Hardware-Friendly 3-D CNN Acceleration With Balanced Kernel Group Sparsity

Mengshu Sun, Kaidi Xu, *Member, IEEE*, Xue Lin, *Member, IEEE*, Yongli Hu, *Member, IEEE*, and Baocai Yin, *Member, IEEE*

***Abstract*—Being capable of extracting more information than 2-D convolutional neural networks (CNNs), 3-D CNNs have been playing a vital role in video analysis tasks like human action recognition, but their massive operations hinder the real-time execution on edge devices with constrained computation and memory resources. Although various model compression techniques have been applied to accelerate 2-D CNNs, there are rare efforts in investigating hardware-friendly pruning of 3-D CNNs and acceleration on customizable edge platforms like FPGAs. This work starts from proposing a *kernel group row-column (KGRC)* weight sparsity pattern, which is fine-grained to achieve high pruning ratios with negligible accuracy loss, and balanced across kernel groups to achieve high computation parallelism on hardware. The reweighted pruning algorithm for this sparsity is then presented and performed on 3-D CNNs, followed by quantization under different precisions. Along with model compression, FPGA-based accelerators with four modes are designed in support of the kernel group sparsity in multiple dimensions. The co-design framework of the pruning algorithm and the accelerator is tested on two representative 3-D CNNs, namely C3D and R(2+1)D, with the Xilinx ZCU102 FPGA platform for action recognition. The experimental results indicate that the accelerator implementation with the KGRC sparsity and 8-bit quantization achieves a good balance between the speedup and model accuracy, leading to acceleration ratios of 4.12× for C3D and 3.85× for R(2+1)D compared with the 16-bit baseline designs supporting only dense models.**

***Index Terms*—3-D convolutional neural network (CNN), edge device inference, FPGA, model compression, weight pruning.**

I. Introduction

With one more dimension for temporal feature extraction than 2-D convolutional neural networks (CNNs) that can only extract spatial features, 3-D CNNs have been employed extensively in multiple video analysis tasks, such as video classification and human action recognition or detection [2], [5], [36], [43], [44], [48]. Nonetheless, the high computation and memory intensity of 3-D CNNs impedes their deployment on resource-constrained edge devices, especially in scenes with real-time inference and privacy requirements, such as unmanned aerial vehicle and autonomous driving. For example, the AMD-Xilinx ZCU102 FPGA as a customizable edge platform has 32 Mbit on-chip memory space and 2520 DSPs as the primary computation resources, which can hardly accommodate C3D [43] as the most basic 3-D CNN structure with 78.41M parameters and 77.24G operations. This necessitates the combination of (i) model compression techniques at the algorithm level to reduce the CNN size and computation complexity with (ii) the accelerator optimizations at the hardware level to increase the computation parallelism (i.e., the number of computations in parallel) and resource utilization efficiency.

The most widely applied model compression approaches are based on pruning that removes redundant weights and thus reduces the number of operations [8], [17], [18], [20], [21], [25], [45], [47], [52], or quantization that reduces the precisions of weights (and activations) [6], [7], [9], [14], [23], [24], [51], [53], [54]. Incorporated into hardware implementations for acceleration, pruning has been studied in [22], [27], [34] and [13], most of which focus on unstructured (irregular) pruning, and quantization has been studied in [15], [16], [31], [32], [46] and [28], most of which work with binary weights. However, prior compression approaches have respective shortcomings. For instance, coarse-grained structured (row, column, etc.) pruning and low-precision (binary, ternary, and fixed-point with 4-bit or less) quantization suffer from non-negligible accuracy loss, and unstructured pruning incurs extra indexing overhead and degradation in computation parallelism.

Motivated by the efforts in sparsifying 2-D CNNs for image domain tasks, this work investigates efficient acceleration of 3-D CNNs on the resource-constrained edge FPGA platforms with algorithm-hardware co-design, considering two critical targets jointly, i.e., maintaining the original model accuracy, and enhancing the computation and resource utilization efficiency that is essential for hardware acceleration. From the perspective of model compression at the algorithm level, a structured and fine-grained sparsity pattern, called kernel group row-column (KGRC) sparsity, is proposed for weight

Manuscript received 11 November 2023; revised 1 March 2024 and 10 April 2024; accepted 12 April 2024. Date of publication 16 April 2024; date of current version 20 September 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62376014, Grant U21B2038, and Grant U19B2039; in part by the National Key Research and Development Program of China under Grant 2021ZD0111902; and in part by the Research and Development Program of Beijing Municipal Education Commission under Grant KZ202210005008. This article was recommended by Associate Editor M. D. Santambrogio. *(Corresponding author: Baocai Yin.)*

Mengshu Sun, Yongli Hu, and Baocai Yin are with the Beijing Key Laboratory of Multimedia and Intelligent Software Technology, Beijing Institute of Artificial Intelligence, and the Faculty of Information Technology, Beijing University of Technology, Beijing 100124, China (e-mail: sunms@bjut.edu.cn; huyongli@bjut.edu.cn; ybc@bjut.edu.cn).

Kaidi Xu is with the Department of Computer Science, Drexel University, Philadelphia, PA 19104 USA (e-mail: kx46@drexel.edu).

Xue Lin is with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115 USA (e-mail: xue.lin@northeastern.edu).

Digital Object Identifier 10.1109/TCAD.2024.3390040

tensors in 3-D CNNs. Each weight tensor to be pruned is partitioned into equal-sized weight kernel groups, in each of which the pruning will be performed in the row (output channel) dimension, the column (within each weight kernel) dimension, or both. This is an extension of the work [33] with the kernel group structured (KGS) sparsity eliminating redundant weights only along the column dimension. It should be noted that the proposed KGRC sparsity is *balanced*, meaning that the number of pruned rows or columns is the same for all the kernel groups in the weight tensor of a 3-D CNN layer. With the fine granularity assisting the model accuracy, the balance of this sparsity facilitates the potential of high computation parallelism in hardware acceleration. A corresponding pruning algorithm based on reweighted training is then conducted to realize the desired sparsity.

From the perspective of acceleration optimizations at the hardware level, an edge FPGA-based 3-D CNN accelerator is designed and implemented, efficiently converting the KGRC sparsity into model inference acceleration. With four operation modes, this accelerator is general for both dense models and sparse models with kernel group sparsity in rows, columns or both, adopting and improving parallel computation techniques to accommodate weight sparsity in multiple dimensions and thus increase the computation parallelism and resource utilization efficiency. For further acceleration, 16, 8, and 4-bit quantization is additionally performed on the models, along with the low-precision computation support added on the accelerator. This accelerator design is implemented on the Xilinx ZCU102 FPGA platform, and tested for two representative 3-D CNNs, namely C3D [43] and R(2+1)D [44]. While C3D has convolutional kernels all of $3\times3\times3$ size, the R(2+1)D structure is more complicated with various kernel sizes across different layers, like $1\times3\times3$ and $3\times1\times1$. Our accelerator implementations show ability of speeding up these two 3-D CNN structures with various sparsity levels efficaciously.

The following list summarizes our main contributions.

1) *A fine-grained and balanced sparsity pattern called KGRC sparsity is proposed for weight tensors in 3-D CNNs, with the corresponding reweighted pruning algorithm*. This sparsity enables more pruning flexibility and increases the potential for higher pruning ratios while maintaining the accuracy.
2) *An edge FPGA-based 3-D CNN accelerator is designed and implemented, efficiently converting the KGRC sparsity into model inference acceleration*. This accelerator is general for both dense models and sparse models through four operation modes, in all of which the computation resources are fully utilized.
3) Experiments on two representative 3-D CNNs [C3D and R(2+1)D] on the Xilinx ZCU102 FPGA platform demonstrate the effectiveness of the proposed algorithm-hardware co-design framework with the KGRC sparsity and the accelerator design. With 8-bit quantization, KGRC achieves a good balance between the speedup, energy efficiency, and model accuracy, leading to acceleration ratios of $4.12\times$ for C3D and $3.85\times$ for R(2+1)D compared with the baselines for 16-bit dense models.

The remainder of this article is organized as follows. Section II introduces typical studies on model compression and hardware acceleration for 2-D and 3-D CNNs. Section III provides an overview of the proposed algorithm-hardware co-design framework. The fine-grained and balanced KGRC weight sparsity with the reweighted pruning algorithm is discussed in Section IV, and the FPGA accelerator design with four modes supporting sparsity in different dimensions is then presented in Section V. Section VI demonstrates the experimental results of model compression and hardware acceleration, and Section VII finally concludes this work.

II. Related Work

A. Weight Pruning for 2-D CNNs

Weight pruning reduces the demand for storage and computations by eliminating the redundant weights of CNNs. It can fall into three categories, namely unstructured pruning, coarse-grained structured pruning, and fine-grained structured pruning. Unstructured pruning removes the weight elements at arbitrary locations [8], [17], [18]. It can achieve the highest compression ratios among the three pruning types, but incurs significant indexing overhead in storage and logic consumption and degrades the parallelism in hardware implementations.

Fig. 1 shows three patterns of structured sparsity, including the coarse-grained row and column sparsity, as well as the fine-grained kernel pattern and connectivity sparsity. The row sparsity is in the output channel (filter) dimension.[1] Fig. 1(d) shows the column sparsity more intuitively than Fig. 1(b) by reshaping the kernels in each output channel into a 1-D array, which is typically adopted for matrix multiplication with the general matrix multiply (GEMM) algorithm. Coarse-grained structured pruning introduces much less indexing overhead, resulting in regular model structures that are friendly for parallel computations on hardware accelerators. Its shortcoming lies in evident drops in model accuracy.

Fine-grained structured sparsity attempts to balance between the fine granularity in unstructured sparsity and the regularity in coarse-grained structured sparsity. One way is to split the weight tensor into 2-D blocks and apply row and column sparsity on each weight block, which is referred to as block pruning [30]. The balance of column pruning in weight blocks is realized by setting the same sparsity rate in all the blocks to maximize GPU parallelism [35], whereas this block-based sparsity is not as suitable for FPGAs, which will be elaborated in Section IV. The work [29] presents another type of fined-grained sparsity at the kernel level, as shown in Fig. 1(c), where the kernel pattern selects the appropriate elements in each kernel from several pattern candidates with the same number of removed elements (4 in 9 here), and the connectivity sparsity prunes the whole kernel to cut the connection between a pair of input and output channels.

[1]Filter sparsity is sometimes mentioned as channel sparsity [21]. They are essentially equivalent, as removing certain filters in the current layer makes the corresponding (input) channels in the next layer invalid. However, in most cases the weight tensor of a convolutional or fully connected layer is followed by bias or batch normalization parameters, which are retained to maintain the model accuracy, and the channels in the next layer cannot be simply removed.

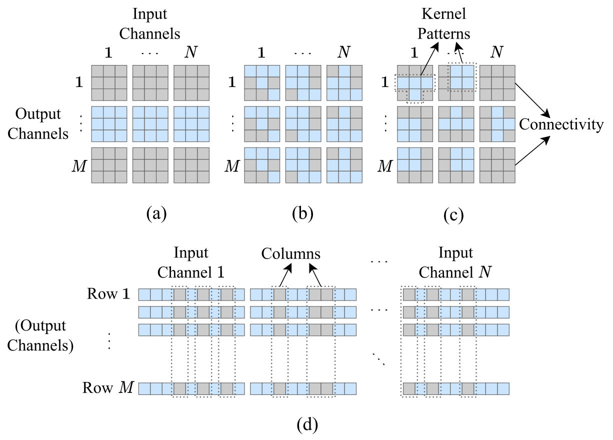


Fig. 1. Structured sparsity in 2-D CNN weight kernels. Pruned weights are set to zero and marked in gray. (a) and (b) are coarse-grained and (c) is fine-grained.

