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

JOSHUA ROTHE

JOHNS HOPKINS WHITING SCHOOL OF ENGINEERING

M.S. THESIS PRESENTATION, 2ND SEMESTER

MAY 2024

TABLE OF CONTENTS

INTRODUCTION

▪BACKGROUND

Automatic Modulation Recognition (AMR) – detects the modulation of a radio frequency (RF) signal. Can be computationally expensive.

Older method: Two-stage system (feature extraction and classification) using various algorithms.

Newer method: Convolutional Neural Networks (CNNs) – using convolutional layers and fully connected layers to classify signal modulation types with no loss of accuracy.

▪PROBLEM

CNNs are resource intensive and thus require specialized (larger) hardware, which does not fit most RF receiver hardware applications.

Schemes to lower resource utilization of models are necessary for a realistic implementation of AMR functions using CNNs.

PRIOR WORK

- **S. KUMAR ET. AL. [1]**

Quantization, low-precision math, residual unit scheme and iterative pruning.

- **HAN ET. AL. [2][3][4]**

SqueezeNet [2], a modification of AlexNet – shrinks filter sizes, input channels, and downsampling.
Deep Compression [3] – SqueezeNet combined with quantization.
EIE Hardware Accelerator [4] - preferring SRAM over DRAM and using ALUs.

- **P. M. GYSEL [5]**

Ristretto – Quantizing parameters and outputs, implemented ReLU and Max Pooling layers.

- **C. ZHANG ET. AL. [6]**

CNN implantation on FPGAs.

- **D. GÓEZ ET. AL. [7]**

AMR implementation on one-dimensional CNNs, using quantization on different layers.

MOTIVATION

- Various methodologies can be used to reduce model size, combining several methods may prove particularly useful.
- Other work Usually focuses on One or two methodologies, and may control other variables for optimal or more noteworthy results.
- Goal is to combine these methodologies together in an unbiased way, and attempt to analyze what works best for performance as well as resource utilization while demonstrating no loss of accuracy.

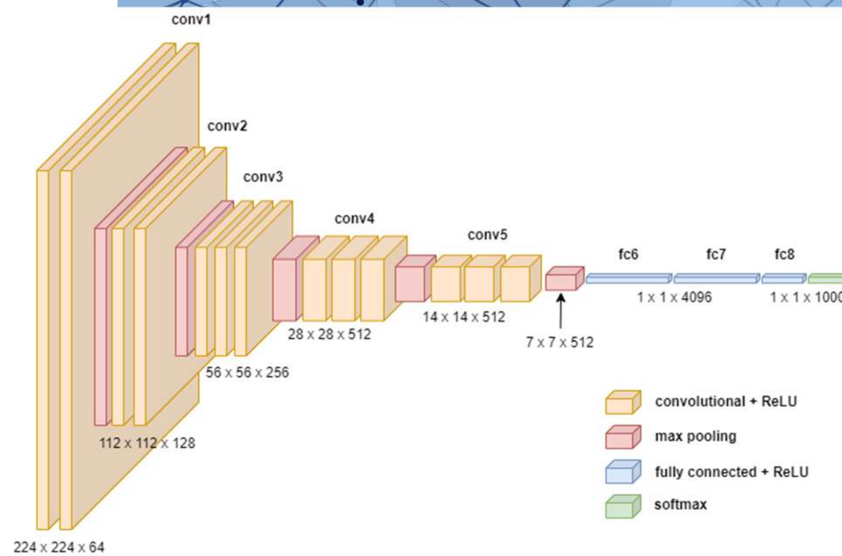
PROPOSED WORK (1/2)

▪COMBINATION OF METHODOLOGIES

1. Bit quantization
2. Pruning
3. ReLU layers
4. Max pooling layers
5. One-dimensional CNNs
6. Smaller model architectures (less overall layer count)
7. Data preprocessing (normalized I and Q values)

▪BENCHMARKING

Performance of various model architectures, quantization rates, and pruning rates while maintaining full accuracy (100% when evaluated against ~2,000 unique data input sets).




```
print('Accuracy of the model on test data: %.2f %%' % accuracy)
print('Average latency: %.2f milliseconds/input' % average_latency)
print('Throughput: %.2f inputs/second' % throughput)

Accuracy of the model on test data: 87.50 %
Average latency: 0.12 milliseconds/input
Throughput: 8431.23 inputs/second
```

PROPOSED WORK (2/2)

▪Signal Generation

Generate preprocessed data inputs.

