TAB: Unified and Optimized Ternary, Binary, and Mixed-precision Neural Network Inference on the Edge

SHIEN ZHU, LUAN H. K. DUONG, and WEICHEN LIU, Nanyang Technological University

Ternary Neural Networks (TNNs) and mixed-precision Ternary Binary Networks (TBNs) have demonstrated higher accuracy compared to Binary Neural Networks (BNNs) while providing fast, low-power, and memory-efficient inference. Related works have improved the accuracy of TNNs and TBNs, but overlooked their optimizations on CPU and GPU platforms. First, there is no unified encoding for the binary and ternary values in TNNs and TBNs. Second, existing works store the 2-bit quantized data sequentially in 32/64-bit integers, resulting in bit-extraction overhead. Last, adopting standard 2-bit multiplications for ternary values leads to a complex computation pipeline, and efficient mixed-precision multiplication between ternary and binary values is unavailable.

In this article, we propose TAB as a unified and optimized inference method for ternary, binary, and mixed-precision neural networks. TAB includes unified value representation, efficient data storage scheme and novel bitwise dot product pipelines on CPU/GPU platforms. We adopt signed integers for consistent value representation across binary and ternary values. We introduce a bitwidth-last data format that stores the first and second bits of the ternary values separately to remove the bit extraction overhead. We design the ternary and binary bitwise dot product pipelines based on Gated-XOR using up to 40% fewer operations than **State-Of-The-Art (SOTA)** methods.

Theoretical speedup analysis shows that our proposed TAB-TNN is $2.3\times$ fast as the SOTA ternary method RTN, $9.8\times$ fast as 8-bit integer quantization (INT8), and $39.4\times$ fast as 32-bit full-precision convolution (FP32). Experiment results on CPU and GPU platforms show that our TAB-TNN has achieved up to $34.6\times$ speedup and $16\times$ storage size reduction compared with FP32 layers. TBN, Binary-activation Ternary-weight Network (BTN), and BNN in TAB are up to $40.7\times$, $56.2\times$, and $72.2\times$ as fast as FP32. TAB-TNN is up to 70.1% faster and 12.8% more power-efficient than RTN on Darknet-19 while keeping the same accuracy. TAB is open source as a PyTorch Extension¹ for easy integration with existing CNN models.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Theory of computation** → *Design and analysis of algorithms*; • **Computer systems organization** → **Embedded software**;

Additional Key Words and Phrases: Ternary neural networks, binary neural networks, edge computing

¹<https://github.com/ntuliuteam/TAB>.

This work is partially supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE2019-T2-1-071) and Tier 1 (MOE2019-T1-001-072), and partially supported by Nanyang Technological University, Singapore, under its NAP (M4082282) and SUG (M4082087).

Authors' addresses: S. Zhu, L. H. K. Duong, and W. Liu (corresponding author), School of Computer Science and Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore, 639798; emails: shien001@e.ntu.edu.sg; {kduong, liu}@ntu.edu.sg.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Association for Computing Machinery.

1539-9087/2022/10-ART50 \$15.00

<https://doi.org/10.1145/3508390>

ACM Reference format:

Shien Zhu, Luan H. K. Duong, and Weichen Liu. 2022. TAB: Unified and Optimized Ternary, Binary, and Mixed-precision Neural Network Inference on the Edge. *ACM Trans. Embedd. Comput. Syst.* 21, 5, Article 50 (October 2022), 26 pages.
<https://doi.org/10.1145/3508390>

1 INTRODUCTION

Object Detection [40], Speech Recognition [5], Machine Translation [11] and many other **Artificial Intelligence (AI)** applications have been widely deployed on the edge and bring convenience to people's daily life. The deep **Convolutional Neural Networks (CNNs)** for those AI applications usually have a large number of parameters and **floating-point operations (FLOPS)**, so they are trained on the cloud data centers with powerful CPUs and GPUs [31]. But the edge devices such as cellphones, smart speakers, and smartwatches have small memory and storage space and less powerful CPUs and GPUs. So the CNN models for edge devices need special optimization methods to obtain high accuracy and low latency inference with less storage and computation cost [27].

Utilizing low-precision numbers to represent the activations and weights of CNNs, quantization is an efficient way to reduce the memory and storage usage and increase the inference speed of CNNs on edge devices [12, 13]. **Binary Neural Networks (BNNs)** [36], **Ternary Neural Networks (TNNs)** [3], mixed-precision **Ternary-activation Binary-weight Networks (TBNs)** [43], and **8-bit integer quantization (INT8)** [52] are representative quantization methods that take advantage of the low-bitwidth and the low-latency operations of low-precision numbers. For example, TNNs [3] quantize the activations and weights to $\{+1, 0, -1\}$ and achieve $16\times$ smaller model size by utilizing 2-bit numbers instead of 32-bit full-precision numbers. Networks quantized as TNNs and TBNs have much higher accuracy compared with BNNs. As Table 1 shows, ResNet-18 quantized as TNN and TBN have 7.6% and 3.9% higher absolute Top-1 accuracy on ImageNet than binarized ResNet-18, respectively.

The performance benefits of quantized neural networks on general-purpose CPU/GPU platforms can only be obtained by dedicated implementation and optimization. Efficient INT8 networks have been implemented by deep learning frameworks and libraries, including TensorFlow Lite, PyTorch QNNPACK [30], and Nvidia CuDNN, and we have obtained near $4\times$ speedup of INT8 on existing CPU/GPU platforms [38, 39]. The acceleration of BNNs has been researched by BMXNet [6, 46], BitFlow [15], daBNN [48], and XOR-Net [53], and we also have obtained more than $55\times$ speedup of BNNs on general-purpose platforms.

However, the performance of TNNs and TBNs on general-purpose platforms is overlooked. Related works on TNNs and mixed-precision networks [3, 7, 9, 23, 24] focus on improving the accuracy, and the absolute Top-1 accuracy on ImageNet of ternarized ResNet-18 has increased more than 4.0% in recent years. But current TNNs and TBNs can only run in full-precision or integer mode, which means they cannot achieve the $16\times$ theoretical speedup.

Ternary, binary, and mixed-precision neural networks have four types in total: TNNs, TBNs, Binary-activation Ternary-weight Networks (BTNs), and BNNs. Taking TNN as an example, the ternary convolution consists of three steps, namely, quantization, bit-packing, and bitwise convolution, as Figure 1 shows. First, we quantize the input activations and weights into 2-bit ternary numbers. Then, we pack the 2-bit numbers into 64-bit integers for high bit-level data parallelism. One operation on the packed data equals 64 operations on the quantized data, so bit-packing is why quantized networks can get high speedup compared with unquantized networks. Third, we perform bitwise convolution on the packed activations and weights. The bitwise convolution may be further decomposed to one **image-to-column (Img2Col)** operation followed by bitwise **General**

Table 1. Effectiveness of Quantization Methods with Accuracy of Quantized ResNet-18 on ImageNet

Type	Method	Activation/Weight	Storage Saving	Theoretical Speedup	Top-1 Acc.	Top-5 Acc.
FP32	Original	32-bit Float Point	1×	1×	69.3%	89.2%
INT8	DA-INT8 [49]	8-bit Integer	4×	4×	70.2%	N.A.
TNN	TRQ [23]	Ternary (2-bit)	16×	16×	65.7%	85.9%
TBN	LS-BQNN [33]	Ternary/Binary	32×	>16×	62.0%	83.6%
BNN	CI-BCNN [44]	Binary (1-bit)	32×	64×	58.1%	80.0%

Fig. 1. Workflow of quantized convolution.

Table 2. Comparison of 2-bit and 1-bit Encoding Schemes in Existing Works

Value	2-bit Encoding					1-bit Encoding		
	Unsigned Int	DoReFa	Signed Int	FATNN	RTN	Signed Int	daBNN	DoReFa
-1			11	00	10	1	1	0
0	00	00	00	01/10	00/01			
+1	01	01	01	11	11	0	0	1
+2	10	10						
+3	11	11						

Matrix Multiplication (GEMM) to take advantage of existing optimizations on GEMM operators to get a high speedup. The dot product between vectors is the main/only operation inside convolution and GEMM, so we will take the dot product to illustrate the optimization of the ternary and binary convolution.

The acceleration of TNNs and mixed-precision TBNs and BTNs on general-purpose platforms has several research problems. First, there are no unified 2-bit and 1-bit encoding schemes for these neural networks' quantized ternary and binary values. DoReFa-Net [51] proposes a unified training and inference method for quantized neural networks with any bitwidth. But DoReFa-Net is designed for unsigned integers whose 2-bit encoding {00, 01, 10, 11} represents {0, 1, 2, 3}, as Table 2 shows, which is not applicable for TNNs that use {-1, 0, +1} as the quantized weights and activations. Current research works on TNNs adopt arbitrary and complex encoding schemes. For example, FATNN [7] designs dedicated quantization and convolution to provide fast and accurate TNN, and it encodes {-1, 0, +1} as {00, 01/10, 11}. RTN [24] re-parameterizes the ternarized activations with scale and offset to improve the accuracy. RTN also proposes dedicated ternary encoding and ternary dot product to speedup TNNs, which encodes {-1, 0, +1} as {10, 00/01, 11}. These special ternary encoding schemes are different from each other without compatible binary encoding, making it a great challenge to do computation between ternary and binary values in mixed-precision networks.

Second, the current data storage schemes for quantized activations and weights are not optimized for CPU and GPU platforms. The most straightforward and widely adopted data storage

Fig. 2. Task graphs of the dot product of DoReFa-Net 2-bit on CPU. Rectangles stand for data, and yellow circles stand for operations. (a) The dot product in traditional data format. It repeats the whole process on the vector and needs more Shift and AND operations. (b) The dot product in optimized data format. It can directly perform the AND operations between the first and second bits.

scheme is storing the 1-bit/2-bit values in 32-bit/64-bit integers sequentially [15, 24]. This storage scheme works for 1-bit values but brings great overhead to the computations on packed 2-bit values. For example, the 2-bit dot product of DoReFa-Net [51] needs to conduct AND operation between each bit of the 2-bit values, as Figure 2 shows. However, the 2-bit values are packed in 4-bit or even 64-bit integers, so it needs to extract the first bit and the second bit by shifting and masking (an AND operation with a constant mask), which is referred to as the bit-extraction overhead. The current storage scheme is inefficient, because it brings five extra Boolean operations to the dot product compared with the optimized data format.

