

Lab 6

Quantization

20201082 H. Kim, 20211184 H. Yang, 20221421 J. Hwang

I. Used quantization method

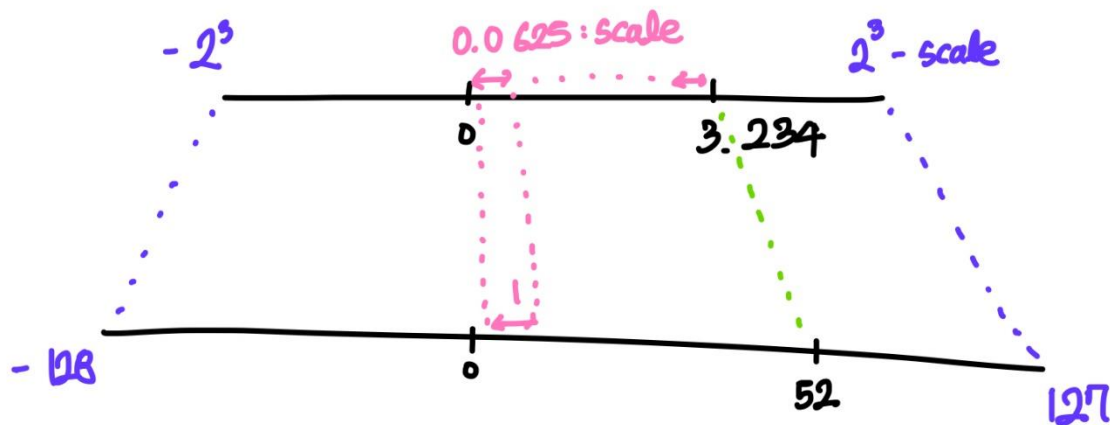
- 8bit fixed point quantization

```
60 ∨ def fixed_point_quantize(x, wl, fl, clamp=True, symmetric=True):
61     scale = 2**(-fl)
62     ∨ if symmetric:
63         min_val = -2**((wl - fl - 1))
64         max_val = 2**((wl - fl - 1)) - scale
65     ∨ else:
66         min_val = -2**((wl - fl - 1)) + scale
67         max_val = 2**((wl - fl - 1)) - scale
68
69     ∨ if clamp:
70         x = torch.clamp(x, min_val, max_val)
71
72     x_scaled = x / scale
73     x_rounded = torch.round(x_scaled).to(torch.int8)
74     return x_rounded
75
```

We used 8-bit fixed-point quantization. After quantizing a floating point number and storing it in int8, we can store real numbers in fixed-point format. wl (word length) represents the total number of bits, and fl (fraction length) represents the number of bits representing the fractional part.

```
quantized value: 7.9375
tensor(7.9375)
binary representation: 0111.1111
tensor(-8.)
quantized value: -8.0
binary representation: 11111000.0000
C:\Users\conqu\Desktop\TPU\Lab6_assign\Lab6_JH>
```

If $wl = 8$, $fl = 4$, float32 from -8.0 to 7.9375 can be represented as int8 from -128 to 127.



For example, the figure above is quantized as 3.324 as $wl = 8$, $fl = 4$. The basis for scale is the number of bits in the decimal part, i.e. fl . If fl is too small, a lot of the amount is discarded during dequantization or rounding operations, which reduces precision, and if fl is too large, there are not enough

bits to represent the integer part, which reduces the range that can be converted, so that when clamping is performed, the values at both ends become saturated, collapsing the data distribution. Therefore, fl should be as large as possible while having enough $(wl - fl)$: the number of bits in the integer part, to sufficiently express real numbers within the range of the data.