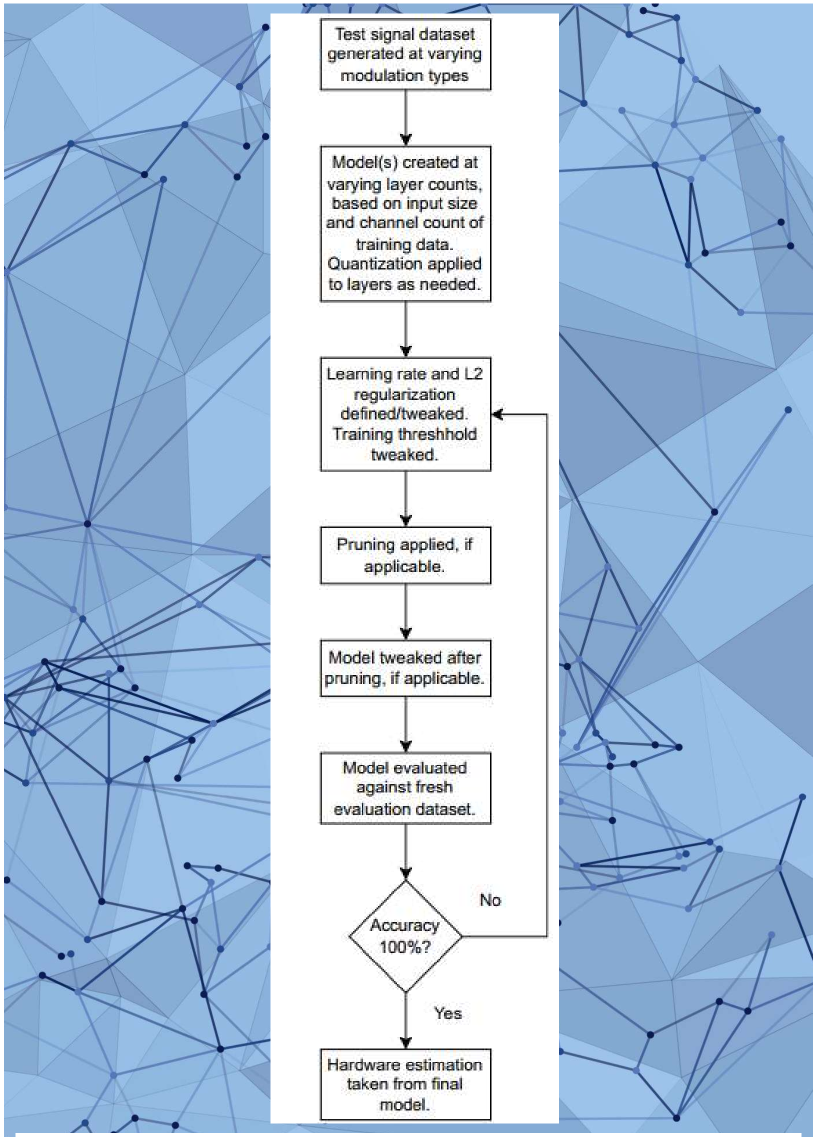
▪Model Generation

Use these inputs to train models with various architectures and quantization rates (including none). Implement pruning at various rates (including none) and re-train as needed.

Evaluate model performance on a unique, newly-generated dataset to avoid detecting overfitting. Tweak parameters as needed for 100% accuracy on the evaluation dataset.

▪Benchmarking

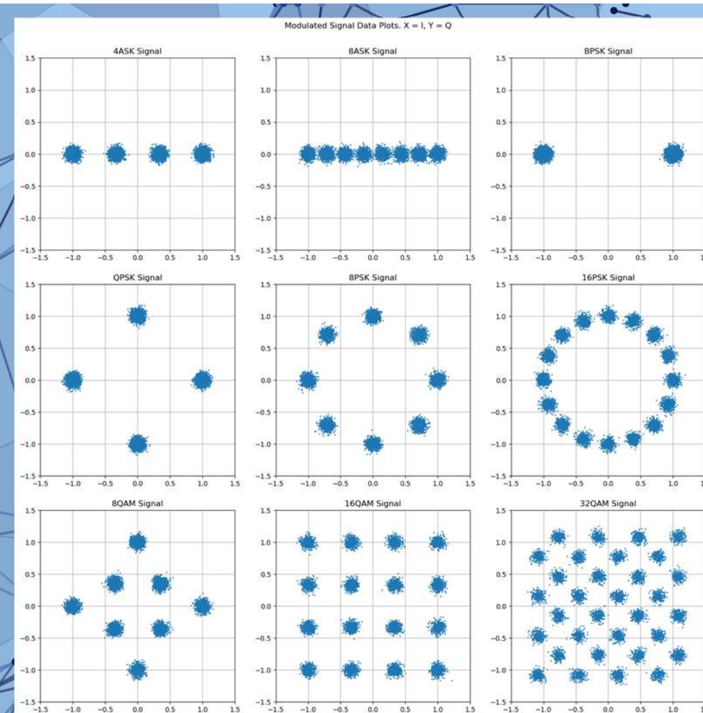
Estimate hardware resource utilization and attempt to measure performance increases.

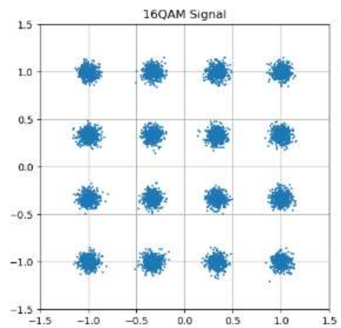


Workflow for Model Creation and Benchmarking

METHODOLOGY – SIGNAL GENERATION (1/2)

- Generate normalized I and Q signal values for various digital modulation types.



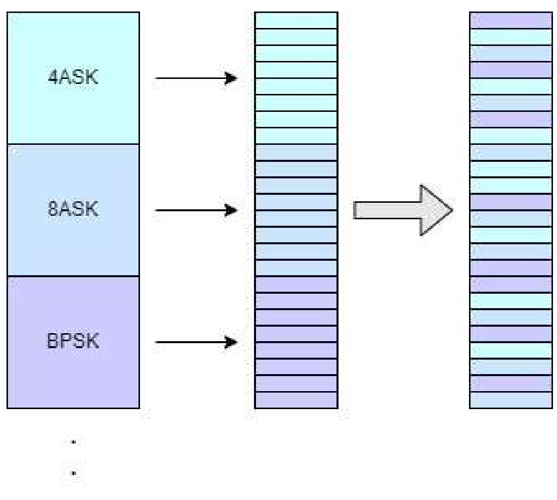



```python
def generate_16qam(self):
    """
    Function for generating data points for a 16-QAM signal.
    """
    i_levels = np.array([-3, -1, 1, 3])
    q_levels = np.array([-3, -1, 1, 3])

    i_points = np.random.choice(i_levels, self.num_symbols)
    q_points = np.random.choice(q_levels, self.num_symbols)

    symbols = i_points/3.0 + 1j * q_points/3.0

    awgn_noise, phase_noise = self.generate_noise(self.num_symbols)
    final_signal = self.generate_signal(symbols, awgn_noise, phase_noise)

    train_data = [[sig.real, sig.imag, '16qam'] for sig in final_signal]

    return train_data, final_signal
```

```python
def generate_noise(self, num_symbols):
    """
    Generates awgn noise and phase noise.
    """
    awgn_noise = (np.random.randn(num_symbols) +
                  1j*np.random.randn(num_symbols))/np.sqrt(2)
    phase_noise = np.random.randn(num_symbols) * self.phase_noise_power
    return awgn_noise, phase_noise
```

METHODOLOGY – SIGNAL GENERATION (2/2)

- Datasets are "chunked" into sets of 32, with the assumption that the model can see 32 I and Q values of the same modulation type before needing to identify the modulation type.
- Dataset chunks are shuffled, with one label carried per chunk.
- Chunks are organized to be fed into models for training and evaluation.



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.

