BitXpro: Regularity-Aware Hardware Runtime Pruning for Deep Neural Networks

Hongyan Li, Hang Lu, Haoxuan Wang, Shengji Deng, and Xiaowei Li, *Senior Member, IEEE*

***Abstract*— Classic deep neural network (DNN) pruning mostly leverages software-based methodologies to tackle the accuracy/speed tradeoff, which involves complicated procedures such as critical parameter searching, fine-tuning, and sparse training to find the best plan. In this article, we explore the opportunities of hardware runtime pruning and propose a regularity-aware hardware runtime pruning methodology, termed "*BitXpro*" to empower versatile DNN inference. The method targets the bit-level sparsity and the sparsity irregularity in the parameters and pinpoints and prunes the useless bits on-the-fly in the proposed *BitXpro* accelerator. The versatility of *BitXpro* lies in: 1) software effortless; 2) orthogonal to the software-based pruning; and 3) multiprecision support (including both floating point and fixed point). Empirical studies on various domain-specific artificial intelligence (AI) tasks highlight the following results: 1) up to 8.27× speedup over the original nonpruned DNN and 10.81× speedup collaborated with the software-pruned DNN; 2) up to 0.3% and 0.04% higher accuracy for the floating- and fixed-point DNNs, respectively; and 3) 6.01× and 8.20× performance improvement over the state-of-the-art accelerators, with 0.068 mm^2 and 74.82 mW (floating point 32) and 40.44 mW (16-bit fixed point) power consumption under the TSMC 28-nm technology library.**

***Index Terms*— Deep learning accelerator, deep neural network (DNN), hardware runtime pruning.**

I. Introduction

LARGE computation intensity is well recognized as one of the main obstacles to deploy deep neural networks (DNNs) into practical applications, because of the rapid evolution of the parameter size from millions (i.e., ResNet [5] family in computer vision) to even hundreds of billions (i.e., BERT [7] or GPT-3 [10] in natural language processing). Although more complex models with enormous layers and complicated neuron connections will benefit the ever-increasing accuracy demand, the real-time performance enhancement, which is the more important and desirable request however, cannot catch up with the development of DNNs, especially for handhelds and cyber-physical devices.

Pruning is universally accepted as an effective way in maintaining the model accuracy and optimizing the computation intensity at the same time. Almost all the conventional pruning methodologies (see [12], [14], [16], [17], and [19]) rely on software-level efforts, which usually consists of the following steps: evaluate the importance of neurons, remove the least important fraction of neurons (contingent to the preset compression ratio), parameter fine-tuning until satisfaction, or get unsatisfactory accuracy that has to change the importance metric and commence pruning again. Generally speaking, software-based pruning has competitive advantages in: 1) obtaining maintained accuracy and controllable compression ratio and 2) easy deployment without considering the underlying hardware (for structured pruning of course). Due to the diversity of deep learning applications, however, it is almost impossible to find a universal software-based pruning method that is applicable to all use cases. A direct consequence is that end users must reconsider the application-specific pruning criteria, in terms of superparameters and DNN structured parameters, and reimplement the above steps from the very inception. The tediously repeated effort limits the fast deployment of DNNs in practical use.

From the model perspective, the DNN itself or its internal sparsity level also impairs the software-based pruning. In specific, pruning leverages the importance metric to identify the least contributive parameters. The metric measures the sparsity variants of the weights or activations, i.e., the average percentage of zeros [12], the absolute value of filters [14], or the entropy of filters [16], trying to eliminate the zero or near-zero variants and retrain the model until the optimal accuracy to justify the employed importance metric. However, one metric may suit certain DNNs very well but might not behave perfectly for others. Besides, the headroom of the sparsity is not always adequate either. Some pruning approaches have to commence retraining to compensate for the information loss or sparse training to manually create more sparsity [17], [19], [24] in the parameter set, which is even more time-consuming and labor-intensive.

From the efficiency perspective, the labor intensity of the software-based pruning also exhibits in the parameter fine-tuning phase. This is because the remaining nonpruned weights cannot always guarantee the initial accuracy of the DNN. Classic procedure hence relies on retraining to redeem the lost accuracy with the same dataset and time-consuming iterations that usually cost days or even weeks according to the equipped GPU facility. The above procedure is usually implemented lay-

Manuscript received 27 May 2022; revised 3 October 2022; accepted 20 October 2022. Date of publication 21 November 2022; date of current version 28 December 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 62172387 and in part by the Youth Innovation Promotion Association of Chinese Academy of Sciences (CAS) under Grant 2021098. An earlier version of this paper was presented in part at the International Conference on Parallel Processing, 2021 [DOI: 10.1145/3472456.3472513]. *(Corresponding author: Hang Lu.)*

Hongyan Li, Haoxuan Wang, and Xiaowei Li are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China.

Hang Lu is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, the Zhongguancun Laboratory, and the Shanghai Innovation Center for Processor Technologies (SHIC), Chinese Academy of Sciences, Beijing 100190, China (e-mail: luhang@ict.ac.cn).

Shengji Deng is with the Second Research Institute of the Civil Aviation Administration of China (CAAC), Beijing 101318, China.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2022.3221732.

Digital Object Identifier 10.1109/TVLSI.2022.3221732

erwise, so if we apply it to VGG-19 [25] for example, we need to retrain the model 19 times with each time iterating tens of epochs to recover the lost accuracy. The long and tedious retraining prevents the instant deployment of the pruned model into the devices and worse still, and if the accuracy is not satiable, it must repeat the same tedious procedure again. Considering other widely used DNNs with hundreds of layers (i.e., ResNet [5] and DenseNet [26]) or even much larger and more complex connections such as 3-D convolution [1], nonlocal convolution [27], or deformable convolution [15], the developers therefore usually face a formidable challenge to obtain both the satisfactory result and the shorter time spent.

From the accelerator perspective, unstructured pruning relies heavily on the underlying hardware. There are plenty of accelerator prototypes proposed to support the particular pruning methodology. For example, Cambricon-S [28] addresses the irregularity of unstructured pruning. Efficient inference engine (EIE) [29] supports the pruning only for the fully connected layers; efficient speech recognition engine (ESE) [30] only focuses on the sparse LSTM model, while the convolutional layers that dominate the CNN inference computation are not supported. Accelerator design also depends on various sparsification methodologies. SCNN [31] exploits the neuron and synapse sparsity, while Cnvlutin [32] only supports neuron sparsity. If the software engineer modifies the pruning policy or simply changes from structured pruning to unstructured pruning, the hardware employed is also about to change, attached with overwhelming transplantation overhead.

Ideally, a pretrained DNN should be pruned as fast as possible for the timely deployment in hardware, and more desirably, the hardware could directly implement runtime pruning without any tedious software-level work to accelerate the DNN inference in a handy and efficient manner. This necessity stimulates us to reconsider the existing classic pruning methodologies and explore a new style to free the developers from the labor-intensive software effort.

Therefore, in this article, we propose *BitXpro*, a hardware runtime pruning methodology to empower versatile DNN inference. Apart from the widely adopted software-based pruning that requires the complex algorithm to identify the trivial values by repeated trial-and-error, *BitXpro* implements the "hardware runtime pruning" by targeting "bits." It pinpoints the essential bits and prunes away the useless bits for acceleration. The useless bits are more easily exposed, especially at the hardware level. For example, the floating-point operand has a long bit width in its mantissa (24 bits) [33], which is always shifted to align the binary point with another mantissa according to the floating-point arithmetic. The shifted binary positions are automatically zero-padded and involved in the floating-point arithmetic. This procedure generates two types of useless bits: the 1st type is the genetic zero bits in the mantissa and the automatically padded zero bits [please see Fig. 2(b)] and the 2nd type is more implicit, that is, the rear bit 1s with extremely trivial significance. As will be shown in Section II, the two types of useless bits both occupy large fractions in the binary-represented weights, which provides a decent condition for *BitXpro* to prune these bits directly in the accelerator at the inference runtime.

Bit pruning inevitably generates more sparsity because more unessential bit 1s are turned into bit 0s. From the accelerator perspective, it creates a fine opportunity to manipulate the newly generated sparsity for faster hardware acceleration. However, the problem that must be tackled is precisely locating the scarce essential bit 1s from the vast amount of unessential bit 0s due to the enlarged sparsity. The difficulty of locating this essential bit 1s stems from the "irregularity" of the sparsity. The distribution of bit 1s is highly arbitrary after pruning. A particular essential bit usually requires meticulous identification in the accelerator. The overhead hence is introduced in the complexity of the circuit and the critical path latency that also makes the accelerator performance stumble.

On top of the precise runtime bit pruning, *BitXpro* is also designed to tackle such sparsity irregularity. The evaluation has shown that the inference speed could be significantly boosted (Section V-B) compared with the regularity-agnostic approach (the "*BitX*" baseline in specific). Most importantly, the whole pruning operation could be implemented on-the-fly in the proposed *BitXpro* accelerator, with lossless accuracy and without any software-related effort.

The contributions of this article are listed as follows.

1) We propose a novel hardware runtime pruning method, termed as *BitXpro*, to empower versatile DNN inference. We highlight the following features of *BitXpro*.
   a) *Software Effortless: BitXpro* directly prunes the original DNN. No retraining, fine-tuning, or special library/framework needed because the method targets the useless binary bits not values.
   b) *Orthogonal to the Existing Software-Pruning Methodologies: BitXpro* implements straightforward bit pruning in the accelerator, so the DNNs, either pruned or nonpruned at the software level, are all suitable for *BitXpro*. In other words, *BitXpro* could further prune the useless bits of the software-pruned DNN and obtain additional speedup.
   c) *Multiprecision Support: BitXpro* applies to not only floating-point DNNs but also fixed-point DNNs, which also demonstrate substantial useless bits. *BitXpro* could accelerate the fixed-point DNNs with even higher accuracy and speedup.
2) We propose a deep learning accelerator capable of unprecedented hardware runtime pruning to mine the maximum potential of *BitXpro*. We highlight the following results.
   a) *Speedup:* Two representatives of *BitXpro*—*BitXpro-mild* and *BitXpro-wild*—could obtain 5.95×–8.16× faster speed over the nonpruned baselines under float-point 32 mode and up to 2.05× under 16-bit fixed-point mode (Sections V-D and V-E). For the object detection model YoloV3, the speedup is up to 8.27× and 10.81× higher over the original model (Section V-F).
   b) *Accuracy:* The accuracy maintains the same or even higher than the baseline for the *BitXpro-mild* and shows a slight loss for the *BitXpro-wild* (Sections V-A and V-D). The accuracy data are reported by the floating-point DNNs. Under 16-bit fixed-point precision, the accuracy behavior is similar. Reporting some of the data: 0.04% higher for ResNext101 for *BitXpro-wild*; 0.15% and 0.01% higher for YoloV3 for the *BitXpro-mild* and the *BitXpro-wild*, respectively (Section V-F). We also present direct visual comparison for some image generation tasks such as CartoonGAN and LapSRN, and the demonstration is extremely identical for *BitXpro* and the vanilla model.
   c) *Accelerator Performance:* We compare the *BitXpro accelerator* performance with other state-of-the-art