1. Weight quantization to fixed point 8bits (During training step)

```
def quantized_model_with_wl_87654321(model):
    best_accuracy = 0
    best_fl = 0
    for wl in range(1, 9): # 1 ~ 8
        for fl in range(1, wl+1):
            quantized_model = copy.deepcopy(model)
            for name, param in quantized_model.named_parameters():
                quantized_param = fixed_point_quantize(param.data, wl, fl=fl)
                param.data.copy_(quantized_param)

            accuracy = evaluate_model(quantized_model, test_loader)
            if accuracy > best_accuracy:
                best_accuracy = accuracy
                best_fl = fl

    print(f"Best fl: {best_fl}, for wl = {wl} with best accuracy: {best_accuracy*100:.2f}%")
```

For the previously learned float32 weight, the wl value was changed from 1 to 8, and the optimal fl value for each wl was obtained. (This can be obtained by comparing the accuracy of the quantized model.)

```
quantized_model = copy.deepcopy(model)
for name, param in quantized_model.named_parameters():
    quantized_param = fixed_point_quantize(param.data, wl, fl=best_fl)
    param.data.copy_(quantized_param)

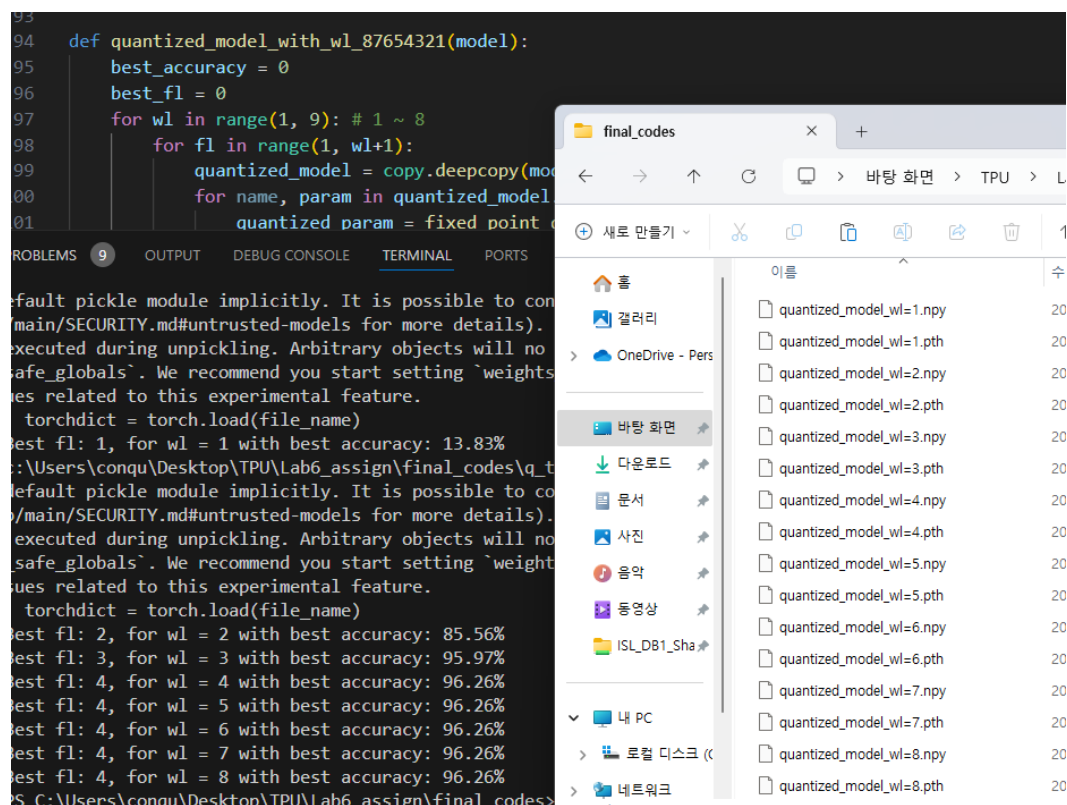
file_name = f'quantized_model_wl={wl}.pth'
torch.save(quantized_model.state_dict(), file_name)
## simple code to change pth to npy..
torchdict = torch.load(file_name)
numpydict = {}
numpydict['fc1w'] = np.array(torchdict['fc1.weight'])
# numpydict['fc1b'] = np.array(torchdict['fc1.bias'])

numpydict['fc2w'] = np.array(torchdict['fc2.weight'])
# numpydict['fc2b'] = np.array(torchdict['fc2.bias'])

numpydict['fc3w'] = np.array(torchdict['fc3.weight'])
# numpydict['fc3b'] = np.array(torchdict['fc3.bias'])

file_name_2 = f'quantized_model_wl={wl}.npy'
np.save(file_name_2, numpydict, allow_pickle=True)
```

