A. Deployment Frameworks

A number of frameworks (Abadi et al., 2016; Chen et al., 2015; 2018; Gulli & Pal, 2017; Jia et al., 2014; Paszke et al., 2017; Seide & Agarwal, 2016; Vasilache et al., 2018) have been developed for deep learning. Many (Abadi et al., 2016; Chen et al., 2015; Jia et al., 2014; Paszke et al., 2017) offer a dataflow DAG abstraction for specifying NN workloads and provide optimization support for inference as well as training with automatic differentiation. These frameworks significantly reduce development cycles for deep learning algorithms and thus facilitate innovations in deep learning. However, a majority of these frameworks (Chen et al., 2015; Jia et al., 2014; Paszke et al., 2017) adopt a library-based approach that maps the NN operations to hardware through existing high-performance libraries, such as cuDNN (Chetlur et al., 2014) for GPUs, and GEMMLOWP (Jacob et al., 2017) and NNPACK (Dukhan, 2016) for CPUs. These libraries currently do not support low-precision inference (INT4), and since they are not open source we could not add that functionality. As such, for our analysis we adopted to use TVM (Chen et al., 2018), which provides a general graph and a tensor expression intermediate representation (IR) to support automatic code transformation and generation. TVM also equips a QNN dialect (Jain et al., 2020) to compile the quantization-specific operators of a quantized model. We choose TVM as our deployment framework for several reasons including: (i) its extensive support in the frontend high-level frameworks and the backend hardware platforms; and (ii) its decoupled IR abstraction that separates the algorithm specifications and the scheduling decisions. Augmenting TVM with our mixed-precision quantization support allows this optimization to be used by NNs written in different frameworks as well as for various target hardware platforms. In addition, the decoupled IR design in TVM allows the mixed-precision quantization optimization to be applied without affecting the specification of algorithms.

B. Quantization Method

Symmetric and Asymmetric Quantization. For uniform quantization, the scaling factor S is chosen to equally partition the range of real values r for a given bit width:

$$S = \frac{r_{max} - r_{min}}{2^b - 1},$$

where r_{max}, r_{min} denotes the max/min value of the real values, and b is the quantization bit width. This approach is referred to as *asymmetric quantization*. It is also possible to use a *symmetric quantization* scheme where $S = 2\max(|r_{max}|, |r_{min}|)/(2^b - 1)$ and $Z = 0$ (since zero will be exactly represented). As such, the quantization mapping can be simplified as:

$$Q(r) = \text{Int}\left(\frac{r}{S}\right). \tag{12}$$

Conversely, the real values r could be recovered from the quantized values $Q(r)$ as follows:

$$\tilde{r} = S\, Q(r). \tag{13}$$

Note that the recovered real values $\tilde{r}$ will not exactly match r due to the rounding operation. For HAWQ-V3, we use symmetric quantization for weights and asymmetric quantization for the activations.

Static and Dynamic Quantization. The scaling factor S depends on r_{max} and r_{min}. These can be precomputed for weights. However, for activations, each input will have a different range of values across the NN layers. In dynamic quantization, this range and the corresponding scaling factor is computed for each activation map during runtime. However, computing these values during inference has high overhead. This can be addressed with static quantization, in which this range is pre-calculated during the quantization phase and made independent of the input data, by analyzing the range of activations for different batches. We use static quantization for all of the experiments with HAWQ-V3. With these definitions, we next discuss how quantized inference is performed.

