HAWQ-V3: Dyadic Neural Network Quantization

Zhewei Yao[*,1], Zhen Dong[*,1], Zhangcheng Zheng[*,1], Amir Gholami[*,†,1], Jiali Yu[2,3]
Eric Tan[1], Leyuan Wang[3], Qijing Huang[1], Yida Wang[3], Michael W. Mahoney[1], Kurt Keutzer[1]
[1]University of California, Berkeley, [2]Shanghai Jiao Tong University, [3]Amazon

***Abstract*—Quantization is one of the key techniques used to make Neural Networks (NNs) faster and more energy efficient. However, current low precision quantization algorithms often have the hidden cost of conversion back and forth from floating point to quantized integer values. This hidden cost limits the latency improvement realized by quantizing NNs. To address this, we present HAWQ-V3, a novel dyadic quantization framework. The contributions of HAWQ-V3 are the following. (i) The entire inference process consists of only integer multiplication, addition, and bit shifting in INT4/8 mixed precision, without any floating point operations/casting or even integer division. (ii) We pose the mixed-precision quantization as an integer linear programming problem, where the bit precision setting is computed to minimize model perturbation, while observing application specific constraints on memory footprint, latency, and BOPS. (iii) To verify our approach, we develop the first open source 4-bit mixed-precision quantization in TVM, and we directly deploy the quantized models to T4 GPUs using only the Turing Tensor Cores. We observe an average speed up of $1.45\times$ for uniform 4-bit, as compared to uniform 8-bit, precision for ResNet50. (iv) We extensively test the proposed dyadic quantization approach on multiple different NNs, including ResNet18/50 and InceptionV3, for various model compression levels with/without mixed precision. For instance, we achieve an accuracy of 78.50% with dyadic INT8 quantization, which is more than 4% higher than prior integer-only work for InceptionV3. Furthermore, we show that mixed-precision INT4/8 quantization can be used to achieve higher speed ups, as compared to INT8 inference, with minimal impact on accuracy. For example, for ResNet50 we can reduce INT8 latency by 23% with mixed precision and still achieve 76.73% accuracy. Our framework and the TVM implementation have been open sourced [1].**

I. Introduction

An important step toward realizing pervasive deep learning is enabling real-time inference, both at the edge and in the cloud, with low energy consumption and state-of-the-art model accuracy. This will have a significant impact on applications such as real-time intelligent healthcare monitoring, autonomous driving, audio analytics, and speech recognition.

Over the past decade, we have observed significant improvements in the accuracy of Neural Networks (NNs) for various tasks. However, the state-of-the-art models are often prohibitively large and too compute heavy to be deployed for real-time use. The approaches proposed so far to address this challenge include rethinking NN architecture, the co-design of NN models and target hardware platforms, and optimization techniques such as pruning and quantization.

The focus of this paper is on one of the key NN optimization techniques—quantization—where low-precision quantized integer values are used to express the model parameters and feature maps. Quantization can significantly reduce the model footprint, and it can improve the inference energy consumption and latency, as calculation on integers is much less expensive than calculating on floating point numbers. However, existing low-precision quantization algorithms use simulated quantization, where the parameters are quantized in memory with integers, but are cast to floating point for inference. As a results, all or part of the inference operations (e.g. convolution, matrix operations, batch norm layers, residual connections) are performed in floating point precision. This does not provide latency speed up, it requires larger die area for the chip, and it can consume more power as compared to fully quantized inference. To address these problems, this paper introduces quantization with *dyadic arithmetic* (integer-only operations without division) in low and mixed precision. In particular, we make the following contributions:

- We develop a low-precision integer-only quantization framework HAWQ-V3 with integer multiplication, addition, and bit shifting. Importantly, no floating point and no integer division calculation is performed in the entire inference computational graph. This includes the batch normalization layers, which are typically kept at floating point precision.
- As a further (and orthogonal) contribution, we pose the problem of mixed-precision quantization as an Integer

[*]Equal contribution.
[†]Correspondence to: Amir Gholami (amirgh@berkeley.edu)

Linear Programming (ILP) problem to find the best bit-precision setting that minimizes model perturbation while observing application-specific constraints on model size, latency, and total bit operations.

- To verify the feasibility of our approach, we deploy the quantized integer-only models using Apache TVM [9] for INT8, INT4, and mixed-precision settings. To the best of our knowledge, our framework is the first that adds INT4 support to TVM. By profiling the latency of different layers, we show that we can achieve an average of $1.47\times$ speed up with INT4, as compared to INT8, when tested on a T4 GPU for ResNet50.
- We extensively test HAWQ-V3 on a wide range of workloads, including ResNet18, ResNet50, and InceptionV3, and show that we achieve a substantial performance improvement, as compared to prior state-of-the-art. For instance, we achieve an accuracy of 78.50% with dyadic INT8 quantization, which is more than 4% higher than prior integer-only work for InceptionV3. Furthermore, we show that mixed-precision INT4/8 quantization can be used to achieve higher speed up as compared to INT8 inference with minimal impact on accuracy. For example, for ResNet50 we can speedup latency by 23% as compared to INT8 and still achieve 76.73% accuracy.

Outline: We discuss related work in Section II. Then, we introduce the dyadic quantization methodology in Section III. Then, we present empirical results for accuracy and measured speedup in hardware in Section IV. Finally, we present a brief conclusion in Section V.

II. Related Work

There have been significant efforts recently to improve the trade-off between accuracy and efficiency of NN models. These can be broadly categorized as follows. (i) Designing new NN architectures that achieve high accuracy, while remaining compact by design [27, 44, 47]. (ii) Co-designing NN architecture and hardware together. This enables adaptation of the NN architecture for deployment to a specific hardware platform. Early work includes [18, 20, 24, 51]. (iii) Pruning redundant filters of NN layers. Seminal works here are [21, 33–35, 37, 53]. (iv) Distilling the knowledge from pre-trained model to a compact student model [23, 36, 42, 55]. (v) Using quantization (reduced precision) instead of FP32 precision. Quantization can significantly speed up inference time and reduce power consumption, and it is the approach taken in this paper. Here, we provide a more detailed overview of this related work.

a) Quantization.: A common solution is to compress NN models with quantization [5, 11, 15, 25, 29, 43, 56–58], where low bit precision is used for weights/activations. Quantization reduces model size without changing the original network architecture, and it could potentially permit the use of low-precision matrix multiplication or convolution.

