BitX: Empower Versatile Inference with Hardware Runtime Pruning

Hongyan Li
Hang Lu
State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China

Jiawen Huang
State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
Beijing, China

Wenxu Wang
State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China

Mingzhe Zhang
State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
Beijing, China

Wei Chen
State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
Beijing, China

Liang Chang
University of Electronic Science and Technology of China
Chengdu, China

Xiaowei Li
State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China

ABSTRACT

Classic DNN pruning mostly leverages software-based methodologies to tackle the accuracy/speed tradeoff, which involves complicated procedures like critical parameter searching, fine-tuning and sparse training to find the best plan. In this paper, we explore the opportunities of *hardware runtime pruning* and propose a hardware runtime pruning methodology, termed as "*BitX*" to empower versatile DNN inference. It targets the abundant useless bits in the parameters, pinpoints and prunes these bits on-the-fly in the proposed *BitX* accelerator. The versatility of *BitX* lies in: (1) software effortless; (2) orthogonal to the software-based pruning; and (3) multi-precision support (including both floating point and fixed point). Empirical studies on image classification and object detection models highlight the following results: (1) up to 4.82x speedup over the original non-pruned DNN and 14.76x speedup collaborated with the software-pruned DNN; (2) up to 0.07% and 0.9% higher accuracy for the floating-point and fixed-point DNN, respectively; (3) 2.00x and 3.79x performance improvement over the state-of-the-art accelerators, with 0.039 mm^2 and 68.62 mW (floating-point 32), 36.41 mW(16-bit fixed point) power consumption under TSMC 28 nm technology library.

ICPP '21, August 9–12, 2021, Lemont, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472513

CCS CONCEPTS

• **Computer systems organization → Neural networks**.

KEYWORDS

hardware pruning, deep learning accelerator, neural networks

ACM Reference Format:
Hongyan Li, Hang Lu, Jiawen Huang, Wenxu Wang, Mingzhe Zhang, Wei Chen, Liang Chang, and Xiaowei Li. 2021. *BitX*: Empower Versatile Inference with Hardware Runtime Pruning. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3472456.3472513

1 INTRODUCTION

Large computation intensity is well recognized as one of the main obstacles to deploy DNNs into practical applications, because of the rapid evolution of the parameter size from millions (i.e. ResNet [10] family in computer vision) to even hundreds of billions (i.e. BERT [7] or GPT-3 [5] 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

Author 1 and 2 contributed equally. Corresponding author is Hang Lu, email: luhang@ict.ac.cn

Table 1: WEIGHT/BIT sparsity comparison for various DNNs pre-trained with *ImageNet dataset*. Bit sparsity is significantly more abundant than weight sparsity. The weights are represented by floating-point 32.

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%

and desirable request however, cannot catch up with the development of DNNs, especially for the 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, i.e. [11] [21] [25] [22] [33] rely on software-level efforts which usually consist 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 the super-parameters and DNN structured parameters and re-implement the above steps from the very inception. The tediously repeated effort limits the fast deployment of DNNs in the 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 [11], the absolute value of filters [21], or the entropy of filters [25] and so on, 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 for 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 the information loss or sparse training to manually create more sparsity [22] [33] [4] in

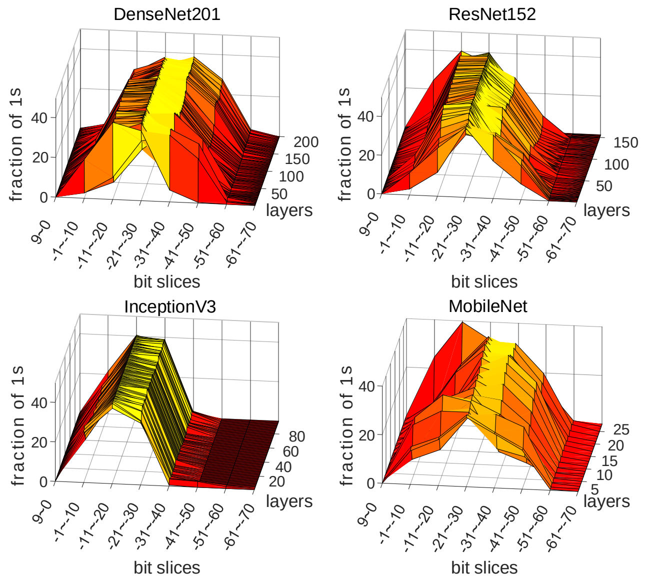


Figure 1: Distribution analysis of bit 1s. The 4 benchmark DNNs demonstrate a similar behavior: the surfplot reaches its peak at $2^{-21} \sim 2^{-30}$, which means this bit slice has the largest fraction of bit 1s (nearly 40%), but most of them are trivial. *BitX* aims to prune these trivial bits to obtain the inference acceleration.

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. That is because the remaining non-pruned 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 layer-wise, so if we apply it to VGG-19 [29] 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, 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 [10], DenseNet [12]) or even much larger and more complex connections like 3D convolution [31], non-local convolution [32] or deformable convolution [6], 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 [34] addresses the irregularity of unstructured pruning. EIE [30] supports the pruning only for the fully-connected layers; ESE [9] 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 [27] exploits

