



ANT: Exploiting Adaptive Numerical Data Type for Low-bit Deep Neural Network Quantization

Cong Guo^{*†1}, Chen Zhang[‡], Jingwen Leng^{*†2}, Zihan Liu^{*†}, Fan Yang[‡], Yunxin Liu[§], Minyi Guo^{*†2} and Yuhao Zhu[¶]

^{*}Shanghai Jiao Tong University, [†]Shanghai Qi Zhi Institute

{guocong, altair.liu}@sjtu.edu.cn, {leng-jw, guo-my}@cs.sjtu.edu.cn

[‡]Microsoft Research, [§]Institute for AI Industry Research (AIR), Tsinghua University
chzhang1990@gmail.com, fanyang@microsoft.com, liuyunxin@air.tsinghua.edu.cn

[¶]University of Rochester, yzhu@rochester.edu

Abstract—Quantization is a technique to reduce the computation and memory cost of DNN models, which are getting increasingly large. Existing quantization solutions use fixed-point integer or floating-point types, which have limited benefits, as both require more bits to maintain the accuracy of original models. On the other hand, variable-length quantization uses low-bit quantization for normal values and high-precision for a fraction of outlier values. Even though this line of work brings algorithmic benefits, it also introduces significant hardware overheads due to variable-length encoding and decoding.

In this work, we propose a fixed-length adaptive numerical data type called **ANT** to achieve low-bit quantization with tiny hardware overheads. Our data type **ANT** leverages two key innovations to exploit the intra-tensor and inter-tensor adaptive opportunities in DNN models. First, we propose a particular data type, **flint**, that combines the advantages of **float** and **int** for adapting to the importance of different values within a tensor. Second, we propose an adaptive framework that selects the best type for each tensor according to its distribution characteristics. We design a unified processing element architecture for **ANT** and show its ease of integration with existing DNN accelerators. Our design results in $2.8\times$ speedup and $2.5\times$ energy efficiency improvement over the state-of-the-art quantization accelerators.

I. INTRODUCTION

Deep neural networks (DNNs) have achieved great success in a variety of application domains, including computer vision [21] and natural language processing [18]. With the tensor-based computations as the dominant patterns, specialized tensor accelerators have been introduced for DNN inference [8], [14], [26], [30], [37], [68], [88] and training [46], [64], [69]. However, the size of DNN models increases by $240\times$ every two years, significantly exceeding the hardware improvement rate ($3.1\times$ every two years) [29]. For instance, the recent large Transformer-based GPT-3 [5] model has 175 billion parameters, whose single inference needs to take around 740 TOPs (tera operations).

Exploiting DNN model sparsity and redundancy through algorithm and hardware co-design is a promising way to overcome the widening computation gap between model and

hardware. There are generally two approaches, model pruning and model quantization, for reducing the computation and memory costs of DNN models. First, pruning away the unimportant elements results in sparse DNN models with reduced parameter counts. For example, NVIDIA has introduced the sparse tensor core since its Ampere architecture [64]. Second, model quantization uses narrow bit length to represent values to save memory and computation. For example, Google's first generation TPU [46] uses an 8-bit integer type for inference, while other commercial accelerators use the floating-point types with reduced precisions, such as FP16 [64], TF32 [64], and BF16 [46], for accelerating the DNN training.

All the above algorithm and hardware co-design works for DNN models leverage the traditional numerical data types such as **int** and **float**, and these types are inherently inefficient for DNN models. The reason is that the values in DNNs' tensors have both a non-uniform distribution and non-uniform importance. For instance, many weight tensors in DNNs follow the Gaussian-like distribution, with many values around zero, which, according to many DNN pruning works [40], [47], are not important and hence can be pruned. However, the **float** type has the highest resolution (called rigid resolution [52]) for these small values, wasting its bit length. On the other hand, the Gaussian-like distribution also has a long tail, whose range is critical for the DNN model accuracy [66], [86]. The **int** type needs a long bit length to represent the large values. Given the above reasons, both **int** and **float** data types usually need more bits to maintain the original model accuracy [61], [78].

To achieve even higher quantization benefits (i.e., lower bit length), prior works have proposed the outlier-aware quantization method and designed special hardware support [66], [86]. The basic idea is to employ the low precision 4-bit **int** for the small values with a high appearance frequency and high precision 32/16-bit **float** or **int** for large values with an extremely low frequency. However, this method results in variable-length encoding and hence unaligned memory access, which are incompatible with the existing DNN accelerators and require a complex and high-cost hardware design.

In this work, we present an adaptive numeric data type

¹ This work started during his internship at Microsoft Research.

² Jingwen Leng and Minyi Guo are corresponding authors of this paper.

called ANT, which can adapt to the importance of different value intervals within a tensor (i.e., intra-tensor adaptivity) and the distribution of different tensors (i.e., inter-tensor adaptivity). More importantly, ANT has aligned memory accesses and efficient low-bit computation, leading to large quantization benefits and low hardware overheads. We first propose a novel data type primitive called `flint` that combines the advantages of `float` with a large range and `int` with high precision for important value intervals. We leverage the variable-length first-one coding technique to encode the exponent field. As a result, the overall encoding has a fixed length, which is friendly for hardware decoding.

We then design a general framework that adapts to different distributions of weight and activation tensors, which include not only the above Gaussian distribution and also Laplace and uniform-like distributions. We build upon the previous works like `POT` (i.e., power of two) type [58], [94], and show how to integrate them into our ANT framework that chooses the best-fit numerical type to minimize the quantization error. All those primitive types have the fixed-length, and a tensor can only have a fixed primitive type. Compared to previous works that only exploit the intra-tensor adaptivity [66], [73], [86] or inter-tensor adaptivity [73], [78], the ANT framework can achieve both with high hardware efficiency.

We further propose a unified TypeFusion processing element (PE) design that can handle the case when the input tensor and weight tensor have different primitive types. The TypeFusion PE can be implemented on top of the original `float` or `int` multiply-accumulate (MAC) unit. The required modification is a simple type decoder, which decodes different primitive types in ANT to a unified format for computing on the underlying `float` or `int` MAC unit.

The proposed ANT framework targets to solve the problem of the low-bit (i.e., 4-bit) quantization, which could still degrade the original model accuracy. We show that ANT is compatible with mixed-precision quantization [6], [7], [23], [24], [57], [64], [66], [73], [74], [81], [86], [95]. In specific, our 4-bit ANT PE design can naturally support 8-bit `int` PE with little modification. Owing to the mixed-precision support, ANT can use the 4-bit representation for over 90% tensors and still maintain the same level of accuracy as the original full-precision models.

We describe how to integrate the above ANT PEs into existing DNN accelerator architectures such as systolic array and tensor core [46], [64]. We present several optimizations to minimize the overhead of using ANT's TypeFusion PE. In particular, we show that ANT only imposes a simple type extension for the multiply-accumulate instruction, leaving the original programming model unmodified. Our evaluation results show that the ANT-based accelerator surpasses the existing mixed-precision accelerator BitFusion [73] by 2.8× performance improvement and 2.5× energy reduction.

We make the following contributions in this paper.

- We demonstrate the opportunities for adaptive quan-

tization at both the intra-tensor level and the inter-tensor level, for which we present a unified qualitative framework to analyze the hardware overhead introduced by previous low-bit quantization works.

- We propose a composite adaptive numeric data type framework called ANT that can exploit the adaptive opportunities at both the intra- and inter-tensor levels in a hardware-friendly fashion.
- We propose a unified TypeFusion processing element (PE) design that can handle the case when the input tensor and weight tensor have different primitive types.
- We describe how to integrate the above ANT PEs to existing DNN accelerator architectures such as systolic array, which achieves the same level of accuracy as original full-precision models and significantly outperforms existing quantization accelerator under the same area.

II. BACKGROUND

This section presents relevant background on DNN quantization, which has been widely studied to reduce the memory and computation cost of DNN models.

A. Quantization Metric

Many prior studies [2], [17], [60], [89] have shown that the optimization target of quantization is to reduce the MSE (Mean Square Error) between the original model and quantized model. In the digital image processing field [45], the MSE metric is formally defined by the following equation:

$$\begin{aligned} MSE &= E[(x - \hat{x})^2] \\ &= \int (x - \hat{x})^2 p(x) dx, \end{aligned}$$

where x and $\hat{x}$ are the original and quantized value, respectively, and $p(x)$ is the probability density function.

To reduce the quantization MSE, it is natural to choose a numeric type whose quantization resolution distribution is similar to the tensor distribution [2], [17], [60], [89]. Many researchers have proposed the distribution-aware quantization to address the non-uniform distribution, e.g., the Huffman encoding [39] and outlier-aware quantization [66], [86].

B. Fixed-length Quantization

Fixed-length quantization usually uses a narrow bit length representation based on `int` type or `float` type. For example, there are works using 4-bit or 8-bit `int` types (called `int4` and `int8`, respectively) to quantize the weight tensors or activation tensors in DNN models. The `float`-based types can be represented by the following equation, with a varying exponent and mantissa filed.

$$\text{Real value} = \text{sign} \times 2^{\text{exponent} - \text{bias}} \times 1.\text{mantissa} \quad (1)$$

For example, FP16 [48] uses a 5-bit exponent and 10-bit mantissa (5E10M), while BF16 [46] and TF32 [64] use the

configuration of 8E7M and 8E10M, respectively. There are also other more aggressive numerical types as follows.

PoT type [58], [94] (power of two) can be viewed as a special format of `float` type with only the exponent field and no mantissa field. As a result, it can represent a large value range and its multiplication can be simplified to addition.

AdaptiveFloat type [78] extends the basic `float` type to reduce the quantization errors for tensors with a non-uniform distribution (e.g., Gaussian-like distribution). Its quantization framework adaptively sets the tensor-wise exponent bias to match the Gaussian-like distribution and reduce the MSE.

The quantization and dequantization function for a quantized element $\hat{w}$ can be generalized to the following equation:

$$\hat{w} = s \cdot \text{Dequant}[\text{Clamp}(\text{Quant}(\frac{w}{s}), \min, \max)] \quad (2)$$

where s is the quantization scale factor and, $\min$ and $\max$ are the lower and upper thresholds for the clipping function $\text{Clamp}(\cdot)$. The operator Quant represents the quantization function. For example, the quantization function for `int` is a simple rounding-to-nearest function. After that, the dequantization operator Dequant can decode the quantized number to the original type (e.g., 32-bit `float` or FP32), which is required for the quantization-aware training [42].

For memory-aligned quantization, we follow the common practice of per-channel weight quantization [61], which applies a separate scale factor for each output channel without additional hardware overhead. For the input activations, we use the per-tensor scale factors because the per-channel activation quantization is challenging to implement [53], [61]. These quantization granularities are widely used and supported by the DNN quantization frameworks such as TensorRT [63]. Moreover, we use the unsigned type to quantize activation tensors after ReLU efficiently [52], [66], [73], [78], as its outputs are all non-negative. However, note that our framework supports both the signed numerical types and unsigned numerical types as we describe later.