METHODOLOGY – MODEL GENERATION (1/2)

- Create 1-D CNNs of various layer counts (convolutional, fully connected, ReLU, and max pooling).
- Train these models, using a generated training dataset. Tweak learning parameters, epoch count, and loss function threshold as needed. Model iteratively trains over the dataset to reach desired loss threshold.

```python
class VGGlike(nn.Module):
    def __init__(self):
        super(VGGlike, self).__init__()

        self.conv1 = nn.Conv1d(2, 32, kernel_size=3, stride=1, padding=1)
        self.maxpool1 = nn.MaxPool1d(2)
        self.conv2 = nn.Conv1d(32, 64, kernel_size=3, stride=1, padding=1)
        self.maxpool2 = nn.MaxPool1d(2)
        self.conv3 = nn.Conv1d(64, 128, kernel_size=3, stride=1, padding=1)
        self.maxpool3 = nn.MaxPool1d(2)
        self.conv4 = nn.Conv1d(128, 256, kernel_size=3, stride=1, padding=1)
        self.maxpool4 = nn.MaxPool1d(2)
        self.conv5 = nn.Conv1d(256, 512, kernel_size=3, stride=1, padding=1)

        self.fc1 = nn.Linear(512 * 2, 1024)
        self.fc2 = nn.Linear(1024, 512)
        self.fc3 = nn.Linear(512, 256)
        self.fc4 = nn.Linear(256, 128)
        self.fc5 = nn.Linear(128, 9)  # 9 classes.

    def forward(self, x):
        x = self.maxpool1(F.relu(self.conv1(x)))
        x = self.maxpool2(F.relu(self.conv2(x)))
        x = self.maxpool3(F.relu(self.conv3(x)))
        x = self.maxpool4(F.relu(self.conv4(x)))
        x = F.relu(self.conv5(x))
        x = x.view(x.size(0), -1)  # Flatten the tensor.
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = self.fc5(x)

        return x
```

```python
import time

# Train your model.
num_epochs = 1000
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # Use GPU if available.
model = model.to(device)

# Training loop.
for epoch in range(num_epochs):
    running_loss = 0.0
    total_batches = 0
    total_samples = 0
    total_time = 0.0
    epoch_start = time.time() # Throughput calc.

    for i, data in enumerate(data_loader, 0):
        # Extract inputs and labels from data.
        inputs, labels = data

        # Move inputs and labels to device.
        inputs = inputs.float()
        inputs = inputs.to(device)
        labels = labels.to(device)

        batch_start = time.time() # Start timing for latency calc.

        # Zero the parameter gradients.
        optimizer.zero_grad()

        # Forward + backward + optimize.
        outputs = model(inputs)  # outputs shape is (batch_size, num_classes).
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```python
        # Forward + backward + optimize.
        outputs = model(inputs)  # outputs shape is (batch_size, num_classes).
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        batch_end = time.time()
        total_time += batch_end - batch_start

        # Print statistics.
        running_loss += loss.item()
        total_batches += 1
        total_samples += inputs.size(0) # Add num of samples to the batch.

    epoch_end = time.time() # End timing for throughput calc.
    elapsed_time = epoch_end - epoch_start
    throughput = total_samples / elapsed_time # Samples processed per sec.

    # Print average loss per epoch. Note - using latency and throughput values from final epoch.
    average_loss = running_loss / total_batches
    average_latency = total_time / total_batches
    print(f'[Epoch {epoch + 1}] Average loss: {average_loss:.3f}, Average latency: {average_latency:.3f} seconds, Throughput

    # Stop training when average loss is below the threshold.
    if average_loss < 0.040:
        print('Training stopped - average loss is less than 0.040.')
        break

print('Finished Training')
```

```
[Epoch 1] Average loss: 1.894, Average latency: 0.015 seconds, Throughput: 1916.99 samples/sec
[Epoch 2] Average loss: 1.318, Average latency: 0.003 seconds, Throughput: 7253.61 samples/sec
[Epoch 3] Average loss: 1.200, Average latency: 0.003 seconds, Throughput: 7474.36 samples/sec
[Epoch 4] Average loss: 1.158, Average latency: 0.003 seconds, Throughput: 8003.94 samples/sec
[Epoch 5] Average loss: 1.196, Average latency: 0.003 seconds, Throughput: 7444.11 samples/sec
[Epoch 6] Average loss: 1.077, Average latency: 0.003 seconds, Throughput: 6803.74 samples/sec
[Epoch 7] Average loss: 1.035, Average latency: 0.003 seconds, Throughput: 6850.67 samples/sec
[Epoch 8] Average loss: 1.056, Average latency: 0.003 seconds, Throughput: 6772.31 samples/sec
[Epoch 9] Average loss: 1.021, Average latency: 0.003 seconds, Throughput: 7397.21 samples/sec
[Epoch 10] Average loss: 1.020, Average latency: 0.003 seconds, Throughput: 7430.49 samples/sec
```

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

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

METHODOLOGY – MODEL GENERATION (2/2)

- Prune and retrain if needed.
- Evaluate against separately generated dataset, tweaking above parameters until 100% accuracy on ~2,300 data “chunks” is achieved.
- Repeat above process for all model architectures, at various quantization rates.
- Repeat THIS above process for all model architectures and quantization rates, at various pruning rates.

```
# This script implements structured pruning on the model's convolutional layers.
import torch.nn.utils.prune as prune

# Open the file and read the value
with open('..\prune_var.txt', 'r') as file:
    pruning_percentage = float(file.read())

# Now you can use pruning_percentage in your code
print(f"The pruning percentage is: {pruning_percentage*100}%")

# Define the pruning function.
def prune_model(model, amount=pruning_percentage): # Amount is percent. So 0.1 is 10%, etc.
    # List of layers to prune
    layers_to_prune = [model.conv1, model.conv2, model.conv3, model.conv4, model.conv5, model.fc1, model.fc2, model.fc3, moc
    for layer in layers_to_prune:
        # Prune the layer based on L1 norm.
        prune.ln_structured(layer, name="weight", amount=amount, n=1, dim=0)
        # Make the pruning permanent.
        prune.remove(layer, 'weight')

# Prune the model.
prune_model(model)
```

```
The pruning percentage is: 30.0%
```

```
# no_grad to save memory.
with torch.no_grad():
    for data in data_loader_e:
        # Move inputs and labels to device.
        inputs = inputs.float()
        inputs = inputs.to(device)
        labels = labels.to(device)

        # Record start time.
        start_time = torch.cuda.Event(enable_timing=True)
        end_time = torch.cuda.Event(enable_timing=True)
        start_time.record()

        outputs = model(inputs)

        # Record end time.
        end_time.record()

        # Waits for everything to finish running.
        torch.cuda.synchronize()

        inference_time = start_time.elapsed_time(end_time)
        total_time += inference_time
        num_batches_done += 1

        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        # Keep track of total inputs processed.
        total_inputs += inputs.size(0)

average_time = total_time / num_batches_done  # Average time per batch.
average_latency = total_time / total_inputs  # Average time per input.
accuracy = (100 * (correct / total))
throughput = total_inputs / (total_time / 1e3)  # Converts total time to seconds.

print('Accuracy of the model on test data: %.2f %%' % accuracy)
print('Average latency: %.2f milliseconds/input' % average_latency)
print('Throughput: %.2f inputs/second' % throughput)
```

```
Accuracy of the model on test data: 100.00 %
Average latency: 0.12 milliseconds/input
Throughput: 8449.13 inputs/second
```

