EBSP: Evolving Bit Sparsity Patterns for Hardware-Friendly Inference of Quantized Deep Neural Networks

Fangxin Liu[1,2], Wenbo Zhao[1], Zongwu Wang[1], Yongbiao Chen[1], Zhezhi He[1], Naifeng Jing[1], Xiaoyao Liang[1] and Li Jiang[1,2,3*]
[1]Shanghai Jiao Tong University, [2]Shanghai Qi Zhi Institute
[3]MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University

ABSTRACT

Model compression has been extensively investigated for supporting efficient neural network inference on edge-computing platforms due to the huge model size and computation amount. Recent researches embrace joint-way compression across multiple techniques for extreme compression. However, most joint-way methods adopt a naive solution that applies two approaches sequentially, which can be sub-optimal, as it lacks a systematic approach to incorporate them.

This paper proposes the integration of aggressive joint-way compression into hardware design, namely EBSP. It is motivated by 1) the quantization allows simplifying hardware implementations; 2) the bit distribution of quantized weights can be viewed as an independent trainable variable; 3) the exploitation of bit sparsity in the quantized network has the potential to achieve better performance. To achieve that, this paper introduces the bit sparsity patterns to construct both highly expressive and inherently regular bit distribution in the quantized network. We further incorporate our sparsity constraint in training to evolve inherently bit distributions to the bit sparsity pattern. Moreover, the structure of the introduced bit sparsity pattern engenders minimum hardware implementation under competitive classification accuracy. Specifically, the quantized network constrained by bit sparsity pattern can be processed using LUTs with the fewest entries instead of multipliers in minimally modified computational hardware. Our experiments show that compared to Eyeriss, BitFusion, WAX, and OLAccel, EBSP with less than 0.8% accuracy loss, can achieve 87.3%, 79.7%, 75.2% and 58.9% energy reduction and 93.8%, 83.7%, 72.7% and 49.5% performance gain on average, respectively.

1 INTRODUCTION

Deep neural network (DNN) models have been designed to solve real-world problems and have achieved significant success in many application domains [16, 22]. However, due to the ever-increasing model size of DNNs, the memory and computation overhead have increased dramatically, making the deployment on embedded and edge devices difficult. For instance, the latest DNN models [7, 13] contain trillions of parameters and requires GFLOPs (giga floating-point operations) of computations in a single inference, making it a challenging task to perform on-device inference.

To address this challenge, the resource-constrained edge computing platforms require two crucial supports. The first one is the specialized hardware acceleration for DNN inference. For example, EIE [12] utilizes sparse neural network models, while requires additional hardware overhead to represent the sparse data format. Laconic [23] tries to optimize the design of multipliers based on Booth coding, which converts inputs into a sequence of signed terms and sequentially multiplies and accumulates them.

The second is the model compression technique, such as network sparsification [11] and network quantization [14], which are widely used for inference. To be specific, sparsification makes network sparse, quantization reduces network precision and both reduce the required memory bandwidth. Among them, non-uniform quantization method, represented by the deep compression [7], applies k-means to cluster the weights, and the quantized values are denoted as indexes. Meanwhile, power-of-2 based quantization (e.g., SP2 [3], AFP [17]) maps the weight values to the exponential space, and then simplifies the multiplication operation into shift operation.

In spite of the significant advances, we identify three challenges for existing compression techniques: 1) quantization methods focus on improving the compression rate of ultra low-precision DNN models, resulting in significant accuracy losses, for example, $> 5\%$ under binary and $> 2\%$ for ternary quantization [7]; 2) generally, sparsification methods can achieve higher performance with greater compression than quantization methods. However, major drawbacks are the additional indexing overhead for addressing non-zero elements and irregular access/execution patterns [11]; 3) sparsification or ultra low-precision quantization methods always introduce ancillary overheads in the circuit or architecture design, which is still complex and implementation-unfriendly [11, 12].

This paper focuses on the DNN compression technique, which becomes imperative to the DNN hardware acceleration, especially on FPGA and ASIC platforms [7]. Thus, we revisit the quantization process from a new angle of bit-level sparsity: the reduction of the precision of an operand can be taken as forcing one or more bits among the operand to be zero, where a lower significant bit is more likely to be zero. Alternatively, quantization can be viewed as increasing bit-level sparsity among the operand. By considering the distribution of bits in the model parameters, we propose the co-design method where the hardware-friendly sparsity patterns are formed under low-cost constraints and enjoy the benefits of quantized DNNs with slight hardware modification. Our contributions can be outlined as follows:

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530660

- We propose a soft quantization scheme that evolves inherently regular bit-sparsity patterns in training and achieves an accuracy that matches the high bit-width quantization.
- We design the Look-Up Table (LUT) based approach for accelerating the DNN inference, which effectively incorporates bit sparsity pattern and quantization for performance gains while reducing the energy consumption of multiplication.
- We describe the minimum required modifications and execution flow on the hardware platform to support such scheme effectively.

