mahapatra and A. Singh, "Automatic Modulation Recognition: An FPGA Implementation," in *IEEE Communications Letters*, vol. 26, no. 9, pp. 2062-2066, Sept. 2022, doi: 10.1109/LCOMM.2022.3184771.

[2] F. N. Iandola, S. Han, H. M. Moskewicz, K. Ashraf, W. J. Dally and K. Keutzer, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size”, arXiv:1602.07360, 2016.

[3] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both Weights and Connections for Efficient Neural Networks," in Advances in Neural Information Processing Systems 28 (NIPS 2015), 2015, pp. 1135-1143.

[4] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing DNNs with Pruning, Trained Quantization and Huffman Encoding," in Proceedings of the 4th International Conference on Learning Representations (ICLR 2016), 2016.

[5] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16), 2016, pp. 243-254.

[6] P. M. Gysel, “Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks”, M.S. thesis, University of California Davis, 2016.

[7] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15), 2015, pp. 161-170.

[8] 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, no. 12, p. 441, Dec. 2022, doi: 10.3390/a15120441.

[9] D. Chauhan and V. Kanwar, “Comparision of BER's of QAM and PSK Modulation Techniques for Channel Estimation by Using LS Estimator in MIMO-OFDM System”, International Journal of Advanced Research in Computer and Communication Engineering, vol. 6, no. 3, pp. 825-832, 2017 , doi: 10.17148/IJARCCE.2017.63194.

[10] O. A. Dobre, A. Abdi, Y. Bar-Ness and W. Su, “A Survey of Automatic Modulation Classification Techniques: Classical Approaches and New Trends”, IET Communications, vol. 1, no. 2, pp. 137-156, April 2007, doi: 10.1049/iet-com:20050176.

[11] M. A. Abdel-Moneim, W. El-Shafai, N. Abdel-Salam, E.-S. M. El-Rabaie, and F. E. Abd El-Samie, "A Survey of Traditional and Advanced Automatic Modulation Classification Techniques, Challenges, and Some Novel Trends," International Journal of Communication Systems, vol. 34, no. 10, pp. 1-36, May 2021, doi: 10.1002/dac.4762.

[12] J. Schmidhuber, "Annotated History of Modern AI and Deep Learning," IDSIA, Tech. Rep. IDSIA-22-22, 2022, arXiv preprint arXiv:2212.11279.

[13] D. F. Specht, "A general regression neural network," in *IEEE Transactions on Neural Networks*, vol. 2, no. 6, pp. 568-576, Nov. 1991, doi: 10.1109/72.97934.

[14] K. Leung, "How to Easily Draw Neural Network Architecture Diagrams," Towards Data Science, Aug. 23, 2021. [Online]. Available: https://towardsdatascience.com/how-to-easily-draw-neural-network-architecture-diagrams-a6b6138ed875.

[15] R. Y. Choi, A. S. Coyner, J. Kalpathy-Cramer, M. F. Chiang, and J. P. Campbell, "Introduction to Machine Learning, Neural Networks, and Deep Learning," Translational Vision Science & Technology, vol. 9, no. 2, pp. 14, Feb. 2020, doi: 10.1167/tvst.9.2.14.

[16] A. Griffin, "Google and ChatGPT face major threat from open source community, leaked document warns," The Independent, May 5, 2023. [Online]. Available: https://www.independent.co.uk/tech/google-chatgpt-open-source-leaked-document-b2333287.html. [Accessed May 5, 2023].

[17] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015.

[18] D. G. Altman and J. M. Bland, "Standard deviations and standard errors," BMJ, vol. 331, no. 7521, pp. 903, Oct. 2005, doi: 10.1136/bmj.331.7521.903.

[19] J. L. Fleiss, B. Levin, and M. C. Paik, Statistical Methods for Rates and Proportions, 3rd ed. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2003, doi: 10.1002/0471445428.

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

[21] S. Peng, H. Jiang, H. Wang, H. Alwageed, Y. Zhou, M. 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. 30, no. 3, pp. 718-727, March 2019, doi: 10.1109/TNNLS.2018.2850703.

[22] C. Zhu, K. Huang, S. Yang, Z. Zhu, H. Zhang and H. Shen, "An Efficient Hardware Accelerator for Structured Sparse Convolutional Neural Networks on FPGAs," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 9, pp. 1953-1965, Sept. 2020, doi: 10.1109/TVLSI.2020.3002779.

[23] A. Rahman, J. Lee and K. Choi, "Efficient FPGA acceleration of Convolutional Neural Networks using logical-3D compute array," *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, Germany, 2016, pp. 1393-1398.