C. Mixed-precision Quantization

Mixed-precision DNN quantization method uses different numbers of bits for a given data type to represent values in DNN tensors. Many works [6], [7], [23], [24], [57], [64], [66], [73], [74], [81], [86], [95] have shown that the mixed-precision method is efficient for quantizing DNN layers that have different importance and sensitiveness for the bit length.

The most widely used approach is tensor-wise mixed-precision, such as Bit Fusion [73] and NVIDIA's latest tensor core [64]. In the tensor-wise quantization, all elements in each tensor use the same fixed-length numerical type. Thus, the tensor-wise mixed-precision is hardware-friendly without incurring much overhead, which has two kinds of implementation, i.e., temporal or spatial mixed-precision. For example, a temporal design needs four cycles to perform an

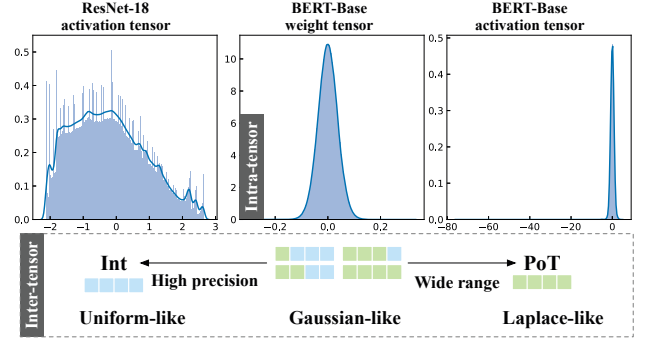


Figure 1: Intra-tensor and inter-tensor adaptivity.

8-bit $\times$ 8-bit multiplication using a 4-bit PE [76], while a spatial design needs only one cycle using four 4-bit PEs [73].

D. Outlier-aware Quantization

Outlier-aware quantization (OLAccel) [66] is tailored for Gaussian (non-uniform) distribution, which is common in DNN models. It divides the values in a tensor into two regions, i.e., outliers and non-outlier (or normal) values. The outlier with a low probability can be represented by high precision (such as FP32 or FP16), and normal values with a high probability can be compressed with fewer bits. GOBO [86] is similar to OLAccel but has fewer outliers. However, they exploit variable-length data encoding, which leads to the non-alignment in the memory sub-system. As a result, these kinds of design increase the hardware complexity and have a non-negligible area overhead as we would show later.

III. MOTIVATION: ADAPTIVE DATA TYPE

In this section, we first analyze the distributions of values in weight tensors and activation tensors from existing DNN models. Prior works have proposed the accelerator microarchitecture with adaptive bit length to achieve low-bit quantization [66], [86], which requires a significant amount of hardware resources to deal with the variable length. In contrast, our work proposes the idea of adaptive numerical data type (ANT) to fulfill the potential of low-bit quantization. We present a qualitative framework to demonstrate the advantage of ANT, which is extremely hardware-friendly.

A. Opportunities for Adaptive Type

We identify two opportunities: **the inter-tensor adaptivity** to the specific distribution of each tensor and **the intra-tensor adaptivity** to the importance of values in each tensor. These two opportunities lay the foundation for constructing the hardware-friendly adaptive quantization scheme.

We first analyze the diversity of value distributions for various tensors in popular DNN models. Most previous works focus on Gaussian-based distribution and propose different techniques to mitigate the accuracy loss of quantization [52],

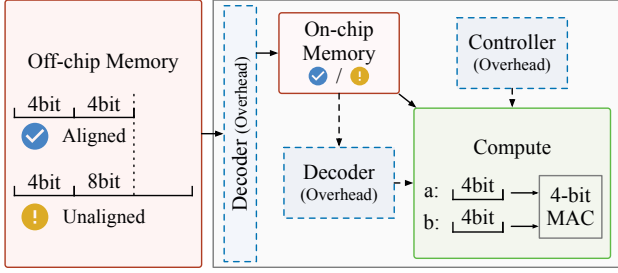


Figure 2: The qualitative framework for describing the hardware overhead for leveraging DNN quantization.

[66], [86]. However, tensors in DNN models exhibit different distributions as shown in Fig. 1. For example, the activation tensor in the first layer of ResNet18 [41] is closer to a uniform distribution, while an activation tensor in BERT [22] has a long tail that is close to Laplace distribution [2], [4].

Inter-tensor Adaptivity Given the diverse distribution of various tensors in DNN models, there naturally exists the opportunity called *inter-tensor adaptivity*. Intuitively, inspired by the non-uniform quantization of digital image processing [45], if we adaptively choose the most suitable numerical type to quantize a tensor according to its distribution, it may achieve a lower quantization error, e.g., MSE (mean square error). For instance, in the left part of Fig. 1, the four-bit `int` type would lead to a smaller quantization error than the four-bit `float` type for the uniform-like distribution with a narrow range. In contrast, the `PoT` type, which can represent a large dynamic range under the same bit length, is more suitable than `int` and `float` for the Laplace-like distribution with a long tail, as shown in the right part.

Intra-tensor Adaptivity Complementary to the inter-tensor adaptivity, the opportunity to reduce the quantization error also exists within a tensor. Specifically, extremely small and large values in a tensor do not require a high precision. First, the key premise of DNN model pruning is that [40], [47] small close-to-zero values are less important so can be pruned. Second, many quantization works have shown that large values can be clipped to a threshold [15], which, however, should be large enough. In other words, the exact numerical value of extremely large values is unimportant too as long as its rough numerical range is captured. To exploit such an intra-tensor adaptivity opportunity, the quantization scheme should allocate fewer precisions for very small and large values while capturing a large enough range.

In the next part, we show that existing works cannot exploit the aforementioned opportunities, leading to marginal quantization benefits or significant hardware overheads.

B. Quantization Architecture Analysis

We first present a unified qualitative framework in Fig. 2 to analyze the hardware overhead introduced by previous

Architecture	Off / On-chip Mem.		Compute	Overhead
	Aligned	Bit Width	Bit Width	Area Ratio
Int	✓	8	8	0
AdaFloat [78]	✓	8	8	14.5%
BitFusion [73]	✓	7.07	7.07	~ 0
BiScaled [43]	✓	6.16	6.16	7.1%**
OLAccel [66]	✗	5.81	4.36	71%
GOBO* [86]	✗	4.04 / 6.81	16	55%
ANT (Ours)	✓	4.23	4.23	0.2%

Table I: Quantization architecture comparison. We collect the average bit of once memory access and computation precision among 13 workloads, including CNN and Transformer, for each quantization method. We also count the area ratio of the decoder and controller. *GOBO only has the weight quantization, where its statistics only involve weight tensor. **We only synthesize the BPE area of BiScaled.

quantization works. We consider the three main components in the baseline DNN accelerator, which include the off-chip memory, on-chip memory, and computation unit. For example, Google’s TPU architecture [46] has the off-chip HBM, large unified on-chip buffers, and weight-stationary-based systolic array as the computation unit. On top of the baseline architecture, a quantization scheme would generally introduce three extra components, which are off-chip data decoder, on-chip data decoder, and compute controller.

We analyze five state-of-the-art quantization schemes, which are `int`, AdaptiveFloat [78], BitFusion [73], BiScaled [43], OLAcel [66], and GOBO [86]. Tbl. I compares their area overhead and quantization benefits with the metrics including averaged off-chip data width, averaged on-chip data width, and averaged compute data width. For a fair comparison, we collect the statistics from over ten models including CNNs [41], [75], [77], ViT (vision transformer) [25], and BERT [22]. We report numbers for all schemes when all models are close to their original FP32 accuracy (CNN with $< 0.1\%$ loss and Transformer with $< 1\%$ loss) except BiScaled. We take the results from the BiScaled paper [43]. Sec. VII provides more experimental details.

The conventional `int`-based quantization stores tensors with the same bit width in both off-chip and on-chip memory, making them all access aligned. Thus, it does not require any additional decoder logics and compute controller (i.e., its area overhead is zero). On the other hand, it does not exploit the inter- and intra-tensor adaptivity. A low-bit `int` can only represent a narrow range, which may clip away few but important large values [15]. As such, it often requires 8-bit `int` type to retain original model’s accuracy. Hence its quantization benefits are limited to 8 bit for off-chip memory, on-chip memory, and computation resources.

AdaptiveFloat [78] (shorted as AdaFloat) extends the

float type with a tensor-wise exponent bias. It has aligned off-chip and on-chip memory accesses but requires an exponent bias decoder for controlling the bias offset, whose area is 14.5% larger than the fixed-point (int). Although floating-point allows AdaFloat to represent a greater value range, it gradually increases quantization resolution as values' magnitude decreases logarithmically, leading to excessively high resolution (called rigid resolution [52]) for much smaller values. Because smaller values are usually less important, the rigid resolution wastes much numerical representation space, rendering its overall quantization benefits to 8 bits.

BitFusion [73] exploits the inter-tensor adaptivity by choosing different bit lengths (or precisions) for different tensors. It incurs an almost zero hardware overhead because the high precision data type (e.g., 8-bit int) can reuse all the low-precision data type (e.g., 4-bit int) components without extra overhead. However, its underlying primitive data type is still int type, which limits its quantization benefits to 7.07 memory bits and computation bits on average.

OLAccel [66] and GOBO [86] leverage the outlier-aware quantization scheme. They store a tensor using a variable-length compressed form (e.g., 4 bits for normal values and 16 bits for outliers with relatively large values) in the off-chip memory, which requires a dedicated data decoder. They also require an additional outlier controller to orchestrate the computation between normal values and outliers. Note that GOBO [86] only supports weight quantization, it requires high precision (i.e., 16-bit) floating-point computations for activation tensors. Even though both designs have a large quantization benefit with low memory bits, their associated hardware complexity and overhead are also significant.

BiScaled [43] also adopts outlier-aware quantization but with a fixed-length compressed form. However, it requires an extra bit mask for indicating different scale factors, which leads to a more considerable area overhead. Moreover, it only considers two value ranges for the intra-tensor adaptivity, leading to the benefits of 6.16 memory and computation bits.

To balance the hardware overhead and quantization benefits, we propose the adaptive numerical data type (ANT) framework that exploits both inter-tensor and intra-tensor adaptivity. ANT further supports mixed-precision. As the last row in Tbl. I shows, ANT achieves the lowest average bit for memory and computation for both activation and weight tensors with a negligible area overhead.

IV. ADAPTIVE NUMERIC DATA TYPE

In this section, we present our adaptive numeric data type ANT that can exploit both the intra- and inter-tensor adaptive opportunities in a hardware-friendly fashion. We first present a novel primitive data type called flint that combines the advantages of float and int for adapting to the importance of different values within a tensor. We then propose a general framework that adapts to each tensor's distribution by

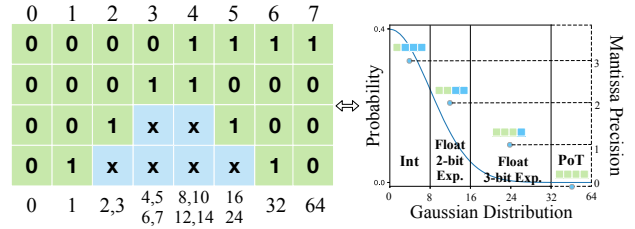


Figure 3: The 4-bit unsigned flint type, with “x” as either 0 or 1. The exponent and mantissa bits are marked with green and blue color, respectively.

selecting different primitive types, including int, float, flint, and PoT. As a result, ANT has aligned memory accesses and efficient low-bit computation, translating to significant benefits with low hardware overheads.