The optimal fl values for each wl are learned and the quantized weight values for each fc layer are saved as a separate .npy file as below.





The image shows a code editor on the left and a file explorer on the right. The code editor displays a function `quantized_model_with_wl_87654321(model):` that iterates over `wl` values from 1 to 8 and `fl` values from 1 to `wl+1`. It quantizes the model parameters and saves them as `.pth` files. The terminal output shows the results of the quantization process, including the best `fl` value for each `wl` and the corresponding accuracy.

The file explorer shows a directory named `final_codes` containing a list of files. The files are organized by `wl` value and `fl` value, with the following structure:

이름	수
quantized_model_wl=1.npy	20
quantized_model_wl=1.pth	20
quantized_model_wl=2.npy	20
quantized_model_wl=2.pth	20
quantized_model_wl=3.npy	20
quantized_model_wl=3.pth	20
quantized_model_wl=4.npy	20
quantized_model_wl=4.pth	20
quantized_model_wl=5.npy	20
quantized_model_wl=5.pth	20
quantized_model_wl=6.npy	20
quantized_model_wl=6.pth	20
quantized_model_wl=7.npy	20
quantized_model_wl=7.pth	20
quantized_model_wl=8.npy	20
quantized_model_wl=8.pth	20

For comparison, we also extract the default weight values without performing quantization.

 float32_model.npy	2024-11-07 오후 4:44	NPY 파일	427KB
 float32_model.pth	2024-11-07 오후 4:44	PTH 파일	429KB

```
print("<weight quantized model, wl=1~8>")
▼ def feed_forward_hw(X0):
    X1 = np.matmul(X0, fc1w.T)
    A1 = np.tanh(X1)

    HW_result = hw_32x8.matmul(A1, fc2w.T)
    X2 = HW_result

    A2 = np.tanh(X2)

    X3 = np.matmul(A2, fc3w.T)
    return X3
```

Here, the converted 8-bit weight values are multiplied by the non-quantized float32 activation values in the inference step through HW in layer 2.

Therefore, a float32 * int8 matrix multiplier HW is required.

In fact, if wl becomes smaller than 8 (bits), the hardware needs to be reconfigured, but it takes a lot of time, so it was omitted in this assignment. Instead, even if wl < 8, it is stored as int8, so accuracy measurement is possible using the above HW.

Preliminary suggestion of how to implement 4-bit and 2-bit hardware for the next assignment: Since the smallest data type is int8, int8 is divided into two or four pieces using bit operations, and then quantized data (4 or 2 bits) is stored.

```

def feed_forward_hw_32x32(X0):
    X1 = np.matmul(X0, fc1w.T)
    A1 = np.tanh(X1)

    HW_result = hw_32x32.matmul(A1, fc2w.T)
    X2 = HW_result

    A2 = np.tanh(X2)

    X3 = np.matmul(A2, fc3w.T)
    return X3

```

For comparison, we also produce float32 * float32 HW (without any quantization).