While the gains on speed/power increase for low-precision quantization, low-precision quantization suffers from accuracy degradation. To address this, recent work uses non-uniform quantizers [56], channel-wise quantization [32], and progressive quantization-aware fine-tuning [57]. Other work tries to include periodic regularization to assist quantization [17, 38], apply post training quantization [6, 7, 26], or improve accuracy by changing the channel counts accordingly for different layers [11]. Despite these advances, performing uniform ultra low-bit quantization still results in a significant accuracy degradation. A promising direction is to use mixed-precision quantization [15, 46, 50, 59], where some layers are kept at higher precision, while others are kept at lower precision. However, a challenge with this approach is finding the right mixed-precision setting for the different layers of the NN. A brute force approach is not feasible since the search space is exponentially large in the number of layers.

HAQ [50] proposes to search this space by applying a Reinforcement Learning (RL) algorithm, while [52] uses a Differentiable Neural Architecture Search (DNAS) to accelerate this searching process. However, these searching methods require large computational resources, and their performance is very sensitive to hyper-parameters and even initialization. To address these issues, HAWQ [14, 15] introduces an automatic way to find good mixed-precision settings, where higher bit-width is assigned to more sensitive layers, based on the sensitivity obtained using the Hessian spectrum. However, the Pareto frontier method in [14] is not flexible enough to satisfy simultaneously different requirements on hardware. To address this, we propose here an ILP solution that can generate mixed-precision settings with various constraints (such as model size, BOPS and latency), and which can be solved within seconds on commodity hardware. The contemporary work of [26] also proposes to use an ILP. However, their approach is not hardware aware, and their approach uses FP32 casting.

Another issue is that the quantized weights and activations need to be converted to floating point precision during inference, as shown in Figure 1. This high-precision casting can have high overhead and limits

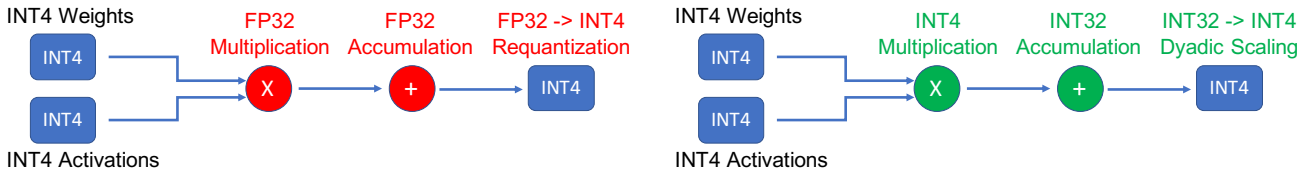

Figure 1: *Illustration of quantization for a convolution (fully-connected) layer. (Left) In the simulated quantization (aka fake quantization approach), weights and activations are simulated as integers with floating point representation, and all the multiplication and accumulation happen in FP32 precision. With this approach, one cannot benefit from low-precision ALUs. (Right) An illustration of integer-only pipeline with dyadic arithmetic. The weights and activations are stored in integer format, and the multiplication is performed using INT4 ALUs and accumulated in INT32 precision. Finally, the accumulated result is requantized to INT4 with bit shifting. Importantly, no floating point or even integer division is performed.*

inference speed, especially for hardware with limited on-chip memory. Using FP32 ALUs also requires larger die area in the chip, further limiting the peak computational capacity of the hardware.

The work of [29] addresses this casting problem by using full integer-only quantization in INT8 precision. However, there are several shortcomings associated with their approach (which are addressed in HAWQ-V3). First, their approach does not support low-precision or mixed-precision quantization. We will show that this is useful in practice, as it can improve the inference speed by up to 50% with small impact on accuracy. Second, both [29] and HAWQ are hardware agnostic and do not co-design/adapt the quantization for the target hardware. In contrast, the ILP approach in HAWQ-V3 is hardware aware, and it directly takes this into account when determining mixed-precision bit setting. Third, as we will discuss in the next section (Section III-B), the approach used in [29] leads to sub-optimal accuracy for INT8 quantization, while our approach can achieve up to 5% higher accuracy for INT8 inference. Finally, to address the absence of low-precision support in previous works [15, 29], we extend TVM to support low-precision (INT4) and mixed-precision quantization, and we validate our results by directly running the quantized model with low bit-width on the hardware.

b) Deployment Frameworks.: A number of frameworks [4, 8, 9, 19, 31, 41, 45, 49] have been developed for deep learning. Many [4, 8, 31, 41] 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 the innovations in deep learning. However, a majority of these frameworks [8, 31, 41] adopt a library-based approach that maps the NN operations to hardware through existing high-performance libraries, such as cuDNN [10] for GPUs, and GEMMLOWP [28] and NNPACK [16] 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 [9].

Apache TVM [9] is an open-source deep learning compiler stack 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 [30] 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.

III. Methodology

Assume that the NN has L layers with learnable parameters denoted as $\{W_1, W_2, ..., W_L\}$, with θ denoting the combination of all such parameters. For a supervised setting, the goal is to optimize the following empirical risk minimization loss function:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i; \theta), \tag{1}$$

where (x, y) is the input data and the corresponding label, $l(x, y; \theta)$ is the loss function (e.g., MSE or Cross Entropy loss), and N is the total number of data points. We assume that we have the trained model parameters θ given in floating point precision. Our goal is to quantize the model with the optimal trade-offs among memory footprint, speed, and accuracy. Below, we first define quantization and we then present our methodology.

Uniform Quantization. Quantization restricts NN weights/activations to a finite set of values defined as follows:

$$Q(r) = \text{Int}\left(\frac{r - Z}{S}\right), \tag{2}$$

where Q is the quantization operator, r is a real valued number (activation or a weight), S is a real valued scaling factor, and Z is the zero point, chosen such that the 0 value would exactly map to quantized values. Furthermore, Int maps a floating point value to an integer value through a rounding operation (e.g., round to nearest, truncation, or round to nearest even number).

This formulation for Q corresponds to uniform quantization. However, some work in the literature has also explored non-uniform quantization [12, 40, 50, 56]. Although non-uniform quantization may achieve higher accuracy for a fixed bit-width, such approaches are typically difficult to deploy on hardware to reduce latency. (However, they can reduce total model footprint.) As such, for HAWQ-V3, we only focus on uniform quantization.

Symmetric and Asymmetric Quantization. For uniform quantization, the scaling factor S is chosen to split equally 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:

$$Q(r) = \text{Int}\left(\frac{r}{S}\right). \tag{3}$$

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

$$\tilde{r} = S\ Q(r). \tag{4}$$

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.

A. *Quantized Matrix Multiplication and Convolution*

Consider a layer with hidden activation denoted as h and weight tensor denoted as W. First, h and W are quantized to $S_h q_h$ and $S_w q_w$, where S_h and S_w are the real valued quantization scales, q_h and q_W are the corresponding quantized integer values. The output result, denoted with a, can be computed as follows:

$$a = S_w S_h (q_w * q_h), \tag{5}$$

where $q_w * q_h$ is the matrix multiplication (or convolution) calculated with integer in low precision (e.g., INT4) and accumulated in INT32 precision. As such, a will be in INT32 precision. This result is then requantized and sent to the next layer as follows:

$$q_a = \text{Int}\left(\frac{a}{S_a}\right) = \text{Int}\left(\frac{S_w S_h}{S_a}(q_w * q_h)\right), \tag{6}$$

where S_a is the pre-calculated scale factor for the output activation.