A. Intra-tensor ANT: Flint

Main Idea As shown in Sec. III-A, extremely small and large values in a tensor do not need high precision. A naive approach is to divide a value range into multiple intervals and assign fewer bits to intervals with small or large values. However, this leads to variable-length for different elements in the tensor and requires expensive hardware logic to handle the incurred unaligned accesses as mentioned in Sec. III-B.

To provide a fixed-length data type overcome while exploiting intra-tensor adaptivity, we propose a new primitive data type called flint. Our main idea is to start with a fixed-length, and allocate fewer mantissa bits (i.e., more exponent bits) to extremely small and large values (as they do not require a high precision) while allocating more mantissa bits (i.e., fewer exponent bits) to middle-range values to preserve their precision. Using more exponents bits for large values also allows us to capture the range of very large values.

To mark the boundary between the exponent and mantissa field, we use the first appearance of bit ‘1’ after the most significant bit (or sign bit). We call this encoding *first-one encoding*. While other strategies exist to split the exponent and mantissa fields, this encoding has the critical advantage of simplicity: the decoder for this encoding only requires a simple leading zero detector as we would show later.

An Example We use the example of four-bit flint in Fig. 3 to illustrate our design. Without loss of generality, we assume the case of unsigned values that have been scaled with the per-channel (or per-tensor) granularity for the weight (activation) tensor as described in Sec. II-B.

Fig. 3 left shows a four-bit unsigned binary number using our flint encoding, which can represent 16 distinctive binary values with the maximum value of 64. We divide this value range to eight intervals corresponding into the eight columns in Fig. 3 left, and highlight the exponent fields in green color. The first four intervals have the encoded exponent

Bits	Exponent Value	Fraction Value	Value in Decimal
0000	-	0	0
0001	$1 - 1 = 0$	1	$2^0 \times 1 = 1$
001x	$2 - 1 = 1$	1, 1.5	2, 3
01xx	$3 - 1 = 2$	1, 1.25, 1.5, 1.75	4, 5, 6, 7
11xx	$4 - 1 = 3$	1, 1.25, 1.5, 1.75	8, 10, 12, 14
101x	$5 - 1 = 4$	1, 1.5	16, 24
1001	$6 - 1 = 5$	1	32
1000	$7 - 1 = 6$	1	$2^6 \times 1 = 64$

Table II: The value table of 4-bit unsigned `flint` with the exponent bias of -1 . The blue numbers are the first-one-encoded exponent and “x” is mantissa with value of 0 or 1.

fields of 0000_2 , 0001_2 , 001_2 , and 01_2 . Under the four-bit fixed-length encoding, the number of mantissa bits for these intervals are 0, 0, 1, and 2, respectively. This mantissa bit allocation scheme is adaptive to the value importance as the first two intervals are closer to zero and hence have the least number of precisions. The exponent-mantissa bit allocation for the last four intervals is inverse to the first four intervals. In specific, the greatest interval 1000_2 has no mantissa bit, which is also desirable because the range is more important than the precision for large values.

Tbl. II shows the value table for the above 4-bit unsigned `flint` with the exponent bias of -1 . Each row refers to the divided interval (i.e., column) in Fig. 3 left. The equivalent exponent value for a given `flint` encoding is the interval number plus the bias. The final decimal value equals the fraction value multiplied by the exponent value raised by the power of two as in Equation 1. For example, the `flint` encoded number 1110_2 has the exponent value of $4 - 1 = 3$ and mantissa bit 10_2 , which corresponds to the fraction value of 1.5. As such, its decimal value is $2^3 \times 1.5 = 12_{10}$.

Essentially, our proposed `flint` is a mixture of `int`, `float` (and its variants), and `PoT` at different intervals. The first four intervals (i.e., rows) in Tbl. II have the binary encoding of 0000_2 , 0001_2 , ..., 0111_2 and represent the integer value of 0, 1, ..., 7, respectively. In other words, *the four-bit flint type is equivalent to int in the first four intervals*. The 5th and 6th intervals have the 2 and 1 mantissa bits, making them equivalent to the `float` with 2 and 1 mantissa bits (i.e., 2 and 3 exponent bits), respectively. The last two intervals have zero mantissa bits, which are equivalent to the `PoT` type.

Given these above insights, we call the proposed type `flint`, which is able to combine the advantages of `float` and `int`. Note that in the right of Fig. 3, the divided eight intervals can be coalesced into four intervals according to their type-equivalence. We can see that the mantissa precision allocation in `flint` highly matches the Gaussian distribution, meaning values with a higher frequency also with more mantissa bits. As most tensors in DNN models are Gaussian-like, we show

Algorithm 1: Element-wise `flint` encoding algorithm.

Input: Element, e ; Bit-width, b ; Scale factor, s .

Output: Quantized Element, q .

```

1 def FlintQuant( $e, b, s$ ):
2    $en = 2 \times b$ ; // Exponent number.
3    $e = \text{IntQuantization}(e, s, 0, 2^{en-2})$ ;
4   if  $e == 0$  then
5     return 0;
6   else
7      $i = \lfloor \log_2(e) \rfloor + 1$ ;
8      $exp = \text{GetExponent}(b, i)$ ; // First-one
      exponent.
9      $mb = b - \text{len}(exp)$ ; // Mantissa bit.
10     $m = \text{Round}[(e/2^{i-1} - 1) \times 2^{mb}]$ ;
11     $m = \text{Binary}(m)$ ;
12     $q = \text{Concat}(exp, m)$ ;
13  return  $q$ ;
```

later that this behavior leads to fewer quantization errors.

Flint Encoding Algorithm To recover the accuracy loss, it is generally required to perform the fine-tuning with the quantization in the training loop. As such, `flint` needs an encoding algorithm to convert the original high precision values, such as FP32, to the low precision `flint`. The software can use this encoding algorithm to mimic the `flint` behavior during fine-tuning. Meanwhile, as we target both weight and activation quantization, the encoding needs to be performed dynamically during inference, which requires a lightweight and hardware-efficient encoding algorithm.

Algo. 1 details the hardware-efficient `flint` encoding algorithm for each tensor element. First, a b -bit `flint` number has $2 \times b$ possible first-one codes for exponents (Line 2), so its value interval is $[0, 2^{2b-2}]$. We first use `int` quantization with the scale factor s to quantize the input value e to its integer value with the value range $[0, 2^{2b-2}]$ (Line 3). We then calculate its value interval index according to the interval boundary in Tbl. II (Line 7), and derive the exponent and mantissa field correspondingly (Line 8 - 12).

For example, the 4-bit unsigned `flint` type has the value range of $[0, 2^{2 \times 4 - 2} = 64]$ (Line 2). For a decimal number 11_{10} , it has been quantized by the `int` quantization with the scale factor s (Line 3). We then calculate its value interval index $i = 4$ (Line 7), for which the encoded exponent is 11_2 (Line 8). After deriving the exponent field, we know its mantissa bit-width is $mb = 4 - 2 = 2$ (Line 9), with value of $m = (11/8 - 1) \times 2^2 = 1.5_{10}$ and rounded to $m = 2_{10}$ (Line 10). This binary code of 2_{10} is 10_2 (Line 11), which is concatenated with exponent exp to get the final `flint` encoded number $q = 1110_2$ (Line 12). Note that after the above quantization process, the original value 11_{10} is now

Algorithm 2: ANT data type selection algorithm.**Input:** Tensor, T ; Candidate list of numeric types, L .**Output:** Quantization function, F_Q .

```

1 def ANT( $T, L$ ):
2    $minMSE = 10^9$ ;
3   foreach  $l \in L$  do
4      $F = \text{GetQuantFunc}(l)$ ; // Get the
        quantization method of  $l$ .
5      $m = \text{ArgminMSE}(T, F)$ ; // Search the
        minimum MSE with range
        clipping.
6     if  $m < minMSE$  then
7        $F_Q = F$ ;
8   return  $F_Q$ 

```

rounded to 12_{10} in *flint* representation.

The above *flint* encoding algorithm is an element-wise function that can be implemented efficiently in both hardware and software. The exponent and mantissa bit settings are constants when the quantization bit-width is given. The quantization of weight tensors can be done offline, while activation quantization needs hardware support. Owing to the simplicity of our encoding algorithm, we can implement it in the hardware by augmenting the hardware’s element-wise computation unit, such as the activation unit.

B. Inter-tensor ANT

To exploit the inter-tensor adaptivity and balance the hardware complexity, we propose to select the data type for a *tensor* according to its distribution. As we have shown previously, *flint* is equivalent to *int*, *float*, and *PoT* in certain value intervals. ANT is natural to support these four primitive data types for inter-tensor adaptivity.

As previously shown in the right of Fig. 1 and Fig. 3, the *int* is most suitable for the uniform-like distribution. The *float* or *PoT* are most suitable for Laplace-like distributions. The *flint* data type is most suitable for Gaussian distribution because *flint* has the highest resolution (most mantissa bits) for the values with the highest frequency. In our work, we propose an automatic algorithm to determine the data type for tensors in a trained DNN model that we describe later. Meanwhile, it is hardware-efficient to support the above four data types. Even with different data types, a tensor is stored in a fix-length format. As such, the memory accesses of ANT are aligned and hence efficient.

C. ANT-based Quantization Framework

To apply ANT for quantizing DNN models, we first need to select the specific type for a given tensor since ANT contains multiple primitive data types. After that, we then perform the fine-tuning to recover the accuracy loss. We describe the

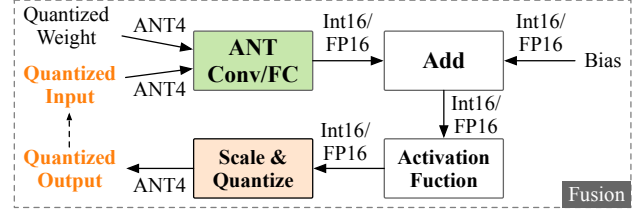


Figure 4: ANT-based quantized inference.

details in each step and explain how to use the ANT-quantized model for inference in the end.

Type Selection Algo. 2 shows the type selection algorithm for ANT, which chooses the primitive data type with minimum mean squared error (MSE) out of the candidate list L (e.g., *flint*/*int*/*float*/*PoT*). We first get the quantization function for each candidate type (Line 3-4). The *flint* encoding algorithm is described previously in Sec. IV-A, and the de-quantization algorithm can be derived by inverting the process. We use the original quantization function for the other primitive types. For a given data type, we also need to determine its range (i.e., the scaling factor). We employ a widely-used range clipping method [6], [17] that determines the clipping range by minimizing the MSE (Line 5). We then determine the most suitable data type for each tensor with minimum MSE from the candidate list (Line 6-7).

We only execute the above type selection algorithm once per tensor before fine-tuning. The reason is that the distribution of tensors of a well-trained model remains roughly similar even during the fine-tuning stage [2], which has been exploited by many other quantization methods [52], [66], [73]. For the weight tensor quantization, we do not require any training samples and directly use the weight tensors from the original, trained DNN models to determine each weight tensor’s data type. For the activation tensor quantization, we need about 100 training samples to collect the statistical information for determining the types.