There also exists research on improving the pruning algorithms. ℓ_1 or ℓ_2 norm is incorporated as the penalty factor into the loss function for output and input channel pruning in [20], [21] and [47], and the alternating direction method of multipliers (ADMM) is further applied in [52] and [25] for accuracy improvement with dynamic regularization penalty.

B. Hardware Acceleration for 2-D and 3-D CNNs

Sparse CNN deployment on FPGAs concentrates primarily on 2-D CNNs like AlexNet and VGG-16. To support unstructured sparsity, the MnnFast architecture [22] is built on the basis that the probability vector is highly sparse for memory-augmented neural networks, and the work [34] handles spectral-domain CNNs converting convolutions in the spatial domain into Hadamard products in the spectral (frequency) domain with Fast Fourier Transform, which may be not general for other types of CNNs. A tile lookup table (LUTs) is designed in [27] to match the index between sparse weights and input pixels, which introduces indexing overhead and irregularity in model structures to bring about evident parallelism degradation, as the tiling size is required to be large along the output and input channel dimensions for sufficient computation parallelism.

Some endeavors pay attention to structured sparsity, or mixing sparsity at multiple granularities. Exploiting block, cyclic, and hybrid memory partition patterns, the OMNI framework [26] prunes the model weights in a structured manner, and prevents the workload imbalance by maintaining the same sparsity rate for different memory partition groups. The SparTen accelerator [13] presents a greedy load-balancing strategy that groups filters by the whole-filter density and finer-grained density like chunking every 128 elements. In [3], filter pruning for convolutional layers is mixed with unstructured pruning for fully connected layers, for which the compressed sparse column format is improved by storing the number of zeros between two contiguous nonzero weights instead of row indices. Aside from selecting and clustering nonzero weights in sparse filters from intralayer unstructured pruning, the method in [49] realizes interlayer compression by reducing convolution operations when the convolutional layer is followed by a max or average pooling layer.

As for 3-D CNNs, frequency domain conversion is applied in [4] on 3-D ResNet-18 and U-Net to sparsify convolutional filters, but there is no related research on hardware implementation. C3D is first implemented on FPGA in [12] by employing pixel blocking and filter parallelism, and block floating-point (BFP) representation is utilized in [10] to reduce the bit-width while maintaining the classification accuracy. The Winograd algorithm is incorporated to reduce the amount of computations for C3D [38], [39]. The work [11] proposes a specific 3-D CNN structure with "3-D−1" bottleneck residual blocks, and applies optimizations, including online blocking and kernel reuse on FPGAs. Nonetheless, these previous studies only focus on the basic C3D or custom 3-D CNN structures and lack generality. For example, the Winograd algorithm is only suitable for the uniform $3\times3\times3$ kernel size in C3D. If applied on other 3-D CNNs with various kernel sizes, it may cause significant accuracy loss.

C. Comparison of Our Solution With Existing Similar Efforts

One of the first studies on FPGA-aware pruning for 3-D CNNs applies a relatively coarse-grained group-wise sparsity pattern [42], making the whole weight kernel group as the basic pruning unit. The coarse pruning granularity brings more difficulty in maintaining the model accuracy than our proposed balanced KGRC sparsity under the same compression setting. In addition, the concentration of this work is only on pruning, but not on quantization, which confines further acceleration.

The fine-grained KGS sparsity for 3-D CNNs has been implemented on mobile phones [33] by employing a compiler-assisted code generation framework. The proposed balanced KGRC sparsity extends from this work. The differences of KGRC from KGS lie in the following aspects.

1) KGRC covers both row and column sparsity in kernel groups, while KGS only involves column sparsity.
2) The reweighted pruning algorithm for KGRC uses pruning ratios to directly control the number of zero weights for each layer from scratch, which is beneficial to parallel computations in FPGA implementations. In the previous KGS designs targeting mobile devices, a threshold value is used for weight pruning, and the number of zero weights is unknown until pruning is finished.
3) The sparsity rate for rows or columns is set the same for all the kernel groups in a specific 3-D CNN layer, which is also favorable for FPGA implementations.

The most similar study [55] to ours with regard to the FPGA implementation adopts the same sparsity pattern as KGS, but evaluates only on fully sequential 2-D CNNs, including LeNet, AlexNet, and VGG-16, with relatively low overall sparsity rates (lower than 40%). Additionally, the computation parallelism is only explored in the output and input channel dimensions, which is not sufficient for 3-D CNNs with high pruning ratios, especially in cases with quantization.

III. Overview of Proposed 3-D CNN Compression-Accelerator Co-Design Framework

Our proposed 3-D CNN acceleration framework leverages algorithm-hardware co-design. At the algorithm level,

TABLE I
NOTATIONS IN THE PROPOSED MODEL COMPRESSION AND ACCELERATION FRAMEWORK

Type	Notation	Description	
General	M (N)	Number of output (input) channels	
	K_D (K_H, K_W)	Kernel depth (height, width);	$K_l = K_D \times K_H \times K_W$
	D (H, W)	Depth (height, width) of output feature map;	D^{in} $(H^{\text{in}}, W^{\text{in}})$ for input feature map
	T_K	Tiling size in kernel dimension	
Pruning	G_K	Kernel group size in kernel dimension;	$G_K = \min(K_l, T_K)$, $T_K \le K_l$ in most cases
	G_M (G_N)	Kernel group size in output (input) channel dimension;	G_M rows and $G_N \times G_K$ columns in each kernel group
	R_R (R_C)	Sparsity rate of rows (columns) in each kernel group;	$0 \le R_R, R_C \le 1$; pruning ratio $= \frac{1}{(1-R_R)(1-R_C)}$
Hardware	A_b	Number of low-bitwidth data concatenated as one	
	T_M (T_N)	Tiling size of output (input) channels;	$T_M\%A_b = 0, T_N\%A_b = 0$ for full memory utilization
	T_D (T_H, T_W)	Tiling size of output feature depth (height, width);	$T_F = T_D \times T_H \times T_W$
	P_M (P_N)	Parallel factor in output (input) channel dimension;	$T_M(1 - R_R)\%P_M = 0$
	P_K (P_F)	Parallel factor in kernel (feature) dimension;	$T_K(1 - R_C)\%P_K = 0$
	S_{DSP} (S_{BRAM})	Available number of DSPs (BRAMs) on FPGA	
	B_{in} $(B_{\text{out}}, B_{\text{wgt}})$	Number of AXI ports for data transfer of input (output, weight) tile	

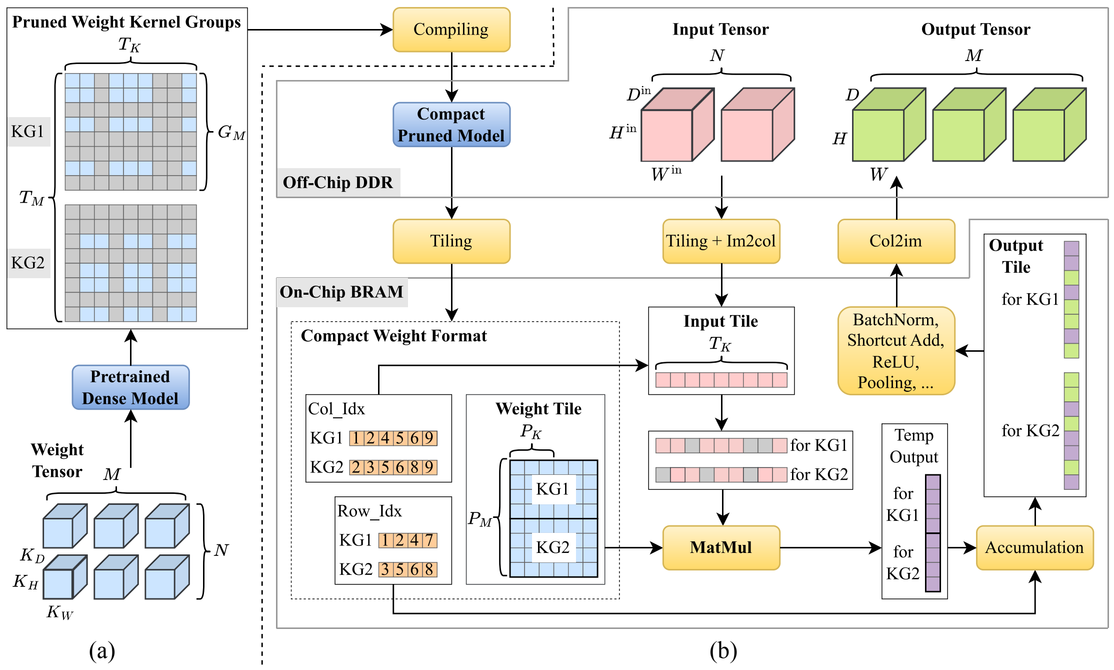


Fig. 2. Algorithm-hardware co-design framework for 3-D CNN acceleration with balanced kernel group sparsity. (a) Algorithm side. (b) Hardware side.

fine-grained and structured sparsity is performed with balance among the kernel groups of a pretrained (dense) 3-D CNN. At the hardware level, an FPGA-based accelerator is designed and implemented to support the proposed sparsity and therefore convert the reduction of operations into actual acceleration of 3-D CNNs.

Table I gives all the notations appearing in the framework. As displayed in Fig. 2, the whole weight tensor in a 3-D CNN layer originally has the size of $M \times N \times (K_D \times K_H \times K_W)$ in five dimensions. Each output channel (namely filter) is treated as a row, and the other four dimensions (input channel, kernel depth, kernel height, and kernel width) are reshaped as one treated as the column dimension. For convenience in representation and exploration of computation parallelism later, each 3-D kernel in the lth layer is reshaped into an array of 1-D called "the kernel dimension," with the size of $K_l = K_D \times K_H \times K_W$. Similarly, each output feature map is reshaped into an array of 1-D called "the feature dimension," with the size of $F = D \times H \times W$. Considering the constrained amount of on-chip storage and computation resources on edge FPGAs, the accelerator is designed leveraging the common-used loop tiling technique [50], which can be generally applied onto the output and input channel, as well as the kernel and feature dimensions. The size of a dense weight tile is $T_M \times T_N \times T_K$, and that of an output tile is $T_M \times T_F$, where $T_F = T_D \times T_H \times T_W$.

It is important to mention the difference between the kernel group size G_M, G_N, and G_K and the tiling size T_M, T_N, and T_K. G_M, G_N, and G_K determine the granularity of pruning, which depends on the model accuracy requirement, while T_M, T_N, and T_K are the numbers of output channels, input channels, and kernel elements to be handled at a time on hardware accelerators, and are affected by the number of available resources on the FPGA and the speed requirement. Considering both the convolutional kernel size and the hardware memory space, G_K is actually set to the lower value of K_l and T_K, and $T_K \le K_l$ in most cases (e.g., K_l is 3×3×3 in C3D and 1×3×3 in R(2+1)D). In spite of this, the following discussion focuses on $G_K = K_l$ cases for simplicity without loss of generality. As pruning will cause sparsity in weights, the accelerator is designed to manage multiple kernel groups for full exploitation of the computation resources, i.e., a weight tile contains $\lceil (T_M/G_M) \rceil \times \lceil (T_N/G_N) \rceil$ kernel groups.