METHODOLOGY – BENCHMARKING

- Standard benchmarking in python for throughput, latency, etc. is done on the evaluation dataset.
- Benchmarking for hardware resource estimation is done using a separate hardware estimation algorithm – assumptions for DSP blocks, LUTs (look-up tables), FFs (flipflops), and BRAM (block RAM) utilization is calculated based on non-zero weights (for pruning purposes), bit width (for quantization purposes), and knowledge of how these resources are generally synthesized.

Algorithm II: FPGA Hardware Resource Estimation

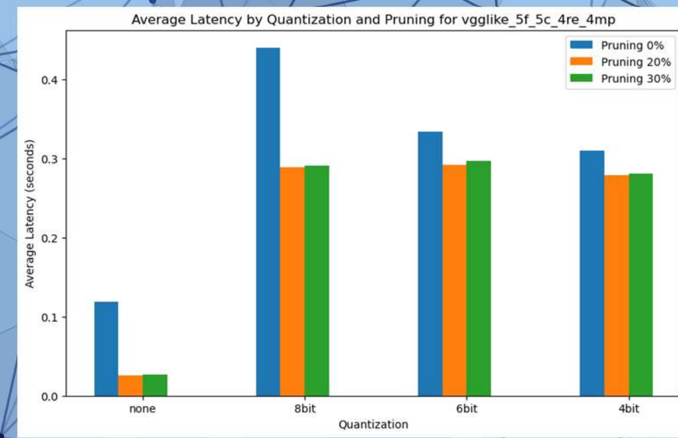
Input: Layer counts (C_C, C_L, C_M,), quantization bit width B_w, non-zero weights N_w, non-zero biases N_B, number of input and output channels for the layer C_i and C_o.

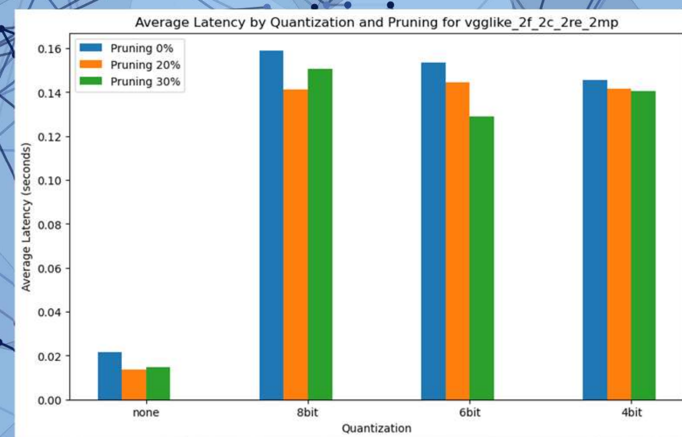
Output: Estimated utilization of FPGA resources: E_D, E_L, E_F, E_B.

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.

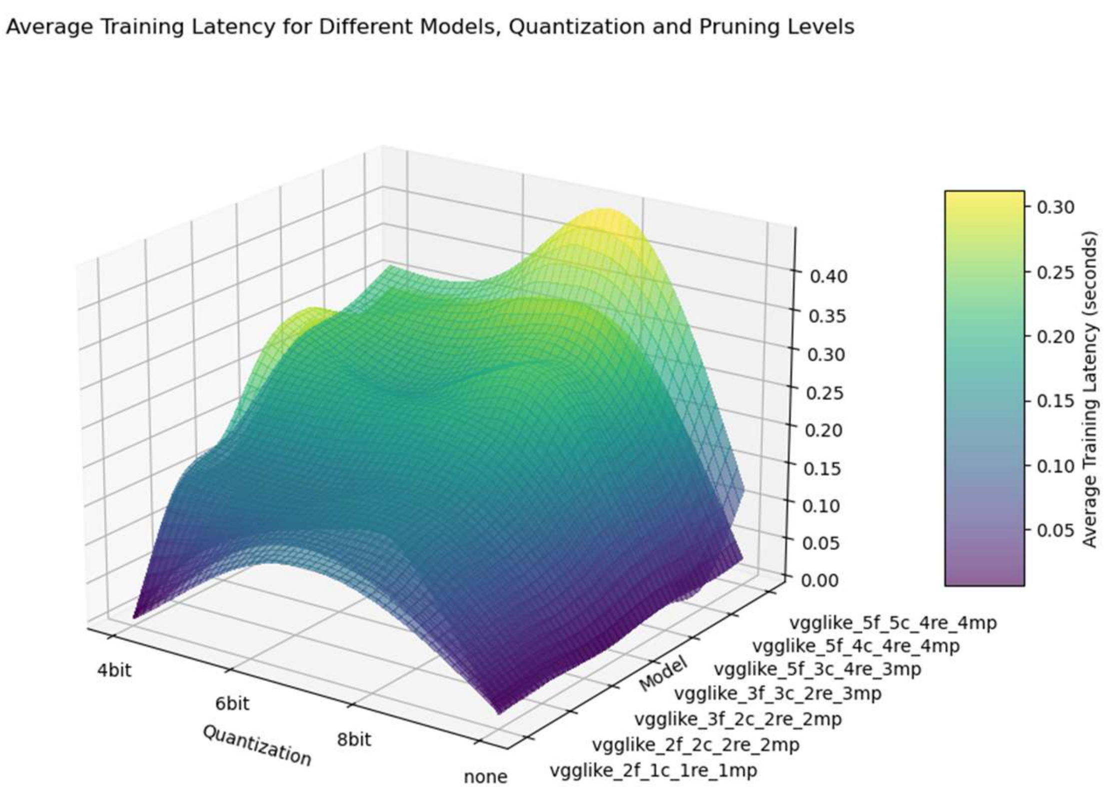
RESULTS – LATENCY (1/2)