Appendix A: Tabulated Model Benchmarks, 0% Pruning

Model	Throughput (smpls/sec)	Inference Time (ms)	DSP	FF	LUT	BRAM
vgglike_5f_5c_4re_4mp	42515.57589	0.75266533	4520576	4523497	3827984	68
vgglike_5f_4c_4re_4mp	43475.39937	0.73604844	2685568	2687977	1730832	38
vgglike_5f_3c_4re_3mp	42560.41057	0.75187245	2488960	2491113	1337516	37
vgglike_3f_3c_2re_3mp	64163.34569	0.49872711	719488	720361	452780	10
vgglike_3f_2c_2re_2mp	69683.10614	0.45922178	670336	671081	354376	10
vgglike_2f_2c_2re_2mp	46037.42437	0.69508667	546176	546793	292296	8
vgglike_2f_1c_1re_1mp	60806.95937	0.52625555	533888	534441	267620	8
vgglike_5f_5c_4re_4mp_8bit	2271.358926	14.0884823	1130144	1130874	956996	14
vgglike_5f_4c_4re_4mp_8bit	3504.610623	9.13082891	671392	671994	432708	9
vgglike_5f_3c_4re_3mp_8bit	3579.024773	8.9409831	622240	622778	334379	9
vgglike_3f_3c_2re_3mp_8bit	4778.816763	6.69621825	179872	180090	113195	2
vgglike_3f_2c_2re_2mp_8bit	5901.700326	5.4221662	167584	167770	88594	2
vgglike_2f_2c_2re_2mp_8bit	6286.205148	5.09051156	136544	136698	73074	2
vgglike_2f_1c_1re_1mp_8bit	7624.456587	4.19702042	133472	133610	66905	2
vgglike_5f_5c_4re_4mp_6bit	2994.663773	10.6856737	847608	848155	717744	9
vgglike_5f_4c_4re_4mp_6bit	3737.778806	8.56123427	503544	503995	324528	6
vgglike_5f_3c_4re_3mp_6bit	4397.859183	7.27626754	466680	467083	250782	6
vgglike_3f_3c_2re_3mp_6bit	4339.754317	7.37368931	134904	135067	84894	1
vgglike_3f_2c_2re_2mp_6bit	5939.85795	5.38733422	125688	125827	66444	1
vgglike_2f_2c_2re_2mp_6bit	6509.589044	4.91582491	102408	102523	54804	1
vgglike_2f_1c_1re_1mp_6bit	7447.408118	4.29679688	100104	100207	50178	1
vgglike_5f_5c_4re_4mp_4bit	3225.036173	9.92236933	565072	565437	478496	6
vgglike_5f_4c_4re_4mp_4bit	3571.89384	8.95883289	335696	335997	216352	4
vgglike_5f_3c_4re_3mp_4bit	3707.013324	8.63228621	311120	311389	167188	4
vgglike_3f_3c_2re_3mp_4bit	4438.620262	7.20944756	89936	90045	56596	1
vgglike_3f_2c_2re_2mp_4bit	5735.850701	5.57894577	83792	83885	44296	1
vgglike_2f_2c_2re_2mp_4bit	6870.390966	4.65766798	68272	68349	36536	1