To summarize, in HAWQ-V3, the $q_w * q_h$ operation is performed with low-precision integer-only multiplication and INT32 accumulation, and the final INT32 result is quantized by scaling it with $S_w S_h / S_a$. The latter is a floating point scaling that needs to be multiplied with the accumulated result (in INT32 precision). A naive implementation requires floating point multiplication for this stage. However, this can be avoided by enforcing this scaling to be a dyadic number.

Dyadic Numbers. Dyadic numbers are a class of p-adic numbers which are fractional numbers with the format of $\frac{b}{p^c}$, where $b,\ c$ are two integer numbers and $p = 2$. Examples dyadic numbers are 1/4, 3/8, 13/16, etc. In fact, all the values stored using IEEE floating point standards use dyadic numbers for the mantissa. If we can represent $\frac{S_w S_h}{S_a}$ in the form $\frac{b}{2^c}$, then the scaling

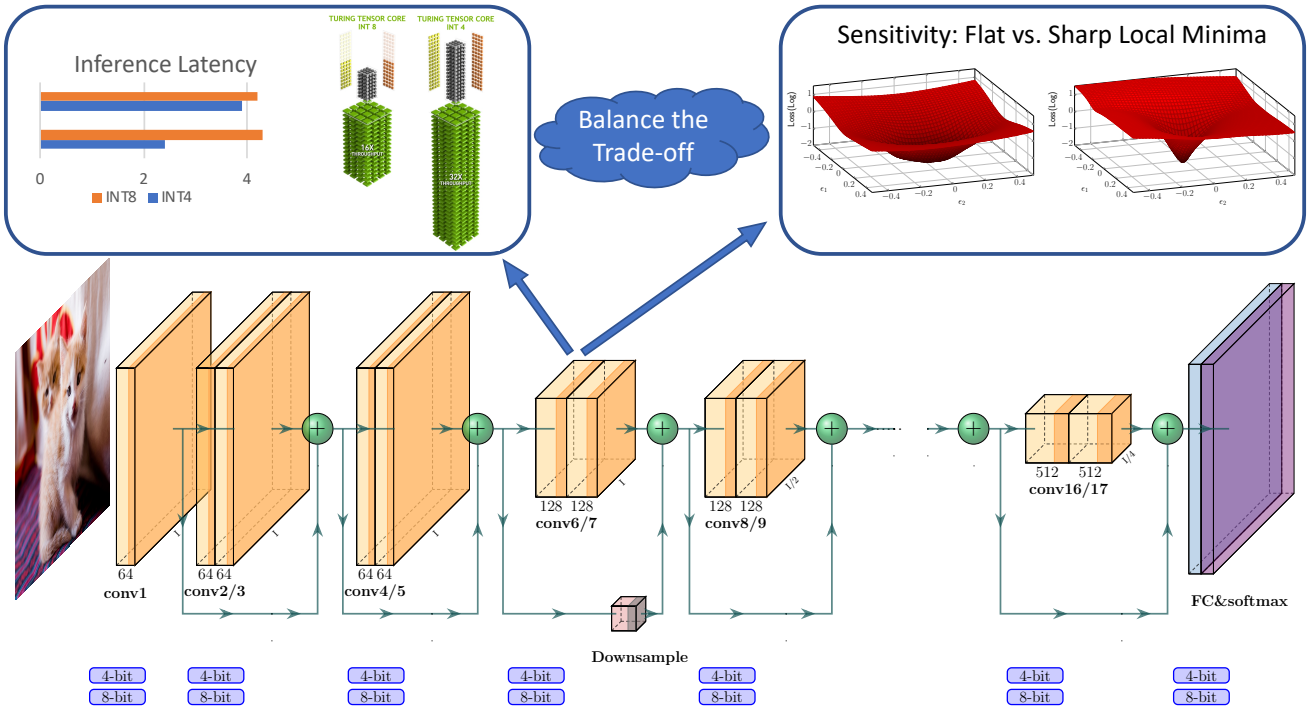

Figure 2: *Illustration of inference speed and generalization performance trade-off of ResNet18. For each single layer, we need to consider the speedup of INT4 versus INT8 as well as the sensitivity based on the second order (Hessian) sharpness [14] of this layer.*

in Eq. 6 can be efficiently performed using INT32 integer multiplication and bit shifting. Given a specific $\frac{S_w S_h}{S_a}$, we use DN (representing Dyadic Number) to denote the function that can calculate the corresponding b and c:

$$\frac{b}{2^c} = \text{DN}\left(\frac{S_w S_h}{S_a}\right). \tag{7}$$

An advantage of using dyadic numbers is that it removes the need to support division (which typically has an order of magnitude higher latency than multiplication) in the hardware. This can also be useful for extreme edge deployment cases such as microprocessors which have very limited die area. Removing or reducing ALUs that support division could allow the architect to add more multipliers instead and increase the performance of the chip with limited resources. This approach is used for INT8 quantization in [29], and we enforce all the rescaling to be dyadic for low-precision and mixed-precision quantization as well.

B. Batch Normalization

Batch normalization (BN) is an important component of most NN architectures, especially for computer vision applications. BN performs the following operation to an input activation a:

$$\text{BN}(a) = \beta \frac{a - \mu_B}{\sigma_B} + \gamma \tag{8}$$

where μ_B and σ_B are the mean and standard deviation of a, and β, γ are trainable parameters. During inference, these parameters are fixed, and as such the BN operations could be *folded* (fused) into the convolution. An important problem is that quantizing the BN parameters could lead to significant accuracy degradation. As such, many prior quantization methods keep BN parameters in FP32 precision (e.g., [7, 11, 12, 14, 40, 56], just to name a few). This is undesirable, as the target hardware has to support FP32 for executing this layer.