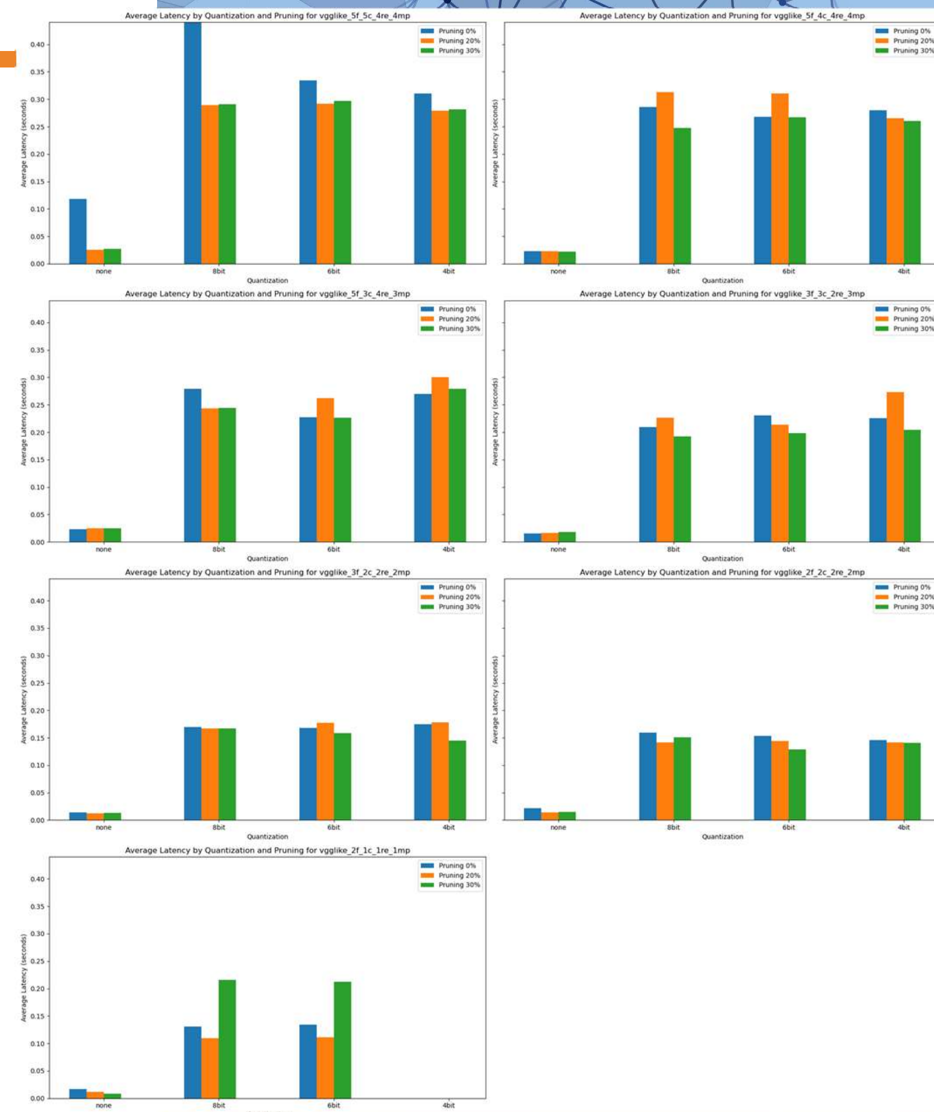
- Training latency was low for non-quantized models, lowering steadily as model architecture was lowered (250 to 200 ms) – while quantizing at any amount seemed to increase it by a factor of 4-5.
- Smaller models had worse throughput, however (16,000 samples/sec for the smallest versus 7,500 samples/sec for the larger models).
- Quantization greatly raises the latency, and shrinking the model lowers it – but quantization has a large impact on throughput, much moreso than model architecture.
- Pruning seemed to lower the latency a bit at times, but mostly negligible.





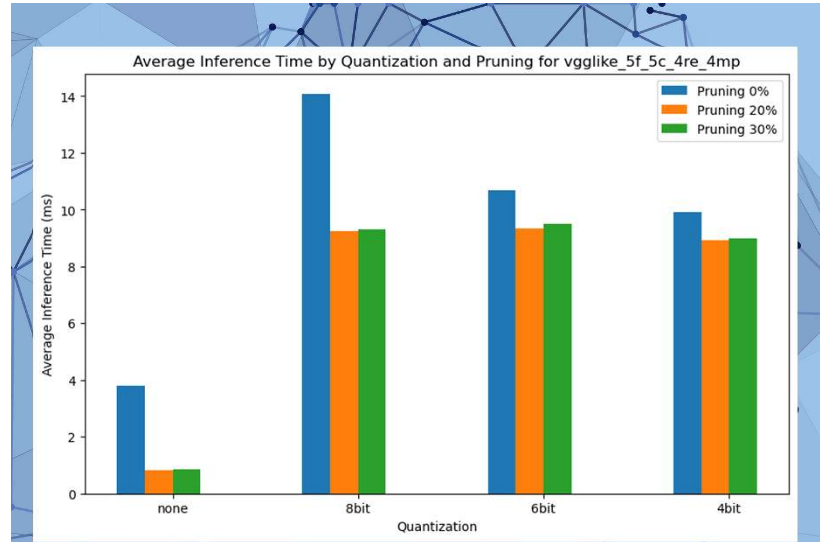
RESULTS – LATENCY (2/2)