TABLE I

WEIGHT/BIT SPARSITY COMPARISON FOR VARIOUS DNNS PRETRAINED WITH IMAGENET DATASET. BIT SPARSITY IS SIGNIFICANTLY MORE ABUNDANT THAN WEIGHT SPARSITY. THE WEIGHTS ARE REPRESENTED BY FP32

Model	Weight Sparity	Bit Sparity
DenseNet121	4.84%	48.64%
ResNet50	0.33%	48.64%
ResNet152	0.75%	48.64%
ResNext50_32x4d	0.37%	48.64%
ResNext101_32x8d	3.43%	48.65%
InceptionV3	0.05%	48.64%
MNASNet0.5	0.00%	48.60%
MNASNet1.0	8.07%	48.98%
MobileNetV2	0.01%	48.67%
ShuffleNetV2_x0_5	0.00%	48.36%
ShuffleNetV2_x1_0	1.53%	48.63%
SqueezeNet1_0	0.05%	48.64%
SqueezeNet1_1	0.02%	48.64%

(SOTA) accelerator prototypes. Equipped with hardware runtime pruning, the accelerator achieves 6.01× and 8.20× performance improvement. The area is 0.068 mm^2 and 74.82 mW [floating-point 32 (fp32)] and 40.44 mW (16-bit fixed point) under the TSMC 28-nm technology library (Section V-G).

d) *Sensitivity Study:* We thoroughly evaluate the sensitivity of the key design parameters to the accuracy and speed (Section V-C).

Powered by the above features, *BitXpro* is designed for flexible and versatile DNN inference at any circumstances. In Section II, we will start by discussing three key observations that justify *BitXpro*.

II. OPPORTUNITIES OF HARDWARE RUNTIME PRUNING

A. Bit-Level Sparsity—The 1st Target

For most of the software-based pruning approaches in the literature [12], [17], [19], [34], the classic procedure basically involves identifying and pruning the trivial "near-zero" parameters. However, as mentioned above, the headroom of the value-level sparsity is very limited. If the compression ratio is mistakenly set, the accuracy loss is inevitable. Under such circumstances, two alternatives are always considered: lower the compression ratio and roll back to the inception [12], [34] or commence sparse training to create more headroom for the employed pruning metric [17], [19]. It is also the root reason why software pruning suffers labor-intensive effort.

In order to circumvent the inconvenience, we reexamine the parameters in-depth. Instead of sticking to the sparse "values," we analyze the more fine-grained bit-level sparsity. As shown in Table I, the "weight sparsity" proportion is obtained by counting the values below 10^{-5} over the total parameter size, while the "bit sparsity" proportion is by counting total bit 0s over the total "bit count" of the mantissas in the parameter set.

Obviously, various benchmark DNNs uniformly demonstrate an obvious gap between the two sides. Most of the weight sparsity results are less than 1%. The bit sparsity, however, is nearly 49% and no exception. It provides an excellent opportunity of exploiting the sparsity at the bit level without resorting to the value-based pruning because ∼49% of bits are already 0s and removing these useless bits off from the multiply-and-accumulation (MAC) computation is definitely harmless to the accuracy. *BitXpro* intends to fully utilize this decent condition to accelerate the DNN inference.

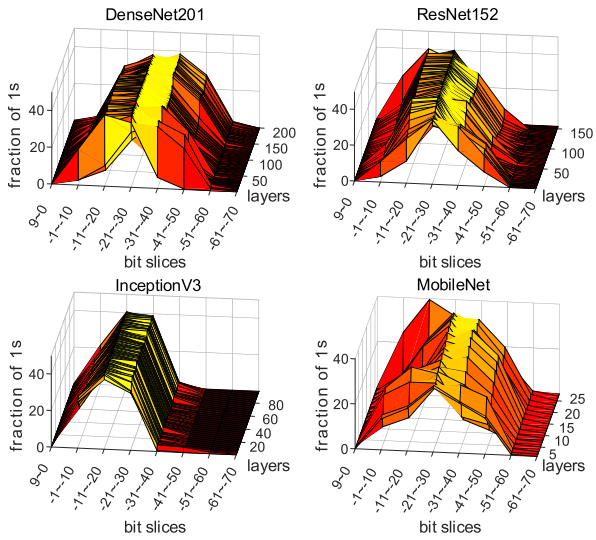

Fig. 1. Distribution analysis of bit 1s. The four benchmark DNNs demonstrate a similar behavior: the surf plot reaches its peak at 2^{-21} to 2^{-30}, which means that this bit slice has the largest fraction of bit 1s (nearly 40%), but most of them are trivial. *BitXpro* aims to prune these trivial bits to obtain the inference acceleration.

B. Trivial Bit 1s—The 2nd Target

Obviously, we can design a particular zero-skipping mechanism to avoid the ineffectual computations caused by the zero bits, which is also the main objective of many previous sparsity-aware acceleration schemes' targets [4], [35], [36], [37]. However, the trivial "bit 1s," as another factor that influences the inference efficiency, are barely considered, but they are exactly the major optimization objective in *BitXpro*. Therefore, having explored the bit-level "sparsity" (or the fraction of bit 0s), we further migrate our focus to the useless "bit 1s." The 49% fraction of 0s also means that the percentage of bit 1s is around 51%, which is also a very large fraction. More importantly, not all the bit 1s are influential to the final accuracy. If we could identify the "essential" bit 1s and prune away the trivial ones, the inference efficiency could be further boosted.

As evidence, we explore the distribution of bit 1s in each bit slice. As shown in Fig. 1, the X-axis denotes the bit slice of the binary represented weight (in fp32). Each bit slice reflects the significance of the bits. For example, if a dummy weight is 1.1101×2^{-4}, its binary representation is 0.00011101 and we record the significance of the 4-bit 1s are the 2^{-4}, 2^{-5}, 2^{-6}, and 2^{-8} after the binary point.

According to Fig. 1, the bit slice could range from bit significance "9–0" before the binary point to "−61 to −70" after the binary point. All the evaluated DNNs exhibit an "arched" shape across each layer on the Y-axis. The central bit slices own most of the bit 1s (∼40%), i.e., ResNet152 and DenseNet201. Taking bit significance 2^{-21} to 2^{-30} as the representative, the equivalent decimals are in the range: 0.000000477 ($\sim 10^{-8}$) to 0.000000000931 ($\sim 10^{-11}$). These tiny values are very likely to be less contributive to the final accuracy. Therefore, *BitXpro* aims to precisely identify the essential bits and prune the large fraction of the trivial bits on

the fly in the accelerator, to reduce the computation intensity under the constraint of tiny accuracy loss.

C. Sparsity Irregularity—The 3rd Target

Even if we could prune the trivial bit 1s to spawn more sparsity, the accelerator design, however, faces a formidable challenge. Intuitively, a bit has a 50% possibility to be 0 or 1, so there is no fixed pattern to determine which bit is essential in the operand. It makes the location of the essential bit very difficult to predict. We term this problem as the "irregularity" of the bit sparsity. The largest concern is its impact on the hardware accelerator. The difficulty of predicting the essential bits complicates the zero-bit skipping mechanism and further the power, performance, and area design tradeoffs.

Aiming at this sparsity *irregularity* problem, a possible solution might be transforming the previous arbitrary-distributed bit 1s into a series of "regular-distributed" bit 1s. Therefore, the accelerator enjoys the predictable pattern of the essential bits neighboring with each other. However, achieving this purpose is not that easy. Simply changing the distribution of the essential bits will inevitably change the original weight value, which will lead to an unpredictable impact on the accuracy of the DNN. Actually, for an fp32 operand, each bit 1 in its binary is associated with an exponent indicating the significance of this bit. For example, a binary weight 1.101 has 2-bit 1s on the right-hand side of the binary point, and the second bit 1 has an exponent of -3 indicating 2^{-3} significance. If we could shift the bit 1 on the 2^{-3} significance to the 2^{-2} significance taking the place of the bit 0, the result is also equal by multiplying an additional 2^{-1} to the associate activation: $1.101^{*}A = A^{*}(2^{0}+2^{-1}+2^{-3}) = A^{*}(2^{0}+2^{-1})+(\mathbf{A}^{*}\mathbf{2^{-1}})\,^{*}\mathbf{2^{-2}}$, in which A is the associate activation. This simple deduction forms a $\mathbf{2^{-2}}$ item by multiplying a $\mathbf{2^{-1}}$ significance to A.

The most important fact of such "essential-bit regularization" is that the previous inconsecutive bit 1s are neighboring this time. The accelerator is more amenable to this scenario because it does not need to employ the complicated mechanisms to spontaneously locate the bit 1s or skip the intermittent zero bits anymore. This operation could be implemented on the fly in the accelerator.

In Section III, we will elaborate on how *BitXpro* is designed to achieve these objectives.

III. BitXpro

A. General Concept

Without loss of generality, a floating-point operand is composed of three portions: the signed bit, mantissa, and exponent, following IEEE 754 [33] which is also the most commonly used floating-point standard in industry. If we employ the fp32 format, the mantissa comprises 23 bits and the exponent occupies 8 bits with the last bit for the sign. A single-precision weight fp could be expressed as $\text{fp} = (-1)^{s}1.m \times 2^{e-127}$, in which e is the actual position of the "binary point" plus 127.

If we take six nonaligned fp32 weights as an example and interpret their mantissas as shown in Fig. 2(a), we get a bit matrix with each column showing the binary mantissa actually stored in memory. Different colors in the legend indicate the bit significance from 2^{0} to 2^{-14} after the binary point (position 0 denotes the hidden 1 of the mantissa [33]). In terms of exponent, we use different background colors in the bit matrix to indicate the actual significance of this bit guided by the exponent. For example, the topmost bit 1 marked as light blue in W_4 is actually the 2^{-3} significance in the fraction.

If we align the mantissas according to the exponents, zeros are padded in the front vacancies just, as shown in Fig. 2(b). The first observation is that the aforementioned bit-level sparsity is more abundant after zero padding, which provides an excellent condition for the bit-level pruning. The second observation is that a large fraction of bit 1s is shifted to the rear direction beyond 2^{-6} significance. A direct consequence is that the practical contribution of these bits is pretty trivial to the final MAC. If we could prune away these insignificant 1s, it could save plenty of bit-level arithmetic and speed up the inference.

As shown in Fig. 2(c), the red box denotes the pruned bit 1s, only leaving several essential bit 1s to form the pruned weights: $W_1^{'}$–$W_6^{'}$, and we term these 1s as the "essential bits." Pruning out the nonessential bits creates more bit-level sparsity, i.e., for the 2^{-4}, 2^{-6}, and 2^{-8}–2^{-14} rows in Fig. 2(c). The original bit 1s turn into bit 0s after pruning. Omitting these pruned all-zero-bit rows, the bit matrix is then shown in Fig. 2(d). As the unessential bit rows are pruned, the MAC computation could be completed in four cycles, compared with the original eight cycles in Fig. 2(a).