Fig. 2 illustrates the acceleration framework overview with tiling sizes $T_N = T_F = 1$, focusing on the output channel and kernel dimensions. The accelerator is designed in favor of the compressed weight format containing three parts: 1) the compact weight tensor of all the nonzero elements that are not pruned; 2) row indices in the original dense kernel group for nonzero weights, which will be used in accumulation after MatMul of the input and weight tensors; and 3) column indices in the original dense kernel group for nonzero weights, which will be used in data reading of the input tensor. When the weight tensor of a 3D CNN layer is pruned, the number of nonzero rows or columns is the same across all the kernel groups, so that the pruned weight tensor can be rearranged in a compact manner, and the computation resources allocated by the implementation can always be exploited adequately to assist the inference acceleration.

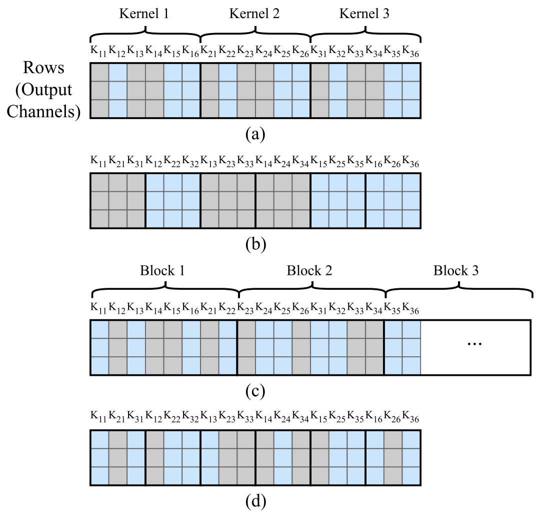


Fig. 3. Comparison between kernel group-based [(a) and (b)] and block-based [(c) and (d)] column sparsity. (a) and (c) indicates the original arrangement of weight elements in kernels. (b) and (d) give another arrangement concatenating the same element in all the kernels together.

IV. 3-D CNN Compression With Balanced Kernel Group Sparsity

Several studies have been conducted on mobile phones and GPUs for fine-grained structured sparsity, such as block-based column (and row) sparsity [30], [35] and kernel group-based column sparsity [33]. Different from these devices that can finish image-to-column of the whole input and weight tensors before computing, edge FPGAs must rely on loop tiling to buffer partial input and weight data due to small on-chip memory. This necessitates novel design principles for weight sparsity. It is worth reemphasizing the key features of our KGRC sparsity and reweighted pruning algorithm, including 1) covering both row and column sparsity in kernel groups; 2) using pruning ratios to directly control the number of zero weights for each layer from scratch; and 3) the same row or column sparsity rate for all the kernel groups in a 3-D CNN layer. The 2) and 3) are both beneficial to parallel computations in FPGA implementations.

The KGRC sparsity carries out column pruning at the kernel group level (one kernel group contains all the elements of one or more kernels), rather than at the block level [30], [35] (the elements in one kernel may be split and allocated to different blocks). The reason for this is explained with Fig. 3, with three rows of three kernels each having six elements, labeled as $K_{ij}, i = 1, 2, 3, j = 1, 2, \ldots, 6$. All the kernels compose a kernel group in (a), while they are split into blocks of size 3×8 in (c), where the sparsity distribution is more irregular. This is more clear when the same elements of the kernels are arranged together in (b) and (d). It can be seen from (b) that the 1st, 3rd, and 4th elements in all the kernels within the kernel group are removed, and only three indices need to be stored for input reading in the accelerator. In contrast, even balanced block-based column sparsity [50% sparsity in all the blocks in (c)] is still distributed irregularly in (d), and needs indexing for every retained element, impeding FPGA implementations with high parallelism. For a kernel group with size G_N along the input channel dimension, KGRC reduces the indexing cost by $G_N\times$ through eliminating irregular and repetitive reading of input elements.

A. Balanced Kernel Group Row and Column Sparsity

The granularity of kernel group sparsity is between those of coarse-grained structured and fine-grained unstructured sparsity, leading to both regular structures friendly to hardware acceleration and fine granularity to maintain the model accuracy. Specifically, the weight tensor of a 3-D CNN layer with M output channels and N input channels is partitioned into equal-sized kernel groups, each containing $G_M \times G_N$ kernels. Fig. 4(a) and (b), respectively, give an example of the row and column sparsity in a kernel group, called the kernel group row (KGR) and column (KGC) sparsity. Within the kernel group, KGR removes the whole rows, while KGC applies the same sparsity pattern for each kernel. The KGRC sparsity illustrated in Fig. 4(c) combines KGR and KGC, removing both rows and columns in a kernel group concurrently, which is displayed more clearly in Fig. 4(d).

The setting of the kernel group sparsity is a tradeoff between the model accuracy and the inference speed. We would like a relatively small kernel group size for high accuracy, but it incurs difficulty for pruned models to fully exploit the parallelism on FPGAs, since the allocation of on-chip buffers and computation resources is fixed in an FPGA implementation, and can hardly adapt to the irregularity from pruned models. Considering the KGR sparsity with $G_M = 8$ and $P_M = 4$, when the number of nonzero rows after pruning is lower than 4, the resource utilization rate cannot reach 100%. If some kernel group contains five nonzero rows, two cycles are needed to finish the computations, and the resource utilization rate in the second cycle is only 25%.

Our solution to this problem is to fix the numbers of pruned rows and columns for all the kernel groups in the weight tensor, to balance the hardware workload between these kernel

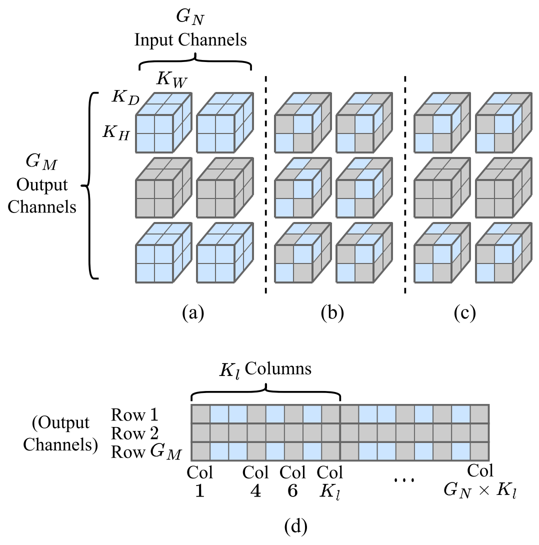


Fig. 4. Kernel group sparsity patterns for in 3-D CNNs, including row sparsity (KGR), column sparsity (KGC), and row-column sparsity (KGRC), with each kernel group containing $G_M \times G_N$ kernels. (a) KGR sparsity. (b) KGC sparsity. (c) KGRC sparsity. (d) KGRC sparsity with reshaped kernels.

groups. In addition, the indices of pruned elements (columns) are set the same for all the kernels within a kernel group, to reduce the indexing cost of input reading on hardware accelerators. For more flexibility, the locations of pruned rows and columns are allowed to be various in different kernel groups. Furthermore, for hardware acceleration, multiple kernel groups are processed simultaneously to improve the parallelism, which will be elaborated in Section V-B.

B. Reweighted Pruning With Kernel Group Sparsity

To realize the desired sparsity, the loss function associated with a CNN needs to be modified. For a 3-D CNN with a total of L layers, the original loss function without pruning is denoted as $f(\mathbf{W}, \mathbf{b})$, where $\mathbf{W} = \{\mathbf{W}_l\}_{l=1}^{L}$ is the collection of weight tensors, and $\mathbf{b} = \{\mathbf{b}_l\}_{l=1}^{L}$ the collection of bias tensors from layers $l \in \{1, \ldots, L\}$. Partitioning $\mathbf{W}_l$ along the output and input channel dimensions generates kernel groups $\{\mathbf{W}_l^{\mathcal{G}_{p,q}}\}$ each with $G_M \times G_N$ kernels, where $p \in \{1, 2, \ldots, \lceil (M/G_M) \rceil\}$, $q \in \{1, 2, \ldots, \lceil (N/G_N) \rceil\}$. Reshaping the kernels into 1-D arrays with size K_l, each element in the kernel group $\mathbf{W}_l^{\mathcal{G}_{p,q}}$ can be expressed as $\mathbf{W}_l^{\mathcal{G}_{p,q}}(m, n, k)$, where $m \in \{(p-1)G_M+1, \ldots, pG_M\}$, $n \in \{(q-1)G_N+1, \ldots, qG_N\}$, and $k \in \{1, \ldots, K_l\}$.

The pruning algorithm with reweighted regularization adds a regularization term to the loss function to penalize the weights, which can be formulated as

$$\underset{\mathbf{W},\mathbf{b}}{\text{minimize}} \;\; f(\mathbf{W}, \mathbf{b}) + \lambda \sum_{l=1}^{L} R\big(\mathbf{W}_l\big) \tag{1}$$

where $R(\mathbf{W}_l)$ is the regularization term for sparsity, and λ acts as a global penalty factor indicating the importance of regularization. No matter which kernel group sparsity is adopted, $R(\mathbf{W}_l)$ for the lth layer is the summation of the terms of all the kernel groups, i.e., $R(\mathbf{W}_l) = \sum_{p=1}^{P} \sum_{q=1}^{Q} R(\mathbf{W}_l^{\mathcal{G}_{p,q}})$, where $R(\mathbf{W}_l^{\mathcal{G}_{p,q}})$ appears in the form of ℓ_g norm that can be selected from ℓ_1 norm, ℓ_2 norm or their combination. $R(\mathbf{W}_l)$ aims to dynamically reweight the penalties, i.e., reduce the penalties on weights with large magnitudes (that are likely to be more critical), and increase the penalties on weights with small magnitudes, which is performed in a systematic, gradual way to avoid a greedy solution pruning a large number of weights at the early stage.

