ESCALATE: Boosting the Efficiency of Sparse CNN Accelerator with Kernel Decomposition

Shiyu Li
Duke University
Durham, NC, USA
shiyu.li@duke.edu

Edward Hanson
Duke University
Durham, NC, USA
edward.t.hanson@duke.edu

Xuehai Qian
University of Southern California
Los Angeles, CA, USA
xuehai.qian@usc.edu

Hai "Helen" Li
Duke University
Durham, NC, USA
hai.li@duke.edu

Yiran Chen
Duke University
Durham, NC, USA
yiran.chen@duke.edu

ABSTRACT

The ever-growing parameter size and computation cost of Convolutional Neural Network (CNN) models hinder their deployment onto resource-constrained platforms. Network pruning techniques are proposed to remove the redundancy in CNN parameters and produce a sparse model. Sparse-aware accelerators are also proposed to reduce the computation cost and memory bandwidth requirements of inference by leveraging the model sparsity. The irregularity of sparse patterns, however, limits the efficiency of those designs. Researchers proposed to address this issue by creating a regular sparsity pattern through hardware-aware pruning algorithms. However, the pruning rate of these solutions is largely limited by the enforced sparsity patterns. This limitation motivates us to explore other compression methods beyond pruning. With two decoupled computation stages, we found that kernel decomposition could potentially take the processing of the sparse pattern off from the critical path of inference and achieve a high compression ratio without enforcing the sparse patterns. To exploit these advantages, we propose *ESCALATE* , an algorithm-hardware co-design approach based on kernel decomposition. At algorithm level, *ESCALATE* reorganizes the two computation stages of the decomposed convolution to enable a stream processing of the intermediate feature map. We proposed a hybrid quantization to exploit the different reuse frequency of each part of the decomposed weight. At architecture level, *ESCALATE* proposes a novel 'Basis-First' dataflow and its corresponding microarchitecture design to maximize the benefits brought by the decomposed convolution.

We evaluate *ESCALATE* with four representative CNN models on both CIFAR-10 and ImageNet datasets and compare it against previous sparse accelerators and pruning algorithms. Results show that *ESCALATE* can achieve up to 325× and 11× compression ratio for models on CIFAR-10 and ImageNet, respectively. Comparing with previous dense and sparse accelerators, *ESCALATE* accelerator averagely boosts the energy efficiency by 8.3× and 3.77×, and reduces the latency by 17.9× and 2.16×, respectively.

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480043

CCS CONCEPTS

• **Computer systems organization → Neural networks**; *Special purpose systems*; • **Computing methodologies → Neural networks**.

KEYWORDS

Convolutional Neural Networks, Kernel Decomposition, Neural Network Compression, Sparse Accelerators

ACM Reference Format:
Shiyu Li, Edward Hanson, Xuehai Qian, Hai "Helen" Li, and Yiran Chen. 2021. ESCALATE: Boosting the Efficiency of Sparse CNN Accelerator with Kernel Decomposition. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480043

1 INTRODUCTION

Convolutional Neural Networks (CNNs) have been widely used in a vast range of applications, such as computer vision, robotics, and medical science. However, most CNN models are both computation- and memory-intensive. Thus, the execution of CNN models requires high computation power and memory bandwidth, making it difficult to achieve high throughput and energy efficiency on general-purpose platforms like CPUs or GPUs. These concerns lead to the proliferation of domain-specific accelerators for CNNs. Various custom architectures [5–8] have been proposed to capture the parallelism and data reuse opportunities for boosting the inference efficiency.

Previous works [13] successfully identified the redundancy in CNN parameters: A large portion of unimportant weights in the convolutional layer can be removed to produce sparse weights with minor or no impacts on model accuracy. Moreover, the widely adopted ReLU activation function filters out all non-positive values in the output feature maps, leading to sparse activations. Multiplications involving zero-valued (or sparse) activations or weights produce zero products. By avoiding storing, transferring, and computing these zeros, the computation and bandwidth requirements

of inference can be reduced. The potential benefit triggers the emergence of sparse-aware accelerators. Cambricon-X [40], for example, skips all computations related to zero weights, while Cnvlutin [2] eliminates all the zero activations. SCNN [30] captures the sparsity in both weights and activations with the Cartesian product. SparTen [12] performs efficient inner-join between sparse vectors to address the inefficiency under certain layer shapes. ExTensor [15] performs efficient intersection operations to remove zero-involved computations. However, the irregular sparse pattern of convolutional weight and activations prevents these hardware solutions from fully utilizing the sparsity.

Researchers propose hardware-aware optimizations of the existing pruning algorithms to address the irregularity. Cambricon-S [43] uses coarse-grained pruning to alleviate the irregularity in sparsity patterns. ADMM-NN [33] jointly performs pruning and quantization while adjusting the layer-wise sparsity ratios to achieve an optimal tradeoff between accuracy and runtime speedup. PatDNN [28] and ADMM-NN-S [26] further enforce specific sparsity patterns during pruning to enable more efficient processing of the regular sparse models. Due to the degraded flexibility of pruning, these optimizations limit the overall compression rate.

The limitations of individually optimizing hardware and algorithm motivate us to seek an integrated solution. In particular, we hope to explore an alternative computation formulation of convolution operations to take the processing of irregular sparsity patterns off from the critical path of inference. Thus, the hardware that supports this formulation can eliminate the negative impact of irregularity. We may also achieve a high compression rate since no sparsity pattern is enforced during pruning.

Our recent work, PENNI [25], proposes a new CNN compression method based on kernel decomposition. It decomposes a weight tensor into two parts – a small number of basis kernels and a coefficient tensor that determines the linear combination of the basis kernels. With the decomposition, the forward pass can be performed in two decoupled stages: basis convolution and weighted accumulation. Sparsity only appears in the weighted accumulation stage, the execution of which overlaps with the basis convolution stage. Thus, compared with previous weight pruning methods, [25] can potentially remove the processing of irregular sparsity pattern from the critical path and obtains *a more hardware-friendly* computation pattern. Moreover, compared with the hardware-aware pruning methods, such kernel decomposition does not enforce the structure constraints and can potentially achieve a higher pruning rate.

It is natural to design corresponding CNN accelerators for the above kernel decomposition method to maximize its advantages. However, directly applying the kernel decomposition incurs some challenges to the hardware design: First, while the computational cost is reduced with a small number of basis kernels, the algorithm creates a large number of intermediate feature maps with inflated channels. These intermediate feature maps become a new computation bottleneck. Moreover, the existing dataflows in mainstream CNN accelerators do not support the two-stage computation of the decomposed convolution. The distinct parameter reuse frequencies of the two decomposed stages also impact the performance differently. Finally, the frequently reused basis kernels require a higher precision, while the unique coefficient of each input-output channel can bear a lower precision. The different precisions must be handled by different hardware configurations.

To tackle these challenges, we propose *ESCALATE*, an algorithm-hardware co-design framework based on kernel decomposition. At algorithm level, we reorganize the two computation stages of the decomposed convolution based on distributive property to enable streaming processing of the large intermediate feature maps and eliminate the computational bottleneck. Based on the observation that the coefficients are unique to each pair of input-output channels and the basis kernels are reused by all output channels, we propose a hybrid quantization scheme to maximize the benefit on both accuracy and compression ratio. At architecture level, we propose a novel *Basis-First* dataflow to increase the parallelism of inference and exploit the reuse opportunities of the decomposed convolution. *ESCALATE* accelerator is equipped with an efficient sparse skipping scheme, namely, 'Dilution-Concentration', based on the compact weight structure created by the decomposed convolution.

We evaluate *ESCALATE* framework with four representative CNN models on both CIFAR-10 [22] and ImageNet [10] datasets. Results show that *ESCALATE* algorithm achieves 11-325× compression ratio for CIFAR-10 models, and 9-11× for ImageNet models. On average, *ESCALATE* accelerator boosts the energy efficiency by 8.3× and 3.77×, and reduce the latency by 17.9× and 2.13× compared to previous dense and sparse accelerator designs, respectively.

The remainder of our paper is organized as follows. Section 2 presents the background and the motivation of our work; Section 3 introduces the proposed *ESCALATE* algorithm; Section 4 presents the *ESCALATE* accelerator design; Section 5 provides the evaluation results of both algorithm and accelerator; Section 6 provides discussions on design trade-offs; Section 7 summarizes related works, and Section 8 concludes this work.