2. Activation value quantization to int8 fixed point (During inference step)

```

def feed_forward_hw_8x8(X0):
    X1 = np.matmul(X0, fc1w.T)
    A1 = np.tanh(X1)

    A1 = fixed_point_quantize(A1, wl=8, fl=4) # arbitrary fl
    #print(A1)
    HW_result = hw_8x8.matmul(A1, fc2w.T) # int8 * int8
    X2 = HW_result

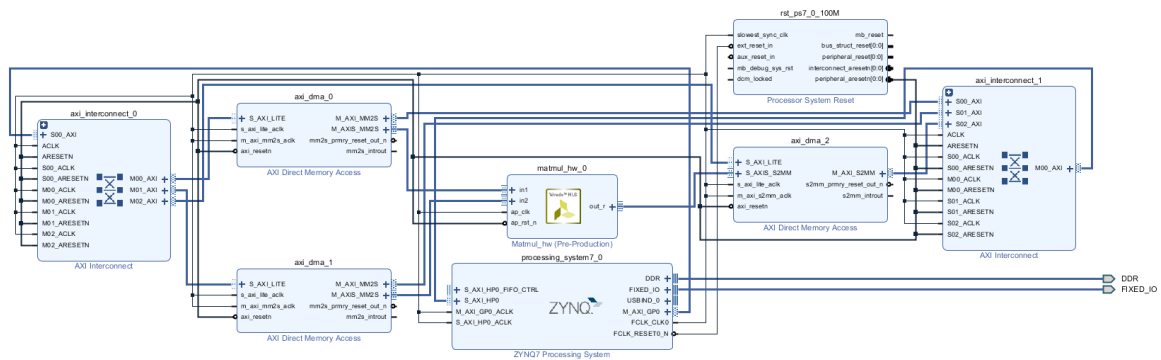
    A2 = np.tanh(X2)

    X3 = np.matmul(A2, fc3w.T)
    return X3

```

Additionally, in the Inference step, we tried to make int8 * int8 matrix multiplication possible by setting the activation value from layer1 to wl=8, fl=4 (set arbitrarily).

II. Hardware synthesis and performance comparison



The top diagrams are all the same.

1. float32 * float32 Matrix multiplier: for non-quantized activation and weights

C-synthesis result

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	15.210 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		Type
min	max	min	max	min	max	
2134102	2134102	32.460 ms	32.460 ms	2121808	2121808	dataflow

Detail

+ Instance

+ Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2	-
FIFO	-	-	-	-	-
Instance	40	5	897	2188	0
Memory	40	-	0	0	0
Multiplexer	-	-	-	-	-
Register	-	-	-	-	-
Total	80	5	897	2190	0
Available	280	220	106400	53200	0
Utilization (%)	28	2	~0	4	0

Detail

The float32 * float32 matrix multiplier uses the most memory resources (BRAM) and compute elements (FF, LUT) among the three HWs.

2. float32 * int8 Matrix multiplier: for non-quantized activation and quantized weights

C-synthesis result

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	15.210 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		Type
min	max	min	max	min	max	
2134108	2134108	32.460 ms	32.460 ms	2121814	2121814	dataflow

Detail

+ Instance

+ Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2	-
FIFO	-	-	-	-	-
Instance	28	5	1503	2838	0
Memory	28	-	0	0	0
Multiplexer	-	-	-	-	-
Register	-	-	-	-	-
Total	56	5	1503	2840	0
Available	280	220	106400	53200	0
Utilization (%)	20	2	1	5	0

Compared to the previous float32 * float32 HW, we can see that the memory usage has decreased. (BRAM = 80 -> 56) This is because one input is int8. On the other hand, the number of operation elements has increased (FF, LUT = 900, 2200 -> 1500, 2800). First of all, the multiplication of float32 and int8 is the same as the multiplication between float32. Therefore, the amount of operation does not decrease. In addition, it is thought that the increase in operation elements is due to the creation of an additional operation element that converts int8 to float32 before the multiplication operation. Since it is the same as the float32 operation, the operation speed is the same as the float32 * float32 HW. (latency = 32ms)

3. int8 * int8 Matrix multiplier: for (both) quantized activation and quantized weights

