CONTRIBUTED PAPER

Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights

This article surveys the efficient execution of sparse and irregular tensor computations of machine learning models on hardware accelerators.

By Shail Dave, *Graduate Student Member IEEE*, Riyadh Baghdadi, *Member IEEE*, Tony Nowatzki, *Member IEEE*, Sasikanth Avancha, *Member IEEE*, Aviral Shrivastava, *Senior Member IEEE*, and Baoxin Li, *Senior Member IEEE*

ABSTRACT | Machine learning (ML) models are widely used in many important domains. For efficiently processing these computational- and memory-intensive applications, tensors of these overparameterized models are compressed by leveraging sparsity, size reduction, and quantization of tensors. Unstructured sparsity and tensors with varying dimensions yield irregular computation, communication, and memory access patterns; processing them on hardware accelerators in a conventional manner does not inherently leverage acceleration opportunities. This article provides a comprehensive survey on the efficient execution of sparse and irregular tensor computations of ML models on hardware accelerators. In particular, it discusses enhancement modules in the architecture design and the software support, categorizes different hardware designs and acceleration techniques, analyzes them in terms of hardware and execution costs, analyzes achievable accelerations for recent DNNs, and highlights further opportunities in terms of hardware/software/model codesign optimizations (inter/intramodule). The takeaways from this article include the following: understanding the key challenges in accelerating sparse, irregular shaped, and quantized tensors; understanding enhancements in accelerator systems for supporting their efficient computations; analyzing tradeoffs in opting for a specific design choice for encoding, storing, extracting, communicating, computing, and load-balancing the nonzeros; understanding how structured sparsity can improve storage efficiency and balance computations; understanding how to compile and map models with sparse tensors on the accelerators; and understanding recent design trends for efficient accelerations and further opportunities.

KEYWORDS | Compact models; compiler optimizations; dataflow; deep learning; deep neural networks (DNNs); dimension reduction; energy efficiency; hardware/software/model codesign; machine learning (ML); pruning; quantization; reconfigurable computing; sparsity; spatial architecture; tensor decomposition; VLSI.

Manuscript received June 1, 2020; revised March 7, 2021 and July 5, 2021; accepted July 15, 2021. Date of publication August 5, 2021; date of current version September 20, 2021. This work was supported in part by NSF under Grant CCF 1723476—NSF/Intel Joint Research Center for Computer Assisted Programming for Heterogeneous Architectures (CAPA). *(Corresponding author: Shail Dave.)*

Shail Dave, **Aviral Shrivastava**, and **Baoxin Li** are with the School of Computing Informatics and Decision Systems Engineering, Arizona State University, Tempe, AZ 85281 USA (e-mail: shail.dave@asu.edu; aviral.shrivastava@asu.edu; baoxin.li@asu.edu).

Riyadh Baghdadi is with the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: baghdadi@mit.edu).

Tony Nowatzki is with the School of Computer Science, University of California at Los Angeles, Los Angeles CA 90095 USA (e-mail: tjn@cs.ucla.edu).

Sasikanth Avancha is with the Parallel Computing Lab, Intel Labs, Bengaluru 560103, India (e-mail: sasikanth.avancha@intel.com).

Digital Object Identifier 10.1109/JPROC.2021.3098483

I. INTRODUCTION

Machine learning (ML) models implement intelligence in computing systems. Different ML models are widely used in several important domains, including computer vision (CV) (object classification [1]–[3] and

detection [4]–[6]), natural language processing (NLP) [7]–[9], media generation [10], recommendation systems [11], [12], medical diagnosis [13], large-scale scientific computing [14], embedded systems [15], mobile and edge processing [16], [17], and even for designing or optimizing hardware and software systems [18], [19]. Domain-customized accelerators can significantly speed up their execution in an energy-efficient manner [20]–[23]. However, the computational and memory requirements for processing these models have surged drastically [24]. Moreover, ML models can be deeper and larger, which improves learning accuracy, but significant redundancy may exist in these often overparameterized models [25], [26]. Therefore, recent techniques for efficient learning and inference have proposed compressing tensors of ML models. Tensors are compressed by inducing and leveraging: 1) sparsity (zero values in tensors) [27]–[31]; 2) size reduction (tensor decomposition, dimension reduction, and shape reduction) [3], [32]–[35]; and 3) quantization (precision lowering and leveraging value similarity) [27], [36]. With significantly lowered computational, storage, and communication requirements, efficient processing of *compressed tensors (sparse, size-reduced, and quantized)* offers notable acceleration and energy efficiency opportunities [37]–[40].

Hardware accelerators can efficiently process tensor computations of ML models. In particular, coarse-grain spatial architectures are a common choice for hardware accelerator designs. They contain an array of processing elements (PEs) with local registers/memory and shared memory. These accelerators feature interconnects, such as mesh or multicast for communicating data to PEs and reusing the data spatially, which reduces the access to the memory hierarchy. With simple PE designs and effective spatial and temporal managements of the data and computations, such architectures achieve high speedups and energy efficiency [20]–[22].

Special mechanisms are needed to exploit the acceleration benefits due to tensor sparsity, size reduction, and quantization. This is because, while hardware accelerators for ML can process low-precision tensors, they inherently cannot benefit from sparsity [41], [42]. They are designed for performing structured computations with regular memory accesses and communication patterns. Without special support for sparse tensors, they fetch all the data, including zero values from memory, and feed into PEs, thereby wasting the execution time. Sparsity, especially unstructured, induces irregularity in processing since nonzeros (NZs) or blocks of NZs are scattered across tensors. So, leveraging sparsity necessitates additional mechanisms to store, extract, communicate, compute, and load-balance the NZs and the corresponding hardware or software support. The goal of exploiting sparsity is to exploit all forms of sparsity possible to considerably reduce computation, communication, and storage of zeros while avoiding adding performance, power, and area overheads. Exploiting sparsity effectively depends on tailoring the data encoding and extraction, dataflow, memory banking structure, interconnect design, and write-back mechanisms. Furthermore, it requires new representations and enables new opportunities for hardware/software/model codesigns. In this survey, we mainly discuss different accelerator designs that have leveraged the sparsity of different tensors and different opportunities for performance gains and energy efficiency. Tensor decomposition and dimension reduction yield tensors of various sizes and asymmetric shapes [3], [43]. Dataflow mechanisms for executing layers of the models are typically optimized well for some commonly used layers (symmetric dimensions). They often become ill-suited for processing tensors with reduced dimensions [43] and different functionality. So, we describe how configurable designs and flexible dataflows can help to achieve efficient execution. Sparse tensors quantized with value sharing require additional support to index a dictionary for obtaining shared values. The survey also discusses how accelerators leverage value similarity across inputs, weights, or outputs and support variable bit-widths of sparse tensors.

A. Contributions

This article provides a comprehensive survey of different techniques for efficiently executing sparse and irregular tensor computations of the compact ML models on hardware accelerators. It describes corresponding enhancements in the hardware architecture and the required software support. Specifically, the following holds.

1) For inference and training of different ML models, we summarize various sources of the sparsity of tensors.
2) We highlight challenges in accelerating computations of sparse (especially unstructured) and irregular shaped tensors (e.g., dot product, convolution, and matrix multiplication) on spatial-architecture-based hardware accelerators that execute with dataflow mechanisms.
3) We present an overview of the accelerator system along with the different hardware/software modules for sparse and irregular computations, their interfacing, and the execution flow. We provide an in-depth discussion of the need of each module, different design choices, and qualitative analysis of the different choices.
4) We survey different accelerator systems and execution techniques for sparse tensors of ML models and provide taxonomies to categorize them based on the various hardware/software aspects of the designs.
5) We analyze how variations in sparsity and tensor shapes of different models impact the storage efficiency of different sparsity-encodings and the reuse of tensors.
6) For designing these accelerator modules and overall accelerator system, we discuss recent trends and outline further opportunities for hardware/software/model codesigns.

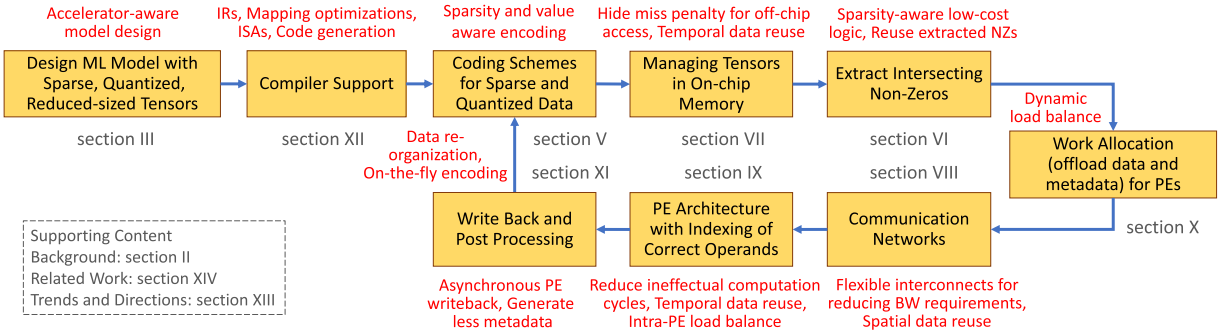


Fig. 1. ***Overview of the accelerator system for processing sparse and irregular tensor computations. (Section IV provides further discussion.)***

