User Guide of Preparing Fixed-point ONNX Model for ACCESS DLA Compiler

Ver.1.2.0

18. July. 2023

Contents

I. Overview 4

A. Hardware-aware Neural Network Optimization 4

A. I/O Data 6

B. Operators 6

1. Unlimited number of operators 6

2. Standard Convolution 6

3. Depth-wise Convolution 6

4. Activation Function 6

5. Pooling 6

6. Fully Connected 6

7. Elementwise Add/division 6

8. Feature Map Concatenation 6

9. Batch Normalization 7

10. Matrix multiplication 7

11. Transpose/Reshape 7

C. Quantization Scale Parameters 7

III. Fixed-point ONNX Model 9

A. PyTorch to ONNX 9

B. Quantized Operators 10

1. Convolution and Fully Connected 10

2. Pooling 11

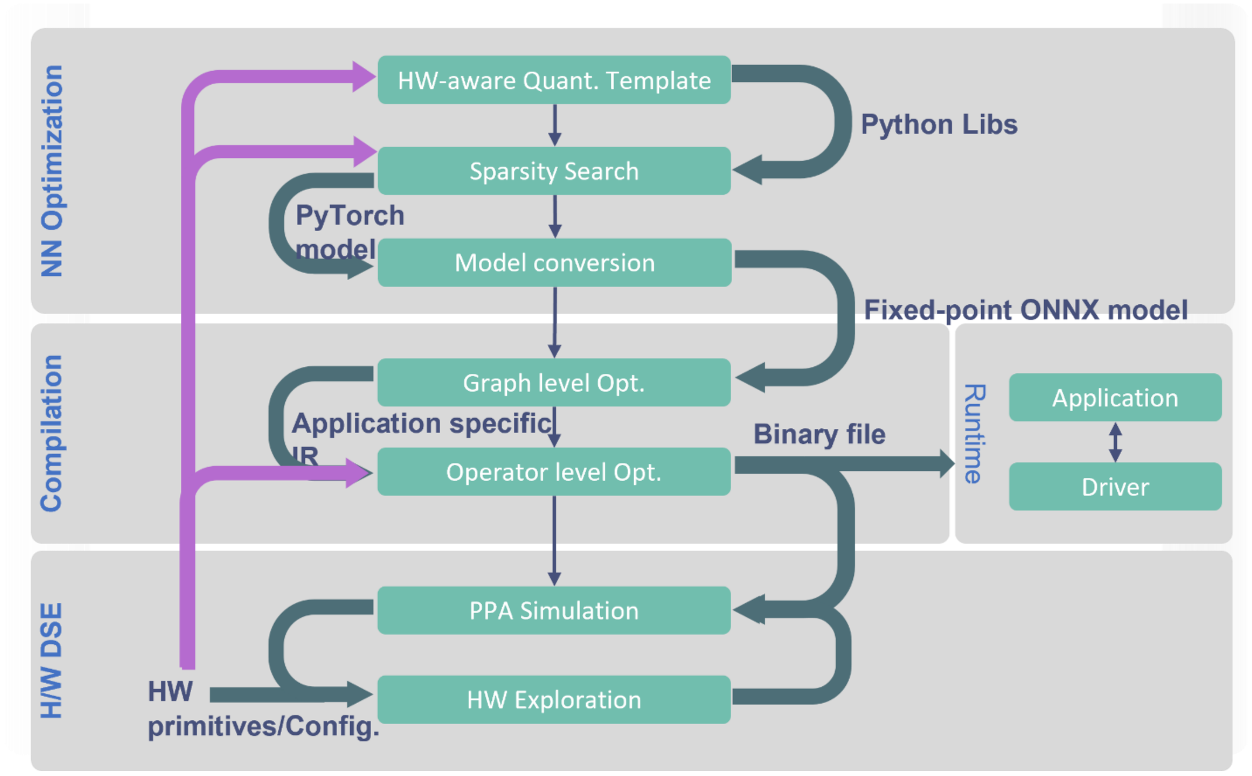
3. MatMul 11

4. Element wise Add/Div 11

IV. Brief Guidance of Quantization-aware Training 12

I. Overview

This guide offers comprehensive instructions for preparing the fixed-point ONNX model, meticulously tailored for our specialized Deep Neural Network (DNN) compiler. To ensure a seamless compilation process, it is crucial to thoroughly verify that the ONNX model meets the specific requirements detailed in this document. Additionally, this guide encompasses a proven and successful method of quantization-aware training.



A. Hardware-aware Neural Network Optimization

Hardware-aware neural network optimization is a specialized approach that tailors the design and training of neural network models to match the capabilities and constraints of specific hardware architectures. The primary goal of hardware-aware optimization is to maximize the performance, energy efficiency, and resource utilization of neural networks when deployed on dedicated AI accelerators.

This optimization process involves several key aspects:

Architecture-specific Model Design: Hardware-aware optimization begins with considering the unique characteristics of the target hardware platform. Neural network models are designed or modified to exploit the hardware's strengths and minimize its limitations. In this document, several high-performance operators are suggested for architecture design.

Quantization and Precision Tuning: Neural network models are often trained and quantized to lower precision (e.g., 8-bit or even lower) to ensure compatibility with hardware's reduced precision capabilities. Precision tuning aims to balance the trade-off between model accuracy and quantization-induced loss of precision, optimizing the model's performance on the hardware.

Memory and Computation Optimization: Hardware-aware optimization takes into account the memory and computational constraints of the target platform. Techniques like model pruning, weight sharing, and layer fusion are employed to reduce the model's memory footprint and computational complexity without sacrificing performance.

Profiling and Fine-tuning: Throughout the process, hardware-aware optimization relies on iterative profiling to measure the model's performance on the target hardware or time accurate simulators. Fine-tuning is then performed to make further adjustments based on the profiling results.

By carefully considering the specific hardware architecture, hardware-aware neural network optimization ensures that the resulting models can take full advantage of the platform's capabilities, delivering superior performance and energy efficiency for AI applications deployed on ACCESS codesign DLA.

II. Model Definition

The deep learning model is defined in a high-level framework such as PyTorch or DarkNet. The ACCESS Co-design CNN Accelerator is designed to support a diverse and extensive range of CNN operators. For optimal execution and performance, the target model must make efficient use of these supported operators.

A. I/O Data

	Input	Feature Map	Weights	Bias
Channel No.	1 to 512	16 to 2048 (times of 16)	Depends on H/W buffer size	
Resolution	1x1 to 512x512	1x1 to 256x256		
Bit Depth	8	8	8	16

B. Operators

1. Unlimited number of operators
2. Standard Convolution
   a) filter size: 1x1 to 7x7, support H != W
   b) feature map size: 1x1 to 512x512
   c) channel number: up to 2048
   d) stride: 1 and 2
   e) padding: 0 to 3 with value=0
   f) bit-width: 8bit
3. Depth-wise Convolution
   a) filter size: 1x1 to 7x7, support H != W
   b) feature map size: 4x4 to 512x512
   c) channel number: up to 2048
   d) stride: 1 and 2
   e) padding: 0 to 3 with value=0
   f) bit-width: 8bit
4. Activation Function
   a) Pipelined function:
      ReLU/PReLU/LeakyReLU
   b) Stepwise linear approximation:
      Tanh/Sigmoid/Hardswish
   c) Disable
5. Pooling
   a) max-pooling and average pooling
   b) filter size:
      max pooling: 2x2 and 3x3,
      avg pooling: 2x2 to 7x7
   c) stride: 1 and 2
   d) padding: 0 to 3
      only bottom and right directions in max pooling,
      top and left in average pooling.
6. Fully Connected
   a) channel number: up to 2048
7. Elementwise Add/division
   a) support two sources
8. Feature Map Concatenation
   a) multiple sources

9. Batch Normalization
   a) Follow after the convolution or matrix multiplication op.

10. Matrix multiplication / Reducesum
11. Transpose/Reshape

C. Quantization Scale Parameters

In activation parameter quantization, the scale refers to the ratio between a floating-point value and its corresponding quantized integer value. This scale factor determines the relationship between the original floating-point range and the quantized integer range. It is used to map the continuous range of floating-point values to a discrete set of quantized integer values.

For example, in fully connected operator:

$$C = wA + B = \frac{S_w S_A(w_{int}A_{int}+B_{int})}{S_C},$$

In which, C, w, A, B denote the floating-point value, $w_{int}, A_{int}, B_{int}$ denote the quantized integer value, and S_C, S_w, S_A, S_B refer to the scale factor of each item.

The scale factor for the weight's tensor can differ in each output channel. Each scale factor is represented as a reciprocal dyadic format ($2^n/a\,, a = 1,2, \ldots ,255, n \in Z$).

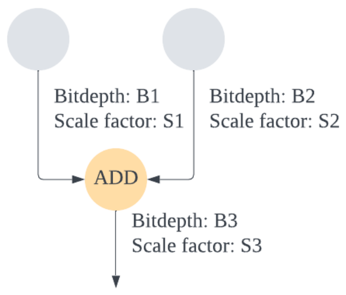
The scale factor for the feature maps is identical across the entire tensor and follows a power of 2 format ($2^n, n \in Z$).

Ensuring coherence in the scale factors across the entire neural network computation graph is crucial for successful quantization and maintaining consistency throughout the network.

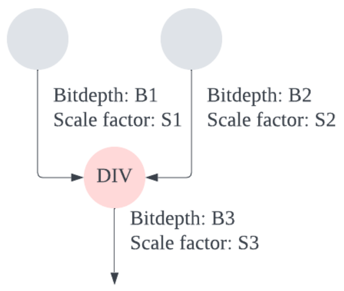
For convolution, Matrix multiplication, average pooling (global reduction), saturated activation function (e.g. tanh, sigmoid) and elementwise add/div, the quantized inputs and output feature maps can have different scale factors. For other operators, such as maximum pooling operator, or any other activation functions, should have its input and output feature maps quantized with the same scale factors. Additionally, any feature map used in different operators within the network should have identical scale factors to maintain consistency.

This coherence in scale factors ensures that the quantized values are consistent throughout the network, preserving the relationships between different layers and avoiding any mismatch or distortion in the quantized representations. It enables seamless propagation of quantized values throughout the network and allows for efficient computation while maintaining the accuracy of the model.

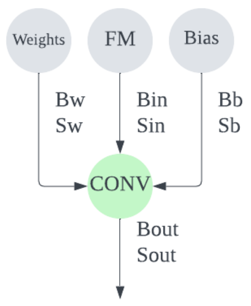
Here are some rules of scale factor propagation.



$B1 = B2 = B3$
$S3 \leq \min(S1, S2)$

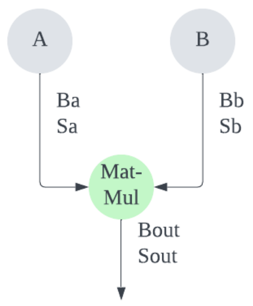


$B1 = B2 = B3$
$S1 \geq S2$
$S3 \leq (S1 - S2)$

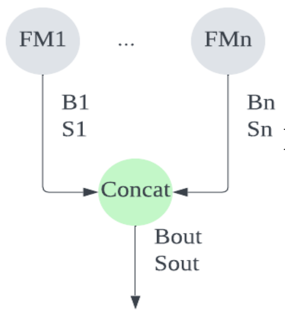


$Bw = Bin = Bout$
$Bb = 16$
$Sb = SwSin$
$Sout \leq SwSin$

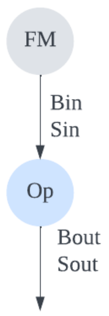
PS. Sb and Sw differs in each output channel



$Ba = Bb = Bout$
$Sout \leq SaSb$



$B1 = \ldots = Bn = Bout$
$S1 = \ldots = Sn = Sout$



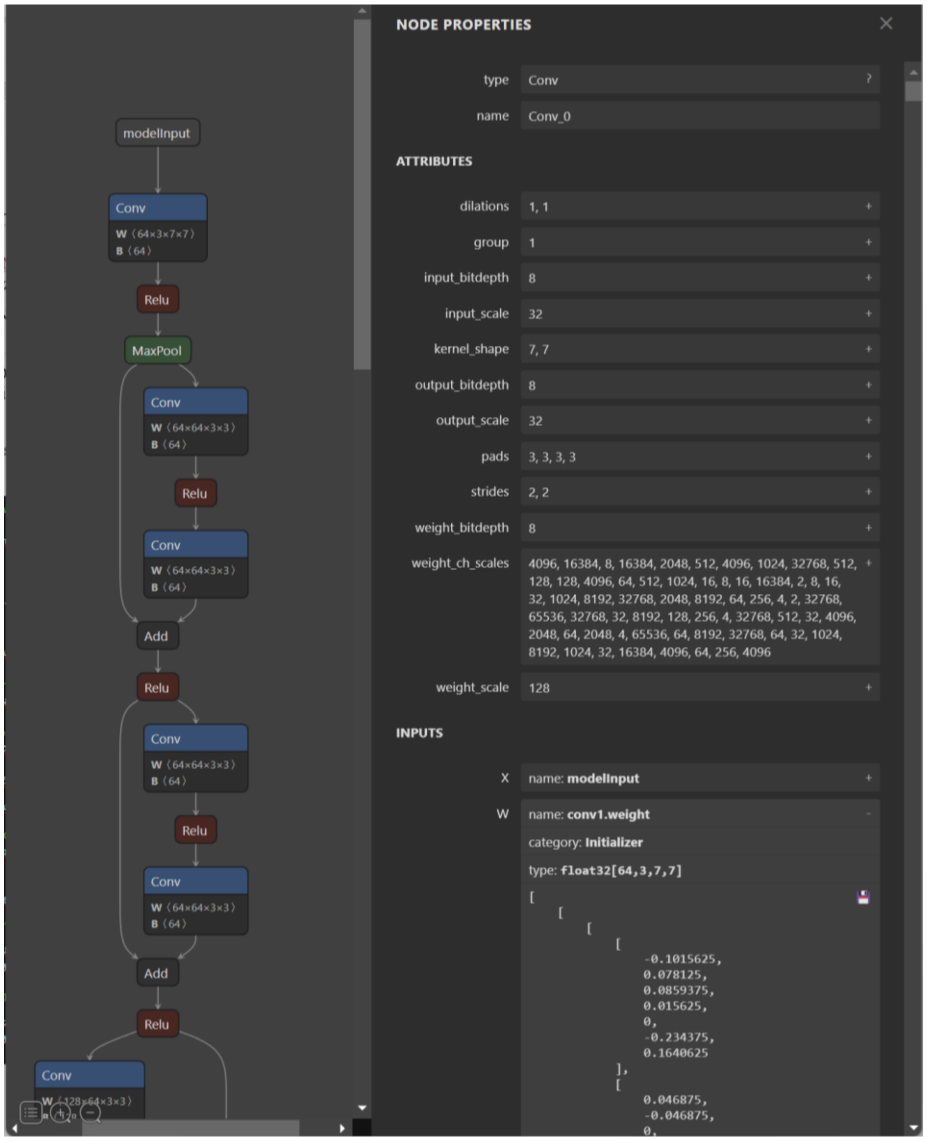
$Bin = Bout$
When Op in [ReLU/PreLU, Reshape, Max Pooling, Transpose]:
$Sin = Sout$
When Op in [Average Pooling, Saturated Act. Function]:
$Sout \leq Sin$

III. Fixed-point ONNX Model

Leverage the Open Neural Network Exchange (ONNX) format, one can define the quantized model, embed quantization parameters and wrap as the input of ACCESS compiler.

A. PyTorch to ONNX

When using PyTorch for model training, one can convert the trained model into the ONNX format using the PyTorch ONNX converter. Subsequently, the ONNX Python frontend's edition tools provide functions to insert the necessary quantization parameters into each operator of the ONNX model.



NODE PROPERTIES

type	Conv
name	Conv_0

ATTRIBUTES

dilations	1, 1
group	1
input_bitdepth	8
input_scale	32
kernel_shape	7, 7
output_bitdepth	8
output_scale	32
pads	3, 3, 3, 3
strides	2, 2
weight_bitdepth	8
weight_ch_scales	4096, 16384, 8, 16384, 2048, 512, 4096, 1024, 32768, 512, 128, 128, 4096, 64, 512, 1024, 16, 8, 16, 16384, 2, 8, 16, 32, 1024, 8192, 32768, 2048, 8192, 64, 256, 4, 2, 32768, 65536, 32768, 32, 8192, 128, 256, 4, 32768, 512, 32, 4096, 2048, 64, 2048, 4, 65536, 64, 8192, 32768, 64, 32, 1024, 8192, 1024, 32, 16384, 4096, 64, 256, 4096
weight_scale	128

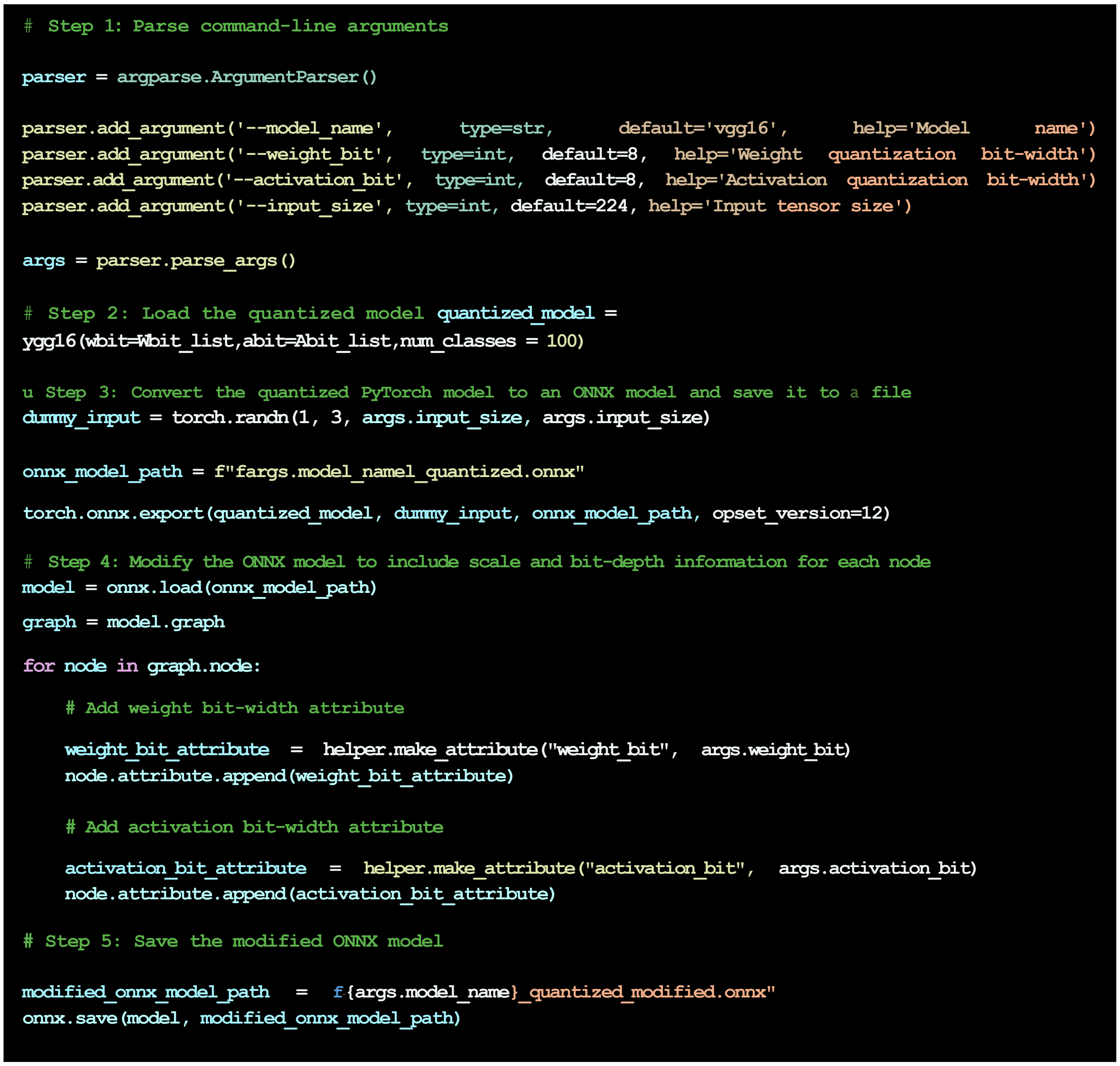
INPUTS

X	name: **modelInput**
W	name: **conv1.weight**
	category: **Initializer**
	type: **float32[64,3,7,7]**

```
[
    [
        [
            [
                -0.1015625,
                0.078125,
                0.0859375,
                0.015625,
                0,
                -0.234375,
                0.1640625
            ],
            [
                0.046875,
                -0.046875,
                0,
```

The model parameters, such as weights and bias, are stored in floating point format, wich is $S_w W_{int}$, $S_b B_{int}$.

Here is a python demostration of converting pytorch model into fixed-point ONNX model.



```
# Step 1: Parse command-line arguments

parser = argparse.ArgumentParser()

parser.add_argument('--model_name',      type=str,      default='vgg16',      help='Model      name')
parser.add_argument('--weight_bit',   type=int,   default=8,   help='Weight  quantization  bit-width')
parser.add_argument('--activation_bit',  type=int,  default=8,  help='Activation  quantization  bit-width')
parser.add_argument('--input_size', type=int, default=224, help='Input tensor size')

args = parser.parse_args()

# Step 2: Load the quantized model quantized_model =
ygg16(wbit=Wbit_list,abit=Abit_list,num_classes = 100)

u Step 3: Convert the quantized PyTorch model to an ONNX model and save it to a file
dummy_input = torch.randn(1, 3, args.input_size, args.input_size)

onnx_model_path = f"fargs.model_namel_quantized.onnx"

torch.onnx.export(quantized_model, dummy_input, onnx_model_path, opset_version=12)

# Step 4: Modify the ONNX model to include scale and bit-depth information for each node
model = onnx.load(onnx_model_path)

graph = model.graph

for node in graph.node:

    # Add weight bit-width attribute

    weight_bit_attribute  =  helper.make_attribute("weight_bit",  args.weight_bit)
    node.attribute.append(weight_bit_attribute)

    # Add activation bit-width attribute

    activation_bit_attribute  =  helper.make_attribute("activation_bit",  args.activation_bit)
    node.attribute.append(activation_bit_attribute)

# Step 5: Save the modified ONNX model

modified_onnx_model_path  =  f{args.model_name}_quantized_modified.onnx"
onnx.save(model, modified_onnx_model_path)
```

B. Quantized Operators

In a fixed-point ONNX model, various operators are quantized and enhanced with specific quantization parameters, such as bit depth, scale factor. The quantization parameters differ depending on the type of operator used. Below are the different fields of quantization parameters for various types of operators:

1. Convolution and Fully Connected

input_scale: scale factor of input tensor. Floating number in power of 2.

input_bitdepth: bit depth of input tensor. Integer in [8], weight_scale: deprecated, scale factor of weight tensor. Floating number in reciprocal dyadic format.

weight_bitdepth: bit depth of weight tensor. Integer in [8],

weight_ch_scales: channel wise scale factors of weight tensor. A list of floating number in reciprocal dyadic format. The length of the list is equal to the number of output channels in weight tensor.

output_scale: scale factor of output tensor. Floating number in power of 2.

output_bitdepth: bit depth of output tensor. Integer in [8],

bias_scale: deprecated, scale factor of input tensor. Floating number in reciprocal dyadic format.

bias_bitdepth: bit depth of input tensor. Integer in [16].

2. Pooling

input_scale: scale factor of input tensor. Floating number in power of 2.

input_bitdepth: bit depth of input tensor. Integer in [8],

output_scale: scale factor of output tensor. Floating number in power of 2.

output_bitdepth: bit depth of output tensor. Integer in [8].

3. MatMul

A_scale: scale factor of input A tensor. Floating number in power of 2.

A_bitdepth: bit depth of input A tensor. Integer in [8],

B_scale: scale factor of input B tensor. Floating number in power of 2.

B_bitdepth: bit depth of input B tensor. Integer in [8],

output_scale: scale factor of output tensor. Floating number in power of 2.

output_bitdepth: bit depth of output tensor. Integer in [8].

4. Element wise Add/Div and Reducesum

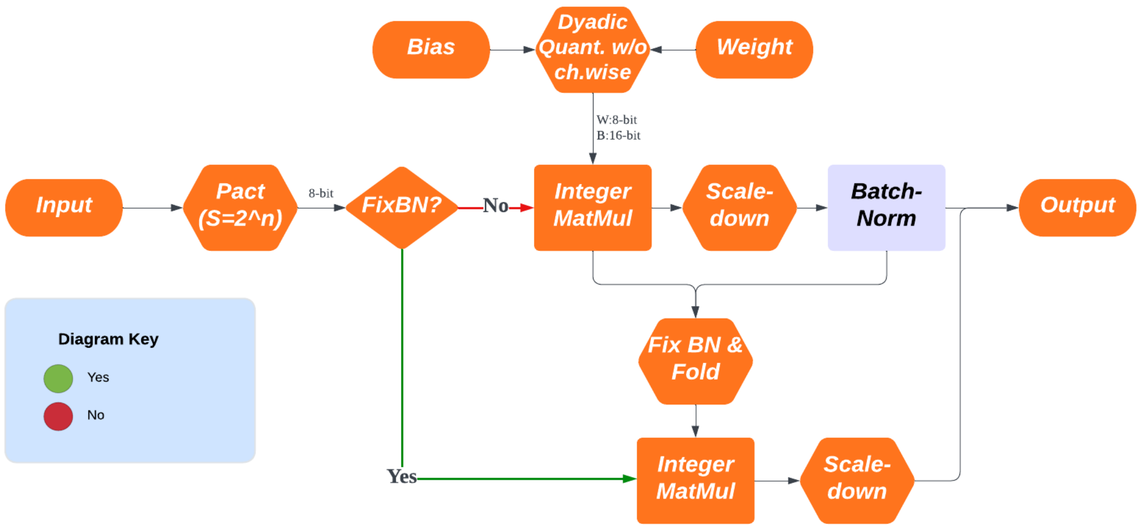
output_scale: scale factor of output tensor. Floating number in power of 2.

output_bitdepth: bit depth of output tensor. Integer in [8].

IV. Brief Guidance of Quantization-aware Training

Quantization-aware training is a technique used in training neural network models with the aim of improving their performance and efficiency in low-precision settings, such as fixed-point or integer arithmetic. The primary goal of quantization-aware training is to enable neural networks to function effectively with reduced precision data types (e.g., 8-bit integers) instead of the standard higher-precision floating-point representations.

The process involves simulating the effects of quantization during the training phase itself. Instead of training the model using standard floating-point precision, quantization-aware training introduces quantization operations that mimic the behavior of lower-precision data types. By doing so, the model becomes more robust to the loss of precision caused by quantization during inference.



During quantization-aware training, the following key steps are typically undertaken:

Quantization-aware Model Architecture: The neural network architecture is adapted to handle quantized inputs and parameters, incorporating specific quantization operations at strategic points in the model.

Quantization Simulation: During training, the forward and backward passes consider the effects of quantization on weights, activations, and gradients. This ensures that the model learns to optimize for lower precision during training.

Scale Factor Calibration: Quantization-aware training involves calibrating scale factors that determine the mapping between floating-point values and quantized integer values. These scale factors are essential for achieving an accurate and efficient conversion.

Fine-tuning: After quantization-aware training, the model is often fine-tuned using standard floating-point training to refine the weights and further enhance performance.

Quantization-aware training helps alleviate the challenges of deploying neural network models on hardware with limited precision support, such as edge devices and specialized accelerators. By training the model with an understanding of quantization effects, it becomes better suited to perform well in low-precision settings, achieving faster execution and reduced memory requirements while maintaining accuracy.