Mixed Precision ANT is also compatible with the mixed-precision quantization method to achieve the same level of accuracy as the original DNN model. We leverage a layer-wise precision selection method [73]. In the beginning, we use the 4-bit ANT type for all layers and perform fine-tuning. We then collect and sort the MSE of all layers in descending order. We enlarge the bit width of a layer with the greatest MSE to 8 bits and then perform another fine-tuning. We repeat the above process until the accuracy of the quantized model is within the preset threshold of the original model.

ANT-based Inference Fig. 4 presents the ANT-based inference framework. For convolution and fully-connected layers, we use the low-bit quantized weights and input activations but keep the output activations the high precision. The reason is two-fold. First, the accumulation in these layers needs to

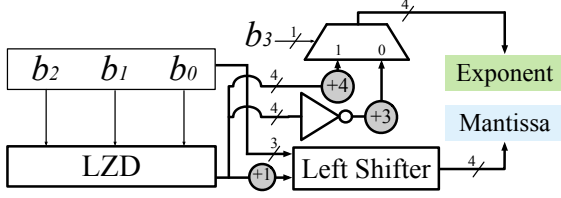


Figure 5: The 4-bit unsigned float-based flint decoder.

maintain a high precision [42]. Second, their following layers are usually activation layers such as SoftMax and GeLU, which also require high-precision numbers [87]. The output tensors of activation layers can be quantized to low-bit values, which can be completed in the hardware by augmenting the activation units (or their equivalence).

V. TYPE-FUSION PROCESSING ELEMENT

The ANT data type introduces unique challenges for the design of the processing element because the PE now needs to handle different primitive types (flint/int/float/PoT). Moreover, the input activation tensor and weight tensor for the same layer may have different data types. To address these challenges, we propose the TypeFusion processing element architecture that supports the multiply-accumulate (MAC) operation between different primitive types. We describe the two cases where we build TypeFusion PE on top of the original float-based PE and int-based PE, respectively. For convenience, we simplify the description below with the focus on unsigned numbers and it is straightforward to adapt the described design to support signed numbers.

A. Float-based PE

We first describe how to augment the original float-based MAC unit to support int, PoT, and flint. As we have described previously in Sec. IV-C, the accumulation needs to be performed in high precision, which is sufficient to cover the ranges of low-precision ANT. As such, we focus on how to augment the multiplication component.

Multiplier For the int and PoT, we can regard them as two special float formats. Int has no exponent and is full of mantissa with the subnormal number. PoT has no mantissa and is full of exponent bits with extreme dynamic range. For each type, we need to identify its exponent bit-length and mantissa bit-length and send them to exponent and mantissa decoders, respectively. Therefore, we need a float multiplier with an n -bit exponent and an n -bit mantissa for n -bit int and PoT. Meanwhile, those exponent and mantissa bits are sufficient for the n -bit flint, which is equivalent to float, int, and PoT in different value intervals.

Decoder To decode int (PoT) to float, we can set the exponent (mantissa) to zero and copy all bits to mantissa

(exponent). The decoding of flint is more complicated because its decoding is value-dependent. Thus, we design an efficient float-based flint decoder to address this issue.

The 4-bit float-based unsigned flint design is illustrated in Fig. 5, and an arbitrary n -bit flint decoder can be designed in a similar way. The decoder uses a leading-zero detector (LZD) [65] and shifters, which are well-known hardware components and both have lightweight implementations. We use the following equations to extract the exponent and mantissa field from flint type:

$$\text{Exponent} = \begin{cases} 3 - \text{LZD}(b_2b_1b_0), & b_3 = 0 \\ 4 + \text{LZD}(b_2b_1b_0), & b_3 = 1 \end{cases}, \quad (3)$$

$$\text{Mantissa} = b_2b_1b_0 \ll (\text{LZD}(b_2b_1b_0) + 1) \quad (4)$$

where the LZD is the leading zero number function and $\ll$ represents left shift. We can decode flint to the original exponent and mantissa. Finally, as shown in Tbl. II, the float decoder will continue to transfer them to real values. For example, a flint number 1110_2 is 12_{10} . Its exponent is $4 + \text{LZD}(110) = 4$. Its mantissa is $110 \ll (0 + 1) = 100_2 = 0.5_{10}$. Therefore, 1110_2 is $2^{4-1} \times 1.5 = 12_{10}$.

B. Integer-based PE

For DNN inference, it is more common to use int-based PE which is simpler and more area efficient than float-based PE. Because of the incompatibility between int and float, we remove the latter from the ANT primitive data types, which now include flint, int, and PoT. To support ANT on the integer-based PE, we first introduce a unified representation that is based on two int values and its corresponding decoder design. We then present the lightweight modification of the original int MAC to support other primitive types in ANT such as flint and PoT.

Decoder For a given integer i , we decompose it to a base integer b_i and an exponent integer e , such that $i = b_i \ll e$. Tbl. III shows such a decomposition for 4-bit flint type. When the most significant bit (MSB) of flint is 0, the base integer value matches the value in int format and the exponent is zero. When the most significant bit is 1, the base integer value matches the value of remaining bits in int format left-shifted by one, and the exponent can be derived by using the leading-zero detection function/logic.

Binary	Exponent	Base Integer	Integer Value
0xxx	0	0, 1, 2, ..., 7	0, 1, 2, ..., 7
11xx	0	8, 10, 12, 14	8, 10, 12, 14
101x	2	4, 6	$4 \ll 2 = 16$, $6 \ll 2 = 24$
1001	4	2	$2 \ll 4 = 32$
1000	6	1	$1 \ll 6 = 64$

Table III: Int-based flint 4-bit value table. The blue numbers are the first-one-encoded exponent and “x” is 0/1.

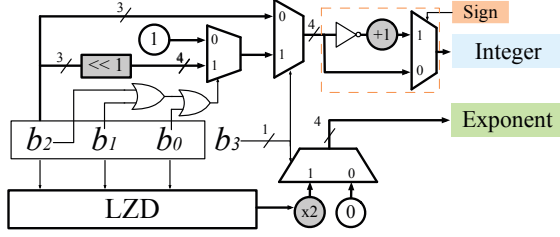


Figure 6: The 4-bit int-based flint decoder.

The following equations describe the base integer and exponent integer for a *flint* number $x = b_3b_2b_1b_0$. Fig. 6 illustrates the corresponding decoder design.

$$\text{Base Integer} = \begin{cases} b_2b_1b_0 & , b_3 = 0 \\ b_2b_1b_0 \ll 1 & , b_3 = 1 \end{cases} , \quad (5)$$

$$\text{Exponent} = \begin{cases} 0 & , b_3 = 0 \\ 2 \times \text{LZD}(b_2b_1b_0) & , b_3 = 1 \end{cases} , \quad (6)$$

This representation also works for *int* and *PoT*. The *int* type has a zero exponent value, while the *PoT* type has the base integer of one and the exponent value from its binary.

In summary, the *flint* type expands the value range if *int* type by using a simple left shifter instead of the complicated hardware logics in *float* type. It can be decoded to two integer numbers instead of a fraction number. Combined with a proper scale factor, the *flint* can be coupled with *int* quantization without extra overhead. The decoder for *flint* with an arbitrary bit-width can be generated in a similar way. The sign bit can be easily combined with the decoded numbers to fit the *int* multiplication.

Multiplier and Accumulator The decoded *flint* is not directly compatible with the original *int*-based MAC because of the extra exponent and shift operations. Therefore, we need an adder and shifter for *flint* computation shown in Fig. 7. Assume we have two *flint* numbers, f_a and f_b , with exponent e_a and e_b and base integer i_a and i_b , respectively. The integer multiplication is the same as original *int*, including the sign bit, i.e., $i_c = i_a \times i_b$. The exponent need an add operation $e_c = e_a + e_b$. Then, we get the final result $i_d = i_c \ll e_c$, which can be represented by a 16-bit *int* number. As we have explained in Sec. IV-C, low-bit *int* MAC usually adopts a high precision accumulator to achieve the precise accumulation results [42], [64]. The *flint* type produces a 16-bit *int* result and is compatible with the original 16-bit accumulator with $i_f = i_e + i_d$.

C. Signed Number Support

The *flint* decoder function and hardware design are generally extensible for arbitrary bit-width with specific constant settings. In particular, we show that the signed

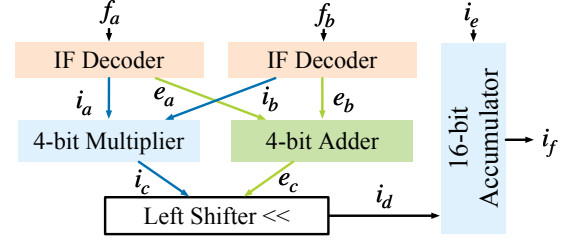


Figure 7: The 4-bit int-based flint MAC unit. “IF Decoder” is the int-based flint decoder.

number decoder can reuse most of the components in the unsigned number decoder without affecting its critical path.

For example, assume that we have a 4-bit signed *flint* number. The most significant bit b_3 is the sign, and the last three bits are $b_2b_1b_0$. For *int*-based *flint*, the following equations are the base integer and exponent decoder for 3-bit *flint*. Obviously, we can easily reuse the 4-bit unsigned *flint* decoder function shown in Equation (5) and (6).

$$\text{Base Integer} = \begin{cases} b_1b_0 & , b_2 = 0 \\ b_1b_0 \ll 1 & , b_2 = 1 \end{cases} , \quad (7)$$

$$\text{Exponent} = \begin{cases} 0 & , b_2 = 0 \\ 2 \times \text{LZD}(b_1b_0) & , b_2 = 1 \end{cases} , \quad (8)$$

To maintain the compatibility with the signed integer MAC unit, we need to convert the base integer to its two's complement form as shown in Fig. 6. However, this conversion process does not affect the critical path of the unsigned *flint* decoder as the critical path still lies in the leading zero detector unit. For the float-based *flint*, we can attach the sign bit to the decoded exponent and mantissa based on the original unsigned *float*-based decoder.

D. Mixed-precision Support

In this work, we propose to couple our ANT with the mixed-precision quantization to achieve the same accuracy of the original high-precision DNN models. According to many prior works [6], [57], [81], [95], the 8-bit *int* is sufficient to maintain the original model accuracy. We explain how our 4-bit ANT PE design can naturally support 8-bit *int* PE.