2 BACKGROUND AND MOTIVATION

2.1 Convolution Operation

We present essential notations and define the terminologies used in our discussion. We refer to each individual element of the input feature maps (IFM) as the *activation* and the convolution parameters as *weight*. We use *kernel* to denote a 2D convolution kernel corresponding to one input-output channel pair and use *filter* to denote a 3D filter corresponding to one output channel. The computation of a regular convolution operation involves 7 dimensions, including input batch (N), input channels (C), output channels (or filter, K), filter row (R), filter column (S), input row (X), and input column (Y). The index of the output row/column can be deduced from the input and filter indices. For inference tasks, we focus on the performance of a single input sample, thus we ignore N in our discussion. We use uppercase letters to denote the size of the dimension and use lowercase letters to denote the index. The computation of one element in the k-th output channel can be represented as,

$$OFM_k^{(x+\lfloor r/2 \rfloor, y+\lfloor s/2 \rfloor)} = \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} W^{(k,c,r,s)} IFM^{(c,x+r,y+s)}, \tag{1}$$

where $W \in \mathbb{R}^{K\times C\times R\times S}$ is the weight tensor, $IFM \in \mathbb{R}^{C\times X\times Y}$ and $OFM \in \mathbb{R}^{K\times X'\times Y'}$ are the input and output feature maps, respectively. For simplicity, we only show the unit stride situation and assume the input feature maps are padded.

2.2 Processing Sparse CNN

The redundancy in CNN models comes from two sources: activations and weights. The widely adopted ReLU activation function filters out all non-positive values in the output feature maps, leading to sparse activations. Pruning unimportant weight values produces sparse weights. Typically, the sparsity introduces on average 2-5× model size reduction and 4-20× computational cost reduction. However, leveraging sparsity poses new challenges to the accelerator design. Specifically, the accelerator needs to identify non-zero pairs of weights and activations from the compressed data and dispatch those pairs correctly to processing elements (PEs). Current sparse-aware accelerators [2, 12, 30, 40] propose various mechanisms to efficiently process sparse CNN. Due to the irregular distribution of the non-zero elements, all these mechanisms incur considerable hardware and energy overheads that significantly offset the benefits brought by sparsity [26]. This challenge can be mitigated by algorithm-hardware co-designed approaches which can perform pruning by taking hardware efficiency into consideration. For example, Cambricon-S [43] proposes a coarse-grained pruning method to alleviate the irregularity. ADMM-NN [33] introduces a joint weight pruning and quantization framework to select a proper layer-wise compression ratio and maximize the overall hardware efficiency. PatDNN [28] and ADMM-NN-S [26] further enforce specific sparsity patterns during pruning to enable more efficient processing of the regular sparse models.

2.3 Kernel Decomposition in CNN

Kernel decomposition utilizes the low-rank assumption of weight matrices to compress the model size for reducing the computational complexity. Decomposition can be performed at different levels of the weight structure. Previous works utilize the matrix decomposition [42] or tensor decomposition [19, 24] to decompose each convolutional filter into two or more lower rank structures. PENNI [25] proposes to conduct the decomposition at the kernel level, i.e., projecting each 2D kernel into a subspace. With the same notations, the 4-D weight tensor can be reshaped into a matrix W' in the shape of $KC \times RS$. The decomposition can be formulated as $W = C_e B$ where $C_e \in \mathbb{R}^{KC\times M}$, $B \in \mathbb{R}^{M\times RS}$ and $M < RS$. The two factor matrices are obtained using singular value decomposition (SVD). Further retraining steps may be required to recover the accuracy of the model. Here, each row of B can be reshaped into $R\times S$ and can be seen as a kernel. Thus, the M basis kernels are shared across the entire layer and the original kernels can be approximated by the linear combinations of the basis kernels. This decomposition creates two imbalanced parameters: a small set of kernels shared across all convolution operations and a large coefficient matrix. Apart from reducing the computation and parameters, the decomposition also makes it much easier to sparsify the coefficient matrix compared with directly pruning the original weights.

2.4 Motivation

This work is motivated by the drawbacks of the existing non-structured and structured pruning. Non-structured (or element-wise) pruning can achieve the highest compression ratio on CNNs among different types of compression methods. However, the irregularity of the sparsity pattern leads to a huge gap between the algorithm-level computation reduction and the actual hardware speedups. For example, Eyeriss v2 [8] only achieve 1.2× speedup on sparse MobileNet. ADMM-NN [33] demonstrates only 3.9× speedup for the pruned convolutional layers in AlexNet with a 25.5× pruning ratio. In essence, non-structured pruning leads to inefficient and expensive hardware implementations which largely offset the benefits of the large compression ratio. On the other side, structured pruning maintains the regularity of weights so that the pruning ratio can be almost fully converted to the speedup. However, the structured constraint limits its pruning ratio. For example, structured pruning on ResNet50 only achieves 2.64× pruning ratio [11] while non-structured pruning could reach 17.4× [33]. In summary, none of the two types of pruning method sufficiently translate the elimination of redundancy in CNNs to the higher inference performance.

To tackle the challenge, we decide to explore the potential of the alternative computation formulation of convolution operation with kernel decomposition. As discussed in Section 2.3, kernel decomposition creates two imbalance parts to approximate the original weight tensor, which naturally form two computation stages. The stage involving coefficient matrix only requires scalar matrix product and reduction. It opens the opportunity of utilizing the sparsity in coefficients in a structured way. Compared to the non-structured pruning, kernel decomposition alleviates the irregularity in the sparsity-aware computation and provides better hardware efficiency. Compared to the structured pruning, without the need to enforce sparsity pattern when pruning the coefficient matrix, we can achieve a higher pruning ratio. Motivated by the potential of obtaining both a high pruning ratio and hardware efficiency, we build an algorithm-hardware co-design framework based on kernel decomposition.

3 ESCALATE ALGORITHM

In this section, we first analyze the computational bottleneck in kernel decomposition algorithm. Then, we propose a reformulated *ESCALATE* algorithm to alleviate the bottleneck. Based on kernel decomposition, *ESCALATE* utilizes the distributive property of the convolution operation to create an efficient computation flow. Then, we present the hybrid quantization which selects the quantization precision based on the reuse frequency of each part of the weights. Moreover, we show that *ESCALATE* can expand the application of the decomposition technique to depth-wise separable convolution (DSC) and unify the computation of both regular convolution and DSC.

3.1 Computation Reorganization

PENNI[25] introduced a kernel level decomposition of the weights. As we discussed in Section 2.3, this decomposition scheme creates two imbalanced weight components that respectively correspond to two computation stages—shared kernel convolution and weighted

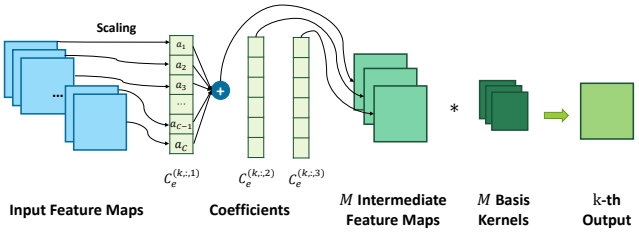

Figure 1: The computation process of one output feature map of the reorganized decomposed convolution. For simplicity, we assume $M = 3$ here.

accumulation. Using the notations defined in Section 2.1 and Section 2.3, if we reshape the coefficients into $K \times C \times M$, the k-th output channel can be computed by

$$OFM_k = \sum_{c=0}^{C-1} \sum_{m=0}^{M-1} C_e^{(k,c,m)} IFM_c * B_m, \tag{2}$$

where IFM_c is the c-th input feature map, B_m is the m-th basis kernel, and $*$ denotes the 2D convolution operation. The shared kernel convolution (i.e., the inner summation) is conducted in a depthwise fashion (i.e., there is no reduction across input channels) and generates M times more feature maps than the input. The weighted accumulation (i.e., the outer summation) of these feature maps becomes the computational bottleneck. While the decomposition can achieve a high compression rate on CNN models, it is difficult to translate the compression rate into the actual speedup. From Equation (2), we can tell that each pixel of the produced feature maps is only reused across output channels. We either need to frequently read and write the output channels to maximize the input reuse, or we have to build a large buffer to hold all intermediate feature maps. Both designs are energy-consuming. In addition, although the coefficients are highly sparse, the irregularity of the sparsity pattern makes it difficult to skip the zeros without degrading the parallelism of the computation.

Based on the above analysis, more efficient computation can be achieve with: (1) reducing of the total number of input feature maps; (2) reducing of the number of input feature maps related to each output channel; and (3) increasing the reuse possibilities of each input feature map in the weighted accumulation stage. These desirable goals can be achieve with an simple observation: since the convolution operator follows the distributive property, the two stages of the computation are interchangeable. Specifically, we can exchange the order of the summations in Equation (2). With such reorganization, we first compute the weighted accumulation of the input feature maps, then conduct the convolution on the accumulated feature maps. The same output channel can be computed by

$$OFM_k = \sum_{m=0}^{M-1} \left(\sum_{c=0}^{C-1} C_e^{(k,c,m)} IFM_c \right) * B_m. \tag{3}$$

The reorganized convolution is illustrated in Figure 1. With the new computation order, we only need to compute the weighted accumulation of C input feature maps. Each input feature map can be reused for CM times and each basis kernel can be reused for K times. In general, the number of output channels is larger than or equal to the number of the input channels. Hence, this reorganization can explore more reuse opportunities and reduce the buffer size and data movements. We note that converting the convolution from the depthwise-like convolution into the normal convolution with M channels also improves the parallelism and data reuse opportunities.