For the KGR sparsity, the regularization term of a kernel group can be expressed as

$$R\big(\mathbf{W}_l^{\mathcal{G}_{p,q}}\big) = \sum_{m \in \mathcal{G}_{p,q}} \left(\mathcal{P}_{l,t}^{\mathcal{G}_{p,q},m} \circ \left\| \mathbf{W}_l^{\mathcal{G}_{p,q}}(m, :, :) \right\|_g \right) \tag{2}$$

where $\circ$ denotes element-wise multiplication. $\|\cdot\|_g$ denotes the ℓ_g norm, and

$$\left\| \mathbf{W}_l^{\mathcal{G}_{p,q}}(m, :, :) \right\|_g = \sqrt[g]{\sum_{n \in \mathcal{G}_{p,q}} \sum_{k}^{K_l} |\mathbf{W}_l(m, n, k)|^g}. \tag{3}$$

$\mathcal{P}_{l,t}^{\mathcal{G}_{p,q},m}$ is the penalty parameter for reweighting, and is updated in every iteration t following:

$$\mathcal{P}_{l,(t+1)}^{\mathcal{G}_{p,q},m} = \frac{1}{\left\| \mathbf{W}_{l,t}^{\mathcal{G}_{p,q}}(m, :, :) \right\|_g^2 + \epsilon} \tag{4}$$

where $\mathbf{W}_{l,t}^{\mathcal{G}_{p,q}}$ is the instance of $\mathbf{W}_l^{\mathcal{G}_{p,q}}$ in iteration t, and ϵ is a small positive number to avoid a zero denominator. The reweighted regularization pruning algorithm updates the penalty parameters based on current weight values without incurring extra hyperparameters, and thus converges in short order. After this, the weights close to zero are pruned (set to zero). Considering the more complicated KGC sparsity, the regularization term of a kernel group can be written as

$$R\big(\mathbf{W}_l^{\mathcal{G}_{p,q}}\big) = \sum_{k=1}^{K_l} \left(\mathcal{P}_{l,t}^{\mathcal{G}_{p,q},k} \circ \left\| \mathbf{W}_l^{\mathcal{G}_{p,q}}(:, :, k) \right\|_g \right) \tag{5}$$

where

$$\left\| \mathbf{W}_l^{\mathcal{G}_{p,q}}(:, :, k) \right\|_g = \sqrt[g]{\sum_{(m,n) \;\in \mathcal{G}_{p,q}} |\mathbf{W}_l(m, n, k)|^g} \tag{6}$$

and the penalty parameter is

$$\mathcal{P}_{l,(t+1)}^{\mathcal{G}_{p,q},k} = \frac{1}{\left\| \mathbf{W}_{l,t}^{\mathcal{G}_{p,q}}(:, :, k) \right\|_g^2 + \epsilon}. \tag{7}$$

An intuitive method to achieve the KGRC sparsity is mixing the KGR and KGC sparsity by solving the problem (1) with the kernel group-based regularization terms (2) and (5) sequentially and separately. If the weight tensors are first pruned with the KGR sparsity, then they are further pruned with the KGC sparsity while masking the pruned rows as zeros. This method can ensure the convergence to a mixture of these two sparsity patterns, but the performance of the latter sparsity is affected by the former one, as training for one sparsity pattern does not consider the potential existence of the

Algorithm 1: Reweighted Pruning With KGRC Sparsity

```
Input : Pretrained model weights $\mathbf{W} = \{\mathbf{W}_l\}_{l=1}^{L}$;
        $M$, $N$, $G_M$, $G_N$, $K_l$ defined in Table I;
        Total training epochs $T$;
        Initialized reweighting penalty parameters
        $\mathcal{P}_{l,0}^{\mathcal{G}_{p,q},m}$, $\mathcal{P}_{l,0}^{\mathcal{G}_{p,q},k}$;
        Iteration times $I \subseteq \{1, \ldots, T\}$ to update them;
Output: Pruned model with KGRC sparsity.
   // Reweighted training
1  for $t \leq T$ do
2      foreach layer $l$ do
3          $\mathbf{W}_l \leftarrow$ SGDTrain($\mathbf{W}_l$) following (8);
4          if $t \in I$ then
5              Update $\mathcal{P}_{l,t}^{\mathcal{G}_{p,q},m}$ following (4);
6              Update $\mathcal{P}_{l,t}^{\mathcal{G}_{p,q},k}$ following (7);
   // Hard pruning
7  $\{\mathbf{W}_l^{\mathcal{G}_{p,q}}\} \leftarrow$ Partition($\mathbf{W}_l$),    $p \in \{1, \ldots, \lceil \frac{M}{G_M} \rceil\}$,
   $q \in \{1, \ldots, \lceil \frac{N}{G_N} \rceil\}$;
8  foreach $\mathbf{W}_l^{\mathcal{G}_{p,q}}$ do
9      $\mathbf{W}_l^{\mathcal{G}_{p,q}}(:, n, k) \leftarrow$ RowPrune($\mathbf{W}_l^{\mathcal{G}_{p,q}}(:, n, k)$),    $(n, k) \in \mathcal{G}_{p,q}$;
10     $\mathbf{W}_l^{\mathcal{G}_{p,q}}(m, n, :) \leftarrow$ ColPrune($\mathbf{W}_l^{\mathcal{G}_{p,q}}(m, n, :)$),    $(m, n) \in \mathcal{G}_{p,q}$,
       $\mathcal{G}_{p,q} = m \in \{(p-1)G_M + 1, \ldots, pG_M\}$, $n \in$
       $\{(q-1)G_N + 1, \ldots, qG_N\}$, $k \in \{1, \ldots, K_l\}$;
   // Retraining
11 $\mathbf{W}_l \leftarrow$ SGDTrain($\mathbf{W}_l$) while fixing the zero elements.
```

other sparsity pattern. The better way is simultaneous training of these two as

$$\underset{\mathbf{W},\mathbf{b}}{\text{minimize}}\ f(\mathbf{W}, \mathbf{b}) + \frac{\lambda}{2} \sum_{l=1}^{L} \sum_{p=1}^{P} \sum_{q=1}^{Q} \left[\sum_{m \in \mathcal{G}_{p,q}} \left(\mathcal{P}_{l,t}^{\mathcal{G}_{p,q},m} \circ \left\| \mathbf{W}_l^{\mathcal{G}_{p,q}}(m, :, :) \right\|_g \right) + \sum_{k=1}^{K_l} \left(\mathcal{P}_{l,t}^{\mathcal{G}_{p,q},k} \circ \left\| \mathbf{W}_l^{\mathcal{G}_{p,q}}(:, :, k) \right\|_g \right) \right] \tag{8}$$

with the global penalty factor halved. This can save the training effort and explore with more flexibility for better overall sparsity.

Algorithm 1 describes the steps in reweighted training for KGRC sparsity. Different from the method in [33] where the pruning ratios for KGS are known only after the pruning is finished (decided by a threshold value for weights), here the pruning ratios of KGR and KGC follow predefined settings of each layer. Masked retraining is then executed to regain the model accuracy without updating the pruned weights fixed to zeros.

C. Quantization of Sparse 3-D CNNs

To further explore the effect of compression on model accuracy and hardware efficiency, quantization is carried out on top of pruning for 3-D CNNs, following the ADMM-based method in [37]. Both weights and activations are quantized in one of 16, 8, and 4-bit precisions. For weights, only the retained nonzero elements are quantized, and the locations of pruned elements are no longer changed, which is similar to the processing in masked retraining.

V. FPGA Accelerator Design and Optimization for Kernel Group Sparsity

The proposed accelerator design for KGRC sparsity processes the convolution computations layer by layer, supporting four operation modes, respectively, for 1) dense models without sparsity; 2) KGR sparsity; 3) KRC sparsity; and 4) KGRC sparsity. For fair comparisons, the baseline designs only supporting dense models with 16, 8, and 4-bit precisions are also implemented. In each of these precisions, the KGRC accelerator design outperforms the baseline in terms of the model inference speed without introducing evident resource overhead, as will be elaborated in Section VI-C.

Fig. 2(b) illustrates the workflow of the proposed framework on the hardware side. The accelerator is designed based on loop tiling due to the limited amount of storage and computation resources on edge FPGAs. The input feature maps and model weight tensor are split into tiles, and when a group of input and weight tiles is loaded from the off-chip DDR to the on-chip block RAM (BRAM) buffers, the image-to-column (Im2col) procedure is performed on the input tile for convenience of array partitioning and thus parallel computations along the feature and kernel dimensions. Correspondingly, column-to-image (Col2im) is performed on the output tile generated by the input and weight tiles before storing from on-chip to off-chip.

MultiDimensional Parallelism: To handle multiply-add (MAC) operations as the primary computations in 3-D CNNs, most of the utilized computation resources are allocated for parallel computations along the output channel, input channel, and feature dimensions, in both baseline design for dense models without pruning and the proposed design for KGRC-sparsified models. In the KGRC design, the parallelism also exists in the kernel dimension to maintain high utilization rates of computation resources after model compression. The parallel factor along each dimension is determined based on the model pruning ratio as well as the quantization precision.

A. Acceleration Mechanism With KGRC Sparsity

Compact Weight Format: For each kernel group within a KGRC-sparsified weight tensor, the row and column indices of nonzero elements are prestored off-chip. Take the two weight kernel groups (denoted as KG1 and KG2) in Fig. 2(a) as an instance, each kernel group has eight output channels (with the original indices $[1, 2, \ldots, 8]$), one input channel, and $T_k = 9$ as the tiling size along the kernel dimension (with the original indices $[1, 2, \ldots, 9]$). The indices Row_Idx for nonzero rows along the output channel dimension are $[1, 2, 4, 7]$ for KG1 and $[3, 5, 6, 8]$ for KG2. Similarly, the nonzero column indices Col_Idx for all the kernels within KG1 are $[1, 2, 4, 5, 6, 9]$, and those for KG2 are $[2, 3, 5, 6, 8, 9]$, as displayed in the Compact Weight Format part in Fig. 2(b). As mentioned in Section IV-A, the pruning ratio for rows or columns is the same across all the kernel groups for a specific 3-D CNN

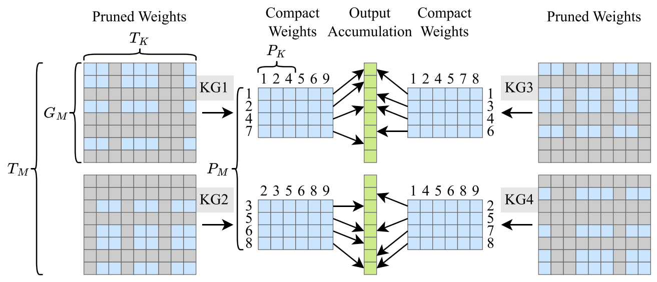


Fig. 5. Processing of multiple sparse kernel groups.

layer, resulting in a uniform size for kernel groups after eliminating the zeros to form a compact model structure. In actual hardware implementations, the row and column indices are stored as relative values within the kernel group to save memory usage. Specifically, the four nonzero rows in KG2 are actually labeled with $[3, 5, 6, 8] - 1 = [2, 4, 5, 7]$ in 3-bit, instead of the absolute indices of output channels in the whole weight tensor. The relative indices of columns depend on T_K, which is in most cases not smaller than $\max_l K_l$ for a model with L layers. While for a large kernel size like $K_l = 27$ in C3D, T_K must be smaller than K_l due to the limited on-chip memory capacity. In the case where a C3D kernel is split into three parts each with $T_K = 9$, the relative indices can be represented by 4-bit 0 to 8 values.

Convolution Processing for Sparse and Dense Layers: The convolution computations are processed on-chip with the following steps.

1) According to the column indices (Col_Idx), select the valid input elements (from the Input Tile) of each kernel group corresponding to the nonzero weight columns (in the Weight Tile) retained after pruning.
2) Execute the MatMul to output a temporary result.
3) According to the row indices (Row_Idx), accumulate the temporary multiplication result corresponding to each nonzero weight row (in the Weight Tile) to the valid output channel of the Output Tile.
4) After accumulating from all the input channels for the result of an output channel, post-process with operations, including batch normalization, shortcut addition, activation (ReLU), pooling, etc.

Fig. 5 further illustrates the convolution processing with more kernel groups and their respective row and column indices. Since multiple kernel groups along the row dimension (e.g., KG1 and KG3) may have different nonzero row indices, all the output channels require to be maintained for the output tile. Similarly, the input tile must maintain all the elements after Im2col expansion to accommodate different nonzero column indices from multiple kernel groups along the column dimension (e.g., KG1 and KG2). A special case is for a dense layer, for which one kernel group can be split into two compact weight matrices, respectively, with continuous row indices 1, 2, 3, 4 and 5, 6, 7, and 8 to compose one weight tile. The convolutions can be managed in the same manner as for sparse layers. In summary, the column indices are used for reading from the Input Tile, while the row indices are for accumulating to the Output Tile.