Fig. 8 shows how to use four 4-bit ANT PEs to multiply two 8-bit *int* numbers. First, we decode the two 8-bit numbers $\langle a, b \rangle$ and $\langle c, d \rangle$ to four numbers in our base integer and exponent representation, which are $\langle a, 4 \rangle$, $\langle b, 0 \rangle$, $\langle c, 4 \rangle$, and $\langle d, 0 \rangle$. Then, we perform four parallel multiplication for those four numbers, as illustrated in Fig. 8, each using a 4-bit ANT PE. Finally, we sum the results of four multiplication using an extra adder tree. In summary, our ANT PE is a good fit for supporting the mixed-precision DNN inference. In the later evaluation, we show

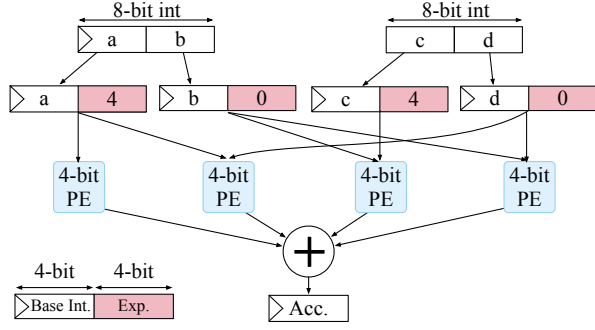


Figure 8: The 8-bit int MAC implementation via four 4-bit ANT MACs, which reuses most components except the adder.

that most tensors (up to 91%) would use 4-bit ANT while only a fraction of tensors would use 8-bit int.

VI. ARCHITECTURE INTEGRATION

In this section, we describe how to integrate the aforementioned ANT processing element into existing DNN accelerator architectures such as systolic array and tensor core [64]. We present our optimizations to minimize the overhead of using ANT’s TypeFusion PE. In the end, we describe the convenience of extending the instruction set for our design.

A. ANT and Dataflow Co-design

We first describe the architectural optimizations for applying ANT to the systolic array, which is also adopted by commercial DNN accelerators like Google’s TPU [46]. As we have explained in Sec. IV-C, our design follows the common practice in which the input and weight tensor have low-bit quantization while the output tensor has high-bit quantization. As such, we find that our design achieves the best benefits on the systolic array with the output-stationary dataflow [73]. Our evaluation results show that the weight-stationary systolic has close benefits as well. Fig. 9 depicts an output stationary systolic array with ANT decoders.

Decoder Placement We place ANT decoders between the on-chip memory buffer and systolic array. This means that quantized tensors are stored with low-bit precision in both off-chip and on-chip buffers. Meanwhile, there is no special hardware requirement for off-chip memory accesses because ANT numbers are decoded before they enter the systolic array. This design decision improves both the performance and energy efficiency because BERT-like models are bounded by the off-chip memory bandwidth [80] while CNN models spend the most energy on on-chip buffer accesses [13].

As Fig. 9 shows, only the boundary PEs in the systolic array access the on-chip buffer. As such, we only place the decoders along the boundary to mitigate the area overhead. For the output-stationary systolic array, the input and weight elements are sent to the PE (process element) array from the

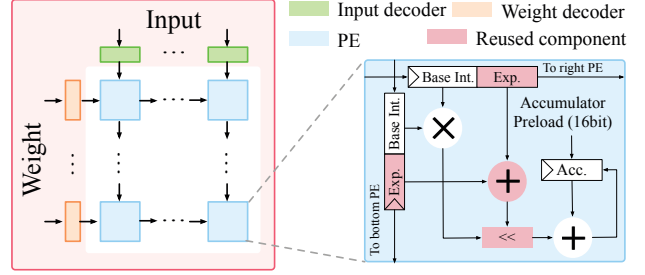


Figure 9: Architectural optimizations for integrating ANT data type to the output-stationary systolic array.

top and left, respectively. Assuming the array size of $n \times n$, we only need $2n$ instead of n^2 decoders, which amortizes the hardware area overhead of our design. The weight-stationary systolic array only needs n decoders for the input tensor as the output tensor is stored with high precision.

PE Connection To use ANT PEs in the systolic array, we need extra wires connecting neighbouring PEs. The reason is that after decoding, an n -bit ANT type has two n -bit binary numbers. For example, a float-based ANT type has an n -bit exponent number and an n -bit mantissa number, while an int-based ANT type has an n -bit exponent number and an n -bit base integer number. However, our evaluation results show that the extra overhead for those wires is negligible due to the extremely short distance between PEs.

Component Reuse The 4-bit int-based ANT MAC unit uses a 4-bit adder and shifter for adding exponent values and shifting the multiplier result, respectively. Those extra hardware overheads can be mitigated in the mixed-precision design. As we have explained in Sec. V-D, the 8-bit int MAC requires four 4-bit ANT PE and a 16-bit adder. In the 8-bit mode, the $n \times n$ systolic array with 4-bit ANT PEs would transform to $n/2 \times n/2$ systolic array with 8-bit int PEs. In this sense, we claim that ANT does not introduce new components for the PE of the mixed-precision systolic array, except for decoders outside the systolic array.

Weight Stationary Similar to output stationary, we move the input decoder to the top instead of the inner PE, as shown in Fig. 9. Because weight elements are preloaded to the PEs, the weight can be decoded before the preloading. Therefore, the weight decoders only need to decode and store the decoded exponent and integer within each PE. Other optimizations for output stationary can also be used similarly as well.

Tensor Core Tensor core already supports mixed precision. For example, the A100 GPU with Ampere architecture [64] provides 624 and 1248 TOPS (tera operations per second) for 8-bit int and 4-bit int, respectively. Meanwhile, the accumulator width for those MAC units is 32-bit int. As

such, the existing tensor core can easily adopt the ANT type by augmenting its MAC units and adding decoders for the two multiplication operands. Moreover, the tensor core-based ANT has aligned memory accesses and does not require any modification of GPUs' memory hierarchy.

B. Instruction Set Extension

The ANT framework introduces new data types for the multiply-accumulate instructions. For `int`-based ANT, we have two new data types, i.e., `PoT` and `flint`. They have the fixed bit-width so that the original load/store instructions are still applicable and hence remain unchanged. Thus, there is no modification for the memory sub-system.

Obviously, ANT does also not break the original programming model for convolutional (CONV) and fully connected (FC) layers. The specific type for each CONV and FC layer are determined after the quantization, and we can replace the original `int`-based version with `flint` or `PoT` to generate the corresponding codes. Thus, our ANT framework has a broad applicability owing to its ease of integration.

VII. EVALUATION

We evaluate ANT in the aspect of model accuracy, performance, area overhead, and energy efficiency in this section.

A. Methodology

Baselines We implement the ANT quantization framework in PyTorch [67]. We evaluate four baselines compared against ANT, including BitFusion [73], OLAcel [66], BiScaled [43], AdaFloat [78], and GOBO [86]. BitFusion [73] uses the mixed-precision of 4-bit and 8-bit `int` types. BiScaled [43] quantizes the tensors with two scale factors to address different ranges. We take the accuracy results from BiScaled paper and only synthesize the 6-bit BiScaled BPE. AdaFloat [78] requires an 8-bit `float` to maintain the original model accuracy. OLAcel and GOBO are both outlier-aware quantization. We extend OLAcel [66] to the Transformer-based models with weight & activation quantization. Note that according to the original paper, the first and last layer require 8-bit instead of 4-bit for normal

values. GOBO [86] only quantizes weights, so we only compare ANT against it in the metrics of area and accuracy.

Benchmark We use both CNN and Transformer-based models, including computer vision and natural language processing tasks listed in Tbl. IV. We exploit the SOTA checkpoint from PyTorch official repository [67]. We report the top-1 accuracies with FP32 in Tbl. IV. The evaluated CNN models with the ImageNet dataset [21] include VGG-16 [75], ResNet-18 [41], ResNet-50 [41], and Inception-V3 [77]. For Transformer-based models, we evaluate BERT-Base [22] with eight datasets of the GLUE dataset suite [79]. Owing to the space limitation, we only present the results on three datasets (MNLI, CoLA, and SST-2), while the other datasets have similar results. We also evaluate ViT (vision transformer) [25], which is a recent Transformer-based model and has achieved excellent results for vision tasks.

Fine-tuning Our ANT along with other baselines except BiScaled [43] are compatible with quantization-aware training for better accuracies. To conduct a fair comparison, we strictly set the same hyper-parameters, including number of fine-tuning epochs and learning rate, for all types. All variables use 32-bit floating-point (FP32) arithmetic operations to simulate quantization effects [42]. We generate and inject trainable weights and activation quantization parameters into the computation graph to fine-tune the quantized weights and activations. To optimize the clipping ranges (i.e., scale factors in Equation (2)), we also employ the straight-through estimator (STE) [3] method in the backward propagation based on the quantization framework PACT [15], [52].

Accelerator Implementation We implement the ANT decoder and PE described in Sec. V with the Verilog RTL. We use Synopsys Design Compiler [50] to synthesize those components with the 28 nm TSMC process, which reports area and static/dynamic power estimation. We use CACTI [59] to estimate the area, latency, and power of memory structures. For the end-to-end performance evaluation of ANT and other baselines, we develop a cycle-accurate simulator based on the DnnWeaver [72]. We use DeepScaleTool [71] to scale all designs to the 28 nm process for the iso-area comparison.

B. Quantization Accuracy

Since ANT uses multiple primitive data types (`flint/int/float/PoT`), we first study the contribution of each primitive for improving the quantization accuracy.

Primitive Combination We study six combinations of the four primitive data types. `Int` is the combination with only a single data type. Two combinations `int-PoT` (IP) and `float-int-PoT` (FIP) excludes `flint` and hence only exploit the *inter-tensor* adaptivity. Correspondingly, we evaluate these two combinations that add `flint` and exploit the *intra- and inter-tensor* adaptivity. They are `int-PoT-flint` (IP-F) and `float-int-PoT-flint` (FIP-F). ANT4-8 uses the mixed-precision of 4-bit `int`-based ANT

Type	CNN			Transformer	
Model	VGG16 [41]	Res.18 / 50 [41]	Incep.V3 [77]	ViT [25]	BERT [22]
Dataset	ImageNet [21]				GLUE [79]
Acc. (%)	73.48	69.59 / 75.97	77.34	80.99	84.42 (MNLI)

Table IV: Details of evaluated model and dataset.

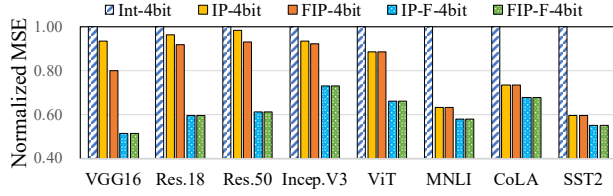


Figure 10: The quantization MSE with the combination of four different primitive types, all of which use 4-bit.

(i.e., IP-F) and 8-bit `int` for accuracy comparison. All types use 4-bit quantization except `ANT4-8`. For the quantization metrics, we use the MSE and model accuracy loss against the original FP32 model. Fig. 10 plots the quantization MSE of these combinations on eight DNN models. Fig. 11 and Fig. 12 demonstrate their accuracy loss compared to original high-precision models before and after fine-tuning, respectively.

Quantization MSE For quantizing each tensor in DNN models, we employ the ANT algorithm described in in Sec. IV-C to choose the primitive data type with minimum MSE. From the results in Fig. 10, we find that adding more primitive data types generally lets us decrease the accuracy loss owing to quantization errors. In specific, adding the `PoT` type is critical for Transformer-based models on NLP datasets (MNLI, CoLA, and SST2), since they have large activation values. The benefit of the `PoT` type is smaller for the vision tasks including ViT. Adding the `flint` type is important for both vision and NLP tasks. Finally, we observe that adding the `float` has the least impact on the quantization errors, whose role is replaced by other primitive types.