C. Fake Quantization for Convolution

In simulated quantization (also referred to as fake quantization in literature), all the calculations happen in FP32, which is different from the approach we used in Section 3.1. Similar to Section 3.1, suppose that the hidden activation is $h = S_h q_h$ and weight tensor is $W = S_w q_w$. In fake quantization, the output is calculated as:

$$a = (S_w q_w) * (S_h q_h). \tag{14}$$

That is the weight and activation are first represented back to FP32 precision, and then the calculation is performed. This result is then requantized and sent to the next layer as follows:

$$q_a = \text{Int}\left(\frac{a}{S_a}\right), \tag{15}$$

where S_a is the pre-calculated scale factor for the output activation. However, notice that here the requantization operation requires FP32 arithmetic (division by S_a), which is different from HAWQ-V3's Dyadic arithmetic that only uses integer operations. Figure C.1 shows the illustration of fake vs true quantization for a convolution (fully-connected) layer, without the BN layer. We also showed the corresponding illustration when BN is used in Figure 1.

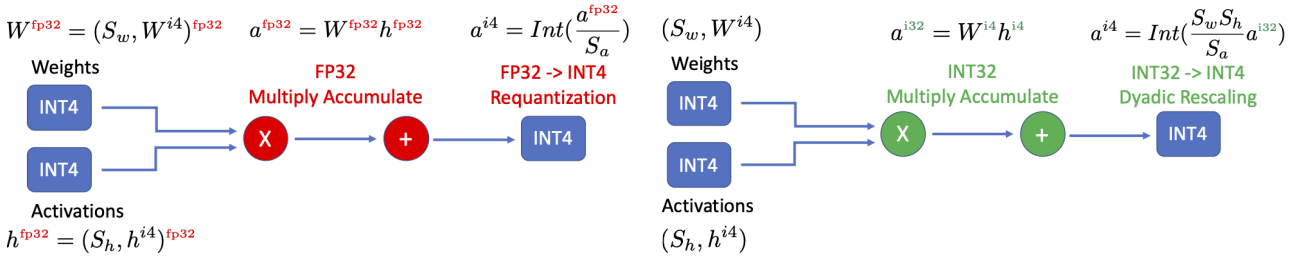

Figure C.1. Illustration of fake vs true quantization for a convolution (fully-connected) layer. (Left) In the simulated quantization (aka fake quantization), weights and activations are simulated as integers with floating point representation, and all the multiplication and accumulation happens in FP32 precision. However, with this approach, one cannot benefit from low-precision ALUs. (Right) An illustration of the integer-only pipeline with integer-only quantization. Note that with this approach, all the weights and activations are stored in integer format, and all the multiplications are performed with INT4 and accumulated in INT32 precision. Finally, the accumulated result is requantized to INT4 with dyadic scaling (denoted by $(\frac{S_w S_h}{S_a})$). Importantly, no floating point or even integer division is performed.

D. Batch Normalization Fusion

During inference, the mean and standard deviation used in the BN layer are the running statistics (denoted as μ and σ). Therefore, the BN operation can be fused into the previous convolutional layer. That is to say, we can combine BN and CONV into one operator as,

$$\begin{aligned} \text{CONV\_BN}(h) &= \beta \frac{Wh - \mu}{\sigma} + \gamma \\ &= \frac{\beta W}{\sigma} h + (\gamma - \frac{\beta \mu}{\sigma}) \equiv \bar{W} h + \bar{b}, \end{aligned} \tag{16}$$

where W is the weight parameter of the convolution layer and h is the input feature map. In HAWQ-V3, we use the fused BN and CONV layer and quantize $\bar{W}$ to 4-bit or 8-bit based on the setting, and quantize the bias term, $\bar{b}$ to 32-bit. More importantly, suppose the scaling factor of h is S_h and the scaling factor of $\bar{W}$ is $S_{\bar{W}}$. The scaling factor of $\bar{b}$ is enforced to be

$$S_{\bar{b}} = S_h S_{\bar{W}}. \tag{17}$$

So that the integer components of $\bar{W}h$ and $\bar{b}$ can be directly added during inference.