B. Size Setting of Kernel Groups and Weight Tiles

A weight tile as the unit in hardware acceleration is different from a weight kernel group as the unit in kernel group-wise pruning. Each tile may consist of one kernel group, or multiple kernel groups that are adjacent along the output channel dimension for concurrent processing. The kernel group size has a direct influence on the accuracy of pruned models, and the number of kernel groups in each tile depends on the parallelism requirement.

Setting Pruning Ratios: The pruning ratios can be different across the layers in a 3-D CNN. In the example in Figs. 2 and 5, the kernel group size is $G_M \times G_N = 8 \times 1$, and the kernel tile size is $T_K = K_l = 9$. The sparsity rate of rows is $R_R = 50\%$, making four of the eight rows retained in each kernel group. Besides, the sparsity rate of columns is $R_C = 33.3\%$, making six of the nine elements retained in each kernel, and the locations of these six elements are the same across all the kernels within each kernel group. The nonzero weight tile size is actually $T_M(1 - R_R) \times T_N \times T_K(1 - R_C) = 8 \times 1 \times 6$, with $T_M = 2 \times G_M$ and $T_N = G_N$, and the overall pruning ratio is $(1/1 - R_R) \times (1/1 - R_C) = 3\times$.

Relationship of Pruning Ratio, Tiling and Parallel Parameters: To fulfill the requirements of parallel factors P_M and P_K, and fully utilize computation resources, the tiling and parallel parameters should satisfy

$$\begin{aligned} T_M(1 - R_R)\%P_M &= 0 \\ T_K(1 - R_C)\%P_K &= 0 \end{aligned} \tag{9}$$

which applies to sparse layers and also dense layers with $R_R = 0$ and $R_C = 0$. Another parallel parameter P_N along the input channel dimension can be determined according to P_M and P_K as well as the number of available resources.

Tiled Convolution Details: The procedure of tiled convolutions for 3-D CNNs is detailed in Algorithm 2, which is applicable for both dense layers and sparse ones. In the Compute module, the *for* loops under the #PIPELINE pragma are all unrolled, generating a total parallelism of $P_{\text{tot}} = P_F \times P_K \times P_M \times T_N$. With 16-bit precision, this parallelism can support P_{tot} simultaneous MAC operations. Along the input channel dimension, the tiling size T_N is equal to the kernel group size G_N, since covering more kernel groups will bring about higher indexing storage demand. The parallel factor is $P_N = T_N$ to exploit the computation potential. Consequently, T_N, G_N, and P_N are set equal to each other.

C. FPGA Resource and Performance Analysis

Computation Resource Analysis: FPGAs mainly contain two types of computation resources, DSPs, and LUTs. In line with the analysis in Algorithm 2, the total DSP usage of a KGRC design can be calculated as

$$U_{\text{DSP}} = U_{\text{DSP}}^{\text{prec}} \times P_M \times T_N \times P_K \times P_F \tag{10}$$

where $U_{\text{DSP}}^{\text{prec}}$ is the DSP usage of one MAC operation in a specific precision. For 16, 8, and 4-bit, the $U_{\text{DSP}}^{\text{prec}}$ values are, respectively, 1, 0.5, and 0.25, as one 27×18-bit DSP can accommodate one 16×16-bit, two 8×8-bit, or four 4×4-bit multiplications, which has been demonstrated in the work [41].

Algorithm 2: Tiled Convolution for 3-D CNNs in Dense and KGRC-Sparsified Modes

```
Parameter: $T_F = T_D \times T_H \times T_W$, $T_F^{\text{in}} = T_D^{\text{in}} \times T_H^{\text{in}} \times T_W^{\text{in}}$;
           $T_N = G_N = P_N$.
1  for $\lceil \frac{D}{T_D} \rceil \times \lceil \frac{H}{T_H} \rceil \times \lceil \frac{W}{T_W} \rceil$ feature tiles do
2      for $\lceil \frac{M}{T_M} \rceil \times \lceil \frac{N}{T_N} \rceil$ output and input channel tiles do
3          Load $\mathbf{I}_{\text{buf}}^{\text{in}}[T_N][T_F^{\text{in}}]$;
4          Load Row_Idx$[T_M(1-R_R)]$;
5          for $\lceil \frac{K_l}{T_K} \rceil$ kernel tiles do
6              $\mathbf{I}_{\text{buf}}[T_N][T_F][T_K] \leftarrow$ Im2Col($\mathbf{I}_{\text{buf}}^{\text{in}}$);
7              Load Col_Idx$[\frac{T_M}{G_M}][T_K(1-R_C)]$;
8              Load $\mathbf{W}_{\text{buf}}[T_M(1-R_R)][T_N][T_K(1-R_C)]$;
9              $\mathbf{O}_{\text{buf}}[T_M][T_F] \leftarrow$ Compute($\mathbf{I}_{\text{buf}}, \mathbf{W}_{\text{buf}}$);
10         Post-Processing (BatchNorm, Shortcut, ReLU, Pool);
11         Store $\mathbf{O}_{\text{buf}}[T_M][T_F]$;
12 Compute($\mathbf{I}_{\text{buf}}, \mathbf{W}_{\text{buf}}$):
13 for $t_f = 0$; $t_f < T_F$; $t_f += P_F$ do
14     for $t_k = 0$; $t_k < T_K(1-R_C)$; $t_k += P_K$ do
15         for $t_m = 0$; $t_m < G_M(1-R_R)$; $t_m += P_M$ do
               #PIPELINE
16             for $g_m \leftarrow 0$ to $\frac{T_M}{G_M}$ do
                   #UNROLL
17                 for $k \leftarrow 0$ to $P_K$, $m \leftarrow 0$ to $P_M$ do
                       #UNROLL
18                     $i_k = $ Col_Idx$[g_m][t_k + k]$;
                       $i_w = g_m \times G_M(1-R_R) + t_m + m$; $i_o = $ Row_Idx$[i_w]$;
19                     for $f \leftarrow 0$ to $P_F$, $n \leftarrow 0$ to $T_N$ do
                           #UNROLL
20                         $\mathbf{O}_{\text{buf}}[i_o][t_f + f] +=$
                           $\mathbf{I}_{\text{buf}}[n][t_f + f][i_k] \times \mathbf{W}_{\text{buf}}[i_w][n][t_k + k]$;
```

Although it is difficult to estimate the total LUT usage, the required LUT amount for accumulation after convolutions can be derived as

$$U_{\text{LUT}} = U_{\text{LUT}}^{\text{prec}} \times T_M \times P_F \tag{11}$$

considering the parallel factors T_M for the output channel dimension and P_F for the feature dimension, with $U_{\text{LUT}}^{\text{prec}}$ as the LUT usage of one MAC in a specific precision.

On-Chip Memory Resource Analysis: The BRAMs are utilized to buffer the input and output feature tiles as well as the weight tiles for the convenience of on-chip computations. For the Im2col conversion, the BRAM usage for the input tile is

$$U_{\text{BRAM}}^{\text{in}} = \frac{T_N}{A_b} \times \left\lceil \frac{T_F \times T_K \times b \times A_b}{18\text{k}} \right\rceil \tag{12}$$

where b stands for the bit-width, and 18k bits is the typical size of a single BRAM bank on Xilinx FPGAs. Data packing is used to save the memory and reduce the data transfer latency by concatenating A_b low-precision numbers into one, and the packing size A_b depends on b. In our implementations, $A_b = 4$ for 16-bit precision, and $A_b = 8$ for 8 and 4-bit precisions. The $\mathbf{I}_{\text{buf}}^{\text{in}}$ data before Im2col needs $(T_N/A_b) \times \lceil([T_F^{\text{in}} \times b \times A_b]/18\text{k})\rceil$ BRAMs, which is negligible compared with $U_{\text{BRAM}}^{\text{in}}$. Similarly, the BRAM usage of weight and output tiles are

$$U_{\text{BRAM}}^{\text{wgt}} = \frac{T_N}{A_b} \times \left\lceil \frac{T_M(1-R_R) \times T_K(1-R_C) \times b \times A_b}{18\text{k}} \right\rceil$$

$$U_{\text{BRAM}}^{\text{out}} = \frac{T_M}{A_b} \times \left\lceil \frac{T_F \times b \times A_b}{18\text{k}} \right\rceil. \tag{13}$$

T_M and T_N are usually set such that $T_M \% A_b = 0$ and $T_N \% A_b = 0$ to optimize the BRAM utilization rate. The constraints of the most intensive FPGA resources (i.e., DSPs and BRAMs, as LUTs and flip-flop (FFs) are usually sufficient) for the accelerator design are therefore expressed by

$$\begin{aligned} 2 \times U_{\text{BRAM}} &\leq S_{\text{BRAM}} \\ U_{\text{DSP}} &\leq S_{\text{DSP}} \times \rho_{\text{DSP}} \end{aligned} \tag{14}$$

employing the double buffering technique to overlap the data transfer latency with computations. The DSP utilization ratio is restricted below a certain level ($\rho_{\text{DSP}} \leq 80\%$ generally) to avoid placement and routing issues during implementations.

Latency Performance Analysis: The model inference latency is proportional to the clock cycle count given a specific working frequency of the accelerator implementation. The latency performance is analyzed based on one group of input feature, output feature, and weight tiles, as the data of one layer are handled group (of tiles) by group. Im2col can be performed by reading each element in $\mathbf{I}_{\text{buf}}^{\text{in}}$ only once, consuming clock cycle counts calculated as

$$L_{\text{in}} = \frac{T_N}{A_b} \times \left\lceil \frac{T_F^{\text{in}}}{B_{\text{in}}} \right\rceil. \tag{15}$$

The clock cycle counts for weight loading, computation, and output storing in one tile group can be expressed by

$$\begin{aligned} L_{\text{wgt}} &= T_M(1-R_R) \times \frac{T_N}{A_b} \times \left\lceil \frac{T_K(1-R_C)}{B_{\text{wgt}}} \right\rceil \\ L_{\text{cmpt}} &= \left\lceil \frac{T_F}{P_F} \right\rceil \times \left\lceil \frac{T_K}{P_K} \right\rceil \times \left\lceil \frac{G_M(1-R_R)}{P_M} \right\rceil \\ L_{\text{out}} &= \frac{T_M}{A_b} \times \left\lceil \frac{T_F}{B_{\text{out}}} \right\rceil. \end{aligned} \tag{16}$$

Double buffering enables the input loading, weight loading, and computation to be executed concurrently. The clock cycle count for this is

$$L_{\text{load\_cmpt}} = \max\left\{L_{\text{in}}, L_{\text{wgt}}, L_{\text{cmpt}}\right\}. \tag{17}$$

Each output channel accumulates results from $\lceil(N/T_N)\rceil$ weight tiles. With the output storing conducted in the meantime, the clock cycle count for this is

$$L_{\text{store}} = \max\left\{\left\lceil \frac{N}{T_N} \right\rceil \times L_{\text{load\_cmpt}} + L_{\text{cmpt}}, L_{\text{out}}\right\}. \tag{18}$$

And the total clock cycle count for the whole layer l is

$$L_l = \left\lceil \frac{D}{T_D} \right\rceil \times \left\lceil \frac{H}{T_H} \right\rceil \times \left\lceil \frac{W}{T_W} \right\rceil \times \left\lceil \frac{M}{T_M} \right\rceil \times L_{\text{store}} + L_{\text{out}}. \tag{19}$$

The data transfers and computations should be balanced in latency to guarantee that the accelerator design is compute-bounded rather than memory-bounded. Specifically, L_{in} and L_{wgt} should be hidden by L_{cmpt}, and L_{out} should be hidden by $L_{\text{load\_cmpt}}$, i.e.,

$$\begin{aligned} \max\{L_{\text{in}}, L_{\text{wgt}}\} &\leq L_{\text{cmpt}} \\ \left(\left\lceil \frac{N}{T_N} \right\rceil + 1\right) \times L_{\text{cmpt}} &\geq L_{\text{out}}. \end{aligned} \tag{20}$$

VI. Experimental Results

The proposed co-design framework of model compression and hardware acceleration is *unified for various 2-D and 3-D CNNs*, while the edge deployment of 3-D CNNs is more challenging than that of 2-D CNNs. In the experiments, this framework is evaluated on two representative 3-D CNN structures, C3D [43] and R(2+1)D [44]. The C3D network has a basic structure comprised of eight convolutional layers with kernel size of 3×3×3 and three fully connected layers. R(2+1)D consists of 37 convolutional layers in 5 blocks and 1 fully connected layers. The first block contains two layers, and the others are residual blocks, each containing eight layers with kernel sizes of 1×3×3 and 3×1×1 in turn. Besides, the final three blocks each have a residual layer. Fully connected layers contribute little to the total number of operations in the two models (0.263% for C3D, and <0.001% for R(2+1)D, so the compression focuses on the convolutional layers.