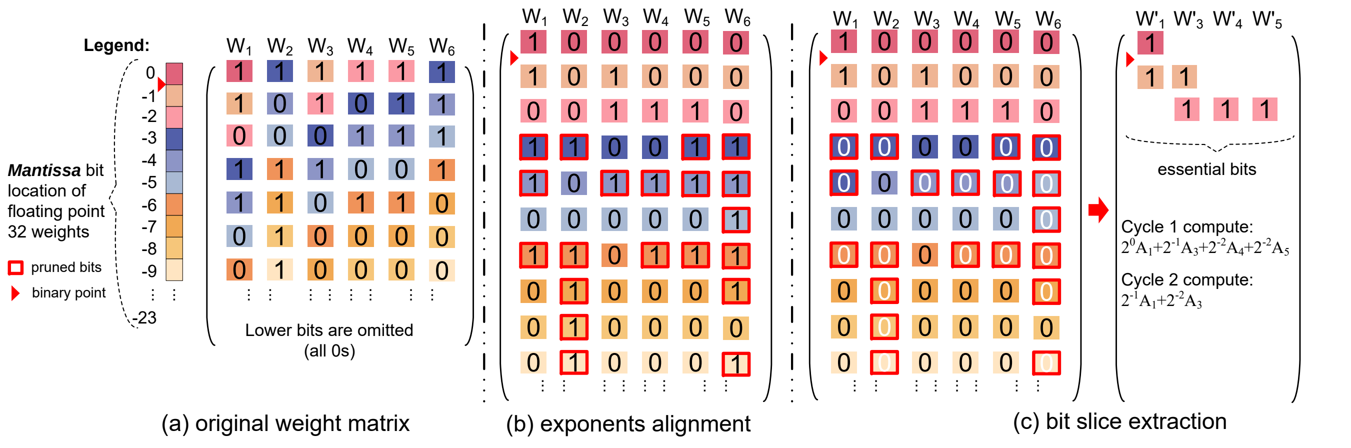

Legend:
Mantissa bit location of floating point 32 weights
pruned bits
binary point
Lower bits are omitted (all 0s)
essential bits
Cycle 1 compute: $2^0A_1+2^{-1}A_3+2^{-2}A_4+2^{-2}A_5$
Cycle 2 compute: $2^{-1}A_1+2^{-2}A_3$
(a) original weight matrix
(b) exponents alignment
(c) bit slice extraction

Figure 2: Core concept of *BitX*. Bit matrix before pruning is shown in (a). (b) demonstrates exponent alignment according to IEEE 754. Six non-essential bit rows are pruned as shown in (c), only leaving a more compact essential-bit matrix.

the neuron and synapse sparsity, while Cnvlutin [3] only supports neuron sparsity. If the software engineer modifies the pruning policy or simply changes from structured pruning to the unstructured pruning, the hardware employed is also about to change, attached with overwhelming transplantation overhead.

Ideally, a pre-trained 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 paper, we propose *BitX*, a *hardware runtime pruning* methodology to empower versatile DNN inference. Apart from the software-based pruning that requires complex algorithm to identify the trivial values by repeated trial-and-error, *BitX* implements pruning by targeting bits. It aims to pinpoint the essential bits and prune away the useless bits in the parameters, because the useless bits are more easily exposed especially at the hardware level. For example, the floating-point value has long bit-width in its mantissa (24 bits) [13], and the exponent always shifts the mantissa to align the binary point with another value for computation. The shifted binary positions are automatically zero-padded and involved in the floating-point arithmetic. This procedure generates two types of useless bits: the 1^{st} type is the genetic zero bits in the mantissa and the automatically padded zero bits; the 2^{nd} type is more implicit, that is, the rear bit 1s with extremely trivial significance. As will be shown in Section 2, the two types of useless bits *both* occupy a large fraction in the binary represented weights, which also provides a decent condition for *BitX* to prune these bits directly in the accelerator at the inference runtime.

Our evaluations have shown that by precisely locating and pruning these useless bits, the inference speed could be significantly boosted (Section 4.2). Most importantly, the whole pruning operation could be implemented on-the-fly in the proposed *BitX accelerator*, with lossless accuracy and without any software related effort.

The contributions of this paper are listed as follows:

- **We propose a novel hardware runtime pruning method, termed as *BitX*, to empower versatile DNN inference.** We highlight the following features of *BitX*:
  (1) **Software effortless:** *BitX* directly prunes the original DNN. No retraining, fine-tuning, or special library/framework needed, because it targets the useless binary bits not values.
  (2) **Orthogonal to the existing software pruning methodologies:** *BitX* implements straightforward bit pruning in the accelerator, so the DNNs, either pruned or non-pruned at the software level, are all suitable for *BitX*. In other words, it could further prune the useless bits of the software-pruned DNN, and obtain additional speedup.
  (3) **Multi-precision support:** *BitX* applies to not only floating-point but also fixed-point DNNs. The fixed-point DNNs also demonstrate substantial useless bits. *BitX* could accelerate these models with even higher accuracy and speedup.
- **We propose a deep learning accelerator capable of unprecedented hardware runtime pruning to mine the maximum potential of *BitX*.** We highlight the following results:
  (1) **Speedup:** two representatives of *BitX* accelerator – *BitX-mild* and *BitX-wild* could respectively obtain 2.61x ~ 4.82x faster speed over the non-pruned baselines under float-point 32 mode, and up to 2.00x under 16-bit fixed-point mode. For the object detection model YoloV3, the speedup is up to 4.98x and 14.76x higher over the original model.
  (2) **Accuracy:** for *ImageNet*, the accuracy loss is 0.13% and 0.44% for *BitX-mild* and *BitX-wild*; for *Cifar-10*, the loss is 0.09% and 0.15%. The above accuracy data are reported by the floating-point DNNs. Under 16-bit fixed-point, the accuracy is even 0.9% and 0.2% higher than the baseline DenseNet121 and ResNext101 for *BitX-mild*; 0.8% and 0.1% higher for *BitX-wild*. For YoloV3, the accuracy is 0.06% and 0.07% higher than the original model for *BitX-mild*; 0.31% and 1.64% lower for *BitX-wild*.
  (3) **Accelerator Performance:** we compare the *BitX* accelerator performance with other state-of-the-art accelerator

prototypes. Equipped by *BitX*, the accelerator achieves 2.00x and 3.79x performance improvement. The area is 0.039 mm^2 and 68.62 mW (floating-point 32) and 36.41 mW (16-bit fixed point) under TSMC 28nm technology library.

(4) **Sensitivity:** we thoroughly evaluate the sensitivity of the key design parameters to the accuracy and speed (see Section 4.3).

Powered by the above features, *BitX* is designed for *flexible* and *versatile* DNN inference at any circumstances. In the next section, we will start from discussing two key observations that justify *BitX*.

2 OPPORTUNITIES OF HARDWARE RUNTIME PRUNING

2.1 Bit-level Sparsity – 1st target

For most of the software-based pruning approaches in the literature [22] [11] [26] [33], 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 [11] [26], or commence sparse training to create more headroom for the employed pruning metric [22] [33]. It is also the root reason that the labor-intensive software effort stems from.