E. Concatenation Layer

The concatenation operation in Inception is an important component, which needs to be quantized carefully to avoid significant accuracy degradation. Concatenation layers are often used in the presence of pooling layers and other convolutions (a good example is the inception family of NNs). In HAWQ-V3, we use INT32 for the pooling layer since performing pooling on 4-bit can result in significant information loss. Furthermore, we perform separate dyadic arithmetic for the following concatenation operator in the inception module. Suppose the input of a concatenation block is denoted as $h = S_h q_h$, the output of the three convolutional branches are $m = S_m q_m$, $n = S_n q_n$, and $l = S_l q_l$,

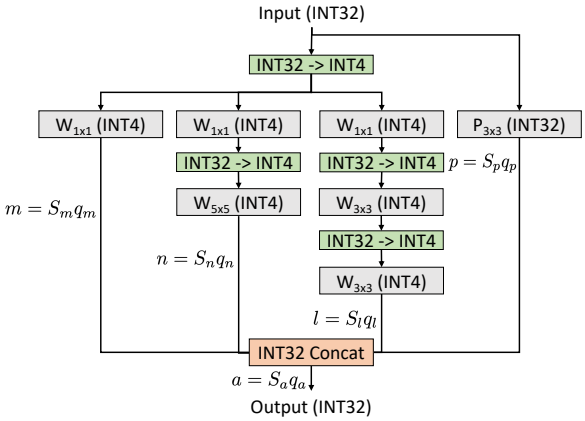

Figure E.1. Illustration of HAWQ-V3 for an inception module. Input feature map is given in INT32 precision, which is requantized to INT4 precision (green boxes) before being passed to the three convolutional branches. The pooling layer, however, is performed on the original input feature map in INT32. This is important since performing pooling on 4-bit data can result in significant information loss. The outputs for all the branches are scaled and requantized before being concatenated.

the output of the pooling branch is $p = S_p q_p$, and the final output is $a = S_a q_a$.

The pooling branch directly takes h as input, and the rest of the three convolutional branches take the quantized 4-bit tensor as input. After the computation of four separate branches, the output q_a is calculated with four DN operators:

$$q_a = \sum_{i \in \{m,n,l\}} \text{DN}\left(\frac{S_i}{S_a}\right) q_i + \text{DN}\left(\frac{S_p}{S_a}\right) q_p. \tag{18}$$

This scheme is represented in Figure E.1.

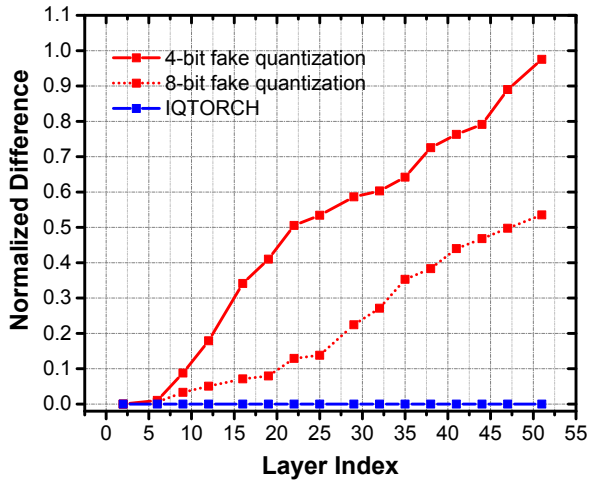

Figure G.1. The normalized difference between activation tensors in TVM and activation tensors in PyTorch during inference. The normalized difference is the L_2 norm of the difference between two activation counterparts divided by the L_2 norm of the TVM activation tensor.

F. Fake Quantization for Residual Connection

Similar to Section 3.3, Let us denote the activation passing through the residual connection as $r = S_r q_r$. the activation of the main branch before residual addition as $m = S_m q_m$. the final output after residual accumulation as $a = S_a q_a$. In fake quantization, the output a is calculated in FP32 as,

$$a = S_r q_r + S_m q_m. \tag{19}$$

Afterwards, requantization is performed,

$$q_a = \text{Int}(\frac{S_r q_r + S_m q_m}{S_a}), \tag{20}$$

where the Int operator requires FP32 multiplication.