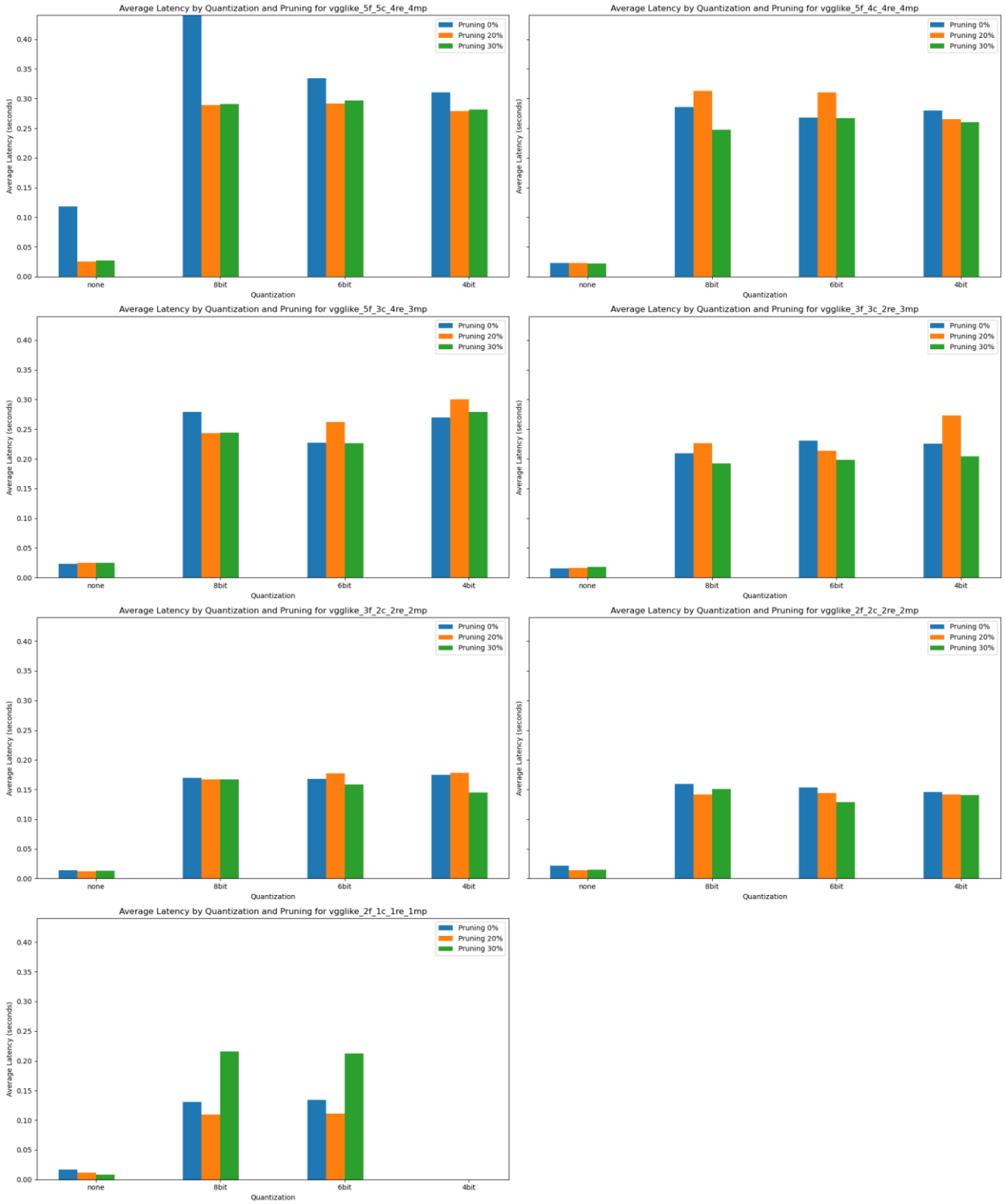
Appendix B: Tabulated Model Benchmarks, 20% Pruning

Model	Throughput (smpls/sec)	Inference Time (ms)	DSP	FF	LUT	BRAM
vgglike_5f_5c_4re_4mp_pr_20	38870.94993	0.8232369	4100680	4103601	3492195	59
vgglike_5f_4c_4re_4mp_pr_20	43908.16842	0.7287938	2467102	2469511	1611759	34
vgglike_5f_3c_4re_3mp_pr_20	40183.21292	0.7963524	2326774	2328927	1255670	33
vgglike_3f_3c_2re_3mp_pr_20	61449.43866	0.5207533	683444	684317	433753	8
vgglike_3f_2c_2re_2mp_pr_20	81126.02952	0.394448	558112	558857	298249	7
vgglike_2f_2c_2re_2mp_pr_20	73300.72132	0.4365578	441646	442263	239968	6
vgglike_2f_1c_1re_1mp_pr_20	85931.15993	0.3723911	429436	429989	215394	6
vgglike_5f_5c_4re_4mp_8bit_pr_20	3457.265253	9.2558707	1020872	1021602	872829	12
vgglike_5f_4c_4re_4mp_8bit_pr_20	3197.104973	10.009055	612291	612893	398674	6
vgglike_5f_3c_4re_3mp_8bit_pr_20	4104.443491	7.7964284	584792	585330	315438	6
vgglike_3f_3c_2re_3mp_8bit_pr_20	4411.225465	7.25422	170939	171157	108261	1
vgglike_3f_2c_2re_2mp_8bit_pr_20	5972.038408	5.3583045	138846	139032	74225	1
vgglike_2f_2c_2re_2mp_8bit_pr_20	7081.714804	4.5186796	110412	110566	59987	1
vgglike_2f_1c_1re_1mp_8bit_pr_20	9109.093927	3.5129729	107359	107497	53848	1
vgglike_5f_5c_4re_4mp_6bit_pr_20	3426.649068	9.3385694	688967	689514	585700	7
vgglike_5f_4c_4re_4mp_6bit_pr_20	3220.484857	9.936392	503544	503995	324528	6
vgglike_5f_3c_4re_3mp_6bit_pr_20	3820.536237	8.3757876	379357	379760	206745	4
vgglike_3f_3c_2re_3mp_6bit_pr_20	4674.838357	6.8451565	112181	112344	72362	1
vgglike_3f_2c_2re_2mp_6bit_pr_20	5628.423662	5.6854284	101906	102045	54551	1
vgglike_2f_2c_2re_2mp_6bit_pr_20	6921.948775	4.6229756	82810	82925	45005	1
vgglike_2f_1c_1re_1mp_6bit_pr_20	8971.964801	3.5666658	80519	80622	40385	1
vgglike_5f_5c_4re_4mp_4bit_pr_20	3590.256219	8.9130129	457484	457849	388926	5
vgglike_5f_4c_4re_4mp_4bit_pr_20	3770.272669	8.4874498	271115	271416	175684	2
vgglike_5f_3c_4re_3mp_4bit_pr_20	3329.464365	9.6111556	252833	253102	137443	2
vgglike_3f_3c_2re_3mp_4bit_pr_20	3662.806307	8.7364707	75319	75428	48873	0
vgglike_3f_2c_2re_2mp_4bit_pr_20	5604.496711	5.7097009	68587	68680	36693	0
vgglike_2f_2c_2re_2mp_4bit_pr_20	7059.077542	4.5331702	55200	55277	30000	0