A. Experimental Setup for Training

Compression Settings for 3-D CNNs: The model compression procedure contains reweighted pruning and masked retraining for weights, and quantization for both weights and activations. The hyperparameter settings are the same for all the sparsity patterns, sparsity rates (pruning ratios), and quantization precisions. The original C3D and R(2+1)D models are pretrained on the Kinetics dataset [2] and transferred onto the UCF101 dataset [40]. The compression steps afterward are all performed on UCF101. The batch size is fixed to 32, and the video clip length adopts 16 frames. The initial learning rate (LR) is 5e−3 when training the dense models, and is reduced to 2e−4 in pruning and retraining for stability. Pruning is split into several stages, and LR is decreased gradually within each stage and increased when entering the next stage. Retraining adopts warmup and cosine decay for LR. Adjusting the initial LR to 5e−4, quantization is also split into stages, with LR halved from the current stage to the next one. It takes 150 epochs for pruning, 90 epochs for retraining, and 200 epochs for quantization. The global penalty factor λ in pruning is set to 1e−6. Additional training tricks [19] are utilized to improve the results, such as label smoothing in pruning and distillation in retraining. The whole compression process is performed with one NVIDIA A100 GPU with the PyTorch 1.13 framework, requiring memory up to 25 GB.

Pruning Ratio Configurations for 3-D CNNs: While the majority of previous weight pruning research places emphasis on reducing the number of parameters (weights generally) to mitigate the storage burden, we aim at reducing the number of operations to speed up the model inference, by setting larger pruning ratios for layers with more computations than others. Tables II and III provide the pruning ratio of each layer or block for C3D and R(2+1)D with KGR, KGC, and KGRC sparsity. Regarding the sparsity granularity, the kernel group size is $G_M = G_N = 8$ and $G_K = T_K = 9$ when the kernel size $K_l \geq 9$ (suitable for the C3D layers with kernel size of 3×3×3 and the R(2+1)D layers with kernel size of 1×3×3), or $G_K = 3$ when $K_l < 9$ (suitable for the R(2+1)D layers with kernel size of 3×1×1). All the layers with 3×1×1 kernels in R(2+1)D are not pruned.

TABLE II
C3D Pruning Configurations

Layer	Original Operations (G)	Pruning Ratio		
		KGR	KGC	KGRC
Conv1	2.1	1×	1×	1×
Conv2	22.2	2×	3×	6×
Conv3a	11.1	2×	1.5×	3×
Conv3b	22.2	2×	3×	6×
Conv4a	5.5	1×	1×	1×
Conv4b	11.1	2×	1.5×	3×
Conv5a+b	2.8	1×	1×	1×
Total	77.2	1.76×	1.93×	3.04×

TABLE III
R2+1-D Pruning Configurations

Layer (Block)	Original Operations (G)	Pruning Ratio		
		KGR	KGC	KGRC
Conv1	1.53	1×	1×	1×
Conv2_x	44.4	2×	2.00×	4.00×
Conv3_x	20.0	2×	2.00×	4.00×
Conv4_x	10.0	2×	1.33×	2.67×
Conv5_x	5.0	1×	1×	1×
Total	81.5	1.84×	1.75×	3.02×

B. Evaluation of Kernel Group Sparsity and Reweighted Pruning

Table IV provides the pruning results of C3D and R(2+1)D on UCF101 under multiple compression settings of sparsity patterns (None, KGR, KGC, and KGRC), pruning ratios, and quantization precisions (32-bit without quantization, 16, 8, and 4-bit). The accuracy of R(2+1)D is higher than that of C3D in the same setting because of the more complicated structure, including residual connections and batch normalization layers. There is hardly accuracy degradation from 16 to 8-bit quantization for both models in each sparsity setting, whilst 4-bit quantization leads to evident accuracy loss [4.15%∼6.68% for C3D and 2.55%∼6.07% for R(2+1)D]. This motivates us to concentrate on settings with 8-bit quantization for acceptable accuracy [controlling the loss within 4.5% for C3D and within 2.5% for R(2+1)D] and high acceleration ratios.

C. Evaluation of 3-D CNN Acceleration With Kernel Group Sparsity on FPGA

The accelerators are implemented on ZCU102 with the working frequency of 150 MHz through Xilinx Vivado 2020.1 with HLS. Table V shows the implementation parameters for tiling and parallelism, as well as the utilization of DSP, LUT, BRAM36, and FF resources of the accelerator design under different compression settings. For 16, 8, and 4-bit precisions, the accelerator is designed in two ways, one for dense models without pruning (denoted as Dense), and the other for pruned models with KGR, KGC or KGRC sparsity (denoted as Sparse). The Sparse designs are compatible with dense models, while their extra logic usage for kernel group sparsity slows the execution of dense models, so a Dense design is additionally implemented for more fair comparisons to demonstrate the efficiency of our Sparse implementations. With the kernel group size $G_M = G_N = 8$, the 16 and 8-bit Sparse designs can accommodate four kernel groups

TABLE IV
3-D CNN ACCURACY, INFERENCE LATENCY, AND POWER ON UCF101 DATASET UNDER DIFFERENT COMPRESSION SETTINGS

Compression Setting		C3D				R(2+1)D			
Sparsity	Precision	Pruning Ratio	Accuracy & Difference (%)	Latency (ms) & Acceleration Ratio	Power (W)	Pruning Ratio	Accuracy & Difference (%)	Latency (ms) & Acceleration Ratio	Power (W)
None	32-bit	1×	82.82	-	-	1×	94.51	-	-
	16-bit		81.55 (-1.27)	177.44	10.17		94.35 (-0.16)	201.38	10.18
	8-bit		81.39 (-1.43)	104.80 (1.69×)	7.06		94.35 (-0.16)	108.38 (1.86×)	7.09
	4-bit		78.67 (-4.15)	50.30 (3.53×)	8.66		91.96 (-2.55)	58.86 (3.42×)	8.65
KGR	32-bit	1.76×	81.84 (-0.98)	-	-	1.84×	93.21 (-1.30)	-	-
	16-bit		80.35 (-2.47)	118.45 (1.50×)	6.57		92.84 (-1.67)	133.77 (1.51×)	6.50
	8-bit		80.80 (-2.02)	64.04 (2.77×)	7.78		92.38 (-2.13)	74.87 (2.69×)	7.75
	4-bit		78.27 (-4.55)	32.24 (5.50×)	11.96		89.96 (-4.55)	41.49 (4.85×)	11.98
KGC	32-bit	1.93×	81.39 (-1.43)	-	-	1.75×	93.24 (-1.27)	-	-
	16-bit		79.31 (-3.51)	104.67 (1.70×)	6.65		93.29 (-1.22)	127.20 (1.58×)	6.73
	8-bit		79.65 (-3.17)	54.07 (3.28×)	7.85		93.24 (-1.27)	66.99 (3.01×)	7.92
	4-bit		77.34 (-5.48)	27.34 (6.49×)	12.05		90.23 (-4.28)	36.21 (5.56×)	12.13
KGRC	32-bit	3.04×	80.21 (-2.61)	-	-	3.02×	92.65 (-1.86)	-	-
	16-bit		78.48 (-4.34)	77.43 (2.29×)	6.48		91.85 (-2.66)	88.78 (2.27×)	6.52
	8-bit		78.40 (-4.42)	43.10 (4.12×)	7.75		92.09 (-2.42)	52.33 (3.85×)	7.73
	4-bit		76.14 (-6.68)	21.88 (8.10×)	11.92		88.44 (-6.07)	28.68 (7.02×)	11.88

TABLE V
3-D CNN ACCELERATOR IMPLEMENTATIONS ON ZCU102 FPGA UNDER DIFFERENT COMPRESSION SETTINGS

Precision	Design	Implementation Parameter Setting					Resource Utilization & Rate			
		T_M	P_M	$T_N (= P_N)$	P_F	P_K	DSP	LUT (k)	BRAM36	FF (k)
16-bit	Dense (Baseline)	56	56	4	8	1	1909 (76%)	127 (46%)	631.5 (69%)	63 (11%)
	Sparse	32	16	8	4	3	1620 (64%)	109 (40%)	428.5 (47%)	77 (14%)
8-bit	Dense	48	48	8	8	1	1651 (66%)	89 (32%)	343.5 (38%)	76 (14%)
	Sparse	32	16	8	8	3	1651 (66%)	131 (48%)	388.5 (43%)	105 (19%)
4-bit	Dense	32	32	8	8	3	1657 (66%)	126 (46%)	194.0 (21%)	83 (15%)
	Sparse	64	32	8	8	3	1655 (66%)	198 (72%)	214.5 (24%)	133 (24%)

TABLE VI
C3D IMPLEMENTATION COMPARISON BETWEEN THE PROPOSED FRAMEWORK, PRIOR WORK USING FPGA, AND MOBILE CPU/GPU EXECUTION (FOR OUR 8-BIT SPARSE IMPLEMENTATIONS, THE THROUGHPUT, ENERGY EFFICIENCY, AND DSP EFFICIENCY ARE PROVIDED IN TWO CASES, ONE CONSIDERING THE ACTUAL AMOUNT OF OPERATIONS AFTER PRUNING, AND THE OTHER IN PARENTHESES CONSIDERING THE ORIGINAL AMOUNT OF OPERATIONS IN THE MODEL)