Third, exiting ternary dot product schemes are not efficient, and the ternary-binary mixed-precision computation on general-purpose platforms is a blank research area to the best of our knowledge. The encoding space of two bits is four, but ternary values only need three, as shown in Table 2. So, standard 2-bit multiplication, which covers the whole encoding space, is functionally correct but too complex for the ternary dot product in TNNs. The dot product in FATNN [7] may produce error results, and it adopts a specially designed quantization algorithm and dynamic masking to prevent the error, bringing great extra computation overhead. RTN [24] implements their proposed ternary dot product on FPGA with flexible bit-manipulation support from the hardware. But RTN incurs bit-extraction overhead on CPU and GPU platforms, which have limited and fixed instructions for bitwise operations. The 1-bit/2-bit mixed-precision computation has been researched by DoReFa-Net [51] and implemented by bit-serial convolution in TVM [4, 8]. But as we have discussed before, the 2-bit numbers in DoReDa-Net and TVM are not the ternary numbers in TNNs, so the ternary-binary mixed-precision computation is still blank.

In this article, we propose TAB as an efficient Ternary And Binary (including mixed-precision) neural network inference algorithm on general-purpose platforms including CPU and GPU. First, we adopt a unified encoding scheme rather than the particular encoding in related works like RTN and FATNN to ensure compatibility between binary and ternary values. Our unified encoding encodes the ternarized values following signed integers and encodes the binarized values using the sign bit. Second, we propose a bitwidth-last data storage scheme to remove the bit-extraction overhead while keeping the same model size as the standard 2-bit storage format. The bitwidth-last format in TAB packs and stores the first and second bits of ternary activations and weights separately after quantization. Third, we design dedicated ternary, binary, and mixed-precision bitwise multiplications utilizing CPU/GPU instructions. Our novel ternary and mixed-precision dot product based on Gated-XOR achieve the functionality with the least bitwise operations compared with related works.

We conduct the theoretical analysis based on the reciprocal throughput of instructions instead of the latency or the number of operations for reliable estimation. The proposed ternary multiplication is the most efficient one compared with DoReFa-Net and related works on TNNs. Our

proposed TAB-TNN has $39.4\times$ theoretical speedup compared with 32-bit full-precision convolution layers and $2.3\times$ theoretical speedup compared with RTN on CPU. Our mixed-precision TBN and BTN make up the lack of optimized ternary-binary mixed-precision dot products, and they have a remarkable theoretical speedup of $42.7\times$ and $73.1\times$, respectively. We implement the proposed TAB as an open-source PyTorch C++ extension with simple Python APIs to solve the lack of reference TNN and mixed-precision network libraries. Adapting to the intrinsic characteristics of CPU and GPU platforms, we fuse the data preparation functions on CPU for good cache locality but separate them on GPU for high parallelism.

End-to-end evaluation results on ResNet-18 and Darknet-19 (the backbone network of YOLOv2) have shown that our theoretical analysis is very close to reality. Our TAB-TNN is up to $34.6\times$ fast as full-precision convolution at the layer level and is $13.4\times$ fast at the model level. And TAB-TNN is up to 70.1% faster and 12.8% more power-efficient than RTN on Darknet-19. Our proposed TAB-TBN, TAB-BTN, and TAB-BNN are up to $40.7\times$, $56.2\times$, and $72.2\times$ fast as full-precision convolution layers, respectively, which are much higher than the $4.0\times$ speedup of INT8.

The rest of this article is organized as follows: Section 2 talks about the background and the differences between TAB and related works. Section 3 presents TAB's data preparation and bitwise dot product schemes and analyzes the theoretical speedup. Section 4 discusses the efficient implementation of TAB on CPU and GPU platforms. Section 5 shows the experimental results on CPU and GPU platforms and gives insights into getting high speedups. Then, Section 6 concludes this article and talks about the future works.

2 BACKGROUND AND RELATED WORKS

Low-bitwidth numbers save memory and storage, and low-precision operations have less latency and consume less energy. These features motivate researchers and engineers to quantize the convolutional neural networks for smaller model sizes and faster and more energy-efficient training and inference. Current quantization methods have adopted BFLOAT16 [21], INT8, INT4, 2-bit ternary, and 1-bit binary numbers as well as mixed-precision schemes to represent the weights and activations in CNNs. Floating-point numbers and integers-based networks still use multiplication accumulation for the dot product in convolution layers. In contrast, TNNs, BNNs, and other mixed-precision neural networks have a similar convolution workflow, as Figure 1 shows, and utilize bitwise operations for replacing the multiplications. Figure 3 shows the computation pipeline of the dot product used in the convolution of different types of CNNs. The computation pattern of floating-point and integer dot product is very straightforward, as Figure 3(a) shows. Even so, the floating-point and integer GEMM implementation has been researched for decades [1, 20, 25]. BNNs quantize the input vector and pack the sign bits into 4-bit integers or integers with longer bitwidth, as the example shows in Figure 3(b). The binary dot product used by BNNs seems to be complex, but the data flow is still in a straight line.

The acceleration of TNNs and mixed-precision TBNs is a complex research problem compared with INT8 and BNNs. FATNN [7] designs the convolution layer implementation for TNNs along with the quantization scheme and achieves good speed and accuracy. But its ternary dot product suffers from many masking operations introduced by the particular encoding and cannot be extended to mixed-precision networks. Moreover, the ternary dot product is much more complex than INT8 and BNNs, because the input data have increased by $2\times$, the pipeline is longer, and data dependencies exist. The authors of daBNN even claim that ternary, 2-bit, and 4-bit networks can not obtain high speedup on existing CPU/GPU without dedicated hardware [18].

Therefore, many TNN accelerators are on dedicated hardware platforms, including FPGA, ASIC, and In-Memory-Computing devices. GXNOR-Net [10] proposes a Gated-XNOR logic for the ternary dot product to inspire hardware design and provides a reference neuron array for TNN

Fig. 3. Task graphs of different dot products on CPU/GPU. The example input vector length and the max bitwidth are all four.

- (a) Float-point or integer dot product. It needs to perform the process for each input pair, four times in total.
- (b) Binary dot product. It calculates the dot product of four values in a single run.
- (c) Ternary dot product of RTN. It needs to perform the process two times due to its inefficient data format.
- (d) Ternary dot product of our proposed TAB. It only performs the short optimized process once.

acceleration. RTN [24] implements the bitwise multiplication-based ternary convolution on FPGA. TiM-DNN [17] proposes a novel ternary processing cell and builds a TiM-DNN accelerator utilizing In-Memory-Computing techniques. These accelerators are specially designed and optimized for TNNs, so they have high speed, low power, and small area cost.

Though the hardware accelerators provide a good reference for TNN acceleration, they rely on the flexible bit manipulation operations on FPGA and ASIC, so their computation pipelines implemented on CPU/GPU may not be as efficient as the original pipelines on the dedicated hardware. For example, GXNOR-Net is not optimized for general-purpose platforms and has no reference encoding. The XNOR operation on CPU needs two instructions XOR+NOT rather than one, so direct GXNOR-Net implementation on CPU is inefficient. Instead, RTN adopts a special encoding of the quantized values and stores the quantized data in integers sequentially. As Figure 3(c) shows, without the flexible bit manipulation operations on FPGA, RTN needs to extract the first bits and second bits by doing an AND operation with a MASK when calculating the dot product of vectors on CPU. In distinction, our TAB contains unified encoding, new bitwidth-last data format, and novel efficient dot product on general-purpose platforms. Moreover, compared with hardware-based GXNOR-Net and RTN that only accelerate TNNs, TAB applies to TNNs, TBNs, BTNs, and BNNs.

3 TAB: UNIFIED AND OPTIMIZED TERNARY AND BINARY CONVOLUTION

TAB aims to accelerate four types of ternary and binary neural networks in one unified framework: **Ternary Neural Networks (TNNs)**, **Ternary-Activation Binary-Weight Networks (TBNs)**, **Binary-Activation Ternary-Weight Networks (BTNs)**, and **Binary Neural Networks (BNNs)**. Therefore, TAB is designed with consistent data representation and dedicated ternary-ternary, ternary-binary, and binary-binary bitwise dot product computation pipelines on general-purpose platforms for both the convolution layers and the fully connected layers. Following the workflow of quantized convolution in Figure 1, we will introduce the data preparation and the bitwise **General Matrix Multiplication (GEMM)** in ternary and binary convolution, then analyze the theoretical speedup of our proposed method in this section.

3.1 Data Preparation

Taking TNNs as an example, the ternary convolution consists of quantizing the activations/weights, packing the 2-bit values into 64-bit integers, and conducting bitwise convolution, as Figure 1 shows. Following the common practice of deep learning frameworks (such as TensorFlow and Darknet) and related works (such as BitFlow [15] and daBNN [48]), we transform

Table 3. Number Representation in TAB

Number	Sign Bit	Binary (1-bit)	Ternary (2-bit)
+1	0	0	01
0	0	N.A.	00
-1	1	1	11

Table 4. Data Storage Scheme Comparison of Different Methods

Scheme	Example Data	Length	Main Difference
INT8	$X = \{+1, 0, -1, -1\}$	32 bits	Use 8-bit integers
RTN	$TX = 11011010$	8 bits	Store data sequentially with special encoding
DoReFa-Net	$TX = \{10, 01, 00, 00\}$	8 bits	Store data in a vector encoded as unsigned int
TAB Ternary	$TX = \{0011, 1011\}$	8 bits	Store first bits and second bits separately
BNN/XNOR-Net	$BX = 1100$	4 bits	Binarization using special encoding
TAB Binary	$BX = 0011$	4 bits	Binarization using the sign bits

the convolution problem into **Image-to-Column (Img2Col)** followed by bitwise GEMM to take advantage of the existing optimizations in accelerating GEMM, as Figure 4 shows. Therefore, the data preparation is composed of quantization, bit-packing, and Img2Col. We fuse or separate the quantization, bit-packing, and Img2Col based on the implementation platform to reduce the overhead of data preparation, which will be discussed in the implementation section. We will still introduce these three steps one-by-one for easy understanding here.

3.1.1 Quantization and Value Representation. TAB quantizes the activations and weights with given/pre-trained thresholds to keep pace with the SOTA research works. Every filter has its thresholds, and the high threshold α and the low threshold β can be asymmetric, as Equation (1) shows. We can also perform symmetrical ternarization as long as $\beta = -\alpha$, $\alpha > 0$. The binarization in TAB follows Equation (2) using an adjustable threshold. We set $th = 0$ in the case of the BNNs without trainable thresholds. Table 3 presents the encoding of quantized values.

$$x^t = \begin{cases} +1 & , x > \alpha \\ -1 & , x < \beta \\ 0 & , \text{Otherwise} \end{cases} , \text{with } \alpha > \beta \quad (1)$$

$$x^b = \begin{cases} +1 & , x \geq th \\ -1 & , x < th \end{cases} \quad (2)$$

3.1.2 Bit-packing and Data Storage. The quantized values are packed into long bit-width data types like 64-bit integers for higher parallelism and easy computation. Since the input channels are usually multiples of 64, we pack the input tensor across the input channel as BitFlow [15] does. We adopt the NHWC (Batch Size, Height, Width, Channel, Bitwidth) data format for the packed data, because we pack the first bits and second bits of ternarized data separately. As a result, the input tensor shape shrinks in the channel dimension by $64\times$ and increases one bitwidth dimension after bit-packing, which keeps a similar convolution logic as standard convolution.