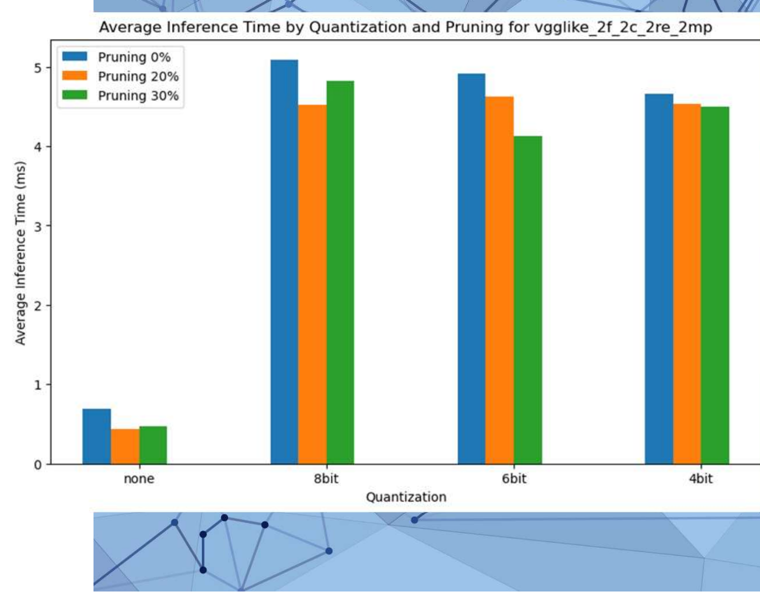




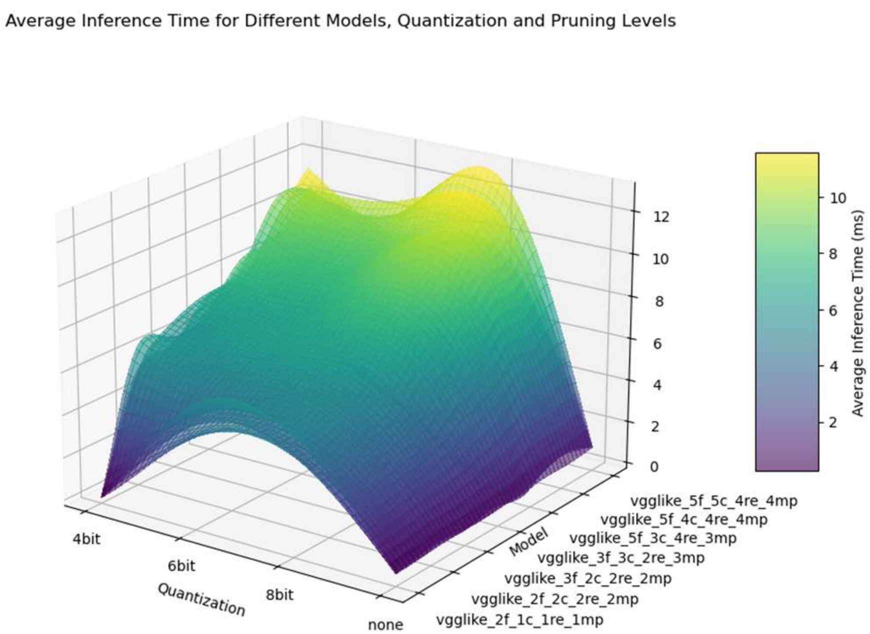
RESULTS – INFERENCE TIME (1/2)

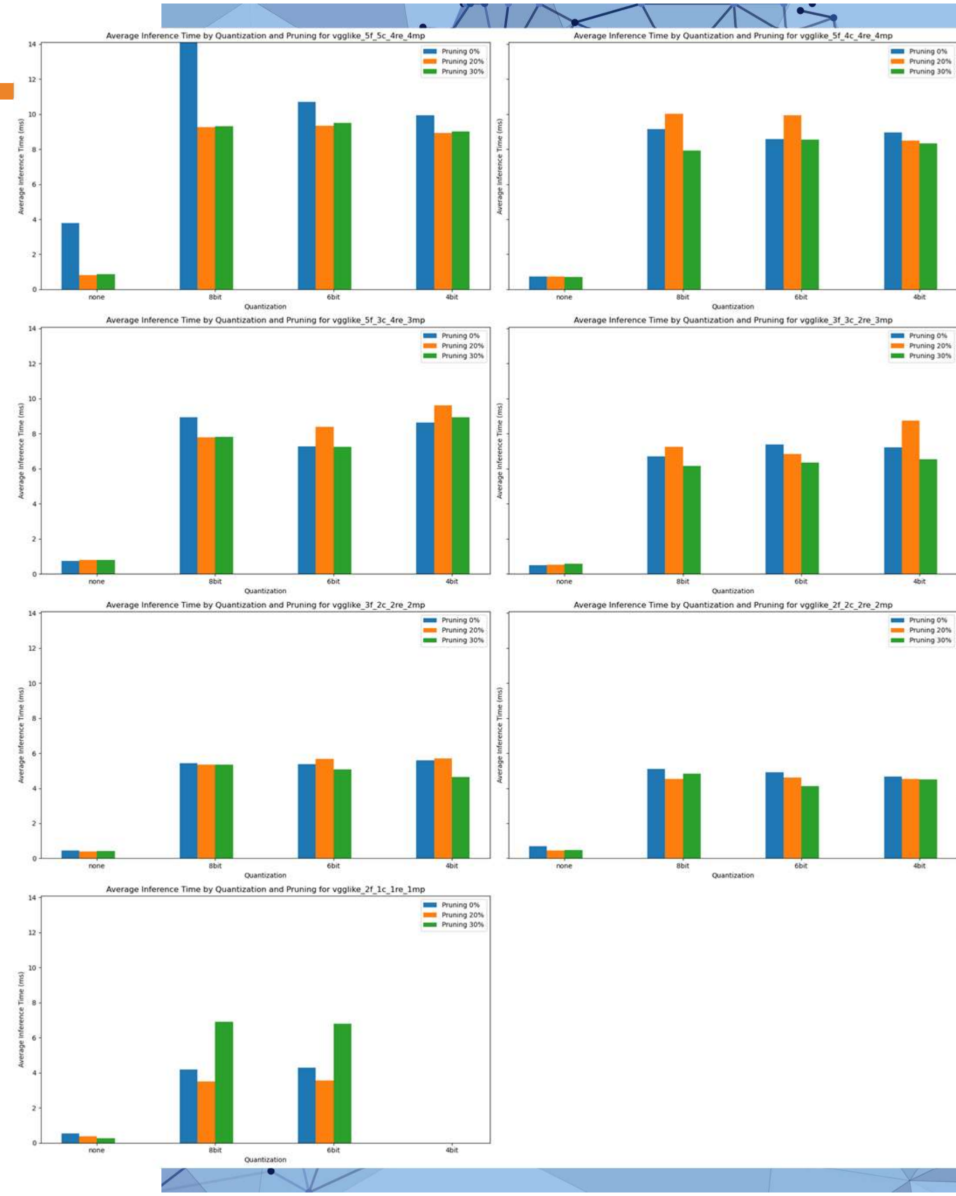
- Inference time (the time it takes a model to make predictions on a single batch of inputs) was better on smaller models, but raised by a factor of ten (e.g. 1ms to 10ms) when quantized by any amount.
- Pruning improved the models' performance slightly in all cases.