The code below shows the implementation of the clamp operation inside HW.

```
35 // Matrix multiplication
36 for (int i = 0; i < ROW_A; i++) {
37     for (int j = 0; j < COL_B; j++) {
38         __int16 sum = 0;
39         for (int k = 0; k < COL_A; k++) {
40             #pragma HLS PIPELINE II=1
41             sum += A[i][k] * B[k][j];
42         }
43         if(sum > 127)
44             sum = 127;
45         else if (sum < -128)
46             sum = -128;
47         // above logics are clamping 16bit result to 8 bit.
48         C[i][j] = sum;
49     }
50 }
```

When multiplying int8 and int8, the result can be up to 16 bits. If not set separately, the lower 8 bits of the 16-bit output are sliced and stored in the 8-bit output. For example, the outputs of the multiplication values 00001111 '00110011' and 00000000 '00110011' are both '00110011'. This causes the distribution of the multiplication values to collapse, which adversely affects the next layer and significantly reduces the accuracy. Therefore, instead of discarding the upper bits of the result value to fit 8 bits, clamping was used (making it saturated).

C-synthesis result

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.097 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		Type
min	max	min	max	min	max	
561234	561234	5.612 ms	5.612 ms	548940	548940	dataflow

Detail

+ Instance

+ Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2	-
FIFO	-	-	-	-	-
Instance	10	1	643	1586	0
Memory	10	-	0	0	0
Multiplexer	-	-	-	-	-
Register	-	-	-	-	-
Total	20	1	643	1588	0
Available	280	220	106400	53200	0
Utilization (%)	7	~0	~0	2	0

Certainly, $\text{int8} * \text{int8}$ HW uses a small data type, so it uses the least memory, and since integer operations are simple, the number of operation elements is the smallest. Since it is an integer operation, the operation speed is also the fastest. (Latency 32ms \rightarrow 5.6ms) It is about 4 times faster than Float32 operations based on clock cycles.

III. Comparison of inference performance by quantization method

Below are the results of running HW matrix multiplication in Jupyter Notebook.

1. No quantized original model (default)

(use float32 * float32 HW)

```
<no_quantized_model>  
Execution time: 142.97 seconds  
Prediction accuracy is 97.58691674290942
```

2. Weight quantized model, w1 = 1~8 (use float32 * int8 HW)

<pre><weight quantized model, w1=1~8> w1=1 Execution time: 122.88 seconds prediction accuracy is 13.14</pre>	<pre>w1=5 Execution time: 120.73 seconds prediction accuracy is 97.11</pre>
<pre>w1=2 Execution time: 124.55 seconds prediction accuracy is 85.75</pre>	<pre>w1=6 Execution time: 122.41 seconds prediction accuracy is 97.11</pre>
<pre>w1=3 Execution time: 122.17 seconds prediction accuracy is 96.61</pre>	<pre>w1=7 Execution time: 120.79 seconds prediction accuracy is 97.11</pre>
<pre>w1=4 Execution time: 122.34 seconds prediction accuracy is 97.11</pre>	<pre>w1=8 Execution time: 122.32 seconds prediction accuracy is 97.11</pre>

The speed is faster than when using non-quantized activations and weights, and the accuracy is 97.11 (weight quantized) < 97.58 (no quantized) for w1 = 8, and the accuracy starts to decrease from w1 = 3. As briefly introduced earlier, the optimal fl is all 4 when w1 = 4~8 in the training stage. That is thought to be the reason why the accuracy is the same in this region.