After the bit-packing, we store the first bits (the sign bits) and the second bits separately, as Table 4 shows. For example, $\{+1, 0, -1, -1\}$ will be stored as $\{0011, 1011\}$ instead of 01001111 (RTN style) or $\{01, 00, 11, 11\}$ (DoReFa-Net Style). Our proposed storage scheme directly provides the

Fig. 4. An example of how `Img2Col` transforms direct convolution into GEMM-based convolution.

first and second bits of the packed activations and weights, removing the bit extraction overhead and simplifying the computation pipeline in bitwise multiplication.

3.1.3 Image-to-Column (`Img2Col`). After quantization and bit-packing, **Image-to-Column (`Img2Col`)** unrolls the quantized data to transform the convolution into a GEMM problem that has been researched for decades, as illustrated in Figure 4. For example, given an activation tensor with the shape (N, H, W, C), the weight tensor in shape (KN, KH, KW, C: Filter Number, Kernel Height, Kernel Width, Channel), and the output feature maps' shape is (N, OH, OW, KN: Batch Size, Output Height, Output Width, Output Channel). `Img2Col` unrolls the activation to the shape of (N*OH*OW, KH*KW*C), taking the padding and convolution stride into account. So, the transformed activation tensor matches the shape of the weight tensor viewed as (KN, KH*KW*C), and a matrix multiplication between them will produce the output feature maps with the shape (N*OH*OW, KN), which is equivalent to (N, OH, OW, KN).

3.2 Bitwise General Matrix Multiplication (GEMM)

Matrix multiplication between transformed activation tensor and weight tensor produces the output feature maps of convolution and fully connected layers. We design dedicated bitwise multiplication schemes for ternary and binary inputs, so the matrix multiplication in TAB is named bitwise **General Matrix Multiplication (GEMM)**. GEMM can be viewed as a collection of dot product operations between vectors, so we take the dot product to illustrate how we perform bitwise GEMM. We will analyze the speedup in the following subsection after introducing our methods here, and the experiment results are provided in the evaluation section.

3.2.1 Ternary Bitwise Dot Product. For ternary-activation and ternary-weights, we perform ternary bitwise multiplications instead of the 2-bit standard multiplications in TVM and DoReFa-Net. Inspired by GXNOR-Net [10] and RTN [24], we implement the bitwise multiplication based on an equivalent **Gated-XOR (GXOR)** logic. When the input operands contain zero(s), the output is always zero. Otherwise, the output is the XOR result of the operands' sign bits (first bits). As Table 5 shows, the AND result of the second bits of operands ("a2 AND b2") is 0 when the operands contain zero(s), so we can regard it as the gate of the Gated-XOR. After being filtered by this gate, the XOR result of sign bits ("a1 XOR b1") shows how many "-1" are in the multiplication result.

$$P1 = X1 \text{ XOR } W1 \quad (3)$$

$$P2 = X2 \text{ AND } W2 \quad (4)$$

$$P3 = P1 \text{ AND } P2 \quad (5)$$

$$c1+ = \text{popcnt}(P3) \quad (6)$$

$$c2+ = \text{popcnt}(P2) \quad (7)$$

$$y = (c2 - c1) - c1 = c2 - 2 \times c1 = c2 - c1 \ll 1 \quad (8)$$

Therefore, we conduct the dot product between two input vectors X and W following the logic of Gated-XOR. As Equations (3)–(8) show, $P1$ is the XOR result between the first bits (sign bits) of

Table 5. The Truth Table of Gated-XOR between Two Ternary Values in TAB

a(a1a2)	b(b1b2)	a1	b1	a1 XOR b1	a2	b2	a2 AND b2	Gated-XOR	a×b
0(00)	-	0	-	-	0	-	0	0	0
-	0(00)	-	0	-	-	0	0	0	0
+1(01)	+1(01)	0	0	0	1	1	1	0	+1
+1(01)	-1(11)	0	1	1	1	1	1	1	-1
-1(11)	+1(01)	1	0	1	1	1	1	1	-1
-1(11)	-1(11)	1	1	0	1	1	1	0	+1

Fig. 5. Task graphs of proposed dot product in TAB. (a) Ternary dot product. (b) Ternary-activation binary-weight dot product. (c) Binary-activation ternary-weight dot product. (d) Binary dot product.

X and W , $P2$ is the zero gate, so $P3$ is the Gated-XOR result of X and W . The population-count operation *popcnt* counts the number of bits “1” in an integer, and it has corresponding instructions on ARM CPU, Intel CPU, AMD CPU, and Nvidia GPU. As mentioned in the previous paragraph, $c2$ equals the total number of non-zero values in the multiplication results, because it is the *popcnt* result of the AND gate $P2$. Similarly, $c1$ is the total number of -1 in the multiplication results, so $c2 - c1$ is the number of $+1$ in the multiplication results, and we can get the dot product result y , as Equation (8) shows. Our proposed method gets the true value of the ternary dot product with only 1 XOR, 2 AND, and 2 *popcnt* operations. The proposed ternary dot product with only five Boolean operations is fast by design and suitable for implementation on CPU/GPU platforms.

We notice that there are two consecutive subtraction operations in Equation (8), which can be transformed to a subtraction followed by an integer multiplication ($2 \times c1$) or a Shift ($c1 \ll 1$). According to the instruction latency table [2], the reciprocal throughput of a Shift operation is smaller than integer multiplication but still larger than integer subtraction. So, we implement Equation (8) and the similar equations in the following subsections as two consecutive subtraction operations to reduce the computation latency. Please note that we only perform Equation (8) once for the whole vector-vector multiplication, so its computation cost is minimal.

3.2.2 Ternary-binary Mixed-precision Dot Product. Following the same logic of Gated-XOR, we build the dot product of mixed-precision dot product, as Equations (9)–(13) and Figure 5(b) show. Suppose W is the binary vector, and $X1$ and $X2$ are the first bits and second bits of the ternary vector. As we encode the binary values using the sign bit, we can get the XOR result between W and $X1$, as Equation (9) shows, then filter it using the gate $X2$. Similar to the ternary dot product, $c1$ is the number of -1 in the multiplication results, and $c2$ is the number of non-zero values in the multiplication results. Therefore, we get the true value of the ternary-binary mixed-precision dot

product, as Equation (13) shows.

$$P1 = X1 \text{ XOR } W \quad (9)$$

$$P2 = P1 \text{ AND } X2 \quad (10)$$

$$c1+ = \text{popcnt}(P2) \quad (11)$$

$$c2+ = \text{popcnt}(X2) \quad (12)$$

$$y = (c2 - c1) - c1 = c2 - 2 \times c1 \quad (13)$$

Our proposed method gets the true value of the ternary-binary dot product with only 1 XOR, 1 AND, and 2 popcnt operations. Moreover, we can further simplify the computation by calculating $c2$ in advance with a pre-known ternary vector, e.g., the weights of binary-activation ternary-weight networks. As the following equations and Figure 5(c) show, the dot product of BTNs only needs 1 XOR, 1 AND, and 1 popcnt during inference.

$$P1 = W1 \text{ XOR } X \quad (14)$$

$$P2 = P1 \text{ AND } W2 \quad (15)$$

$$c1+ = \text{popcnt}(P2) \quad (16)$$

$$y = (c2 - c1) - c1 = c2 - 2 \times c1, \quad c2 = \text{sum}(\text{popcnt}(W2)) \quad (17)$$

3.2.3 Binary Dot Product. TAB is compatible with binary neural networks and supports binary dot products. Inspired by related work XOR-Net [53], we build the binary dot product operation using only 1 XOR and 1 popcnt, as Figure 5(d) and Equations (18)–(20) show. The popcnt result c of the XOR output P is the number of -1 in the binary multiplication results, and NUM is the vector length, which is also the total number of non-zero values in the binary multiplication results. We get the binary dot product result in Equation (20), similar to previous cases.

$$P = X \text{ XOR } W \quad (18)$$

$$c+ = \text{popcnt}(P) \quad (19)$$

$$y = (NUM - c) - c = NUM - 2 \times c \quad (20)$$

We have designed efficient dot product methods for TNN, TBN, BTN, and BNN, which use 5, 4, 3, and 2 bitwise operations in the multiplication stage. Our TAB-TNN reduces two operations in the ternary dot product compared with RTN on CPU/GPU platforms. TAB supports mixed-precision and binary neural networks, thanks to the consistent encoding and optimized storage format. Our TAB also has fewer operations than DoReFa-Net and TVM in ternary and binary dot product.

3.3 Theoretical Speedup Analysis

We analyze the theoretical speedup of our method compared with standard convolution and related convolution methods on CPU/GPU to show the efficiency of TAB in this subsection. All the corresponding experiment results are provided in the evaluation section. Given an activation tensor with the shape being $(N, H, W, 64C)$ and the convolution filter tensor having a shape of $(KN, KH, KW, 64C)$, the number of **32-bit Floating-Point (FP32) Multiplication Accumulation (MAC)** in standard convolution is showed in Equation (21).

$$Ops_{FP32} = N \cdot KN \cdot H \cdot W \cdot 64C \cdot KH \cdot KW \cdot (1 + 1) \quad (21)$$

$$Ops_{pack} = N \cdot H \cdot W \cdot 64C \cdot (1 + 2) \quad (22)$$

$$Ops_{GEMM} = N \cdot KN \cdot H \cdot W \cdot C \cdot KH \cdot KW \cdot (5 + 2) \quad (23)$$

Taking TAB-TNN ternary convolution as an example, we need one COMPARE and at most two OR operations for each input value in the ternarization and bit-packing stage (packing zeros

has no overhead, packing “+1” only needs one OR). So, the total number of bitwise operations in data preparation is three, as shown in Equation (22). The shape of the packed input activation tensor is (N, H, W, C, 2), and the shape of the packed weight tensor is (KN, KH, KW, C, 2). We need five bitwise operations during the bitwise multiplication and two ADDITION operations for accumulation, so the total operations in TAB-TNN bitwise convolution are seven, as Equation (23) shows.

$$Speedup_{old} = \frac{Ops_{FP32}}{Ops_{pack} + Ops_{GEMM}} = \frac{KN \cdot KH \cdot KW \cdot 64 \times 2}{64 \times 3 + KN \cdot KH \cdot KW \cdot 7} = \frac{128 \cdot KN \cdot KH \cdot KW}{192 + 7 \cdot KN \cdot KH \cdot KW} \quad (24)$$

Traditionally, researchers calculate the theoretical speedup based on the number of **Floating-Point Operations (FLOPS)** or the number of **Operations (Ops)**. The speedup of TAB-TNN compared with full-precision networks is shown in Equation (24) following this logic. We can neglect the data preparation overhead when the number of filters is enormous, so the speedup upper bound of TAB-TNN is $128/7 = 18.3\times$. Though this calculation method works for the comparison of full-precision networks, it is not suitable for calculating the theoretical speedup of ternary and binary networks, as the latency values of bitwise instructions and 32-bit floating-point instructions are different [2].