3.2 Hybrid Quantization

In kernel decomposition, the weight sparsity mainly exists in the coefficients since they constitute the main part of the parameters. We observe that the coefficients are unique to each pair of input-output channel, while the basis kernels are reused by all output channels. Selecting the same quantization precision for coefficients and basis kernels will waste the opportunity—a high precision is redundant for coefficients, while a low precision for basis kernels will severely affect the accuracy. The results are discussed in Section 5.1. Based on this observation, we propose a hybrid quantization scheme to maximize the benefit on both accuracy and compression rate. The basic idea is to use high precision for the frequently reused basis kernel while using low precision for the coefficients. We choose to quantize the basis kernels to 8 bits and the coefficients to ternary values. Directly quantizing the whole coefficients into ternary value will cause a severe accuracy drop. We observe that, for a given output channel k, only the $C_e^{(k,:,:)}$ slice is involved in the computation. Thus, we apply the *filter-wise* quantization and allow different positive and negative scaling factors for each slice. The output feature maps can be re-quantized to match the range of each output channel.

To obtain the ternarized value of the coefficients, we adopt a quantization-aware training scheme. We use a similar method as described in [44]. Specifically, we store the full precision coefficients during the quantization. During the forward pass, we select a threshold for each coefficient slice corresponding to one output channel. The quantized value of the k-th slice is obtained by

$$\tilde{C_e}^{(k,:,:)} = \begin{cases} w_k^{pos}, & C_e^{(k,:,:)} > t \cdot \max(|C_e^{(k,:,:)}|) \\ 0, & |C_e^{(k,:,:)}| \le t \cdot \max(|C_e^{(k,:,:)}|) \\ -w_k^{neg}, & C_e^{(k,:,:)} < -t \cdot \max(|C_e^{(k,:,:)}|) \end{cases}, \tag{4}$$

where $\max|(C_e^{(k,:,:)})|$ is the maximum magnitude of the k-th coefficient slice, t is a hyper-parameter controlling the threshold, and w_k^{pos} and w_k^{neg} are the scaling factors for positive and negative values, respectively. We use different scaling factors for *each slice* of the coefficient matrix C_e, i.e., each accumulation operation, and obtain the scaling factor through training. We use the gradient of quantized coefficients to update the full precision parameters as well as the scaling factors. To simplify the hardware design, we divide the negative scaling factor by the positive one and quantize the quotient into 2 bits. During inference, we can attach the coefficient as a sign bit to each activation, and shift the negative one by the quotient. With these optimizations, we completely remove the multiplication in the first stage and enable arbitrary reordering of the activations without worrying about the relative order of the activations and weights. We claim those benefits in the microarchitecture design developed in section 4. Since the first layer of CNNs

usually only has a very small number of channels (e.g., 3 channels for color image-related tasks), quantizing the first layer will incur a severe loss of information. Moreover, since k is normally larger than the number of the input channels in the first layer, applying decomposition will not bring any benefit to improve the computational efficiency. Thus, we do not apply compression to the first layer.

3.3 Decomposing the Compact Model

Depthwise separable convolution (DSC) splits the normal convolution into two steps: depthwise convolution and pointwise convolution. The depthwise convolution (DW) assigns an exclusive kernel to each input channel and eliminates the cross-channel interactions to reduce the computational cost. Since the input feature maps are not shared by different kernels in the depthwise stage, DW leads to under-utilization of PEs on the accelerators optimized for normal convolution. The lack of input reuse also results in a lower computation-to-memory ratio and the demand for a higher on-chip buffer bandwidth. To efficiently support both DSC and normal convolution, we unify the computation flow of both types of convolution with *ESCALATE* algorithm. We apply the same decomposition scheme to DSC and use the same decomposed form as described in Equation (3). With the unified computation flow, our design can efficiently support both types of convolution (see Section 4). With the notations defined before, the weights of DSC can be represented by the kernels in the depthwise convolution $W_{DW} \in \mathbb{R}^{C \times RS}$ and the coefficients in pointwise convolution $W_{PW} \in \mathbb{R}^{C \times K}$. We can decompose the weights of the depthwise convolution as $W_{DW} = C'_e B$ where $C'_e \in \mathbb{R}^{C \times M}$ and $B \in \mathbb{R}^{M \times RS}$. Then, we can combine W_{PW} and C'_e into the coefficient matrix by computing the Hadamard product of each column of C'_e and W_{PW}, which can be represented as:

$$C_e^{(c,k,m)} = W_{PW}^{(c,k)} C_e'^{(c,m)}. \tag{5}$$

Thus, we have a unified representation with Section 2.3. Although this decomposition increases the number of convolution operations by k times, the convolution kernels are shared across input channels. This kernel sharing allows us to efficiently support both types of convolution on the same architecture.

4 *ESCALATE* ARCHITECTURE

We illustrate the *ESCALATE* accelerator design in Figure 4(a). *ESCALATE* presents a hierarchical PE design by splitting each PE into slices (lines). Each PE slice has two parts, corresponding to the two stages of the decomposed convolution. The first part, namely channel accumulator (CA), uses the *Dilution-Concentration* mechanism to efficiently eliminate ineffectual computations. The second part consists of multiply-accumulate (MAC) units organized in a row. Each MAC has a small FIFO to hold one basis kernel. The basis kernels are loaded into the FIFO before the computation begins and remain in the FIFO. The MAC row design actively exploits the channel parallelism of the intermediate feature maps and the reuse of the basis kernels. We build separate buffers to store each slice of the input feature maps to process multiple rows of input feature maps in parallel. The PE slices of the same position in different PE blocks are connected to the same input buffer. Since the coefficients are unique to the computation of each PE Block, we build individual coefficient buffers in each PE Block.

ESCALATE makes three contributions. First, *ESCALATE* accelerator features a novel Basis-First dataflow to exploit the unique parallelism and data reuse opportunities brought by kernel decomposition. Second, we propose a Dilution-Concentration scheme to efficiently eliminate ineffectual computation with bit gather. Lastly, *ESCALATE* provides an efficient input buffer design to support asynchronously running PE slices to alleviate the impact of workload imbalance.

4.1 Basis-First Dataflow

Existing CNN accelerator dataflows are designed for regular convolution. Those dataflows do not fit the two-stage computation of the decomposition convolution. We propose the *Basis-First (BF)* dataflow for decomposed convolution. Our objectives are to maximize the reuse of each batch of inputs, reduce the global buffer accesses, and avoid stalling the MACs. The BF dataflow is illustrated in Figure 3.

BF dataflow confines the computation of one output channel to one PE block so that we can fully utilize the distributed coefficient buffer and avoid cross-PE communication. We spatially map one row of the output feature map to one PE slice with the offset of the total number of PE slices. Inside each PE slice, we process one position of all input feature maps at one time and spatially map each intermediate channel (i.e., index m) to each CA-MAC pair. Based on the same observation with [30], we multiply each element of the intermediate feature maps by all weights in the corresponding kernel (partial weights for the elements on the edge). The product is sent to the partial sum buffer and accumulated to the corresponding output position through the read-modify-write operation. Since the output accumulation is not at the critical path of the processing and is less sensitive to the latency, we do not attempt to reduce bank conflicts at the partial sum buffer with additional optimizations.

We also support regular convolution in *ESCALATE* as a fallback for those layers that cannot be compressed (e.g., the first convolutional layer). For those layers, we employ the input stationary dataflow to align with the objective of maximizing input reuse for decomposed convolution. We directly bypass the channel accumulator and only use the MACs for these layers. For the fully connected layer, we convert it into 1×1 convolution with 1×1 input feature maps to maintain a unified mapping. We do not build separate units to support the sparsity in those layers because: 1) they only take up a small portion of the overall computation (< 5%); 2) those layers usually have a low sparsity ratio and the overhead of supporting sparsity for those layers can outweigh the computation reduction. For example, as shown in [33], the pruning ratio of the first layer is only 1.2-1.6× and the sparse weight even causes performance degradation comparing with its dense version.