Accuracy Comparing Fig. 10, Fig. 11, and Fig. 12, we find that the model accuracy loss correlates well with the quantization MSE, and the fine-tuning plays an essential role in recovering the accuracy to the original values before quantization. The 4-bit IP-F and FIP-F provide more numerical types for selection, and both achieve the minimum accuracy loss. The former only requires the `int`-based PE while the latter requires the `float`-based PE. We show later that the `float`-based PE for ANT consumes almost $3\times$ area of `int`-based PE. As such, we choose the IP-F

Model	ANT	BiScaled	Source
AlexNet [49]	55.85%	54.90%	56.56%
VGG16	72.80%	66.56%	73.48%
ResNet50	75.08%	70.46%	75.97%
ResNet152	77.30%	73.41%	78.25%

Table V: Accuracy comparison between ANT and BiScaled without fine-tuning under 6-bit quantization.

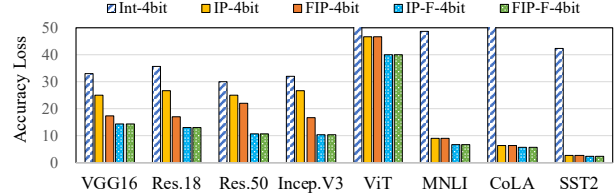


Figure 11: The accuracy loss without fine-tuning.

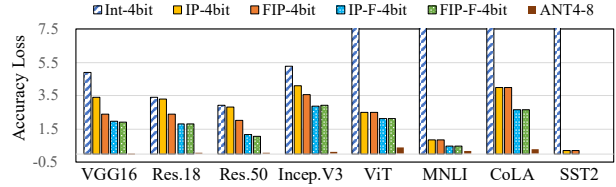


Figure 12: The accuracy loss with fine-tuning.

configuration (i.e., `int-PoT-flint`) as the final ANT for the rest of evaluation. Note that the 4-bit ANT type is still not able to maintain the original model accuracy, which justifies the choice of mixed-precision in our work. The mixed-precision `ANT4-8` type can achieve original model accuracy in CNN models and less than 1% accuracy loss for ViT and BERT. We also observe that our proposed `flint` data type is important for the accuracies of both vision and NLP tasks. Meanwhile, the `PoT` type is more important for Transformer-based models on NLP tasks than vision tasks.

Comparison against BiScaled We first compare the accuracy of IP-F configuration (i.e., `int-PoT-flint`) of ANT against the BiScaled [73] without fine-tuning. Tbl. V shows the 6-bit quantization without fine-tuning results for ANT and BiScaled. We find that ANT offers much better accuracy than BiScaled because ANT can exploit inter-tensor adaptivity and intra-tensor adaptivity with more exponent domains.

Comparison against GOBO We compare the accuracy of ANT against the prior outlier-aware quantization work GOBO [86]. Unlike ANT that performs both weight and activation quantization, GOBO only performs weight quantization. For a fair comparison, Tbl. VI shows that the weight-only quantization using ANT achieves a similar accuracy, while ANT’s fixed-length feature is more hardware-friendly than the GOBO’s variable-length encoding scheme.

Bit Width	ANT	GOBO	Source
3-bit	83.86%	83.76% (3.04 bit)	84.42%
4-bit	84.39%	84.45% (4.04 bit)	

Table VI: Accuracy comparison between weight-only quantization using ANT and GOBO for BERT on MNLI dataset.

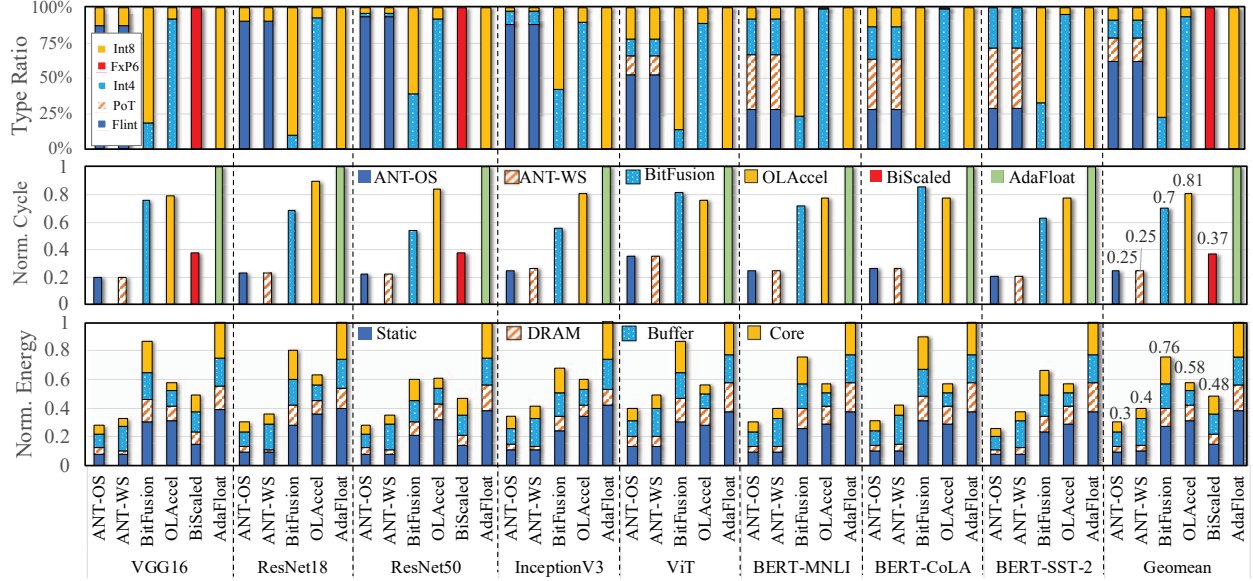


Figure 13: Comparison of the tensor type ratios (top), normalized latency (middle), and energy in different designs (bottom).

C. Area

According to our evaluation, the `float`-based PE has about $3\times$ area of `int`-based PE. Given their similar accuracies, we choose to use the `int`-based decoder and PE for ANT accelerator. We compare the accelerator area breakdown in Tbl. VII. Overall, the `int`-decoder overhead is about 0.2% for the systolic array. In the rest of evaluation, we scale other accelerators to 28 nm and perform an iso-area comparison. All accelerators have the same on-chip buffer configuration.

D. Performance and Energy

We implement ANT with output-stationary (ANT-OS) and weight-stationary (ANT-WS). We adjust the mixed-precision ratio to make all models close to their original accuracy (CNN with $< 0.1\%$ loss and Transformer with $< 1\%$ loss) for the iso-accuracy and iso-area comparison except BiScaled.

We only compare BiScaled on VGG16 and ResNet50, which have unignorable ($> 5\%$) accuracy loss, as shown in Tbl. V. Since AdaFloat [78] does not support mixed-precision, we

Architecture	Core			Buffer
	Component	Number	Area (mm^2)	
ANT	Decoder ($4.9\mu m^2$)	128	0.327	512 KB 4.2 mm^2
	4-bit PE ($79.57\mu m^2$)	4096		
BitFusion	4-bit PE	4096	0.326	
OLAcel8	4-bit & 8-bit PE	1152	0.320	
BiScaled	6-bit BPE	2560	0.328	
AdaFloat	8-bit PE	896	0.327	

Table VII: The configuration and area breakdown of ANT and other baselines under 28 nm process.

only conduct 8-bit quantization based on AdaFloat. Fig. 13 compares ANT design against various baselines with the metrics including the ratio of tensor types, normalized latency, and energy. The batch size is 64 for all experiments.

Tensor Type Ratio The top plot of Fig. 13 compares the ratio of 4-bit (`flint`, `PoT`, and `int`) and 8-bit (`int`) tensors in different designs. ANT-OS and ANT-WS use the same quantization algorithm but different microarchitectures, so that they have the same ratio of various data types. By inspecting the tensor ratio, we find that CNN models and vision transformer model ViT choose to use a significant portion of 4-bit `flint` type, while NLP Transformer models use a roughly same portion of 4-bit `flint` and `PoT` type.

Compared to the prior mixed-precision work BitFusion [73], ANT has a much greater ratio of 4-bit tensors because its inter-tensor and intra-tensor adaptivity make the 4-bit ANT achieve much lower quantization errors. Especially for BERT on SST-2, ANT can get the original accuracy with 100% 4-bit quantization. OLAcel [66] is not tensor-wise quantization as it uses variable-length encoding for different values within a tensor. We show its element-wise ratio of 4-bit and 8-bit values in the plot. Owing to its fine-grained element-wise quantization, it has a slightly higher proportion of 4-bit values than ANT, but also incurs a much greater hardware overhead with low end-to-end latency.

Performance The middle plot of Fig. 13 compares the normalized execution time of different designs, which shows that ANT achieves the best latency performance. We also find that ANT-OS and ANT-WS have very similar performances because their architectural differences can be mitigated through tiling optimizations [73]. BitFusion has more 8-bit

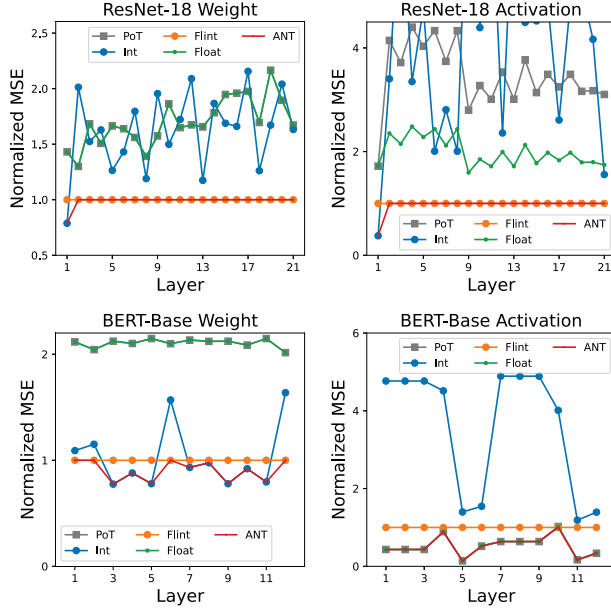


Figure 14: Numerical type (4-bit) mean square error (MSE) results that are normalized to `flint`.

tensors, which lead to its worse performance. Even though OLAcel has a higher proportion of 4-bit tensors, it needs the additional outlier controller with significant overhead to orchestrate the computation among normal values and outliers. In the end, ANT achieves averaged $2.8\times$, $3.24\times$, $1.48\times$, and $4\times$ speedup over BitFusion [73], OLAcel [66], BiScaled [73], and AdaFloat [78], respectively.

Energy The bottom plot of Fig. 13 compares the normalized energy consumption of different designs, which includes the static energy and dynamic energy (DRAM, on-chip buffer, and core). ANT-OS and ANT-WS have the lowest and second-lowest energy, respectively. Even though ANT-WS has a similar performance to ANT-OS, ANT-WS needs more buffer accesses for the high-precision output activation. Thus, ANT-WS spends more energy on accessing on-chip buffers. OLAcel consumes less energy than BitFusion because it has more 4-bit values, which reduces the energy of DRAM and on-chip buffer. In the end, ANT-OS achieves averaged $2.53\times$, $1.93\times$, $1.6\times$, and $3.33\times$ energy reduction over BitFusion, OLAcel, BiScaled, and AdaFloat, respectively.