B. Article Organization

1) Section II provides a brief background on different ML models, hardware accelerators for their tensor computations, and the need for further efficiency by reducing computation, storage, and communication requirements.
2) Section III discusses tensor compression and opportunities due to sparse, size-reduced, and quantized tensors and why their efficient processing requires special support.
3) Section IV provides an overview of the accelerator system with enhanced architectural modules and software support for sparse and irregular tensor computations (see Fig. 1). It also presents a case study of accelerations of recent, sparse DNNs and analyzes execution bottlenecks. In-depth discussions of individual modules follow through Sections V–XII. Opportunities for optimizing each module further are discussed at the end of corresponding sections or subsections.
4) Section V illustrates common sparse data encodings, analyzes their implications in terms of storage and coding overheads, and describes the groupwise encoding of tensors.
5) Section VI discusses techniques for extracting matching NZs from tensors for computations. It analyzes the advantages and limitations of the centralized and in-PE extractions.
6) Section VII discusses managing noncoherent, multibanked, global scratchpads, and hiding the memory access latency behind computations. It also discusses data reuse of the sparse tensors and cross-layer reuse opportunities.
7) Section VIII discusses interconnect designs for distributing data from memory and reducing partial outputs, their bandwidth requirements, spatial data reuse, and their configurability to support multiple dataflows for execution.
8) Section IX describes sparsity-aware dataflows and pipelined PE architecture, including tailoring functional units for sparsity, bit-adaptive computing, and leveraging value similarity.
9) Section X discusses sources of the inter-PE and intra-PE imbalance due to sparsity and their impact, software-directed balancing, and hardware structures for dynamic balancing.
10) Section XI describes different write-back mechanisms for collecting data from PEs and assembling the data locally in PEs or on a central module. It also discusses data layout transformations and on-the-fly encoding of sparse outputs.
11) Section XII discusses compiler support for targeting hardware accelerators, including intermediate representations (IRs) for deep learning models, compiler optimizations and their automation, and ISAs and code generation for accelerators.
12) Section XIII describes recent trends and future directions in terms of developing tools and techniques for systematic exploration of hardware/software/model codesigns.
13) Section XIV discusses relevant surveys that describe additional details (domain-specific models, tensor compression techniques, and so on) and can be useful to readers.

II. BACKGROUND: NEED FOR EFFICIENT EXECUTION OF ML MODELS ON HARDWARE ACCELERATORS

A. Domain-Specific Machine Learning Models

Learning through ML models can be supervised (where labeled data are available), unsupervised (training samples are unlabeled), or semisupervised. We refer nonexpert readers to surveys [44]–[46] for a detailed discussion on different learning approaches and inference and training of various models. Discussions through this survey mainly focus on accelerating different *deep neural networks (DNNs)* that are commonly used for supervised learning.

Convolutional neural networks (CNNs) are used for object classification and detection in image processing, video analysis, and autonomous vehicle systems. CNNs majorly consist of many *convolution layers (CONV)* and a

few fully connected (FC) layers. Early CONV layers capture low-level features from the images (e.g., edges and corners), which are used for constructing high-level features (e.g., shapes) by subsequent layers. Finally, the classifier, also known as the FC layer, determines the type of the objects [44].

Sequence-to-sequence models include recurrent neural networks (RNNs), gated recurrent units (GRUs), long short-term memory (LSTM) [15], and attention mechanisms [7], [8]. These models are used for *NLP* and media processing tasks. They essentially use unidirectional or bidirectional recurrent cells at their core and process *multilayer perceptrons (MLPs)* also known as FC structures.

Models for semantic segmentation and language translation use encoder–decoder structures with convolutions [5], [14], recurrent cells, or attention layers [8], respectively.

Generative adversarial networks (GANs) [10] are used by media generation applications. GANs use generators and discriminative networks that consist of convolution layers.

Graph neural networks (GNNs) and other graph learning models [47] are used for applications, such as text classification and translation, and node classification and link predictions in large social graphs. They learn graph properties and infer about unforeseen information. To achieve this objective, each node contains an embedding feature vector with the information mixture about own and neighborhood features. The nodes then recurrently aggregate features of local neighbors, perform neural network computations on aggregated data (e.g., MLP for down-scaling embeddings), and update their embeddings.

Recommendation system models consist of embedding layers (look-ups and matrix operations) [12], CNNs for object detection and video understanding, and RNNs for processing language models [11].

Primitives, such as MLP or GEMM (general matrix multiply) and CONV, are at the core of many models and dominate the execution. So, ML frameworks, such as PyTorch [48], TensorFlow [49], and Intel MKL [50], provide efficient implementations of these primitives for execution on commodity hardware (CPUs, GPUs, and FPGAs) or even specialized accelerators. So, our discussions mainly focus on efficiently accelerating tensor computations of MLP, CONV, and RNN operators.

B. Hardware Accelerators for Machine Learning

In the "new golden age of computer architecture," recent research efforts and commercial solutions have extensively demonstrated that domain-customized hardware accelerators significantly speed up the execution of ML models in an energy-efficient way [20]–[22], [51]–[54]. Typically, these specialized solutions feature *spatial architectures*, which are those that expose low-level aspects

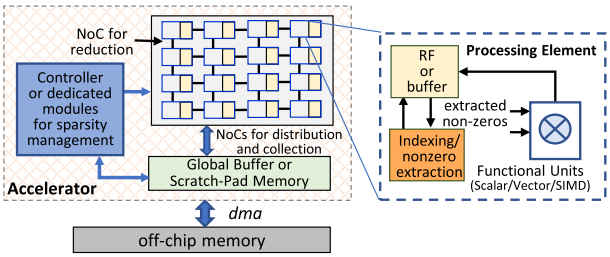


Fig. 2. ***Abstract accelerator design for processing sparse tensors of ML applications. Execution of applications requires explicit management of computational, communication, and memory resources.***

of the hardware's interconnect and storage to the hardware–software interface. Spatial architectures can be coarse-grained or fine-grained. Coarse-grained architectures feature arrays of interconnected PEs, and fine-grained designs are realized by programming FPGAs. *Coarse-grained spatial architectures* are a common implementation choice for designing hardware accelerators for ML [20]–[23], [55]. As illustrated in Fig. 2, the accelerator comprises an array of PEs that may contain private register files (RFs) and shared buffers or a scratchpad memory. PEs are simple in design (functional units with little local control), and the shared scratchpad is noncoherent with software-directed execution. Therefore, these accelerators are a few orders of magnitude more power-efficient than out-of-order CPU or GPU cores [20]–[22]. They lead to highly energy-efficient execution of ML models that are compute-intensive and memory-intensive. Performance-critical tensor computations of ML models are relatively simple operations, such as elementwise or tensor additions and multiplications. So, they can be processed efficiently with structured computations on the PE-array. Moreover, private and shared memories of PEs enable high temporal reuse of the data [56], [57]; with efficient data management, PEs can be continuously engaged in tensor computations, while the data are communicated via memories [20]. In addition, interconnects, such as mesh or multicast, enable data communication among PEs and spatial reuse of the data, lowering the access to off-chip memory. So, with minimized execution time, spatial-architecture-based hardware accelerators yield very high throughput and low latency for processing ML models.

C. Need for Further Efficient Execution

With recent advances in the development of ML models, their computational and memory requirements have increased drastically [18], [24]. Fig. 3 provides an overview of this dramatic surge. One major reason is the rise of deeper models. For example, for processing ImageNet images, AlexNet [1] contained five CONV and three FC layers (eight parameter layers) with the model size of 61 M parameters (weights and bias) and computation of 724 MFLOPs. DNNs, such as ResNet-101 [2], achieved

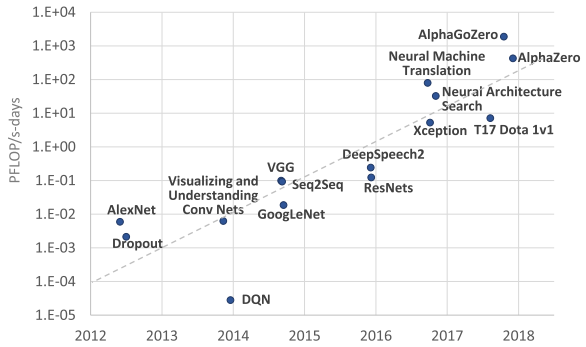


Fig. 3. ***Computation requirements for the training of AI algorithms almost double every few months. (Figure adopted from [24].)***

higher classification accuracy but contained 100+ parameter layers and required processing about 7.6 GFLOPs per image. *Memory requirements for NLP models have increased massively*, e.g., from 50 M–100 M parameters (Transformer [7], 2017) to 175 billion (GPT-3 [9], 2020).

While deeper and larger models achieve high efficiency for various tasks [44], they consume high execution time, energy, and memory. Previous studies showed that *significant data redundancy exists in these often overparameterized models* [25], [26]. So, researchers have developed techniques that compress tensors and obtain compact models, reducing computational, communication, and storage requirements significantly.

III. ACCELERATION OPPORTUNITIES DUE TO COMPACT MODELS AND THE NEED FOR SPECIAL SUPPORT

The efficiency of executing ML models can be improved further by drastically reducing computation, communication, and memory requirements. This can be achieved by compressing tensors of ML models. Tensors are compressed by inducing and leveraging: 1) sparsity (zero values) [27]–[30]; 2) size reduction (tensor decomposition, dimension reduction, and shape reduction) [3], [30], [32]–[35]; and 3) quantization (precision lowering and value similarity) [27], [36]. Previous techniques have achieved highly compact models without incurring accuracy loss. For example, after applying pruning, quantization, and the Huffman encoding, deep compression [27] reduced the model size of AlexNet and VGG-16 by 35× and 49× (e.g., from 552 to 11.3 MB), respectively. Accelerator-aware designs can compress the model further. For AlexNet and GoogLeNet models, Yang *et al.* [40] pruned 91% and 66% of weights and reduced computational requirements by 6.63× and 3.43×, respectively. ADMM-NN [38] applied weight pruning and quantization, thereby reducing the model size of AlexNet, VGG-16, and ResNet-50 (with up to 0.2% accuracy loss) by 99×, 66.5×, and 25.3×, respectively.