4.2 Dilution-Concentration

4.2.1 Sparse Encoding. The sparsity in weights and activations can reduce the storage and bandwidth consumption. However, we need a proper encoding scheme to index the non-zero values. Previous works [30, 40] adopt the Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC), which use separate arrays to store

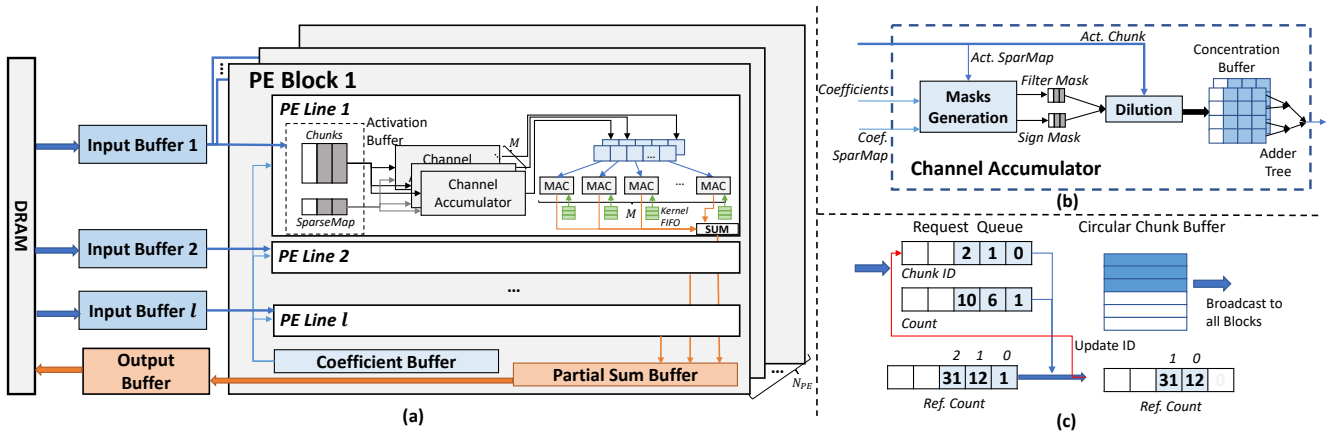

Figure 2: (a) The microarchitecture of *ESCALATE*. M corresponds to the hyperparameter in the decomposed convolution, while l and N_{PE} is the design space parameter. (b) The design of channel accumulator. (c) The design of input buffer. Each input buffer is implemented as a circular queue and use the reference count to evict the finished chunks.

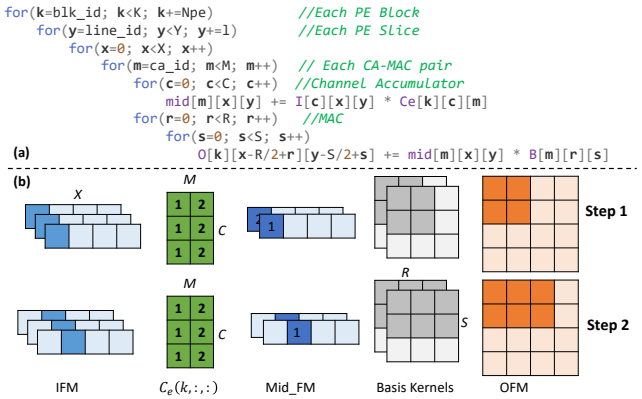

Figure 3: The Basis-First Dataflow. (a) The nested-loop expression. N_{pe} and l are design space parameters corresponding to number of PE blocks and number of slices per block. (b) The illustration of BF dataflow in one PE slice of the k-th PE Block. The number represents the index of CA-MAC pair. We assume $M = 2, C = 3$ and current PE slices is the first one. We assume a unit-stride convolution for simplicity.

the row/column index of the non-zero element. As discussed in [12], CSR and CSC are not efficient when the sparsity ratio is relatively low. Making it worse, the cost of storing one index is much higher than storing multiple ternary values, making the indexing overhead outweigh the benefits brought by sparsity.

We adopt the same SparseMap encoding of [12]. As shown in Figure 4(a), we slice the input feature map along the X dimension (i.e., row dimension) and store the rows of stride l in a contiguous array, where l is the number of concurrent accumulations in the first stage. The coefficients are sliced along the K dimension and stored in separate arrays for each output channel. We store a separate bit mask for each array indicating whether each position has non-zero values. Both activations and coefficients are stored in C-order to match the computation in the first stage of the decomposed convolution. To maintain the space efficiency under high sparsity situation, we also introduce a 2-level SparseMap encoding. We split the sparse map into 16 bit chunks and use one bit per chunk to indicate whether the chunk is all zero. The all zero chunks are not stored. Apart from the space efficiency, SparseMap encoding also simplifies the processing of the compressed arrays. We are able to match the index of non-zero elements in activation and weight arrays with bit-wise AND and bit gather operation, which has a low energy cost. Moreover, SparseMap allows us to fully utilize the on-chip bandwidth since we can implicitly extract the number of processed elements through the index-matching process. We can insert a barrier indicating moving to a new position, so we do not need to split and pad the input as fixed-length chunks or have varied access lengths.

4.2.2 Dilution. The purpose of the dilution process is to match the input chunk of activations with the coefficients and filter out the activations corresponding to the zero coefficients. The activations are kept in high-precision (e.g., 8 or 16 bits) and the shuffling cost is relatively high. Thus, we keep the "holes" in the filtered chunks and leave it for the concentration step. Since the coefficients are quantized to ternary, we only need to generate two masks, one for filtering out the activations and the other for the sign. The generation of these two masks can be implemented with bit gather operations, which has been well-studied [16]. In the gather operation, we use a mask to indicate the valid elements. We collect all bits corresponding to the valid bits of the mask to the same side of the array while keeping their original order.

We show an example of bit gather implementation with an inverse butterfly network of $log_2(n)$ steps in Figure 4(b). With the bit gather operator, we depict the dilution process in Figure 4(c). Specifically, we calculate the intersection of input activations and coefficients by bit-wise AND. The filtering mask is generated by gathering the intersection with the activation sparse map as the mask. To obtain the sign mask, we first generate a coefficient mask by gathering the intersection with the sparse map of coefficients.

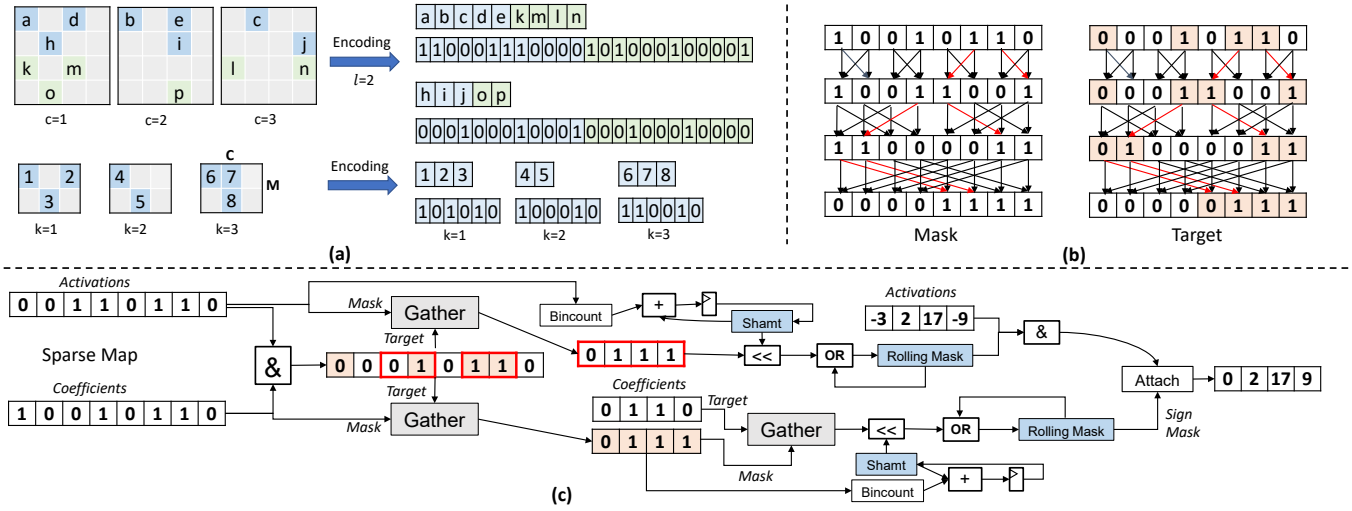

Figure 4: The illustration of sparse encoding and the dilution process. (a) The SparseMap encoding of the activations and coefficients. we assume $C = 3, M = 2, K = 3$ for this example. (b) Bit gather operation with an inverted butterfly network. (c) Dilution process. We generate activation masks and coefficient masks through bit gather, and use the masked coefficients as a sign-mask to obtain the sign-extended activations for concentration process. We omit the higher 4-bits during process in the figure for simplicity.