Therefore, we take the latency of instructions into account to get more accurate speedup values of TAB series convolution. We analyze the speedup based on the reciprocal throughput of relevant instructions, because throughput-sensitive pipelines usually implement the GEMM process. According to the instruction table [2], the reciprocal throughput of one 32-bit floating-point multiplication on Intel 8-10th generation CPUs is 1 cycle, bitwise operations NOT (~), AND (&), OR and XOR (^) all need 0.25 cycle, and one popcnt takes 1 cycle. Therefore, the speedup of TAB-TNN compared with FP32 standard convolution is calculated, as Equations (25)–(27) show. We can infer from Equation (27) that the speedup will be higher with larger KN, KH, and KW, because the data preparation overhead is a constant value 96. So, we can predict that the layer level speedup will be higher with more filters, and layers with 3×3 kernels have higher speedup than layers with 1×1 kernels when the number of filters is the same. Without considering the bit-packing cost, our proposed TAB-TNN has $128/3.25 = 39.4\times$ speedup compared with FP32 standard convolution.

$$Speedup_{new} = \frac{Ops_{FP32} \cdot t_{FPMAC}}{Ops_{pack} \cdot t_{pack} + Ops_{GEMM} \cdot t_{TernaryMAC}} \quad (25)$$

$$= \frac{KN \cdot KH \cdot KW \cdot 64 \times (1 + 1)}{64 \times (1 + 2 \times 0.25) + KN \cdot KH \cdot KW \cdot (0.25 + 0.25 + 0.25 + 1 + 1 + 2 \times 0.25)} \quad (26)$$

$$= \frac{128 \cdot KN \cdot KH \cdot KW}{96 + 3.25 \cdot KN \cdot KH \cdot KW} \quad (27)$$

The speedup of TAB-TNN based on the reciprocal throughput of the instructions ($39.4\times$) is higher than that calculated based on the number of operations ($18.3\times$), and it is consistent with the experiment results in the evaluation section. Following the new speedup analyzing logic, we list the speedup values of TAB and related works in Table 6. The “1F” means that the FP32 uses float-point multiplication and float-point accumulation, while the other methods use integer operations. The **GEMM time (GEMM t)** is calculated based on the reciprocal throughput (Latency) of multiplication accumulation and the shape of the activation X (X shape), where $T = N \cdot H \cdot W \cdot C$ is a constant. Though the latency of one bitwise multiplication is higher than one FP32 multiplication, the bitwise GEMM time is much smaller than FP32, thanks to the bit-packing that reduces the number of channels.

Table 6. Operation and Reciprocal Throughput of Different Quantization Methods on Intel 8-10th CPUs

Method	X shape	$\times$	$\sim$	$\wedge$	$\&$	popcnt	Ops	Latency	Add	GEMM t	Speedup
FP32	N, H, W, 64C	1F					1	1.00	1F	128.00T	1.0 $\times$
INT8	N, H, W, 64C	1					1	1.00	1	80.00T	1.6 $\times$
DoReFa 2-bit	N, H, W, C, 2				4	4	8	6.00	4	7.00T	18.3 $\times$
RTN	N, H, W, 2C				1	4	2	3.25	2	7.50T	17.0 $\times$
TAB-TNN	N, H, W, C, 2				1	2	2	2.75	2	3.25T	39.4 $\times$
DoReFa 1/2-b	N, H, W, C				2	2	4	2.50	2	3.00T	42.7 $\times$
TAB-TBN	N, H, W, C, 2				1	1	2	2.50	2	3.00T	42.7 $\times$
TAB-BTN	N, H, W, C				1	1	1	1.50	1	1.75T	73.1 $\times$
XNOR-Net	N, H, W, C		1	1		1	3	1.50	1	1.75T	73.1 $\times$
XOR-Net	N, H, W, C				1		2	1.25	1	1.50T	85.3 $\times$
TAB-BNN	N, H, W, C				1		2	1.25	1	1.50T	85.3 $\times$

INT8 has only about 1.6 $\times$ speedup when implemented without **Single-Instruction-Multi-Data (SIMD)** instructions in this table. Indeed, INT8 networks can achieve up to 4.0 $\times$ speedup with the help of SIMD instructions, and this has been proved by existing works of Intel [38], e.g., 2.98 $\times$ on ResNet-18. So, our implementation includes SIMD optimization for INT8 for a fair comparison.

Without considering the bit-packing cost, our proposed TAB-TNN has $7.50/3.25 = 2.31\times$ speedup compared with RTN, and $7.00/3.25 = 2.15\times$ speedup compared with DoReFa-Net 2-bit with the optimized format. The packed activation tensor shape is (N, H, W, 2C), and the packed weight tensor is (KN, KH, KW, 2C) in RTN style bit-packing, so RTN repeats the multiplication for two times, which leads to slightly higher GEMM time than DoReFa-Net 2-bit.

Our mixed-precision TAB-TBN has the same 42.7 $\times$ speedup as mixed-precision DoReFa-Net 1-bit/2-bit GEMM. Our TAB-BTN pushes the speedup boundary of mixed-precision convolution to be 73.1 $\times$, the same with existing binary methods including BNN [16], XNOR-Net [36], and DoReFa-Net 1-bit. This improvement is achieved by our careful analysis of the properties of BTN and calculating some intermediate results in advance. Our TAB-BNN has the same 85.3 $\times$ speedup with SOTA method XOR-Net [53]. The binary dot product pipeline has achieved the optimal state with only one XOR and one popcnt after being researched by many related works [16, 48, 53]. So, we adopt the best existing binary dot product in TAB with unified encoding.

In summary, our TAB is the best ternary and binary network inference method in all four cases with a unified encoding and optimized bitwise dot products. The popcnt instruction now bounds the speedup of TAB, because popcnt has the same reciprocal throughput as the FP32 multiplication operation on CPU while other bitwise operations have less latency. Therefore, TAB can achieve even higher speedup on platforms with more efficient popcnt instruction. For example, TAB-TNN, TAB-TBN, and TAB-BTN may achieve 73.1 $\times$, 85.3 $\times$, and 128.0 $\times$ theoretical speedup, respectively, on AMD Zen 3 CPUs whose popcnt instruction has the same reciprocal throughput as bitwise operations [2].

4 IMPLEMENTATION

We implement our proposed TAB on both CPU and GPU platforms. We adopt two different approaches for the data preparation stage on these two platforms considering the intrinsic difference of the hardware, e.g., fused algorithm for CPU and separated algorithm for throughput-driven GPU. We implement the bitwise GEMM following the existing optimization methods on common GEMM, which have been researched for decades. The code is open source as a PyTorch Extension for easy integration with current CNN models.

ALGORITHM 1: Fused ternarization, bit-packing, and Img2Col Algorithm**Input:** input tensor X , high threshold α , low threshold β , padding $p1$ and $p2$, stride $s1$ and $s2$ **Output:** Img2Col tensor $A^{(t)}$

```

1: packed channel = input channel / 64; Initialize a zero tensor  $X^{(t)}$ 
2: for each input height  $h$  do
3:   for each input width  $w$  do
4:     for each packed channel  $pc$  do
5:        $b1 = b2 = 0$  // Initialize the container integers as zeros
6:       for  $i$  from 0 to 63 do
7:         if  $X_{h,w,64pc+i} > \alpha$  then
8:           // Ternarize  $x$  as +1 encoded as "01." Set the second bit to be "1"
9:            $b2 = b2 \text{ OR } \text{onebit}[i]$ 
10:        else if  $X_{h,w,64pc+i} < \beta$  then
11:          // Ternarize  $x$  as -1 encoded as "11." Set the first and second bits to be "1"
12:           $b1 = b1 \text{ OR } \text{onebit}[i]$ ,  $b2 = b2 \text{ OR } \text{onebit}[i]$ 
13:        end if
14:      end for
15:       $X_{h+p1,w+p2,pc,0}^t = b1$ ,  $X_{h+p1,w+p2,pc,1}^t = b2$ 
16:      // Do Img2Col when the quantized data is ready
17:      if ( $h > KH$ ) AND ( $w > KW$ ) AND ( $(h+p1)\%s1 == 0$ ) AND ( $(w+p2)\%s2 == 0$ ) then
18:        for  $j$  from 0 to  $KH*KW$  do
19:          // Copy corresponding data to the new array  $A$ .
20:           $A_{...,j,0}^t = X_{...,0}^t$ ,  $A_{...,j,1}^t = X_{...,1}^t$  // The index are omitted for easy understanding
21:        end for
22:      end if
23:    end for
24:  end for
25: end for
26: return  $A^t$ 

```

4.1 Fused Data Preparation Algorithm on CPU

The data locality and the cache hit rate are essential for high CPU performance. So, we fuse the quantization, bit-packing, zero padding, and Img2Col to reduce the total number of for() loops for higher efficiency. The fused data preparation only goes through the activation once, so the reused data may give better cache hit rate to boost the performance.

Algorithm 2 is the fused ternary data preparation algorithm of TAB. The first and second bits of the ternarized values are stored into intermediate container integers $b1$ and $b2$ first, then stored to the quantized tensor. We initialize the container integers as zeros so we only need to set the corresponding bits of $b1$ and $b2$ to "1" during bit-packing. As the coding of 0 is "00," our initialization on the container integers removes the bit-packing overhead of zeros.

In the ternarization and bit-packing, we compare the input value with the two thresholds α and β , as Equation (1) shows, then set the corresponding bits of the containers accordingly. We utilize $\text{onebit}[i]$, a pre-defined 64-bit integer with only the i th bit to be "1," to set the i th bit of the container to "1" using an OR operation. The quantized data are stored to $X^{(t)}$ according to the padding values around height and width, e.g., $p1$ and $p2$. This storing scheme hides the zero-padding overhead, because $X^{(t)}$ is also initialized as zeros.