This section describes various sources of tensor sparsity, which are either inherent or induced by model architecture or regularization. It describes how sparsity reduces computations, storage, and communication requirements. It also discusses techniques for reducing the size and quantization of the tensors and how they offer advantages in terms of storage/performance/energy efficiency. Then, it describes how compression techniques may induce irregularity in the processing and why special support is needed for efficiently processing the compressed tensors on hardware accelerators.

A. Opportunities Due to Sparse Tensors

1) Sparsity Structures: Inherent sparsity is usually unstructured (e.g., of activations, gradients, or tensors of scientific computing applications), where NZ elements are randomly scattered [shaded elements in Fig. 4(a)]. Applying ReLU, dropout, quantization, or fine-grain pruning also induces unstructured sparsity in input activations (IAs) or weights (W). For improving execution efficiency, pruning techniques or model operators induce structured sparsity. For example, weights can be pruned in coarse-grain blocks where block shape can vary from 1-D (vector) to n-D for an n-dimension tensor [28], [37], [58], [59]. Fig. 4(b) shows 4 × 4 blocks for a block-sparse tensor, where each block contains all zeros or all NZs. With larger blocks, techniques often prune entire dimensions (e.g., channels or filters in CNN models) [28]. The selection of block size and shape depends on task accuracy requirements. Alternatively, tensors are sparsified with density bounded blocks [see Fig. 4(c)], where each n-D block contains a fixed (k) number of NZs (NNZs) [60]–[62]. It equally scatters NZs throughout the tensor. NZs are located arbitrarily in the whole block, or a fixed NNZs can be induced across each dimension of the block. Values of k can be selected based on the sensitivity of the pruning to accuracy. For example, analysis of [60] showed that, for VGG-16 and ResNet-50, about 12 out of 16 elements can be pruned without any accuracy loss, and about ten out of 16 elements for compact models, such as MobileNetV1 and SqueezeNetV1. To preserve accuracy while achieving high sparsity, a mixture of blocks (with different block sizes or sparsity) can also be introduced [63]. Finally, tensors can be pruned in patterns or

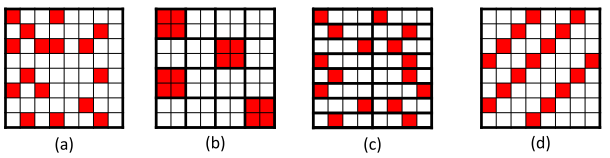


Fig. 4. ***Common sparsity structures (e.g., for a 75% sparse 8 × 8 matrix). (a) Unstructured. (b) Block sparse (coarse-grain), block size: 2 × 2. (c) Density-bounded (k:n) block sparse (fine-grain), block size: 1 × 4; k = 1. (d) Conditional or patterned.***

conditionally with sophisticated rules [e.g., diagonally, as shown in Fig. 4(d)].

2) Sources of Sparsity: Tensors of different ML models can be sparse due to multiple reasons.

1) CNNs use the ReLU activation function [1] that clamps negative values to zero. So, sparsity of IAs (IA-sparsity) can be 40% in CNNs, on average [64] and higher in later layers (about up to 70% [64], [65]). Cao *et al.* [64] reported that max-pooling can amplify it, e.g., up to 80% for VGG-16 layers. Lee *et al.* [66] showed that IA-sparsity eliminated about 40% and 55% of the multiply-and-accumulate (MAC) operations during CNN training and inference, respectively. For recent compact models, such as MobileNetV2 [32], IA-sparsity eliminates about 20% of the MACs.
2) Neural networks use dropout layers to avoid overfitting. After applying the dropout, only partial activations are retained [26]. Dropping the activations induces sparsity [26].
3) Pruning techniques remove unimportant weights and alleviate the overfitting of the model while maintaining the classification accuracy. Typically, weights with the least significant values can be safely pruned [25], [31] (in training or posttraining). Pruning can bring regularity in the learning of the model and can even increase accuracy slightly [37], [60]. Pruning algorithms introduce significant sparsity, e.g., more than 60% weights of CONV and more than 90% of the weights of FC layers can be removed [25] (W-sparsity). For recent compact models, such as MobileNetV2 and EfficientNetB0, W-sparsity can be from 50% to 93% [80%–85% in pointwise convolutions (PW-CONVs)] [67], which reduces MACs by 2.5×–4.2×. Similarly, more than 80% weights of RNN, GRU, or LSTMs can be pruned [39], [68], [69], especially for medium or large models, without significantly increasing error rate. For NLP models, Transformers [7] and BERT [8], recent techniques induce 80% [70] and 93% [71] W-sparsity, which reduces total MACs by about 4.8× and 12.3×, respectively. Besides, regularization of the models (e.g., L1 or group-lasso-based) can induce unstructured or structured W-sparsity [28].
   Pruning of activations is also shown as effective [72]–[76]. DasNet [73] reported eliminating about 27% and 12% MACs by activation sparsification for AlexNet and MobileNet. It achieved 79% IA-sparsity for AlexNet FC layers along with pruning 81% weights, without dropping top-1 accuracy. Similarly, MASR [77] refactored batch normalization, achieving about 60% IA-sparsity for RNNs. For attention-based NLP models, SpAtten [78] pruned unimportant tokens and heads. It reported reducing computations and DRAM accesses by up to 3.8× and 1.1×, respectively, without accuracy loss.
4) CNNs use Atrous (dilated) convolutions where filters are upsampled by inserting zeros between weights [5].
5) GANs use transposed convolution in a degenerator network, where input data is upscaled first by inserting zeros between values, and then, convolution is applied. For transposed convolutions in different GANs, about 60% MACs can be zero [79]. Additional sparsity is introduced when GANs are forced to forget generating specific objects [80].
6) Input data for object detection tasks can be inherently sparse, as only specific regions of frames are valid [81]. For example, object detection models of autonomous driving systems process 3-D LiDAR data by constructing point clouds and projecting them from the bird's eye view (top view) [82], [83]. The resultant images are then fed to object detection algorithms for locating the regions of interest. Recent techniques have reported that the sparsity of the input data for object detection can be 80% or more [81], [82].
7) For efficient communication in distributed training, *gradients* (Grad) are sparsified and compressed. For example, *Grad-sparsity* can be 99%+ for *CV* or language processing tasks [84] and 95%–99% for recommendation models [85].
8) Input data for the tasks of recommendation systems (e.g., user-item matrix) can be inherently highly sparse, e.g., from 95% [86] to 99% [11]. Recommendation models compute dot products on dense-sparse or sparse-sparse data [85], [87].
9) GNNs process large graphs, e.g., with thousands of vertices. Depending on the real-world interactions of objects (vertices), data contain high (e.g., 75%–99%) or hyper (99%+) unstructured sparsity [88], [89]. For example, in processing large graphs with GCNs, many features of vertices are local and lead to zeros in adjacency matrices for remote nodes [89]. GNN computations involve aggregation on sparse data and multiplications of dense matrices with dense or sparse matrices [89], [90], which are often processed on separate modules of the accelerator (e.g., in HyGCN [88] and EnGN [91]).
10) Text corpus in text analytics applications leads to high sparsity since each document contains only a fraction of the words from the vocabulary. Such analytics applications include PCA for dimensionality reduction of the sparse data, support vector machines and regression for classification, collaborative filtering for the recommendation, and k-means for clustering the data [30]. These operations involve multiplications of sparse matrices with dense or sparse vectors, where the matrix sparsity can vary from 67% to 99% [30].

While we describe leveraging sparsity for ML models, applications of many domains, including linear algebra, graph processing, and scientific computing [92], [93], can be accelerated by exploiting sparsity.

3) Advantages: Sparsity allows: 1) *eliminating ineffectual computations*, i.e., reduces execution time and energy by processing only NZs; 2) *reducing storage* by encoding only NZ values, so more data fits in on-chip memory and off-chip memory accesses (extremely energy-consuming [42], [94]) are reduced; and 3) improving speedup due to *reduced communication requirements* for data-intensive ML models.

B. Opportunities Due to Size-Reduced Tensors

Symmetric or high-dimensional tensors have large sizes and their processing requires more computation and memory. So, ML models are designed to reduce such requirements by using group or parallel operators [1], [95], 1×1 or PW-CONVs [33], [96], or dimensionality reduction with PCA [30], [34]. Moreover, tensors can be decomposed with spatial factorization [34], [97], depthwise separation for convolutions [3], [32], or low-rank approximations [34]. Furthermore, tensors can be ragged [35] to eliminate the need for structured or rectangular shapes. While these transformations significantly reduce storage and computations, they make tensors irregular shaped (asymmetric).

C. Opportunities Due to Quantized Tensors

Quantization includes precision lowering [36] and leveraging value similarity [27], [98], [99]. Precision lowering allows representing tensors (weights, activations, gradients, and weight updates) at much lower bit-width (e.g., 8 b or lower for inference and 8/16 b for learning). Moreover, elements with similar values can be clustered and approximated by sharing common values (centroids of clusters). Furthermore, similar values of outputs are reused with memoization (partially or the entire layer). In general, significant redundancy exists in tensor elements (particularly in the parameters of large models), and a successfully trained model is generalized and immune to noisy data. So, the error induced by quantization or approximation may often be tolerated by a well-trained model [100]. It can also obviate overfitting caused otherwise by excessive precision, thereby bringing generality in learning [101]. For compensating accuracy drop due to quantization, learning algorithms fine-tune the model or use quantization-aware training [36]. So, quantization or approximation techniques typically do not degrade inference accuracy [66] or trade it off for notable execution efficiency [64], [102], [103].