3. 8bits activation & weight both quantized model

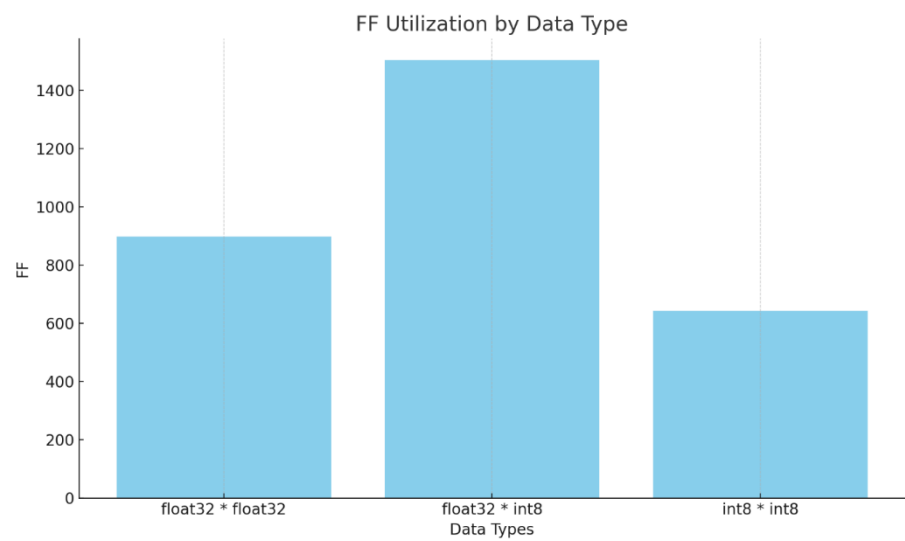
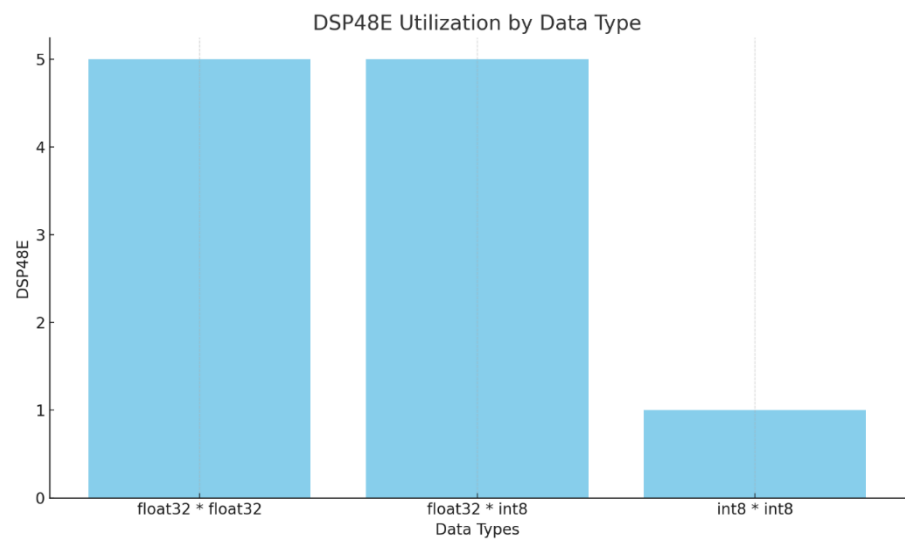
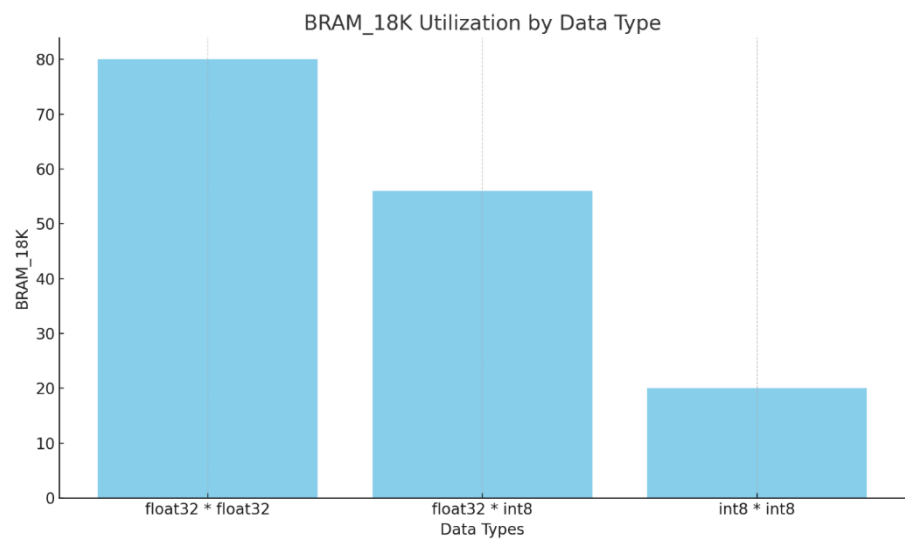
(use int8 * int8 HW)