We conduct the `Img2Col` when the quantized data at corresponding rows and columns are ready. `Img2Col` deals with the convolution stride s_1 and s_2 , so we also need to judge whether current data pointers like the height h and width w are in the correct place for `Img2Col`. We copy the data from the quantized tensor $X^{(t)}$ to the `Img2Col` tensor $A^{(t)}$ after checking the conditions. Finally, the `Img2Col` tensor $A^{(t)}$ is ready for GEMM with the weights to generate the output feature maps, because we have dealt with the padding and stride of the convolution here.

4.2 Separated Data Preparation Algorithm on GPU

Unlike CPU, GPU has a parallel programming model where parallelism is most important. The CUDA cores inside Nvidia GPU perform functions in parallel on different parts of the input data based on the index of the running threads. So, we adopt the separated data preparation algorithm for GPU to avoid the data dependency or data transfer between CUDA cores. The separated functions can be mapped to the GPU efficiently, because the four `for()` loops of data preparation can be fused to only two or three `for()` loops in ternarization, padding and `Img2Col`, as Algorithm 2 shows. Thus, the separated data preparation algorithm reduces the depth of the `for()` loops and relieves the data dependencies to achieve high parallelism, making it easier to find the index and more friendly for GPU programming.

Given the activation shape (N, H, W, C) , which can be viewed as $(N \times H \times W, C)$, the ternarization can be processed in parallel across the (N, H, W) , and there is only two `for()` loops left. So, the CUDA cores can do the ternarization along the channel directly in the assigned points based on the block and thread ID.

The ternarized activation shape is now (N, H, W, PC) , viewed as $(N \times H, W \times PC)$. The bitwidth dimension is omitted here for easy understanding. The packed channel PC is $64\times$ smaller than the original channel C , so the padding and `Img2Col` overhead will be more negligible than the ternarization. We can efficiently do the parallel padding in this logical shape $(N \times H, W \times PC)$ by calling memory copy functions. The padded activation shape is (N, PH, PW, PC) , which is also the input tensor shape for the `Img2Col`. The activation shape after `Img2Col` will be (N, OH, OW, KH, KW, PC) , which can be viewed as $(N \times OH \times OW, KH, KW \times PC)$. We parallel `Img2Col` across (N, OH, OW) according to the output shape to achieve high throughput.

In summary, the data preparation on GPU trades memory for high throughput and low latency. The separated algorithm is highly optimized for GPU implementation in loop fusion and avoids the data dependency in the same function call. Algorithm 2 can also avoid unnecessary function calls on `Padding()` and `Img2Col()`. For example, we do not need padding and `Img2Col` on fully connected layers and most convolution layers with 1×1 kernels. So, the separated data preparation algorithm on GPU will not bring high overhead for quantized networks.

4.3 Bitwise GEMM Implementation

Algorithm 3 shows the basic ternary bitwise GEMM algorithm. Mixed-precision and binary bitwise GEMM follow the same workflow with different dot products. We implement our bitwise GEMM in C++ with the reference of related works on CPU including BitFlow [15], NCNN [41], and How-to-Optimize-GEMM [34]. We have optimized the basic algorithm using blocking, SIMD instructions, and OpenMP parallel processing in our PyTorch Extension where available. We will still use the basic GEMM algorithm for a fair comparison across different methods in our experiments.

5 EVALUATION

We show the experiment setup in Section 5.1, then conduct an ablation study on the performance gain of the proposed encoding and computation pipeline. Next, we present the end-to-end

ALGORITHM 2: Separated ternarization, bit-packing, and Img2Col Algorithm**Input:** input tensor X , high threshold α , low threshold β , padding $p1$ and $p2$, stride $s1$ and $s2$ **Output:** Img2Col tensor $A^{(t)}$

```

1: packed channel PC = input channel C / 64;
2: // Current tensor shape is (N, H, W, C), which can be viewed as (N*H*W, C)
3: for each assigned point  $nhw$  do // Ternarization parallelism across (N, H, W)
4:    $b1 = b2 = 0$ ,  $pc = 0$  // Initialize the container integers and packed channel iterator as zeros
5:   for each input channel  $c$  do
6:     if  $X_{nhw,c} > \alpha$  then
7:       // Ternarize  $x$  as +1 encoded as "01." Set the second bit to be "1"
8:        $b2 = b2 \text{ OR } \text{onebit}[i]$ 
9:     else if  $X_{nhw,c} < \beta$  then
10:      // Ternarize  $x$  as -1 encoded as "11." Set the first and second bits to be "1"
11:       $b1 = b1 \text{ OR } \text{onebit}[i]$ ,  $b2 = b2 \text{ OR } \text{onebit}[i]$ 
12:    end if
13:    if  $c \% 64 == 0$  then
14:       $X_{nhw,pc,0}^t = b1$ ,  $X_{nhw,pc,1}^t = b2$ 
15:       $b1 = b2 = 0$ ,  $pc = pc + 1$ 
16:    end if
17:  end for
18: end for
19: // Only do Padding() in need. Current tensor shape (N, H, W, PC) is viewed as (N*H, W*PC)
20: for each assigned point  $nh$  do
21:   Memcpy(from  $X^t$  to  $X^p$ , vector length =  $W*PC$ )
22: end for
23: // Only do Img2Col() in need. Current tensor shape (N, OH, OW, KH, KW, PC) is viewed as (N*OH*OW, KH, KW*PC)
24: for each assigned point  $nOhOw$  do
25:   for each kernel height  $kh$  do
26:     Memcpy(from  $X^p$  to  $A^t$ , vector length =  $KW*PC$ )
27:   end for
28: end for
29: return  $A^t$ 

```

evaluation on CPU and GPU in Sections 5.3 and 5.4. Finally, Section 5.5 analyzes the speedup across the batch size and quantizes the first and last layers.

5.1 Experiment Setup

We implement TAB and related methods on a **Raspberry Pi (Rpi)** 400 CPU, a low-power laptop CPU, and a desktop GPU, as listed in Table 7. The baselines are standard **32-bit full-precision networks (FP32)**, **8-bit integer quantization (INT8)**, reference method RTN, and DoReFa-Net 2-bit with optimized data format. The evaluated methods, including INT8, RTN, and DoReFa-Net, are optimized with OpenMP multi-thread and **Single-Instruction-Multi-Data (SIMD)** APIs for a fair comparison. The end-to-end test implemented on ARM and Intel CPU enables the Neon and AVX2 SIMD instructions with 8 threads. The end-to-end test on GPU utilizes 1,568 CUDA cores for all GEMM functions. Moreover, the quantization functions of INT8, RTN, and DoReFa-Net are optimized using Algorithms 1 and 2 wherever applicable. More implementation details

ALGORITHM 3: Basic Ternary Bitwise GEMM Algorithm**Input:** Activation matrix A with the shape (M, K) , weight matrix B with the shape (N, K) **Output:** Output feature map C

```

1: for each activation row  $m$  do
2:   for each weight column  $n$  do // Replace operations in this for() loop to realize other bitwise
      GEMMs.
3:     cnt1 = cnt2 = 0; // Initialize the popcnt summations to zeros
4:     for each point  $k$  do
5:       p1 =  $A_{m,k,0}$  XOR  $B_{n,k,0}$ ;
6:       p2 =  $A_{m,k,1}$  AND  $B_{n,k,1}$ ;
7:       cnt1 += popcnt64(p2);
8:       cnt2 += popcnt64(p1 AND p2);
9:     end for
10:     $C_{m,n} = \text{cnt1} - \text{cnt2} - \text{cnt2}$ ;
11:  end for
12: end for
13: return  $C$ 

```

Table 7. The Experiment Platforms and Compiling Environment

Type	Name	OS and Compiler	Power Measurement
Embed. CPU	Broadcom BCM2711 4 Cores@1.8GHz	Raspbian OS; GCC 8.3.0 arm-linux-gnueabi	UM25C Power Meter Resolu.: 0.5s, 0.001mW
Laptop CPU	Intel Core i7-8565U 4 Cores@1.8-4.6GHz	Windows 10 Home 21H1 VS Community 2019 (v142)	Not Applicable
Desktop GPU	GeForce RTX 3080 8704 C@1.44-1.81GHz	Windows 10 Pro 21H1 CUDA 11.4.2, Driver 471.41	Nvidia-smi 471.41 Resolu.: 0.5s, 0.01W

are provided in the ablation study. We run ResNet-18 and Darknet-19 (the backbone network of YOLOv2) of the evaluated methods in C++. The experiments repeat 20 times to report the average execution time using the high-resolution clock in C++ standard library Chrono. A USB power meter UM25C [37] measures the power of the Raspberry Pi 400 CPU, and the Nvidia-smi provides the power of Nvidia GPU in the command line. As the resolution of power measurement is 0.5 s, we repeat the quantized inference up to 100 times for enough power points.

To maintain the accuracy of quantized neural networks, we keep the common practice as other papers [19, 36, 44, 47, 48] and leave the first and the last layers unquantized. We also present a case study on the impact of quantizing these two layers in Section 5.5.

5.2 Ablation Study on Implementation Optimizations and Proposed Method

We conduct an ablation study on the performance gain from the proposed method and the implementation in this subsection. Our TAB alters two variables: the encoding scheme and the computation pipeline of the dot product. Also, the multi-threading and SIMD instruction in implementation affect the performance. So, we conduct the ablation study on these factors to show how they contribute to the performance.

Table 8 compares the speedup, power, and energy efficiency of different optimizations on the baseline **DoReFa-Net 2-bit (DRF 2-bit)**. We utilize the **OpenMP (OMP)** APIs for 8-thread processing and SIMD optimizations on both the quantization algorithms and the basic GEMM in

Table 8. Ablation Study on Rpi 400 CPU (Batch Size = 4)

		DRF 2-bit				DRF + ODF	TAB-TNN
Optimizations	OMP Multi-Thread?		✓	✓	✓	✓	✓
	Use libpopcnt?			✓			
	SIMD Optimization?				✓	✓	✓
	New Encoding?				✓	✓	✓
	New Computation?						✓
ResNet-18	Overall Speedup (×)	1.0	3.6	7.5	8.5	19.0	23.9
Darknet-19	Overall Speedup (×)	1.0	3.9	7.4	8.8	18.8	22.6
	Average Power (W)	3.4	5.5	5.5	5.6	5.1	4.9
	Energy Efficiency (×)	1.0	2.4	4.6	5.4	12.6	15.8

Algorithm 3. The libpopcnt [22] in the optimizations provides fast **population count (popcnt)** across different CPU architectures. As the dot products in RTN, DoReFa-Net, and TAB contain AND and XOR besides the popcnt operations, we implement dedicated SIMD kernels for the dot products and receive 13%–19% better performance than libpopcnt. So, we keep the dedicated SIMD kernels for RTN, DoReFa-Net, and TAB.