Quantization significantly reduces storage requirements and accesses to off-chip memory. It also *reduces area and power* since, for quantized tensors, functional units can be simpler and energy-efficient (e.g., int8 multiplier consumes $20\times$ less energy than FP32 multiplier [94] for a 45-nm process). Bus sizes can be smaller as bandwidth requirements are reduced.

So, with sparse, size-reduced, and quantized tensors, compact models can achieve higher accuracy as models with uncompressed tensors while becoming amenable for deployment at the edge, mobile, or online-learning platforms [17], [27] due to scope for low latency, energy, and storage. So, leveraging such opportunities is crucial for further accelerations.

D. Need for Special Support to Accelerate Sparse and Irregular Tensor Computations

Hardware accelerators efficiently process different models [21], [22], [104]. However, they inherently cannot benefit from the sparsity because all the data, including the zero values of activations, weights, and gradients, have to be fetched from memory and communicated to PEs; PEs are also unable to skip ineffectual computations, wasting the execution time. Sparsity, especially unstructured, induces irregularity in processing since NZs or blocks of NZs are scattered across the tensor. Therefore, leveraging sparsity necessitates additional mechanisms to store, extract, communicate, compute, and load-balance the NZs, and corresponding hardware and software support [41], [105]. Different sparsity levels and patterns from various sources lead to unique challenges and solutions in hardware/software codesign. Therefore, our discussions throughout this survey mainly focus on exploiting tensor sparsity for accelerating compact models.

Tensor dimension reduction and tensor decomposition make tensors irregular shaped (asymmetric), and they may also modify the functionality of the computational primitives, e.g., depthwise convolution (DW-CONV). Since execution on hardware accelerators is typically well-optimized for processing symmetric tensors with a specific dataflow mechanism, these shape transformations and supporting different functionality (e.g., DW-CONV, randomized, or approximated matrix multiply [106]) may introduce irregularity in processing requirements. To sustain high utilization of computational resources, it requires additional support including configurable hardware architectures and flexible mappings of the functionality onto architectural resources [43], [105], [107].

Hardware accelerators have supported low-precision tensors of fixed bit-widths and, even more recently, tensors with mixed precision [66]. However, when sparse tensors are quantized with value sharing, it requires indexing the codebook through indices for approximated elements [27]. Such irregular accesses are handled by implementing separate indirection tables in the pipelined hardware datapath [37], [42]. Moreover, value similarity is leveraged further by reusing computations with memoized outputs, which requires additional processing. Furthermore, supporting different bit-widths of various sparse tensors of different models requires configurable architectures for *bit-adaptive* computing [108]–[110].

To sum up, compressed tensors lead to sparse and irregular computations. Their efficient accelerations require special support, which is described in Section IV. The appendix describes that exploiting sparsity (especially unstructured) is relatively hard for execution on

Table 1 Accelerators for Processing Sparse Tensors

Objective	Techniques
Compressed data in off-chip memory (storage)	[20], [30], [37], [41]–[43], [60], [65], [66], [68], [93], [105], [107], [109], [111]–[124]
Compressed data in on-chip memory (storage)	[30], [37], [41]–[43], [60], [65], [66], [68], [72], [93], [105], [107], [111]–[119], [121], [123]–[125]
Skip processing zeros (energy efficiency)	[20], [30], [37], [41]–[43], [60], [65], [66], [68], [72], [93], [105], [107], [109], [112], [114]–[119], [121]–[134]
Reduce ineffectual computation cycles (performance & energy)	[30], [37], [41]–[43], [60], [65], [66], [68], [72], [93], [105], [107], [112]–[119], [121], [123]–[125], [129], [130], [134]
Load balancing (performance)	[37], [42], [43], [60], [65], [66], [68], [116], [118], [123], [126], [127], [130]

CPUs and GPUs; with special support, hardware accelerators can achieve notable gains.

IV. ACCELERATOR DESIGN FOR EFFICIENT SPARSE AND IRREGULAR TENSOR COMPUTATIONS

A. Overview

To efficiently process sparse and irregular tensor computations, designers of the accelerator systems can integrate special hardware or software modules. It enables orchestration of the structured computations while processing the tensors in compressed formats. Consequently, it can lead to efficient utilization of the accelerator resources and allows exploiting acceleration opportunities. Fig. 1 provides an overview of the accelerator system equipped with such modules. This section briefly describes these system modules.

Sparse, size-reduced, and quantized tensors of ML models offer various opportunities for storage, performance, and energy efficiency. Hence, several accelerators have provided marginal or comprehensive support and leveraged some or all the opportunities. Table 1 lists such common objectives and corresponding accelerator solutions that meet these objectives.

Different accelerators for inference and learning exploit W-sparsity, IA-sparsity, or both, which impacts acceleration gains [130]. Several accelerators, including Cambricon-X [41], exploit only static sparsity (see Table 2), e.g., when locations of zeros in weights are known beforehand for inference. Static sparsity allows off-line encoding and data transformations for arranging structured computations (e.g., for systolic arrays [62], [126], [136]). Recent accelerators, including ZENA [130], SNAP [107], and EyerissV2 [43], leverage dynamic sparsity also. It requires determining locations of intersecting NZs in both tensors at runtime to feed functional units, on-the-fly decoding (encoding) NZs, and often balancing computations on PEs. Table 2 lists different accelerators that support static and dynamic sparsity of tensors. Now, we describe different hardware and software aspects of the accelerator system that helps in leveraging sparsity effectively.

1) Sparsity Encodings: Sparse tensors are compressed using encodings, where only NZ values are stored in a "data" tensor and one or more "metadata" tensors encode locations of NZs. Section V discusses different formats and associated costs for encoding and decoding. For different sparsity levels, it analyzes their effectiveness in terms of storage efficiency. For example, tensors can be compressed by 1.8× and 2.8× for 50% and 70% sparsities (bitmap or RLC-2) and 7.6× and 55×–60× for 90% (RLC-4) and 99% sparsities (CSC or RLC-7). Structured sparsity (coarse-grain block-sparse) can alleviate the overheads of metadata and fine-grained data extraction by encoding indices for only large dense blocks. For accelerating ML models, sparse tensors are also quantized, i.e., their precisions are lowered (typically int8 or int16 for inference [43], [115], [130] and FP16 for learning [66], [105]) and often approximated by clustering data of similar values [37], [42], [111]. Therefore, encoded sparse data contain quantized values of NZs.

2) NZ Detection and Data Extraction: In processing sparse tensors of different primitives, corresponding elements of the weight and activation tensors are multiplied and accumulated. Depending on the sparsity, accelerators need to use data extraction logic that decodes compressed tensors, search within a window of NZs or index the buffer, and obtain matching pairs of NZs to feed the functional units for computation. Section VI provides a taxonomy of different data extraction mechanisms and analyzes their implications for various sparsity levels. Up to moderate IA-sparsity and high W-sparsity, these indexing or intersection-based mechanisms efficiently extract sufficient NZs at every cycle for keeping functional units engaged. For efficient compute-bounded executions at such sparsity, accelerators reported achieving near-ideal speedups (e.g., about 80%–97% of the speedup corresponding to reduced operations, i.e., *sparsity-speedup ratio*) [41], [42], [130]. However, extraction becomes challenging at high (e.g., 90%+) or hypersparsity as NZs are scattered at distant locations [89], and execution is usually memory-bounded with low arithmetic intensity. Section VI also discusses sharing of the data extraction mechanism among PEs or employing in PEs. Then, it discusses opportunities for further optimizations.

3) Memory Management: Compressed tensors are often stored in the shared on-chip memory that is noncoherent, multibanked, and often nonunified. For a predetermined sequence of execution, a controller or PEs initiates the accesses between off-chip and on-chip memory; their latency needs to be hidden behind computations on PEs. Section VII discusses corresponding memory architectures and techniques for hiding miss penalty for sparse tensors via double-buffering or asynchronous computation and memory accesses. It describes the data reuse opportunities for various sparsities and dimensions of tensors of common DNNs and how sparsity lowers the reuse. It also discusses

Table 2 Accelerator Systems Leveraging Sparsity of Different Tensors for Different ML Models

Dynamicity of Sparsity	Static	[41], [60], [62], [68], [113], [119], [122]–[124], [126], [127]	
	Dynamic	[20], [30], [37], [42], [43], [65], [66], [72], [78], [93], [105], [107], [109], [111], [112], [114]–[118], [121], [125], [128]–[133], [131], [135]	
Tensors Treated as Sparse	Weight	Unstructured	[41], [113], [122]–[124], [127], [136]
		Structured	[37], [58], [60]–[62], [68], [118], [126], [137]
	Activation	[20], [66], [72], [78], [111], [121], [125], [128], [133], [131], [135]	
	Both	[30], [37], [42], [43], [65], [93], [105], [107], [109], [112], [114]–[119], [129], [130], [132], [134]	
Primitive Operation	Matrix-Vector Multiply	[30], [37], [41]–[43], [60], [66], [93], [107], [114], [116], [119], [129], [133]	
	Matrix-Matrix Multiply	[41], [93], [105], [116], [126], [127], [132], [134]	
	Convolution	[20], [37], [41], [43], [60], [65], [66], [72], [107], [109], [111]–[118], [121]–[124], [129]	
	Recurrent / Attention Layer	[68], [77], [78], [125], [128], [131], [135], [137]	
Accelerators for Learning		[66], [105]	

techniques that leverage cross-layer reuse of intermediate output layers and reduce latency.