In order to circumvent the inconvenience, we re-examine the parameters in-depth. Instead of sticking to the sparse "values", we analyze the more fine-grained bit-level sparsity. As shown in Table 1, 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, are nearly 49% and no exception. It provides an excellent opportunity of exploiting the sparsity at the bit level without resorting to the value pruning, because ~49% bits are already 0s and removing these useless bits off the MAC computation is definitely harmless to the accuracy. *BitX* intends to fully utilize this decent condition to accelerate the DNN inference.

2.2 Trivial bit 1s – 2nd target

Obviously, we can design particular zero-skipping mechanism to avoid the ineffectual computations caused by the zero bits, which is also the main objective many previous sparsity-aware acceleration schemes target [23] [24] [2] [19]. 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 *BitX*. 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 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.

Algorithm 1 *BitX* Pruning Procedure

```
Require: $n$ number of fp32 weights, and bit-sparsity level $N$
Ensure: essential bit matrix $W'$
1: Interpret $n$ exponents $E = [e_1, ..., e_n]$ and mantissas $M = [m_1, ..., m_n]$;
2: Adding the hidden '1' and obtain $M' = [m'_1, ..., m'_n]$;
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: end for
11: $P.append(p_i)$; // $P$ stores each $p_i$
12: $I, P = sort(P)$; // sort $P$ in descending order, obtain index vector $I$
13: $I' = max(I, N)$; // $I' = [i_1, ..., i_n]$, obtain the first $N$ indices
14: foreach index $i$ in $I'$ :
15:     $mask[i] = 1$;
16: end for
17: $W \leftarrow W \otimes mask$; // prune $W$ with $mask$, Figure 2(c)
18: $n = 0$;
19: foreach row $i$ in $W$ :
20:     foreach column $j$ in $W$ :
21:         if $W(i, j) == 1$ then:
22:             $W'(i, n) = 1$;
23:             $n+ = 1$;
24:         end if
25:     end for
26:     $n = 0$; // reset $n$ for the next row
27: end for
28: return $W'$; // essential bit matrix
```

As evidence, we explore the distribution of bit 1s in each bit slice. As shown in Figure 1, the X-axis denotes the bit slice of the binary represented weight (in floating-point 32). 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 four bit 1s are the 2^{-4}, 2^{-5}, 2^{-6}, and 2^{-8} after the binary point, respectively.

According to Figure 1, the bit slice could range from bit significance "9 ~ 0" before the binary point to "-61 ~ -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} ~ 2^{-30} as the representative, the equivalent decimals are in range: 0.000000477 (~10^{-8}) to 0.000000000931 (~10^{-11}). These tiny values are very likely to be less contributive to the final accuracy. Therefore, *BitX* 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. In the next section, we will elaborate how it is designed to achieve this objective.

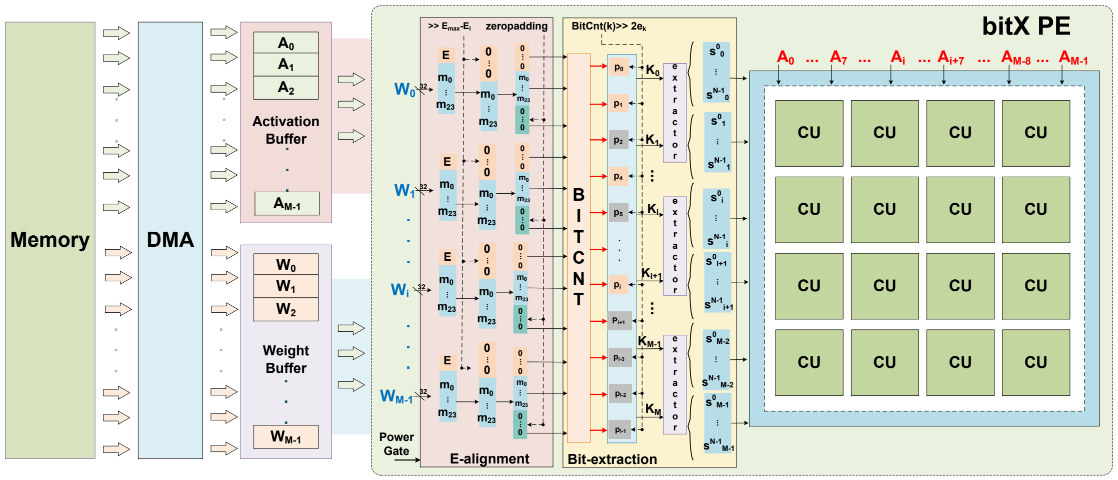


Figure 3: Overview architecture of the *BitX accelerator*.

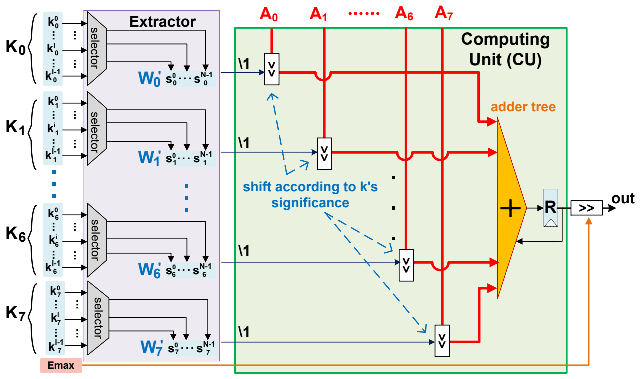


Figure 4: Microarchitecture of the “compute unit” (CU).

3 BITX

3.1 General Concept

Without losing generality, a floating-point operand is composed of three portions: the signed bit, mantissa, and exponent, following IEEE 754 [13] which is also the most commonly used floating-point standard in industry. If we employ the single precision (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 $f_p = (-1)^s 1.m \times 2^{e-127}$, in which e is the actual position of the “binary point” plus 127.

If we take 6 non-aligned fp32 weights as an example and interpret their mantissas as illustrated in Figure 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^{-1} to 2^{-9} after the binary point (position 0 denotes the hidden 1 of the mantissa [13]). In terms of the 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 dark blue in W_2 is actually the 2^{-3} significance in the fractional part.