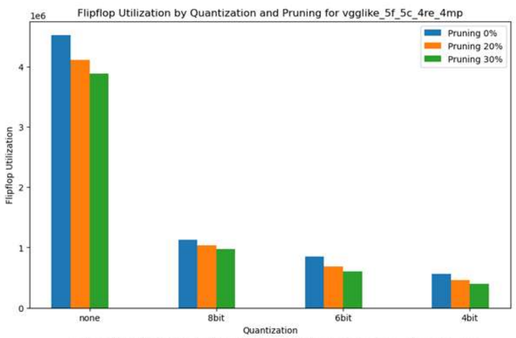
RESULTS – INFERENCE TIME (2/2)

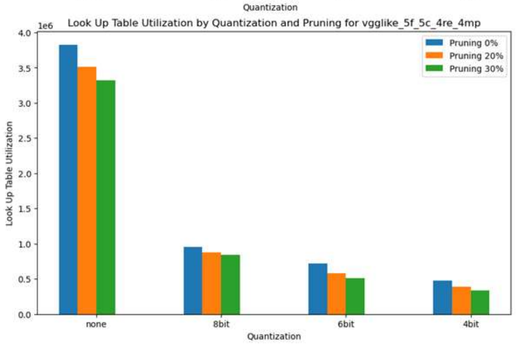


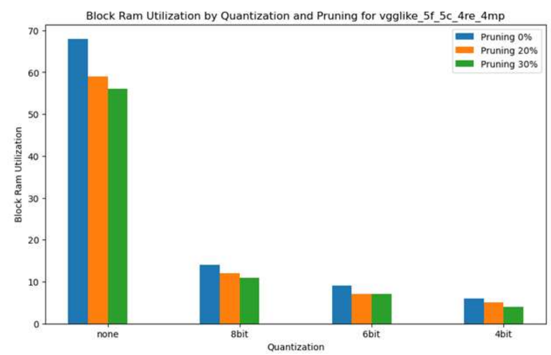


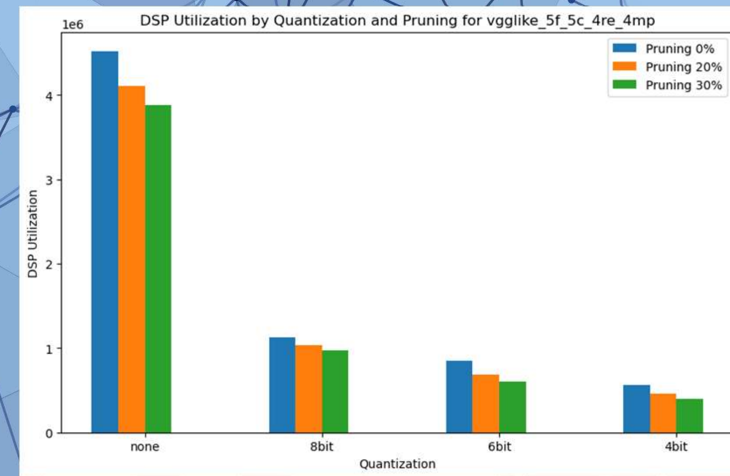
RESULTS – HARDWARE UTILIZATION (1/2)

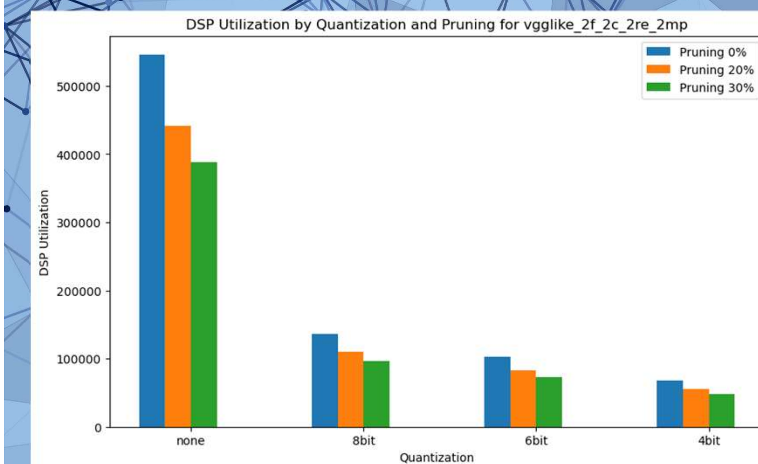
- Hardware utilization went down predictably with smaller models, with quantization providing a massive cut in utilization (factor of 4) – pruning providing a smaller but not-insignificant cut as well (20% pruning leading to roughly 10% less resource utilization).