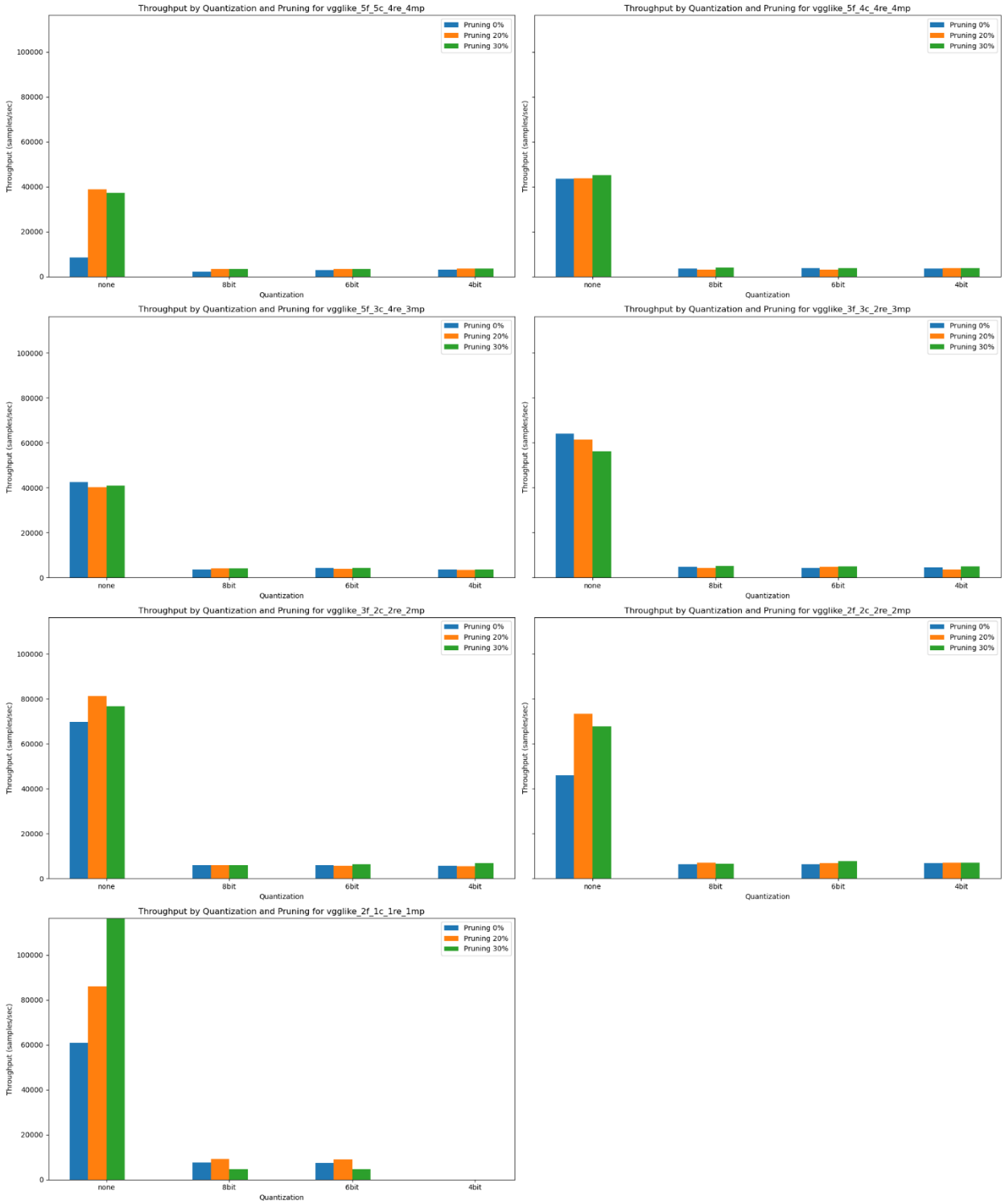
Appendix C: Tabulated Model Benchmarks, 30% Pruning