Although tremendous MACs are eliminated, there still leaves a problem regarding efficient accelerator design. Taking W_2' in Fig. 2(d) as an example, there are three essential bit 1s in the column. The location (or the significance) of each bit 1, however, is hardly predicted in advance, which means the accelerator must traverse the bits one after another to locate the 3-bit 1s from the other five bit 0s. Worse still, other weights, such as $W_1^{'}$ or $W_6^{'}$ in the figure, encounter the similar situation, whose essential bits are arbitrarily distributed as well.

This aforementioned "irregularity" problem naturally exists because either the remained bits or the pruned bits cannot be precisely predicted under any circumstances. Therefore, a straightforward solution might be making the sparsity from irregular to "regular." Still taking W_2' as an example, the "irregular" MAC is $A_2\ ^{*}\ (2^{-1}+2^{-2}+2^{-7})$. Obviously, the essential bits become continuous if the 2^{-7} bit is shifted to the 2^{-3} significance. The MAC result remains equivalent by multiplying an additional 2^{-4} to A_2. We term such operation as "bit regularization," as shown in Fig. 2(e). In the figure, the matrix after bit regularization only retains the essential bits, and they are organized more regular than Fig. 2(d). Such regularity contributes to the efficient inference in the accelerator because the zero-bit skipping mechanism is no longer needed and the MAC computation is able to accomplish in controllable and predictable cycles.

B. Methodology

Leveraging the essential bits in Fig. 2 is an effective way to simplify a series of MACs into bit-level arithmetic [35], [38]. However, for the millions of parameters in DNN, if we use fp32 to represent these values, the impact of a single bit on the whole network is not that easy to be determined. Therefore, the leftover problem is how to create an effective yet hardware-friendly mechanism to pinpoint the useless bits while maintaining the initial accuracy, without labor-intensive software tricks. In this section, we first formalize the problem and then elaborate on the *BitXpro* procedure.

1) Problem Formulation: Given an $n \times l$ matrix A (activations) and an $l \times n$ matrix W (weights), the result of $A \times W$ can be represented by the summation of n rank-one

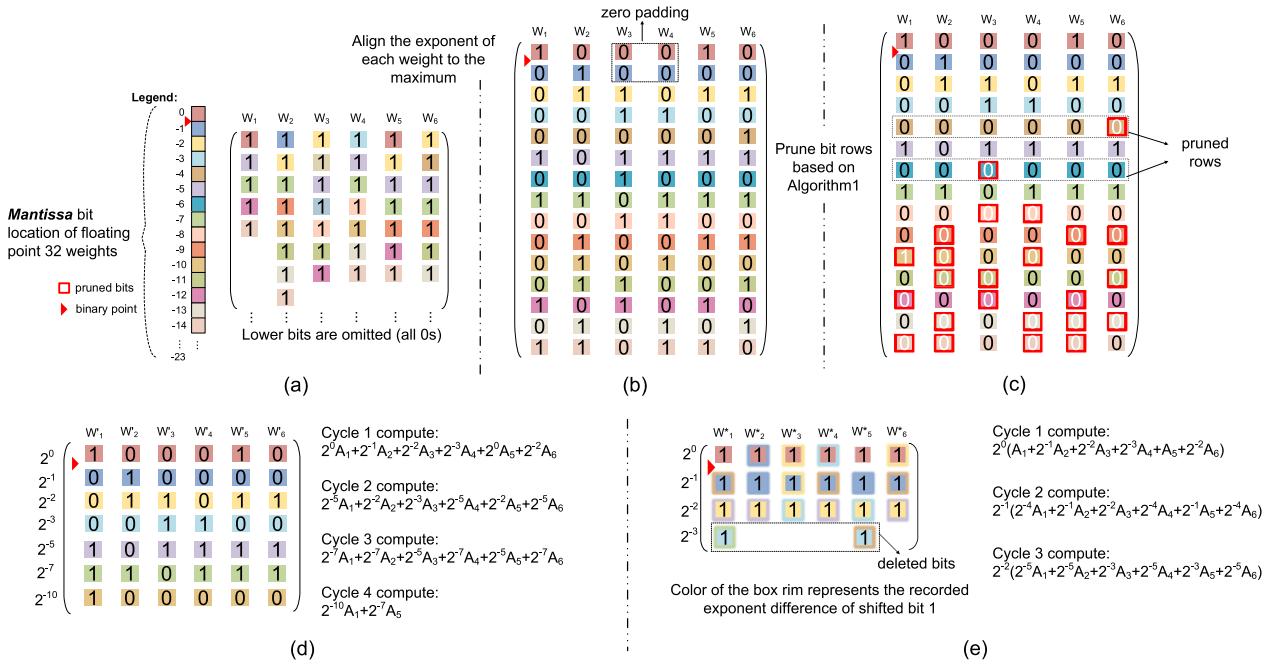

Fig. 2. Core concept of *BitXpro*. (a) Bit matrix before pruning (original weight matrix). (b) Exponent alignment according to IEEE 754. (c) Six nonessential-bit rows are pruned. (d) Only leaving a more compact essential-bit matrix (bit matrix after pruning). (e) Our special treatment regarding the "irregularity" of the remaining essential bit 1s, termed "bit regularization."

matrices: $A^{(i)}$ represents the ith row of A and $W_{(i)}$ for the ith column of W, as shown in (1). The criticality of these rank-one matrices could be easily decided by the fast Monte Carlo algorithm [39], in which some rank-one matrices are randomly sampled to approximate $A \times W$. The most common sampling method [39] to select these rank-one matrices is by referring to their respective probability as shown in (2). It is obtained by computing the Euclidean distance of $A^{(i)}$ and $W_{(i)}$, which reflects the importance of the rank-one matrix multiplication

$$p_i = \frac{\left|A^{(i)}\right|\left|W_{(i)}\right|}{\sum_{i'=1}^{l}\left|A^{(i')}\right|\left|W_{(i')}\right|}. \tag{1}$$

Inspired by the fast Monte Carlo algorithm, we enroll the same probability concept in *BitXpro* to measure the importance of the weight *bits* instead of values. Bits with smaller probability tend to play a trivial role when multiplied with the activations compared with other more important bits in the same weight. Therefore, we abstract the bit matrix in Fig. 2(a) as W and our objective is seeking out the (in)significant bit rows in Fig. 2(b) and simplifying MAC computations. The problem remains how we can utilize the probability in (1) to sample each bit row in W and determine the to-be-pruned bit rows.

2) Bit-Slice Extraction: In W [Fig. 2(a)], we target the mantissa of n normal fp32 weights. Each mantissa is instantiated as a column vector comprised of its bits. Obviously, n weights are associated with the same number of activations for MAC. N activations consist of another column vector $[A_1, A_2, \ldots, A_j, \ldots, A_n]^T$. We put the two column vectors into (1), so it could be rewritten as follows:

$$p_i = \frac{\left|A^{(i)}\right| \times \sqrt{\sum_{j=1}^{n}\left(2^{E_i^j} \times v_j\right)^2}}{\sum_{i'=1}^{l}\left(\left|A^{(i')}\right| \times \sqrt{\sum_{j=1}^{n}\left(2^{E_{i'}^j} \times v_j\right)^2}\right)}. \tag{2}$$

A_j is the element of the activation vector, and v_j is the jth bit of the ith row vector in the bit matrix W. Each bit in the same row i has its own exponent, so we use E_i^j in (2) to represent the exponent at position j. The Euclidean distance of the row vector is calculated as $(\sum_{j=1}^{n}(2^{E_i^j} \times v_j)^2)^{1/2}$.

In *BitXpro*, the exponent alignment procedure is almost identical to the normal floating-point addition [33], except for one special difference, that is, *BitXpro* does not implement weight/activation MAC one by one. Instead, it aligns a group of weights simultaneously to the maximum exponent. Therefore, after exponent matching, the bits in the same row i share the same exponent just as Fig. 2(b) has shown, and we use a uniform E_i to denote the actual exponent of the row i.

v is a row vector in W composed of bits. For the case that a bit element v_j equals 0, there is obviously no impact on calculating the Euclidean distance and thus no impact on p_i. Therefore, acquiring the Euclidean distance is equivalent to counting the number of bit 1s in row i. We use $\mathrm{BitCnt}(i)$ to indicate such operation, so the probability p_i of the ith row can be represented as follows:

$$\begin{aligned} p_i &= \frac{\left|A^{(i)}\right| \times \left(\sqrt{\left(2^{E_i}\right)^2 \times \mathrm{BitCnt}(i)}\right)}{\left|A^{(i')}\right| \times \sum_{i'=1}^{l}\left(\sqrt{\left(2^{E_{i'}}\right)^2 \times \mathrm{BitCnt}(i')}\right)} \\ &= \frac{\sqrt{\left(2^{E_i}\right)^2 \times \mathrm{BitCnt}(i)}}{\sum_{i'=1}^{l}\sqrt{\left(2^{E_{i'}}\right)^2 \times \mathrm{BitCnt}(i')}}. \end{aligned} \tag{3}$$

In (3), E_i stands for the aforementioned exponent of the ith row. Each column vector in matrix A is the same, so $|A^{(i')}|$ equals $|A^{(i)}|$. For the given W with l column vectors, $\sum_{i'=1}^{l}|W_{(i')}|$ is a constant, so we let $C = \sum_{i'=1}^{l}((2^{E_{i'}})^2 \times \mathrm{BitCnt}(i'))^{1/2}$, and the final p_i is deduced by the following

Algorithm 1 Pruning

Input: n number of fp32 weights, and bit-sparsity level N
Output: essential bit matrix W'

1: Interpret n exponents $E = [e_1, \ldots, e_n]$ and mantissas $M = [m_1, \ldots, m_n]$;
2: Adding the hidden '1' and obtain $M' = [m'_1, \ldots, m'_n]$;
3: Align $M'$with $e_{max} = max(E)$ and obtain updated M'
4: Obtain bit matrix W with each $m' \in M'$ in column # *Figure 2(a)*
5: $r, c = W$.shape # *matrix row and column*
6: $mask = zeros(r)$; # *initialize* $mask_{r\times 1}$ *with 0*
7: **foreach** row i **in** W: # *iterate each row in W*
8: $E_i = e_i - e_{max}$
9: $p_i = 2^{E_i} \times \sqrt{BitCnt(i)}/C$ # *Eq.(4)*
10: $P.append(p_i)$ # *P stores each* p_i
11: $I, P' = sort(P)$; # *sort P in descending order, obtain index vector I*
12: $I' = max(I, N)$; # $I' = [i_1, \ldots, i_N]$, *obtain the first N indices*
13: **foreach** index i **in** I':
14: $mask[i] = 1$
15: $W \leftarrow W \otimes mask$ # *prune W with mask, Figure 2(c)*
16: n = 0
17: [1]**foreach** each row i **in** W:
18: **foreach** column j **in** W:
19: **if** $W(i, j) == 1$:
20: $W'(i, n) = 1$
21: n += 1
22: n = 0 # *reset n for the next row*
23: **return** W' # *essential bit matrix*

[1]The outer "**foreach**" loop could be parallelized in hardware.