E. ANT Type Selection Analysis

In this subsection, we study the effectiveness of the data type selection algorithm in ANT, as previously described in Sec. IV-C. Specifically, we present the MSE for all weight and activation tensors in DNN models with diverse distributions.

We collect weight and activation tensors from ResNet-18 (CNN model) on ImageNet and BERT-Base (Transformer model) on MNLI dataset. Fig. 14 shows the MSE values

of different 4-bit data types that are all normalized to `flint`. We adopt the unsigned numerical type for ResNet-18 activation tensors because of the ReLU function, which is a common practice for CNN quantizations [52], [66], [73], [78]. We use signed types for ResNet-18 weight tensors and all tensors of BERT. Fig. 14 shows that ANT *always* chooses the most appropriate data type, i.e., the type with the minimum MSE. Note that signed 4-bit `float` and `PoT` are identical so that they overlap in ResNet-18 weight MSE and BERT-Base weight and activation MSE.

We also justify the choice of data types by inspecting the distribution of different tensors. Recall that in Sec. II-A, it is natural to choose a numeric type whose quantization resolution distribution is similar to the tensor distribution to reduce the quantization error. For CNN models, `int` has pretty low MSEs with the first convolutional layer. Our manual inspection shows that the first layer is more like a uniform distribution than Gaussian. This is especially so for the activation tensor, which is the original image and not the featured map. After the first layer, tensors in CNN models are closer to Gaussian distribution so that `flint` almost dominates these layers with very low MSE values.

For BERT, we only collect the former two Transformer blocks as the representative, which have a similar trend to the rest Transformer blocks. BERT model has relatively more complex tensor distributions. The weight tensors show both uniform-like and Gaussian-like distributions so both `int` and `flint` are chosen. On the other hand, activation tensors have significant outliers so that they prefer `PoT` or `float`.

VIII. RELATED WORK

This section presents related work on DNN acceleration, sparse accelerators, and low-bit quantization accelerators.

DNN Acceleration To accelerate the DNN models efficiently, researchers proposed both various hardware and software solutions. For hardware acceleration, the proposed architectures are tailored to fit the computation characteristics of DNN models which leverage the regular access pattern, high data reuse and tremendous parallelism to save the area and latency from control logic [12], [14], [27], [28], [30], [36], [37], [51], [68], [69], [88], [97]. In these hardware accelerators, weight or weight data flow through multiple stages to maximize reuse, with example like systolic array [26], [46] and other spatial architectures [13], [62], [85], [91]. Modern GPUs have already deployed SIMD-friendly matrix-matrix multiplication (GEMM) accelerator like tensor core [64].

For software acceleration, the efforts are mainly put into the compilation and scheduling optimizations. To fully utilize the hardware resources, various automated compilers or graph optimizers are proposed to find optimal implementations on different hardware [11], [44], [92], [93], [98]. Researchers proposed various scheduling techniques [9], [10], [16], [19],

[20], [54], [55], [56], [84] to manage resource usage, task queuing, runtime batching, and so on.

Sparse DNN Accelerators Given the increasing computation demand of DNN models, it is of paramount importance to leverage the algorithm and hardware co-design. Researchers have proposed pruning and quantization methods to exploit the redundancy property of DNNs for such a purpose. Pruning means removing part of the weight, input, or even output of DNN layers, which leads to a sparse model with a portion of model size. However, sparse models contain irregular memory accesses, which could negate the benefits of sparsity. To overcome this challenge, it is important to design sparsity-optimized algorithms and hardware architectures [1], [31], [32], [33], [34], [39], [69], [70], [82], [90], [96], [99].

Quantization Accelerators The DNN model quantization exploits the insight that DNN inference does not need high-precision representations like FP32, and is orthogonal to model pruning. It uses a narrow bit width to reduce memory and computation requirement. The fixed-length value encoding is convenient for architectural integration because it only requires processing element design, such as FP16, INT8, or even INT4 [46], [64], [66], [73], [76], [86], and BF16 [46], TF32 [64], and Posit [38]. Posit is a general data type and a potential replacement for IEEE 754 [48]. It uses variable length encoding for the regime bits to extend the exponent range. Our proposed `flint` is different from Posit in the aspect that `flint` has no regime bit and an efficient encoding/decoding process based on `float` or `int` type.

BitFusion [73] and DRQ [76] can support different bit-width via a spatial and temporal combination of low-bit PEs, respectively. There are also outlier-aware quantization accelerator designs, such as OLAcel [66], DRQ [76], and GOBO [86], which are more aggressive and require heavy architectural modifications. Moreover, these outlier-aware quantization accelerators have unaligned computation and memory accesses, resulting in their limited benefits. In contrast, our work provides the adaptive numerical data type, which provides low-bit fixed-length value presentations and hence is also easy for architectural integration.

Quantization Methods In our work, we use two popular quantization methods, i.e., quantization-aware training (QAT) [37], [42], [83], [100] and post-training quantization (PTQ) [35], [37], [42], [83], [100]. The QAT requires fine-tuning to restore the model accuracy, while the latter leverages heuristics such as constraint optimization to avoid fine-tuning.

IX. CONCLUSION

In this work, we present a novel, composite data type called ANT to achieve low-bit quantization for accelerating DNN models. The key insight is adapting the data type to value importance within a tensor and different tensors' value distributions. For the intra-tensor adaptivity, we propose

`flint`, a new data type that combines the advantages of `int` (maintaining a high precision for important value ranges) and `float` (maintaining a large value range). For the inter-tensor adaptivity, we propose the composite ANT type, which selects a data type (e.g., `int/flint/PoT`) for each tensor according to its distribution. We design a unified processing element architecture for ANT and show its ease of integration to existing DNN accelerators. Our design demonstrates $2.8\times$ latency reduction and $2.5\times$ energy improvement over the state-of-the-art quantization accelerators.

ACKNOWLEDGMENT

This work was supported by the National Key R&D Program of China under Grant 2021ZD0110104, the National Natural Science Foundation of China (NSFC) grant (U21B2017, 62072297, and 61832006). The authors would like to thank the anonymous reviewers for their constructive feedback for improving the work. We also thank Tailong Wangliu, Weiming Hu, and Yuxian Qiu for their technical supports and beneficial discussions.

APPENDIX

A. Abstract

Our experiments have two major parts: the evaluation of DNN model accuracy and the performance of the ANT simulator.

We evaluate the results with models in image classification and NLP. The image classification tasks include five models, i.e., VGG16, ResNet18, ResNet50, Inception-V3, and ViT. We adopt the BERT model for the NLP task with three datasets, MNLI, CoLA, and SST-2. We provide the fine-tuning source code for all models to measure the accuracy. However, that may need dozens of hours to complete the fine-tuning process. Therefore, we provide the checkpoints for the fast evaluation of image classification models, which can finish in one hour. For measuring the performance, we evaluate all models with six simulator configurations. In all experiments, we run those models according to the experiment setup on a Ubuntu server that equips an NVIDIA A100 GPU and multiple servers with four NVIDIA A10 GPUs for distributed fine-tuning.

B. Artifact check-list (meta-information)

- **Compilation:** NVCC 11.3, GCC 7.5.0.
- **Model:** VGG-16, ResNet-18, ResNet-50, Inception-V3, ViT, and BERT-Base.
- **Data set:** ImageNet, and GLUE dataset.
- **Run-time environment:** Ubuntu 18.04.5 LTS, CUDA 11.3, and PyTorch 1.11.
- **Hardware:** A server with an x86 processor, an NVIDIA A100 GPU, and a server with four NVIDIA A10 GPUs.
- **Output:** Model accuracy, simulator energy, and performance.
- **How much disk space required (approximately)?:** 20GB.
- **How much time is needed to prepare workflow (approximately)?:** It takes about 30 minutes to prepare the environment.

- **How much time is needed to complete experiments (approximately)?:** It takes approximately 50 hours to execute all experiments using the server equipped with GPUs. The fast evaluation can take only one hour with the checkpoints.
- **Publicly available:** Our framework is publicly available on GitHub https://github.com/clevercool/ANT_Micro22.
- **Code licenses:** Apache-2.0 license.
- **Data licenses:** The datasets are publicly available through their original licensing terms.
- **Archived:** <https://doi.org/10.5281/zenodo.7002114>.

C. Description

1) *How to access:* We archive the source code at <https://doi.org/10.5281/zenodo.7002114>. We recommend you access our GitHub repository: https://github.com/clevercool/ANT_Micro22 for the latest version.

2) *Hardware dependencies:* We fine-tune the DNN models with two types of server configuration: A server is equipped with a single NVIDIA A100 (40GB) GPU, and a server is equipped with four NVIDIA A10 (24GB) GPUs for distributed fine-tuning.

3) *Software dependencies:* The experiments rely on the following software components.

- Ubuntu 18.04.5 LTS
- Python 3.8
- PyTorch 1.11
- Andconda 4.10.1
- GCC 7.5.0
- CUDA 11.3
- Cacti 7.0

4) *Data sets and models:* The evaluated image classification models with the ImageNet dataset [21] include VGG-16 [75], ResNet-18 [41], ResNet-50 [41], Inception-V3 [77], and ViT (vision transformer) [25]. For NLP models, we evaluate BERT-Base [22] with the GLUE dataset suite [79]. Owing to the space limitation, we only present the results on three datasets (MNLI, CoLA, and SST-2).

D. Installation

We have well-documented README files to detail the installation instruction for each experiment at https://github.com/clevercool/ANT_Micro22.

E. Evaluation and expected results

Our experiments have two major parts: the evaluation of DNN model accuracy and the performance of the ANT simulator.

- The directory `ant_quantization` contains the ANT framework based on PyTorch for the DNN model accuracy evaluation.
- The directory `ant_simulator` contains the performance and energy evaluation of the ANT simulator.

To evaluate the experiments, you can utilize the scripts in each directory according to the README files. We also release all expected results in the README files for Figure 12, Figure 13, and Table V.

F. Methodology

Submission, reviewing and badging methodology:

- <https://www.acm.org/publications/policies/artifact-review-badging>
- <http://cTuning.org/ae/submission-20201122.html>
- <http://cTuning.org/ae/reviewing-20201122.html>

REFERENCES

- [1] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, “Cnvlutin: Ineffectual-neuron-free deep neural network computing,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 1–13, 2016.
- [2] R. Banner, Y. Nahshan, and D. Soudry, “Post training 4-bit quantization of convolutional networks for rapid-deployment,” *Advances in Neural Information Processing Systems*, vol. 32, 2019.
- [3] Y. Bengio, N. Léonard, and A. Courville, “Estimating or propagating gradients through stochastic neurons for conditional computation,” *arXiv preprint arXiv:1308.3432*, 2013.
- [4] A. Bhandare, V. Sripathi, D. Karkada, V. Menon, S. Choi, K. Datta, and V. Saletore, “Efficient 8-bit quantization of transformer neural machine language translation model,” *arXiv preprint arXiv:1906.00532*, 2019.
- [5] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [6] Y. Cai, Z. Yao, Z. Dong, A. Gholami, M. W. Mahoney, and K. Keutzer, “Zeroq: A novel zero shot quantization framework,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 13 169–13 178.
- [7] Z. Cai and N. Vasconcelos, “Rethinking differentiable search for mixed-precision neural networks,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 2349–2358.
- [8] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, “A dynamically configurable coprocessor for convolutional neural networks,” in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 247–257.
- [9] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, “Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi’an, China, April 8-12, 2017*. ACM, 2017, pp. 17–32.