Model	Throughput (smpls/sec)	Inference Time (ms)	DSP	FF	LUT	BRAM
vgglike_5f_5c_4re_4mp_pr_30	37357.05082	0.8565987	3921346	3924267	3372933	56
vgglike_5f_4c_4re_4mp_pr_30	45075.67083	0.7099173	2368706	2371115	1554533	33
vgglike_5f_3c_4re_3mp_pr_30	40861.68214	0.7831298	2274602	2276755	1230313	32
vgglike_3f_3c_2re_3mp_pr_30	56124.85914	0.5701573	670438	671311	427970	8
vgglike_3f_2c_2re_2mp_pr_30	76768.47387	0.4168378	501186	501931	269675	6
vgglike_2f_2c_2re_2mp_pr_30	67727.16125	0.472484	388256	388873	213054	5
vgglike_2f_1c_1re_1mp_pr_30	116310.6271	0.2751253	376186	376739	188769	5
vgglike_5f_5c_4re_4mp_8bit_pr_30	3442.857481	9.2946049	984448	985178	845998	12
vgglike_5f_4c_4re_4mp_8bit_pr_30	4039.304879	7.9221552	595788	596390	392452	6
vgglike_5f_3c_4re_3mp_8bit_pr_30	4093.893893	7.8165191	565397	565935	305848	6
vgglike_3f_3c_2re_3mp_8bit_pr_30	5199.511001	6.1544249	167006	167224	106431	1
vgglike_3f_2c_2re_2mp_8bit_pr_30	5990.637622	5.3416685	125122	125308	67361	1
vgglike_2f_2c_2re_2mp_8bit_pr_30	6642.894936	4.8171769	97084	97238	53307	1
vgglike_2f_1c_1re_1mp_8bit_pr_30	4639.921786	6.896668	94046	94184	47192	1
vgglike_5f_5c_4re_4mp_6bit_pr_30	3369.401223	9.4972364	618622	619169	533372	7
vgglike_5f_4c_4re_4mp_6bit_pr_30	3745.555469	8.5434591	380459	380910	252706	4
vgglike_5f_3c_4re_3mp_6bit_pr_30	4419.803478	7.2401409	341611	342014	187390	4
vgglike_3f_3c_2re_3mp_6bit_pr_30	5035.425974	6.3549738	109904	110067	71919	1
vgglike_3f_2c_2re_2mp_6bit_pr_30	6318.503118	5.0644907	91765	91904	49482	1
vgglike_2f_2c_2re_2mp_6bit_pr_30	7754.058481	4.1268711	72817	72932	40006	1
vgglike_2f_1c_1re_1mp_6bit_pr_30	4715.629233	6.7859449	70536	70639	35394	1
vgglike_5f_5c_4re_4mp_4bit_pr_30	3555.162991	9.0009938	403783	404148	344337	4
vgglike_5f_4c_4re_4mp_4bit_pr_30	3845.95414	8.320432	242682	242983	160765	2
vgglike_5f_3c_4re_3mp_4bit_pr_30	3576.935028	8.9462067	235009	235278	129074	2
vgglike_3f_3c_2re_3mp_4bit_pr_30	4886.274699	6.5489564	74157	74266	48486	0
vgglike_3f_2c_2re_2mp_4bit_pr_30	6899.709485	4.6378764	60405	60498	32601	0
vgglike_2f_2c_2re_2mp_4bit_pr_30	7121.460947	4.49346	48540	48617	26666	0