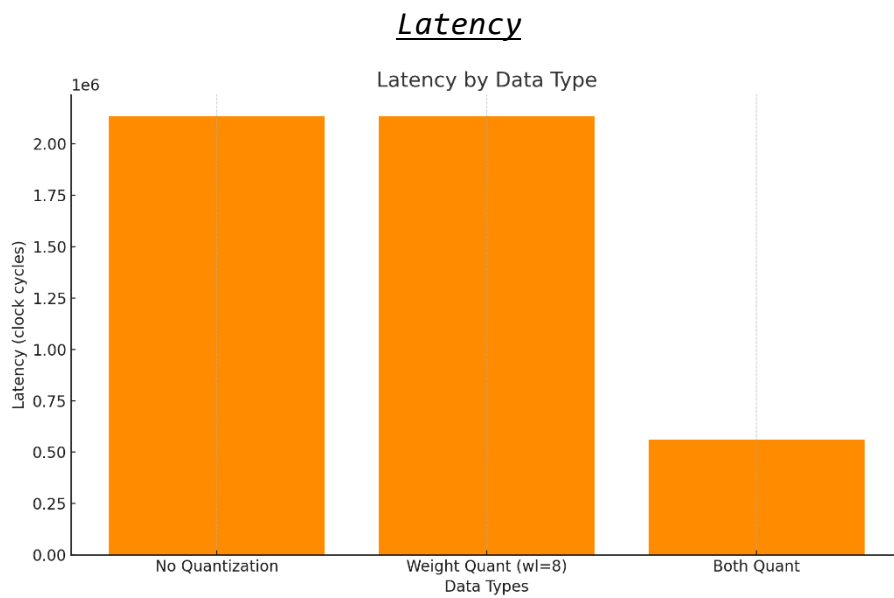
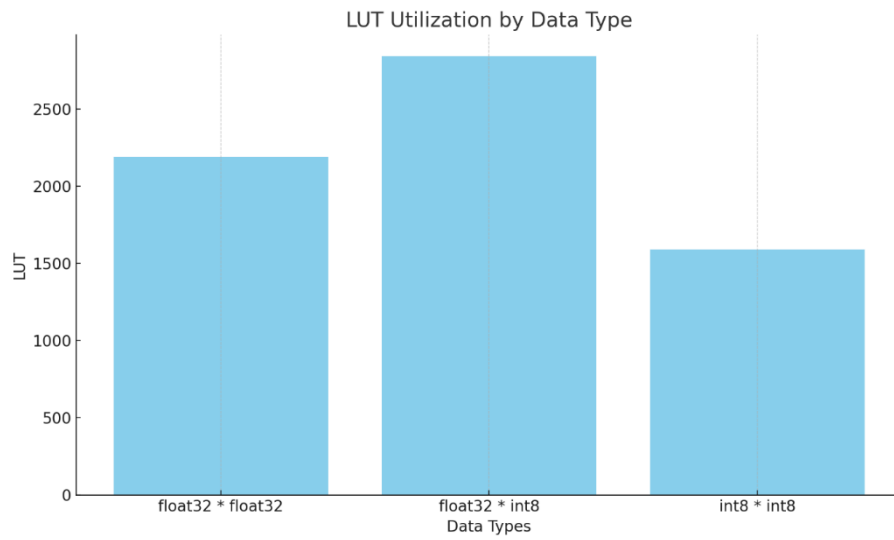
```
<both quantized model : 8bits active & weighth>  
Execution time: 109.99 seconds  
activation quantized prediction accuracy is 97.02
```