2 BACKGROUND AND MOTIVATION

2.1 Network Quantization

Quantization algorithms compress the network by reducing the number of bits required for weight and activation. Representative quantization methods can be categorized into two classes:

1) Uniform quantization, which is one of the most widely-used quantization scheme, include binary, ternary, and fixed-point [14]. Binary and ternary quantization uses extremely low bit-width to represent DNN models, that is, 1-bit (-1, +1) for binary quantization and 2-bit (-1, 0, +1) for ternary quantization. Although binary and ternary quantization can significantly reduce the operand precision and simplify the hardware implementation, it introduces a non-negligible accuracy loss (e.g., > 5% accuracy drop for binary). In contrast, as represented by INT8 [14] and LSQ [8], fixed-point quantization schemes use modest bit-width to achieve comparable accuracy as the original model. Weights and activations are quantized to the nearest integer up to a scaling factor that is shared through all the weights or activations in the same layer:

$$\hat{w} = \alpha \cdot clip(round(w/\alpha), -2^{m-1}+1, 2^{m-1}-1), \quad (1)$$

where $clip(x, \max, \min)$ clamps the value x into range $[\max, \min]$ and $\hat{w}$ is the quantized value of w with m-bit fixed-point quantization.

2) Non-uniform quantization, which uses the distribution of weights and activations. One way is to cluster weights into several groups [7, 11]. However, such method didn't bring computation benefits, since the cluster centers are still stored as floating numbers. Others are power-of-2 based methods that quantize weights to power-of-2s up to a scaling factor [3, 17]. These methods utilize the fact that weights and activations have a denser distribution near zero. Thus, they can replaces the expensive multiplications by cheap shifting operations. Although power-of-2 quantization can simplify the hardware implementation by eliminating multiplication, it cannot improve the accuracy by increasing bit-width as uniform quantization. This is because the interval between quantization levels increases exponentially with bit-width, resulting in finer resolution near the mean with increasing the bit-width while the tails (weights with large value) still remain coarse.

2.2 DNN Accelerators for Network Sparsification and Quantization

General network sparsification and quantization are widely used for inference. To be specific, sparsification reduces the number of operands, and quantization reduces the bitwidth of the data flowing through a neural network model. SnaPEA [1] exploits activation sparsity to shorten the computation time of the convolution operation, where the data format is still the 16-bit fixed-point number. WAX [10] uses a deeply distributed memory hierarchy, leads to data being moved with small overhead, and quantizes operands from 32-bit to 8-bit. BitFusion [24] proposes to execute quantized neural network models. OLAccel [20] is the mixed-precision accelerator that utilizes quantization with 4-bit and 16-bit MACs. These designs mainly cater to the general single-way network sparsification or quantization. Consequently, their architectures struggle for peak performance because it is difficult to find a perfect fitted compression method to incorporate sparsity in quantization.

Our work shares conceptual similarities with prior works on quantization, however, it distinguishes itself by deriving the proposed *bit sparsity pattern* to minimize hardware implementation. Such a sparsity pattern is evolved through training to yield regular bit distributions, rather than imposing an overall bit-width constraint by quantizing the model. As a result, EBSP can deliver a competitive accuracy on par with high-precision quantization, while the forced sparsity constraint keeps the hardware implementation overhead at a minimum level.

3 ALGORITHM FOR QUANTIZATION WITH BIT SPARSITY PATTERN

In this section, we propose a novel quantization scheme combined with the hardware-friendly bit sparsity pattern, which enjoys the non-multiplication operations for the DNN inference while achieving negligible inference accuracy loss.

3.1 Coupling Quantization with Hardware

The most success of the quantization with low bit-width (e.g., HAQ [7], DeepCompression [11], and power-of-2 quantization [3]) can be largely attributed to introducing ancillary overheads, such as indexes. Hence, it is not trivial to quantize neural networks with low bit-width for both inputs and weights without indexes aiding or accuracy loss during the inference. In this work, EBSP aims at eliminating multiplication operations in the (quantized) DNN inference models (as the low-bit-width quantization scheme), and at the same time, is designed to address the non-negligible accuracy loss of quantization with low bit-width. The proposed hardware-friendly quantization scheme incorporating the bit sparsity pattern can be considered as a variant of the non-uniform quantization. It is possible to merge the quantization and sparsification constraints together.