Algorithm 2 Regularization

Require: essential bit matrix W'
Ensure: compact bit matrix W^*

1: Move bit 1s in W' to the former 0 bits in a same column and record
2: changes on exponents
3: **foreach** row i **in** W^*:
4: if $E_i < \varepsilon$ and $BitCnt(i) < \theta$:
5: $W^*[i] = 0$ # *all bits in the row i are set to 0 for regularity*
6: **else:**
7: $W^*[i] = W^*[i]$
8: **return** W^* # *regularized bit matrix*

equation:

$$p_i = \frac{\sqrt{\left(2^{E_i}\right)^2 \times \text{BitCnt}(i)}}{C}. \tag{4}$$

Discussion: The probability p_i reveals the magnitude of "*significance*." This is reasonable because E_i reflects the bit significance of row i and BitCnt(i) reflects the number of essential bit 1s in row i. Larger E_i or BitCnt(i) definitely leads to more contribution on the final MAC. *BitXpro* takes advantage of (4) to pinpoint the essential bit rows and, in the meantime, prunes away the trivial bit rows directly in the accelerator.

C. BitXpro Procedure

BitXpro includes two-level procedures.

1) Pruning: Algorithm 1 serves as the 1st level of the procedure, termed pruning. It first interprets the exponent E and mantissa M of n fp32 weights as input (lines 1–3), aligns each exponent according to $e_{\max}$ (line 4), and then calculates and sorts the row probabilities in descending order (lines 5–12). For the other input parameter N, it denotes the remaining bit rows in W after pruning. In other words, *BitXpro* selects the top n bit rows with relatively larger p_is. The indices of the n rows are reflected in I' (lines 14). The pruning is finalized by the vector mask with the selected n bit rows marked as "1" (lines 7 and 16). Right after pruning, *BitXpro* extracts the essential bits and stores them into W' (lines 17–25).

2) Regularization: Algorithm 2 serves as the 2nd level of the procedure, termed regularization. The essential bit matrix W', the output of Algorithm 1, is now the input of Algorithm 2. Lines 1 and 2 first move the essential bits in W' to former positions following the concept of Fig. 2(e).

In Fig. 2(e), the 2^{-3} row in W^* only contains two essential bits obtained from Algorithm 1. Intuitively, if this row is pruned, the whole matrix becomes regularized. However, we cannot simply eliminate the two bits without verifying their significance to the accuracy. As the second step of the regularization procedure, line 4 instantiates two design parameters, ε and θ, to decide whether each row in W^* could be further pruned or not for the regularization purpose. The tail rows of the matrix usually have smaller exponent [i.e., the 2^{-3} row in Fig. 2(e)], which means that the significance to the accuracy might also be trivial. ε thereby serves as a threshold to decide the relationship with the exponent of each row. θ, on the other hand, denotes the threshold of the essential bits. They collaboratively determine the regularized W^* (lines 4–8).

Discussion: If the regularity is not considered, the accuracy of pruning is also guaranteed by only performing Algorithm 1, even if the accelerator performance is suboptimal. We call this simplified version of *BitXpro* as *BitX*, which is published in our previous literature [40], and this article is an extended work of [40]. Clearly, the difference is that *BitX* has no regularization procedure. Section V-D will demonstrate their performance comparison and prove the efficacy of the regularization on improving the acceleration speedup.

Besides, we have three design parameters in *BitXpro*, including the reserved bit rows N in Algorithm 1 and ε and θ in Algorithm 2. They all control the granularity of the pruning. For example, the smaller N and θ and larger ε combination will lead to a higher speedup in the accelerator because more bit rows are pruned. However, the accuracy may also degrade. In Section V, we will thoroughly study the sensitivity and tradeoff of these design parameters and their impact on the performance of *BitXpro*.

IV. BitXpro Accelerator Architecture

BitXpro is a hardware runtime pruning approach. Specialized modules for pruning are integrated in the accelerator hardware design. The overall accelerator architecture is shown in Fig. 3. Two modules "E-alignment" and "bit-extraction" are designed to perform Algorithm 1. "Bit-shifting" module is designed for Algorithm 2, and the *BitX* implementation does not involve this module. We instantiate 16 computing units to form one *BitXpro* or *BitX* processing elements (PE). Each "compute unit" (CU) takes M weights/activation pairs [indexed as 0–$(M-1)$] as input.

The "E-alignment" module performs the steps in Fig. 2(a) and (b) by aligning the exponents of all the weights to their maximum. The aligned weights are then preprocessed by the "bit-extraction" module with the trivial bits pruned to 0 [performing Fig. 2(c) and (d)]. The pruned essential-bit matrix is finally regularized by the "bit-shifting" module [performing Fig. 2(e)]. Each CU executes the bit-level MAC according to the regularized W^* in Fig. 2(e) to finalize the convolution arithmetic.

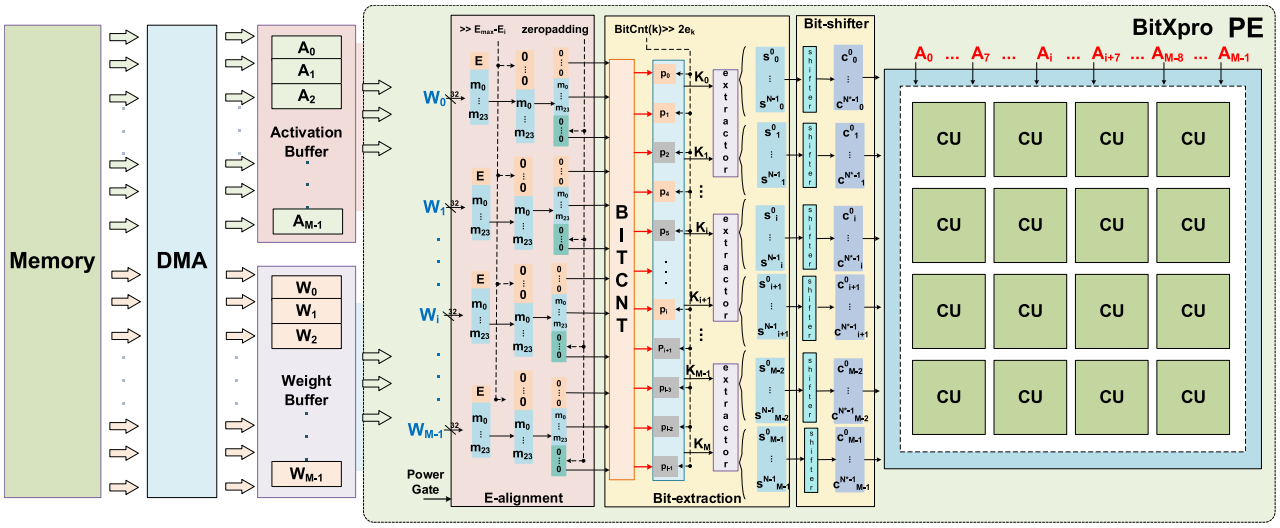

Fig. 3. Overall architecture of the BitXpro accelerator.

On our field-programmable gate array (FPGA) platform, the memory access is through DMA with two state machines to coordinate the data fetch and store, associated with the weight and activation buffer. The detailed hardware configurations are elaborated in Section V.

For the fixed-point DNN, the E-alignment module is bypassed and safely clock gated. The original weights are directly connected to the input of the bit-extraction module because the fixed-point arithmetic does not involve exponent matching. Clock gating is favorable to significant energy savings in the fixed-point mode, as shown in Section V-G.

A. E-Alignment

E-alignment module is designed to align the exponent of each weight uniformly to the maximum. It is mainly comprised of the data shifter and zero padding. First, the weight is split into the corresponding exponent and mantissa (for the floating-point data). Then, the maximum exponent $E_{\max}$ is obtained and stored. The exponents of all weights are aligned to this maximum value following Algorithm 1. The data shifter performs this operation through right shifting the ith mantissa by $E_{\max} - E_i$. The shifted vacancies are zero-padded in the front part of the mantissa, marked as orange in Fig. 3. For different weights, E_i is possibly not identical, so we will obtain arbitrary bit widths after zero padding. To deal with this scenario, this module also pads a series of zero bits to the maximum bit width, marked as green in the figure. Although zero padding is frequent in this module, our register transfer level (RTL) implementation could easily hardcode this operation without violating the timing constraint. The only overhead introduced is the complicated wire organization that might potentially increase the circuit area.

B. Essential-Bit Extraction

The padded mantissa from the E-alignment module is then delivered to the bit-extraction module for the actual pruning. The 1st functionality in this module is the BITCNT, which is designed to implement the BitCnt function in (4). In our FPGA implementation, the $(2^{E_i})^2 \times \text{BitCnt}(i)$ operation inside square root calculations (SQRT) could be equalized as shifting $\text{BitCnt}(i)$ by $2E_i$. SQRT is not necessary because it will not influence the final significance ranking. Therefore, only combinatorial circuits could fulfill this purpose. The 2nd functionality of the bit-extraction module is sorting the shifted $\text{BitCnt}(i)$ and selecting the top n largest rows, while the disqualified rows are completely pruned.

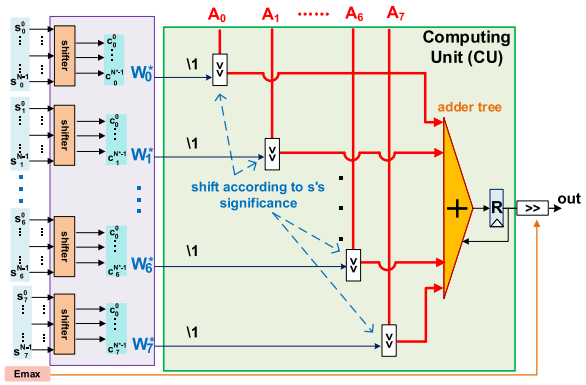

Fig. 4. Microarchitecture of the "CU."

C. Bit Shifter

This module performs the regularization procedure in Algorithm 2. The shifter in Fig. 3 neighbors the essential bits as the first step of the regularization. Each shifter targets one pruned weight from the previous bit-extraction module (M weights in total), and k_i denotes each bit i in the pruned weight. As the second step, the significance of each k_i is also recorded (s_i in the figure) for multiplying into each associate activation during MAC, marked by different colors on the box rim in Fig. 2(e). Finally, the tail bit rows with BitCnts and E below the threshold are directly pruned (line 4 in Algorithm 2), leaving the whole bit matrix finally regularized.