The prior integer-only work of [29] attempts to address the accuracy drop of BN quantization by folding Convolution (Conv) and BN layers together and then updating the running statistics of the BN layer. This requires computing each convolution layer twice, once without BN and then with BN (as illustrated in [29, Figure C8]). However, we found this is unnecessary and can degrade the accuracy. Instead, in HAWQ-V3, we follow a simpler approach where we first keep the Conv and BN layer unfolded, and then we perform standard QAT by backpropagating the

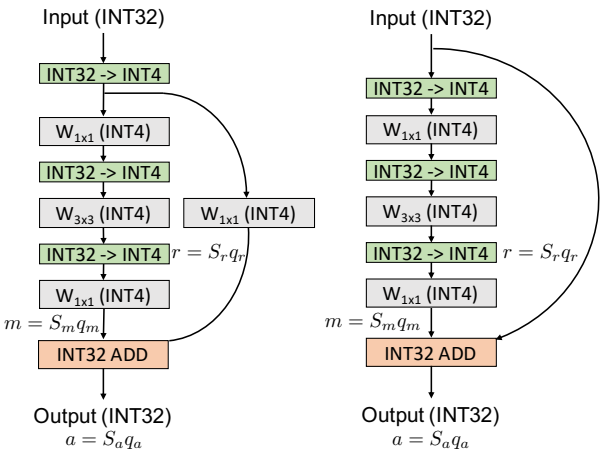

Figure 3: *Illustration of* HAWQ-V3 *for a residual block with and without transition layer. Input feature map is given in INT32 precision, which is requantized to INT4 precision (green boxes) before any convolution layer (gray boxes). The BN layer is folded into the convolution. The residual addition is performed in INT32 precision, and the final accumulated result is re-scaled and sent to the next layer. For blocks with a transition layer, we only quantize the input once to INT4 and we use the same result for both* 1×1 *convolutions.*

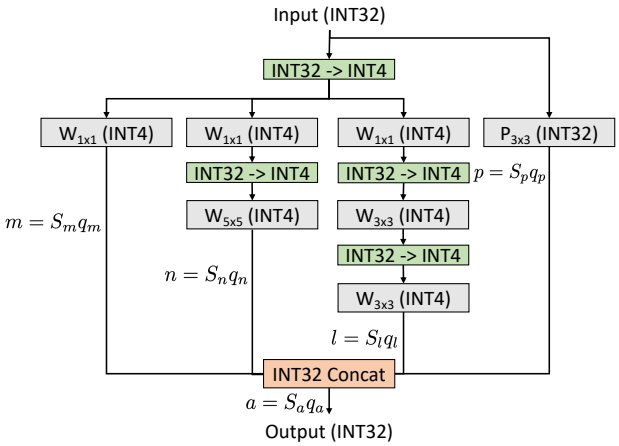

Figure 4: *Illustration of* HAWQ-V3 *for an inception module. Input feature map is given in INT32 precision, which is requantized to INT4 precision (green boxes) before being passed to the three convolutional branches. The pooling layer, however, is performed on the original input feature map in INT32. 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.*

gradients as usual. After several epochs, we freeze the running statistics in the BN layer and only then perform CONV+BN folding and quantized to the target bit-width. As we will show in Section IV, this approach can result in significantly better accuracy as compared to [29].

C. Residual Connection

Residual connection [22] is another important component in many NN architectures. Similar to BN, quantizing the residual connections often leads to significant accuracy degradation. As such, many prior works keep the residual connection in FP32 precision [12, 50, 56].

In HAWQ-V3, we only use INT32 for the residual branch, and we perform the following steps to ensure that the addition operation can happen with dyadic arithmetic. Let us denote the activation passing through the residual connection as $r = S_r q_r$.[1] Furthermore, let us denote the activation of the main branch before residual addition as $m = S_m q_m$. We are interested to find the final output after residual accumulation denoted by $a = S_a q_a$, which can be computed as follows:

$$q_a = \text{DN}\left(\frac{S_m}{S_a}\right) q_m + \text{DN}\left(\frac{S_r}{S_a}\right) q_r. \tag{9}$$

[1]This is either the input or the output activation after the downsampling layer.

This is schematically illustrated in Figure 3. Note that during inference the scale values are known (since we use static quantization for improved speed). As such, the residual addition can be performed with only integer multiplication, addition, and bit shifting (since the scalar divisions are enforced to be dyadic).

D. Concatenation Layer

The concatenation operation in Inception is an important component, which needs to be quantized carefully to avoid significant accuracy degradation. 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 $l = S_l q_l$, 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:

$$q_a = \sum_{i \in \{m,n,l\}} \text{DN}\left(\frac{S_i}{S_h}\right) q_i + \text{DN}\left(\frac{S_p}{S_h}\right) q_p. \quad (10)$$

This scheme is represented in Figure 4.

E. Mixed Precision and Integer Linear Programming

Uniformly quantizing all layers of an NN to ultra low bit-width could lead to significant accuracy degradation. However, it is possible to benefit from low-precision quantization by keeping a subset of sensitive layers at high precision. The basic idea is to keep more sensitive layers at higher precision and less sensitive layers at lower precision. An important component of HAWQ-V3 is that we directly consider hardware specific metrics such as latency. This is important since a layer's latency does not necessarily halve when quantized from INT8 to INT4 precision. In fact, as we will discuss in Section IV, there are specific layer configurations that do not have any speed up when quantized to low precision, and some that superlinearly benefit from quantization. As such, quantizing the former will not lead to any latency improvement, and will only hurt accuracy. Therefore, it is better to keep such layers at high precision, even if they have low sensitivity. These trade-offs between accuracy and latency should be taken into consideration when quantizing them to low precision. Importantly, these trade-offs are hardware specific as latency in general does not correlate with model size and/or FLOPS. However, we can consider this by measuring directly the latency of executing a layer in quantized precision on the target hardware platform. This trade-off is schematically shown in Figure 2 (and later quantified in Figure 5). We can use an Integer Linear Programming (ILP) problem to formalize the problem definition of finding the bit-precision setting that has optimal trade-off as described next.

Assume that we have B choices for quantizing each layer (i.e., 2 for INT4 or INT8). For a model with L layers, the search space of the ILP will be B^L. The goal of solving the ILP problem is to find the best bit configuration among these B^L possibilities that results in optimal trade-offs between model perturbation Ω, and user specified constraints such as model size, latency, and accuracy. Each of these bit-precision settings could result in a different model perturbation. To make the problem tractable, we assume that the perturbations for each layer are independent of each other (i.e., $\Omega = \sum_{i=1}^{L} \Omega_i^{(b_i)}$, where $\Omega_i^{(b_i)}$ is the i-th layer's perturbation with b_i bit). This allows us to precompute the sensitivity of each layer separately, and it only requires BL computations. For the sensitivity metric, we use the Hessian based perturbation proposed in [14, Eq. 2.11]. The ILP problem tries to find the right bit precision that minimizes this sensitivity, as follows:

$$\text{Objective: } \min_{\{b_i\}_{i=1}^{L}} \sum_{i=1}^{L} \Omega_i^{(b_i)}, \quad (11)$$

$$\text{Subject to: } \sum_{i=1}^{L} M_i^{(b_i)} \leq \text{ Model Size Limit}, \quad (12)$$

$$\sum_{i=1}^{L} G_i^{(b_i)} \leq \text{ BOPS Limit}, \quad (13)$$

$$\sum_{i=1}^{L} Q_i^{(b_i)} \leq \text{ Latency Limit}. \quad (14)$$

Here, $M_i^{(b_i)}$ denotes the size of i-th layer with b_i bit quantization, $Q_i^{(b_i)}$ is the associated latency, and $G_i^{(b_i)}$ is the corresponding BOPs required for computing that layer. The latter measures the total Bit Operations for calculating a layer [48]:

$$G_i^{(b_i)} = b_{w_i} b_{a_i} \text{MAC}_i,$$

where MAC_i is the total Multiply-Accumulate operations for computing the i-th layer, and b_{w_i}, b_{a_i} are the bit precision used for weight and activation (which is the same in HAWQ-V3). Note that it is not necessary to set all these constraints at the same time. Typically, which constraint to use depends on the end-user application.