Similarly, fake quantization for concatenation layer is calculated as (see Appendix E for notations):

$$q_a = \text{Int}(\frac{m + n + l + p}{S_a}). \tag{21}$$

G. Error Accumulation of Fake Quantization

There has been a common misunderstanding that using fake quantization is acceptable since one can use FP32 precision to perform Integer operations exactly. First, this is only true if the matrix multiplications only use integer numbers, without using very large numbers. The latter is the case in most ML applications. However, the problem is that many quantization approaches use fake quantization in a way that is different than the above argument.

For example, keeping the BN parameters in FP32 and not quantizing them is a major problem. It is not possible to simply ignore that and deploy a quantized model with FP32 BN parameters on integer-only hardware. This difference was discussed and illustrated in Figure 1.

Another very important subtle issue is how the residual connection is treated. As discussed in the previous section, the fake quantization approaches use FP32 arithmetic to perform the residual addition. The common (but incorrect) argument here again is that the INT arithmetic can be performed without error with FP32 logic. However, this is not the problem, since there is a subtle difference in how requantization is performed. In fake quantization, the results are first accumulated in FP32 and then requantized. However, it is not possible to perform such an operation on integer-only hardware, where the results are always quantized and then accumulated. This difference can actually lead to O(1) error.

For example consider the following case: assume $S_a = 1$, $r = 2.4$, $m = 4.4$ (see definition in Appendix F), and the requantization operator (Int) uses the "round to the nearest integer". Then using fake quantization, the output q_a is

$$q_a = \text{Int}(4.4 + 2.4) = 7. \tag{22}$$

However for true quantization, the output q_a is

$$q_a = \text{Int}(4.4) + \text{Int}(2.4) = 6. \tag{23}$$

This is an O(1) error that will propagate throughout the network. Also note that the problem will be much worse for low precision error. This is because an O(1) error for INT8 quantization is equivalent to a constant times (1/256), while for INT4 quantization it will be a constant times (1/16).

We also performed a realistic example on ResNet50 for the uniform quantization case. We perform fake quantization in PyTorch for fine-tuning and then deploy the model in TVM using integer-only arithmetic. Afterwards, we calculate the error between the feature map of PyTorch (fake quantization) and TVM (integer-only). In particular, we measure the normalized difference using L_2 norm:

$$\text{Normalized\_Difference} = \frac{\|x_1 - x_2\|}{\|x_1\|}, \tag{24}$$

where x_1, x_2 are the feature maps with fake quantization and the corresponding values calculated in hardware with integer-only arithmetic. In Figure G.1 we show the normalized difference between activation tensors in TVM and activation tensors in PyTorch during inference. As one can see, the numerical differences of the first layers are relatively small. However, this error accumulates throughout the layers and becomes quite significant in the last layers. Particularly, for uniform 4-bit quantization, the final difference becomes $> 95\%$.

H. Implementation Details

Models All the empirical results are performed using pretrained models from PyTorchCV (pyt, 2020) library. In

particular, we do not make any architectural changes to the models, even though doing so might lead to better accuracy. We consider three NN models, ResNet18, ResNet50, and InceptionV3, trained on the ImageNet dataset (Deng et al., 2009). For all the NNs, we perform BN folding to speed up the inference. All the calculations during inference are performed using dyadic arithmetic (i.e., integer addition, multiplication, and bit shifting), with no floating point or integer division anywhere in the network, including requantization stages.

Training details We use PyTorch (version 1.6) for quantizing models with HAWQ-V3. For all the quantization results, we follow the standard practice of keeping the first and last layer in 8-bit (note that input data is encoded with 8-bits for the RGB channels, which is quantized with symmetric quantization). We only use uniform quantization along with channel-wise symmetric quantization for weights, and we use layer-wise asymmetric quantization for activations. In order to perform static quantization, we set our momentum factor of quantization range (i.e., minimum and maximum) of activations to be 0.99 during training. Although further hyperparameter tuning may achieve better accuracy, for uniformity, all our experiments are conducted using learning rate 1e-4, weight decay 1e-4, and batch size 128.