4) Communication Networks: Once tensor blocks are fetched from memory, they are distributed to appropriate PEs via interconnect networks (often one per operand). Efficient designs ensure that sufficient data can be fed to PEs, while they perform computations. Reuse is leveraged spatially by multicast or mesh networks that communicate common data blocks to multiple PEs. It lowers access to memory hierarchy and communication latency. However, spatial reuse opportunities vary depending on the sparsity, NZ extraction mechanism, and mapping of the functionality on the accelerator. Section VIII discusses different designs for distributing sparse and quantized tensors and reducing partial outputs. It also describes challenges in executing inter-PE communications that may become unstructured due to sparsity and the temporal and spatial mechanisms for reduction/collection of the outputs. It describes how configurable designs support various communication patterns for different sparsity, reuse, and functionality.

5) PE Architecture: Several accelerators consist of scalar PEs with fused MAC units (e.g., EIE [42], LNPU [66], and Envision [109]). Others contain SIMD PEs (multiple functional units) (e.g., EyerissV2 [43]) or vector PEs consisting of multiplier-arrays and adder trees (e.g., Cambricon-X [41] and SNAP [107]). PE architectures either directly process pairs of matching NZs extracted from tensors or use hardware logic for data extraction or coordinate computation (see Fig. 2). Effectively utilizing functional units can be challenging for variations in sparsity, precisions, and functionality, and it may require configurable designs. Section IX provides corresponding discussions and describes sparsity-aware dataflow mechanisms (mapping of tensor computations on accelerator resources) used by different accelerators. It also describes how accelerators have leveraged value similarity of tensors and the corresponding modifications in the PE architecture.

6) Load Balancing: Depending on the distribution of zeros, the execution may end up with processing a different amount of NZs on different PEs or their functional units, which creates inter-PE or intra-PE load imbalance. Section X analyzes such sources of the imbalance and introduces a taxonomy of different load balancing techniques. Accelerators achieve load balance through either software techniques (e.g., structured pruning or data reorganization) or by providing a hardware module for dynamic work balance (through asynchronous execution or work sharing), which provides further accelerations. For example, ZENA [130] leveraged the sparsity of both activation and weight tensors for AlexNet and VGG-16 models and reported about 32% additional performance gains through load balancing. Dynamic load balancing can provide notable speedups for high, unstructured sparsity [89].

7) Write-Back and Postprocessing: Tensor elements produced by PEs need to be collected, postprocessed for further operations, and written back to the memory. PEs in different accelerators either write back sequentially or asynchronously through a shared bus or via point-to-point links. In addition, accelerators usually contain a postprocessing unit that reorganizes the data (as per the dataflow mechanism of the current and next layer of the model) and encodes sparse output on the fly. Section XI discusses such mechanisms.

8) Compilation Support: It is important to support the execution of various ML models on accelerators and easier programming of models from ML libraries. Section XII discusses compiler support for sparse models and hardware accelerators. It discusses polyhedral and nonpolyhedral IRs and their implications on the compiler's ability to represent the code and apply code transformations. It describes challenges in supporting sparse tensors and DNN compilers that facilitate sparse tensor computations. Then, it discusses compiler optimizations, including common loop optimizations and those specific to hardware intrinsics. It also describes semiautomatic optimizations for transforming the loops, data layout, and automatic optimizations using cost models. Finally, it discusses ISAs used by accelerators and their code generation by using libraries of high-level primitives.

B. Case Study: Acceleration of DNNs and Bottleneck Analysis

This section analyzes the sparsity of recent DNN models (for NLP and CV) and the acceleration that can be achieved with some of the popular accelerator-alike architectures.

Table 3 Sparsity of Some Popular DNNs

Model	Domain	Dataset	GOps (dense)	Sparsity %			Sparse Model
				IA	W	Ops	
MobileNetV2 [32]	CV	ImageNet	0.3	34	52	81	[67]
EfficientNetB0 [124]	CV	ImageNet	0.5	0	68	60	[67]
Transformer [7]	NLP	WMT En-De	4.6	0	79	79	[70]
BERT-base-uncased [8]	NLP	SQuAD	9.3	0	92	92	[71]

1) DNN Models: Table 3 summarizes analyzed DNN models and their overall sparsity across all CONV, GEMM, and DW-CONV operations. For each of these DNN operations, W-sparsity was obtained from sparse DNN models (listed in the last column). IA-sparsity was obtained by performing inference with sample data (images and text sequences).

2) Accelerators: Table 4 summarizes analyzed accelerators and their sparsity-centered features. Their architectures targeted unstructured or block-sparse sparsity of activations and/or weights. Their features represent variations across data encoding, data extraction, vector processing, memory hierarchy, NoC, and load balancing.

3) Methodology: To determine the impact of sparsity on achievable acceleration, we performed a data-driven analysis of the execution latency. For each DNN layer, zeros (or blocks of zeros) were induced randomly according to the sparsity of its tensors. The overall execution time was determined from the latency of processing on functional units, data decoding, extraction of NZs, work synchronization, and off-chip memory transfers, which were calculated based on analytical modeling of the microarchitectural features. The speedups were calculated over oracle processing of dense tensors at the accelerator's peak utilization of computational resources and off-chip bandwidth. In this study, we do not consider the processing of DW-CONV on these accelerators since they are often not pruned, and their execution needs to be groupwise, which is extremely inefficient. Such unsupported performance-critical operators were assumed to be processed with dense tensors at peak utilization of hardware resources.

4) Analysis: Fig. 5(a) shows speedups of accelerators for targeted DNN models, for leveraging the sparsity of supported DNN operators. It illustrates speedups for: 1) reduction in the operations due to sparsity (desired); 2) peak utilization of accelerator's computational resources and off-chip bandwidth while leveraging sparsity, over such oracle processing of dense tensors (potential); and 3) actual processing on accelerator over oracle processing of dense tensors (obtained). For understanding implications of execution overheads including those incurred by metadata processing and load imbalance, Fig. 5(b) illustrates fractions for desired computation time and execution overheads in a stacked format. The overheads were extracted for layerwise processing and then accumulated to determine the overall impact. Fractions include the following.

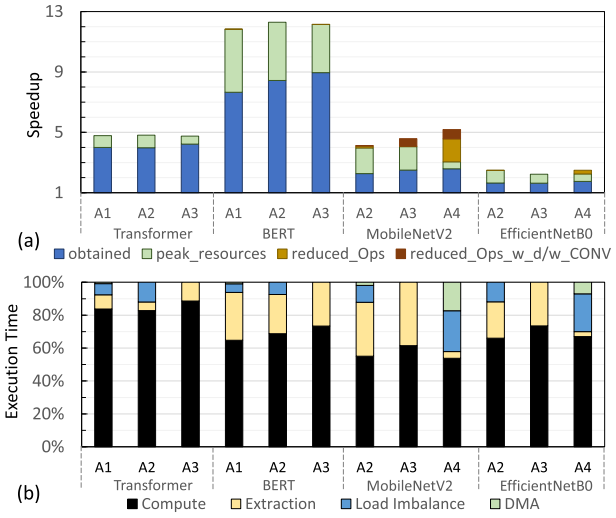


Fig. 5. ***(a) Obtained speedups for accelerators listed in Table 4. (b) Analysis of execution time overheads for obtained accelerations.***

1) *Computation time:* Minimum execution time required for processing at peak on accelerator's functional units.
2) *NZ extraction:* Time required for decoding NZs from communicated operands and extracting matching operands for feeding the functional units. It also corresponds to balanced computations.
3) *Load imbalance:* Time required for on-chip processing on the accelerator, considering the imbalanced computations subjected to the accelerator's work synchronization and work-sharing schemes.
4) *DMA time*: Time required for off-chip data communication via DMA transfers, in addition to on-chip processing.

Table 4 Architectural Features of Analyzed Accelerators for Sparse DNNs

ID	Reference Architecture	Supported Operators	Sparsity Leveraged		Non-zero Data Extraction			PE Architecture	Work Synchronization	Freq. (GHz)	DRAM BW (GBPS)	Bit-width			
												data		metadata	
			IA	W	Encoding	Discovery	Loc.	FU				IA / O	W	IA	W
A1	EIE [42]	GEMM	unstructured		CSR	Indexing	in-PE	Scalar	Prefetch	0.8	256	16	4	N/A	4
A2	Cambricon-X [41]	CONV, GEMM	dense	unstructured	COO-1D		central (per PE)	Vector (16 multipliers & adder tree)	Every Output Activation	1	256	16	16	N/A	8
A3	Cambricon-S [37]		unstructured	block-sparse	Bitmap	Intersection	central, shared			1	256	16	8	1	1
A4	ZENA-IA-W [130]	CONV	unstructured				in-PE	Scalar	Intra-Workgroup	0.2	12	16	16	1	1

Fig. 5(a) shows that accelerators efficiently exploited moderate sparsity. For example, for 4.8× reductions in operations of Transformer due to W-sparsity, they achieved about 4×–4.2× speedups. The exploitation of speedup lowers when activations are dense and weights are highly or hypersparse. This is because accelerators, such as EIE and Cambricon-X, broadcast activations to PEs and extract matching pairs corresponding to NZ weights. So, communication of activations and extraction of matching NZ operands consume significant execution time, while there are fewer operations to feed the functional units [see Fig. 5(b)]. For example, for BERT-base-uncased [8] (92% sparse weights [71]) on SQuAD [138], they achieved about 7.7×–8.9× speedups out of 12.2× speedup for processing at peak. Due to block-sparse weights, computations on PEs of Cambricon-S are always balanced. Therefore, it achieved higher speedups. By using blocks of 16 × 16 or even 1 × 16 (across input and output channels) for pruning, inducing similar sparsity is not possible sometimes. So, the reduction in operations and potential for the speedup was slightly lower for Cambricon-S (e.g., for EfficientNetB0). In general, due to high DRAM bandwidth, overheads incurred by DMA transfers were hidden (for Cambricon-X/S) or negligible for noninterleaved transfers (e.g., for EIE).