The coefficient mask indicates whether the corresponding non-zero value of coefficients appears in the intersection. Then, we gather the non-zero value (i.e., the sign bit of ternary coefficients) with the coefficient mask and generate the sign mask. With the filtering mask and sign mask, we can filter the input chunks and change the sign of each filtered activation. The sign bit is also attached to each activation for the concentration step to apply proper scaling. Since the sparse map is transferred and stored separately with the non-zero values, we can start the generation of the masks ahead of the reading of the chunks. The whole process can be pipelined to satisfy the timing constraints without impacting the throughput.

Since the distribution of non-zero elements could vary between activations and coefficients, the size of the mask generated through one pass might not be able to cover the current input chunk. We address this issue by keeping a rolling mask, which is shown in Figure 5. The newly generated masks are left-shifted by the length of the current rolling masks and attached to the rolling mask through bit-wise OR. Once the size of the rolling mask is large enough to

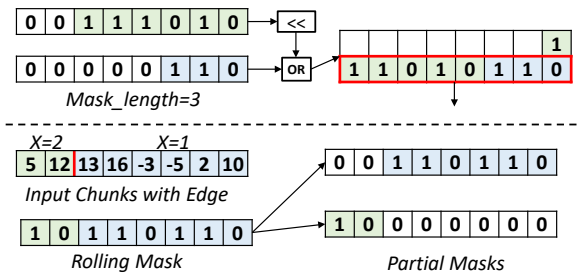

Figure 5: Rolling mask and implicit barrier.

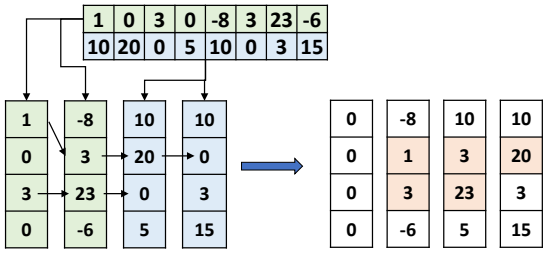

Figure 6: Concentration through look-ahead and look-aside.

cover the current chunk, the corresponding part is evicted from the buffer. This scheme also supports the implicit barrier between consecutive input positions. We keep a counter for the rolling mask. If the count indicates that all elements that correspond to the current position have been processed, we break the current mask into two partial masks and separately apply them to the current input chunk. In other words, the rolling mask creates a barrier to separate the activations corresponding to different positions. With the barrier, we can avoid the expensive element rotation operation and fully utilize the bus bandwidth at the cost of two partially utilized cycles.

4.2.3 Concentration. With the "diluted" and signed chunks of activations, the concentration process is relatively simple. As shown in Figure 6, we collect the chunks produced by the dilution process, folding each of the chunks into multiple columns, and trying to fill all zeros with the column-wise "look-ahead" and "look-aside". This process is equivalent to reorder the activations. Since we have attached the coefficients as the sign bit, we can permute the activations in arbitrary order. We use double buffers in this step. When

one buffer is used to feed the activations to the reduction tree, we collect and concentrate the processed chunks in the other buffer. The functionalities of the two buffers swap when the adder tree has processed all non-zero chunks. The concentration process is performed in both buffers until the buffer is full or there's no more possible movement of the elements. The aforementioned barrier results in an immediate flush of the buffer, which guarantees that the chunks from different positions will not get mixed up.

We implement the Dilution-Concentration mechanism in Channel Accumulator Unit (CA), which is depicted in Figure 2(b). Since the SparseMaps are stored and processed separately with the input chunks, we can start the mask generation in advance and overlap the process with other parts of the computation. The parallel-running mask generation could process the sparse pattern in advance and avoid blocking the major datapath.

4.3 Buffer Design

Unlike previous sparse accelerators, *ESCALATE* does not require a reduction across PE slices. The load imbalance can be mitigated by asynchronously running PE slices. However, it could also reduce the reuse opportunities of the input feature maps and result in a higher cost of data movement. We address this issue by altering the input buffer design. Instead of using a large, unified input buffer, we break it into multiple individual buffers. Each buffer is connected to the PE slices of the same position in all PE blocks since they process the same lines of the input feature map. Inside the buffer, we organize the non-zero elements of the input feature maps into chunks. Since the chunks are accessed strictly in order, we store those chunks in a circular queue. We use registers to keep the head and tail pointers to the queue. For each chunk, we keep a chunk ID as well as a counter to count the number of PE slices that have not processed this chunk. The buffer access requests are collected through an H-Tree connecting each PE slice. The node of the tree is an arbitrator that will also calculate the number of PE slices which the current winning requests could cast to. A separate request queue in the buffer stores all outstanding access requests. The chunk as well as its ID corresponding to the head of the request queue is broadcast to all PE slices. The counter of the accessed chunk is updated by subtracting the count in the current request. If the counter is decreased to zero, the corresponding chunk will be evicted from the queue. A signal will be sent to all slices to update the ID of the chunk. The chunk ID in the request queue will also be updated. To efficiently utilize the capacity of the buffer, we apply a greedy policy in each arbitrator. The arbitrator will prioritize the request to earlier chunks. The design of the input buffer is shown in Figure 2(c).

5 EVALUATION

To evaluate the effectiveness of the *ESCALATE* framework, we test a variety of representative CNN models on both CIFAR-10[22] and ImageNet[10] datasets. VGG16[36], which is known to be redundant and easy to compress, is used as a sanity check for the proposed framework. We select ResNet[14] as the target model for evaluation, including two variants for CIFAR-10: a shallow ResNet18 and deep ResNet152. For ImageNet, we use ResNet50, which is widely evaluated by previous compression works. We also select MobileNet [17] and MobileNetV2 [34] to evaluate the effectiveness of our framework on compact models.

Table 1: Compression result of *ESCALATE* algorithm.

Model (Method)	Top-1 (%)	CONV (MB)	Comp. (×)	Spar. (%)	Prun.[1] (%)
CIFAR-10					
VGG16	93.49	56.12	-	-	-
STQ	92.38	2.21	25.10	N/A	N/A
ADMM-NN-S	93.10	0.54	109	98.3	
Ours	92.74	0.71	79.04	89.24	96.1
ResNet18	93.79	42.58	-	-	-
ADMM-NN-S	93.3	0.33	135	98.6	
Ours	93.63	0.4	106.45	97.4	98.21
ResNet152	95.36	221.19	-	-	-
Naive_L1	94.12	20.45[2]	13.31	92.49	
Ours	93.86	0.68	325.27	99.2	99.4
MobileNetV2	94.09	8.40	-	-	-
ADMM-NN-S[3]	94.90	0.54	18.8	83.6	
Ours	93.32	0.73	11.51	96.98	91.86
ImageNet					
ResNet50	76.25	78.03	-	-	-
STR	74.31	7.62[2]	10.24	90.23	
ResRep	75.3	48.16[4]	1.62	N/A	62.1
Ours	73.89	7.17	10.92	88.22	92.16
MobileNet	70.10	26.94	-	-	-
STR	68.35	6.65[2]	4.05	75.28	
ResRep	68.02	13.81[4]	1.95	N/A	73.91
Ours	67.89	3.02	8.92	67.6	63.9

[1]w.r.t the original weights before decomposition.
[2]Does not include indices.
[3]Trained with Mixup augmentation.
[4]Estimated through FLOPs reduction reported in paper.

5.1 Algorithm

5.1.1 Experiment Settings. On CIFAR-10, baseline models are trained for 350 epochs using Nestrov accelerated SGD optimizer with 0.9 momentum and 0.0001 weight decay. The learning rate is set to 0.1 initially and multiplied by 0.1 at the 50% and 75% epochs. For *ESCALATE* , we select $M = 6$ for decomposition. The retraining process lasts for 300 epochs using ADAM [21] optimizer with 0.001 initial learning rate and the same decay policy. We set $t = 0.05$ in the ternary quantization process of coefficients . On ImageNet, we use the pre-trained model provided by PyTorch [31] as the ResNet50 baseline, and train the baseline model of MobileNet with 0.1 initial learning rate. The retraining process lasts for 60 epochs with ADAM optimizer with 0.0001 initial learning rate. Other parameters are the same with CIFAR-10 training.

5.1.2 Analysis. The compression results of *ESCALATE* algorithm are summarized in Table 1. We only show the result of convolutional layers since the *ESCALATE* algorithm only processes convolutional layers. Since other layers take up a small portion of the overall computation cost, they have minor impact on the overall performance. We assume 32bit floating-point precision for the baseline models.

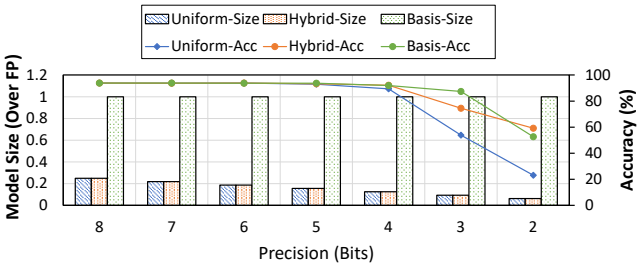

Figure 7: The comparison of model size and accuracy with uniform, hybrid and basis-only quantization. We omit coef-only since it shows identical behavior with hybrid.

For *ESCALATE* , we use 8bit for the first convolutional layer and all the basis kernels and use the SparseMap encoding as mention in Section 4.2 for the coefficients. We compare our result with the quantization method STQ[27], the structured prunning method ResRep [11], non-structured pruning method STR [23], and joint pruning and quantization method presented in ADMM-NN-S[26]. Although ADMM-NN-S presents the results of both structured and non-structured pruning methods, its structured pruning is a fine-grained variant; the hardware still needs to be specialized to skip the pruned columns in the filters. Since non-structured pruning shows a higher pruning ratio, we select it as the baseline model both in algorithm and hardware evaluation. For structured pruning methods, we ignore the sparse encoding overhead and assume the parameters are represented using 32bit floating point. The proposed *ESCALATE* algorithm approaches the non-structured baselines in terms of both compression ratio and accuracy. On CIFAR-10, we achieve a compression rate ranging from 11× to 325×. *ESCALATE* algorithm reaches a similar compression rate on all CIFAR-10 models with the non-structured ADMM-NN-S. The encoding overhead of SparseMap results in the gap. *ESCALATE* is also able to efficiently explore the redundancy in a large model like ResNet152 and prune 99.2% of parameters, compressing the orignal model by 325×. Some downsampling layers in ResNet152, especially those in the last three residual blocks, are completely pruned. These results indicate that *ESCALATE* algorithm can effectively identify and eliminate the redundancy in the CNN models. For the compact MobileNetV2 model, we can also achieve over 11× compression ratio. *ESCALATE* incurs a 1.5% accuracy loss on ResNet152, and less than 0.8% accuracy loss on the remaining three models compared to the uncompressed models. On ImageNet, we achieve a similar sparsity with non-structured pruning method STR with less than 0.5% accuracy gap. For MobileNet, we maintain the accuracy at 67.89% while compressing the original model into 3.02MB.

We also show the advantage of hybrid quantization by performing post-training quantization on decomposed ResNet18 model. Uniform quantization policy enforces the same precision on both basis and coefficients, while hybrid quantization keeps the basis in 8 bits and only further quantizes the coefficients. The result is shown in Figure 7. Since the basis kernels only occupy a small portion of the parameters while being frequently reused in computation, keeping the basis kernels in high precision effectively maintains model accuracy. These results show that hybrid quantization can achieve almost the same compression ratio as uniform quantization while maintaining accuracy.

Table 2: Configurations of *ESCALATE* and baselines.

ESCALATE			
M	6	Input Buf.	8KB
N_{PE}	32	Coef. Buf.	512Bytes
l	5	Output Buf.	4KB
Input Bus	16Byte	Psum Buf.	2KB
Precision	8bit	Act. Buf.	16Byte×4
Other Baselines			
Proportional scaling of on-chip SRAM buffer. 1024 8-bit multipliers			

Table 3: Unit energy cost per 8-bit integer operation extracted from commercial TSMC 65nm technology.

	DRAM	MAC	Multiply	Add
Energy(pJ/8-bit Int)	100	0.407	0.186	0.036

5.2 Accelerator

5.2.1 Experiment Settings. To estimate power and area, we implement the RTL design of *ESCALATE* and synthesize it using Synopsys Design Compiler with TSMC 65nm library under the typical corner, 1V, and 25°C. The design achieved 800MHz frequency. The power and area estimated from the synthesized result is shown in Table 4. We then implement a cycle-accurate simulator and verify it against the RTL implementation. The simulator generates SRAM and DRAM access traces. For SRAM, we use CACTI 7.0[3] to estimate the power consumption. For DRAM, we simulate the trace using ramulator[20] and extract the energy consumption from the command trace with DRAMPower[4]. We only evaluate the convolutional layers since *ESCALATE* does not process other types of layers, and the SCNN baseline only supports convolutional layers.

For performance baseline, we select a DNN accelerator optimized for dense networks, Eyeriss[7], and two sparse CNN accelerators, SCNN[30] and SparTen[12]. We use TimeLoop [29] to simulate Eyeriss, and use DNNSim [18] to simulate SCNN, respectively. For SparTen, we implement a cycle-accurate simulator and verify it against results reported by the original paper. CACTI is used to estimate the area and energy consumption of on-chip buffers. We extract the energy consumption of unit operations under the same

Table 4: Power and Area estimation of PE Block(65nm)

Component	Area(mm^2)	Power(mW)
Activation Buffer	0.0098	5.44
MAC Row	0.0159	7.79
Dilution	0.0450	17.77
Concentration	0.0906	46.74
Coef.&Psum Buffer	0.0538	8.33
Total	0.2150	86.07

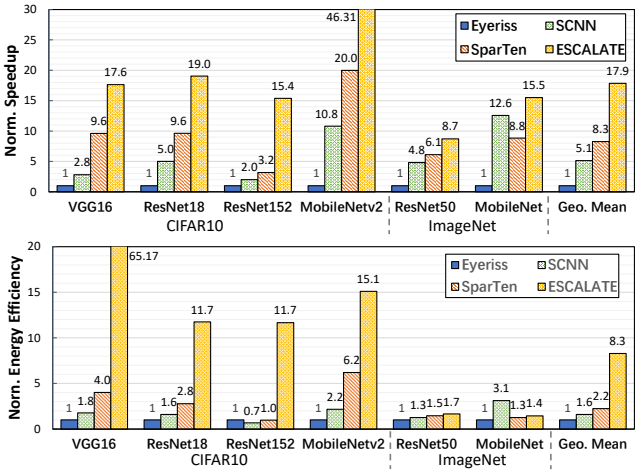

Figure 8: The normalized speedup and energy efficiency *(over Eyeriss)* of *ESCALATE* and baseline accelerators.

65nm technology node, as shown in Table 3, to estimate the energy cost of other hardware components. According to [39], the energy cost of DRAM accesses can be approximated as 100pJ per 8 bits. The configuration of all baseline designs are adjusted to have the same number of multipliers. For baseline accelerators, we use the model checkpoints from ADMM-NN-S [26] for all CIFAR-10 models except ResNet152 and the checkpoints from STR [23] for all ImageNet models. Since no previous work provides a checkpoint for ResNet152, we use the naïve magnitude-based pruning method with $l1$-regularization to build a sparse model. The sparsity of the models used for baseline accelerators can be found in Table 1. Since the result is also related to the activation sparsity, the result may vary with different input samples. We randomly generate 10 input samples and present the average speedup and energy consumption. We list the configurations of *ESCALATE* accelerator and baseline accelerators in Table 2.

5.2.2 Main Result. Figure 8 showcases the normalized speedup and energy efficiency of all accelerators over Eyeriss. Compared with the baseline accelerators, *ESCALATE* achieves the best performance under all evaluated models. Comparing with Eyeriss, which does not exploit sparsity, we achieve a speedup ranging from 8.7× to 46.31×. *ESCALATE* benefits from the computation reduction

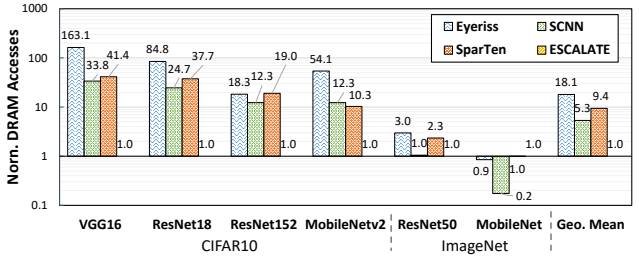

Figure 9: The normalized DRAM accesses *(w.r.t. ESCALATE)* of baseline accelerators on all evaluated models.

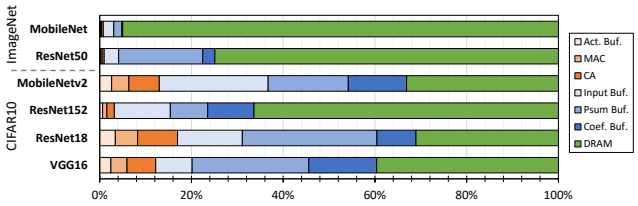

Figure 10: The inference energy breakdown of all evaluated models. We omit the output buffer since its energy consumption is negligible compared with other hardware components.

brought by both sparsity and the decomposed form of convolution. Since the second stage of the decomposed convolution is dense, the upper limit for the speedup of each layer is determined by the layer shape, specifically M/C. For CIFAR-10 models, the high sparsity (>90%) enables the channel accumulator to produce intermediate feature maps in time. Thus, if there is no idle MAC, the speedup is bounded by the layer shape. For ImageNet models, the sparsity is relatively low. The CA requires more cycles to produce an intermediate element than a MAC needs to consume. The issue of idle MAC cycles limits the performance, which we discuss in Section 6.2. Comparing with sparse baselines, *ESCALATE* also respectively outperforms SCNN and SparTen by 3.5× and 2.16× on average.

As for the energy efficiency, the result diverges based on the type of CNN models. Input feature maps of CIFAR-10 models are relatively small. Thus, the off-chip access of weights dominates the energy consumption. The coefficient compression makes it possible for *ESCALATE* to retain most of the weights on-chip during the whole computation process. *ESCALATE* can almost eliminate the expensive DRAM accesses for weights during the computation, resulting in over 10× improvement in energy efficiency. For ImageNet models, the movement of input feature maps dominates the DRAM traffic. *ESCALATE* has a similar energy consumption with SparTen due to the same channel-first order in processing the input feature maps. The large activation buffer of SCNN effectively reduces this part of the cost, resulting in up to 3.1× energy efficiency improvement. On average, we achieve an 8.3× energy efficiency improvement over Eyeriss, 5.19× improvement over SCNN, and 3.78× improvement over SparTen.

Figure 9 illustrates the normalized number of DRAM accesses over *ESCALATE* . As we mentioned above, for ImageNet models, the DRAM accesses for input feature maps dominate the overall energy consumption. *ESCALATE* requires a similar or larger number of DRAM accesses compared to baseline for these models. On CIFAR-10, *ESCALATE* effectively reduces the expensive DRAM accesses. The reduction relative to SCNN results from the eliminated off-chip weight accesses. Comparing with SparTen, our input buffer design exhaustively exploits the input reuse without enforcing a large-scale synchronization barrier. On average, *ESCALATE* reduces the DRAM accesses by 18.1× over Eyeriss, 5.3× over SCNN, and 9.4× over SparTen, respectively.

Figure 10 shows the energy breakdown of *ESCALATE* accelerator on all evaluated models. Apart from the DRAM accesses we have discussed before, the breakdown also reveals a divergence in the

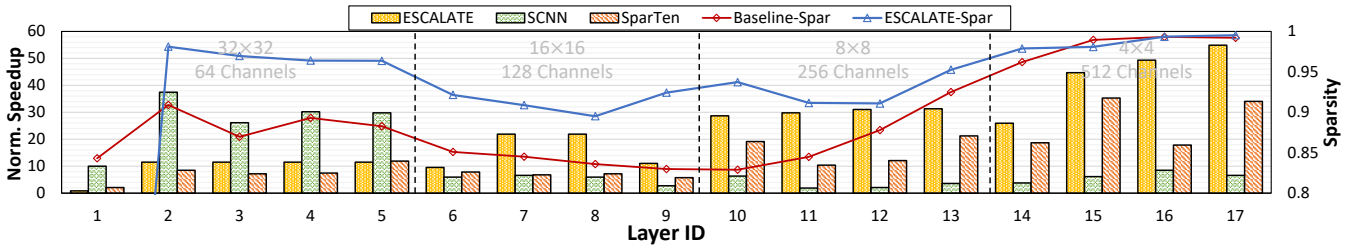

Figure 11: Layerwise sparsity and speedup over dense accelerator (Eyeriss) in ResNet-18 model. We also show the width (i.e., number of input channels) and the size of the input/output feature map of each layer.

energy consumption of buffers. For shallow models like ResNet18 or VGG16, the partial sum buffer dominates the buffer energy consumption; this is because the output feature maps are kept dense in the partial sum buffer and require frequent read-modify-write operations. For deeper models like ResNet152, the cost of reading input activations outweighs the partial sum accumulation. The large number of 1×1 convolutional layers amortize the read-modify-write cost of other types of layers. Moreover, the ResNet152 model features more 'late layers', which have a large number of input channels but a relatively small input size. This observation also applies to MobileNetv2 result since MobileNetv2 quickly downscales the size of feature maps in early layers that have a small number of channels. In summary, the reduction in DRAM accesses, especially off-chip weight accesses, is the main reason for the improved energy efficiency of *ESCALATE* .

5.2.3 Layer-wise Analysis. The computational efficiency of *ESCALATE* accelerator is affected by the layer width and the size of the input feature maps. To evaluate the impacts of these factors, we perform a layer-wise analysis. Figure 11 shows the layer-wise normalized speedup of *ESCALATE* and all baseline sparse accelerators in ResNet-18. We also mark the layer shape of each residual block and present the sparsity of each layer. For the first layer, *ESCALATE* is slower than the dense baseline for the following reasons, (1) directly mapping the computation into MAC row without skipping the zero activations or coefficients; (2) the fallback input stationary dataflow is not as efficient as the row stationary dataflow in Eyeriss. Since the first layer only contributes to a small portion of the overall computational cost, we do not further optimize for the first layer in *ESCALATE* to avoid increasing the design complexity and degrading the efficiency of other layers. For the other layers, we can tell a clear boundary between SCNN and SparTen—SCNN effectively utilizes spatial parallelism in early layers, while SparTen exploits the channel parallelism in late layers. *ESCALATE* presents a similar layer-wise speedup pattern to SparTen since both designs use input channels as the inner-most loop of dataflow. Comparing with SparTen, the overlapped two computation stages of *ESCALATE* further boosts the efficiency. As we mentioned before, due to the high sparsity in CIFAR-10 models, the performance is bounded by layer shapes. Within the first three blocks of layers, *ESCALATE* almost reaches the C/M limit of speedup. For the last block, the speedup varies across layers. Since the input feature maps of these layers are very small, most intermediate results require less than *RS* cycles in MAC row, leaving the MACs running idle.

6 DISCUSSION

6.1 Design Trade-off

ESCALATE provides a trade-off between accuracy and latency/energy by adjusting the number of basis kernels M in the decomposition step. With a different M, we can adjust l to maintain the same number of MACs and resource consumption. Increasing the number of basis kernels can effectively increase the model accuracy. However, it also reduces the number of PE slices per block, leading to a reduction in row parallelism. We show the trade-off with both ResNet18 and ResNet50 models in Figure 12. For $M = 7$, we achieve 93.83% accuracy on ResNet18 and 74.09% on ResNet50. When increasing M, we have a smaller l under the constraints of maintaining the number of MACs, thus reducing the row parallelism and increasing the latency. The change in l also affects the number of input buffers. A larger l requires more input buffers but reduces the cost of off-chip DRAM accesses. The energy consumption of other components shows negligible change with the number of basis kernels.

6.2 Overhead Analysis

In *ESCALATE* design, both CA and MAC can be idle. If CA cannot produce a new intermediate element before MAC finishes the current computation, the MAC has to stall. Conversely, if MAC requires more cycles to consume one element (e.g., a large kernel), the CA has to stall. We only consider the idle MACs as overhead since the idle CA can still process masks or perform concentration. As we mentioned before, in CIFAR-10 models, we did not observe significant idle MACs ($< 0.05\%$ of overall cycles) thanks to the high sparsity ratio, while the issue of idle MACs limits the speedup of ImageNet models. We show the layer-wise portion of MAC idle cycles in Figure 13. The portion of idle cycles is determined by both

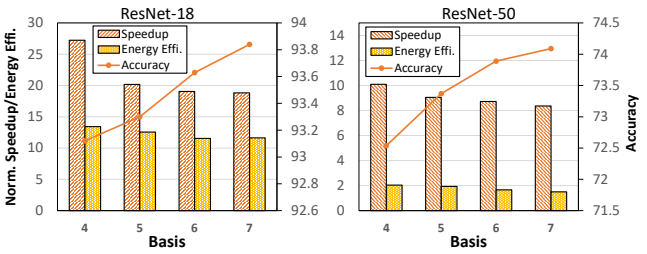

Figure 12: The accuracy and latency/energy trade-off with different number of basis kernels (M).

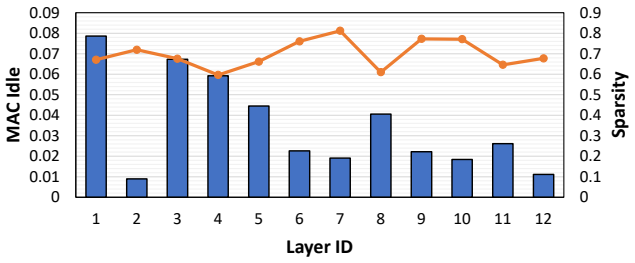

Figure 13: The MAC idle cycles and sparsity of each layer in MobileNet.

the sparsity ratio and the distribution of non-zero elements. CA requires more cycles to process the computation corresponding to denser coefficients, leading to more idle cycles of MAC.

6.3 Modern Compact CNNs

Through our experiments, we observe that the variation in layer dimensions could complicate the accelerator design. For sparse accelerators, it's difficult to be efficient under *all* types of layer configurations. Modern compact CNNs, like EfficientNet [37], usually include a rich set of layer variants, making it hard to be efficiently deployed to a sparse-aware accelerator. We also noticed that, in our experiments, sparse VGG16 is 1.5× faster than sparse MobileNetv2, while they achieve a similar accuracy (0.5% difference). Modern compact models are mostly designed for general-purpose processors on the edge. Since these platforms cannot efficiently support processing sparse data, the compact models do not include the performance of the sparse version into their design considerations. They are not suitable for sparse-aware accelerators and might even be outperformed by the sparse version of large and redundant models. This situation motivates us to explore the possibilities of jointly designing sparse-aware accelerator and hardware-aware CNN models in future works.

7 RELATED WORKS

CNN Compression. Pruning and decomposition are two important compression techniques for CNN models. Pruning removes the redundant structures in the CNN weights. Among all pruning methods, weight pruning [13, 23, 41] removes individual weight values based on its magnitude and recovers the accuracy using different retraining techniques. Previous weight pruning methods pursuit a high compression ratio, ignoring the hardware efficiency of the compressed model. Structured pruning[11, 38] eliminates the whole filter or channel to maintain its original computation flow and data locality. Although structured pruning can directly benefit the hardware, it can only achieve a limited compression ratio due to the limited pruning pattern. Decomposition[24, 42] adopts matrix decomposition or tensor decomposition to represent the original weight tensor with multiple low-rank structures. These works did not exploit the sparsity in the decomposed models. In summary, previous compression methods focus on algorithm level metrics or the speedup on existing hardware, ignoring the potential of the co-optimization.

CNN Accelerators. A vast number of CNN accelerator designs have been proposed to boost the inference efficiency. Dense accelerators [5–7] optimize the inference efficiency by exploring the reuse opportunities and designing efficient dataflows. Later designs add the support for sparse CNN models: Cambricon-X [40] and Cnvlutin [2] only utilize the sparsity from one side. SCNN [30] and SparTen [12], which we have extensively discussed in our paper, exploit the two-sided sparsity. The bit-serial accelerators[1, 9, 35] reduce the computation by removing ineffectual bits during the multiplication. Since the whole value, other than just effectual bits, needs to be stored and transferred, these designs may not reduce the bandwidth cost. Another type of sparsity-aware accelerators optimizes for sparse tensor or matrix operations and maps CNN model onto these primitives. ExTensor [15] optimizes for sparse tensor operations, while SIGMA[32] optimizes for sparse GEMM operations. These accelerators are not specifically optimized for the inference of sparse CNN.

8 CONCLUSION

In this paper, we present *ESCALATE* , a kernel decomposition-based algorithm-hardware co-design framework to boost the inference efficiency of sparse CNN models. We reorganize the decomposed convolution to eliminate the computation bottleneck and apply hybrid quantization to exploit the discrepancy in parameter reuse frequencies. We propose 'Basis-First' dataflow and corresponding microarchitecture design to support the *ESCALATE* -compressed CNN model. Extensive experiments show that *ESCALATE* algorithm achieves up to 325× compression rate with negligible additional accuracy loss compared to previous compression techniques, while *ESCALATE* accelerator outperforms previous sparse CNN accelerator designs with up to 2.16× reduction in latency and 3.77× improvement in energy efficiency.

ACKNOWLEDGMENTS

This work was supported in part by NSF-2112562, NSF-1937435, ARO-W911NF-19-2-0107, NSF-1822085, and NSF IUCRC for ASIC memberships including Samsung, etc. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the grant agencies or their contractors.

REFERENCES

[1] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-Pragmatic Deep Neural Network Computing. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 382–394.

[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. 1–13. https://doi.org/10.1109/ISCA.2016.11

[3] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[4] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. [n. d.]. DRAMPower: Open-source DRAM Power & Energy Estimation Tool. http://www.drampower.info.

[5] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proceedings of the 19th International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14). 269–284. https://doi.org/10.1145/2541940.2541967

[6] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. DaDianNao: A Machine-Learning Supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 609–622.

[7] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[8] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308. https://doi.org/10.1109/jetcas.2019.2910232

[9] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. 2019. Bit-Tactical: A Software/Hardware Approach to Exploiting Value and Bit Sparsity in Neural Networks *(ASPLOS '19)*. 749–763. https://doi.org/10.1145/3297858.3304041

[10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition(CVPR)*. 248–255.

[11] Xiaohan Ding, Tianxiang Hao, Ji Liu, Jungong Han, Yuchen Guo, and Guiguang Ding. 2020. Lossless CNN Channel Pruning via Gradient Resetting and Convolutional Re-parameterization. *arXiv preprint arXiv:2007.03260* (2020).

[12] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 151–165.

[13] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations (ICLR)* (2016).

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition(CVPR)*. 770–778.

[15] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. 319–333. https://doi.org/10.1145/3352460.3358275

[16] Yedidya Hilewitz and Ruby B Lee. 2008. Fast Bit Gather, Bit Scatter and Bit Permutation Instructions for Commodity Microprocessors. *Journal of Signal Processing Systems* 53, 1 (2008), 145–169.

[17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861* (2017).

[18] isakedo. [n. d.]. isakedo/DNNsim. [EB/OL]. https://github.com/isakedo/DNNsim.

[19] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2016. Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

[20] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (2015), 45–49.

[21] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[22] A. Krizhevsky and G. Hinton. 2009. Learning Multiple Layers of Features from Tiny Images. *Master's thesis, Department of Computer Science, University of Toronto* (2009).

[23] Aditya Kusupati, Vivek Ramanujan, Raghav Somani, Mitchell Wortsman, Prateek Jain, Sham Kakade, and Ali Farhadi. 2020. Soft Threshold Weight Reparameterization for Learnable Sparsity. In *International Conference on Machine Learning*. PMLR, 5544–5555.

[24] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan V. Oseledets, and Victor S. Lempitsky. 2015. Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[25] Shiyu Li, Edward Hanson, Hai Li, and Yiran Chen. 2020. PENNI: Pruned Kernel Sharing for Efficient CNN Inference. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research)*, Vol. 119. PMLR, 5863–5873.

[26] Xiaolong Ma, Sheng Lin, Shaokai Ye, Zhezhi He, Linfeng Zhang, Geng Yuan, Sia Huat Tan, Zhengang Li, Deliang Fan, Xuehai Qian, Xue Lin, Kaisheng Ma, and Yanzhi Wang. 2021. Non-Structured DNN Weight Pruning–Is It Beneficial in Any Platform? *IEEE Transactions on Neural Networks and Learning Systems* (2021), 1–15. https://doi.org/10.1109/TNNLS.2021.3063265

[27] Grégoire Morin, Ryan Razani, Vahid Partovi Nia, and Eyyüb Sari. 2019. Smart Ternary Quantization. *arXiv preprint arXiv:1909.12205* (2019).

[28] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-Based Weight Pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 907–922. https://doi.org/10.1145/3373376.3378534

[29] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315. https://doi.org/10.1109/ISPASS.2019.00042

[30] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. 27–40. https://doi.org/10.1145/3079856.3080254

[31] PyTorch. [n. d.]. torchvision.models – PyTorch 1.7.0 documentation. [EB/OL]. https://pytorch.org/docs/1.7.0/torchvision/models.html.

[32] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[33] Ao Ren, Tianyun Zhang, Shaokai Ye, Jiayu Li, Wenyao Xu, Xuehai Qian, Xue Lin, and Yanzhi Wang. 2019. ADMM-NN: An Algorithm-Hardware Co-Design Framework of DNNs Using Alternating Direction Methods of Multipliers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 925–938. https://doi.org/10.1145/3297858.3304076

[34] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[35] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. 2019. Laconic Deep Learning Inference Acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. 304–317. https://doi.org/10.1145/3307650.3322255

[36] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[37] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research)*, Vol. 97. PMLR, 6105–6114.

[38] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. 2082–2090.

[39] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Emberton Bell, Jeff Ou Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, et al. 2018. DNN Dataflow Choice Is Overrated. *arXiv preprint arXiv:1809.04070* (2018).

[40] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783723

[41] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. 2018. A Systematic DNN Weight Pruning Framework using Alternating Direction Method of Multipliers. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[42] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. 2016. Accelerating Very Deep Convolutional Networks for Classification and Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38, 10 (2016), 1943–1955. https://doi.org/10.1109/TPAMI.2015.2502579

[43] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 15–28. https://doi.org/10.1109/MICRO.2018.00011

[44] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. 2017. Trained Ternary Quantization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.