D. Compute Unit

The microarchitecture of CU is shown in Fig. 4. Equipped with the regularization performed by the previous "bit shifting," CU no longer deals with the sparsity. The MAC operation is implemented bit serially, with one cycle accomplishing one essential-bit MAC.

The activation A_i could be either floating-point or fixed-point datum. The fixed-point A_i could be directly shifted and accumulated for MAC as shown in the figure. However, for the floating point A_i, the significant loss due to regularization must be multiplied by each A_i, which is the fixed-point arithmetic of the exponent. Therefore, the operation will not introduce severe overhead. The adder tree performs the final partial-sum accumulation. It distinguishes the precisions, so the overall power consumption of CU is also distinct under different precision configurations. Section V-G decomposes the power consumption of the *BitXpro* accelerator in the hardware implementation to present a comprehensive study.

V. Evaluation

Benchmark: We use plenty of domain-specific DNNs as the benchmark. The parameters pretrained by various datasets are directly obtained from PyTorch [41]. The image recognition benchmark models involve "big" models with the parameter size ranging from 76.35M (DenseNet201) to 356.71M (ResNext101_32 × 8d), as well as "little" models with the parameter size of 4.71M (SqueezeNet1_1). YoloV3 [42], PRN [6], and FCOS [9] trained on the CoCo [8] dataset are employed to evaluate the performance on the object detection task. Besides, we also employ 3-D convolution, deformable convolution, transformer, and GAN model to evaluate our approach.

Hardware Implementation: At the RTL level, we employ Vivado HLS (v2018.2) to conduct postsynthesis simulation on Xilinx Virtex-7 FPGA. The actual inference time is recorded at each run. We instantiate 16 CUs in PE, clocked at 200 MHz. Runtime memory access data of our FPGA platform are recorded and then fed to the DRAMsys tool [43] to estimate the energy consumption of the memory accesses. For the RTL synthesis, Synopsys Design Compiler (v2016) is used to measure power and area. The frequency is set to 1 GHz. The whole design is synthesized with the TSMC 28-nm technology library.

BitXpro Specifics: The configurations of M and N in Algorithm 1 and θ and ε in Algorithm 2 directly affect the performance of *BitXPro*. We choose several discrete values for the design space exploration, to explore the sensitivity of these design parameters to the accuracy and speed. Also, based on these parameters, we define two representatives—*BitX/BitXpro-mild and BitX/BitXpro-wild to tackle various performance tradeoffs.*

A. Accuracy and Sparsity

Table II shows the accuracy/sparsity results for *Cifar-10* dataset, grouped by the parameter N. Smaller N means that more bit rows are pruned, so the bit-level sparsity also turns larger. For example at $N = 4$, the sparsity increases to 1.80× compared with the original model. More sparsity is undoubtedly beneficial to the inference speedup (as proved in Section V-B). On the other hand, larger N means that less bit rows are pruned so the sparsity only shows 1.41× and 1.53× for $N = 10$ and $N = 8$, respectively.

The results for *ImageNet* dataset shown in Table III exhibit a similar trend as in Table II: less than 0.5% average accuracy loss at $N = 10$, 8, and 6, and 1.40×, 1.52×, and 1.66× sparsity increment apiece.

Discussion: First, it proves that the proposed *BitXpro* pruning methodology will not affect the accuracy of DNNs. The average accuracy loss is less than 0.5% at $N = 10$, 8, and 6, for both Cifar-10 and ImageNet datasets. Second, this experiment demonstrates the tradeoff between accuracy and sparsity. A borderline configuration also exists with the maintained accuracy and satisfied sparsity. As shown in Tables II and III, there is a significant accuracy drop at $N = 4$ and $N = 6$. Therefore, we can safely choose N values in the range from 10 to 6 in *BitXpro*. This experiment also verifies that there are tremendously redundant bits in the parameters that can be safely pruned without hurting the accuracy.

TABLE II

Cifar-10 Performance. The Last "Avg." Row Denotes the "Accuracy Loss/Sparsity Increment." The Accuracy Loss Is Obtained by Itemizing the Accuracy Loss of Each Benchmark Model Versus the Original and Calculate Their Average (in %). The Sparsity Increment Is Obtained by Counting the Bit 0s After Pruning and Normalizing the Data to the Original (in ×)

Model	Original	N=10	N=8	N=6	N=4
DenseNet121	95.25/1x	95.21/1.36x	95.19/1.49x	95.10/1.63x	93.17/1.77x
DenseNet161	95.66/1x	95.55/1.33x	95.52/1.47x	95.52/1.61x	93.98/1.76x
DenseNet169	95.50/1x	95.48/1.33x	95.49/1.47x	95.44/1.61x	93.28/1.76x
Densenet201	95.35/1x	95.39/1.31x	**95.35**/1.45x	95.25/1.60x	93.71/1.75x
ResNet18	95.18/1x	94.96/1.66x	94.90/1.75x	94.80/1.83x	93.69/1.91x
ResNet34	95.33/1x	95.27/1.66x	95.27/1.74x	95.20/1.83x	93.69/1.91x
ResNet50	95.14/1x	95.07/1.42x	95.04/1.53x	95.07/1.66x	91.79/1.78x
ResNet101	95.51/1x	95.34/1.44x	95.35/1.56x	95.39/1.69x	91.46/1.81x
ResNet152	95.56/1x	95.41/1.45x	95.47/1.57x	95.36/1.69x	90.48/1.81x
ResNext29_2x64d	95.82/1x	95.71/1.45x	95.73/1.57x	95.71/1.69x	91.95/1.81x
ResNext29_4x64d	95.69/1x	95.57/1.37x	95.55/1.50x	95.50/1.64x	92.82/1.78x
ResNext29_8x64d	96.19/1x	96.13/1.31x	96.16/1.45x	96.08/1.60x	93.57/1.75x
ResNext29_32x64d	95.61/1x	95.56/1.25x	95.55/1.40x	95.48/1.56x	89.47/1.73x
Avg. loss / sparsity	**0.000/1x**	**0.090/1.41x**	**0.094/1.53x**	**0.145/1.66x**	**2.979/1.80x**

TABLE III

ImageNet Performance. The Computing Method Is Identical to Table I

Model	Original	N=10	N=8	N=6	N=4
DenseNet121	71.96/1x	71.95/1.34x	71.00/1.47x	71.00/1.62x	65.00/1.76x
DenseNet161	75.28/1x	75.20/1.32x	75.14/1.46x	74.79/1.61x	72.00/1.76x
DenseNet169	73.75/1x	73.56/1.31x	73.55/1.45x	73.55/1.60x	68.62/1.75x
Densenet201	74.56/1x	74.46/1.30x	74.40/1.44x	74.24/1.59x	69.00/1.74x
ResNet18	67.28/1x	67.09/1.64x	67.00/1.73x	66.72/1.81x	62.52/1.90x
ResNet34	71.32/1x	71.11/1.65x	71.10/1.73x	70.92/1.82x	68.00/1.90x
ResNet50	74.50/1x	74.50/1.41x	**74.51**/1.54x	74.10/1.67x	67.00/1.80x
ResNet101	76.00/1x	76.06/1.43x	76.05/1.55x	75.76/1.68x	69.02/1.81x
ResNet152	77.02/1x	76.56/1.44x	76.55/1.56x	76.46/1.69x	72.30/1.81x
ResNext50_32x4d	76.29/1x	75.99/1.24x	75.96/1.39x	75.67/1.56x	65.01/1.72x
ResNext101_32x8d	78.24/1x	78.20/1.27x	**78.30**/1.42x	78.10/1.58x	73.00/1.74x
SqueezeNet1_1	54.84/1x	**54.86**/1.42x	54.70/1.54x	54.40/1.67x	47.30/1.80x
Avg. loss / sparsity	**0.000/1x**	**0.131/1.40x**	**0.242/1.52x**	**0.444/1.66x**	**6.023/1.79x**

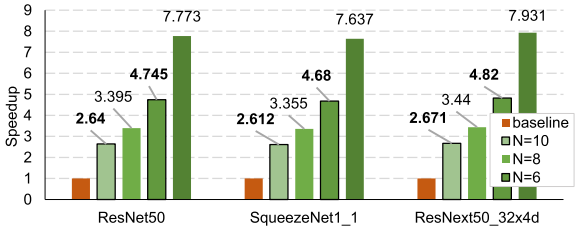

Fig. 5. Inference speedup comparison under different N settings.

B. Speedup

We first evaluate the inference speed at different sparsity levels indicated by the N configuration. As a hardware runtime pruning approach, the speedup contains the time of both "runtime pruning" of the vanilla model and the "inference" using the pruned model. The data are recorded according to the actual runtime on our Xilinx V7 FPGA platform and normalized to the original nonpruned DNN. As shown in Fig. 5, *BitXpro* exhibits ∼2.6× speedup at $N = 10$ and

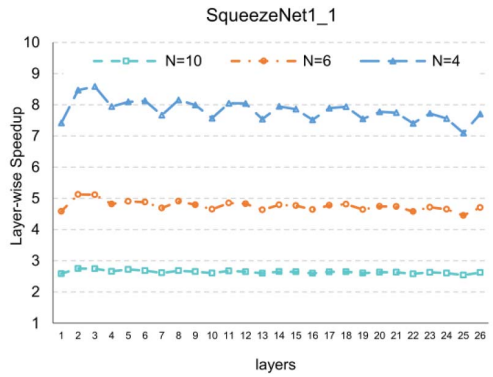

Fig. 6. Layerwise speedup for *ImageNet* dataset. The speedup of the original model is regarded as 1 on the Y-axis.

$\sim$4.8$\times$ speedup at $N = 6$. The above datum is obtained under channel granularity. Fig. 6 and 7 complement the layer-wise speedup of *BitXpro*, which also show the well accelerating result. The promising speedup stems from the enriched bit sparsity enforced by *BitXpro*. More abundant sparsity means less essential-bit computations in the accelerator and thus leads to much faster inference speed.

Discussion: The *BitXpro* accelerator directly integrates the pruning module in hardware and executes the hardware runtime pruning during inference. This is totally different from software-based pruning that targets value sparsity to acquire the reduced parameter size and FLOPs. *BitXpro* leverages the abundant useless bits and bit sparsity irregularity to directly accelerate the original DNN after deployment and does not involve any software work. The higher speedup and lossless accuracy can provide attractive convenience for the end users to deploy their models into products much faster.