If we align the mantissa according to the exponent, zeros are padded in the front vacancies just as shown in Figure 2(b). The first observation is that the aforementioned bit-level sparsity is more abundant after zero padding, which provides excellent condition for the bit-level pruning. The second observation is that a large fraction of bit 1s are shifted to the rear direction beyond 2^{-6} significance. A direct consequence is that the practical contribution of these bits are 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 illustrated in Figure 2(c), the red box denotes the pruned bit 1s, only leaving several essential 1s to form the pruned weights: $W_1{}'$, $W_3{}'$, $W_4{}'$ and $W_5{}'$, and we term these 1s as the “essential bits”.

3.2 Methodology

Leveraging the essential bits in Figure 2(c) is an effective way to simplify a series of value-level MACs as bit-level additions [23] [20]. However, for the millions of parameters in DNN, if we use fp32 to represent these values, the impact of a single bit to the whole network is not that easy to be determined. The leftover problem is how to create an effective yet hardware-friendly mechanism to make full use of the abundant useless bits and maintain the initial accuracy, without labor-intensive software tricks. In this subsection, we firstly formalize the problem and then elaborate the *BitX* procedure.

3.2.1 ***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 matrices: $A^{(i)}$ represents the i-th row of A and $W_{(i)}$ for the i-th column of W, as shown in Eq. (1). The criticality of these rank-one matrices could be easily decided by the Fast Monte-Carlo Algorithm [8], in which some rank-one matrices are randomly sampled to approximate $A \times W$. The most common sampling method [8] to select these rank-one matrices is by referring to their respective probability as shown in Eq. (2). It is obtained by computing the Euclidean distance of

Table 2: *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 x).

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 3: *ImageNet* performance. The computing method is identical to Table 2.

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**

$A^{(i)}$ and $W_{(i)}$, which reflects the importance of the rank-one matrix multiplication.

Inspired by the Fast Monte-Carlo Algorithm, we enroll the same probability concept in *BitX* to measure the importance of the weight bits instead of values. Bits with smaller probability tends 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 Figure 2(a) as W and our objective is seeking out the (in)significant bit rows in Figure 2(b) and simplifying MAC computations. The problem remains how we can utilize the probability in Eq. (1) to sample each bit row in W, and determine the to-be pruned bit rows.

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

3.2.2 ***Bit-slice Extraction***. In W (Figure 2 (a)), we target the mantissa of n normal floating-point 32 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 $[A_1, A_2 ... A_j ... A_n]^T$. We put the two column vectors into Eq. (1), so it could be rewritten as Eq. (2):

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

A_j is the element of the activation vector, and v_j is the j-th bit of the i-th row vector in the bit matrix W. Each bit in the same row i has its own exponent, so we use $2^{E_i^j}$ in Eq. (2) to represent the

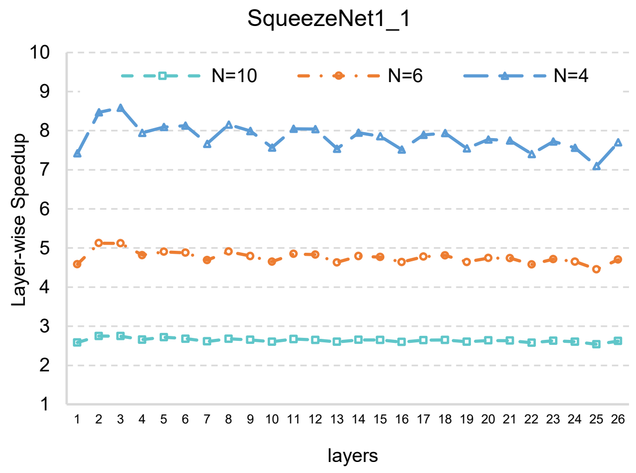


Figure 5: layer-wise speedup for *ImageNet* dataset. The speedup of the original model is regarded as 1 on the Y-axis.

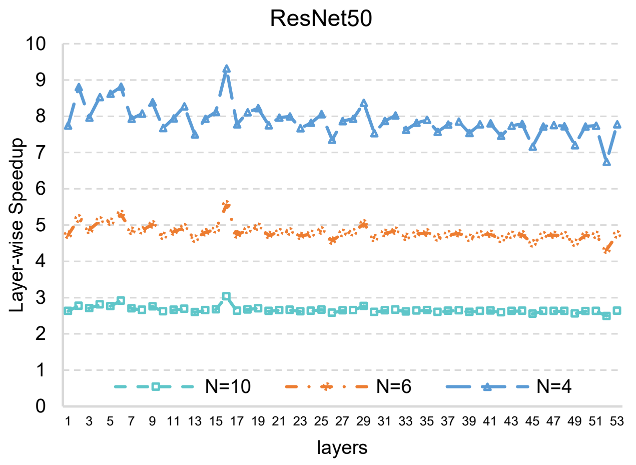


Figure 6: the method is the same as Figure 5. Higher is better.

exponent at position j. The Euclidean distance of the row vector is calculated as $\sqrt{\sum_{j=1}^{n}(2^{E_i^j}\times v_j)^2}$.

In *BitX*, exponent alignment procedure is almost identical to the normal floating-point addition [13], except one special difference, that is, *BitX* 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 Figure 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 to 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 $BitCnt(i)$ to indicate such operation, so the probability p_i of the i-th row can be represented as Eq. (3):

$$p_i = \frac{|A^{(i)}|\times\sqrt{(2^{E_i})^2\times BitCnt(i)}}{|A^{(i')}|\times\sum_{i'=1}^{l}(\sqrt{(2^{E_{i'}})^2\times BitCnt(i')})} = \frac{\sqrt{(2^{E_i})^2\times BitCnt(i)}}{\sum_{i'=1}^{l}(\sqrt{(2^{E_{i'}})^2\times BitCnt(i')})} \tag{3}$$

In Eq. (3), E_i stands for the aforementioned exponent of the i-th row. Each column vector in matrix A is the same, so $|A^{(i')}|$ equals to $|A^{(i)}|$. For the given W with l column vectors, $\sum_{i'=1}^{l}|W_{(i')}|$ is a constant, so we let $C=\sum_{i'=1}^{l}(\sqrt{(2^{E_{i'}})^2\times BitCnt(i')})$, and the final p_i is deduced by the following Eq. (4):

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

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. *BitX* takes advantage of Eq. (4) to pinpoint the essential bit rows and in the meantime, prunes away the trivial bit rows directly in the accelerator.