Since multiplication operation is the most widely used compute operation in the DNN workloads. Thus, we attempt to design a novel hardware-friendly quantization method that takes full advantage of LUTs to replace multipliers, considering LUTs can be reconfigured to support different bit-width and the reduced operand precision enables fewer LUT entries [9, 21]. This approach minimally increases the memory area by introducing only hardware that assists in combining LUT entries to realize multiplications.

The LUT-based scheme inevitably face the problem that there is an excessive number of entries to cover all the possible combinations of weights and activations. The number of entries in the LUT plays a significant role in determining the system performance. For example, to compute a multiplication with INT8 quantization in one cycle, 65,536 ($2^8 \times 2^8$ combinations) entries are needed in the LUT. If the partial sum result is saved with single precision (16-bit),

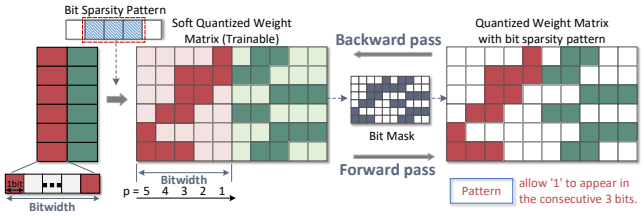

Figure 1: Quantization with bit sparsity pattern and weight update in training.

128 MB of momery is needed to store them, which makes the design very impractical. In order to further improve the LUT-based scheme efficiency, we carefully optimize the LUT overhead in conjunction with the quantization method.

Inspired by the fact that only the bits from the most significant bit of a number takes parts in the computation, and the leading '0' bits are redundant, we introduce an innovative co-design paradigm of compression and hardware to enable fine-grained bits exploited at a low cost and enjoy the benefits of the quantized DNNs. This method quantizes weights and activations into adaptive floating-point numbers. Specifically, each layer shares a shift factor k, and each value has its exponent part e and mantissa part m. In such case, each value can be represented as

$$x = 2^k \left(2^e \cdot \left(1 + m \cdot 2^{-n_{\text{man}}}\right)\right) = 2^k \left(2^e \cdot M\right) \tag{2}$$

where n_{man} is the bit-width of the mantissa m. Given a weight w and an activation a, the quantized multiplication of them can be written as

$$\begin{aligned} a \cdot w &= \left[2^{k_a} \left(2^{e_a} \cdot M_a\right)\right] \times \left[2^{k_w} \left(2^{e_w} \cdot M_w\right)\right] \\ &= 2^{k_a + k_w} (2^{e_a + e_w} \cdot (M_a M_w)) \end{aligned} \tag{3}$$

From Eq. (3), we see that the only need to implement the multiplication of M_a and M_w. Then, the result can be derived after element-wise shifting by $e_a + e_w$ and the layer-wise shifting by $k_a + k_w$. Thus, we use LUTs to realize the computation of M_a and M_w, with $2^{n^a_{\text{man}} + n^w_{\text{man}}}$ entries, much fewer than the computation of whole w and a. Then, we introduce the evolving sparsity pattern in quantization to further decrease the bit-width of M_a and M_w, which lead to yet further computational benefits.

3.2 Evolving Sparsity Pattern in Quantization

We propose a novel integrated training method that evolves sparsity patterns by enforcing the constraint on the bit distribution (called bit sparsity constraint) by performing soft quantization in each training iteration. We divide the training process into three phases sequentially according to the execution order: masking, forward passing, and backward passing, respectively. During the masking phase, the weight matrix is quantized to the target bit-width and imposes bit sparsity constrain in the bits within the target bit-width. As shown in Fig. 1, bit sparsity constraint is that a maximum of $s = 3$ consecutive '1's exists in the bits within the weights at a given bit-width. Next, the forward pass is performed using the quantized weight matrix with the bit sparsity constraint, which can be formulated as Eq. (2) and Eq. (3)

In the masking phase, the position of the most significant bit is located and denoted as the exponent part e of the quantized weight. Then, the bit mask is generated that covers at most s '1' bits after the most significant bit, the position $p = e$ to $e + s - 1$ (p from 5 to 1 in Fig. 1). Then, the quantized weight is generated by the bit-wise multiplication of the original weight and the mask. In the forward phase, the original weight matrices are quantized prior to passing through masking. The layer computations are carried out with the bit sparsity pattern of the quantized weight matrices.

In the backpropagation phase, we aim at training the weights towards the quantization sparsity pattern while maintaining network accuracy. Therefore, we add a normalization term

$$\frac{\lambda}{2} \sum \cdot (w - w_q)^2 \cdot 2^{-e} \tag{4}$$

to the loss to decay the weights toward the quantized one. Then, the gradient will be calculated as

$$\frac{\partial L'}{\partial w} = \frac{\partial L}{\partial w} + \lambda (w - w_q) \cdot 2^{-e} \tag{5}$$

For large weights whose exponent e is large, the gradient is still focused on the first term to make sure that the weight is updated in the direction to reduce the loss and increase accuracy. On the other hand, for small weights whose exponent e is small, the second term will account for a larger part in the gradient, guiding them towards the quantized value and reducing the quantization error.

In addition, we formulate the cost $C\left(N_q, n_{\text{exp}}\right)$ in a bit manner: the multiplication in EBSP need $2^{n^a_{\text{man}} + n^w_{\text{man}}}$ LUT entries, which also determines the bits for adders (more LUT entries, the larger the number of bits for adders). Therefore, we combine the cost function Eq. (6) with accuracy as the overall objective of training a DNN model based on ADMM-regularized optimization [19]:

$$C = 2^{n^a_{\text{man}} + n^w_{\text{man}}} \tag{6}$$

$$\mathcal{L} = \mathcal{L}(W_q^{1:L}) + \alpha \sum_{l=1}^{L} C_l \tag{7}$$

where $\mathcal{L}(W_q^{1:L})$ is the original entropy loss evaluated with the quantized weights W_q, L denotes the number of layers in the DNN model, α is a hyperparameter controlling the balance between the entropy and cost term. As a result, our evolving bit sparsity pattern for incorporating sparsity in quantization offers a viable path towards efficient hardware inference will be discussed in Section 4.1.

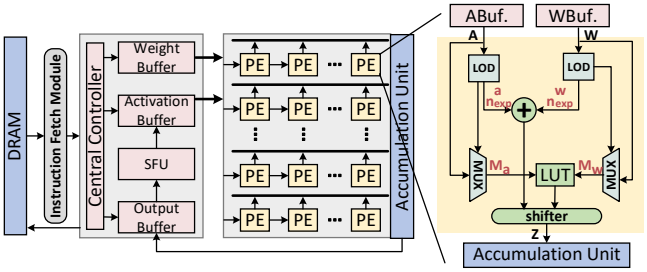

Figure 2: Block diagram of the PE implementation in EBSP.

4 ARCHITECTURE FOR QUANTIZATION WITH BIT SPARSITY PATTERN

In this section, we discuss the efficient LUT-based PE implementation, the organization of EBSP, and its execution flow support for DNN workloads incorporated in the proposed EBSP architecture.

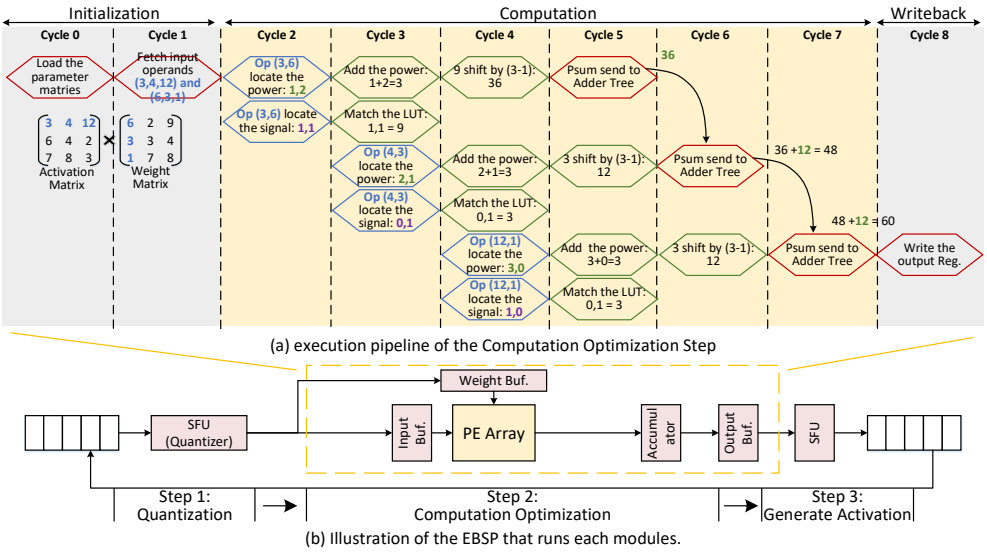

Figure 3: The data computation for convolution. (a) An example of execution pipeline with the EBSP-based computation step includes initialization, computation, and writeback phase. The blue blocks indicate steering logic, while the green blocks indicate arithmetic logic. (b) Illustration of the EBSP that runs various modules for computation.

4.1 Non-Multiplication Engine (NME)

In this section, we will implement the non-multiplication engine that can efficiently quantize the given weight and use LUT instead of the multiplier. As is shown in Fig. 2, the processing unit (PE) is designed with two components: steering logic and arithmetic logic. In detail, the steering logic is composed of leading one detector (LOD) that dynamically locate the most significant '1' bit and a multiplexer that extracts significant digits to send to the LUT. In contrast, based on the *bit sparsity pattern*, arithmetic logic is composed of a LUT with few entries, adder, and shifter (e.g., barrel shifter) that implements the multiplication of Eq. (3). Take the 5-bit quantization as an example, 1-bit sign, 2-bit exponent, 2-bit mantissa for weight and 2-bit exponent, 3-bit mantissa for activation (0-bit sign since ReLU activation function is used). First, $e_a + e_w$, the 2-bit addition, is realized by adder; meanwhile, EBSP uses LUTs to realize $M_a M_w$. Finally, the output of the LUT through the shifter, which is shifted by $e_a + e_w$ bits yield by the adder. In this way, we can achieve the circuit-implementation-friendly for the multiplication. In addition, our method has a better adaptability since LUT entries only fix the bit-width of mantissa and different bit-widths of exponents can use the same LUT. In this example, only $2^{2+3} = 32$ LUT entries are needed to stored pre-calculated constant, which is quite impressive for implementation. Our architecture's other benefit is not restricting designers from using their preferred design as the smaller core multiplier. If a more efficient multiplier can be designed in the future, the arithmetic logic can be replaced. The much smaller arithmetic logic justifies the addition of steering logic, which provides significant overall power and area savings than the exact multiplier.

4.2 Execution Flow with NME

With our NME, we achieve accelerating the quantized NN through minor hardware modifications to boost hardware utilization and inference performance significantly, as explained in Fig. 3:

Step 1: Data preparing (Quantization). At the beginning of the EBSP run, the quantized weights are loaded into an on-chip buffer, i.e., the weight buffer as shown in Fig. 3 (b). We use the EBSP quantized data throughout the execution flow, where the quantized data consists of the exponent and mantissa, as expressed in Eq. (3). Therefore, the activation of each layer will also be in the same format. To use these results as input activations and multiply them with the low precision weights, we need first to quantize them using the quantizer and then store them in the input buffer.

Step 2: Computation Optimization. As shown in the Fig. 3 (b), after quantization, the low precision inputs and weights are stored in the input buffer and the weight buffer. We then perform efficient MAC computations using the PE array. Fig. 3(a) shows a detailed illustration of the EBSP-based computation steps with an example. The matrix multiplication operation with once superposition of adjacent exponents. In the initialization phase, the parameter matrix (Cycle 0) is loaded, and the operands involved in the calculation (Cycle 1) are fetched. Since the initialization is performed only once at the beginning, which cause the small overhead of fetching operands, and only the cycles of computation is proportional to the number of multiplications. In the next six cycles (computation phase), three shifts, three additions and two accumulations are performed to produce an element of the output matrix. In Cycle 2, according to steering logic, the most significant digit '1' is located and combined with our quantization method to obtain the power and the signal whether the superposition is needed. In Cycle 3, the addition is performed since the operands of multiplication are simplified to power-of-2. In Cycle 4, a left-shift operation is performed on the result of matching the LUT according to the adder's output. In Cycle 5, the psum is fed into the adder tree for accumulation. Then, the subsequent operands are repeated for Cycle 2-4. The output will be written back in Cycle 8. This execution pipeline continues until the matrix multiplication is completed.

Step 3: Generate Activation. The results of the PE array are accumulated with the partial sum and sent to the Special Function Unit (SFU) to calculate the final output. The SFU performs activation functions (e.g., ReLU), pooling functions (e.g., average pooling and max pooling), or even other operations.

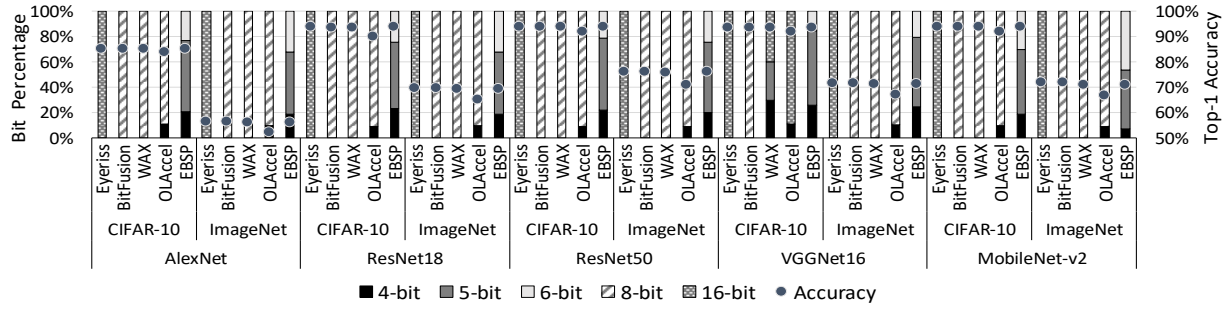

Figure 4: Comparison of DNN accuracy and percentage of bit-width for different networks.

It is worth noticing that other quantization methods such as INT8 quantization [14] are also applicable to our EBSP scheme, which will be discussed in Section 5.2.

5 EXPERIMENTS

5.1 Experimental Methodology

1) Validation on NN Accuracy. We evaluate the quantization framework on CIFAR-10 [15] and ImageNet [6] datasets on widely applied CNN networks for image classification, including AlexNet [16] VGG-16 [7], ResNet-18, ResNet-50 [13], and MobileNet-v2 [22]. The EBSP algorithm is implemented by PyTorch. We use pre-trained network models from Pytorch Model Zoo as the basis for EBSP algorithm. The network structure is derived from the model in the torchvision [18]. We reconstruct the convolutional layers, which insert the quantization process for the weights and activations before the built-in convolution function call. The experiment runs on CUDA 10.2 environment with Tesla V100 GPU.

Table 1: Configurations of different accelerators under 45-nm standard-cell library.

	Eyeriss [4]	BitFusion [24]	WAX [10]	OLAccel [20]	EBSP
Bit-width	16-bit	4-bit	8-bit	4&16-bit	6-bit (3)†
Data Format	Integer	Integer	Fixed-point	Integer	Integer
# PEs	224	3168	102	2499	4818
Area (mm2)	0.32	0.32	0.32	0.32	0.32

† this denotes the length of bit sparsity pattern, which determines LUT entries.

2) Modeling Accelerator Architecture. Since the Eyeriss architecture [4] is widely used as a baseline in many accelerators [10, 23] and only relevant results are provided in these works, we also choose it as a baseline. Meanwhile, for a fair comparison, we use the same global buffer capacity (5 MB) and memory bandwidth for all these accelerators and use CACTI [2] to estimate it that can satisfy our design goals. In addition, we use the 45nm technology library and Synopsys Design Compiler [5] to study the area and energy of the PE unit that we designed with various bit-width and data-format. Tab. 1 shows the configuration of all accelerators in our experiments. Under 500MHz PE frequency, we verify that the required memory bandwidth is much smaller than the typical memory bandwidth provided by DDR3. Thus, we can sustain a non-blocking convolution. Meanwhile, the EBSP architecture can be extended to a larger number of PEs under the same area budget.

5.2 Experimental Results

1) Accuracy, Performance and Energy Consumption. Fig. 4 first shows the DNN accuracy for the five networks with CIFAR-10 and ImageNet datasets. Taking ResNet-50 as an example, compared to Eyeriss, our EBSP shows nearly no accuracy loss for CIFAR-10 compared to Eyesiss (full INT16), BitFusion (full INT8) and WAX (full fixed-point 8-bit), and a 2.2% accuracy improvement over the OLAccel. For ImageNet, our EBSP shows a 0.31% accuracy loss compared to Eyesiss, WAX and BitFusion and a significant 4.32% accuracy improvement over OLAccel. Fig. 4 also shows the percentage of bitwidth used in the computation for these designs. Both EBSP and OLAccel take full advantage of low-bit quantization, where the computation in our EBSP is done with LUT and is realized by the bit sparsity pattern. However, OLAccel performs quantization with fixed across all the layers and datasets. In contrast, EBSP is evolved to generate the bit sparsity pattern in the quantization and takes into account the cost of LUTs and network accuracy in training, so that the number of bits used for weight and activation values can vary from layer to layer. As demonstrated in the up-to-date compact MobileNet-v2 which has a much smaller parameter size, our EBSP is more robust for preserving DNN accuracy (0.78% loss) with 7% of 4-bit, 51% of 5-bit and 42% of 6-bit percentage than OLAccel (5.4% loss) with 85% of 4-bit percentage.