C. Design Space Exploration

*1) Pruning Granularity (*N*) and Throughput (*M*):* Sections V-A and V-B have evaluated the sensitivity of N and its impact on the *BitXpro* performance. We further explore the impact of another key parameter M in this experiment. We use four DNNs trained with the *ImageNet* dataset, as shown in Table IV. M indicates the number of input weights that the accelerator could simultaneously prune (Fig. 3); generally speaking, M barely influences the overall accuracy scaling from 8 to 512 for all the four DNNs. For example, in ResNet50, the accuracy at $M = 8$ is lower than the accuracy at $M = 16$ but is higher than the accuracy at $M = 32$ or 64. For SqueezeNet1_1, the accuracy at $M = 8$ (54.86%) is even higher than the original model accuracy (54.84%). The average accuracy loss at other M configurations is less than 0.3%. We conclude that the number of simultaneous input weights has a negligible impact on the performance of *BitXpro*.

Discussion: The major factor that steers the accuracy and speedup is the N configuration. Table IV shows that at different scales of M, the accuracy consistently degrades from $N = 10$ to $N = 4$, which is in line with the observation in Table III. It is N that decides the granularity of pruning, while M only controls the input throughput.

Two BitXpro Instances: As discussed above, M barely influences the accuracy, so we choose $M = 8$ for the efficient accelerator implementation. Upon $M = 8$, we select two N settings: $N = 10$ and $N = 6$ to form two *BitXpro* instances, termed *BitXpro-mild* ($N = 10$ and $M = 8$) and *BitXpro-wild* ($N = 6$ and $M = 8$). *BitXpro-mild* has the better accuracy but limited speedup, while *BitXpro-wild* has little-degraded accuracy but relatively abundant speedup. A similar configuration is also set for the simplified *BitX* version, termed *BitX-mild* and *BitX-wild*. In the next set of evaluations, we show the results of both *BitX* and *BitXpro*.

TABLE IV
DESIGN SPACE EXPLORATION OF TWO KEY DESIGN PARAMETERS M AND N ON *ImageNet*

ResNet50	**Original Accuracy: 74.50**			
	N=10	N=8	N=6	N=4
M=8	74.50	**74.51**	74.10	67.00
M=16	**74.54**	74.40	73.60	61.00
M=32	74.00	74.50	73.50	58.20
M=64	74.39	74.00	73.00	53.30
M=128	74.41	74.32	72.70	46.30
M=256	**74.51**	74.40	72.80	46.80
M=512	74.30	74.26	71.90	39.4
ResNext101_32x8d	**Original Accuracy: 78.24**			
	N=10	N=8	N=6	N=4
M=8	78.20	**78.30**	78.10	73.00
M=16	78.20	78.00	77.50	66.00
M=32	78.20	78.00	78.20	65.00
M=64	78.20	78.20	77.30	62.00
M=128	78.20	78.10	77.30	57.00
M=256	78.20	78.20	77.20	49.00
M=512	78.20	78.20	77.20	49.00
DenseNet121	**Original Accuracy: 71.96**			
	N=10	N=8	N=6	N=4
M=8	71.95	71.00	71.00	65.00
M=16	**71.97**	72.00	71.00	62.00
M=32	**72.03**	72.00	71.00	58.00
M=64	71.00	71.00	70.00	55.00
M=128	71.00	71.00	70.00	49.20
M=256	71.84	71.00	69.00	49.00
M=512	71.83	71.6	69.00	34.00
SqueezeNet1_1	**Original Accuracy: 54.84**			
	N=10	N=8	N=6	N=4
M=8	**54.86**	54.70	54.40	47.30
M=16	54.80	54.74	53.00	41.60
M=32	54.00	54.50	53.64	41.70
M=64	54.70	54.77	53.50	37.10
M=128	54.40	54.48	52.80	34.66
M=256	54.62	54.4	52.11	32.10
M=512	54.81	54.72	52.60	32.00

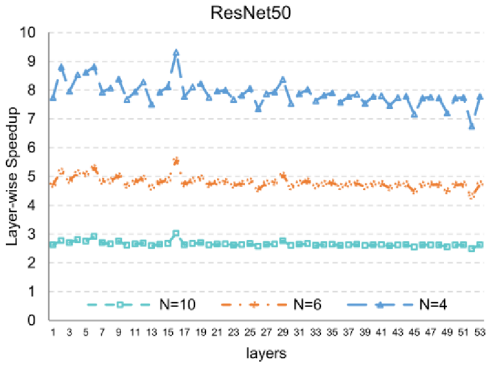

Fig. 7. Layerwise speedup for *ImageNet* dataset. The speedup of the original model is regarded as 1 on the Y-axis. Higher is better.

2) Regularization (ε and θ): For the *BitXpro*, there are two extra design parameters used for the regularization—ε and θ. We employ two DNNs, FCOS and LapSRN, to explore the design space. As shown in Fig. 8, the horizontal and vertical axes, respectively, represent the parameter scaling and the corresponding performance. The datum on each mark denotes the corresponding speedup.

ε represents the significance of the exponent. With ε tuned from -7 to -5, both PSNR and mAP decrease with θ scaled larger. We also observed similar behavior when ε was configured with other values. *BitXpro-mild* is more robust and

TABLE V

ACCURACY AND SPEEDUP EVALUATION OF *BitX*/*BitXpro*. NOTE THAT WE PRESENT THE DIRECT VISUAL COMPARISON FOR THE IMAGE GENERATION TASK—CARTOONGAN AND LAPSRN, SO ITS METRIC VALUE IS NOT LISTED IN THIS TABLE

Model	Domain	Type	Dataset	Metric	Baseline	*BitX-mild*	***BitXpro-mild***	*BitX-wild*	***BitXpro-wild***
C3D [1]	Video Understanding	3D Convolution	UCF101 [3]	Top_1(%)	98.97/1x	97.27/2.30x	98.54/5.95x	95.11/3.73x	97.44/**8.14x**
PRN [6]	Pose Estimation	2D Convolution	CoCo [8]	mAP	0.89/1x	**0.89**/2.32x	**0.89**/6.00x	**0.89**/3.76x	**0.89**/**8.16x**
FCOS [9]	Object Detection	Feature Pyramid	CoCo [8]	mAP	0.38/1x	0.38/2.30x	0.38/5.98x	0.35/3.73x	0.37/**8.16x**
Vit [11]	Video Understanding	Transformer	ILSVRC2012 [13]	Top_1(%)	83.89/1x	**84.19**/2.31x	**84.03**/5.98x	**84.00**/3.75x	**84.19**/**8.17x**
D3DNet [15]	Video Super Resolution	3D Deformable	Viemo-90k [18]	PSNR	36.05/1x	35.96/2.29x	36.03/5.91x	35.67/3.71x	35.77/**8.06x**
				SSIM	0.94	**0.94**	**0.94**	0.93	0.93
LapSRN [20]	Image Super Resolution	2D De-Convolution	SET14 [21]	PSNR	31.65/1x	31.31/2.31x	31.61/5.96x	30.00/3.76x	30.37/**7.91x**
				SSIM	0.90/ --	0.89/ --	0.89/ --	0.88/ --	0.88/ --
CartoonGAN [22]	Style Transfer	GAN	Flickr [23]	See Figure 9	-- /1x	-- /2.30x	-- /5.96x	-- /3.75x	-- /**8.16x**

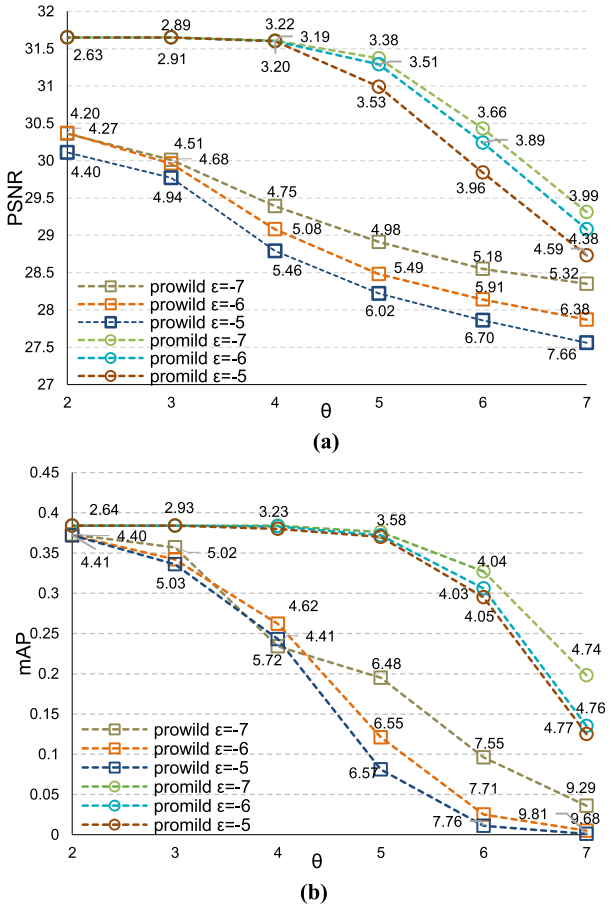

Fig. 8. Design space exploration of parameters ε and θ on (a) LapSRN and (b) FCOS.

the value remains nearly stable with θ scaled from 2 to 4. This experiment concludes that both θ and ε affect the performance and speedup of *BitXpro*. Considering that the accuracy is the first-order design constraint, we select $\varepsilon = -6$ and $\theta = 2$ for *BitXpro-mild* and $\varepsilon = -7$ and $\theta = 4$ for *BitXpro-wild* in other evaluations.

D. Performance in Other Artificial Intelligence Domains

In addition to the image classification models, Table V shows the results of other famous DNNs in various domains. The metric of the model distinguishes from each other. It is obvious that *BitX*/*BitXpro* exhibits a very tiny accuracy decrease (less than 0.1%) compared with the vanilla baseline. A more surprising result is that *BitX*/*BitXpro* performs on par or even better on some of the DNNs: it demonstrates 0.3% improvement on *BitXpro*; PRN and D3DNet demonstrate the identical mAP and SSIM value to the baseline. *BitXpro-wild* owns the highest sparsity improvement. This is reasonable because it prunes more bit rows than *BitX-wild* governed by the two regularization parameters—ε and θ.

We also present two visual comparisons for the image generation tasks—CartoonGAN and LapSRN. Fig. 9 compares the cartoon style transfer effect using *BitX*/*BitXpro*. The pruned results are visually quite similar to the nonpruned CartoonGAN, which again proves that *BitXpro* is supposed to be a better option for balancing the tradeoff between accuracy and speed. A similar result is also presented in Fig. 10 for the single-image super-resolution task—LapSRN (zoom in for better visual comparison).

E. Performance of the Fixed-Point DNN

1) Accuracy: *BitXpro* is also feasible to 16-b fixed-point DNNs, as part of their versatility. Fixed-point weight also exhibits substantial useless bits for pruning, but the difference with floating-point weight is that it does not need exponent matching. Therefore, the "E-alignment" module in *BitXpro* accelerator is not needed and could be clock gated (Fig. 3). The weights directly pass through to the "bit-extraction" module for sorting the probabilities of each bit row. As shown in Table VI, *BitX-mild*, *BitX-wild*, and *BitXpro-mild* all exhibit the higher accuracy than the nonpruned ResNet50 and ResNext101. *BitXpro-wild* shows a slight accuracy decrease except for SqueezeNet—about 1.3%. As proved in Table VI, we can conclude that *BitXpro* could precisely pinpoint the useless bits in both floating- and fixed-point DNNs.

