Pre-Quantized Deep Learning Models Codified in ONNX to Enable Hardware/Software Co-Design

Ulf Hanebutte¹, Andrew Baldwin², Senad Durakovic¹, Igor Filipovich³, Chien-Chun (Joe) Chou¹, Damian Adamowicz⁴, Derek Chickles¹, and David Hawkes⁵

¹Marvell Technology Inc.
²Nokia Solutions and Networks Oy
³Nokia of America Corporation
⁴Nokia Solutions and Networks Sp. z o.o.
⁵Nokia Paris Saclay

Abstract

This paper presents a methodology to separate the quantization process from the hardware-specific model compilation stage via a pre-quantized deep learning model description in standard ONNX format. Separating the quantization process from the model compilation stage enables independent development. The methodology is expressive to convey hardware-specific operations and to embed key quantization parameters into a ONNX model which enables hardware/software co-design. Detailed examples are given for both MLP and CNN based networks, which can be extended to other networks in a straightforward fashion.

1 Introduction

In recent years, deep learning models have shown to provide superior predictive capabilities in many domains. However, these models, as they are typically developed in full \( fp32 \) floating-point precision, are both extreme in compute and memory intensive. The mathematical concept of quantization which generates models in lower precision, e.g. \( int8 \), has gained importance as a way to alleviate the computational requirements. In particular, quantization is a critical step in generating hardware-platform optimized models of today’s deep learning accelerators.

Typically, models are developed as full-precision \( (fp32) \) models and the quantization is part of the compilation flow. Pre-quantized models are models that are already quantized by the time the models are passed to compilation.

Why is this important for co-design? The ability to separate the quantization process from the hardware platform-specific compilation stage is important for optimal deep learning model execution. It allows researchers and modeling toolchain developers to focus on the quantization process independent of the specific hardware-platform. Thus, researchers/developer are able to create domain specific pre-quantized models, rather than rely on a general-purpose quantization approach that a hardware platform-specific compilation flow provides. However, to utilize hardware architecture specific features, the description of a pre-quantized model needs to be expressive. The design of this separated quantization process has the following goals:

1. Key quantization parameters should be embedded into the ONNX model.
   - No additional target-specific external metadata must be required.

2. Model should be directly executable with standard ONNX tools, such as ONNXruntime [1].

3. Standardized ONNX operators should only be used.
   - No custom operators which would prevent model usage in standard tools.
   - Closely matching output (within narrow margins) on all inference environments (software or hardware).

4. The description of the model should be expressive to convey hardware-specific operations.
• E.g. codify integer scale and right bit shift used by hardware to perform rescaling.

In the following section, the paper provides an introduction to symmetric quantization, followed by detailed examples of ONNX representation of pre-quantized Fully Connected Layer and Convolution Layers. This methodology is further applied to Tanh and Sigmoid activation functions.

2 Related Work

As quantization has gained importance, many deep learning frameworks and compilers have implemented some form of quantization. Below we cite related work. It should be noted, that in general the cited work is focused on quantized networks within their frameworks and compile chain. However, they are not addressing the need to codify an already quantized, i.e., a pre-quantized model, in a standard format. We cite here:

• TensorFlow lite [3]
• Nvidia® TensorRT™ [4]
• Onnxruntime how to quantize [5]
• Profile-guided quantization in Glow [6]
• Quantized networks with TVM [7]

3 Symmetric Quantization

The most common quantization approach represents floating point 32-bit (fp32) numbers with integer 8-bit (int8 or uint8) numbers [8–10], which reduces the memory footprint by a factor of four. Furthermore, 8-bit integer operations can be executed highly efficiently (low power and high performance) on machine learning accelerator hardware.

For symmetric quantization, where the zero offset is zero, equation 1 expresses the relationship between quantized tensor $X_q$ and original fp32 tensor $X$.

$$ X = scale_X \cdot X_q \tag{1} $$

There are multiple ways to determine the $scale_X$ in equation 1. One approach might be to profile the fp32 tensor to determine the maximum numerical range and mapping this range to the full int8 range. Another might be to minimize the overall quantization error by creating profile histograms and saturating the numerical range prior to mapping. Precisely, this is one of the motivations for this paper, i.e. decoupled quantized model development from the target hardware platform and its compiler. Giving the scale, and the data type of the quantized tensor (i.e. int8 or uint8), the quantized tensor $X_q$ can be calculated as $\frac{1}{scale_X} \cdot X$, with additional rounding and clipping stage to ensure that the quantized tensor is represented as proper int8 or uint8 values.