We solve the ILP using open source PULP library [2] in Python, where we found that for all the configurations tested in the paper, the ILP solver can find the solution in less than 1 second given the sensitivity metric. (For comparison, the RL based method of [50] could take tens of hours to find the right bit-precision setting [14].) In Section IV, we show the results with different constraints.

F. Hardware Deployment

Model size alone is not a good metric for the efficiency of NNs. In fact, it is quite possible that a small NN model would have higher latency and consume a larger amount of energy for inference. The same is also true for FLOPs. The reason is that neither model size nor FLOPs can account for cache misses, data locality, memory bandwidth, underutilization of hardware, etc. To address this, we need to deploy the quantized NN model on a system and directly measure the latency.

We provide support for both INT4 and INT8, as well as mixed-precision INT4/8 inference. We target

Nvidia Turing Tensor Cores of T4 GPU for deployment, as it supports both INT8 and INT4 precision and has been enhanced for deep learning network inference. The throughput of INT4 on T4 is higher than the tensor performance on V100 GPU while its AWS instance cost is 10× lower. While the T4 hardware supports INT4 Tensor Core operations, however, as of writing this paper no library supports INT4 inference. The only API available is the WMMA kernel call which is a micro-kernel for performing matrix-matrix operations in INT4 precision on Tensor Cores. And the only open source library which supports INT4 Tensor Core GEMM operations is called Cutlass [39]. There is also no existing compiler that would map a NN quantized to INT4 to Tensor Cores using WMMA instructions.

To address this challenge, another contribution of our work is extending TVM [9] to support INT4 inference with/without mixed precision with INT8. This is important so we can verify the speed benefits of mixed-precision inference. To accomplish this, we had to add new features in both graph-level IR and operator schedules to make INT4 inference efficient. For instance, when we perform optimizations such as memory planning, constant folding, and operator fusion, at the graph-level IR, 4-bit data are involved. However, on byte-addressable machines, manipulating 4-bit data individually leads to inefficiency in storage and communication. Instead, we pack eight 4-bit elements into an INT32 data type and perform the memory movement as a chunk. In the final code generation stage, the data type and all memory access will be adjusted for INT32.

By adopting similar scheduling strategies to Cutlass [39], we implement a new direct convolution schedule for Tensor Cores for both 8-bit and 4-bit data in TVM. We set the knobs for the configurations such as thread size, block size, and loop ordering so that the auto-tuner in TVM could search for the best latency settings.

Another important point is that we have completed the pipeline to test directly the trained weights and to avoid using random weights for speed measurements. This is important, since small discrepancies between the hardware implementation may go unnoticed from the quantization algorithm in NN training framework (PyTorch in our case) which does not use TVM for the forward and backward propagation. To avoid any such issue, we made sure that the results between TVM and PyTorch match for every single layer and stage to machine-precision accuracy, and we verified the final Top-1 accuracy when executed in the hardware with integer-only arithmetic.

IV. Results

In this section, we first discuss the results of dyadic quantization HAWQ-V3 on various models (ResNet18/50 and InceptionV3) on ImageNet dataset. Detailed discussion on the implementation and set up is provided in Appendix Section A. We first show that dyadic quantization can achieve high accuracy (sometimes even higher than some prior work using simulated quantization with FP32 arithmetic). We show results for INT8, INT4, and mixed-precision INT4/8 with/without distillation. Afterwards, we study different use cases of how the ILP problem can be used and the corresponding trade-offs between model size, latency, and accuracy.

A. Dyadic Quantization Results

We first start with ResNet18/50 and InceptionV3 quantization on ImageNet, and compare the performance of HAWQ-V3 with other approaches, as shown in Table I. For all the experiments we made sure to report and compare with the highest accuracy known for the baseline NN model in FP32 (i.e., we use a strong baseline for comparison). This is important since using a weak baseline accuracy could lead to misleading quantization accuracy.

Uniform 8-bit Quantization. Our 8-bit dyadic quantization achieves similar accuracy compared to the baseline (0.09% higher for ResNet18 and 0.15% lower top-1 accuracy for ResNet50, as compared to the strong ResNet baseline). Furthermore, for all models HAWQ-V3 achieves higher accuracy than RVQuant (which uses non-uniform quantization with FP32 arithmetic) and the Integer-Only approach. For instance, on ResNet50, we achieve 2.68% higher accuracy as compared to [29]. This is in part due to our BN folding strategy that was described in Section III-B.

Uniform 4-bit Quantization To the best of our knowledge, 4-bit results of HAWQ-V3 are the first dyadic quantization results reported in the literature. The accuracy results for ResNet18/50, and InceptionV3 are quite high, despite the fact that all of the inference computations are restricted to be integer multiplication, addition, and bit shift. While there is some accuracy drop, this should not be incorrectly interpreted that uniform INT4 is not useful. On the contrary, one has to keep in mind that certain use cases have strict latency and memory footprint limit for which this may be the best solution. Moreover, for slightly more relaxed constraints, one could use mixed precision which can significantly improve the accuracy trade-off. For completeness, we also include comparison with the recent contemporary work of [26], where we achieve similar accuracy.

Table I: *Quantization results for ResNet18/50 and InceptionV3. Here, we abbreviate Integer-Only Quantization as "IntOnly", Uniform Quantization as "Uniform", Weight Precision and Activation Precision as "Precision", Model Size as "Size", Bit Operations as "BOPS", and Top-1 Accuracy as "Top-1". Also, "WxAy" means weight with x-bit and activation with y-bit, and 4/8 means mixed precision with 4 and 8 bits. "MP" means mixed precision with bitwidth ranging from 1-bit to 8-bit, and "W1*" means the bitwidth is 1-bit but the network architecture is changed (by using more channels). Our result with/without distillation is represented as* HAWQ-V3+DIST/HAWQ-V3.