2) Speedup: As shown in Fig. 11, *BitXpro-wild* exhibits up to 2× speedup over the original SqueezeNet1_1. For ResNet50, the speedup is 1.74×; for DenseNet121, the datum is 1.84×. As for *BitXpro-mild*, the largest speedup emerges at SqueezeNet1_1—1.23×. Compared with *BitXpro*, *BitX* exhibits a relatively slower acceleration because the regularization is not involved. Within the *BitX* context, *BitX-mild* demonstrates tiny acceleration because each weight only has

Fig. 9. Visual comparison for CartoonGAN. We apply the style transfer for the original image on *BitX* and *BitXpro*. The results are extremely identical, which proves that *BitX* and *BitXpro* could attain faster inference with the maintained quality of result. Zoom in for better view.

Fig. 10. Visual comparison for LapSRN. We apply 4× super-resolution on the original image. The results are nearly indistinguishable for *BitX* and *BitXpro*. Zoom in for better view.

TABLE VI

IMAGENET PERFORMANCE OF *BitX* AND *BitXpro* REPRESENTATIVES UNDER 16-BIT FIXED-POINT DNNS

Model	Baseline (16b)	*BitX -mild*	*BitX -wild*	*BitXpro -mild*	*BitXpro -wild*
ResNet50	74.56	**74.57 (+0.01)**	**74.57 (+0.01)**	**74.57 (+0.01)**	73.51 (-0.05)
SqueezeNet1_1	54.86	54.80 (-0.06)	54.40 (-0.46)	53.31 (-1.55)	53.49 (-1.31)
DenseNet121	71.99	71.84 (-0.15)	71.84 (-0.15)	71.84 (-0.15)	71.03 (-0.86)
ResNext101_ 32x8d	78.27	**78.31 (+0.04)**	**78.31 (+0.04)**	**78.31 (+0.04)**	77.75 (-0.52)

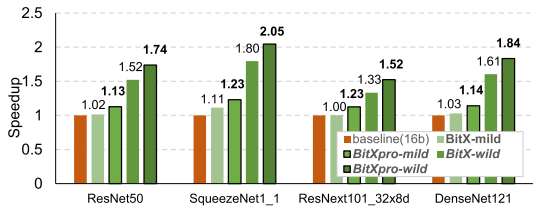

Fig. 11. Inference speedup comparison of *BitX* and *BitXpro* representatives on 16-bit fixed-point DNNs.

16-bit width and setting $N = 10$ means that only 6-bit width is pruned. The trivial bit 1s pruned are very limited. By sharp contrast, *BitX-wild* will prune 10 bits for each weight. Hence, the speedup is abundant.

F. Working With Software-Based Pruning

As a hardware runtime pruning approach, *BitXpro* is orthogonal to any software-based pruning scheme. In this experiment, we use YoloV3 [42] and wide_ResNet16 × 8 [44] as the benchmark DNNs, to implement the structured channel pruning (network slimming [17]) and the unstructured pruning (sparse loss [45]) separately. The results are shown in Table VII. It takes both the vanilla DNN and the software-pruned DNN as the baseline, but all the results are normalized to the vanilla DNN.

TABLE VII

PERFORMANCE OF *BitX* AND *BitXpro* COLLABORATING WITH SOFTWARE-BASED PRUNING (INDICATED BY "∗")

Structured Pruning	mAP (%)	Speedup (x)	Unstructured Pruning	Top-1 (%)	Speedup (x)
YoloV3 **(baseline)**	**52.73**	1	WRN16x8 **(baseline)**	**79.91**	1
YoloV3$_{mild}$	51.92	2.31	WRN16x8$_{mild}$	79.52	2.28
YoloV3$_{promild}$	**52.88**	6.13	WRN16x8$_{promild}$	79.80	5.37
YoloV3$_{wild}$	51.13	3.73	WRN16x8$_{wild}$	78.79	3.68
YoloV3$_{prowild}$	**52.74**	8.27	WRN16x8$_{prowild}$	79.03	8.03
YoloV3* **(baseline)**	**52.47**	1.32	WRN16x8* **(baseline)**	**77.96**	2.07
YoloV3$^*_{mild}$	51.24	3.05	WRN16x8$^*_{mild}$	77.56	4.67
YoloV3$^*_{promild}$	**52.48**	7.90	WRN16x8$^*_{promild}$	77.88	8.91
YoloV3$^*_{wild}$	50.42	4.94	WRN16x8$^*_{wild}$	77.41	7.51
YoloV3$^*_{prowild}$	52.22	10.81	WRN16x8$^*_{prowild}$	77.31	13.92

BitXpro-mild and *BitXpro-wild* exhibit 6.13× and 8.27× higher speedup than the baseline YoloV3, respectively. The speedup of *BitXpro* is even more considerable with structured pruning: 10.81× for *BitXpro-wild* and 7.90× for *BitXpro-mild*. Compared with the baseline WRN16 × 8, the speedups of *BitXpro-mild* and *BitXpro-wild* are 5.37× and 8.03×, respectively. Collaborated with the unstructured pruning, the speedup is 8.91× and 13.92×. The pure hardware-pruning speedup can be calculated as the *BitXpro* datum over the software-pruning baseline. Taking YoloV3$^*_{\textbf{prowild}}$ as the example, the pure speedup brought by *BitXpro-wild* is $10.81/1.32 \approx 8.20\times$. The promising acceleration result only costs with a negligible accuracy loss. This experiment clearly proves that our method is completely compatible with the software-pruning scheme, either structured or unstructured. The users could obtain additional speedup and even higher accuracy by collaborating *BitXpro* with software pruning in an effortless way.

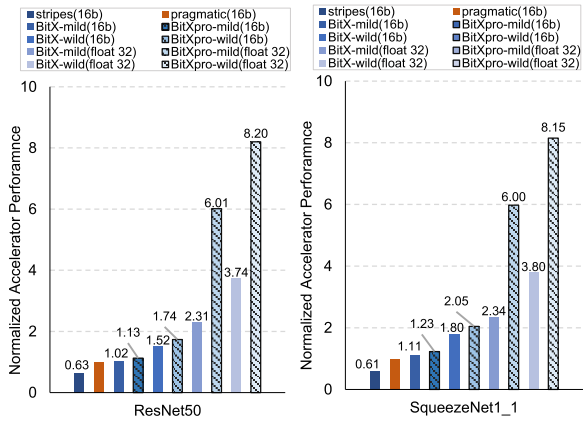

Fig. 12. Speedup comparison with other SOTA accelerators.

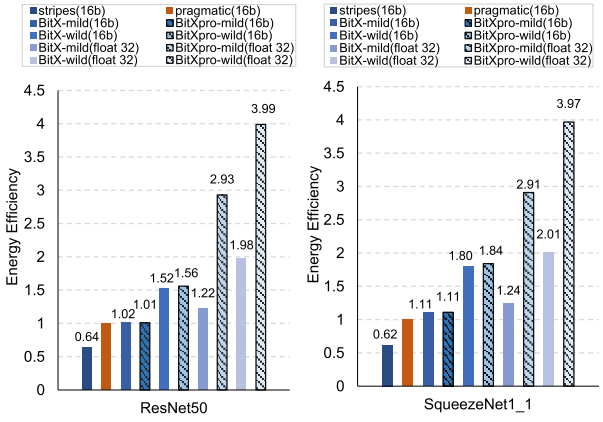

Fig. 13. Energy efficiency comparison. Higher is better.

An interesting observation is that *BitXpro-mild* achieves better performance than *BitX-mild* for both YOLOV3 and WRN16 × 8. The similar observation also exhibits in Table V on other models. The reason is that after regularization in Algorithm 2, *BitXpro* could pinpoint the essential bits according to ε and θ. However, *BitX* only sorts the importance of bit rows, making it suboptimal in essential-bit identification.

G. Comparison With SOTA Accelerators

In this section, we compare the *BitXpro* representatives with the SOTA fixed-point accelerators. Stripes [2] and Pragmatic [4] are two bit-serial accelerators. Stripes implement the MAC computation using bit-level arithmetic but do not consider the sparsity. Pragmatic, on top of stripes, exploits the bit sparsity by dynamically skipping the zero bits. However, it is not designed for bit pruning. *BitXpro* targets two types of useless bits and solves the irregularity problem to shorten the critical path latency of locating the essential bits.

1) Speedup: As shown in Fig. 12, the speedup over pragmatic (the normalized baseline) and stripes is 2.05× and 8.2× for *BitXpro-wild* and 1.23× and 6.01× for *BitXpro-mild.* A more interesting observation is that the floating-point results are even better than the fixed-point *BitXpro*, still because the sparsity in the floating-point weight is much larger, especially after the exponent alignment operation in Fig. 2(b). The functional flexibility provided by *BitXpro* releases more artificial intelligence (AI) tasks that could run on these two pruning modes. The users could freely customize their DNNs for the practical use.

2) Energy Efficiency: Similar to the speedup result, the energy efficiency of *BitXpro* also outperforms other baselines as shown in Fig. 13. For the 16-b mode, *BitX* and *BitXpro* behave on par with the representatives and demonstrate much better result than the *BitX* representatives (3.97× SqueezeNet1_1, 3.99× ResNet50 in *wild*, 2.91× SqueezeNet1_1, and 2.93× ResNet50 in *mild*). This is because *BitXpro* has a higher speed after the regularization, and the power consumption introduced by the bit shifter is not that large (see next). Again, the efficiency of fp32 mode is better than that of the 16-b mode, except for *BitX-mild* (0.34× lower than *BitXpro-wild* 16 b and 0.3× lower than *BitX-wild* 16 b for ResNet50).

3) Energy Breakdown: Our Xilinx V7 FPGA platform involves the DDR3 memory. We use DRAMsys to estimate the runtime memory access energy. Fig. 14 shows the energy breakdown from two aspects: 1) Fig. 14(a) shows the full-system energy breakdown and clearly the memory accesses dominate the energy consumption, for example, the memory access energy of the 16-b *BitXpro-wild* could attain 98% and the PE energy only occupies less than 2% and 2) in Fig. 14(b), we further decompose the PE-only energy for each *BitXpro* instance. CU energy dominates this time (58%, 81%, 74%, and 72%) because we have 16 CUs with a large number of buffers to store the bit-pruned weights. For other modules, the bit shifter and control circuits consume around 8% and 5%–35% energy, respectively.