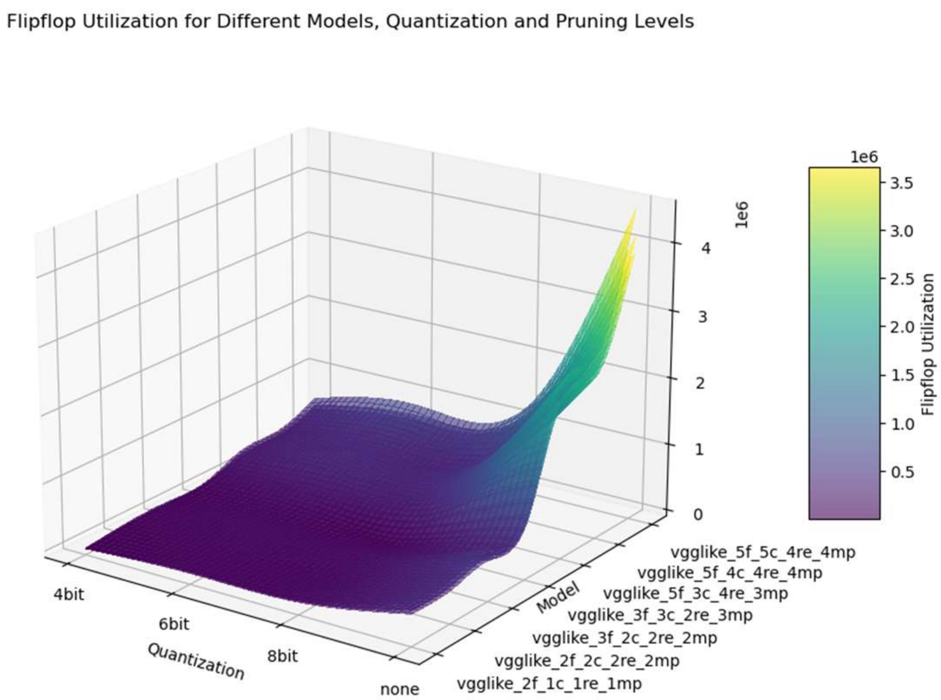


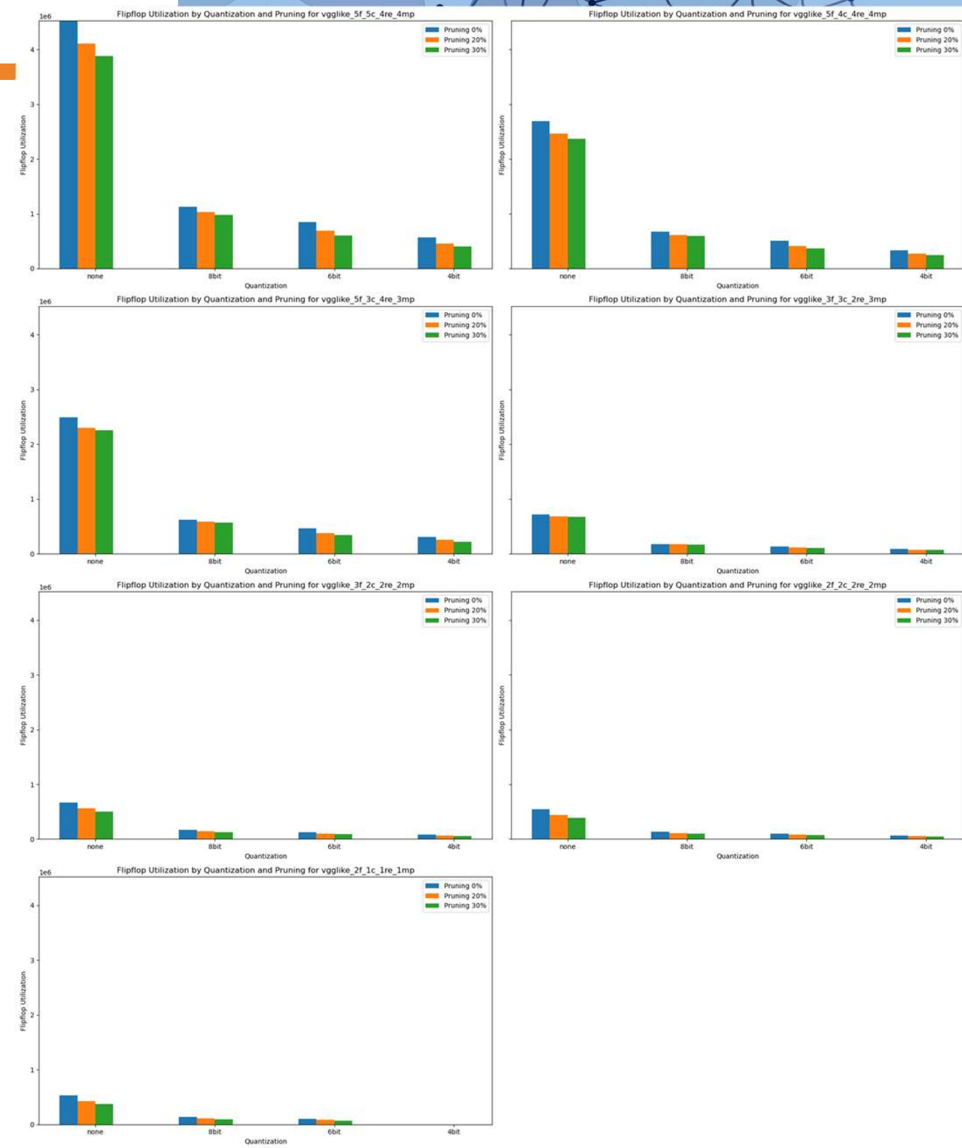






RESULTS – HARDWARE UTILIZATION (2/2)





RESULTS –TABULATED (1/3) – 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

RESULTS –TABULATED (2/3) – 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

RESULTS –TABULATED (3/3) – 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

CONCLUSION/FUTURE WORK

- Combining smaller architectures, quantization, pruning, and ReLU/Max Pooling layers can be done while maintaining high accuracy.
- System will want lowest possible inference time – so smallest architecture models provide best performance. Reducing number of layers is the best technique overall, with pruning providing a “free” performance bonus.
- Hardware estimation drastically reduced by smaller model sizes, with quantization providing a significant extra drop in utilization. Quantization needs to be evaluated on a performance basis to determine acceptability based on minimum required inference time – trading off speed for utilization. Pruning provides a small but linear drop in utilization, can be a “free” gain in valuable utilization space while maintaining accuracy.
- Would like to attempt to get models synthesized onto FINN architecture and actual hardware utilization measured, if possible.

PUBLICATIONS

- **In Preparation**

Thesis Paper (submitted)

- **Under Submission**

J. Rothe, H. Shajaiah, “Quantization and Pruning of Convolutional Neural Networks for Efficient FPGA Implementation of Digital Modulation Detection Firmware” – ICCCN2024, July 29-31, Hawaii, USA

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] F. N. Iandola, S. Han, H. M. Moskewicz, K. Ashraf, W. J. Dally and K. Kuetzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size," arXiv:1602.07360, 2016.
- [3] S. Han, H. Mao and W. Dally, "Deep Compression: Compressing DNNs with Pruning, Trained Quantization and Huffman Encoding," in ICLR, 2016.
- [4] S. Han, X. Liu, M. Huizi, J. Pu, A. Pedram, M. A. Horowitz and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," ACM SIGARCH Computer Architecture News, vol. 44, 2016.
- [5] P. M. Gysel, Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks, University of California Davis, 2016.
- [6] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao and J. Cong, Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks, Monterey, California: PKU/UCLA Joint Research Institute in Science and Engineering, 2015.
- [7] D. Góez, P. Soto, S. Latré, N. Gaviria and M. Camelo, A Methodology to Design Quantized Deep Neural Networks for Automatic Modulation Recognition, Basel, Switzerland: MDPI, 2022.