(a) *ResNet18*

Method	IntOnly	Uniform	Open Source	Precision	Size (MB)	BOPS (G)	Top-1
Baseline	✗	–	✓	W32A32	44.6	1858	71.47
RVQuant [40]	✗	✗	✗	W8A8	11.1	116	70.01
HAWQ-V3	✓	✓	✓	W8A8	11.1	116	**71.56**
PACT [12]	✗	✓	✗	W5A5	7.2	50	69.80
LQ-Nets [56]	✗	✗	✓	W4A32	5.8	225	70.00
HAWQ-V3	✓	✓	✓	W4/8A4/8	6.7	72	70.22
HAWQ-V3+DIST	✓	✓	✓	W4/8A4/8	6.7	72	**70.38**
CalibTIB[26]	✗	✓	✓	W4A4	5.8	34	67.50
HAWQ-V3	✓	✓	✓	W4A4	5.8	34	**68.45**

(b) *ResNet50*

Method	IntOnly	Uniform	Open Source	Bit Precision	Size (MB)	BOPS (G)	Top-1
Baseline	✗	–	✓	W32A32	97.8	3951	77.72
Integer Only [29]	✓	✓	✗	W8A8	24.5	247	74.90
RVQuant [40]	✗	✗	✗	W8A8	24.5	247	75.67
HAWQ-V3	✓	✓	✓	W8A8	24.5	247	**77.58**
PACT [12]	✗	✓	✗	W5A5	16.0	101	76.70
LQ-Nets [56]	✗	✗	✓	W4A32	13.1	486	76.40
RVQuant [40]	✗	✗	✗	W5A5	16.0	101	75.60
HAQ [50]	✗	✗	✓	WMPA32	9.62	520	75.48
OneBitwidth [11]	✗	✓	✗	W1*A8	12.3	494	76.70
HAWQ-V3	✓	✓	✓	W4/8A4/8	18.7	154	75.39
HAWQ-V3+DIST	✓	✓	✓	W4/8A4/8	18.7	154	**76.73**
CalibTIB[26]	✗	✓	✓	W4A4	13.1	67	73.70
HAWQ-V3	✓	✓	✓	W4A4	13.1	67	**74.24**

(c) *InceptionV3*

Method	IntOnly	Uniform	Open Source	Bit Precision	Size (MB)	BOPS (G)	Top-1
Baseline	✗	–	✓	W32A32	90.9	5850	78.88
Integer Only [29]	✓	✓	✗	W8A8	22.7	366	74.20
RVQuant [40]	✗	✗	✗	W8A8	22.7	366	74.22
HAWQ-V3	✓	✓	✓	W8A8	22.7	366	**78.76**
Integer Only [29]	✓	✓	✗	W7A7	20.1	280	73.70
HAWQ-V3	✓	✓	✓	W4/8A4/8	19.6	265	74.65
HAWQ-V3+DIST	✓	✓	✓	W4/8A4/8	19.6	265	**74.72**
HAWQ-V3	✓	✓	✓	W4A4	12.3	92	70.39

Mixed 4/8-bit Quantization. The mixed-precision results improve the accuracy by several percent for all the models, while slightly increasing the memory footprint of the model. For instance, the mixed-precision result for ResNet18 is 1.88% higher than the INT4 counterpart with an increased 1.9MB model size. Further improvements are also possible with distillation (represented as HAWQ-V3+DIST in the table). For ResNet50, the distillation can boost the mixed-precision by 1.34%. We found that distillation helps most for mixed-precision quantization, and we found little to no improvement for uniform INT8, or uniform INT4 quantization cases.[2]

HAWQ-V3 achieves comparable accuracy, as compared to prior quantization methods including both uniform and mixed-precision quantization (e.g., PACT, RVQuant, OneBitwidth, HAQ which use FP32 arithmetic, and/or non-standard bit precision such as 5 bits, or different bit-width for weights and activations). Similar observations hold for InceptionV3, as reported in Table Ic.

B. Mixed-precision Results with Different Constraints

Here, we discuss various scenarios where different constraints could be imposed for quantization, and the interesting trade-offs associated with these scenarios. The ILP problem in Eq. 11 has the three constraints of model size, BOPS, and latency. We consider three different thresholds for each of the constraints and study how the ILP balances the trade-offs to obtain an optimal quantized model. In particular, we set the threshold values (model size, BOPS, or latency) to be higher than the corresponding ones for INT4.

The table rows of Table II are split in three sections of Size (model size), BOPS, and Latency. The constraint that is enforced in the ILP is shown in bold for each section (only one constraint is enforced at a time). For example, the first section shows the result that the ILP problem Eq. 11 is solved to find the optimal mixed-precision bit setting with limited model size (without enforcing BOPS or latency) that achieves minimum model perturbation. For all mixed-precision results, we train the model with/without distillation and report both results in the table. As before, the distillation can improve the performance, especially for those hard constraints (i.e., the model that is closer to INT4). The Low-Latency results for both ResNet18/50 are boosted more than 1% as compared to standard training, which is significant.

[2]We used simple distillation without extensive tuning. One might be able to improve the results further with more sophisticated distillation algorithms.

Table II: *Mixed-precision quantization results for ResNet18 and ResNet50 with different constraints. Here, we abbreviate constraint level as "Level". Model Size as "Size", Bit Operations as "BOPS", the speedup as compared to INT8 results as "Speed", and Top-1 Accuracy as "Top-1", The last column of Top-1 represents results of* HAWQ-V3 *and* HAWQ-V3+DIST.

(a) *ResNet18*

	Level	Size (MB)	BOPS (G)	Speed	Top-1
INT8	–	11.2	114	1x	71.56
Size	High	**9.9**	103	1.03x	71.20/71.59
	Medium	**7.9**	98	1.06x	70.50/71.09
	Low	**7.3**	95	1.08x	70.01/70.66
BOPS	High	8.7	**92**	1.12x	70.40/71.05
	Medium	6.7	**72**	1.21x	70.22/70.38
	Low	6.1	**54**	1.35x	68.72/69.72
Latency	High	8.7	92	**1.12x**	70.40/71.05
	Medium	7.2	76	**1.19x**	70.34/70.55
	Low	6.1	54	**1.35x**	68.56/69.72
INT4	–	5.6	28	1.48x	68.45

(b) *ResNet50*

	Level	Size (MB)	BOPS (G)	Speed	Top-1
INT8	–	24.5	247	1x	77.58
Size	High	**21.3**	226	1.09x	77.38/ 77.58
	Medium	**19.0**	197	1.13x	75.95/76.96
	Low	**16.0**	168	1.18x	74.89/76.51
BOPS	High	22.0	**197**	1.16x	76.10/76.76
	Medium	18.7	**154**	1.23x	75.39/76.73
	Low	16.7	**110**	1.30x	74.45/76.03
Latency	High	22.3	199	**1.13x**	76.63/76.97
	Medium	18.5	155	**1.21x**	74.95/76.39
	Low	16.5	114	**1.28x**	74.26/76.19
INT4	–	13.1	67	1.45x	74.24