Fig. 5(a) also shows that Cambricon-S and ZENA-IA-W achieved higher speedups for CV models by leveraging unstructured sparsity of activations. High IA-sparsity amplified total sparsity during processing several layers (e.g., MobileNetV2), incurring considerable excess processing in data extraction for Cambricon-X/S and in load imbalance for ZENA-IA-W. With zero-aware static sorting of filters and dynamic load balance, ZENA [130] could overcome such imbalance. However, it would suffer through high on-chip communication time since it used only one shared bus for multicast via NoC and collecting outputs. We disregarded such communication overhead for ZENA-IA-W in this study, as most accelerators use separate NoCs or buses for alleviating communication overheads. Also, due to low DRAM bandwidth, overheads incurred by DMA transfers were higher for ZENA-IA-W, mainly for executing DW-CONVs with dense tensors.

V. ENCODINGS FOR COMPRESSING SPARSE TENSORS

A sparse tensor is compressed with an encoding format. An encoded tensor contains actual *data* (NZ values) and *metadata* (information about positions of NZs). Later, metadata are used by an accelerator's data indexing logic to locate and extract NZs. This section discusses commonly used encodings through an example (see Fig. 7) and their implications on the storage and processing requirements. For different formats, Fig. 6 introduces a taxonomy for processing metadata during data extraction, and Table 5 lists the corresponding storage overhead. Depending on the mapping of a layer onto the accelerator, tensors are divided into blocks (per PE-wise work) that are encoded separately. We refer to such processing as a *groupwise encoding*, which is discussed later. Finally, this section briefly describes encoding on the fly and further opportunities.

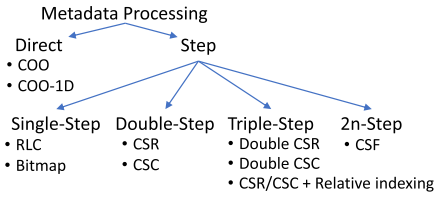


Fig. 6. ***Taxonomy for the required processing on the metadata during data extraction when a sparse tensor is encoded using different formats.***

A. Encoding Formats and Implications

1) Coordinate (COO): It stores absolute positions of NZs. As shown in Fig. 7(b), all NZs of an uncompressed tensor T are stored in a data vector val, and vectors coord_y and coord_x indicate the coordinates of each NZ value. So, COO is a natural way to express sparse tensors and is used commonly (e.g., in PyTorch). Formats adopted by FROSTT [140] and matrix market [141] closely resemble COO.

The COO format stores all coordinates in the uncompressed format. For example, as shown in Fig. 7(b), the metadata for values "2" and "3" (same row) or "2" and "5" (same column) are not compressed, i.e., duplicate values of row and column indices exist in coordinate vectors. So, the overhead of storing n coordinates per NZ value is about $\sum_1^n \lceil \log_2 d_i \rceil$ bits (vector d contains the tensor's dimensions). It makes COO inefficient for storing tensors with low or moderate sparsity.

Fig. 8 shows storage benefits for encoding 2-MB matrices of various sparsities in different formats. We calculated storage requirements with the analysis presented in Table 5 and normalized them to the matrix's size in dense format. We used the Scipy library [142] to generate matrices of various sparsities and encode them in COO, CSR, and CSC. Fig. 8 shows that, for a 2-MB matrix, COO achieves storage efficiency for 70%+ sparsity. However, COO may yield simple indexing logic, as both the data and metadata can be directly extracted.

Table 5 Storage Overhead for Common Encodings. Vector d Stores n Dimensions of a Tensor That Contains NNZ NZ Elements

Format	Storage Overhead (bits)
COO	$NNZ \times \sum_1^n \lceil \log_2 d_i \rceil$
COO-1D	$NNZ \times \lceil \log_2 \prod_1^n d_i \rceil$
RLC	$NNZ \times B$
Bitmap	$\prod_1^n d_i$
CSR	$NNZ \times \lceil \log_2 d_1 \rceil + (d_0 + 1) \times \lfloor \log_2 NNZ + 1 \rfloor$
CSC	$NNZ \times \lceil \log_2 d_0 \rceil + (d_1 + 1) \times \lfloor \log_2 NNZ + 1 \rfloor$

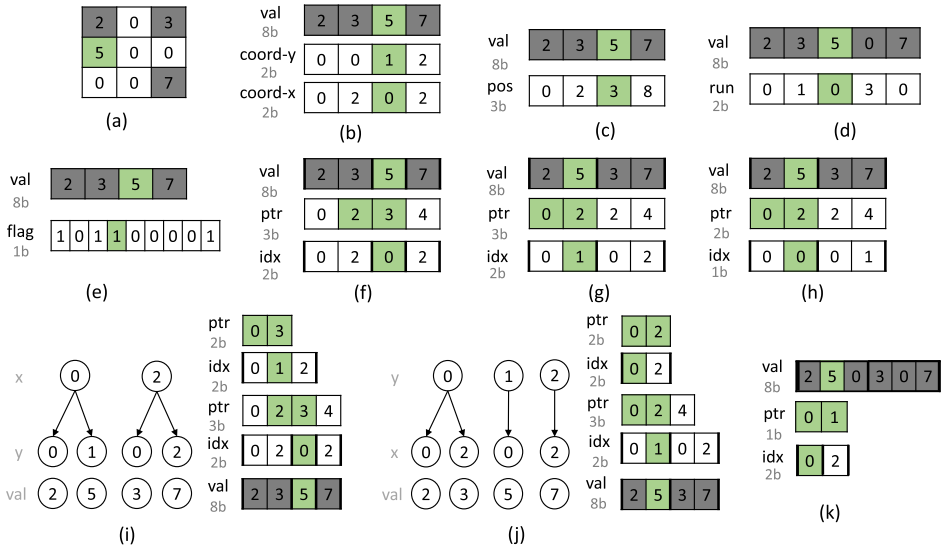


Fig. 7. ***Encodings to store sparse tensors in different formats. Elements with green shade encode the same NZ element. (Figure inspired from [139].) (a) 2D Tensor T (uncompressed). (b) COO. (c) COO-1D. (d) RLC. (e) Bitmap. (f) CSR. (g) CSC. (h) CSC (relative encoding of row idx). (i) CSF (mode-0 tree; y-x compression). (j) CSF (mode-1 tree; y-x compression). (k) Block CSR (block size: 3 × 1).***

2) COO-1D: For *tilewise processing* of an encoded tensor, accelerators often process only a block of NZs at a time, where block elements vary across only a single dimension. For example, Cnvlutin [72] processes the IAs and weights across the channel direction. Therefore, the data block is encoded with COO-1D, which is just like COO, but there is only one pos vector for storing coordinates of NZs in the flattened block. For instance, if we flatten T and consider it a block, then the value "5" is indexed by position "3."

3) Run-Length Coding (RLC): It compresses a sequence of values by replacing consecutive duplicate values with a single value and the number of repetitions (also known as *run*). For RLC-encoded sparse tensor, "run" indicates a total number of zeros before (after) an NZ. Fig. 7(d) shows RLC encoding of T. Run values for "2" and "3" are "0" and "1," respectively. A few accelerators, including Eyeriss [20], encode both the NZs and run altogether in the same vector val. For example, T can be encoded as val: (0, 2, 1, 3, 0, 5, 4, 7).

RLC requires a *step processing* on metadata, as run length needs to be calculated by accumulating runs and preceding *NNZs*, for determining the position of an NZ. The storage overhead for RLC-B is NNZ $\times$ B bits, where B is the bit-width of the run. If a vector d contains tensor dimensions, then B can be set as up to $\lceil \log_2 (\prod_1^n d_i) \rceil$ bits for accommodating the number of leading zeros in a highly sparse tensor. When B is set lower, it cannot always capture the number of zeros as *run*. Fig. 7(d) shows RLC-2b encoding, where leading zeros before "7" are four. This cannot be expressed in 2 bits. As a work-around, padding zeros [42] are inserted and treated as NZs. In this example, a padding zero is inserted between "5" and "7"; run values corresponding to the padding zero and "7" are "3" and "0," which contributes to the total run of four.

To accelerate CNNs with 30%–90% sparsity of tensors, designers have set B as two or four bits. In general, setting the B as $\lfloor \log_2 (\text{sparsity}/\text{density}) \rfloor + 1$ bits can effectively compress tensors and provide a feasible bit-width to indicate leading zeros. Here, sparsity and density are fractional numbers indicating the actual or anticipated number of zeros and NZs in the tensor, respectively. So, setting the B as 1, 1, 1, 2, 4, and 7 efficiently encodes tensors with sparsity of 10%, 30%, 50%, 70%, 90%, and 99%, which is depicted in Fig. 8.

As RLC requires step processing on metadata, the indexing logic needs an accumulator to determine the position of an NZ. When an encoded tensor is not processed blockwise but rather indexed by n-dimensions, the indexing logic may require performing division and modulo operations on the metadata. Alternatively, a multidimension representation can be used, where run for the coordinates of each

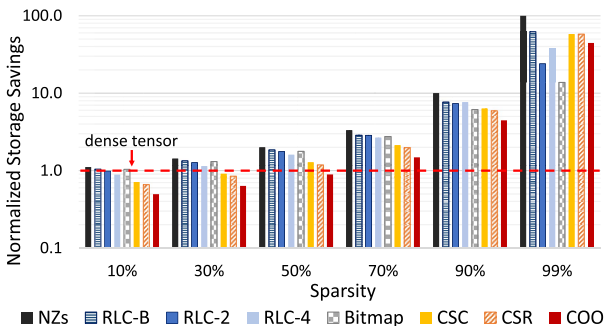


Fig. 8. ***Storage benefits for encoding a sparse tensor (512 × 2048 matrix with 16b elements) in different formats, normalized to the size of the fully dense tensor. (Figure inspired from [105].)***

dimension can be calculated separately and stored. The overall computational cost (arithmetic and logical operations realized in hardware) for such step processing can be low. Therefore, several accelerator designs, including Eyeriss [20] and SCNN [115], used RLC or its variant. As run indicates repetition of a value, CompAct [111] used an enhanced RLC format for encoding both the sparse and similar-value activations.