Distillation As pointed out previously (Polino et al., 2018), for extra-low bit quantization (in our case uniform 4 bit and mixed 4/8 bit quantization), distillation may alleviate the performance degradation from quantization. Therefore, in addition to our basic results, we also present results with distillation (denoted with HAWQV3+DIST). Among other things, we do confirm the findings of previous work (Polino et al., 2018) that distillation can boost the accuracy of quantized models. For all different models, we apply ResNet101 (He et al., 2016) as the teacher, and the quantized model as the student. For simplicity, we directly use the naive distillation method proposed in (Hinton et al., 2014). (More aggressive distillation or fine-tuning with hyperparameter may lead to better results).

Latency Measurement We use TVM to deploy and tune the latency of the quantized models using Google Cloud Platform virtual machines with Tesla T4 GPUs and CUDA 10.2. We build the same NN models in TVM and tune the layerwise performance by using the autotuner. Once we have the tuned models, we run the end-to-end inference multiple times to measure the average latency. For the accuracy test, we load the parameters trained from PyTorch and preprocess it to the corresponding data layout that TVM requires. Then, we do inference in TVM and verify that the final accuracy matches the results in PyTorch.

Mixed-precision configuration For mixed-precision configuration, we first compute the trace of each layer (Dong et al., 2020) using PyHessian (Yao et al., 2019), and then solve the ILP problem using PULP (Roy & Mitchell, 2020). Our mixed-precision ILP problem can find the right bit-precision configuration with orders of magnitude faster run time, as compared to the RL based method (Wang et al., 2019; Wu et al., 2018). For instance, the entire trace computation can be finished within 30 minutes for all layers of ResNet50/InceptionV3 with only 4 RTX 6000 GPUs. Afterward, the ILP problem can be solved in **less than a second** (on a 15 inch MacBook Pro), as compared to more than 10/50 hours searching using RL (Wang et al., 2019) with 4 RTX 6000 GPUs.

I. ILP Result Interpolation

We plot the bit-precision setting for each layer of ResNet18 that the ILP solver finds for different latency constraints, as shown in Figure I.1. Additionally, we also plot the sensitivity (Ω_i in Eq. 8) and the corresponding speed up for each layer computed by quantizing the respective layer in INT8 quantization versus INT4. As can be seen, the bit configuration chosen by the ILP solver is highly intuitive based on the latency speed-up and the sensitivity. Particularly, when the mixed-precision model is constrained by the High-Latency setting (the first row of Figure I.1), only relatively insensitive layers, along with those that enjoy high INT4 speed-up, are quantized (i.e., layers 9, 14, and 19). However, for the more strict Low-Latency setting (last row of Figure I.1), only very sensitive layers are kept at INT8 precision (layer 1, 2, 3, 5, and 7).[7]

[7]Note that here layer 7 is the downsampling layer along with layer 5, so it is in the same bit setting as layer 5 even though the latency gain of layer 7 is limited.