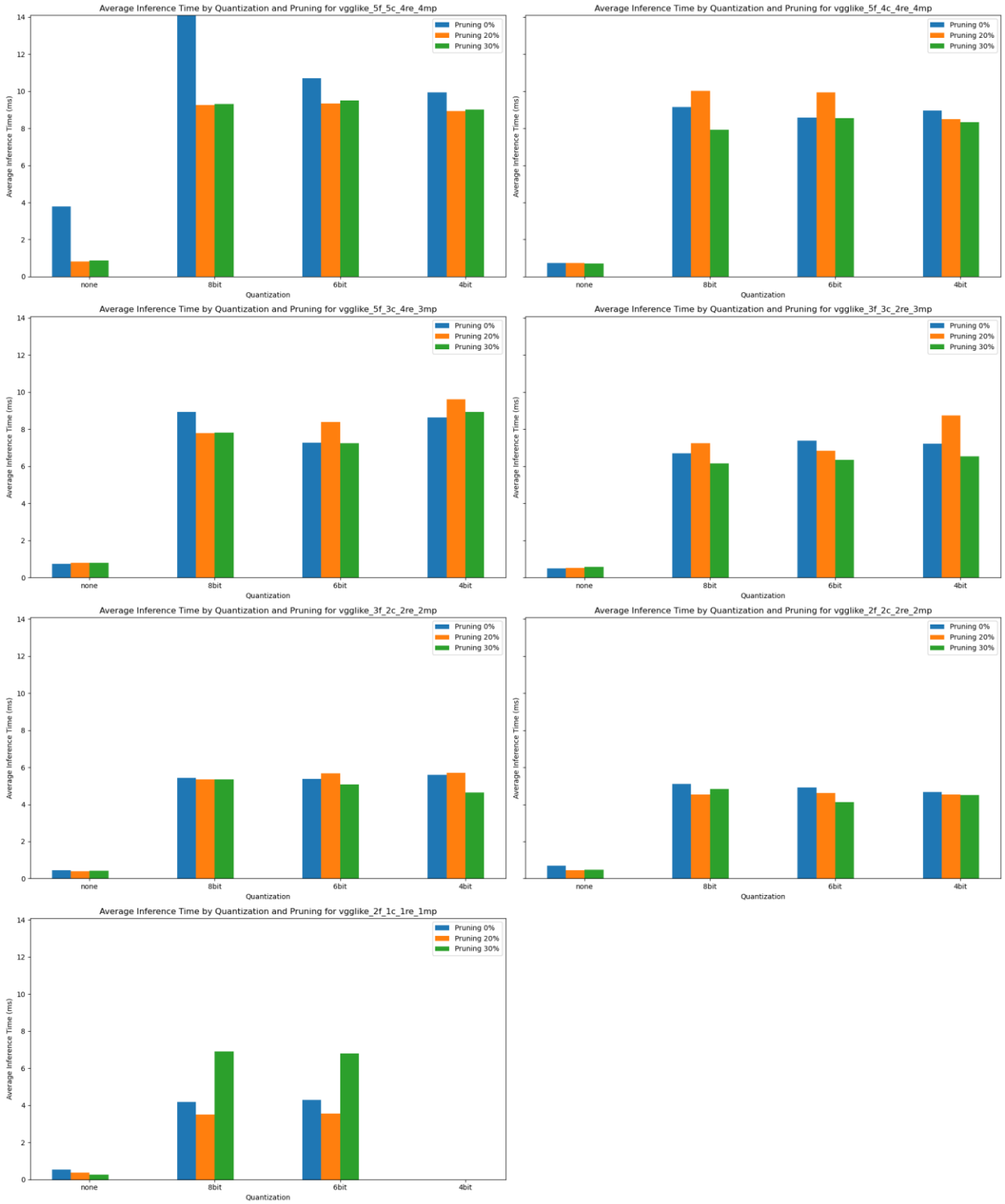
Appendix D: Average Training Latency, All Models



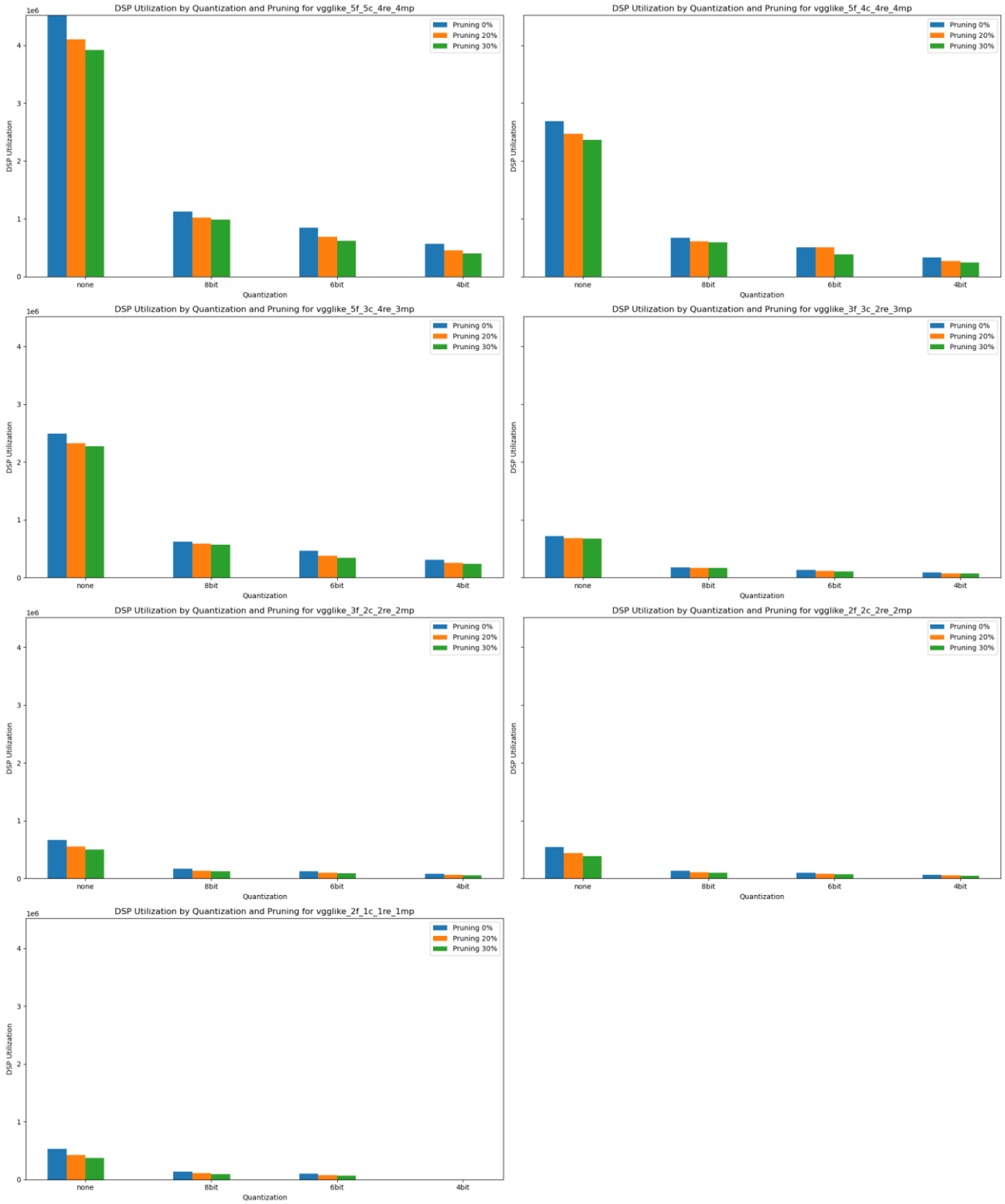
Appendix E: Average Evaluation Throughput, All Models