As a result of quantizing both activation and weight to int8, the speed is the fastest because int8 * int8 HW is used, and the accuracy is 97.02 (both activation & weight quantized) < 97.11 (weight quantized) < 97.58 (no quantized) based on w1 = 8.

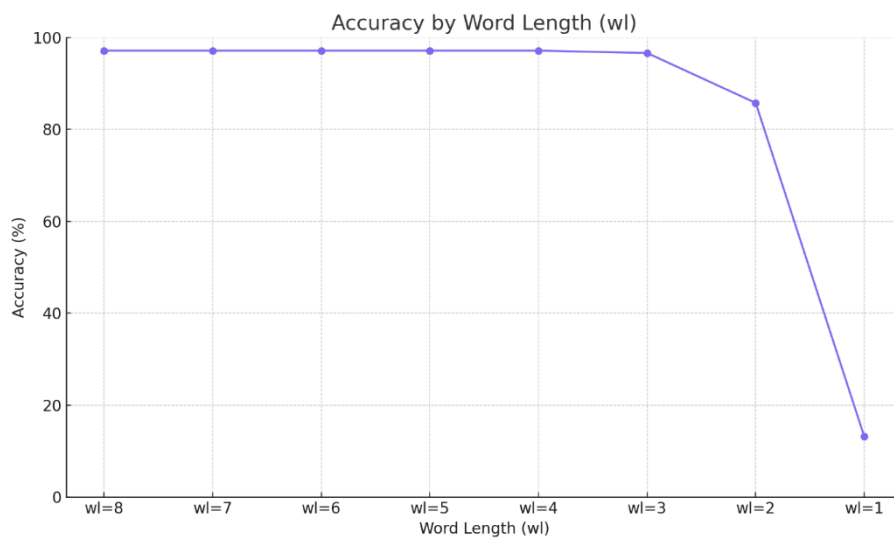
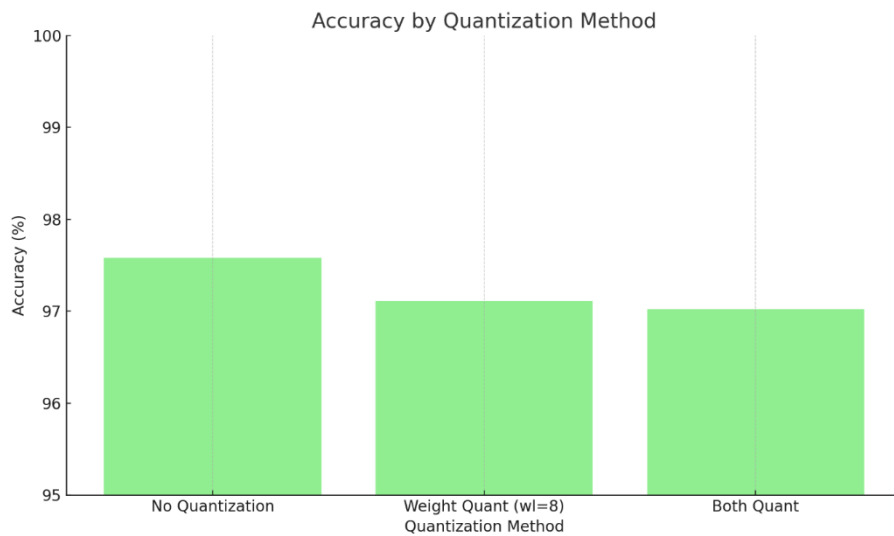
IV. Appendix

Used HW resource





Accuracy



Inference time