4) Bitmap: It stores all NZs in a tensor val along with a tensor *flag* that contains 1-bit flags for all elements of an uncompressed tensor T. As shown in Fig. 7(e), a flag indicates whether an element is NZ or not. Storage overhead for the bitmap (also known as bit-mask) is $\prod_1^n d_i$ bits (where vector d stores n dimensions of T) [121]. Since bitmap stores metadata for all elements, it is effective for compressing the tensors of low or moderate sparsity. Like RLC, decoding or indexing bitmap also requires step processing. The indexing logic to locate an NZ typically consists of at least an adder and a comparator [41]. Due to moderate storage overhead and low encoding/decoding cost, several accelerators used bitmap, including Cambricon-X [41], SparTen [116], and SIGMA [105], as shown in Table 6.

5) Compressed Sparse Row (CSR): It compresses a matrix by processing each row as a sparse vector. In a CSR-coded tensor, an array val contains all NZ values (ordered rowwise), and an array idx stores their column indices [143]. Array ptr stores information about total NZs in each row i, which is obtained by calculating ptr$[i+1]$ - ptr$[i]$. The last element of ptr contains the total NNZs in T. Rowwise compression enables random accesses to any row.

While COO redundantly stores row coordinates of NZs in the same row, CSR compresses such metadata by storing NZs rowwise [139]. For example, in Fig. 7(b) (COO), coord-y stores row indices "0" and "0" for NZs "2" and "3." This redundancy is removed in the CSR coding of Fig. 7(f), as ptr stores only total NZs in each row. For compressing an $M \times N$ matrix using CSR, the total storage overhead is NNZ $\times \lceil \log_2 N \rceil$ (for idx) $+ (M+1) \times \lfloor \log_2 \text{NNZ} + 1 \rfloor$ (for ptr). Due to high storage overhead (proportional to NNZs and size of the row), CSR coding is efficient at high sparsity [41], [105], e.g., 90% or higher (see Fig. 8).

Decoding a CSR-encoded tensor can require a two-step processing of metadata. The first step locates NZs of a row by iterating over ptr, and the next step locates an NZ element in the NZs of the row through the column index. Accelerators efficiently process CSR-coded matrices rowwise such that ptr is accessed once for fetching each row, and then, the decoder iterates through idx (to locate column positions).

CSR variants can improve efficiency further. For example, ptr stores duplicate values when consecutive rows are zero. Doubly CSR (DCSR) [144] eliminates this redundancy and achieves additional compression for hypersparse matrices. Block CSR (BCSR) [145] stores a block of elements in val if the block contains at least one NZ. As shown in Fig. 7(k), in BCSR, idx indicates the column index of a block, and ptr informs about the number of dense blocks located in the same rows. BCSR avoids storing blocks of all zeros and populates dense regions, hence suitable for encoding block-sparse structured weight tensors. So, BCSR-coded tensors can be efficiently executed not only on conventional processors but also on hardware accelerators (with additional support for appropriately indexing dense regions, e.g., [126]).

6) Compressed Sparse Column (CSC): CSC is similar to CSR, except that NZs are stored columnwise [143]. As shown in Fig. 7(g), an array val contains NZs (organized columnwise); idx stores their row indices; and ptr informs about the total NZs in each column. The storage overhead and hardware costs for encoding/decoding tensors in CSC format are similar to those for CSR. Accelerators, including EIE [42] and Sticker [117], processed high-sparsity tensors with CSC format.

For alleviating the high storage overhead of CSR or CSC formats due to storing idx and ptr arrays, a few accelerators further encode the metadata idx or ptr. For example, EIE [42] and EyerissV2 [43] encode idx in RLC such that elements in idx indicate zeros between column indices of NZs (similar to run in RLC for NZ values). Fig. 7(h) shows CSC encoding with such an RLC-encoded row index array. Values "2" and "5" have column indices "0" and "1," respectively, which can be encoded as "0" and "0" since there are no leading zeros before NZs "2" and "5." Similarly, if the first column of T is (0, 2, 0, 0, 5), then the row indices for "2" and "5" can be encoded as "1" and "2." ptr can also be encoded likewise (store NZs per column instead of a cumulative number). However, encoding positions relatively requires additional step processing on the metadata. Therefore, decoding a CSR or CSC encoded matrix with RLC-encoded metadata can require triple-step processing on metadata (additional hardware cost).

7) Compressed Sparse Fiber (CSF): CSF [146] provides a generalization of CSR for higher-order (n-dimensional) tensors by forming a tree (with n levels). Nodes at level l contain indices for the lth mode (dimension) of an uncompressed tensor T. The path from a root to a leaf node encodes different coordinates of an NZ, which are stored in the nodes throughout the path; each leaf node stores an NZ value. So, the height of the tree is the total dimensions of T; the width is NNZs in T.

Fig. 7(i) illustrates a mode-0 tree and corresponding arrays of index pointers. Root nodes represent the major mode (0 or y), and their child nodes represent the consecutive dimension (1 or x). Like in CSR, ptr informs about a group of indices corresponding to a dimension. For instance, ptr array at the beginning informs that one group of three coordinates corresponds to the mode 0 (idx stores coordinates). Similarly, the next ptr array informs about three different groups of coordinates for the next mode (dimension 1). The corresponding idx array stores

Table 6 Commonly Used Sparsity Encodings by Accelerators

COO	[93], [117]
COO-1D	[60], [72], [107], [114], [124], [125], [129], [132]
RLC	[20], [65], [66], [78], [111], [113], [115], [149]
Bitmap	[37], [41], [62], [105], [109], [111], [112], [116]–[118], [120], [121], [130]
CSR	[30]
CSC	[30], [42], [43], [68], [119], [123], [150]
CSF	[93]

the coordinates for mode 1, separated into three groups (marked by thick outer vertical borders).

Layering the arrays of index pointers reduces duplication of indices [147]. Each time when a node directs to children, it eliminates duplicating indices for the corresponding mode. Storage benefits increase with the increase in dimensions and redundancy among coordinates of NZs. The organization of the data also impacts storage efficiency. For example, Fig. 7(j) shows another ordering, which eliminates storing redundant coordinates of column (mode 1), achieving fewer nodes. For an n-mode CSF tensor, the storage overhead corresponds to more than NNZ + $n - 1$ coordinates and typically much less than $n \times$ NNZ coordinates. Works [146], [147] provide further details about managing higher order tensors with CSF format. Processing metadata at each dimension requires two-step processing (just like processing ptr and idx in CSR), thereby up to 2n-step processing for an n-dimensional tensor. So, accelerator designers may opt for CSF format when processing high-dimensional tensors with high sparsity.

8) Huffman Coding: It typically is applied for compressing sparse tensors once they are quantized using precision lowering or value sharing. After quantization, values of the reduced range appear with different frequencies and can be compressed further with the Huffman encoding [27], for example, deep compression [27] pruned and quantized weights of AlexNet [1] and VGG-16 [148], achieving 8/5-b indices with a codebook of 256/32 weights for CONV/FC layers. With the Huffman encoding, it compressed the models further by 22% and 36% (total compression of 35× and 49×).

9) Encodings for Tensors With Structured Sparsity: Density-bounded blocks [see Fig. 4(c)] can be encoded similarly as blocks with unstructured sparsity, e.g., with bitmap [62], COO-1D [60], or RLC. So, for the same sparsity and block size, the overhead is similar to tilewise processing of a tensor with unstructured sparsity. It is usually low for small block sizes (e.g., 8 × 1 [62] and 1 × 4 in NVIDIA A100 Tensor Core GPU [61]) since the position of each NZ is indicated by a few bits. Coarse-grain block-sparse tensors [see Fig. 4(b)] can be encoded at block-granularity, which can significantly reduce the metadata size (almost eliminated for dimensional pruning [151]). Cambricon-S [37] used bitmap to indicate the presence of each 1 × 16 dense block with a single bit. Similarly, ERIDANUS [126] used few bytes to process each 8 × 8 dense block on systolic arrays. Such encodings require indicating the position of a dense block across rows or columns and additional indices for higher dimensions that indicate dense blocks packed per dimension, e.g., in BCSR [see Fig. 7(k)].

10) Other Formats: Various encoding formats have been proposed, which improves the compression or efficiently access sparse tensors during execution on CPUs/GPUs (for high-performance and scientific computing). It includes compressed sparse blocks (CSBs) [152], libsvm [153], ELLPACK [154], diagonal (DIA) [155], dynamic CSR [156], delta-coded CSR [157], and mode-generic and mode-specific formats [158]. Prior works, including [139], SPARSKIT [143], and [147] and [159]–[161], surveyed them along with additional formats and discussed their implications. Different libraries that provide support for encoding the tensors and sparse tensor computations on CPUs or GPUs include MATLAB tensor toolbox [162], Intel MKL [50], SciPy [142], and cuSPARSE [163].