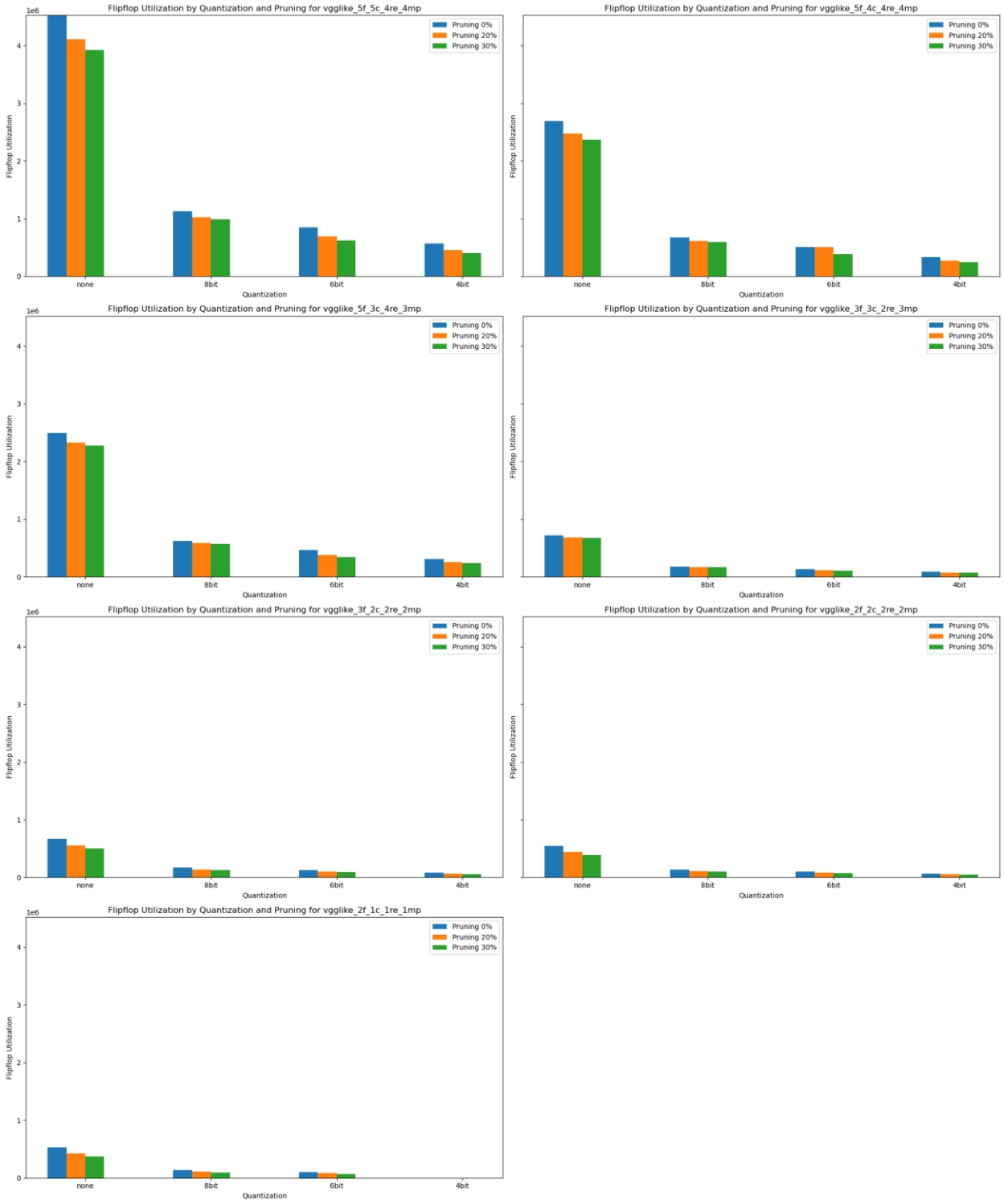
Appendix F: Average Inference Time (Evaluation), All Models