3.3 Pruning Procedure

In Algorithm 1, *BitX* firstly interprets the exponent E and mantissa M of n fp32 weights as input (line 1 ~ 3), aligns each exponent according to the e_{max} (line 4), and then calculates and sorts the row probabilities in descending order (line 5 ~ 12). For the other input parameter N, it denotes the remaining bit rows in W after pruning. In other words, *BitX* selects the top n bit rows with relatively larger $p_i s$. The indices of the n rows are reflected in I' (line 13). The pruning is finalized by the vector *mask* with the selected n bit rows marked as '1' (line 7 and 16). Right after pruning, *BitX* extracts the essential bits and stores them into W' (line 17 ~ 23).

The design parameter N in the algorithm controls the granularity of pruning. Smaller N inevitably leads to larger sparsity because more bit rows are pruned, which also benefits the inference by skipping more zero bits. In Section 4, we will thoroughly study the impact of N on *BitX* performance.

3.4 BitX Accelerator Architecture

BitX is a hardware runtime pruning approach, so specially designed hardware pruning modules are integrated in the accelerator design. The overall full-system accelerator architecture is shown in Figure 3. Two module "E-alignment" and "Bit-Extraction" are designed to perform Algorithm 1. We instantiate 16 computing units to form one *BitX* PE. Each CU takes M weights/activation pairs as input. The input weights are pre-processed by the "Bit-Extraction" module with the trivial bits pruned to 0. CU executes the acceleration according to the significance of each bit in the pruned weight.

On our 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. Detailed FPGA and ASIC configuration are elaborated in Section 4.

For the fixed-point DNN, the E-alignment module is bypassed and safely powered down. Original weights are directly connected to the input of the Bit-Extraction module, because the fixed-point

Table 4: Design space exploration of two key design parameters on *ImageNet*.

ResNet50	**Original Accuracy: 74.50**				**DenseNet121**	**Original Accuracy: 71.96**			
	N=10	N=8	N=6	N=4		N=10	N=8	N=6	N=4
M=8	74.50	**74.51**	74.10	67.00	M=8	71.95	71.00	71.00	65.00
M=16	**74.54**	74.40	73.60	61.00	M=16	**71.97**	72.00	71.00	62.00
M=32	74.00	74.50	73.50	58.20	M=32	**72.03**	72.00	71.00	58.00
M=64	74.39	74.00	73.00	53.30	M=64	71.00	71.00	70.00	55.00
M=128	74.41	74.32	72.70	46.30	M=128	71.00	71.00	70.00	49.20
M=256	**74.51**	74.40	72.80	46.80	M=256	71.84	71.00	69.00	49.00
M=512	74.30	74.26	71.90	39.40	M=512	71.83	71.60	69.00	34.00
ResNext101_32x8d	**Original Accuracy: 78.24**				**SqueezeNet1_1**	**Original Accuracy: 54.84**			
	N=10	N=8	N=6	N=4		N=10	N=8	N=6	N=4
M=8	78.20	**78.30**	78.10	73.00	M=8	**54.86**	54.70	54.40	47.30
M=16	78.20	78.00	77.50	66.00	M=16	54.80	54.74	53.00	41.60
M=32	78.20	78.00	78.20	65.00	M=32	54.00	54.50	53.64	41.70
M=64	78.20	78.20	77.30	62.00	M=64	54.70	54.77	53.50	37.10
M=128	78.20	78.10	77.30	57.00	M=128	54.40	54.48	52.80	34.66
M=256	78.20	78.20	77.20	49.00	M=256	54.62	54.40	52.11	32.10
M=512	78.20	78.20	77.20	49.00	M=512	54.81	54.72	52.60	32.00

arithmetic does not involve exponent matching. Power gating is favorable to the abundant energy savings in the fixed-point *BitX accelerator*, as shown in the evaluations.

3.4.1 ***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. Firstly, 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 i-th mantissa by $E_{max} - E_i$. The shifted vacancies are zero-padded in the front part of the mantissa, marked as orange in Figure 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 RTL implementation could easily hard code this operation without violating the timing constraint. The only overhead introduced is the complicated wire organization that might potentially increase the circuit area.

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

3.4.3 ***Compute Unit (CU)***. The pruned weights will surely exhibit more sparsity, and due to the pruning of the trivial bit ones, the left essential bits are substantially scarce. Therefore, a zero-skipping mechanism is designed in the "extractor" of the "Bit-Extraction" module to pinpoint the essential bits and further feed them to the compute unit (CU) module.

The microarchitecture of CU is shown in Figure 4. Each "selector" in the extractor targets one pruned binary weight (M weights in total), and k denotes the bit in the pruned weight. The extractor records the significance of each essential bit k, represented as s. s is used to shift the corresponding activation A in each shifter.

The activations could be either floating-point or fixed-point data. The fixed-point A could be directly shifted. However, for the floating-point A, the shifting operation is actually the exponent accumulation in A, which is the fixed-point arithmetic as well. Therefore, the shifters 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 precisions. Section 4.6 decomposes the power consumption of the *BitX* accelerator in ASIC design to give a comprehensive study.

4 EVALUATION

Benchmark and framework. The deep learning models and the parameters pre-trained for *Cifar-10* [18] and *ImageNet* [15] dataset are directly obtained from PyTorch [1]. The benchmark models involve "big" models with the parameter size ranging from 76.35M (DenseNet201) to 356.71M (ResNext101_32x8d),as well as "little" models with the parameter size of 4.71M (SqueezeNet1_1). YoloV3 [28] trained on CoCo [14] dataset is employed to evaluate the performance on the object detection task. For the design parameter N and M, we choose several discrete values for the design space exploration, to explore the sensitivity of *BitX* on the accuracy and speed.