We start with the model size and BOPS constraints, where the results for different ILP constraints are shown in Table II. Several very interesting observations can be made from these results. (i) The correlation between model size and BOPS is weak, i.e., larger model size does not mean higher BOPS and vice versa (see Medium-Size and High-BOPS for ResNet18). (ii) The model size does not directly correlate with accuracy. For ResNet18, High-BOPS has larger model size as compared to Medium-Size. However, the accuracy of Medium-Size is higher. Similar

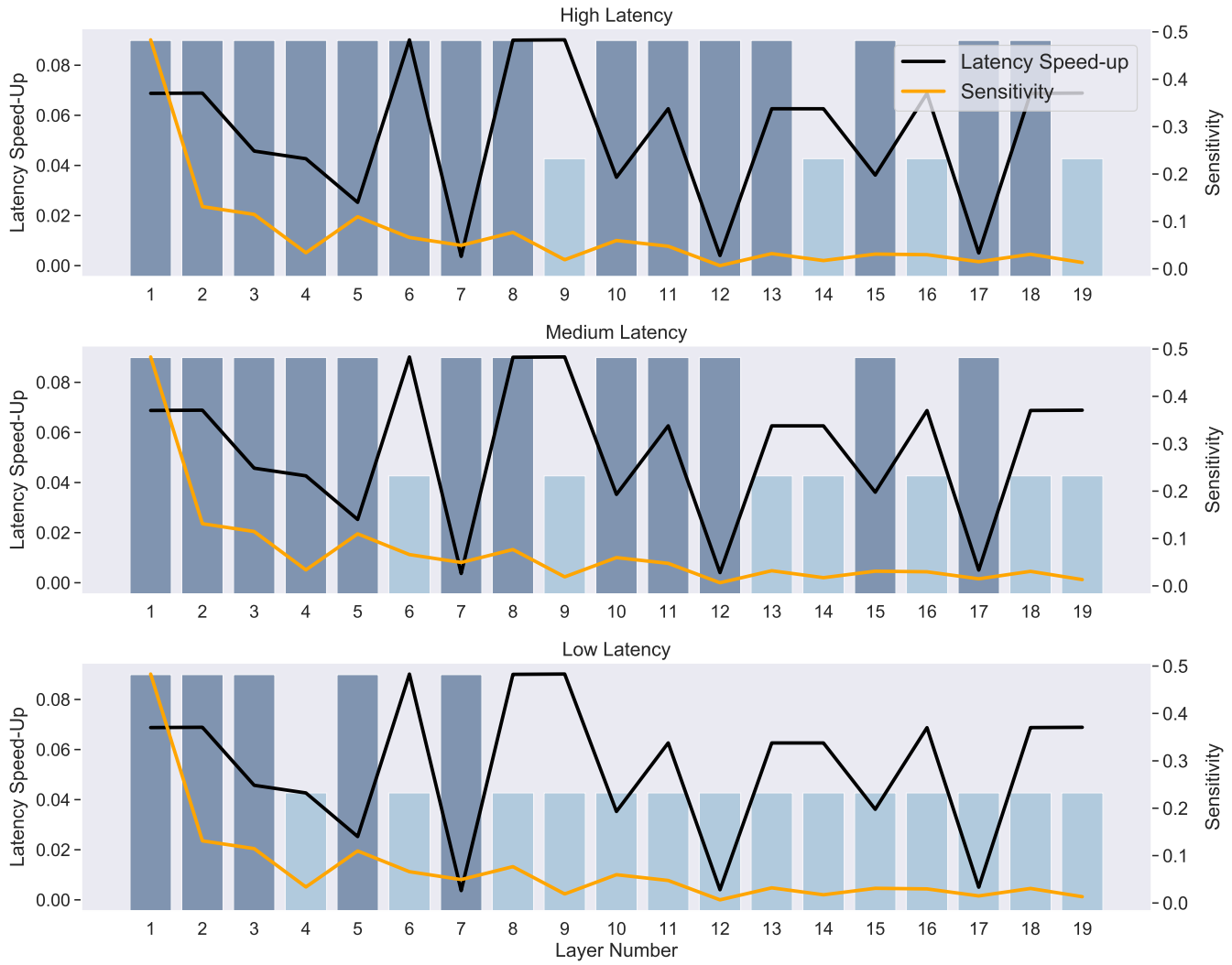

Figure 5: *Illustration of the final model specification that the ILP solver finds for ResNet18 with latency constraint. The black line shows the percentage of latency reduction for a layer executed in INT4 versus INT8, normalized by total inference reduction. Higher values means higher speedup with INT4. The orange line shows the sensitive difference between INT8 and INT4 quantization using second order Hessian sensitivity [14]. 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 IIa 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).*

observation can be made for High-BOPS v.s. High-Size for ResNet50.

We also plot the bit-precision setting for each layer of ResNet18 that the ILP solver finds for different latency constraints, as shown in Figure 5. Additionally, we also plot the sensitivity (Ω_i in Eq. 11) 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 High-Latency setting (the first row of Figure 5), only relatively insensitive layers, along with those that enjoy high INT4 speed-up, are quantized (i.e., layer 9, 14, and 19). However, for the more strict Low-Latency setting (last row of Figure 5), only very sensitive layers are kept at INT8 precision (layer 1, 2, 3,

5, and 7).[3]

Though directly using INT4 quantization may result in large accuracy degradation, we can achieve significantly improved accuracy with much faster inference as compared to INT8 result. This gives the practitioner a wider range of choices beyond just INT8 quantization. Finally, we should mention that the accuracy and speed for all of the results shown for ResNet18/50 and Inception have been verified by directly calculating them when executed in quantized precision in hardware through TVM. As such, these results are actually what the practitioner will observe, and the results are not simulated.

V. Conclusions and Future Work

The deployment of prior quantized networks on hardware with FP32 casting can have high overhead, both in terms of inference and also hardware die area. The latter is because FP32 ALUs require larger die area, resulting in limited computational peak as compared to low precision integer ALUs. In this work, we solved this problem by developing a low-precision dyadic quantization framework HAWQ-V3, where the entire inference is executed with only integer multiplication, addition, and bit shifts, i.e., with no FP32 arithmetic in the entire inference. We presented results for uniform INT4/INT8 and mixed-precision INT4/INT8. For the latter, we proposed a hardware-aware ILP based method that finds the optimal trade-off between model perturbation and application specific constraints such as model size, inference speed, and total BOPS. We found that the ILP problem can be solved within a second for all the models considered. We showed that our approach can achieve up to 5% higher accuracy as compared to prior art, and it even exceed the performance of quantization methods that use FP32 arithmetic. Finally, we directly implemented the low-precision quantized models in hardware by extending TVM to support INT4 and INT4/8 inference. We verified all the results, by matching the activation of each layer with our PyTorch framework (up to machine precision), and including the final accuracy of the model. The framework and the TVM implementation have been open sourced [1].