DoReFa-Net 2-bit with optimized data format (DRF + ODF) has 2.14–2.24× speedup, 91% power, and 2.33× energy efficiency compared with that without new encoding. TAB-TNN with the optimized dot product computation has 1.20–1.26× speedup, 96% power, and 1.25× energy efficiency compared with DoReFa + ODF. The new encoding and data format improves the data access and reduces the masking operations in the dot product, so it provides good performance and energy efficiency. The new ternary dot product in TAB-TNN further reduces the operations in GEMM, so it also brings higher performance and energy efficiency but not as much as the new encoding.

5.3 End-to-End Test on CPU

We present the evaluation of ResNet-18 and Darknet-19 on the embedded and laptop CPUs in this subsection. The DoReFa-Net in this subsection and the “DoReFa”/“DRF” in figures and tables are the same **DoReFa-Net 2-bit with optimized data format (DoReFa + ODF)**, as presented in the ablation study and Figure 2(b). The “F+L Layers” refers to the first and last layers and “E-to-E” means end-to-end. The batch size is 4 in all experiments on CPU.

5.3.1 Speedup and Energy Efficiency of ResNet-18. We list the layer level and end-to-end speedup values of our method on ResNet-18 compared with FP32 on Rpi 400 CPU in Figure 6. The kernel size and number of filters in each layer are listed below the layer number for reference. The “DS” in the row of “Layer” means a down-sampling layer. The “FC” in the “Kernel” row means a fully connected layer.

Our proposed TAB-TNN on Rpi 400 CPU has up to 38.3× (RTN: 17.5×, DoReFa + ODF: 27.8×) single-layer speedup on ResNet-18. TAB-TNN is 1.3–2.0× faster than RTN, which is very close to the 2.3× theoretical speedup. It is also 1.1–1.5× faster than DoReFa-Net 2-bit with optimized data format. Our TAB-TBN, TAB-BTN, and TAB-BNN have up to 44.8×, 54.4×, and 72.3× layer level speedup compared with FP32, while INT8 only has up to 5.5× speedup.

We infer from Figure 6 that the single layer speedup is higher when the computation workload is higher. The down-sampling layers using 1×1 kernels have very few operations in the bitwise GEMM, so the data preparation overhead accounts for a large part of the execution time in this layer and makes the layer level speedup very small. Similarly, the last few convolution layers have

Fig. 6. The layer-level and network-level speedup of TAB on ResNet-18 compared with FP32 on Rpi 400 CPU.

Table 9. Execution Time, Speedup, and Power Consumption of ResNet-18 on CPU (Batch Size = 4)

Time: ms. Speedup: ×. Power: W.			Reference Methods				TAB Series Methods				Max
			FP32	INT8	RTN	DRF	TNN	TBN	BTN	BNN	Ref.
Rpi 400 CPU	Quant.	Time	3,446.4	1,104.9	683.1	503.2	410.6	409.0	298.8	285.5	34.5
	Layers	Speedup	1.0	3.1	5.0	6.8	8.4	8.4	11.5	12.1	100.0
	F + L	Time	226.5	220.0	223.3	221.3	223.6	223.3	222.6	224.2	226.5
	Layers	T/Total	6.2%	16.6%	24.6%	30.5%	35.3%	35.3%	42.7%	44.0%	86.8%
	End-to-End	Speedup	1.0	2.8	4.1	5.1	5.8	5.8	7.0	7.2	14.1
		Power	4.7	4.8	5.1	5.0	4.8	4.9	4.5	4.6	4.7
		Energy	100.0%	36.4%	26.4%	20.6%	17.5%	17.6%	13.8%	13.6%	7.1%
Lap- top CPU	Quant.	Time	1,403.7	442.1	168.6	106.9	87.0	84.4	63.3	57.2	14.0
	Layers	Speedup	1.0	3.2	8.3	13.1	16.1	16.6	22.2	24.5	100.0
	F + L	Time	103.5	105.3	103.4	103.9	103.0	103.7	103.6	103.4	103.5
	Layers	T/Total	6.9%	19.2%	38.0%	49.3%	54.2%	55.1%	62.1%	64.4%	88.1%
	E-to-E	Speedup	1.0	2.8	5.5	7.1	7.9	8.0	9.0	9.4	12.8

a large number of filters, so the data preparation overhead only accounts for a small part of the execution time, and the speedup is near to the theoretical GEMM speedup.

We summarize the execution time, speedup, and power consumption of FP32 and quantized ResNet-18 on CPU in Table 9. TAB-TNN is 2.7×, 1.7×, and 1.2× as fast as INT8, RTN, and **DoReFa-Net (DRF)** with optimized format across all quantized layers on Rpi 400 CPU. At the network level, INT8, RTN, and DoReFa-Net 2-bit have 2.8×, 4.1×, and 5.1× speedup, respectively, compared with FP32, while our proposed TAB-TNN, TAB-TBN, TAB-BTN, and TAB-BNN have higher 5.8×, 5.8×, 7.0×, and 7.2× speedup. The power of INT8, RTN, DoReFa-Net, TAB-TNN, and TAB-TBN is higher than the power of FP32, because SIMD instructions bring better utilization of CPU resources. The power values of TAB series methods are smaller than RTN and DoReFa-Net, thanks to the efficient data format and the simplified computation. TAB-TNN is 2.1×, 1.5×, and 1.2× as energy-efficient as INT8, RTN, and DoReDa-Net with optimized data format.

The speedup values of quantized networks are higher on laptop CPU. Our proposed TAB-TNN, TAB-TBN, TAB-BTN, and TAB-BNN have 16.1×, 16.6×, 22.2×, and 24.5× speedup on the quantized layers compared with FP32. The end-to-end speedup values of TAB on laptop CPU are only 7.9–9.4×, lower than the quantized layers. As we do not quantize the first and the last layers, these two layers take a long execution time and cause the end-to-end speedup to be smaller than the single-layer speedup. As Table 9 shows, though the execution time of the

Fig. 7. The layer-level and network-level speedup of TAB on Darknet-19 compared with FP32 on laptop CPU.

Table 10. Speedup of Darknet-19 on CPU (Batch Size = 4)

Speedup: ×.		Reference Methods				TAB Series Methods				Max Ref.
		FP32	INT8	RTN	DRF	TNN	TBN	BTN	BNN	
Rpi	Quant. Speedup	1.0	3.7	6.7	7.5	8.9	9.3	11.8	13.3	100.0
	(F + L)/Total	3.1%	10.1%	17.0%	18.5%	21.5%	22.0%	26.6%	29.0%	76.4%
400 CPU	E-to-E Speedup	1.0	3.5	5.8	6.3	7.2	7.5	9.0	9.7	24.4
	Power (W)	4.7	5.0	5.3	4.9	4.7	4.9	4.5	4.3	4.7
	Energy	100.0%	30.3%	19.1%	16.6%	13.8%	13.9%	10.7%	9.6%	4.1%
Lap-top	Quant. Speedup	1.0	4.4	7.6	12.8	14.8	16.9	20.6	22.8	100.0
	(F + L)/Total	2.2%	9.0%	14.4%	21.9%	24.8%	27.5%	31.7%	33.7%	68.7%
CPU	E-to-E Speedup	1.0	4.1	6.7	10.2	11.4	12.5	14.4	15.5	32.0

first and last layers is a constant, it accounts for an increasingly large part of the total execution time with a higher speedup of quantized layers, e.g., from 6.9% in FP32 to 64.4% in TAB-BNN. The maximum reference (Max Ref.), which accelerates the quantized layers to 100.0%, only gets 12.8% end-to-end speedup. So, the 7.9–9.4% end-to-end speedup of TAB is reasonable and good.

5.3.2 Speedup and Energy Efficiency of Darknet-19. We evaluate our method on Darknet-19 (the backbone network of YOLOv2) with higher computation workloads than ResNet-18. Figure 7 shows the single layer and end-to-end speedup of TAB on laptop CPU. Our TAB-TNN has more than 31.0% (RTN: 11.0%, DoReFa + ODF: 24.0%) speedup on layer 16 and 18 of Darknet-19 compared with FP32. So, TAB-TNN is 2.8% and 1.3% as fast as RTN and DoReFa-Net 2-bit in a single convolution layer. Similarly, TAB-TBN, TAB-BTN, and TAB-BNN have more than 39.0%, 45.0%, and 62.0% speedup in these two layers. The speedup values are near to the theoretical speedup analysis (TAB-TNN: 39.4%, TAB-TBN: 42.7%, TAB-BTN: 73.1%, TAB-BNN: 85.3%).

The speedup breakdown and the power consumption on Darknet-19 are listed in Table 10. The speedup of quantized layers in TAB ranges from 8.9% to 13.3% on Rpi 400 CPU, and the first and last layers only account for no more than 30% of the total execution time, which reveals that the acceleration of quantized layers is not high enough. The quantized convolution has quantization, bit-packing, Img2Col, and bitwise GEMM stages, but the theoretical speedup upper bound is only calculated by the GEMM. Hence, the real-world speedup is lower than the upper bound. Applying better algorithm and implementation optimizations for all these four stages will bring higher end-to-end speedup.

Fig. 8. The layer-level and network-level speedup of ResNet-18 on desktop GPU (Batch size = 16).

The power of Darknet-19 on Rpi 400 CPU goes down from RTN to TAB-BNN, thanks to the proposed efficient encoding and bitwise dot product with fewer operations. Our TAB-TNN is 2.2 $\times$, 1.4 $\times$, 1.2 $\times$ energy efficiency as INT8, RTN, and DoReFa-Net on Rpi 400 CPU. TAB series methods have 14.8–22.8 $\times$ speedup across quantized layers and 11.4–15.5 $\times$ end-to-end speedup compared with FP32 on laptop CPU. TAB-TNN has 2.8 $\times$, 1.7 $\times$, and 1.1 $\times$ speedup compared with INT8, RTN, and DoReFa-Net on Darknet-19 on laptop CPU.

5.4 End-to-End Test on GPU

We present the evaluation of ResNet-18 and Darknet-19 on GPU in this subsection. All the convolution and fully connected layers, including the FP32 baseline, are implemented and executed on the GPU. And the DoReFa-Net in this subsection still stands for the DoReFa-Net 2-bit with optimized data format, as presented in Figure 2(b).

5.4.1 Speedup of ResNet-18. The speedup of ResNet-18 on GPU shown in Figure 8 is quite different from the speedup graphs on CPU. The quantized layers with 3×3 kernels have consistent high speedup from layer 2. Thanks to the massive parallelism on GPU, we can finish the data preparation stage faster, and the speedup values depend less on the activation shape and the number of filters. So, the speedup values from layer 2 to layer 10 are very steady, then the speedup goes up with the increased filter number.