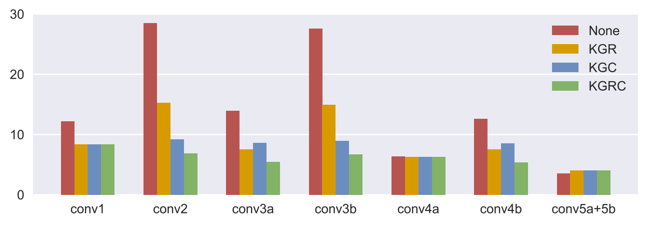
Platform (Implementation)	Jeston Orin NX			FPGA								
	CPU	GPU-PyTorch	GPU-TensorRT	ZC706		VUS440	VCU118	ZCU102	ZCU102 (Ours)			
				[12]	[10]	[38]	[39]	[42]	16-bit Dense	8-bit Sparse		
Technology	5 nm	8 nm		28 nm		20 nm	14 nm	16 nm	16 nm			
Frequency (MHz)	1190.4	918		176	200	200	200	150	150			
Precision	FP32	FP32	Int8	Fix16	BFP	Fix16	Fix16	Fix16	Fix16	Int8		
Sparsity	-	-	-	-	-	-	-	Block-Wise	-	KGR (1.76×)	KGC (1.93×)	KGRC (3.04×)
Power (W)	9.89	18.71	13.50	9.7	9.9	26	32	6.7	10.17	7.78	7.85	**7.75**
DSP Utilization	-	-	-	810	780	1536	5184	1215	1909	1651		
Throughput (GOPS)	110.1	1631.4	1308.7	142.0	161.6	784.7	5054.1	158.6	435.3	685.3 (1206.2)	**739.8** (1427.8)	589.5 (**1792.2**)
Energy Effi. (GOPS/W)	11.1	87.2	96.9	14.6	16.2	30.2	157.9	23.7	54.5	88.1 (155.0)	**94.2** (181.9)	76.1 (**231.3**)
DSP Effi. (GOPS/DSP)	-	-	-	0.175	0.207	0.511	0.975	0.131	0.228	0.415 (0.731)	**0.448** (0.865)	0.357 (**1.086**)
Latency (ms)	700.88	47.32	58.99	542.5	476.8	49.1	33.7*	487	177.44	64.04	54.07	**43.10**

FP32 – floating-point 32-bit Int8 – interger 8-bit Fix16 – fixed-point 16-bit BFP – block floating-point
* Pipeline latency of the computation engine

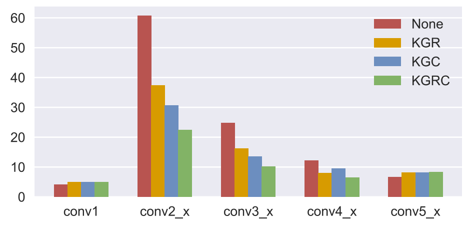
along the output dimension at a time, and the 4-bit Sparse design can hold eight kernel groups. In addition to the implementation parameters listed in Table V, other parameters include $T_D = 4, T_R = T_C = 14$, and $T_K = 9$ for all the designs.

The 16-bit Dense design is treated as the overall baseline. The DSP usage of the 16-bit Sparse design is lower, because the parallelism is set lower to prevent LUT over-utilization causing placement and routing failures. In 8-bit designs with the same DSP usage, the Sparse one utilizes 47% more LUTs than the Dense one. In 4-bit designs, the Sparse one consumes 52% more LUTs than the Dense one, as the tiling size T_M increases from 32 to 64, requiring more LUTs for input and output indexing.

The inference latency and power results of C3D and R(2+1)D with different precisions and sparsity settings are provided in Table IV. Compared with the 16-bit designs under a specific sparsity and pruning ratio, 8-bit quantization can accelerate C3D by 1.69×∼1.93× and R(2+1)D by 1.70×∼1.86×. The 4-bit quantization further improves the acceleration ratio to 3.53×∼3.83× for C3D and 3.42×∼3.51× for R(2+1)D. Compared with the Dense designs under a specific quantization precision, the KGR sparsity results in 1.50×∼1.64× acceleration for C3D with 1.76× pruning and 1.42×∼1.51× acceleration for R(2+1)D with 1.84× pruning. The acceleration ratio with KGC is 1.70×∼1.94× for C3D with 1.93× pruning, and 1.58×∼1.63× for R(2+1)D with 1.75× pruning. With KGRC,



(a)



(b)

Fig. 6. Latency of each layer (block) in C3D and R(2+1)D with 8-bit quantization and four sparsity settings (None, KGR, KGC, and KGRC). (a) C3D layer latency (ms). (b) R(2+1)D layer/block latency (ms).

TABLE VII
R(2+1)D Implementation Comparison Between the Proposed Framework and Mobile GPU/CPU Execution (For Our 8-bit Sparse Implementations, the Throughput, Energy Efficiency, and DSP Efficiency Are Provided in Two Cases, One Considering the Actual Amount of Operations After Pruning, and the Other in Parentheses Considering the Original Amount of Operations in the Model)

Platform (Implementation)	Jeston Orin NX			FPGA				
	CPU	GPU-PyTorch	GPU-TensorRT	ZCU102	ZCU102 (Ours)			
				[42]	16-bit Dense	8-bit Sparse		
Technology	5 nm	8 nm		16 nm	16 nm			
Frequency (MHz)	1190.4	918		150	150			
Precision	FP32	FP32	Int8	Fix16	Fix16	Int8		
Sparsity	-	-	-	Block-Wise	-	KGR (1.84×)	KGC (1.75×)	KGRC (3.02×)
Power (W)	9.11	17.02	15.00	6.7	10.18	7.75	7.92	**7.73**
Throughput (GOPS)	90.7	1114.3	1346.9	348.5	404.9	591.9 (1089.1)	**695.2** (1216.6)	516.0 (**1558.2**)
Energy Effi. (GOPS/W)	10.0	65.5	89.8	52.0	50.7	76.4 (140.5)	**87.8** (153.6)	66.8 (**201.6**)
Latency (ms)	898.58	73.14	60.51	234	201.38	74.87	66.99	**52.33**

the speedup ratio increases to 2.29×∼2.43× for C3D with 3.04× pruning, and 2.05×∼2.27× for R(2+1)D with 3.02× pruning. Generally, the power consumption of R(2+1)D is similar to that of C3D under the same compression setting, except for the KRC case where R(2+1)D dissipates slightly more power. Despite the higher peak throughput, the 4-bit Sparse design consumes 4.15 W∼4.23 W (53.2%∼54.6%) higher power than the 8-bit Sparse one, degrading its energy efficiency. This makes the 8-bit Sparse implementation more preferable. In summary, the KGRC sparsity with 8-bit quantization achieves a good balance between the speedup, model accuracy, and power consumption, leading to acceleration ratios of 4.12× for C3D and 3.85× for R(2+1)D compared with the baselines.

Fig. 6 illustrates the latency of each layer or block in C3D and R(2+1)D with 8-bit quantization and the four sparsity settings. As the pruning ratios are set higher for convolutional layers (blocks) with massive operations [e.g., conv2 and conv3b in C3D and conv2_x in R(2+1)D], the decrease in latency is correspondingly higher for the whole model. The latency of KGR, KGC, and KGRC designs might be higher than that of the Dense design for some layers (blocks) [e.g., conv5a+5b in C3D, as well as conv1 and conv5_x in R(2+1)D], because the Dense design is specially for dense models without sparsity. Despite this, the overall latency is shortened for both models, as shown above in Table IV.

Furthermore, Tables VI and VII compare the C3D and R(2+1)D performance on our FPGA accelerators with that on previous FPGA implementations as well as executions on the NVIDIA Jetson Orin NX 16 GB platform [1] with an 8-core Arm Cortex CPU and an Ampere GPU in the MAXN mode. The GPU execution based on CUDA is tested, respectively, with the PyTorch framework in 32-bit floating-point precision, and the TensorRT builder on one deep learning accelerator (DLA) compiler. For our 8-bit Sparse implementations, the throughput, energy efficiency, and DSP efficiency are provided in two cases, one considering the actual amount of operations after pruning, and the other in parentheses considering the original amount of operations in the model. It can be seen from the latter case that higher pruning ratios result in better performance. For the former case, the KGC executions perform better, as the KGC sparsity incurs less irregularity than KGR when running on hardware. There is a negative correlation between the irregularity and the actual throughput enhancement. An intuitive example is that the totally unstructured sparsity can hardly accelerate the model inference. The KGRC sparsity combines KGR and KGC, becoming more irregular than either of the two, although it is still hardware-friendly. It is worth emphasizing that the significance of KGRC lies in the higher pruning flexibility than KGR and KGC, meaning higher potential to maintain the model accuracy under similar pruning ratios. Generally, for an overall pruning ratio lower than 2×, KGR or KGC sparsity is preferred, while for an overall pruning ratio higher than 3×, KGRC is more suitable.

Compared with [10], our implementations can achieve 2.69×∼11.06× acceleration on C3D and 20.7%∼21.7% less power consumption, indicating 4.70×∼5.81× higher energy efficiency. For fair comparison with implementations based on the Winograd algorithm in [38] and [39], the original amount of model operations should be considered. The equivalent efficiency of the accelerator in [39], realized on a large-scale FPGA (VCU118 with 6840 DSPs), reaches 157.9 GOPS/W and 0.975 GOPS/DSP. Our 8-bit KGRC design attains 1.46× higher energy efficiency, and comparable DSP efficiency (1.11×). Executed on the same platform, all of our implementations acquire higher performance than those in [42]. In detail, our C3D designs acquire 3.72×∼4.66× higher throughput, and this ratio is 1.48×∼1.99× for our R(2+1)D designs.

In comparison with the Orin NX CPU executions, the energy efficiency of our implementations is 6.86×∼8.49× higher for C3D and 6.68×∼8.78× higher for R(2+1)D. As for the Orin NX GPU executions, the TensorRT framework has a better optimization effect on power consumption and thus on energy efficiency. The energy efficiency of our 8-bit Sparse implementation with the KGC sparsity is higher than that of the PyTorch-based execution, and competitive with that of the TensorRT-based execution for both C3D and R(2+1)D. Finally, our KGRC implementation is 6.96×∼9.81× more energy efficient than the custom-designed E3DNet implementation [11] with 9.60 GOPS/W.

VII. Conclusion

This work designs and implements a 3-D CNN acceleration framework through algorithm-hardware co-design, targeting edge FPGAs with constrained resources. The proposed KGRC sparsity is fine-grained to achieve high pruning ratios with negligible accuracy loss (as nonstructured pruned models do), and balanced across kernel groups to achieve high computation parallelism on hardware accelerators (as structured pruned models do). The corresponding reweighted pruning algorithm is performed with quantization in different precisions for the tradeoff between the model accuracy and inference speed. An FPGA-based accelerator with four modes is then designed to support the kernel group sparsity in multiple dimensions. This co-design framework is tested on the C3D and R(2+1)D networks for action recognition on the ZCU102 FPGA. The experimental results indicate that the accelerator implementation with the KGRC sparsity and 8-bit quantization can achieve speedup of 4.12× for C3D and 3.85× for R(2+1)D, compared with the 16-bit baseline designs supporting only dense models.

References

[1] "Data sheet NVIDIA Jetson Orin NX series," Data Sheet DS-10712, Nvidia, Santa Clara, CA, USA, Apr. 2022. [Online]. Available: https://developer.nvidia.com/downloads/jetson-orin-nx-series-data-sheet

[2] J. Carreira and A. Zisserman, "Quo vadis, action recognition? A new model and the kinetics dataset," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2017, pp. 6299–6308.

[3] X. Chang, H. Pan, W. Lin, and H. Gao, "A mixed-pruning based framework for embedded convolutional neural network acceleration," *IEEE Trans. Circuits Syst. I, Reg. Papers (TCASI)*, vol. 68, no. 4, pp. 1706–1715, Apr. 2021.

[4] H. Chen et al., "Frequency domain compact 3d convolutional neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2020, pp. 1641–1650.

[5] Y. Chen, Y. Kalantidis, J. Li, S. Yan, and J. Feng, "Multi-fiber networks for video recognition," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 352–367.

[6] G. Cheng et al., "μL2Q: An ultra-low loss Quantization method for DNN," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, 2019, pp. 1–8.

[7] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," 2018, *arXiv:1805.06085*.

[8] X. Dai, H. Yin, and N. K. Jha, "NeST: A neural network synthesis tool based on a grow-and-prune paradigm," 2017, *arXiv:1711.02017*.

[9] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, "Learned step size quantization," 2019, *arXiv:1902.08153*.

[10] H. Fan et al., "Reconfigurable acceleration of 3D-CNNs for human action recognition with block floating-point representation," in *Proc. 28th Int. Conf. Field-Program. Logic Appl. (FPL)*, 2018, pp. 287–2877.

[11] H. Fan et al., "F-E3D: FPGA-based acceleration of an efficient 3D convolutional neural network for human action recognition," in *Proc. IEEE 30th Int. Conf. Appl. Specif. Syst., Archit. Process. (ASAP)*, 2019, pp. 1–8.

[12] H. Fan, X. Niu, Q. Liu, and W. Luk, "F-c3d: FPGA-based 3-dimensional convolutional neural network," in *Proc. 27th Int. Conf. Field-Program. Logic Appl. (FPL)*, 2017, pp. 1–4.

[13] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitect. (MICRO)*, 2019, pp. 151–165.

[14] R. Gong et al., "Differentiable soft quantization: Bridging full-precision and low-bit neural networks," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, 2019, pp. 4852–4861.

[15] K. Guo et al., "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018.

[16] P. Guo, H. Ma, R. Chen, P. Li, S. Xie, and D. Wang, "FBNA: A fully Binarized neural network accelerator," in *Proc. 28th Int. Conf. Field-Program. Logic Appl. (FPL)*, 2018, pp. 51–513.

[17] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient DNNs," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2016, pp. 1379–1387.

[18] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2015, pp. 1135–1143.

[19] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li, "Bag of tricks for image classification with convolutional neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2019, pp. 558–567.

[20] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," in *Proc. Int. Joint Conf. Artif. Intell. (IJCAI)*, 2018, pp. 2234–2240.

[21] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, 2017, pp. 1398–1406.

[22] H. Jang, J. Kim, J.-E. Jo, J. Lee, and J. Kim, "MnnFast: A fast and scalable system architecture for memory-augmented neural networks," in *Proc. 46th Int. Symp. Comput. Architect. (ISCA)*, 2019, pp. 250–263.

[23] S. Jung et al., "Learning to quantize deep networks by optimizing quantization intervals with task loss," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2019, pp. 4350–4359.

[24] C. Leng, Z. Dou, H. Li, S. Zhu, and R. Jin, "Extremely low bit neural network: Squeeze the last bit out with ADMM," in *Proc. 32nd AAAI Conf. Artif. Intell. (AAAI)*, 2018, pp. 3466–3473.

[25] T. Li, B. Wu, Y. Yang, Y. Fan, Y. Zhang, and W. Liu, "Compressing convolutional neural networks via factorized convolutional filters," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2019, pp. 3977–3986.

[26] Y. Liang, L. Lu, and J. Xie, "OMNI: A framework for integrating hardware and software optimizations for sparse CNNs," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 40, no. 8, pp. 1648–1661, Aug. 2021.

[27] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, "An efficient hardware accelerator for sparse convolutional neural networks on FPGAs," in *Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2019, pp. 17–25.

[28] C. Luo, W. Cao, L. Wang, and P. H. Leong, "RNA: An accurate residual network accelerator for quantized and reconstructed deep neural networks," *IEICE Trans. Inf. Syst.*, vol. 102, no. 5, pp. 1037–1045, 2019.

[29] X. Ma et al., "PCONV: The missing but desirable sparsity in DNN weight pruning for real-time execution on mobile devices," in *Proc. 34th AAAI Conf. Artif. Intell. (AAAI)*, 2020, pp. 1–8.

[30] X. Ma et al., "BLK-REW: A unified block-based DNN pruning framework using reweighted regularization method," 2020, *arXiv:2001.08357*.

[31] H. Nakahara, T. Fujii, and S. Sato, "A fully connected layer elimination for a Binarized convolutional neural network on an FPGA," in *Proc. 27th Int. Conf. Field-Program. Logic Appl. (FPL)*, 2017, pp. 1–4.

[32] H. Nakahara, H. Yonekawa, T. Sasao, H. Iwamoto, and M. Motomura, "A memory-based realization of a Binarized deep convolutional neural network," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, 2016, pp. 277–280.

[33] W. Niu et al., "RT3D: Achieving real-time execution of 3D convolutional neural networks on mobile devices," in *Proc. AAAI Conf. Artif. Intell. (AAAI)*, 2021, pp. 9179–9187.

[34] Y. Niu, R. Kannan, A. Srivastava, and V. Prasanna, "Reuse kernels or activations? A flexible dataflow for low-latency spectral CNN acceleration," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2020, pp. 266–276.

[35] C. Park et al., "Balanced column-wise block pruning for maximizing GPU parallelism," in *Proc. AAAI Conf. Artif. Intell. (AAAI)*, 2023, pp. 9398–9407.

[36] Z. Qiu, T. Yao, and T. Mei, "Learning spatio-temporal representation with pseudo-3d residual networks," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, 2017, pp. 5533–5541.

[37] A. Ren et al., "ADMM-NN: An algorithm-hardware co-design framework of DNNs using alternating direction methods of multipliers," in *Proc. 24th Int. Conf. Architect. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2019, pp. 925–938.

[38] J. Shen, Y. Huang, Z. Wang, Y. Qiao, M. Wen, and C. Zhang, "Towards a uniform template-based architecture for accelerating 2D and 3D CNNs on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2018, pp. 97–106.

[39] J. Shen, Y. Huang, M. Wen, and C. Zhang, "Towards an efficient deep pipelined template-based architecture for accelerating the entire 2-D and 3-D CNNs on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 7, pp. 1442–1455, Jul. 2020.

[40] K. Soomro, A. R. Zamir, and M. Shah, "UCF101: A dataset of 101 human actions classes from videos in the wild," 2012, *arXiv:1212.0402*.

[41] M. Sun et al., "FILM-QNN: Efficient FPGA acceleration of deep neural networks with intra-layer, mixed-precision quantization," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2022, pp. 134–145.

[42] M. Sun, P. Zhao, M. Gungor, M. Pedram, M. Leeser, and X. Lin, "3D CNN acceleration on FPGA using hardware-aware pruning," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, 2020, pp. 1–6.

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

[44] D. Tran, H. Wang, L. Torresani, J. Ray, Y. LeCun, and M. Paluri, "A closer look at spatiotemporal convolutions for action recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2018, pp. 6450–6459.

[45] M. Van Keirsbilck, A. Keller, and X. Yang, "Rethinking full connectivity in recurrent neural networks," 2019, *arXiv:1905.12340*.

[46] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA," in *Proc. 28th Int. Conf. Field-Program. Logic Appl. (FPL)*, 2018, pp. 163–1636.

[47] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2016, pp. 2074–2082.

[48] S. Xie, C. Sun, J. Huang, Z. Tu, and K. Murphy, "Rethinking spatiotemporal feature learning: Speed-accuracy trade-offs in video classification," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 305–321.

[49] C. Yang, Y. Meng, K. Huo, J. Xi, and K. Mei, "A sparse CNN accelerator for eliminating redundant computations in intra-and inter-convolutional/pooling layers," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 30, no. 12, pp. 1902–1915, Dec. 2022.

[50] C. Zhang, G. Sun, Z. Fang, P. Zhou, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. ACM Turing Award Celeb. Conf.*, 2023, pp. 47–48.

[51] D. Zhang, J. Yang, D. Ye, and G. Hua, "LQ-Nets: Learned quantization for highly accurate and compact deep neural networks," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 365–382.

[52] T. Zhang et al., "A systematic DNN weight pruning framework using alternating direction method of multipliers," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 184–199.

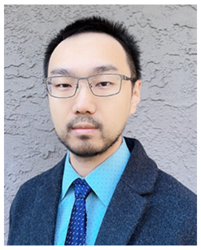
[53] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless CNNs with low-precision weights," 2017, *arXiv:1702.03044*.

[54] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," 2016, *arXiv:1606.06160*.

[55] C. Zhu, K. Huang, S. Yang, Z. Zhu, H. Zhang, and H. Shen, "An efficient hardware accelerator for structured sparse convolutional neural networks on FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 9, pp. 1953–1965, Sep. 2020.

Mengshu Sun received the M.S. degree in electrical engineering from the University of Southern California, Los Angeles, CA, USA, in 2016, and the Ph.D. degree in computer engineering from Northeastern University, Boston, MA, USA, in 2022.

She is currently an Assistant Professor with Beijing Institute of Artificial Intelligence, Beijing University of Technology, Beijing, China. Her research interests include deep learning acceleration via algorithm-hardware co-design, edge computing, and hardware security.



Kaidi Xu (Member, IEEE) received the B.S. degree from Sichuan University, Chengdu, China, in 2015, the M.S. degree from the University of Florida, Gainesville, FL, USA, in 2017, and the Ph.D. degree from Northeastern University, Boston, MA, USA, in 2021.

He is an Assistant Professor with the Department of Computer Science, Drexel University, Philadelphia, PA, USA. His research interest is trustworthy AI, including real-world attacks, formal neural network verification, and certified defenses.

Dr. Xu is the winner of the International Verification of Neural Networks Competition (VNN-COMP 2021–2023) for three consecutive years.

Xue Lin (Member, IEEE) received the bachelor's degree in microelectronics from Tsinghua University, Beijing, China, in 2009, and the Ph.D. degree from the Department of Electrical Engineering, University of Southern California, Los Angeles, CA, USA, in 2016.

She is an Associate Professor with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA. Her research interests include deep learning, machine learning and computing in cyber–physical systems, high-performance and mobile computing systems, and VLSI.

Dr. Lin work got multiple research awards, including the Third Place Winner in Stage II of the National 2022 Inclusive Design Challenge from U.S. Department of Transportation, the Top Highest Score Award in VNN-COMP'21 at the 33rd ICCAV, the Best Paper Award at HAET Workshop at ICLR 2021, the First Place at ISLPED 2020 Design Contest, and the Best Technical Poster Award at NDSS 2020.

Yongli Hu (Member, IEEE) received the Ph.D. degree from the Beijing University of Technology, Beijing, China, in 2005.

He is currently a Professor with the Faculty of Information Technology, Beijing University of Technology. He is also a Researcher with the Beijing Key Laboratory of Multimedia and Intelligent Software Technology and the Beijing Institute of Artificial Intelligence, Beijing. His research interests include computer vision, pattern recognition, machine learning, intelligent transportation systems, and multimedia technology.

Baocai Yin (Member, IEEE) received the Ph.D. degree from the Dalian University of Technology, Dalian, China, in 1993.

He is currently a Professor with the Faculty of Information Technology, Beijing University of Technology, Beijing, China. He is also a Researcher with the Beijing Key Laboratory of Multimedia and Intelligent Software Technology and the Beijing Institute of Artificial Intelligence, Beijing. His research interests include multimedia, multifunctional perception, virtual reality, and computer graphics.

Dr. Yin is a member of the China Computer Federation.