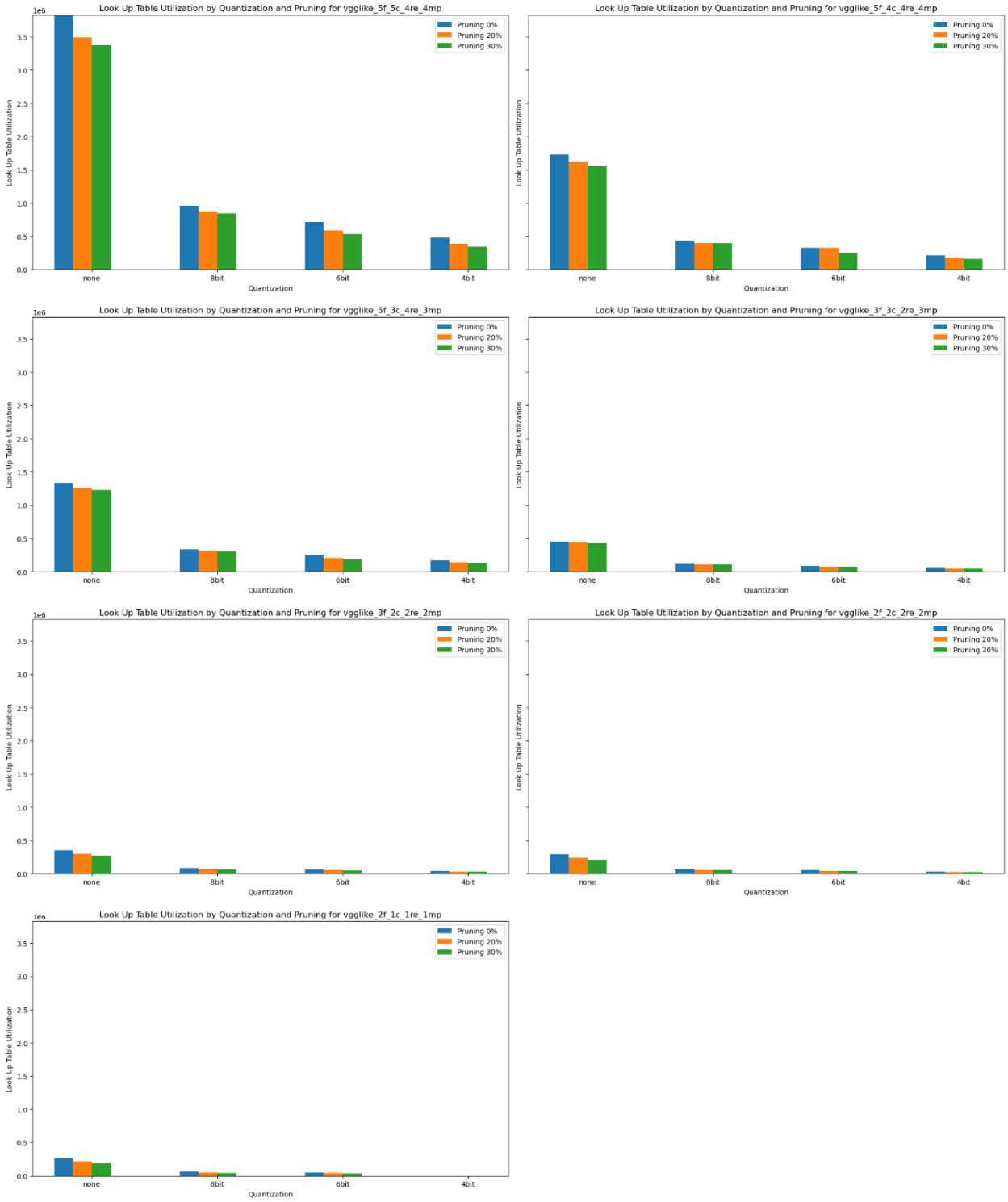
Appendix G: DSP Utilization, All Models



Appendix H: Flipflop Utilization, All Models



Appendix I: Look-up Table Utilization, All Models



Appendix J: Block RAM Utilization, All Models