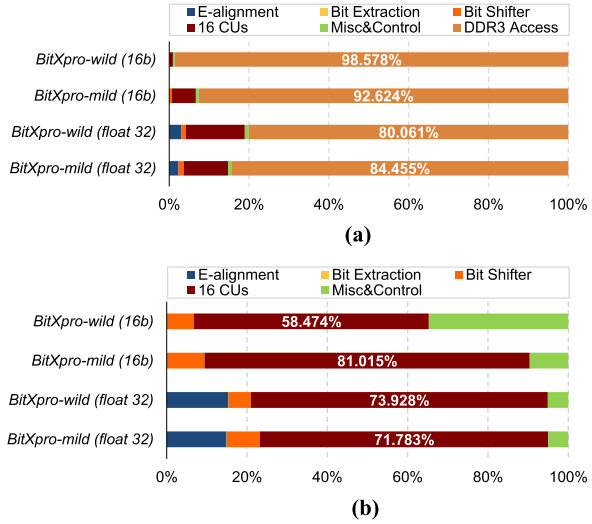

Fig. 14. (a) Full-system and (b) *BitXpro* PE-only energy breakdown for SqueezeNet.

4) Area and Power Breakdown: Under the TSMC 28-nm technology node, *BitXpro*-fp32 exhibits a 0.068-mm^2 area, while the area of stripes and pragmatic are 0.191 and 0.359 mm^2, respectively, under TSMC 65 nm. The area and power breakdown of *BitXpro* is provided in Table VIII. It illustrates that the largest area is occupied by the bit shifter module (42.6%) because it involves frequent shifting operation and some of the wires are inevitably prolonged to avoid the intersection. However, it is not the largest power consumer (only 8.3%) because no computation circuits are involved in this module. Comparatively, the 16 CUs occupy the smallest area (4.4%) but consume most of the power (71.8%) due to the internal arithmetic logic. *BitXpro* in 16-b fixed-point mode powers down the E-alignment module and the CU arithmetic is based on the fixed-point activations, so its overall power

TABLE VIII
PE AREA AND POWER BREAKDOWN COMPARISON

Item	Area (mm^2)	Power (mW)	
		floating-point 32	16b fixed-point
E-alignment Module	0.017 (25%)	11.150 (14.9%)	N/A
Bit Extraction	0.008 (11.8%)	0.040 (0.1%)	0.026 (0.1%)
Bit Shifter	0.029 (42.6%)	6.203 (8.3%)	4.032 (10.0%)
16 CUs	0.003 (4.4%)	53.710 (71.8%)	35.810 (88.5%)
Misc&Control	0.011 (16.2%)	3.720 (4.9%)	0.576 (1.4%)
***BitXpro* Accelerator Total**	**0.068**	**74.823**	**40.444**
Stripes [2]	0.191	N/A	30.2
Pragmatic [4]	0.359	N/A	51.6

consumption reduces to 40.44 mW, compared with 74.82 mW in the fp32 mode.

VI. CONCLUSION

In this article, we propose a novel hardware runtime pruning method—*BitXpro* to empower the versatile DNN inference. By targeting the abundant bit-level sparsity, it implements the pruning on-the-fly in hardware without any software work. The method precisely locates the essential bits, prunes away the trivial bits, and finally regularizes the pruned weights at different precisions, including both floating point and fixed point. The empirical studies have proved the efficacy of *BitXpro*, by providing abundant sparsity, faster inference speed, and lossless (or even higher) accuracy on various domain-specific AI tasks. We also hope that the *BitXpro* methodology and the associate accelerator design would stimulate more insightful perspectives on the hardware runtime pruning, to provide both promising DNN acceleration and excellent user experience at the same time.

REFERENCES

[1] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3D convolutional networks," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 4489–4497.

[2] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[3] K. Soomro, A. R. Zamir, and M. Shah, *UCF101: A Dataset 101 Human Actions Classes From Videos the Wild*, document CRCV-TR-12-01, 2012.

[4] J. Albericio et al., "Bit-pragmatic deep neural network computing," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2017, pp. 382–394.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[6] M. Kocabas, S. Karagoz, and E. Akbas, "MultiPoseNet: Fast multi-person pose estimation using pose residual network," in *Proc. ECCV*, Sep. 2018, pp. 417–433.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[8] T. Y. Lin, M. Maire, S. Belongie, J. Hays, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Proc. ECCV*, Sep. 2014, pp. 740–755.

[9] Z. Tian, C. Shen, H. Chen, and T. He, "FCOS: Fully convolutional one-stage object detection," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2019, pp. 9627–9636.

[10] T. B. Brown et al., "Language models are few-shot learners," in *Proc. NIPS*, 2020, pp. 1877–1901.

[11] A. Dosovitskiy et al., "An image is worth 16×16 words: Transformers for image recognition at scale," 2021, *arXiv:2010.11929v2*.

[12] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," 2016, *arXiv:1607.03250*.

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.

[14] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," 2017, *arXiv:1608.08710v3*.

[15] X. Ying, L. Wang, Y. Wang, W. Sheng, W. An, and Y. Guo, "Deformable 3D convolution for video super-resolution," 2020, *arXiv:2004.02803*.

[16] J.-H. Luo and J. Wu, "An entropy-based pruning method for CNN compression," 2017, *arXiv:1706.05791*.

[17] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 2736–2744.

[18] T. Xue, B. Chen, J. Wu, D. Wei, and W. T. Freeman, "Video enhancement with task-oriented flow," 2017, *arXiv:1711.09078*.

[19] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. NIPS*, Dec. 2016, pp. 2082–2090.

[20] W.-S. Lai, J.-B. Huang, N. Ahuja, and M.-H. Yang, "Deep Laplacian pyramid networks for fast and accurate super-resolution," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 624–632.

[21] R. Zey De, M. Elad, and M. Protter, "On single image scale-up using sparse-representations," in *Proc. Int. Conf. Curves Surf.*, 2010, pp. 711–730.

[22] Y. Chen, Y.-K. Lai, and Y.-J. Liu, "CartoonGAN: Generative adversarial networks for photo cartoonization," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 9465–9474.

[23] *Flickr Image Dataset*. Accessed: Apr. 27, 2022. [Online]. Available: https://www.kaggle.com/hsankesara/flickr-image-dataset

[24] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, pp. 1–18, May 2017.

[25] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[26] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 4700–4708.

[27] X. Wang, R. Girshick, A. Gupta, and K. He, "Non-local neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 7794–7803.

[28] X. Zhou et al., "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 15–28.

[29] H. Song, X. Liu, H. Mao, P. Jing, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ISCA*, Jun. 2016, pp. 243–254.

[30] S. Han et al., "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2017, pp. 75–84.

[31] A. Parashar et al., "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 27–40.

[32] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ISCA*, Jun. 2016, pp. 1–13.

[33] *IEEE Standard for Floating-Point Arithmetic (754–2019)*. Accessed: Jan. 10, 2021. [Online]. Available: https://standards.ieee.org/standard/754-2019.html

[34] J.-H. Luo, J. Wu, and W. Lin, "ThiNet: A filter level pruning method for deep neural network compression," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 5058–5066.

[35] H. Lu, X. Wei, N. Lin, G. Yan, and X. Li, "Tetris: Re-architecting convolutional neural network computation for machine learning accelerators," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, pp. 1–8.

[36] H. Lu, M. Zhang, Y. Han, Q. Wang, H. Li, and X. Li, "Architecting effectual computation for machine learning accelerators," in *Proc. TCAD*, Oct. 2020, pp. 2654–2667.

[37] A. D. Lascorz et al., "Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 749–763.

[38] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: A 50.6TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2018, pp. 218–220.
[39] P. Drineas, R. Kannan, and M. W. Mahoney, "Fast Monte Carlo algorithms for matrices III: Computing a compressed approximate matrix decomposition," *SIAM J. Comput.*, vol. 36, no. 1, pp. 184–206, Jan. 2006.
[40] H. L. H. Li et al., "BitX: Empower versatile inference with hardware runtime pruning," in *Proc. ICPP*, Aug. 2021, pp. 1–12.
[41] *Facebook*. PyTorch. Accessed: Jan. 26, 2021. [Online]. Available: https://pytorch.org/
[42] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," in *Proc. CVPR*, Apr. 2018, pp. 1–6.
[43] M. Jung, C. Weis, and N. Wehn, "DRAMSys: A flexible DRAM subsystem design space exploration framework," *IPSJ Trans. Syst. LSI Des. Methodol.*, vol. 8, pp. 63–74, Jan. 2015.
[44] S. Zagoruyko and N. Komodakis, "Wide residual networks," 2016, *arXiv:1605.07146*.
[45] G. Retsinas, A. Elafrou, G. Goumas, and P. Maragos, "Weight pruning via adaptive sparsity loss," 2020, *arXiv:2006.02768*.

Hongyan Li is currently working toward the second-year master's degree at the State Key Laboratory of Computer Architecture, Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China.

Her research interests include deep learning network pruning, video understanding, and dynamic inference.

Hang Lu is currently an Associate Professor and a Master Tutor at the State Key Laboratory of Computer Architecture, Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China. He is also a Research Scientist with the Shanghai Innovation Center for Processor Technologies, CAS. He is a member of the Youth Innovation Promotion Association of CAS and the New Best Star of ICT. His research interests include power-efficient computing platforms, artificial intelligence (AI) chip design, and deep learning algorithm optimization.

Haoxuan Wang is currently working toward the second-year master's degree at Fujian Normal University, Fuzhou, China.

He is currently a Visiting Master Student at the State Key Laboratory of Computer Architecture, Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China. His main research interests are neural network hardware acceleration and dedicated domain architecture (DSA).

Shengji Deng is currently an Associate Researcher at the Second Research Institute of the Civil Aviation Administration of China (CAAC), Chengdu, China. His research interests include digital signal processing, electronic system reliability design, and edge computing.

Xiaowei Li (Senior Member, IEEE) received the B.Eng. and M.Eng. degrees in computer science from the Hefei University of Technology, Hefei, China, in 1985 and 1988, respectively, and the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 1991.

He was an Associate Professor with the Department of Computer Science and Technology, Peking University, Beijing, from 1991 to 2000. In 2000, he joined ICT, CAS, as a Professor, where he is currently the Deputy Director of the State Key Laboratory of Computer Architecture. He has coauthored over 280 papers in journals and international conferences. He holds 60 patents and 30 software copyrights. His current research interests include VLSI testing, design for testability, design verification, dependable computing, and wireless sensor networks.

Dr. Li was the Chair of the Technical Committee on Fault-Tolerant Computing, China Computer Federation, from 2008 to 2012, and the Steering Committee Chair of the IEEE Asian Test Symposium from 2011 to 2013. He was the Steering Committee Chair of the IEEE Workshop on RTL and High Level Testing from 2007 to 2010. He has been the Vice Chair of the IEEE Asia and Pacific Regional Test Technology Technical Council since 2004. He served as an Associate Editor for the *Journal of Computer Science and Technology*, the *Journal of Low Power Electronics*, the *Journal of Electronic Testing: Theory and Applications*, and the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS.