FlexBCM: Hybrid Block-Circulant Neural Network and Accelerator Co-Search on FPGAs

Wenqi Lou, *Member, IEEE*, Yunji Qin, Xuan Wang, Lei Gong, *Member, IEEE*, Chao Wang, *Senior Member, IEEE*, and Xuehai Zhou

***Abstract*—Block-circulant matrix (BCM) compression has garnered much attention in the hardware acceleration of convolutional neural networks (CNNs) due to its regularity and efficiency. However, constrained by the difficulty of exploring the compression parameter space, existing BCM-based methods often apply a uniform compression parameter to all CNN models' layers, losing the compression's flexibility. Additionally, independently optimizing models or accelerators makes achieving the optimal tradeoff between model accuracy and hardware efficiency challenging. To this end, we propose FlexBCM, a joint exploration framework that efficiently explores both the parameter compression and hardware parameter space to generate customized hybrid BCM-compressed CNN and field-programmable gate array (FPGA) accelerator solutions. On the algorithmic side, leveraging the idea of neural architecture search (NAS), we design an efficient differentiable sampling method to rapidly evaluate the accuracy of candidate subnets. Additionally, we devise a hardware-friendly frequency domain quantization scheme for BCM computation. On the hardware side, we develop the efficient and parameter-configurable convolutional core (ConvPU) alongside the BCM computing core (BCMPU). The BCMPU can flexibly accommodate different compression parameters at runtime, incorporate complex-number DSP packing and conjugate symmetry optimizations. For model-to-hardware evaluation, we construct accurate latency and resource consumption models. Moreover, we design a fast hardware generation algorithm based on the coarse-grained search to provide prompt feedback on the hardware evaluation of the current subnet. Finally, we validate FlexBCM on the Xilinx ZCU102 FPGA and compare its compressed CNN-accelerator solutions with previous state-of-the-art works. Experimental results demonstrate that FlexBCM achieves 1.21–3.02 times higher-computational efficiency for ResNet18 and ResNet34 models while maintaining an acceptable accuracy loss on the ImageNet dataset.**

Manuscript received 31 July 2024; accepted 1 August 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4501600 and Grant 2022YFB4501603; in part by the National Natural Science Foundation of China under Grant 62102383, Grant 61976200, and Grant 62172380; in part by the Jiangsu Provincial Natural Science Foundation under Grant BK20210123; and in part by the Youth Innovation Promotion Association CAS under Grant Y2021121. This article was presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding authors: Lei Gong; Chao Wang.)*

Wenqi Lou is with the School of Software Engineering, University of Science and Technology of China, Hefei 230026, China, and also with the Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou 215123, China (e-mail: louwenqi@ustc.edu.cn).

Yunji Qin, Xuan Wang, Lei Gong, Chao Wang, and Xuehai Zhou are with the School of Computer Science, University of Science and Technology of China, Hefei 230026, China (e-mail: leigong0203@ustc.edu.cn; cswang@ustc.edu.cn).

Digital Object Identifier 10.1109/TCAD.2024.3439488

***Index Terms*—Algorithm-hardware co-exploration, convolutional neural network (CNN) compression, field-programmable gate array (FPGA).**

I. Introduction

CNNS HAVE achieved a series of remarkable achievements in computer vision [1], [2], [3]. However, their ever-increasing computational and memory volume makes their deployment challenging, especially in resource-constrained embedded scenarios. To this end, model compression has emerged as an effective method to reduce model redundancy [4], [5], [6]. Early unstructured pruning reduces the model's size but introduces irregularity in computation and memory access, significantly complicating the hardware design. Therefore, researchers have subsequently proposed regular compression methods. Among them, block-circulant matrix (BCM) compression has become a promising technique for deploying neural networks [7], [8], [9] on field-programmable gate arrays (FPGAs) due to its regular structure and expressive power.

Despite the remarkable results of BCM compression in model deployment, the current BCM-based hardware acceleration works still face the following limitations.

1) *Constrained Compression Space:* Due to the FFT/IFFT operations involved in the computation, the BCM-based work usually sets the block size (*BS*) to a power of 2, e.g., 4, 8, and 16. Then, all layers in the model are subjected to a uniform compression parameter [7], [8], [9]. This scheme limits the compression space and neglects that convolutional layers vary in their sensitivity to compression. Fig. 1 illustrates the variation in accuracy among different compression schemes with a similar compression ratio for the ResNet18 (RN18) and ResNet34 models.
2) *Separate Model Compression and Hardware Design:* Previous works typically rely on expert experience to set appropriate model compression parameters and corresponding hardware design parameters. Although separate optimization at the algorithmic and hardware levels is feasible, research has shown that there are interactions between algorithms and hardware, and optimization at different stages often yields only suboptimal solutions [10], [11], [12].

However, solving the above problem encounters the following challenges.

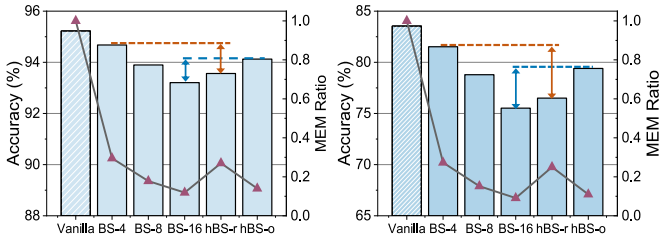

Fig. 1. Effect of block size on model accuracy in BCM compression for (a) RN18 on CIFAR-10 and (b) RN34 on ImageNet-100 datasets. "Vanilla" denotes the uncompressed model; "hBS" indicates layer-wise hybrid block size selection; and "-r/o" indicates random or optimized selection.

1) At the algorithmic level, hybrid BCM compression with layer-wise granularity entails a considerably larger search space. For instance, RN34 has a compression space greater than 10^{19}, and the expanded compression space does not guarantee good accuracy (e.g., "hBS-r" in Fig. 1). Training all possible subnets to obtain the accuracy ranking involves prohibitive time costs.
2) At the hardware level, previous accelerators based on BCM compression adopt a fixed dataflow. After static configuration, the accelerators cannot support different block sizes at runtime. Hence, a flexible computing core is required to support compressed convolutional layers with different block sizes.
3) At the model-to-hardware evaluation level, in addition to accuracy evaluations, joint search considering layer-wise compression/hardware parameters necessitates frequent feedback on the subnet's hardware metrics. Therefore, accurately and promptly generating optimized accelerators for subnets is also vital to the quality and cost of the co-search [13], [14], [15].

To address the above challenges, we propose FlexBCM, a joint search framework for layer-wise compression and hardware parameters to balance model accuracy and hardware efficiency. First, regarding the accuracy evaluation of subnets, we establish a supernet structure where each layer encompasses all candidate operators. By designing effective differentiable sampling methods, we can address the problem of operator selection using gradient optimization. Second, for convolutional layers with different parameter configurations, we devise dedicated hardware computation cores to ensure the execution efficiency of the model. Particularly, a highly optimized BCM computing core (BCMPU) is designed to support different compression parameters flexibly. Finally, in the hardware evaluation of subnets, we model the hardware computation cores and design a rapid hardware generation algorithm using genetic algorithms to achieve rapid feedback on hardware evaluation.

In summary, this work makes the following contributions.

1) To avoid the heavy retraining overhead of numerous subnets, we design a supernet based on weight sharing and propose a simple yet effective differentiable sampling method to assess candidate subnets' accuracy. Besides, we develop a hardware-friendly frequency-domain quantization scheme to facilitate hardware gains.

TABLE I
KEY PARAMETERS AND VARIABLES FOR THE DESIGN

Acronyms	Description
R, C	Rows and columns of the feature map
N, M	Input and output channels of the feature map
K, BS	Kernel size, the block size in BCM compression
$\mathcal{O}, \mathcal{P}$	Operator search space and operator selection probability
T_r, T_c	The tiling factor on the row and column dimension
T_n, T_m	The unroll factor on the input and output channels
$\mathcal{D}, \mathcal{R}$	Usage of digital signal processing (DSP) and BRAMs
$F, \mathcal{L}$	Hardware parameter variables, latency value

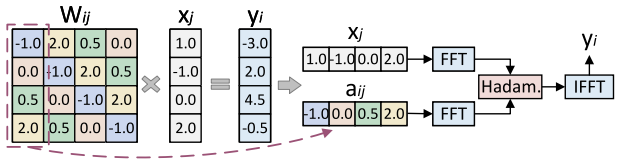

Fig. 2. BCM compression flow of matrix-vector multiplication.

2) To efficiently and flexibly support different compression parameters, we design the customized convolutional core and BCM core, where the BCM core can accommodate different compression parameters at runtime and contains targeted optimizations, such as complex-number DSP packing and conjugate symmetry.
3) To rapidly provide the hardware evaluation of the subnets, we accurately model the latency and resource consumption of the computational cores, and based on this, we design a heuristic hardware generation algorithm based on the coarse-grained search and parallel optimization.
4) We validate FlexBCM on the Xilinx ZCU102 FPGA. Experimental results demonstrate that FlexBCM effectively explores the joint search space in a brief time (1 GPU day), and compared to previous works, the searched solutions achieve 1.21–3.02 times higher-computational efficiency with acceptable accuracy degradation.

II. Background and Related Work

In this section, we introduce the basis of the BCM compression, the application of mainstream neural architecture search (NAS) methods, and related work. Table I summarizes the key parameters and variables involved in the remainder of this article.

A. Block-Circulant Matrix Compression

BCM compression splits the matrix ($W \in \mathbb{R}^{M\times N}$) into $p \times q$ square sub-blocks, where $p = \lceil (M/BS) \rceil$ and $q = \lceil (N/BS) \rceil$. A circulant sub-block ($W_{ij}, i \in [0, p), j \in [0, q)$), shown in Fig. 2, has each column vector as a cyclic shift of the previous one by one element. Thus, only one column is required, reducing the memory storage complexity from $O(n^2)$ to $O(n)$. Moreover, the product of W_{ij} and x_j equals the circular convolution of a_{ij} and x_j, which can be accelerated by the fast Fourier transform (FFT), as follows:

$$W_{ij} \times x_j = \mathcal{F}^{-1}\left(\mathcal{F}(a_{ij}) \odot \mathcal{F}(x_j)\right) \quad (1)$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ represent FFT and IFFT transformations, respectively, and $\odot$ denotes Hadamard product. Finally, BCM compression reduces computational complexity from $O(n^2)$to $O(n \log n)$ in a regular manner, making it suitable for hardware acceleration.

For convolution layer, it can be regarded as a series of matrix-vector multiplication operations, that is

$$O[:][r][c] = W\big[:, :, k_i, k_j\big] \times I[:][r + k_i]\big[c + k_j\big]. \tag{2}$$

where $W \in \mathbb{R}^{M\times N\times K\times K}$ is the weight tensor, I and O are the input and output tensors, respectively, and $r/c/k_i/k_j$ is the index of the cycle on their respective dimensions. We constrain each submatrix $(W_{i,j,k_i,k_j}, i \in [0, \lceil (M/BS)\rceil), j \in [0, \lceil (N/BS)\rceil)$) to follow a circulant pattern. It can be seen that setting BS to the power of 2 enhances the computational gains due to the FFT/IFFT operations. Balancing the computation benefits with model accuracy, previous studies [8], [9] have often empirically set the BS to 4/8/16 and applied it to the entire model.

B. Neural Architecture Search

NAS has garnered considerable attention for its automated exploration of neural network architectures. The initial NAS method trains each subnet from the search space to assess its accuracy level and update the controller, incurring a prohibitive cost [16]. As a remedy, ENAS [17] constructs an over-parametrized supernet, enabling the evaluation of all architectures using its parameter subset. This strategy, commonly called *weight sharing*, has been widely adopted. Later, DARTS [18] highlights the inefficiency of previous methods based on reinforcement learning (RL) and evolutionary algorithms (EAs) in searching within discrete spaces, which leads to substantial architecture evaluations required. To this end, DARTS introduces the differentiable NAS (DNAS) concept, relaxing the discrete search strategy through a *Softmax* way

$$\mathcal{A}^l = \sum_{i=1}^{|\mathcal{O}|} \frac{\exp\big(\alpha_i^l\big)}{\sum_{i'=1}^{|\mathcal{O}|} \exp\big(\alpha_{i'}^l\big)} \times \mathcal{O}_i\Big(\mathcal{A}^{l-1}\Big). \tag{3}$$

Here, $\mathcal{A}^{l-1}$ is the output of the previous layer, and $\mathcal{O}$ denotes the predefined operator space. α_i^l means the architectural parameter of operator $\mathcal{O}_i$ in the lth searchable layer. Similarly, FBNet [19] employs a Gumbel-Softmax (GS) approximation, probabilistically sampling a path from the candidate paths

$$\mathcal{A}^l = \sum_{i=1}^{|\mathcal{O}|} GS\Big(\alpha_i^l \,|\, \alpha^l\Big) \times \mathcal{O}_i\Big(\mathcal{A}^{l-1}\Big). \tag{4}$$

Thus, DNAS has evolved into an optimization process for a set of continuous variables $\alpha = \{\alpha_i^l\}$. On this basis, researchers further incorporate hardware constraints into the objective function and optimize the search process for a target hardware platform [20], [21], [22]. This type of work is commonly referred to as hardware-aware NAS.

C. Algorithm and Hardware Co-Optimization

Researchers initially focused on computation and memory access optimizations for convolutional neural networks (CNNs) and designed a series of dedicated accelerators [23], [24], [25], [26], [27]. They then introduced algorithmic optimizations and adapted them on the hardware side to achieve a synergy between algorithm and hardware [28], [29], [30], such as accelerators for quantized or compressed neural networks. However, recent research has shown that staged algorithmic/hardware optimization makes it hard to achieve an optimal solution. Therefore, algorithm and hardware co-search research has emerged. Jiang et al. [10] performed a joint search for the network architecture and the accelerator's parallel parameters on FPGAs using the RL method, achieving significant overall performance improvement. However, the scalability was limited by the cost of the search. To address this, Li et al. [11] introduced EDD, which conducted co-search for networks and accelerators in a differentiable manner, thereby enhancing search efficiency. Nevertheless, EDD used hardware-agnostic metrics to model the hardware. To this end, Fan et al. [12] established a latency predictor for their single-core hardware architecture to achieve rapid feedback for the hardware evaluation. Furthermore, Lou et al. [31] designed a rapid evaluation function for model deployment on multicore accelerators to enable a broader co-search space.

As NAS technology has evolved, researchers are beginning to explore a wider range of hardware and algorithm co-optimization opportunities [32]. For example, Fasfous et al. [33] jointly explored the layer-wise quantization bit-width and accelerator design. Liang et al. [34] jointly searched the compressed model (irregular) and accelerator based on hardware analytical modeling. Thus, in this article, drawing inspiration from the above advancements, we endeavor to surmount the limitations in BCM compression with the help of DNAS technology and further explore the potential of hardware-software co-search.

III. Framework

Given the target model, FPGA specifications, and frames per second (FPS) settings, FlexBCM automatically generates tailored BCM-compressed CNNs and accelerators to balance model accuracy and hardware efficiency. The overall framework (refer to Fig. 3) consists of two key components: 1) the differentiable compressor and 2) the fast hardware evaluator. These components work together to enable the joint exploration of compression parameters (BS-1/4/8/16) and accelerator structures (tiling and parallelism factors).

Section IV details the differentiable compressor, where we construct a supernet containing all candidate compression operators based on weight sharing. This supernet includes the vanilla convolution operator (BS-1) to copy with different scenario requirements. We then introduce a novel differentiable sampling algorithm to efficiently explore the compression space ($\mathcal{O}$). Besides, we adopt a hardware-friendly frequency-domain quantization scheme for the BCM compression.

Following this, Section V describes the hardware architecture designed to support these compressed models. We construct a generic convolutional computing core (ConvPU) and a specialized BCMPU. The BCMPU is designed to flexibly support various compression parameters and incorporates

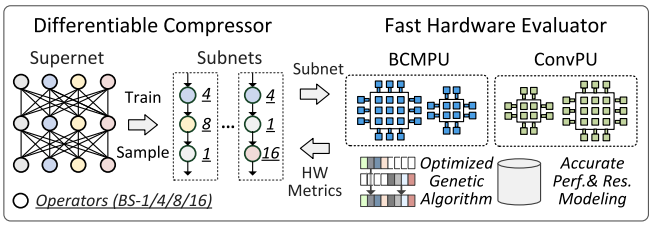

Fig. 3. Overview of our FlexBCM co-search framework.

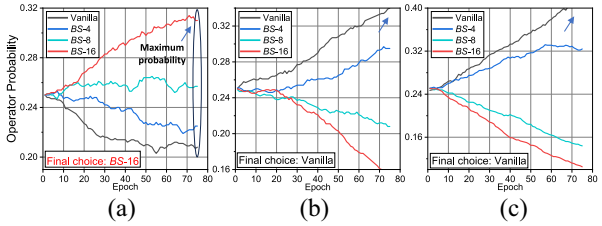

Fig. 4. Probability distributions of each operator in the sixth layer of the RN18 without imposing any computation-cost constraints under different search methods: (a) *GS* method; (b) *Softmax* method; and (c) our proposed *M-GS* method.

optimizations, such as complex conjugate symmetry and multiplication DSP packing, which help to reduce computation and storage overhead.

On this basis, we perform accurate resource and performance modeling of the accelerator to reflect the actual execution of the convolutional layers. We develop a fast hardware generation algorithm based on the genetic algorithm, as detailed in Section VI. This section explains how we leverage genetic algorithms to optimize hardware generation, ensuring efficient mapping of the CNN operations onto the FPGA in a brief time.

In summary, our framework not only provides a method for compressing CNNs but also includes the tools necessary for implementing these models on FPGA hardware, optimizing for both performance and efficiency. Following the DNAS work, we use bi-level optimization (5) to solve this joint search problem. In addition, we introduce a hardware loss term (L_{hw}) in the final loss function to guide the compression operator search process at the algorithmic level

$$\begin{aligned} \min_{\alpha} \quad & L_{\text{val}}\left(w^*, \alpha\right) + \lambda L_{hw}\left(\alpha, F^*\right) \\ \text{s.t.} \quad & w^* = \arg\min_{w} L_{\text{train}}(w, \alpha) \\ \text{s.t.} \quad & F^* = \arg\min_{F} L_{hw}(\alpha, F) \\ \text{s.t.} \quad & hw_{\text{cost}}\left(\alpha, F^*\right) < hw_{\text{limit}} \end{aligned} \tag{5}$$

where w is the supernet weight; α denotes the operator selection probability, also called architectural parameters; and L_{val} and L_{train} are the validation and training loss of the supernet, respectively. F^* denotes the optimized FPGA accelerator, searched under the target hardware resource constraints (hw_{limit}), to quickly provide hardware evaluation. λ is a hyperparameter that controls the tradeoff between terms.

IV. Algorithm Design

In this section, we provide a detailed explanation of the methods and steps taken to achieve efficient compression and quantization, including a moderate sampling algorithm and a hardware-friendly frequency-domain quantization algorithm.

A. Moderate Differentiable Sampling

To avoid repeatedly training subnets with different *BS* values, we first construct a supernet that contains all candidate compression operators in each layer, following the approach of the DNAS works [11].

However, directly applying the *GS* method in the BCM compression scenario results in a biased search, as shown in Fig. 4(a). For clarity, we use layer six as an example, but this trend holds for most layers in the model. We observe that the *GS* method favors the *BS*-16 operator with the fewest parameters over the vanilla operator with higher accuracy in searching. The searched subnet's final accuracy is 94.69% (Table II), an apparent drop from the 95.23% accuracy of the original RN18 model. We attribute the biased search to the fact that the *BS*-16 operator has a higher priority than other operators due to its fewer parameters. However, the *GS* method picks only the highest-probability operator at each feedforward. In contrast, the *Softmax* method enables all operators to participate in computations, prioritizing the vanilla operator during the search. This method achieves the expected outcome (95.40% accuracy) but involves all operators, significantly increasing GPU computation and memory costs (refer to Table II).

TABLE II
Comparison of Search Methods (Tested on a Single NVIDIA RTX 3090 GPU With 24 GB of Video Memory)

Method	Memory [GPU-GB]	Search Time [GPU-hours]	Constraints [Max.FLOPs]	Top-1 Accuracy [%]
Search for RN18 on CIFAR10:				
GS	6.7	1.59	None / 30%	94.69 / 93.86
Softmax	11.7	5.25	None / 30%	95.40 / 94.17
M-GS	8.5	2.64	None / 30%	95.56 / 94.26
Search for RN34 on CIFAR100:				
GS	18.1	12.50	None / 50%	79.02 / 78.15
Softmax	7.9	4.31	None / 50%	76.14 / 76.19
M-GS	12.3	8.03	None / 50%	78.93 / 77.86

To address the above issues, we propose a moderate GS sampling (M-GS) technique, selecting more activation paths based on *GS* sampling as follows:

$$\mathcal{P}_i^l = GS\left(\alpha_i^l\right) = \frac{\exp\left(\left(\log S(\alpha_i^l) + \mathcal{G}_i^l\right)/\tau\right)}{\sum_{i'=1}^{|\mathcal{O}|} \exp\left(\left(\log S(\alpha_{i'}^l) + \mathcal{G}_{i'}^l\right)/\tau\right)} \tag{6}$$

$$\mathcal{A}^{l+1} = \sum_{i=1}^{|\mathcal{O}|} \left(\left[\mathcal{P}_i^l \in \text{largest}(\mathcal{P}^l, 2) \right] \times \mathcal{P}_i^l \times \mathcal{O}_i\left(\mathcal{A}^l\right) \right) \tag{7}$$

where P_i^l denotes the selection probability of the ith operator in the lth layer after *GS* approximation (details can refer to [35]); $\mathcal{A}^{l+1}$ denotes the lth layer output. [exp]=1 when exp is true. The *M-GS* method selects the top two paths with the highest probabilities from the forward path, increasing the diversity of operators and providing a suitable balance between performance and search cost. To further validate the

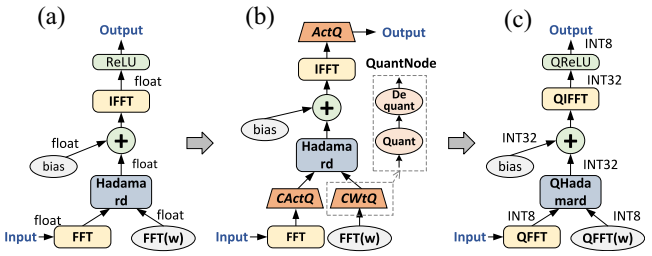

Fig. 5. Quantization flow: (a) regular training in BCM compression; (b) quantization-aware training; and (c) integer inference.

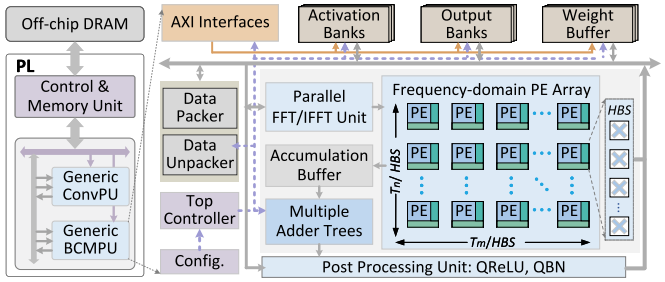

Fig. 6. Overall architecture (left) and the BCMPU structure (right).

effectiveness of the *M-GS* method, we impose a computational constraint during the search process. We achieve this by adding a regularization term to the objective function, which is the sum of the computational costs of each layer weighted by α^l. Results show that reducing the computation by 70% lowers the model's accuracy compared to the unconstrained case ("None"). However, the *M-GS* method still achieves a better-cost-performance tradeoff than the *GS* and *Softmax* methods. Note that the FLOPs metric is hardware-agnostic and cannot reflect the actual latency of the compressed model on the target hardware. We use this metric here to simplify the verification process of the search algorithm.

B. Hardware-Friendly Frequency Domain Quantization

Quantization has become a *de facto* step for implementing CNNs on FPGAs, offering practical advantages for hardware-related benefits [36]. However, the complexity of quantization increases under BCM compression, as evidenced by the following: 1) FFT/IFFT operations introduce additional quantization errors and 2) complex quantization requires determining how to quantize both the real and imaginary parts.

For issue 1), we propose a quantization flow (see Fig. 5) that simulates the quantization errors caused by FFT/IFFT operations. This flow also converts the data involved in off-chip access and computation to integers, which enables actual hardware benefits. Specifically, we employ the $\texttt{Quantize}-\texttt{Dequantize}$ process, also known as fake quantization node, to introduce quantization errors during the model quantization process, as follows:

$$\mathcal{A}_Q^l = \texttt{Quant}_k\left(\mathcal{A}^l\right) = \texttt{clamp}\left(\left\lfloor \frac{\mathcal{A}^l}{S_a^l} \right\rceil, \texttt{min}, \texttt{max}\right) \quad (8)$$

$$\mathcal{W}_Q^l = \texttt{Quant}_k\left(\mathcal{W}^l\right) = \texttt{clamp}\left(\left\lfloor \frac{\mathcal{W}^l}{S_w^l} \right\rceil, \texttt{min}, \texttt{max}\right) \quad (9)$$

where $\mathcal{A}^l$ and $\mathcal{W}^l$ denote the activation and weights of the lth layer, respectively, and S_a^l and S_w^l are the corresponding quantization scales. $\texttt{clamp}(\cdot)$ is a truncation function and k denotes the integer bit-width; A typical approach for determining the scale factor based on the absolute maximum value is illustrated as follows:

$$S_a^l = \max\left(|\mathcal{A}_a^l|\right)/\left(2^{k-1}-1\right). \quad (10)$$

For issue 2), we adopt a unified quantization method [see (11)]. It applies the same scaling factor to the real and imaginary parts, which simplifies the hardware implementation of fast complex multiplication optimization (refer to Section V-C)

$$\texttt{Quant}_k(z_r + j * z_i) = \texttt{Quant}_k(z_r, z_i). \quad (11)$$

To ensure the model accuracy, we set k=8. Note that although we use the `INT8` type here, we exclude the -128 value (set min to -127) for DSP packing optimization.

In model quantization training, since integer-based data cannot be directly inserted back for training or optimization, integer values are rescaled back to the floating-point domain, referred to as "$\texttt{Dequantize}$"

$$\mathcal{A}_{DQ}^l = \texttt{Dequant}\left(\mathcal{A}_Q^l\right) = \mathcal{A}_Q^l \times S_a^l \in \texttt{float} \quad (12)$$

$$\mathcal{W}_{DQ}^l = \texttt{Dequant}\left(\mathcal{W}_Q^l\right) = \mathcal{W}_Q^l \times S_w^l \in \texttt{float}. \quad (13)$$

In integer inference, weights and activations are deployed at low-precision values ($\mathcal{A}_Q^l$ and $\mathcal{W}_Q^l$), with the next layer's quantized activations generated by multiplying by a rescaling factor ($S_a^l \times S_w^l / S_a^{l+1}$). To prevent accuracy degradation due to hardware implementation, we employ a conservative approach by uniformly using 32-bit fixed-point numbers to approximate floating-point factors, such as rescale factors and twiddle factors (ω) in FFT/IFFT. We perform integer inference of the quantized BCM-based RN18 on CIFAR-10, and the results show that the hardware implementation introduces an accuracy degradation within 0.1%, which is consistent with previous work and is negligible [37], [38].

V. Hardware Design

In this section, we describe the architecture of the accelerator that can adapt to different compression parameters (*BS*) and provide insights into the architectural choices and their impact on performance and resource overhead.

A. Overall Architecture

We design a generic and parameter-configurable convolution computation core (ConvPU) and a BCM compression computation core (BCMPU) based on the operator space (Vanilla, *BS*-4/8/16), shown in Fig. 6 (left). The host CPU configures the computation core with the layer information, compression parameter, and memory address. The two cores fetch the required data through the AXI4 interface and store them in multiple BRAM banks or LUTs using the memory interleaving technique to provide the on-chip bandwidth for the computation array. Both computational cores act as independent AXI4 masters, receiving and outputting feature maps in

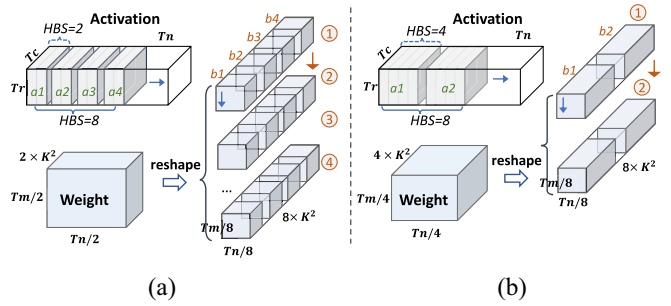

Fig. 7. To reuse the PE array (HBS=8), the dataflow of activations and weights when *BS* equals 4 or 8. (a) BS=4. (b) BS=8.

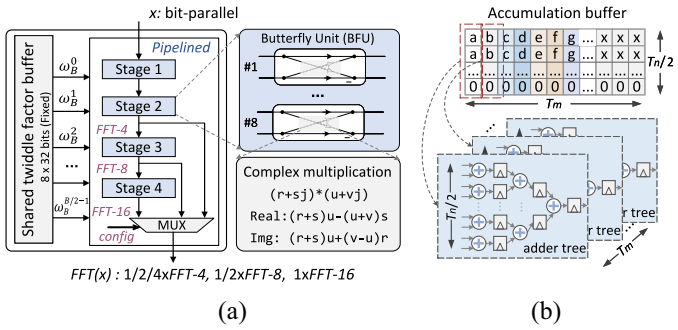

Fig. 8. FFT/IFFT structure and accumulation unit. (a) Parallel FFT units that support variable-length input. (b) Partial result accumulation operation flow.

channel-first format, and thus can directly perform task-level parallel operations with the support of off-chip DRAM.

For the ConvPU design, we employ a tiling architecture, with tiling performed along the horizontal dimension (T_r, T_c) and parallel unfolding conducted over the input/output channels (T_n, T_m). Tiled inputs and weights are sequentially loaded onto the chip for computation, and the final results are outputted off-chip. ConvPU can accommodate various convolutional layer parameters, with differences in the number of times tiled data is loaded. For brevity, we refrain from presenting the internal structure of ConvPU. Subsequently, our focus will be on elucidating the design of the general BCMPU.

B. BCMPU Design

The BCMPU structure, shown in Fig. 6, mainly consists of data processing, on-chip storage, and computation units. The data processing unit unpacks or packs the data, and the on-chip storage unit provides the necessary on-chip buffer and bandwidth for computation. Besides, the post-processing unit performs ReLU and BN operations on the quantized data. The computation unit is the heart of the design to support convolutional layers with flexible block sizes. Next, we describe the details of the computation unit.

1) Frequency-Domain PE Array: The BCM-based computation flow involves FFT-Hadamard-IFFT operations on data of length *BS*. Thus, we further partition and unroll the tiled data by *BS* size in the channel dimensions. However, if we adopt the same unrolling strategy as the ConvPU, the parallelism of the PE array is only $(T_m/BS) \times (T_n/BS)$, which is a significant decrease compared to ConvPU. To complement the parallelism, we simultaneously unroll the *BS* dimension rather than the convolution kernel's spatial dimensions for generality, i.e., $(T_m/BS) \times (T_n/BS) \times BS$. Specifically, we chose the size required for maximum compression (*BS*=16) for the *BS* dimension unrolling to reuse the PE array. We finally set the hardware block size (HBS) to 8, considering the conjugate symmetry. We package the data at positions "0" and "*BS*/2" for the feature maps and weight after FFT because they only contain real parts. Finally, the PE array can be reused for the smaller *BS* values in a time-multiplexing manner.

Fig. 7 illustrates how BCM-based calculations with different block sizes reuse the same frequency domain PE array. Each PE requires HBS activation and weight data per cycle to perform the parallel computation. For activation, we obtain HBS continuous data along the channel dimension. Regarding weights, we employ the reshape operation to rearrange the elements at the corresponding positions. For instance, when *BS* equals 4, the complex weight size is $T_m/2 \times T_n/2 \times 2K^2$ after conjugate symmetry optimization. Then, the weight is reshaped to four $T_m/8 \times T_n/8 \times 8K^2$. Thus, the PE array is reused four times. Finally, the size of the on-chip weight buffer is set to $T_m/2 \times T_n$ to accommodate maximum demand. Notable, if the PE completes the multiply-accumulate operation, it needs to accumulate different data lengths depending on the *BS* value. Accordingly, we only perform complex multiplication inside the PE to simplify its implementation. Then, we set up a unified accumulation unit to facilitate partial sum accumulation along the channel dimension.

2) Parallel FFT/IFFT Unit: For BCM-based convolutional layers, their weights are processed offline by FFT and stored off-chip as complex numbers. Conversely, activations are stored off-chip as real numbers and require online FFT/IFFT operations. Fig. 8(a) illustrates the FFT unit that supports up to 16 parallel term inputs and can produce 4 FFT-4, 2 FFT-8, or 1 FFT-16 transform output, depending on the configuration. It has four stages, each with eight butterfly units (BFUs). The twiddle factors for the operations are stored as INT32 numbers in the on-chip ROM. Since both complex activations and weights possess the conjugate property, the property $(Z_1^* \times Z_2^* = (Z_1 \times Z_2)^*)$ can reduce the complex storage and computational overhead by nearly half. The IFFT structure is identical to the FFT, except for the twiddle factor value. Hence, we do not repeat it. To minimize the FFT/IFFT latency, we adopt this parallel architecture, which is justified, considering that the computation in each stage is relatively small. Moreover, we employ fast complex multiplication optimization, which replaces the real multiplication operation with extra real addition operations to reduce the computational overhead.

3) Accumulation Unit: To generate the output of $T_m \times T_r \times T_c$, we still need to accumulate the partial sums in the channel dimension, done by the accumulation unit. It contains the accumulation buffer and multiple adder trees, shown in Fig. 8(b). The buffer size is $T_m \times T_n/2$, with a single adder tree for each input data column to accumulate the partial sum. $T_n/2$ is the maximum number of accumulation lengths after applying the conjugate symmetric optimization $(T_n/(4/2))$. For the *BS*-8/16, we additionally set up control logic to explicitly set unused data in the buffer to zero to

maintain correctness. The depth of the adder tree is $\log_2(T_n/2)$, where we use registers to synchronize between adjacent stages to optimize the timing. After fully pipelined, the accumulation unit can output the accumulation sum in every cycle.

C. DSP Optimization

The DSP packing of INT8 data multiplication to reduce the computational cost has become a popular optimization method [27]. In ConvPU, two real-number multiplications are packed as $a \times (w_1 << 18 + w_2)$ onto a DSP with 18×27 multiplier support. In contrast, in BCMPU, DSP packing involves optimized complex-number multiplication, which has yet to be explored. The multiplication of activation $(ca = ca_r + j * ca_i)$ and weights $(cw_1 = cw_{1r} + j * cw_{1i}$, $(cw_2 = cw_{2r} + j * cw_{2i}))$ is calculated as follows:

$$\begin{aligned} ca \times cw_1 &= (A_1 - C_1) + j * (A_1 + B_1) \\ ca \times cw_2 &= (A_2 - C_2) + j * (A_2 + B_2) \\ A_1 &= (ca_r + ca_i) \times cw_{1r} \\ A_2 &= (ca_r + ca_i) \times cw_{2r}. \end{aligned} \tag{14}$$

Taking A_1 and A_2 as an example, after packing, it becomes $(ca_r + ca_i) \times (cw_{1r} << 18 + cw_{2r})$. Similarly, B_1 and B_2 is packed as $((w_{i1} - w_{r1}) << 18 + (w_{i2} - w_{r2})) \times ca_r$. We do not repeat the $C_{1/2}$ details here for brevity. It can be observed that an extra addition or subtraction operation is necessary for the DSP packing of complex-number activations/weights. Significant dequantization overhead occurs if the scaling factors of the real and imaginary parts differ. Therefore, we have chosen a unified quantization approach (see Section IV-B). In this way, we only need to apply the rescaling factor for the output. In addition, to keep the packed data within the 27-bit input range, we dropped the −128 in the INT8 data to ensure it would not overflow downward.

VI. Network-Accelerator Mapping

This section presents a model of the accelerator's resource and latency and proposes a fast hardware generation function based on genetic algorithms to provide feedback for subnets.

A. Accelerator Modeling

Here, we focus on modeling the latency and consumption of DSP and BRAM of the accelerator and use the identifiers "cu" and "bu" to distinguish ConvPU and BCMPU cores.

Latency Modeling: We consider the latency of the pipeline's start and exit phases in all units. For off-chip access latency, we adopt more refined modeling for all AXI4 ports by considering the number of requests versus the cost of burst access

$$\begin{cases} \mathcal{L}_{\text{axi}} = \underbrace{(T \cdot \lceil L/(bw \cdot bl) \rceil)}_{\text{number of requests}} / O_{\text{num}} \cdot \underbrace{(bl \cdot O_{\text{num}} + \mathcal{L}_{\text{stall}})}_{\text{burst read cycles}} \\ \mathcal{L}_{\text{stall}} = \max(\, \mathcal{L}_{ddr} - bl \times (O_{\text{num}} - 1),\ 0\,) \end{cases} \tag{15}$$

where T, L, bw, bl, and O_{num} represent the number of accesses, the access length, the bandwidth, the burst length, and concurrent transmission transactions, respectively. $\mathcal{L}_{ddr}$ indicates one DRAM access latency. Regarding the computation latency, for brevity, we illustrate it with an example of the latency of the BCMPU to generate one output tile ($\mathcal{L}_{bu}^{\text{tile}}$)

$$\begin{cases} \mathcal{L}_{bu}^{\text{tile}} = \max\left(\mathcal{L}_{bu}^{W}, \mathcal{L}_{bu}^{I}\right) + \left(\lceil N/(2 \cdot T_n^{bu}) \rceil - 1\right) \cdot \mathcal{L}^{m} + \mathcal{L}_{bu}^{\text{arr}} \\ \mathcal{L}_{bu}^{\text{arr}} = \mathcal{L}_{\text{setup}} + K \cdot K \cdot T_r^{bu} \cdot T_c^{bu} \cdot (2 \cdot \texttt{HBS})/BS \end{cases} \tag{16}$$

where $\mathcal{L}_{bu}^{W}$, $\mathcal{L}_{bu}^{I}$, $\mathcal{L}_{bu}^{\text{arr}}$, and $\mathcal{L}^{m}$ denote the weight latency, input latency, computing latency, and maximum latency of the pipeline, respectively. N means the input channel size.

Resource Modeling: The PE arrays account for most of the DSP consumption ($\mathcal{D}$), which we estimate by multiplying the computation cost per unit by the degree of parallelism

$$\mathcal{D}_{cu} = T_n^{cu} \cdot \left\lceil \frac{T_m^{cu}}{2} \right\rceil;\ \mathcal{D}_{bu} = 3 \cdot \left\lceil \frac{T_m^{bu}}{2 \cdot \texttt{HBS}} \right\rceil \cdot \left\lceil \frac{T_n^{bu}}{\texttt{HBS}} \right\rceil \cdot \texttt{HBS}. \tag{17}$$

Here, the DSP cost after complex multiplication optimization in BCMPU is 3. For brevity, we have not reflected the additional consumption introduced by the FFT/IFFT, quantization, and BN operation in (17). The consumption of BRAM ($\mathcal{R}$) typically hinges on the requisite port width and depth. Within both computing cores, BRAM is primarily utilized to construct input and output buffers, which can be estimated as follows:

$$\begin{aligned} \mathcal{R}_{cu} &= 2 \cdot \left(\left\lceil \frac{T_n^{cu} \cdot 8b}{36b} \right\rceil \cdot \left\lceil \frac{\text{len}_{\text{in}}^{cu}}{512} \right\rceil + T_m^{cu} \cdot \left\lceil \frac{T_r^{cu} \cdot T_c^{cu}}{512} \right\rceil \right) \\ \mathcal{R}_{bu} &= 4 \cdot \left(T_n^{bu} \cdot \left\lceil \frac{\text{len}_{\text{in}}^{bu}}{2048} \right\rceil + T_m^{bu} \cdot \left\lceil \frac{T_r^{bu} \cdot T_c^{bu}}{512} \right\rceil \right) \end{aligned} \tag{18}$$

where len_{in} denotes the maximum depth required for input buffers. T_r and T_c denote the tiling size in the feature map's row and column dimensions. The factor of 4 accounts for double buffering of complex data. A single BRAM18K block can be configured as 9×2048 or 36×512.

B. Fast Hardware Generation

In the joint search of parameter compression and hardware, the framework necessitates not only an accuracy assessment but also a hardware assessment for the sampled subnets. This process involves generating hardware parameters for the subnet on the target FPGA and providing feedback on its hardware performance, as shown in the following equation:

$$\begin{aligned} \min_{F:\{v_{cu}, v_{bu}\}} \quad & \max(\ \mathcal{L}_{cu}(\theta_{cu}, v_{cu}),\ \mathcal{L}_{bu}(\theta_{bu}, v_{bu})\ ) \\ \text{s.t.} \quad & \mathcal{D}_{\text{used}}, \mathcal{R}_{\text{used}}, BW_{\text{used}} < hw_{\text{limit}} \end{aligned} \tag{19}$$

where θ_{cu} and θ_{bu} denote the layer set assigned to the ConvPU and BCMPU, respectively. v_{cu} and v_{bu} denote the hardware (HW) parameter variables of ConvPU and BCMPU, respectively. The two HW variables have the same parameter term, both requiring the determination of values for $[T_r, T_c, T_m, T_n, bw_{\text{in}}, bw_{wt}, bw_{\text{out}}]$. hw_{limit} represents FPGA resource limitation. Due to the vast sampled subnets in joint search, the traditional work's minute-level evaluation time becomes impractical [39]. For example, one search epoch on the ImageNet-1k training set often involves thousands of subnet samples and evaluations.

To this end, we design a fast hardware generation algorithm, as shown in Algorithm 1, to reduce the hardware evaluation time. Specifically, we apply genetic algorithms to accelerate

Algorithm 1: Fast Hardware Generation Algorithm

Initialize the target latency (lat_{target}) and hw_{limit}
Initialize population size ($\mathcal{M}$), iteration number ($\mathcal{I}$)
Function $FitScore(\cdot)$ call latency and resource model to evaluate individuals
◁ ***coarse-grained search***
Randomly initialize the individuals in $\mathcal{P}opu[\mathcal{M}]$ with the index values in Table III
while *iteration* $< \mathcal{I}$ **do**
  ◁ ***multiprocess parallel***
  for *ind in* $[1, \mathcal{M}]$ **do**
    // Get the fitness index with accelerator model
    Evaluate: $Fit_{ind} = \textbf{\textit{FitScore}}(\mathcal{P}opu[ind])$
  Update the top-k global optimal individuals
  for *ind in* $[k, \mathcal{M}]$ **do**
    Select parents (s, t) according to probability
    Crossover: $Temp = \textbf{\textit{cross}}(\mathcal{P}opu[s], \mathcal{P}opu[t])$
    Mutation: $\mathcal{P}opu[ind] = \textbf{\textit{mutat}}(Temp)$
  Keep the best $\mathcal{L}_{best} = Obj[1]$ and $F^* = \mathcal{P}opu[1]$
if $\mathcal{L}_{best} \leq lat_{target}$ **then**
  break
Return best latency ($\mathcal{L}_{best}$) and hardware parameters (F^*)

TABLE III
HARDWARE ARCHITECTURE SEARCH SPACE

	Candidate values
Tiling size (T_r, T_c)	7, 14, 28, 56
Unrolling size (T_n^{cu}, T_m^{cu})	2, 4, 8, 12, 16, 24, 32, 48, 64
Unrolling size (T_n^{bu}, T_m^{bu})	8, 16, 24, 32, 40, 48, 56, 64
Bandwidth (bw)	1, 2, 4, 8, 16, 32

the exploration of the hardware search space and further conduct a *coarse-grained search* with *multiprocess parallel* optimization. In the coarse-grained search, we restrict the search for decision variable (F) to a candidate list, as shown in Table III, to significantly speed up the search process. The values in the list are determined based on the convolutional layer parameters and hardware execution efficiency (e.g., integer divisibility). On the other hand, considering the independence of individual evaluations within the population, we employ multiprocess parallelism to leverage modern CPUs' computing power fully. In this study, we set the values of $\mathcal{M}$ and $\mathcal{I}$ to 250 and 50, respectively. The search process exhibits a stable convergence behavior, reaching a definitive value after about 40 iterations. Finally, on a desktop-class CPU i7-8700K with 32GB of DDR4, the search took just 1.63 s at parallelism 4. It is noteworthy that our aim is not to pursue the optimal solution but rather to rapidly attain an optimized solution, facilitating the hardware assessment of the subnet.

VII. EVALUATION

A. Experimental Setup

To evaluate the effectiveness of FlexBCM, we conduct a joint search to deploy RN18 and RN34 models on the embedded Xilinx ZCU102 platform, aiming for 100 FPS and 80 FPS, respectively. FlexBCM searches on ImageNet-100,

TABLE IV
COMPRESSION PARAMETERS OF RN18 ON ZCU102

	Conv1-2	*Conv3*	*Conv4*	*Conv5*	*Conv6-7*
Param	*BS*-1	*BS*-8	*BS*-1	*BS*-8	*BS*-4
	Conv8	*Conv9-11*	*Conv12-14*	*Conv15*	*Conv16-17*
Param	*BS*-8	*BS*-4	*BS*-1	*BS*-4	*BS*-1

TABLE V
COMPRESSION PARAMETERS OF RN34 ON ZCU102

	Conv1	*Conv2*	*Conv3-4*	*Conv5*	*Conv6-7*	*Conv8-13*
Param	*BS*-1	*BS*-4	*BS*-8	*BS*-1	*BS*-8	*BS*-4
	Conv14	*Conv15*	*Conv16*	*Conv17*	*Conv18*	*Conv19*
Param	*BS*-8	*BS*-1	*BS*-4	*BS*-1	*BS*-4	*BS*-1
	Conv20-22	*Conv23*	*Conv24*	*Conv25*	*Conv26-27*	*Conv28-33*
Param	*BS*-4	*BS*-1	*BS*-4	*BS*-8	*BS*-4	*BS*-1

TABLE VI
HW PARAMETERS FOR BCM-BASED RN18 RUNNING ON ZCU102

F	T_r	T_c	T_m	T_n	bw_{in}	bw_{wt}	bw_{out}
ConvPU (v_{cu})	7	7	24	48	16	32	4
BCMPU (v_{bu})	7	7	32	32	8	2	4

TABLE VII
HW PARAMETERS FOR BCM-BASED RN34 RUNNING ON ZCU102

F	T_r	T_c	T_m	T_n	bw_{in}	bw_{wt}	bw_{out}
ConvPU (v_{cu})	7	7	64	32	4	16	2
BCMPU (v_{bu})	14	14	32	32	16	4	32

a random subset of ImageNet-1k training set that consists of 1000 classes to optimize both supernet weights (w) and the compression parameters (α). We train the supernet for 60 epochs, with a batch size of 256, a learning rate of 0.1, and cosine annealing, where we freeze α in the first 20 epochs. It takes 12.2 and 21 h on a single NVIDIA A100 GPU (80 GB video memory), respectively. The searched compression parameters of the RN18 and RN34 are shown in Tables IV and V, respectively. We train the searched subnets for 250 epochs with a batch size of 512 on the complete training set to obtain the final accuracy.

The corresponding FPGA accelerator parameters are shown in Tables VI and VII, respectively. The accelerator design is implemented using HLS-compatible C++ code in Vitis HLS (v2021.2), with a clock cycle of 200 MHz. RTL simulation is performed through the tool's C/RTL co-simulation feature. The exported RTL is synthesized and placed-and-routed in Vivado (v2021.2). The built-in reports in Vivado provide power consumption data for the accelerator. Detailed power and resource consumption data can be found in Table VIII.

B. Quantization Algorithm Validation

We evaluate the proposed quantization method on RN18 and RN34 models using the classical CIFAR10 and ImageNet-100 datasets. For comparison, we present the accuracy of the floating-point model and the baseline scheme in which the real and imaginary parts are quantized separately.

TABLE VIII
COMPREHENSIVE COMPARISON WITH PREVIOUS FPGA IMPLEMENTATIONS

	ICCAD 2019 [40]	*TCASI 2021 [41]*	*FCCM 2021 [5]*	*TCAD 2022 [34]*	*FPGA 2022 [42]*	*TVLSI 2023 [43]*	*DATE 2023 [9]*	*This Work*	
Platform	ZC706	Arria10	ZC706	ZCU102	ZCU102	ZCU102	PYNQ-Z2	ZCU102	
Frequency (MHz)	200	170	150	200	150	200	100	200	
Model	RN18	RN18	RN34	VGG16	RN18	VGG16	RN18	RN18	RN34
Bitwidth	W^8A^8	W^8A^8	$W^{16}A^{16}$	$W^{16}A^{16}$	W^4A^5	$W^{16}A^{16}$	$W^{16}A^{16}$	W^8A^8	W^8A^8
Compression	None	Regular	Regular	Irregular	None	Irregular	BCM[a]	BCM	BCM
Top-1/5 Acc. Drop	-/-	-/1.0%	1.0%/-	no loss	0.9%/0.6%	0.9%/0.4%	3.0%/1.5%	2.0%/0.98%	1.1%/0.5%
DSP Usage	818	512	900	1144	2096	1061	117	957	1392
SRAM Usage[b]	708	465	-	912	878	502	225	470	506
Logic Resource[b]	100.2K	103K	218.6K	-	174.5K	348K	18.2K	156.0K	192.3K
Power (W)	7.31	4.6	-	23.6	13.4	11	1.83	9.91	11.4
Frame Rate (FPS)	30.99	-	31.1	-	72.8	-	12.5	131.1	84.1
Throughput (GOP/s)	124.9	89.09	230.1	309	263.7	409.6	42.75	448.0	589.0
Density (GOP/s/DSP)	0.153	0.174	0.256	0.270	0.126	0.386	0.365	0.468	0.423
Efficiency (GOP/s/W)	17.09	19.41	-	13.09	19.68	37.24	23.36	**45.21**	**51.67**

[a] *BCM compression with further pruning.*
[b] *"SRAM" means BRAM18K in Xilinx FPGA and M20k in Intel FPGA. "Logic" means LUTs in Xilinx FPGA and ALMs in Intel FPGA.*

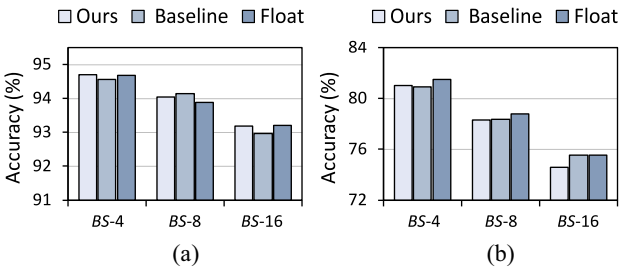

Fig. 9. Accuracy of frequency domain quantization algorithms under different *BS* values. (a) RN18 on CIFAR10. (b) RN34 on ImageNet-100.

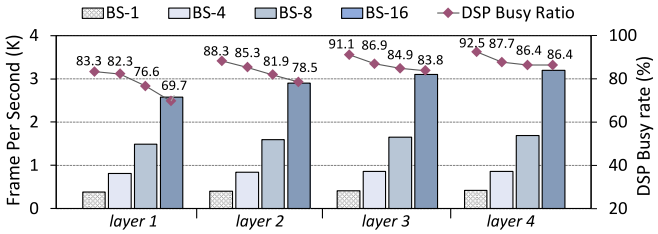

Fig. 10. Throughput and DSP busy rate of ConvPU and BCMPU running the same convolutional layer with the same parallel parameters.

The results are shown in Fig. 9. We observe that both quantization schemes achieve comparable accuracy under the INT8 quantization for both the RN18 model and RN34 model with different block sizes. The maximum accuracy drop for RN18 on CIFAR10 and RN34 on ImageNet-100 in our quantization is 0.2% and 0.9% (adopting *BS*-16), respectively. Kindly note that we here do not focus on the full recovery of accuracy but rather on validating the effectiveness of the quantization scheme. Therefore, we only perform a limited number of 90 training epochs on the ImageNet-100 dataset. At this point, a precision error of 0.9% is acceptable. The quantized model can improve its accuracy further as training epochs increase. Despite the broader numerical representation range exhibited by the baseline method in comparison to the approach we propose, performance between the two is similar in most cases. This phenomenon is mainly attributed to the fact that the RN18 and RN34 models possess sufficient feature extraction capability. Ultimately, considering hardware friendliness, our approach exhibits a marked superiority in overall performance.

C. ConvPU/BCMPU Validation

To validate the effectiveness of the computational cores, we select the configurations of the main convolutional layers in ResNet-18/34 networks and conduct performance and efficiency tests on the computational core based on different compression parameters (*BS*-1/4/8/16). The specific network layer configurations (*layer*1/2/3/4) are $\{(56/28/14/7)^2 \times 64/128/256/512\} \rightarrow \{(56/28/14/7)^2 \times 64/128/256/512\}$, with $K/S/P$ values of 3/1/1 and a W^8A^8 bitwidth. For a fair comparison, we separately deploy ConvPU and BCMPU on Xilinx ZCU102, both executing the above four convolutional layers with the same level of parallelism. ConvPU has a parallelism of 16×16, while BCMPU has a parallelism of $32 \times 16/8 \times 4$ (complex computation). We obtain performance through C/RTL co-simulation in Vitis HLS (v2021.2).

Fig. 10 illustrates the performance and efficiency of both cores when executing the respective layers independently. In terms of *throughput*, it can be observed that BCMPU exhibits a significant throughput advantage (more than 2×) compared to ConvPU. Across *layer*1 to *layer*4, as the compression ratio increases (*BS*-1 to *BS*-16), BCMPU's throughput for each layer gradually improves, demonstrating the adaptability of our designed BCMPU to different *BS* values. Regarding *DSP utilization*, except for *layer*1, ConvPU's DSP busy ratio is consistently above 85%, primarily due to the relatively small channel count (64) in *layer*1. The optimized BCMPU maintains an acceptable level of DSP utilization. However, compared to ConvPU, BCMPU's DSP utilization decreases, mainly because the computation after BCM compression is less and more complex than the original convolution.

Additionally, we observe the following patterns in BCMPU execution: 1) within the same convolutional layer, DSP utilization during BCMPU runtime decreases

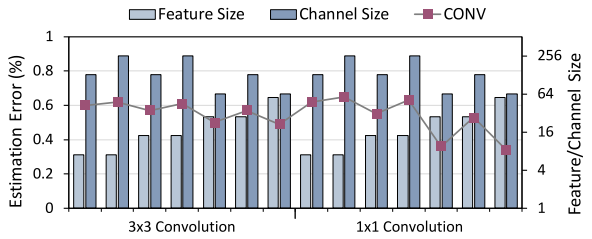

Fig. 11. Latency estimation errors of the ConvPU.

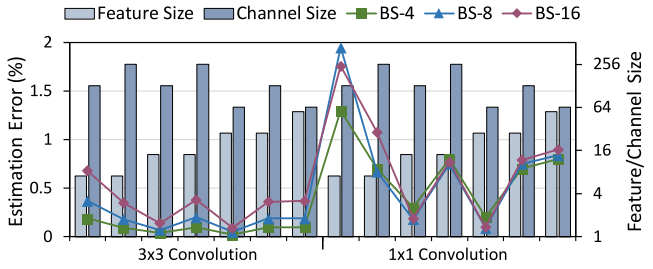

Fig. 12. Latency estimation errors of the BCMPU.

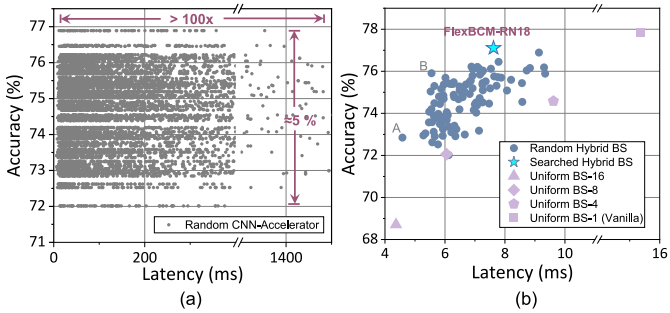

Fig. 13. Accuracy and latency tradeoffs. (a) Random compressed network-accelerator pairs versus (b) random networks with Algorithm 1 enhanced.

as the compression ratio increases and 2) across different convolutional layers, the impact of compression ratio on DSP utilization gradually diminishes with increasing channel depth. The reasons for these phenomena are twofold: 1) larger compression parameters imply a smaller computational load for the compressed convolutional layer, making it more challenging to utilize DSP under the same parallelism and 2) we perform tiling in the channel dimension, and as channel depth increases, the deepening of the pipeline reduces the impact of injection and ejection phases on overall latency. In conclusion, our designed ConvPU and BCMPU can effectively handle different layers while maintaining an acceptable DSP utilization. Furthermore, low-bit-width multiplication packing contributes to further enhancing DSP efficiency.

D. Accelerator Modeling Validation

To avoid time-consuming hardware deployment, we have performed precise modeling of the accelerator's performance and resources to reflect the actual deployment of the model. Therefore, this section will verify the prediction accuracy of the accelerator model to evaluate its effectiveness. We compare the result with the C/RTL co-simulation result in Vitis HLS (v2021.2) and the synthesis result in Vivado (v2021.2).

Here, the tested feature map sizes include $[56, 28, 14]$, channel sizes include $[64, 128, 256, 512]$, and the 3×3 and 1×1 convolution kernel sizes. Instead of a full permutation, we chose 14 representative layer configurations as test cases. We perform performance verification on the ConvPU and BCMPU units separately and present the verification results in Figs. 11 and 12. The hardware parameters in both cores are randomly sampled. It can be observed that, for the general convolution core, our performance model exhibits excellent performance in terms of error rate, remaining within 0.7%. In contrast, the prediction model described in the work [39] fluctuated within 10% in error rate, highlighting the significant error reduction we achieved. Compared to ConvPU, BCMPU has a slightly higher-error rate, mainly due to: 1) BCMPU's operation is relatively complex, containing more pipeline delays and 2) the overall latency is small, increasing its memory access delay ratio. Overall, BCMPU's prediction error rate is still acceptable, remaining within 2%. Furthermore, we select five common hardware configurations where the resource prediction results deviate from the actual outcomes by single-digit discrepancies, which will not be further demonstrated.

E. Validation of Co-Search Effectiveness

To validate the effectiveness of our co-exploration approach, we conduct experiments using RN18 on the Xilinx ZCU102 platform. We randomly sample 100 compressed subnets from the compression parameter space ($\approx$ `1E+10`). Each subnet undergoes the same training setting over 36 epochs on the ImageNet-100 dataset to assess the accuracy. Further, for each sampled subnet, we generate 100 distinct hardware configuration sets from a pruned parameter space ($\approx$ `1E+11`, detailed in Table III), adhering to constraints ensuring DSP utilization exceeds 50% and BRAM utilization does not surpass 90% of the ZCU102's capacity.

Fig. 13(a) shows the distribution of accuracy and latency among randomly generated compressed CNN-accelerator pairs. Notably, even with a limited sample size, we observe accuracy fluctuations reaching up to 5% at matching latencies, suggesting that this effect could be even more significant when extended to larger datasets, such as the ImageNet-1k dataset. Fig. 13(b) depicts the accuracy and latency distribution of compressed CNN-accelerator pairs after optimizing hardware parameters via Algorithm 1, effectively demonstrating the criticality of the hardware generation algorithm.

To further elucidate the effectiveness of the co-exploration, we use annotations with different shapes to differentiate model-accelerator pairs under fixed compression strategies (*BS*-4/8/16) and uncompressed (*BS*-1). Compared to fixed BCM compression strategies [8], [9], which are commonly used in previous work, our searched solutions notably achieve superior tradeoffs between accuracy and latency. Furthermore, we observe that the hybrid schemes more effectively balance accuracy and latency, underscoring the need for layer-wise hybrid *BS* strategies. Although random exploration occasionally produces near-optimal outcomes (e.g., cases "A" and "B"), it is generally inefficient, which requires 215.4 GPU hours to evaluate 100 random subnets–approximately 17 times more computationally expensive than our method. For deeper networks, such as ResNet34, the random method becomes impractical. In contrast, our differentiable search method

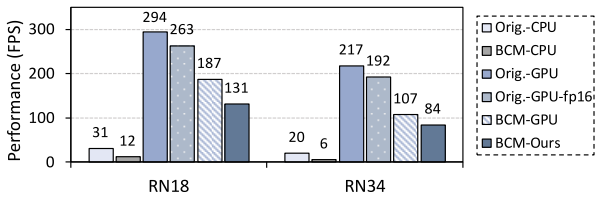

Fig. 14. Performance comparison with CPU and GPU versions.

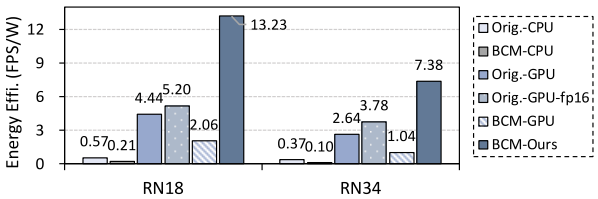

Fig. 15. Energy efficiency comparison with CPU and GPU versions.

scales efficiently with model training overhead and offers tailored tradeoff strategies to meet various latency requirements.

F. Comparison With CPU and GPU

In this section, we undertake a comparative analysis of performance and energy efficiency across CPU, GPU, and the searched FPGA accelerators. We use the PyTorch (v2.0.1) framework with CUDA 11.7 and cuDNN 8.5.0 to run model inference on CPU and GPU platforms, and detailed information is listed as follows.

1) *CPU Baseline:* Intel i7-8700K processor with 12-MB cache, six physical cores, 12 threads operating at 3.7 GHz, and the thermal design power (TDP) is 95 W.
2) *GPU Baseline:* NVIDIA Tesla V100S with 32-GB HBM2 and 5120 hardware threads operating at 1.245 GHz.

We also implement the original model on CPU and GPU, denoted as "Orig.." We exclusively perform FP16 inference on the original model due to the absence of half-precision support in PyTorch's FFT/IFFT operations. CPU power consumption is estimated as the product of CPU utilization and TDP power, and GPU power is measured using *nvidia-smi*. Both the CPU and GPU versions run with a batch size of two.

Fig. 14 illustrates the performance comparison between the accelerator and CPU/GPU during the execution of BCM-compressed RN18 and RN34. Our accelerators demonstrate substantial throughput improvements of 10.92× and 14.0×, respectively, in comparison to the CPU baseline. Our accelerators, when benchmarked against the GPU baseline, deliver performances at 70% and 79% of the GPU's level, respectively, indicating that they are indeed outpaced by GPUs. This discrepancy can be attributed to the embedded nature of the ZCU102 platform, which possesses constrained hardware resources. Despite the reduction in model computation resulting from BCM compression, we observe a noteworthy decline in performance rather than an enhancement in both CPU and GPU performance. This phenomenon is primarily attributable to the PyTorch framework's specific optimizations for standard convolution operations, including computational and memory optimizations on the CPU and *TensorCore* optimizations on the GPU. In contrast, the BCM compression operations, relying on FFT/IFFT, are inherently intricate and lack dedicated optimizations. In terms of energy efficiency (FPS/W), as depicted in Fig. 15, our accelerators exhibit noteworthy enhancements of 63.0× and 73.8× for BCM-RN18 and BCM-RN34, respectively, when juxtaposed with their CPU counterparts. Relative to the GPU versions, our accelerators also realize improvements of 6.42× and 7.96×, respectively. Furthermore, within the GPU version employing FP16, the half-precision inference reduces the video memory and power consumption instead of performance improvements for RN18 and RN34. Considering these factors, it is reasonable that our accelerators tradeoff some performance for substantial gains in energy efficiency when compared to GPUs.

G. Comprehensive Performance Comparison

We compare the solutions generated by FlexBCM with some state-of-the-art FPGA accelerators in terms of throughput, computational efficiency, and model accuracy, as shown in Table VIII.

Regarding *throughput*, our accelerators demonstrate outstanding performance compared to prior research, achieving throughputs of 448 and 589 GOP/s on RN18 and RN34, respectively, representing improvements of 1.10–2.56 times over previous works. Regarding *computational efficiency*, our accelerators achieve 45.21 and 51.67 GOP/s/W on RN18 and RN34, respectively, showcasing improvements of 1.21–3.02 times compared to prior work. These significant enhancements primarily stem from two aspects: 1) the application of BCM regular compression and frequency domain quantization algorithms facilitates hardware gains and 2) the efficient and flexible hardware core design exploits the algorithmic potential. Specifically, on the same platform [34] and [43], based on the INT16 quantization strategy, achieve high throughput on the compressed VGG16. However, it is noteworthy that the irregular pruning adopted in both works cannot guarantee the requirement of common multipliers during INT8 data packing. Therefore, INT8 quantization does not ensure the DSP efficiency improvement reported in [34] and [43]. We compare our work with [9], which similarly employs BCM compression but applies further pruning to enhance flexibility. Nonetheless, this work sets a fixed compression rate for each network layer, requiring manual tuning and leading to a noticeable decline in accuracy (3% in Top-1 accuracy). Therefore, our design excels beyond [9] in both accuracy and computational efficiency, owing to our implementation of more flexible compression strategies and optimized quantization algorithms.

Regarding *model accuracy*, following [41] and [43], we opt for accuracy degradation as a metric to assess the impact of compression on accuracy. This metric mitigates the variations caused by different models and settings. The results show that the accuracy of the model compressed by FlexBCM is comparable to previous research results [5], [41]. By incorporating innovative quantization algorithms, Sun et al. [42] achieved higher accuracy in RN18, yet our design maintains an advantage in resource consumption and energy efficiency. Moreover, our automated search method demonstrates superior scalability. FlexBCM can rapidly explore the algorithm-hardware design space and yield efficient model-accelerator

pairs in various application scenarios without requiring manual iterative experimentation.

VIII. Conclusion

In this article, we propose an automated framework, FlexBCM, for co-exploring hybrid BCM-compressed CNNs and accelerators, which overcomes the limitations of prior BCM compression methods and further explores the hardware-algorithm joint design space. First, we efficiently explore the compression space in a differentiable manner. Then, we design and model the hardware architectures that flexibly support different compression parameters. Finally, we efficiently integrate algorithm exploration and hardware design based on fast hardware generation. Compared with prior works, FlexBCM achieves significant computational efficiency improvement.

References

[1] T. Chen et al., "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 269–284, 2014.

[2] M. Blott et al., "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfig. Technol. Syst.*, vol. 11, no. 3, pp. 1–23, 2018.

[3] C. Wang, L. Gong, X. Li, and X. Zhou, "A ubiquitous machine learning accelerator with automatic parallelization on FPGA," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2346–2359, Oct. 2020.

[4] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.

[5] S. I. Venieris, J. Fernandez-Marques, and N. D. Lane, "unzipFPGA: Enhancing FPGA-based CNN engines with on-the-fly weights generation," in *Proc. FCCM*, 2021, pp. 165–175.

[6] S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, "Hardware acceleration of sparse and irregular tensor computations of ML models: A survey and insights," *Proc. IEEE*, vol. 109, no. 10, pp. 1706–1752, Oct. 2021.

[7] C. Ding et al., "CirCNN: Accelerating and compressing deep neural networks using block-circulantweight matrices," in *Proc. MICRO*, 2017, pp. 395–408.

[8] J. Yue et al., "STICKER-T: An energy-efficient neural network processor using block-circulant algorithm and unified frequency-domain acceleration," *IEEE J. Solid-State Circuits*, vol. 56, no. 6, pp. 1936–1948, Jun. 2021.

[9] H. Song, J. Yoon, D. Kim, E. Kwon, T.-H. Oh, and S. Kang, "FPGA-based accelerator for rank-enhanced and highly-pruned block-circulant neural networks," in *Proc. DATE*, 2023, pp. 1–6.

[10] W. Jiang et al., "Hardware/software co-exploration of neural architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4805–4815, Dec. 2020.

[11] Y. Li et al., "EDD: Efficient differentiable DNN architecture and implementation co-search for embedded ai solutions," in *Proc. DAC*, 2020, pp. 1–6.

[12] H. Fan et al., "Algorithm and hardware co-design for reconfigurable CNN accelerator," in *Proc. ASP-DAC*, 2022, pp. 250–255.

[13] Y. Zhang et al., "DIAN: Differentiable accelerator-network co-search towards maximal DNN efficiency," in *Proc. ISLPED*, 2021, pp. 1–6.

[14] K. Choi et al., "DANCE: Differentiable accelerator/network co-exploration," in *Proc. DAC*, 2021, pp. 337–342.

[15] W. Lou et al., "Unleashing network/accelerator co-exploration potential on FPGAs: A deeper joint search," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Apr. 19, 2024, doi: 10.1109/TCAD.2024.3391688.

[16] B. Zoph and Q. Le, "Neural architecture search with reinforcement learning," in *Proc. ICLR*, 2017, pp. 1–16.

[17] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 4095–4104.

[18] H. Liu et al., "DARTS: Differentiable architecture search," in *Proc. ICLR*, 2019, pp. 1–13.

[19] B. Wu et al., "FBNet: Hardware-aware efficient ConvNet design via differentiable neural architecture search," in *Proc. CVPR*, 2019, pp. 10734–10742.

[20] H. Benmeziane et al., "Hardware-aware neural architecture search: Survey and taxonomy," in *Proc. 13th IJCAI*, 2021, pp. 4322–4329.

[21] X. Luo, D. Liu, H. Kong, S. Huai, H. Chen, and W. Liu, "LightNAS: On lightweight and scalable neural architecture search for embedded platforms," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 6, pp. 1784–1797, Jun. 2023.

[22] H. Bouzidi, M. Odema, H. Ouarnoughi, M. A. Al Faruque, and S. Niar, "HADAS: Hardware-aware dynamic neural architecture search for edge performance scaling," in *Proc. DATE*, 2023, pp. 1–6.

[23] L. Gong, C. Wang, X. Li, H. Chen, and X. Zhou, "MALOC: A fully pipelined FPGA accelerator for convolutional neural networks with all layers mapped on chip," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2601–2612, Nov. 2018.

[24] R. Xu, S. Ma, Y. Wang, Y. Guo, D. Li, and Y. Qiao, "Heterogeneous systolic array architecture for compact CNNs hardware accelerators," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2860–2871, Nov. 2022.

[25] Y. Liang, L. Lu, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 4, pp. 857–870, Apr. 2020.

[26] C. Wang, L. Gong, X. Ma, X. Li, and X. Zhou, "WooKong: A ubiquitous accelerator for recommendation algorithms with custom instruction sets on FPGA," *IEEE Trans. Comput.*, vol. 69, no. 7, pp. 1071–1082, Jul. 2020.

[27] Q. Liu, M. Sun, J. Sun, L. Lu, J. Zhao, and Z. Wang, "SSiMD: Supporting six signed multiplications in a DSP block for low-precision CNN on FPGAs," in *Proc. FPT*, 2023, pp. 161–169.

[28] S. Huang et al., "Mixed precision quantization for ReRAM-based DNN inference accelerators," in *Proc. ASP-DAC*, 2021, pp. 372–377.

[29] W. Lou, L. Gong, C. Wang, Z. Du, and X. Zhou, "OctCNN: A high throughput FPGA accelerator for CNNs using octave convolution algorithm," *IEEE Trans. Comput.*, vol. 71, no. 8, pp. 1847–1859, Aug. 2022.

[30] B. Liu et al., "Frequency-domain inference acceleration for convolutional neural networks using ReRAMs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 12, pp. 3133–3146, Dec. 2023.

[31] W. Lou, J. Qian, L. Gong, X. Wang, C. Wang, and X. Zhou, "NAF: Deeper network/accelerator co-exploration for customizing CNNs on FPGA," in *Proc. DATE*, 2023, pp. 1–6.

[32] E. Luo et al., "DeepBurning-MixQ: An open source mixed-precision neural network accelerator design framework for FPGAs," in *Proc. ICCAD*, 2023, pp. 1–9.

[33] N. Fasfous et al., "AnaCoNGA: Analytical HW-CNN co-design using nested genetic algorithms," in *Proc. DATE*, 2022, pp. 238–243.

[34] Y. Liang et al., "An efficient hardware design for accelerating sparse CNNs with NAS-based models," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 3, pp. 597–613, Mar. 2022.

[35] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with Gumbel-Softmax," 2016, *arXiv:1611.01144*.

[36] A. Kouris, S. I. Venieris, and C.-S. Bouganis, "A throughput-latency co-optimised cascade of convolutional neural network classifiers," in *Proc. DATE*, 2020, pp. 1656–1661.

[37] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. CVPR*, 2018, pp. 2704–2713.

[38] Y. Li et al., "MQBench: Towards reproducible and deployable model quantization benchmark," in *Proc. NeurIPS*, 2021, pp. 1–26.

[39] X. Zhang et al., "DNNExplorer: A framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator," in *Proc. 39th ICCAD*, 2020, pp. 1–9.

[40] Q. Xiao and Y. Liang, "Zac: Towards automatic optimization and deployment of quantized deep neural networks on embedded devices," in *Proc. ICCAD*, 2019, pp. 1–6.

[41] X. Xie, J. Lin, Z. Wang, and J. Wei, "An efficient and flexible accelerator design for sparse convolutional neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 7, pp. 2936–2949, Jul. 2021.

[42] M. Sun et al., "FILM-QNN: Efficient FPGA acceleration of deep neural networks with intra-layer, mixed-precision quantization," in *Proc. FPGA*, 2022, pp. 134–145.

[43] W. Sun, D. Liu, Z. Zou, W. Sun, S. Chen, and Y. Kang, "Sense: Model-hardware codesign for accelerating sparse CNNs on systolic arrays," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 31, no. 4, pp. 470–483, Apr. 2023.