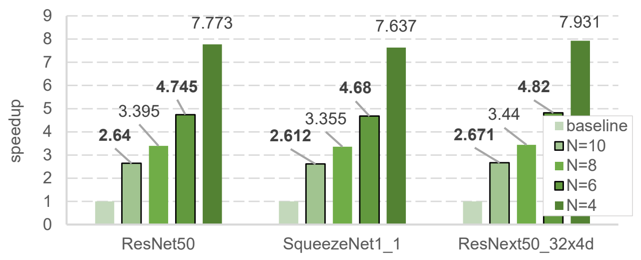


Figure 7: Inference speedup comparison under different N settings.

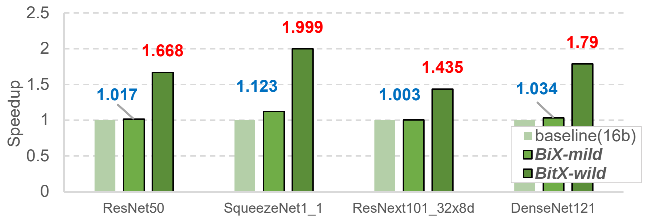


Figure 8: Inference speedup comparison of two *BitX* representatives on 16-bit fixed-point DNNs.

Also based on these parameters, we define two representatives of *BitX* – *BitX-mild* and *BitX-wild*.

FPGA & ASIC implementation. At the RTL level, we employ Vivado HLS (v2018.2) to conduct post-synthesis 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 [17] to estimate the energy consumption of the memory accesses. For ASIC, Synopsis Design Compiler (v2016) is used to measure power and area. The frequency is set to 1 GHz. The whole design is synthesized with TSMC 28nm technology library.

4.1 Accuracy & Sparsity

4.1.1 ***Cifar-10****.* Table 2 shows the accuracy/sparsity results for *Cifar-10* dataset, grouped by the parameter N. Smaller N means more bit rows are pruned, so the bit-level sparsity also turns larger. For example for $N = 4$, the sparsity increases to 1.80x compared with the original model. More sparsity is undoubtedly beneficial to the inference speedup (as proved in Section 4.2). On the other hand, larger N means less bit rows are pruned so the sparsity only shows 1.41x and 1.53x for $N = 10$ and $N = 8$.

BitX shows promising pruning accuracy. The average accuracy loss is about 0.1% at $N = 10, 8$ and 6 for various evaluated DNNs.

4.1.2 ***ImageNet****.* The results for *ImageNet* dataset shown in Table 3 exhibit a similar trend as Table 2: less than 0.5% average accuracy loss at $N = 10, 8, 6$, and 1.40x, 1.52x, 1.66x sparsity increment apiece.

Discussion: firstly, it proves that the proposed *BitX* 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. Secondly, this experiment clearly demonstrates the tradeoff between accuracy and sparsity. A borderline configuration also exists with the maintained accuracy and satisfied sparsity. As shown in Table 2 and Table 3, there is a significant accuracy drop at $N = 4$ and $N = 6$. Therefore, we can safely choose N values in the range 10 ~ 6 in *BitX*. This experiment also verifies that there are tremendously redundant bits in the parameters that can be safely pruned without hurting the accuracy.

Table 5: *ImageNet* performance of two *BitX* representatives under 16-bit fixed-point DNNs.

Model	Baseline(16b)	*BitX-mild*	*BitX-wild*
ResNet50	74.50	**74.50 (0.00)**	74.10 (-0.40)
SqueezeNet1_1	54.86	54.80 (-0.06)	54.40 (-0.46)
DenseNet121	71.00	**71.90 (+0.90)**	**71.80 (+0.80)**
ResNext101_32x8d	78.00	**78.20 (+0.20)**	**78.10 (+0.10)**

4.2 Speedup

We evaluate the inference speed at different sparsity levels indicated by the N configuration. The speedup data are recorded according to the actual inference cycles on our Xilinx V7 FPGA platform and normalized to the original non-pruned DNN. As shown in Figure 7, *BitX* exhibits ~2.6x speedup at $N = 10$, and ~4.8x speedup at $N = 6$. The promising speedup stems from the enriched bit sparsity enforced by the *BitX* pruning. More abundant sparsity enables more zero-bit skipping in the *BitX accelerator*, and thus leads to much faster inference speed.

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

4.3 Design Space Exploration

Section 4.1 and Section 4.2 have evaluated the sensitivity of N and its impact to the *BitX* performance. We further explore the impact of another key parameter M in this experiment. We use 4 DNNs trained with the *ImageNet* dataset as shown in Table 4. M indicates the number of input weights that the accelerator could simultaneously prune (Figure 3), and generally speaking, M barely influences the overall accuracy scaling from 8 ~ 512 for all the 4 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 negligible impact to the performance of *BitX*.

Table 6: Performance of BitX collaborating with software-based pruning. We use the genetic and channel-pruned YoloV3 model.

Method	mAP(%)	Speedup(x)
YoloV3 **(baseline)**	50.36	1
YoloV3 + ***BitX-mild***	**(50.42) (+0.06)**	2.75
YoloV3 + ***BitX-wild***	50.05 (-0.31)	**4.98**
YoloV3 + Slimming [22] **(baseline)**	50.23 (-0.13)	2.35
YoloV3 + Slimming [22] + ***BitX-mild***	**50.30 (+0.07)**	7.22
YoloV3 + Slimming [22] + ***BitX-wild***	48.72 (-1.64)	**14.76**

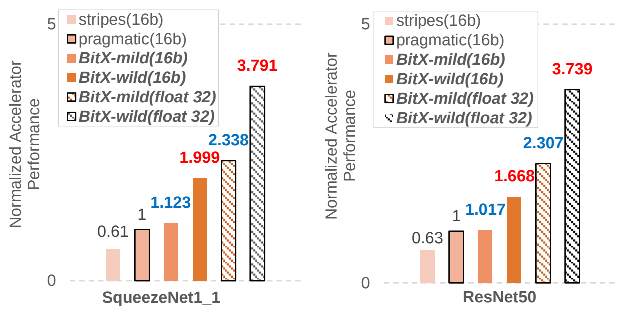


Figure 9: Speedup comparison with other SOTA accelerators.

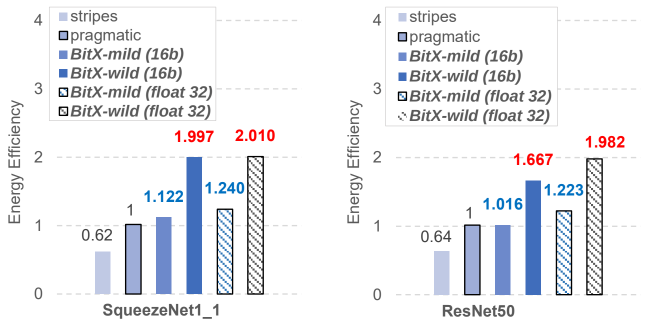
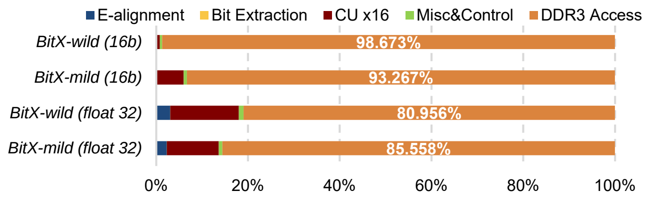


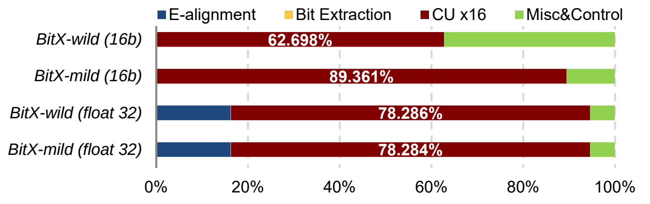
Figure 10: Energy efficiency comparison. Higher is better.

Discussion: the major factor that steers the accuracy and speedup is the N configuration. Table 4 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 3. It is N that decides the granularity of pruning, while M only controls the input throughput.

Two *BitX* 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 *BitX instances*, termed as *BitX-mild* ($N = 10$, $M = 8$) and *BitX-wild* ($N = 6$, $M = 8$). *BitX-mild* has the topmost accuracy but limited speedup, while *BitX-wild* has little-degraded accuracy but relatively abundant speedup. Two instances are used to verify the "accuracy-speedup" tradeoff, and in the rest of this section, we will use the two instances as the representatives of *BitX* to compare with other SOTA accelerator baselines.



(a) Full system energy breakdown



(b) BitX PE-only energy breakdown

Figure 11: full-system and PE-only energy breakdown for SqueezeNet.

Table 7: PE area and power breakdown @TSMC 28nm.

Precision	*BitX* (floating-point 32)		*BitX* (16b fixed point)
Item	Area (mm ^{2})	Power (mW)	Power (mW)
E-alignment	0.017 (43.60%)	11.15 (16.20%)	–
Bit Extraction	0.008 (20.10%)	0.04 (0.05%)	0.026 (0.07%)
16 CUs	0.003 (7.70%)	53.71 (78.30%)	35.81 (98.40%)
Misc&Control	0.011 (28.20%)	3.72 (5.40%)	0.576 (1.60%)
Total	**0.039**	**68.62**	**36.41**

4.4 Performance of the Fixed-point DNN

Accuracy. *BitX* is also feasible to 16b fixed-point DNNs, as part of its 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 *BitX accelerator* is not needed and could be power gated (Figure 3). The weights directly pass through to the "Bit-Extraction" module for sorting the probabilities of each bit row. As Table 5 shows, the accuracy is exactly equal to the non-pruned ResNet50 for *BitX-mild*. More promisingly, *BitX-mild* and *BitX-wild* both exhibit even higher accuracy than the non-pruned DenseNet121 and ResNext101.

The accuracy improvement attains up to nearly 1%. *We conclude that BitX could precisely pinpoint the useless bits in both floating-point and fixed-point DNNs.*

Speedup. As shown in Figure 8, *BitX-wild* exhibits up to 2x speedup over the original model. Reporting some of the results, for ResNet50, the speedup on *BitX-wild* is 1.67x; for DenseNet121, the datum is 1.79x. As for the *BitX-mild*, the largest speedup emerges at SqueezeNet1_1 – 1.12x. Other DNNs demonstrate tiny acceleration, primarily because each weight only has 16-bit width, setting $N = 10$ means 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.

4.5 Working with Software-based Pruning

As a hardware runtime pruning approach, *BitX* is orthogonal to any software-based pruning schemes. In this experiment, we use the famous object detection model – YoloV3 [28] as the benchmark DNN, including the genetic YoloV3 model and software-pruned model based on the structured channel pruning [22]. As presented in Table 6, *BitX-mild* still displays better performance than the genetic YoloV3 with 2.75x speedup. *BitX-wild* exhibits tiny-degraded accuracy but higher speedup – 4.98x. For the YoloV3+Slimming baseline, Bit-mild has 0.07% accuracy improvement and *BitX-wild* has less than 1.6% accuracy degradation. The speedup is very much considerable over the genetic YoloV3: YoloV3+Slimming+*BitX-wild* attains 14.76x; YoloV3+Slimming+*BitX-mild* attains 7.22x. This experiment clearly proves that our method is completely compatible to the previous software pruning schemes. The users could effortlessly obtain more superior speedup and accuracy by collaborating *BitX* with software pruning.

4.6 Comparison with SOTA Accelerators

In this subsection, we compare the two *BitX* representatives with the state-of-the-art fixed-point accelerators. Stripes [16] and Pragmatic [2] are two bit-serial accelerators. Stripes implements the MAC computation using bit-level arithmetic, but does 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. *BitX* targets two types of useless bits. The trivial bit 1s are also pruned which means the amount of 0 bits that can be skipped becomes more, hence yielding better speedup performance.

Speedup. As proved by Figure 9, the speedup over Pragmatic (normalized baseline) and stripes is 2.00x and 3.79x for *BitX-wild*; and 1.12x, 2.34x for *BitX-mild.* A more interesting observation is that the floating-point results are even better than the fixed-point *BitX*, still because of the limited 16-bit width. Floating-point weight has 24-bit mantissa and spawns even more useless bits after exponent matching. The functional flexibility provided by *BitX* also releases more AI tasks that could run on *BitX*. The users could freely customize their DNNs in the practical use.