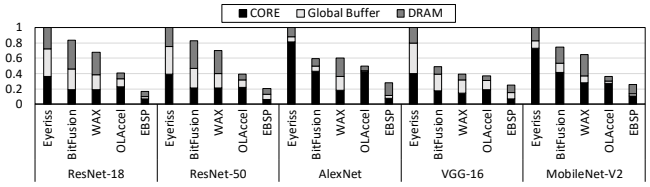

Figure 5: Energy breakdown of various networks.

Fig. 5 shows the the energy consumption of various accelerators for the five networks, which is decomposed into DRAM, global buffer and processing cores (Core). For example, in ResNet-50, compared to Eyeriss, BitFusion, WAX and OLAccel, EBSP consumes 87.3%, 79.7%, 75.2% and 58.9% less energy, respectively. The energy reduction of EBSP against other accelerators is mainly due to the simplification and reduced precision on PEs, and the resultant narrower bit-width data transferred between DRAM and global buffer.

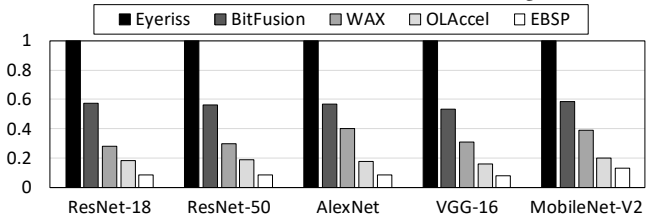

Figure 6: Performance of various DNNs.

Fig. 6 shows the total execution cycles on different accelerators for the five networks, which are all normalized to Eyeriss. Still taking ResNet-50 as an example, compared to Eyeriss, EBSP achieves

nearly 93% performance improvement because EBSP mostly uses simplified PE to realize the MACs for DNNs. However, only EBSP considers fine-grained bit sparsity in the quantization, EBSP achieves the highest throughput (93.87% performance improvement compared with Eyeriss) among these designs. Compared to OLAccel with 16-bit for the first and last layer and 4-bit for the rest layers, EBSP can achieve average 49% performance improvement thanks to the larger number of PEs under the same area budget (see Table 1) in our scheme. Note that, for benchmark tests like ImageNet, EBSP delivers a much larger speedup with only 0.42% accuracy loss than OLAccel achieved at the cost of about 5% accuracy loss.

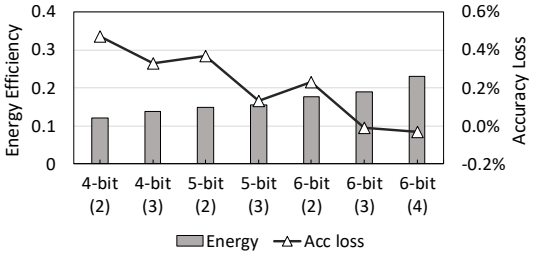

Figure 7: Analyzation of the pattern length. m-bit (n) denotes the value is quantized to m-bit with the bit sparsity pattern of length n.

2) Design Space Exploration. Fig. 7 studies the impact of pattern length on ImageNet. This plot reports the average accuracy loss and energy across the five networks. Note that EBSP has the capability of tuning the length of bit sparsity pattern to sustain the same accuracy levels as Eyesiss, while gaining notable energy efficiency. Specifically, EBSP incurs average 0.31% accuracy loss compared to Eyesiss, BitFusion and WAX, while OLAccel has 4.3% accuracy loss instead. In terms of energy efficiency, EBSP with different pattern length reduces the energy consumption by an average of 87% compared to Eyeriss. Moreover, longer pattern length leads to higher DNN accuracy, but it will increase the energy consumption. From this plot, we find that quantized DNNs with bitwidth of 5-bit and pattern length of 3, EBSP achieves an optimal point (0.13% accuracy loss with 97.3% energy reduction over Eyeriss) on ImageNet.

Fig. 8 shows the impact of bit sparsity pattern on INT8 for ResNet-50. We sweep pattern length from 2 to 7. Generally, a shorter pattern length means fewer LUT entries and allows for less energy consumption, but it will degrade the network accuracy. In Fig. 7, we normalize the energy efficiency to Eyeriss. We find that INT8 with the pattern length of 7, the energy efficiency is inferior to that of the direct INT8 scheme (i.e., BitFusion), which is caused by the excessive number of LUT entries. However, as the pattern length decreases, the energy efficiency increases. We see that the bit sparsity pattern of the length of 4 and 5 will be most beneficial for the energy-saving with almost no accuracy loss (0.53% loss for length of 4 and 0.27% loss for length of 5).