Our TAB-TNN has up to 33.5 $\times$ (RTN: 24.2, DoReFa-Net: 26.8 $\times$) speedup compared with FP32 and is up to 1.4 $\times$ and 1.3 $\times$ as fast as RTN and DoReFa-Net. The TAB-TBN, TAB-BTN, and TAB-BNN have up to 40.8 $\times$, 54.3 $\times$, and 71.7 $\times$ speedup, respectively. As Table 11 shows, the overall speedup in quantized layers of TAB series methods between 19.5–36.1 $\times$ are higher than that on CPU, which proves that our proposed method is effective on GPU as well. The end-to-end speedup of TAB ranges from 8.5–10.5 $\times$ due to the first and last layers. The first and last layers account for 60%–70% of total execution time and become the new bottleneck in TAB quantized ResNet-18. Therefore, reducing the executing time of these two layers, e.g., quantizing them using INT8, may further accelerate the networks. The power of different methods on GPU is almost the same (123–125 W), because we activate the same number of CUDA cores. As more cores bring more computation capability, we distribute the workloads of all the methods to the same number of CUDA cores for a fair comparison. So, the energy efficiency of TAB on the desktop GPU is almost the same as the speedup.

5.4.2 Speedup of Darknet-19. Figure 9 presents the GPU speedup of our proposed TAB on Darknet-19. TAB-TNN has 8.2–34.6 $\times$ (RTN: 8.8–25.8 $\times$, DoReFa-Net: 7.3–29.1 $\times$) speedup in

Table 11. Execution Time and Speedup Analysis of Quantized Resnet-18 on Desktop GPU (Batch Size = 16)

Speedup: $\times$.		Reference Methods				TAB Series Methods				Max Ref.
		FP32	INT8	RTN	DoReFa	TNN	TBN	BTN	BNN	
Quant.	Time (ms)	494.7	291.5	32.1	29.6	25.3	22.7	15.6	13.7	4.9
Layers	Speedup	1.0	1.7	15.4	16.7	19.5	21.8	31.7	36.1	100.0
F + L	Time (ms)	37.2	35.2	36.8	36.9	36.9	36.8	37.0	36.9	37.2
Layers	Time/Total	7.0%	10.8%	53.5%	55.5%	59.3%	61.8%	70.3%	73.0%	88.3%
End-to-End	Speedup	1.0	1.6	7.7	8.0	8.5	8.9	10.1	10.5	12.6
	Power (W)	122.5	122.9	122.6	123.2	123.1	123.3	123.5	123.0	122.5
End	Energy	100.0%	58.3%	12.9%	12.5%	11.7%	11.2%	9.9%	9.5%	7.8%

Fig. 9. The layer-level and network-level speedup of Darknet-19 on desktop GPU (Batch size = 16).

quantized layers, while INT8 only has up to 1.2–2.4 $\times$ speedup. TAB-TBN, TAB-BTN, and TAB-BNN have 8.6–40.7 $\times$, 11.2–56.2 $\times$, and 11.6–72.2 $\times$ speedup in quantized layers of Darknet-19. We notice that the RTN’s speedup on layer 2 is 1.5–1.8 $\times$ high as DoReFa-Net, TAB-TNN, and TAB-TBN. The performance gain of TAB mainly comes from the bit-level parallelism across packed channels. Layer 2 only has 32 channels. RTN stores the quantized data in one packed channel because the 64-bit container is enough for the 2-bit quantized data. But TAB-TNN stores the quantized data in two packed channels due to the separated first and second bits. This difference favors RTN to access fewer data during GEMM and get higher speedup than TAB-TNN in layer 2.

Those convolution layers with 1×1 kernels have much fewer operations than others in GEMM, so these layers have small speedup due to the relatively larger data preparation overhead. We also notice that the speedup of layers with 1×1 kernels is gradually increasing with the number of filters. As we have optimized data preparation of the layers with 1×1 kernels to be only quantization without padding and Img2Col overhead, the speedup increase with the increasing GEMM workload on more filters. The speedup increase of layers with 1×1 kernels is also noticeable on CPU.

The execution time analysis of Darknet-19 on GPU is provided in Table 12. RTN is 3% faster than DoReFa-Net on quantized layers due to the reason mentioned the first paragraph. TAB-TNN outperforms RTN and DoReFa-Net by 18% and 22%. Our TAB series methods have 13.4–19.1 $\times$ end-to-end speedup compared with FP32. The power of different methods is very steady due to the same number of utilized CUDA cores. TAB-TNN is 7.5 $\times$, 1.1 $\times$, and 1.1 $\times$ energy-efficient as INT8, RTN, and DoReFa-Net on Darknet-19.

Table 12. Speedup of Darknet-19 on Desktop GPU (Batch Size = 16)

Speedup: $\times$.	Reference Methods				TAB Series Methods				Max Ref.
	FP32	INT8	RTN	DoReFa	TNN	TBN	BTN	BNN	
Quant. Layers Speedup	1.0	1.8	17.1	16.6	20.2	23.1	31.3	38.5	100.0
(F + L Layers)/Total	2.7%	4.8%	32.0%	31.3%	35.8%	39.1%	46.6%	51.7%	73.6%
End-to-End Speedup	1.0	1.8	11.9	11.7	13.4	14.4	17.2	19.1	27.2
Power (W)	124.3	124.0	124.1	124.1	123.6	124.5	123.8	124.5	124.3
Energy	100.0%	55.6%	8.3%	8.4%	7.4%	6.8%	5.7%	5.1%	3.5%

Fig. 10. The overall speedup of quantized layers in ResNet-18 on laptop CPU.

5.5 Case Studies

The speedup of quantized methods is related to the batch size, and Sections 5.3 and 5.4 have presented the speedup of every layer at particular batch sizes. So, this subsection will compare the overall speedup of all quantized layers across the batch size, then show the average maximum single layer speedup across the batch size. Last, the third case study will explore quantizing the first and the last layers to provide faster inference without sacrificing accuracy.

5.5.1 Speedup of Quantized Layers across Batch Size. Figure 10 shows the relationship between the quantized layers and the batch size on the laptop CPU. The difference between methods is slight when the batch size is smaller than 8. The speedup of quantized layers increases when the batch size is no more than 8 and becomes steady with larger batch size. Using a large batch size on the CPU will increase the latency, but the speedup will be the same, which shows that the CPU is suitable for inference using a small batch size.

We conduct the same case study on the desktop GPU and present the result in Figure 11. The speedup of quantized layers on GPU increases with the batch size, and it converges around a large batch size, e.g., 32 or 64. GPU is throughput-driven with a parallel programming model and a large graphic RAM, so it needs relatively large workloads to utilize its computational resources fully. FP32 networks conduct computational intensive 32-bit multiplication accumulation while TNNs perform light-weight Boolean operations, which means Fp32 networks can utilize more GPU resources than TNNs under the same batch size. Therefore, ternary and binary networks need large batch sizes to feed the GPU and get a high speedup.

5.5.2 Maximum Layer Level Speedup across Batch Size. We explore the maximum layer level speedup across batch size on the CPU, as Figure 12 shows. There is not much change in the speedup of RTN and TAB-TNN compared with FP32. Our TAB-TNN keeps 29–35 $\times$ (RTN: 10–11 $\times$, DoReFa + ODF: 18–24 $\times$) speedup across different batch size. The maximum layer level speedup has no obvious trend, and the relative magnitude of the speedup values is stable under experimenting variation. The execution time of these layers are minimal, so a variation around 0.5 ms may bring

Fig. 11. The overall speedup of quantized layers in ResNet-18 on desktop GPU.

Fig. 12. The average speedup of top three layers in Darknet-19 on laptop CPU.

obvious change to the speedup. Our TAB-TBN, TAB-BTN, and TAB-BNN can achieve around 35 $\times$, 45 $\times$, and 60 $\times$ speedup at most cases compared with FP32.

Figure 13 presents the maximum layer level speedup of Darknet-19 on GPU. We observe the same increasing trend as the speedup of quantized layers on GPU. The tiny difference between RTN (19.9–20.2 $\times$) and DoReFa-Net (20.7–21.1 $\times$) is consistent with the theoretical speedup analysis. The difference between quantization methods is minimal when the batch size is small. Then the speedup values increase with the batch size and become steady with a large batch size. This phenomenon is consistent with our previous analysis on the utilization of GPU resources. The small batch size cannot fully feed the GPU. For example, most TAB quantized layers of ResNet-18 take smaller than 1.5 ms when the batch size is 16. Therefore, we need to improve the GPU utilization of TAB convolution layers with small batch sizes.

5.5.3 Quantizing the First and the Last Layers. This case study explores how quantizing these two layers contributes to overall speedup. BNN [16], XNOR-Net [36], and DoReFa-Net [51] do not quantize the first and last layers to extreme 1-bit values for three reasons. First, the first layer is sensitive to the accuracy [14] and quantizing them leads to a significant accuracy drop. Second, the first layer usually only contains three channels and the filter size is 1×1 in the last layer, so they account for less execution time and quantizing them leads to slightly higher speedup. Third, they can be quantized to INT8 for a good accuracy-speed tradeoff. Many related works keep this practice and leave the first and last layers in FP32 [26, 29, 35, 48] or INT8 [28, 32, 45]. Though some related works quantize the whole network [42, 50], they only quantize the network with higher bitwidth (e.g., 2-bit/5-bit mixed-precision), not in ternary or binary mode.

Therefore, we conducted the case study in FP32, INT8, TAB-TNN, and TAB-BNN on the laptop CPU, as Table 13 shows. The “Normal” method does not quantize the first and the last layer, but “Layer 1 in INT8” also quantizes the first layer in INT8. The “F+L Difference” shows the difference of total execution time of the first and the last layers, taking FP32 as the baseline. Our findings are as follows: First, whether quantizing the last layer does not matter, because its execution time in FP32 is shorter than 2 ms. Second, quantizing all the layers may not increase speed due to the data

Fig. 13. The average speedup of top three layers in Darknet-19 on GPU.