Energy efficiency. Similar to the speedup result, the energy efficiency of *BitX* outperforms other accelerator baselines, but this time the 16b *BitX-wild* (2.00x, SqueezeNet, 1.68x ResNet50) demonstrates better result than the float-32 *BitX-mild* (1.24x, 1.22x). That is because the floating-point *BitX* has higher power consumption due to the "E-alignment" module which is power gated in the 16b mode. However, combined with the inference speed, *BitX-wild* (float 32) still wins for 2.01x and 1.98x better efficiency.

Energy breakdown. Our Xilinx V7 FPGA platform involves DDR3 memory. We use DRAMsys to estimate the runtime memory access energy. Figure 11 shows the energy breakdown from two aspects. Figure 11(a) shows the full-system energy breakdown and clearly the memory accesses dominate the energy consumption. Especially for the two 16-bit *BitX* representatives, the memory access energy could attain 98% and PE energy only occupies less than 2%. In Figure 11(b), we further decompose the PE-only energy for each *BitX* instance. CU energy dominates this time (63%, 89%, 78%, 78%), because we have 16 CUs with a large number of buffers to store the bit-pruned weights. For other modules, E-alignment and the control circuits consume around 16% and 5%~37% energy, respectively.

Area and Power breakdown. Under TSMC 28 nm technology node, *BitX* in floating-point 32 mode exhibits 0.039 mm^2 area. Table 7 illustrates the largest area is occupied by the E-alignment module (43.6%), because it involves frequent shifting operation and some of the wires are inevitably prolonged to avoid intersection. However, it is not the largest power consumer (only 16.2%) because no computation circuits are involved in this module. Comparatively, the 16 CUs occupy the smallest area (7.7%) but consume most of the power (78.3%) due to the internal arithmetic logic. *BitX* in 16-bit fixed-point mode powers down the E-alignment module and the CU arithmetic is based on the fixed-point activations, so its overall power consumption reduces to 36.41 mW, compared with 68.62 mW in the floating-point 32 mode.

5 CONCLUSION

In this paper, we propose a novel *hardware runtime pruning* method - *BitX*, to empower versatile DNN inference. By targeting the abundant bit-level sparsity and trivial bit 1s, it implements pruning on-the-fly in hardware without any software work. It precisely locates the essential bits by the proposed *BitX* pruning algorithm, and prunes away the trivial bits at different precisions including both floating point and fixed point. The empirical studies have proved the efficacy of *BitX*, by providing abundant sparsity, faster inference speed and lossless (or even higher) accuracy on various image classification and object detection models. We also hope *BitX* pruning methodology and the associate accelerator design would stimulate more insightful perspectives on hardware runtime pruning, to provide both promising DNN acceleration and excellent user experience at the same time.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China under grant No. 62002339 and grant No. U2030204, and the Strategic Priority Research Program of Chinese Academy of Sciences under grant No. XDB44030200, and the Key Research Program of State Key Laboratory of Computer Architecture under grant No. CARCH5301.

REFERENCES

[1] pytorch 1.6. https://pytorch.org/

[2] Jorge Albericio, Patrick Judd, Alberto Delmas, Sayeh Sharify, and Andreas Moshovos. 2016. Bit-pragmatic Deep Neural Network Computing. In *Proceedings of the International Symposium on Microarchitecture (MICRO).*

[3] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *Proceedings of the International Symposium on Computer Architecture (ISCA).*

[4] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2015. Structured Pruning of Deep Convolutional Neural Networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC).*

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *arXiv:2005.14165.*

[6] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. 2017. Deformable Convolutional Networks. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR).*

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805.*

[8] Petros Drineas, Ravi Kannan, and Michael W Mahoney. 2006. Fast Monte Carlo Algorithms for Matrices III: Computing a Compressed Approximate Matrix Decomposition. *SIAM Journal on Computing (SICOMP).*

[9] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William J. Dally. 2016. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA).*

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR).*

[11] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. 2016. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv:1607.03250.*

[12] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR).*

[13] IEEE 754. https://standards.ieee.org/standard/754-2019.html

[14] CoCo Dataset. https://cocodataset.org/

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR).*

[16] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, and Andreas Moshovos. 2017. Stripes: Bit-Serial Deep Neural Network Computing. In *Proceedings of the International Symposium on Microarchitecture (MICRO).*

[17] Matthias Jung, Christian Weis, and Norbert Wehn. 2015. DRAMSys: A Flexible DRAM Subsystem Design Space Exploration Framework. *IPSJ Transactions on System LSI Design Methodology (T-SLDM).*

[18] Alex Krizhevsky and Geoff Hinton. 2009. Learning multiple layers of features from tiny images. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR).*

[19] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, and Mostafa Mahmoud. 2019. Bit-Tactical: A Software/Hardware Approach to Exploiting Value and Bit Sparsity in Neural Networks. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[20] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, and Hoi Jun Yoo. 2018. UNPU: A 50.6 TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC).*

[21] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning filters for efficient convnets. In *Proceedings of the International Conference on Learning Representations (ICLR).*

[22] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. 2017. Learning Efficient Convolutional Networks through Network Slimming. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV).*

[23] Hang Lu, Xin Wei, Ning Lin, and Guihai Yan. 2018. Tetris: Re-architecting Convolutional Neural Network Computation for Machine Learning Accelerators. In *Proceedings of the International Conference On Computer Aided Design (ICCAD).*

[24] Hang Lu, Mingzhe Zhang, Yinhe Han, Qi Wang, Huawei Li, and Xiaowei Li. 2020. Architecting Effectual Computation for Machine Learning Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD).*

[25] Jian-Hao Luo and Jianxin Wu. 2017. An entropy-based pruning method for CNN compression. *arXiv:1706.05791.*

[26] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV).*

[27] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA).*

[28] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR).*

[29] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556.*

[30] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the International Symposium on Computer Architecture (ISCA).*

[31] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. 2014. Learning Spatiotemporal Features with 3D Convolutional Networks. *arXiv:1412.0767.*

[32] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. 2014. Non-local Neural Networks. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR).*

[33] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Proceedings of the Conference and Workshop on Neural Information Processing Systems (NeurIPS).*

[34] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, and Yunji Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *Proceedings of the International Symposium on Microarchitecture (MICRO).*