Similarly, to quantizing individual tensors, each layer of a network needs to be quantized. The fully connected layer, i.e. a matrix-matrix-multiply followed by addition of a bias tensor, serves as an example here. The fully connected layer with input tensor $X$, weight $W$, bias $B$ and output tensor $Y$ is given in equation 2

$$ Y = W \cdot X + B \tag{2} $$

$$ Y_q = \left(\frac{scale_W \cdot scale_X}{scale_Y}\right) \cdot \left(\frac{1}{scale_X} \cdot X_q + B_q\right) \tag{3} $$

$$ Y_q = \left(\frac{scale_W \cdot scale_X}{scale_Y}\right) \cdot Y_{\text{intermediate}} \tag{4} $$

The intermediate tensor $Y_{\text{intermediate}}$ is of INT32 data type and is the result of the MatMulInteger of the quantized weight and quantized input tensor with the integer addition of the quantized bias as given in equation 5

$$ Y_{\text{intermediate}} = W_q \cdot X_q + B_q \tag{5} $$

The bias $B$ is quantized to be of same scale as the output of the MatMulInteger operation and is given as INT32 value.

$$ B_q = \frac{1}{scale_W \cdot scale_X} \cdot B \tag{6} $$

The $\frac{scale_W \cdot scale_X}{scale_Y}$ represents the rescaling (aka output quantization) of the fully connected layer and is used in equation 4 to determine the quantized layer output. Just as for the individual tensor above, a rounding and clipping stage follows equation 4 to ensure that the output tensor is represented as proper int8 or uint8 values. Similarly, convolution layers will include such rescaling stage.

3.1 Rescaling

As shown in the previous section for fully connected layer, a rescaling with rounding and clipping is required after certain network layers. The rescaling values are floating point values, which can be greater
or smaller than 1. To perform the rescaling with integer arithmetic, the floating-point multiplication is replaced with an integer multiplication by an integer value followed by a bitwise right shift. A right shift by \( N \) bits is equivalent to a division by \( 2^N \). Utilizing 2 Mul operators both the integer value and the number of right shift bits can be codified within the ONNX network.

**Method with 2 Mul operators**

1. \( \text{Quant} \_\text{scale} \) is an Integer value represented as FLOAT.
2. \( \text{Quant} \_\text{shift} = 1/2^\text{shift} \), representing a right shift by \( N \) bits

Alternatively, only the floating-point scaling value is codified in ONNX and the conversion to integer value and number right shifts is the responsibility of the hardware-specific tool chain.

**Method with 1 Mul operator**

1. \( \text{Quant} \_\text{multiplier} = \text{Quant} \_\text{scale} \times \text{Quant} \_\text{shift} \)

For example, a \( \text{Quant} \_\text{multiplier} \) of 0.25 can be represented by \( \text{Quant} \_\text{scale} \) of 1 and \( \text{Quant} \_\text{shift} \) of \( 1/4 \). A \( \text{Quant} \_\text{multiplier} \) of \( \frac{1}{3} \) can be represented by \( \text{Quant} \_\text{scale} \) of 11184810 and \( \text{Quant} \_\text{shift} \) of \( 1/27 \). It should be noted, as the \( \text{Quant} \_\text{scale} \) integer value is stored as FLOAT, the largest exactly represented integer value is \( 2^{24} = 16,777,216 \)

The rounding and clipping that follows rescaling is codified in the pre-quantized ONNX network with the ONNX operator QuantizeLinear with \((\text{scale} = 1, \text{zero} \_\text{point} = 0)\). per QuantizeLinear API, the data type of the \text{zero} \_\text{point} argument determines the data type of the output tensor. I.e., an int8 \text{zero} \_\text{point} argument results in int8 output, while an uint8 \text{zero} \_\text{point} arguments results in uint8 output. Here, the QuantizedLinear is not used for rescaling, and \text{scale} is set to 1, as the scaling has already been codified using one or two MUL operators.

**4 Fully Connected Layer**

As a first example we show the fully connected layer of a Multi Layer Perceptron (MLP) network. The example shows the methodology applied to a complete network with input and output that can be run within the ONNX runtime. This is the base pattern that can be applied to larger MLP models and other networks containing fully connected layers. Fig. 1 shows the ONNX flow for a fully connected layer without an activation function, while Fig. 2 shows the layer with ReLU activation. In these figures, the ONNX graphs are visualized using Netron [11] tool on the left and the individual operator steps are shown on the right. For each operator, the data types of it’s input and output tensors are given. The ONNX operator MatMulInteger is used to express the matrix-matrix-multiply of the layer input of type int8 or uint8, with the weight coefficients given as int8 resulting in an output tensor of type int32. Following the MatMulInteger, the bias, as int32 data type is added using the ONNX Add operator. The rescaling is expressed here using two ONNX Mul operator with fp32 inputs, thus a ONNX Cast operator is added to cast the int32 into fp32. The final stage in this pattern is the rounding and clipping performed by the ONNX QuantizeLinear operator.