Table 13. Case Study on Quantizing All the Layers and Quantizing the First Layer using INT8 on Laptop CPU

Network		ResNet-18				Darknet-19			
Method		FP32	INT8	TNN	BNN	FP32	INT8	TNN	BNN
Execution Time (ms)	First Layer	120.8	40.9	93.2	54.7	52.3	47.1	71.6	48.0
	Mid Layers	1,636.4	334.3	101.6	68.2	2,510.6	522.7	170.2	110.2
	Last Layer	0.92	0.19	0.04	0.03	1.82	0.36	0.08	0.05
	F+L Difference	0.0	-80.6	-28.4	-67.0	0.0	-6.7	+17.5	-6.1
End-to-end Speedup (×)	Normal	1.0	3.9	7.9	9.3	1.0	4.4	11.4	15.6
	Quantizing All	1.0	4.7	9.0	14.3	1.0	4.5	10.6	16.2
	Improvement	0.0%	21.5%	14.6%	54.5%	0.0%	1.2%	-7.3%	3.8%
	Layer 1 in INT8	1.0	4.7	12.3	16.0	1.0	4.5	11.7	16.1
	Improvement	4.8%	21.2%	55.7%	72.7%	0.2%	0.9%	2.4%	3.3%

preparation overhead and the underutilization of bitwise GEMM. The containers of quantized data are usually long bitwidth integers like 64-bit integers, but the first layer with only three channels utilizes about $3/64 = 4.7\%$ of the containers. So, the execution time may not reduce as shown in Darknet-19. Third, quantizing the first layer in INT8 can provide higher speedup and maintain accuracy [28, 32, 45]. It improves TAB-TNN's and TAB-BNN's speed compared to quantizing all the layers on ResNet-18 and Darknet-19. In summary, quantizing the first (and the last) layer(s) to INT8 provides faster inference without a noticeable accuracy drop.

6 CONCLUSION

In this article, we propose TAB as an efficient ternary, binary, and mixed-precision CNN inference method on the edge. TAB contains unified encoding, efficient bit-packing, and data storage scheme, and optimized bitwise dot products for TNNs, TBNs, BTNs, and BNNs. We remove the bit-extraction overhead and increase bit-level data parallelism by introducing a bitwidth-last data format. We simplify the bitwise multiplications of ternary and binary networks to 5, 4, 3, and 2 bitwise operations by carefully analyzing their computation patterns. Thus, our TAB pushes the boundary of the theoretical speedup of TNNs from $16.0\times$ to $39.4\times$. Experiments on ARM CPU, Intel CPU, and Nvidia GPU show that TAB-TNN achieves up to $34.6\times$ single layer speedup and $13.4\times$ end-to-end speedup on ResNet-18 and Darknet-19.

REFERENCES

- [1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Performance, design, and autotuning of batched GEMM for GPUs. In *International Conference on High Performance Computing*. Springer, 21–38.
- [2] Agner Fog, Technical University of Denmark. 2020. *Instruction tables*. Retrieved from https://www.agner.org/optimize/instruction_tables.pdf.

- [3] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. 2017. Ternary neural networks for resource-efficient AI applications. In *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2547–2554.
- [4] Apache (incubating) TVM. 2021. `tvm.relay.nn.bitserial_conv2d`. Retrieved from https://tvm.apache.org/docs/api/python/relay/nn.html#tvm.relay.nn.bitserial_conv2d.
- [5] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. 2020. wav2vec 2.0: A framework for self-supervised learning of speech representations. *Adv. Neural Inf. Process. Syst.* 33 (2020).
- [6] Joseph Bethge, Marvin Bornstein, Adrian Loy, Haojin Yang, and Christoph Meinel. 2018. Training competitive binary neural networks from scratch. *ArXiv e-prints* (2018). arXiv:1812.01965.
- [7] Peng Chen, Bohan Zhuang, and Chunhua Shen. 2020. FATNN: Fast and accurate ternary neural networks. *arXiv preprint arXiv:2008.05101* (2020).
- [8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 578–594.
- [9] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. 2018. PACT: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085* (2018).
- [10] Lei Deng, Peng Jiao, Jing Pei, Zhenzhi Wu, and Guoqi Li. 2018. GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework. *Neural Netw.* 100 (2018), 49–58.
- [11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
- [12] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630* (2021).
- [13] Yunhui Guo. 2018. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752* (2018).
- [14] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. *Adv. Neural Inf. Process. Syst.* 28 (2015).
- [15] Y. Hu, J. Zhai, D. Li, Y. Gong, Y. Zhu, W. Liu, L. Su, and J. Jin. 2018. BitFlow: Exploiting vector parallelism for binary neural networks on CPU. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 244–253.
- [16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. *Adv. Neural Inf. Process. Syst.* 29 (2016).
- [17] Shubham Jain, Sumeet Kumar Gupta, and Anand Raghunathan. 2020. TiM-DNN: Ternary in-memory accelerator for deep neural networks. *IEEE Trans. Very Large Scale Integ. (VLSI) Syst.* 28, 7 (2020), 1567–1577.
- [18] JDAl Computer Vision. 2020. *daBNN GitHub*. Retrieved from https://github.com/JDAI-CV/dabnn/blob/master/README_CN.md.
- [19] Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. 2019. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4350–4359.
- [20] Bo Kågström, Per Ling, and Charles Van Loan. 1998. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.* 24, 3 (1998), 268–302.
- [21] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. 2019. A study of BFLOAT16 for deep learning training. *arXiv preprint arXiv:1905.12322* (2019).
- [22] Kim Walisch. 2021. `libpopcnt`. Retrieved from <https://github.com/kimwalisch/libpopcnt>.
- [23] Yue Li, Wenrui Ding, Chunlei Liu, Baoshang Zhang, and Guodong Guo. 2021. TRQ: Ternary neural networks with residual quantization. In *AAAI Conference on Artificial Intelligence*.
- [24] Yuhang Li, Xin Dong, Sai Qian Zhang, HaoLi Bai, Yuanpeng Chen, and Wei Wang. 2020. RTN: Reparameterized ternary network. In *AAAI Conference on Artificial Intelligence (AAAI)*. 4780–4787.
- [25] Yinan Li, Jack Dongarra, and Stanimire Tomov. 2009. A note on auto-tuning GEMM for GPUs. In *International Conference on Computational Science*. Springer, 884–892.
- [26] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. 2021. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing* 461 (2021), 370–403.
- [27] Di Liu, Hao Kong, Xiangzhong Luo, Weichen Liu, and Ravi Subramaniam. 2021. Bringing AI to edge: From deep learning’s perspective. *Neurocomputing* 485 (2021), 297–320. DOI:<https://doi.org/10.1016/j.neucom.2021.04.141>
- [28] Hongyang Liu, Sara Elkerdawy, Nilanjan Ray, and Mostafa Elhoushi. 2021. Layer importance estimation with imprinting for neural network quantization. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2408–2417.

- [29] Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. 2018. Bi-Real Net: Enhancing the performance of 1-bit CNNs with improved representational capability and advanced training algorithm. In *European Conference on Computer Vision (ECCV)*. 722–737.
- [30] Dukhan Marat, Wu Yiming, Lu Hao, and Maher Bert. 2019. QNNPACK. Retrieved from <https://github.com/pytorch/QNNPACK>.
- [31] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. 2020. MLPerf: An industry standard benchmark suite for machine learning performance. *IEEE Micro* 40, 2 (2020), 8–16.
- [32] Phuoc Pham and Jaeyong Chung. 2021. Improving model capacity of quantized networks with conditional computation. *Electronics* 10, 8 (2021), 886.
- [33] Hadi Pouransari, Zhucheng Tu, and Oncel Tuzel. 2020. Least squares binary quantization of neural networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 698–699.
- [34] Robert van de Geijn. 2018. *How To Optimize Gemm*. Retrieved from <https://github.com/flame/how-to-optimize-gemm>.
- [35] Haotong Qin, Ruihao Gong, Xianglong Liu, Mingzhu Shen, Ziran Wei, Fengwei Yu, and Jingkuan Song. 2020. Forward and backward information retention for accurate binary neural networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2250–2259.
- [36] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.
- [37] RuiDeng Technologies. 2020. UM25C USB Tester Meter Instructions. Retrieved from <https://phuketshopper.com/software/UM25C/UM25C%20USB%20tester%20meter%20Instructions.pdf>.
- [38] Shufan Wu, Feng Tian, Haihao Shen. 2019. *Deep Learning Performance Boost by Intel VNNI*. Retrieved from <https://www.intel.com/content/www/us/en/artificial-intelligence/posts/deep-learning-performance-boost-by-intel-vnni.html>.
- [39] Szymon Migacz, NVIDIA. 2017. *8-bit Inference with TensorRT*. Retrieved from <https://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf>.
- [40] Mingxing Tan, Ruoming Pang, and Quoc V. Le. 2020. EfficientDet: Scalable and efficient object detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10781–10790.
- [41] THL A29 Limited. 2020. *NCNN GitHub*. Retrieved from <https://github.com/Tencent/ncnn>.
- [42] Bram-Ernst Verhoef, Nathan Laubeuf, Stefan Cosemans, Peter Debacker, Ioannis Papistas, Arindam Mallik, and Diederik Verkest. 2019. FQ-Conv: Fully quantized convolution for efficient and accurate inference. *arXiv preprint arXiv:1912.09356* (2019).
- [43] Diwen Wan, Fumin Shen, Li Liu, Fan Zhu, Jie Qin, Ling Shao, and Heng Tao Shen. 2018. TBN: Convolutional neural network with ternary inputs and binary weights. In *European Conference on Computer Vision (ECCV)*.
- [44] Ziwei Wang, Jiwen Lu, Chenxin Tao, Jie Zhou, and Qi Tian. 2019. Learning channel-wise interactions for binary convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*. 568–577.
- [45] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. 2020. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602* (2020).
- [46] Haojin Yang, Martin Fritzsche, Christian Bartz, and Christoph Meinel. 2017. BMXNet: An open-source binary neural network implementation based on MXNet. In *25th ACM International Conference on Multimedia*. 1209–1212.
- [47] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. LQ-Nets: Learned quantization for highly accurate and compact deep neural networks. In *European Conference on Computer Vision (ECCV)*. 365–382.
- [48] Jianhao Zhang, Yingwei Pan, Ting Yao, He Zhao, and Tao Mei. 2019. daBNN: A super fast inference framework for binary neural networks on arm devices. In *27th ACM International Conference on Multimedia*. 2272–2275.
- [49] Kang Zhao, Sida Huang, Pan Pan, Yinghan Li, Yingya Zhang, Zhenyu Gu, and Yinghui Xu. 2021. Distribution adaptive INT8 quantization for training CNNs. In *AAAI Conference on Artificial Intelligence*.
- [50] Xiangdong Zhao, Ying Wang, Xuyi Cai, Cheng Liu, and Lei Zhang. 2020. Linear symmetric quantization of neural networks for low-precision integer hardware. In *International Conference on Learning Representations*. Retrieved from <https://openreview.net/forum?id=H1lBj2VFPS>.
- [51] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).
- [52] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. 2020. Towards unified INT8 training for convolutional neural network. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1969–1979.
- [53] Shien Zhu, Luan H. K. Duong, and Weichen Liu. 2020. XOR-Net: An efficient computation pipeline for binary neural network inference on edge devices. In *26th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*.

Received July 2021; revised November 2021; accepted December 2021