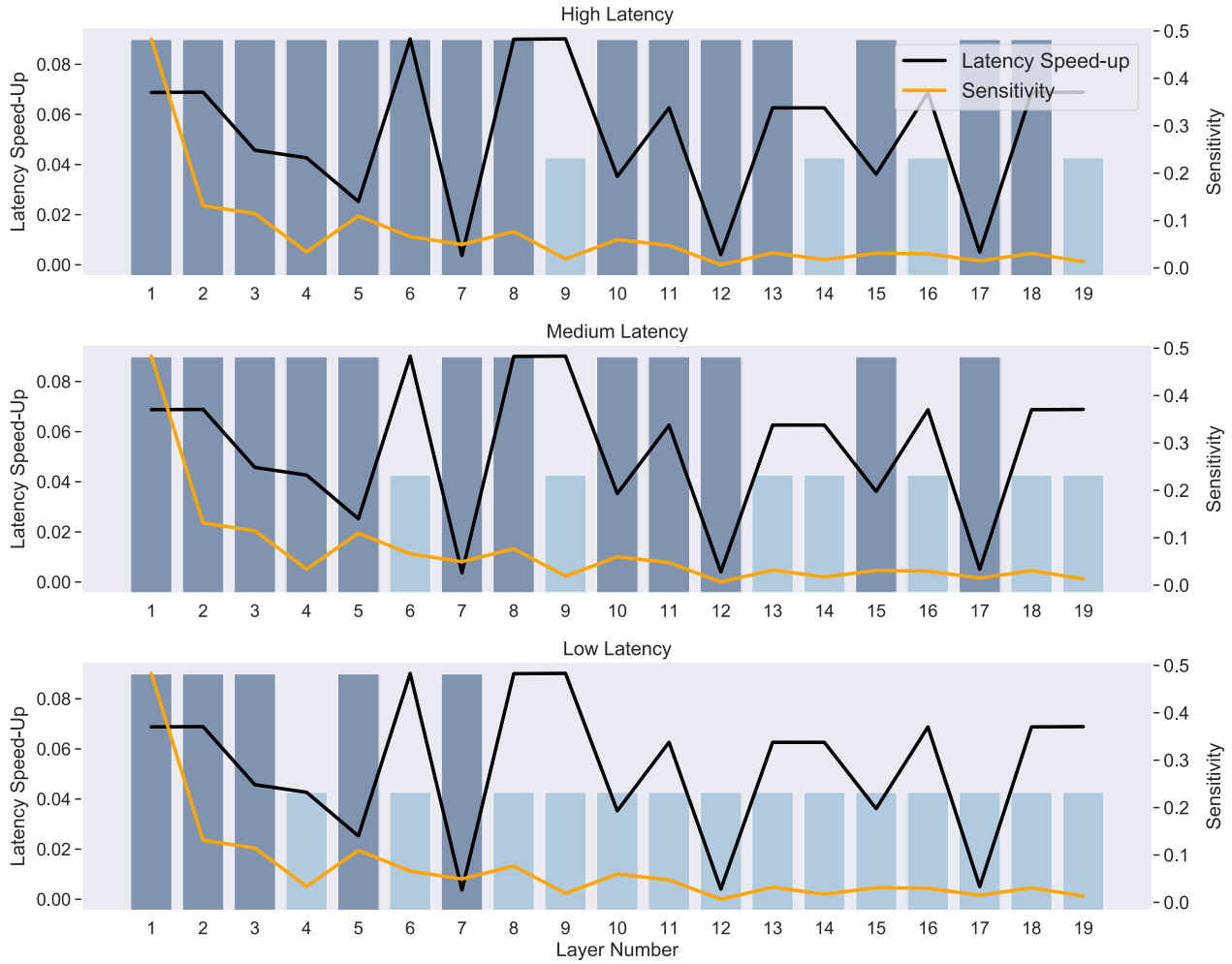

Figure I.1. Illustration of the final model specification that the ILP solver finds for ResNet18 with latency constraint. The black line shows the percentage of latency reduction for a layer executed in INT4 versus INT8, normalized by total inference reduction. Higher values mean higher speedup with INT4. The orange line shows the sensitivity difference between INT8 and INT4 quantization using second order Hessian sensitivity (Dong et al., 2020). The bit-precision setting found by ILP is shown in bar plots, with the blue and taller bars denoting INT8, and cyan and shorter bars denoting INT4. Each row corresponds to the three results presented in Table 2a with latency constraint. For the low latency constraint, the ILP solver favors assigning INT4 for layers that exhibit large gains in latency when executed in INT4 (i.e., higher values in dark plot) and that have low sensitivity (lower values in the orange plot).