**Figure 1:** Fully Connected Layer without activation function. Rescaling expressed with two Mul operations.

**Figure 2:** Fully Connected Layer with ReLU activation function. Rescaling expressed with one Mul operation.
5 Convolution Layer

As second example we show the Convolution 2D layer of a Convolutional Neural Network (CNN). The methodology applied to a complete network with input and output that can be run within the ONNX runtime. Fig. 3 shows a Convolution layer without activation function. A ReLU activation function will be similarly handled as for the Fully Connected layer shown in Fig. 2 The ONNX operator ConvInteger is used to express the Convolution with the kernel coefficients given as int8 weights. Following the convolution, the bias, as int32 data type is added using the ONNX Add operator. The rescaling is expressed using the ONNX Mul operator with fp32 inputs, thus a ONNX Cast operator is added to cast the int32 into fp32. The final stage in this pattern is the rounding and clipping performed by the ONNX QuantizeLinear operator.

Figure 3: Convolution Layer without activation function. Rescaling expressed with one Mul operation.

6 Tanh and Sigmoid activation functions within an int8 quantized network

The above described methodology is further expanded to Tanh and Sigmoid activation functions. The figures in this section show the individual operator steps in detail only and omit the ONNX graph visualization. Fig. 4 shows the sequence of ONNX operators to codify an int8 quantized network with int8 tanh activation function. The Quant_scale and Quant_shift values are set such that the full input range of tanh is mapped to the quantized int8 range. The y_scale is determined by mapping the int8 range to the full output range of tanh. It should be noted that the ONNX operator for tanh is for fp32 input resulting in fp32 output. Setting the rescale to map to full input range and the y_scale in the above described way results in using a int8 tanh approximation.

Figure 4: Fully Connected Layer with int8 Tanh activation function. Rescaling expressed with two Mul operations.

An alternative approach is to perform tanh or sigmoid as floating-point functions. These functions might be implemented as fp32 or, as shown in the following examples, as fp16. Fig. 5 shows a mixed int8/fp16 flow which allows rescaling to a narrow input range (symmetric around zero) of tanh and execution of tanh function in fp16 precision. Fig. 6 shows the mixed int8/fp16 flow for the sigmoid acti-
Figure 6: Fully Connected Layer with fp16 Sigmoid activation function. Rescaling expressed with one Mul operation.

7 Conclusion

This paper presents a methodology to separate the quantization process from the model compilation stage which enables independent development, while allowing hardware/software co-design. Detailed examples are given for both MLP and CNN based networks, which can be extended to other networks in a straightforward fashion.

References

[1] “Onnx runtime,” https://onnxruntime.ai/
[2] “Onnx,” https://onnx.ai/
[3] “Tensorflow lite, post training quantization,” https://www.tensorflow.org/lite/performance/post_training_quantization
[4] “Nvidia tensorRT,” https://developer.nvidia.com/tensorrt
[5] “Onnx runtime, how to quantize onnx models,” https://www.onnxruntime.ai/docs/how-to/quantization.html
[6] N. Rotem, J. Fix, S. Abdurasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, “Glow: Graph lowering compiler techniques for neural networks,” CoRR, vol. abs/1805.00907, 2019. [Online]. Available: https://arxiv.org/abs/1805.00907
[7] A. Jain, S. Bhattacharya, M. Masuda, V. Sharma, and Y. Wang, “Efficient execution of quantized deep learning models: A compiler approach,” CoRR, vol. abs/2006.10226, 2020. [Online]. Available: https://arxiv.org/abs/2006.10226
[8] P. Warden, “How to quantize neural networks with tensorflow,” https://petewarden.com/2016/05/03/how-to-quantize-neural-networks-with-tensorflow/
[9] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” CoRR, vol. abs/1712.05877, 2017. [Online]. Available: http://arxiv.org/abs/1712.05877
[10] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, “Quantized convolutional neural networks for mobile devices,” CoRR, vol. abs/1512.06473, 2015. [Online]. Available: http://arxiv.org/abs/1512.06473
[11] “Netron viewer for neural network, deep learning and machine learning models,” https://github.com/lutzroeder/netron