Acknowledgments

The UC Berkeley team acknowledges gracious support from Intel corporation, Intel VLAB team, Google Cloud, Google TFTC team, and Nvidia, as well as valuable feedback from Prof. Dave Patterson, Prof. Joseph Gonzalez, and Lianmin Zheng. Amir Gholami was supported through a gracious fund from Samsung SAIT. We are also grateful for contributions from Tianmu Lei, and Hanbing Zhan. Michael Mahoney would also like to acknowledge the UC Berkeley CLTC, ARO, IARPA, NSF, and ONR. Our conclusions do not necessarily reflect the position or the policy of our sponsors, and no official endorsement should be inferred.

[3]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.

References

[1] https://github.com/zhen-dong/hawq.git, October 2020.

[2] PuLP is an LP modeler written in Python. 2020.

[3] PyTorchCV Library, 2020.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[5] Krste Asanovic and Nelson Morgan. *Experimental determination of precision requirements for back-propagation training of artificial neural networks*. International Computer Science Institute, 1991.

[6] Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Advances in Neural Information Processing Systems*, pages 7950–7958, 2019.

[7] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. ZeroQ: A novel zero shot quantization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13169–13178, 2020.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[11] Ting-Wu Chin, Pierce I-Jen Chuang, Vikas Chandra, and Diana Marculescu. One weight bitwidth to rule them all. *arXiv preprint arXiv:2008.09916*, 2020.

[12] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. PACT: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. Ieee, 2009.

[14] Zhen Dong, Zhewei Yao, Daiyaan Arfeen, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. HAWQ-V2: Hessian aware trace-weighted quantization of neural networks. *Advances in neural information processing systems*, 2020.

[15] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. HAWQ: Hessian AWare Quantization of neural networks with mixed-precision. In *The IEEE International Conference on Computer Vision (ICCV)*, October 2019.

[16] Marat Dukhan. NNPACK, 2016.

[17] Ahmed T Elthakeb, Prannoy Pilligundla, Fatemehsadat Mireshghallah, Tarek Elgindi, Charles-Alban Deledalle, and Hadi Esmaeilzadeh. Gradient-based deep quantization of neural networks through sinusoidal adaptive regularization. *arXiv preprint arXiv:2003.00146*, 2020.

[18] Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai, Xiangyu Yue, Peter Jin, Sicheng Zhao, and Kurt Keutzer. SqueezeNext: Hardware-aware neural network design. *Workshop paper in CVPR*, 2018.

[19] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[20] Song Han and B Dally. Efficient methods and hardware for deep learning. *University Lecture*, 2017.

[21] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[23] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *Workshop paper in NIPS*, 2014.

[24] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for MobileNetV3. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1314–1324, 2019.

[25] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.

[26] Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry. Improving post training neural quantization: Layer-wise calibration and integer programming. *arXiv preprint arXiv:2006.10518*, 2020.

[27] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[28] Benoit Jacob et al. gemmlowp: a small self-contained low-precision gemm library.(2017), 2017.

[29] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[30] Animesh Jain, Shoubhik Bhattacharya, Masahiro Masuda, Vin Sharma, and Yida Wang. Efficient execution of quantized deep learning models: A compiler approach. *arXiv preprint arXiv:2006.10226*, 2020.

[31] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[32] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[33] Yann LeCun, John S Denker, and Sara A Solla.

Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[34] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[35] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *Workshop paper in CVPR*, 2017.

[36] Asit Mishra and Debbie Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. *arXiv preprint arXiv:1711.05852*, 2017.

[37] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[38] Maxim Naumov, Utku Diril, Jongsoo Park, Benjamin Ray, Jedrzej Jablonski, and Andrew Tulloch. On periodic functions as regularizers for quantization of neural networks. *arXiv preprint arXiv:1811.09862*, 2018.

[39] NVIDIA. Cutlass library, 2020.

[40] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. Value-aware quantization for training and inference of neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 580–595, 2018.

[41] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[42] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.

[43] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-Net: ImageNet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[44] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[45] Frank Seide and Amit Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135, 2016.

[46] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-BERT: Hessian based ultra low precision quantization of bert. In *AAAI*, pages 8815–8821, 2020.

[47] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[48] Mart van Baalen, Christos Louizos, Markus Nagel, Rana Ali Amjad, Ying Wang, Tijmen Blankevoort, and Max Welling. Bayesian bits: Unifying quantization and pruning. *arXiv preprint arXiv:2005.07093*, 2020.

[49] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[50] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-aware automated quantization. *In Proceedings of the IEEE conference on computer vision and pattern recognition*, 2019.

[51] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. FBNet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019.

[52] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv preprint arXiv:1812.00090*, 2018.

[53] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5687–5695, 2017.

[54] Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael W. Mahoney. PyHessian: Neural networks through the lens of the Hessian. *arXiv preprint arXiv:1912.07145*, 2019.

[55] Hongxu Yin, Pavlo Molchanov, Jose M Alvarez, Zhizhong Li, Arun Mallya, Derek Hoiem, Niraj K

Jha, and Jan Kautz. Dreaming to distill: Data-free knowledge transfer via deepinversion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8715–8724, 2020.

[56] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. LQ-Nets: Learned quantization for highly accurate and compact deep neural networks. In *The European Conference on Computer Vision (ECCV)*, September 2018.

[57] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless CNNs with low-precision weights. *International Conference on Learning Representations*, 2017.

[58] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[59] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. Adaptive quantization for deep neural network. *arXiv preprint arXiv:1712.01048*, 2017.

APPENDIX

A. Implementation Details

a) Models: All the empirical results are performed using pretrained models from PyTorchCV [3] 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 [13]. 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.

b) 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.

c) Distillation: As pointed out previously [42], 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 HAWQ-V3+DIST). Among other things, we do confirm the findings of previous work [42]. For all different models, we apply ResNet101 [22] as the teacher, and the quantized model as the student. For simplicity, we directly use the naive distillation method proposed in [23]. (More aggressive distillation or fine-tuning with hyperparameter may lead to better results.)

d) 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 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.

e) Mixed-precision configuration: For mixed-precision configuration, we first compute the trace of each layer [14] using PyHessian [54], and then solve the ILP problem using PULP [2]. 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 [50, 52]. For instance, the entire trace computation can be finished within 30 minutes for all layers of ResNet50/InceptionV3 with only 4 RTX 6000 GPUs. Afterwards, 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 [50] with 4 RTX 6000 GPUs.