6 CONCLUSIONS

Sparsity and quantization are appealing tools for resource-efficient DNN design. However, it is challenging to incorporate sparsity into quantization and convert it to practical benefits. We have introduced a novel methodology to form bit sparsity patterns in quantization-aware training and reap the full advantages of sparsity and quantization while reserving better DNN accuracy. Proposed quantization

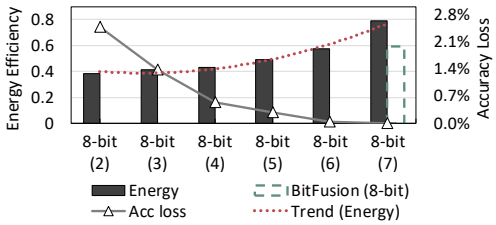

Figure 8: Analyzation of the bit sparsity pattern on INT8.

with the bit sparsity pattern allows the design of LUT-based PEs to replace multiplication with minimally modified existing hardware platforms, thus delivering remarkable performance benefits. Our evaluation shows that the proposed EBSP scheme outperforms other similar schemes in performance, energy or accuracy.

ACKNOWLEDGMENTS

This work was partially supported by the National Natural Science Foundation of China (NSFC) (Grant No. 61834006 and 62102257) and the National Key Research and Development Program of China (2018YFB1403400). We thank Wu Wen Jun Honorary Doctoral Scholarship, AI Institute, Shanghai Jiao Tong University. Li Jiang is the corresponding author (ljiang_cs@sjtu.edu.cn).

REFERENCES

[1] Vahideh Akhlaghi et al. 2018. SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks. In *ISCA*.
[2] Rajeev Balasubramonian et al. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *TACO* (2017).
[3] Sung-En Chang et al. 2021. Mix and Match: A novel FPGA-centric deep neural network quantization framework. In *HPCA*. IEEE.
[4] Yu-Hsin Chen et al. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *JSSC* (2016).
[5] Synopsys Design Compiler. 2019. [Online]. Available: https://www.synopsys.com/support/training/rtlsynthesis/design-compiler-rtl-synthesis.html.
[6] Jia Deng et al. 2009. Imagenet: A large-scale hierarchical image database. In *CVPR*. Ieee, 248–255.
[7] Lei Deng et al. 2020. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proc. IEEE* 108, 4 (2020), 485–532.
[8] Steven K Esser et al. 2020. Learned step size quantization. In *ICLR*.
[9] Michael Gautschi, Michael Schaffner, et al. 2016. 4.6 A 65nm CMOS 6.4-to-29.2 pJ/FLOP@ 0.8 V shared logarithmic floating point unit for acceleration of nonlinear function kernels in a tightly coupled processor cluster. In *ISSCC*. IEEE.
[10] Sumanth Gudaparthi et al. 2019. Wire-Aware Architecture and Dataflow for CNN Accelerators. In *MICRO*.
[11] Song Han et al. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *ICLR*.
[12] Song Han et al. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* (2016).
[13] Kaiming He et al. 2016. Deep residual learning for image recognition. In *CVPR*.
[14] Benoit Jacob et al. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*.
[15] Alex K. et al. 2009. Learning multiple layers of features from tiny images.
[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* (2017).
[17] Fangxin Liu et al. 2021. Improving Neural Network Efficiency via Post-training Quantization with Adaptive Floating-Point. In *ICCV*.
[18] Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the Machine-Vision Package of Torch. In *MM*.
[19] Wei Niu et al. 2020. Patdnn: Achieving real-time DNN execution on mobile devices with pattern-based weight pruning. In *ASPLOS*.
[20] Eunhyeok Park et al. 2018. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *ISCA*.
[21] Akshay Krishna Ramanathan et al. 2020. Look-up table based energy efficient processing in cache support for neural network acceleration. In *MICRO*. IEEE.
[22] Mark Sandler, Andrew Howard, Menglong Zhu, et al. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*. 4510–4520.
[23] Sayeh Sharify et al. 2019. Laconic Deep Learning Inference Acceleration. In *ISCA* (Phoenix, Arizona) *(ISCA '19)*. 304–317.
[24] Hardik Sharma et al. 2018. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *ISCA*.