- [10] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*. ACM, 2016, pp. 681–696.
- [11] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: an automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 2018, pp. 578–594.
- [12] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.
- [13] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.
- [14] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 609–622.
- [15] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," *arXiv preprint arXiv:1805.06085*, 2018.
- [16] Y. Choi, Y. Kim, and M. Rhu, "Lazy batching: An slaware batching system for cloud machine learning inference," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 2021, pp. 493–506.
- [17] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev, "Low-bit quantization of neural networks for efficient inference," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE, 2019, pp. 3009–3018.
- [18] K. Chowdhary, "Natural language processing," *Fundamentals of artificial intelligence*, pp. 603–649, 2020.
- [19] W. Cui, M. Wei, Q. Chen, X. Tang, J. Leng, L. Li, and M. Guo, "Ebird: Elastic batch for improving responsiveness and throughput of deep learning services," in *37th IEEE International Conference on Computer Design, ICCD 2019, Abu Dhabi, United Arab Emirates, November 17-20, 2019*. IEEE, 2019, pp. 497–505. [Online]. Available: <https://doi.org/10.1109/ICCD46524.2019.00075>
- [20] W. Cui, H. Zhao, Q. Chen, H. Wei, Z. Li, D. Zeng, C. Li, and M. Guo, "DVABatch: Diversity-aware Multi-Entry Multi-Exit batching for efficient processing of DNN services on GPUs," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 183–198.
- [21] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.
- [22] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
- [23] Z. Dong, Z. Yao, D. Arfeen, A. Gholami, M. W. Mahoney, and K. Keutzer, "Hawq-v2: Hessian aware trace-weighted quantization of neural networks," *Advances in neural information processing systems*, vol. 33, pp. 18 518–18 529, 2020.
- [24] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "Hawq: Hessian aware quantization of neural networks with mixed-precision," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 293–302.
- [25] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.
- [26] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 92–104.
- [27] Y. Gan, Y. Qiu, L. Chen, J. Leng, and Y. Zhu, "Low-latency proactive continuous vision," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 329–342.
- [28] Y. Gan, Y. Qiu, J. Leng, M. Guo, and Y. Zhu, "Ptolemy: Architecture support for robust deep learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 241–255.
- [29] A. Gholami, Z. Yao, S. Kim, M. W. Mahoney, and K. Keutzer, "Ai and memory wall," *RiseLab Medium Post*, 2021.
- [30] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2014, pp. 682–687.
- [31] Y. Guan, J. Leng, C. Li, Q. Chen, and M. Guo, "How far does bert look at: Distance-based clustering and analysis of bert 's attention," *arXiv preprint arXiv:2011.00943*, 2020.
- [32] Y. Guan, Z. Li, J. Leng, Z. Lin, and M. Guo, "Transkimmer: Transformer learns to layer-wise skim," *arXiv preprint arXiv:2205.07324*, 2022.
- [33] Y. Guan, Z. Li, Z. Lin, Y. Zhu, J. Leng, and M. Guo, "Block-skim: Efficient question answering for transformer," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 10, 2022, pp. 10 710–10 719.

- [34] C. Guo, B. Y. Hsueh, J. Leng, Y. Qiu, Y. Guan, Z. Wang, X. Jia, X. Li, M. Guo, and Y. Zhu, "Accelerating sparse dnn models without hardware-support via tile-wise sparsity," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.
- [35] C. Guo, Y. Qiu, J. Leng, X. Gao, C. Zhang, Y. Liu, F. Yang, Y. Zhu, and M. Guo, "SQuant: On-the-fly data-free quantization via diagonal hessian approximation," in *International Conference on Learning Representations*, 2022. [Online]. Available: <https://openreview.net/forum?id=JXhROKNZzOc>
- [36] C. Guo, Y. Zhou, J. Leng, Y. Zhu, Z. Du, Q. Chen, C. Li, B. Yao, and M. Guo, "Balancing Efficiency and Flexibility for DNN Acceleration via Temporal GPU-Systolic Array Integration," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.
- [37] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International conference on machine learning*. PMLR, 2015, pp. 1737–1746.
- [38] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing frontiers and innovations*, vol. 4, no. 2, pp. 71–86, 2017.
- [39] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.
- [40] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *arXiv preprint arXiv:1506.02626*, 2015.
- [41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [42] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. 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*, 2018, pp. 2704–2713.
- [43] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, K. Gopalakrishnan, and L. Chang, "Biscaled-dnn: Quantizing long-tailed datastructures with two scale factors for deep neural networks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.
- [44] Z. Jia, O. Padon, J. J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2019, pp. 47–62.
- [45] M. A. Joshi, *Digital image processing: An algorithmic approach*. PHI Learning Pvt. Ltd., 2018.
- [46] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.
- [47] S. Jung, C. Son, S. Lee, J. Son, J.-J. Han, Y. Kwak, S. J. Hwang, and C. Choi, "Learning to quantize deep networks by optimizing quantization intervals with task loss," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4350–4359.
- [48] W. Kahan, "Ieee standard 754 for binary floating-point arithmetic," *Lecture Notes on the Status of IEEE*, vol. 754, no. 94720-1776, p. 11, 1996.
- [49] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.
- [50] P. Kurup and T. Abbasi, *Logic synthesis using Synopsys®*. Springer Science & Business Media, 2012.
- [51] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, Q. Chen, M. Guo, and V. J. Reddi, "Asymmetric resilience: Exploiting task-level idempotency for transient error recovery in accelerator-based systems," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 44–57.
- [52] Y. Li, X. Dong, and W. Wang, "Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks," *arXiv preprint arXiv:1909.13144*, 2019.
- [53] Y. Li, M. Shen, J. Ma, Y. Ren, M. Zhao, Q. Zhang, R. Gong, F. Yu, and J. Yan, "Mqbench: Towards reproducible and deployable model quantization benchmark," *arXiv preprint arXiv:2111.03759*, 2021.
- [54] Z. Liu, J. Leng, Z. Zhang, Q. Chen, C. Li, and M. Guo, "VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 388–401. [Online]. Available: <https://doi.org/10.1145/3503222.3507752>
- [55] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.
- [56] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.
- [57] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *International Conference on Learning Representations*, 2018.

- [58] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.
- [59] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.
- [60] M. Nagel, R. A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort, "Up or down? adaptive rounding for post-training quantization," in *International Conference on Machine Learning*. PMLR, 2020, pp. 7197–7206.
- [61] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," *arXiv preprint arXiv:2106.08295*, 2021.
- [62] A. V. Nori, R. Bera, S. Balachandran, J. Rakshit, O. J. Omer, A. Abuhatzera, B. Kuttanna, and S. Subramoney, "Reduct: Keep it close, keep it cool!: Efficient scaling of dnn inference on multi-core cpus with near-cache compute," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 167–180.
- [63] NVIDIA, "Tensorrt: A c++ library for high performance inference on nvidia gpus and deep learning accelerators," <https://github.com/NVIDIA/TensorRT>, accessed: 2021-04-27.
- [64] Nvidia, "Nvidia a100 tensor core architecture," in *Technical report*. NVIDIA, 2020.
- [65] V. G. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 1, pp. 124–128, 1994.
- [66] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 688–698.
- [67] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.
- [68] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013, pp. 13–19.
- [69] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 58–70.
- [70] Y. Qiu, J. Leng, C. Guo, Q. Chen, C. Li, M. Guo, and Y. Zhu, "Adversarial defense through network profiling based path extraction," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.
- [71] S. Sarangi and B. Baas, "Deepscaletool: A tool for the accurate estimation of technology scaling in the deep-submicron era," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2021, pp. 1–5.
- [72] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.
- [73] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 764–775.
- [74] S. Shen, Z. Dong, J. Ye, L. Ma, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "Q-bert: Hessian based ultra low precision quantization of bert," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, 2020, pp. 8815–8821.
- [75] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
- [76] Z. Song, B. Fu, F. Wu, Z. Jiang, L. Jiang, N. Jing, and X. Liang, "Drq: dynamic region-based quantization for deep neural network acceleration," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 1010–1021.
- [77] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.
- [78] T. Tambe, E.-Y. Yang, Z. Wan, Y. Deng, V. J. Reddi, A. Rush, D. Brooks, and G.-Y. Wei, "Algorithm-hardware co-design of adaptive floating-point encodings for resilient deep learning inference," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.
- [79] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.
- [80] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 97–110.
- [81] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Hq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 8612–8620.
- [82] Y. Wang, C. Zhang, Z. Xie, C. Guo, Y. Liu, and J. Leng, "Dual-side sparse tensor core," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1083–1095.

- [83] Z. Wang, J. Lu, C. Tao, J. Zhou, and Q. Tian, "Learning channel-wise interactions for binary convolutional neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 568–577.
- [84] H. Yang, A. D. Breslow, J. Mars, and L. Tang, "Bubble-flux: precise online qos management for increased utilization in warehouse scale computers," in *The 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.
- [85] G. Yuan, P. Behnam, Z. Li, A. Shafiee, S. Lin, X. Ma, H. Liu, X. Qian, M. N. Bojnordi, Y. Wang *et al.*, "Forms: fine-grained polarized reram-based in-situ computation for mixed-signal dnn accelerator," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 265–278.
- [86] A. H. Zadeh, I. Edo, O. M. Awad, and A. Moshovos, "Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 811–824.
- [87] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8bert: Quantized 8bit bert," in *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*. IEEE, 2019, pp. 36–39.
- [88] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.
- [89] D. Zhang, J. Yang, D. Ye, and G. Hua, "Lq-nets: Learned quantization for highly accurate and compact deep neural networks," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 365–382.
- [90] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.
- [91] Y. Zhao, Z. Du, Q. Guo, S. Liu, L. Li, Z. Xu, T. Chen, and Y. Chen, "Cambricon-f: machine learning computers with
- fractal von neumann architecture," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 788–801.
- [92] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
- [94] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *arXiv preprint arXiv:1702.03044*, 2017.
- [95] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefanet: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.
- [96] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 15–28.
- [97] Y. Zhou, M. Yang, C. Guo, J. Leng, Y. Liang, Q. Chen, M. Guo, and Y. Zhu, "Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021, pp. 214–225.
- [98] H. Zhu, R. Wu, Y. Diao, S. Ke, H. Li, C. Zhang, J. Xue, L. Ma, Y. Xia, W. Cui, F. Yang, M. Yang, L. Zhou, A. Cidon, and G. Pekhimenko, "ROLLER: Fast and efficient tensor compilation for deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 233–248.
- [99] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 359–371.
- [100] B. Zhuang, M. Tan, J. Liu, L. Liu, I. Reid, and C. Shen, "Effective training of convolutional neural networks with low-bitwidth weights and activations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.