B. Groupwise Encoding

One way of processing sparse tensors is to encode the whole tensor. Then, the accelerator's data management logic extracts an appropriate tile (optionally decodes it) and communicates to the PEs. In contrast, for groupwise encoding, tensor tiles are encoded separately, based on predetermined per-PE work. Depending on the mapping, each tile is typically communicated to a unique PE (or a PE-group) during execution. So, the encoding considers the dataflow, i.e., mapping of the tensor computations onto PEs. It can make the decoding and data extraction easier, as each group corresponds to execution on a distinct PE (or a PE-group). EIE [42], Cambricon-X [41], and CompAct [111] used groupwise encoding.

C. On-the-Fly Encoding

Accelerator designers often target only static sparsity of weights and encode them off-line, e.g., DNN inference accelerators, including EIE [42], Cambricon-X [41], and [113]. However, on-the-fly encoding is required for efficiently processing dynamically sparsified tensors (sparse activations in the inference and tensors in training the models). Therefore, accelerators, such as CompAct [111], SCNN [115], NullHop [121], Cnvlutin [72], and Sticker [164], employ an on-the-fly encoder. Typically, before encoding a tensor, the data are reorganized as per requirements of the groupwise encoding and dataflow mechanism for processing the subsequent layer. So, on-the-fly encoding is often combined with assembling the outputs from PEs (Section XI-D provides further details).

D. Optimization Opportunities

Tailoring encoding formats for sparsity levels and patterns: Various layers of deep learning models exhibit a wide range of sparsity (*interlayer and intratensor*

Table 7 Classification of NZ Data Extraction Techniques

Target Sparsity	PE Architecture	Functional Unit Operation	Accelerators
One Tensor	Scalar	MAC	[30], [68], [113], [121]
	SIMD/ Vector	Sc-Vec-Mul	[60], [66], [72]
		Vec-Vec-Mul	[41], [60], [123]
Both Tensors	Scalar	MAC	[30], [42], [93], [114], [116], [119], [130]
	SIMD/ Vector	Sc-Vec-Mul	[43], [112]
		Vec-Vec-Mul	[37], [105], [107]

Location of Extraction Units	Accelerators
Centralized/ Shared	[30], [37], [41], [42], [65], [66], [105], [107], [112], [119], [121]
In-PE	[42], [43], [60], [68], [72], [93], [113]–[116], [118], [123], [130], [134]

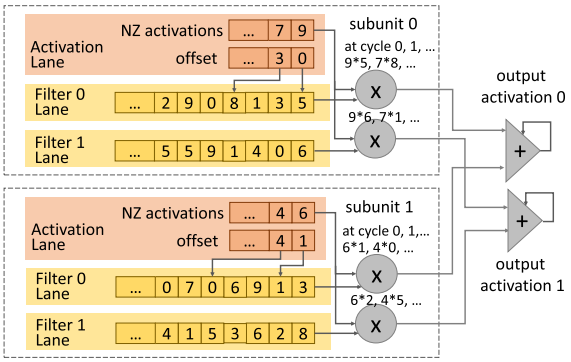


Fig. 9. ***Data extraction in subunits of Cnvlutin PE. (Figure adopted from [72].)***

sparsity variation). Moreover, even within a DNN layer, sparsity among tensors can be different (*intralayer and intertensor sparsity variation*). Accelerators need to support such sparsity variations effectively without incurring significant overheads for storage, encoding, and indexing. When the sparsity range or pattern of multiple tensors is diverse, designers can opt for the separate encoding of different tensors (e.g., [117]). These different sparsity-encodings can be utilized for off-chip storage, zero-guarding the PEs, or reducing the latency of on-chip extraction to locate intersecting NZs. When different formats are used for performance gains, the accelerator should provide hardware logic for decoding different tensors that are stored in different formats (and support for any on-the-fly encoding). Such decoding logic may use existing data extraction mechanisms, but it will require separate/configurable decoding logic for supporting multiple formats.

VI. EXTRACTION OF MATCHING DATA FOR COMPUTATIONS ON NONZEROS

Tensors are typically stored in the compressed format in the accelerator's memory. Therefore, locations of NZs that need to be processed are determined from the metadata. Once a matching pair is extracted (elements of two tensors that need to be added or multiplied), a PE can proceed for computations. Identifying effective NZs is the primary step toward eliminating ineffectual computations due to the sparsity of weights and/or activations. This section describes different data extraction mechanisms (Table 7 provides a taxonomy), their management in PEs or centrally, and their tradeoffs. Then, it discusses further acceleration opportunities to exploit various sparsity levels.

A. Nonzero Detection and Extraction Mechanisms

A data extraction mechanism needs to feed functional units of PEs every cycle. So, based on their processing of scalars or vectors of NZs, Table 7 categorizes extraction mechanisms for: 1) MAC operation on scalars; 2) scalar–vector multiplication; and 3) vector–vector multiplication.

1) Indexing Dense Tensors by Indices of NZs of a Sparse Tensor: Depending on sparsity, only one tensor may be treated as sparse and compressed (e.g., activations for Cnvlutin [72] or weights for Cambricon-X [41] and NVIDIA A100 [61]). So, the position of an NZ can be used for indexing the other (i.e., dense) tensor to extract the corresponding value.

a) MAC: Consider the activation lane and filter lane 0 of subunit 0 in Fig. 9, which can be visualized as processing on a scalar PE. For an NZ streaming from the activation lane, matching weight can be looked up and provided to the multiplier or MAC unit. For COO-1D encoded blocks, absolute positions of NZs can be obtained directly from metadata. Otherwise, absolute positions of NZs need to be computed explicitly by decoding metadata (e.g., bitmap or RLC) through simple combinational logic consisting of AND gates, multiplexers, and adders (e.g., in [41] and [113]).

b) Sc-Vec Mul: For SIMD processing, multiple arrays are indexed with the position of an NZ. Fig. 9 shows such mechanism used in Cnvlutin PEs [72]. Each of 16 subunits in Cnvlutin PE featured an activation lane (streamed an input channel vector), 16 multipliers, and 16 filter lanes. A common NZ activation was fetched from the activation lane, and its position was used for looking up in all 16 filter lanes to obtain corresponding weights for multiplication.

c) Vec-Vec Mul: PEs of some accelerators spatially process vectors at every cycle (e.g., with 16 multipliers and an adder tree in Cambricon-X). As illustrated in Fig. 10,

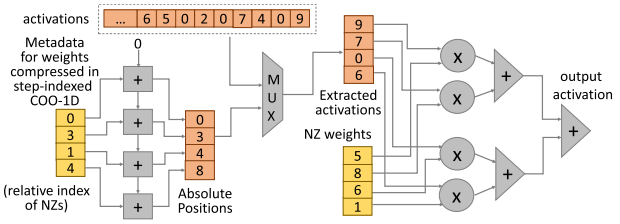


Fig. 10. ***Data extraction via central indexing module in Cambricon-X [41] accelerator. The indexing module decodes weights encoded in step-indexed COO-1D format to obtain the absolute positions of NZs. Then, it extracts the activations via a parallel look-up, which are later communicated to a PE via fat-tree NoC for a vector–vector multiplication. (Figure adopted from [41].)***

based on positions of NZs of a vector, a combinational logic with multiplexers can select matching data elements to feed the arithmetic units (e.g., in [41], [60], and [61]). *An associated challenge is overheads of parallel look-up*. To exploit high sparsity, larger multiplexers need to be used for indexing the dense tensor, as positions of scattered NZs are likely distant. With the search length set as 256 (supports 93.75% sparsity for fetching 16 NZ elements), a central indexing module in Cambricon-X occupied about 31% and 35% of total on-chip area and power, respectively (exceeded total power of all 16 PEs) [41].

2) Compare Metadata of Sparse Tensors for Extracting Matching Pairs of NZs: For effectual computations over multiple compressed tensors, the extraction logic determines pairs of NZs (intersections) by comparing indices either from metadata streams or in multistage indexing.

a) MAC: Circuitry for extracting NZ scalars can consist of one or more comparators (or AND gates for comparing bitmaps) and an additional indexing logic (e.g., in ZENA [130] and SparTen [116]). The comparators match positions of NZs, and the indexing logic uses their outputs to extract the leading pair. Due to the diverse sparsity of tensors, positions of NZs may not match during comparison. Therefore, the detection logic uses several comparators to search within a large window, which usually can provide at least one pair at every cycle. Priority encoders provide the leading n-pairs for feeding n computational units ($n = 1$ for scalar PEs). The data extraction unit can use skip mechanisms (e.g., in ExTensor [93]) to quickly navigate through the lanes.

Alternatively, multistage indexing logic is used for extracting the pair. The first stage obtains a position of an NZ from one tensor for indexing another tensor. The later stage checks if there is a corresponding NZ in another tensor and extracts it upon matching the positions. For example, in EIE [42], each PE loads an NZ activation from a queue; when it does not have any matching weights, it fetches the next activation from the queue in the next cycle. Depending on the sparsity level and pattern, the indexing-based design occasionally may not find the matching data, wasting the execution cycles, i.e., functional units in the pipeline are not utilized.

b) Sc-Vec Mul: PEs in EyerissV2 [43] use multistage extraction. Each SIMD PE fetches a CSC-coded activation and its position, and checks positions of NZ weights. Upon a match, it forwards the activation and weights to two MAC units.

c) Vec-Vec Mul: The data extraction logic to feed multiple arithmetic units of a vector PE requires multiple comparators followed by priority encoders or multiplexers. For example, in SNAP architecture [107], an associate index matching module (AIM; see Fig. 11) determines the positions of NZs in case of valid matches. Each PE of a row is interfaced with a shared AIM. Using comparison outcomes from AIM, a sequencer in each PE determines leading pairs of matching data, which are then fed to three

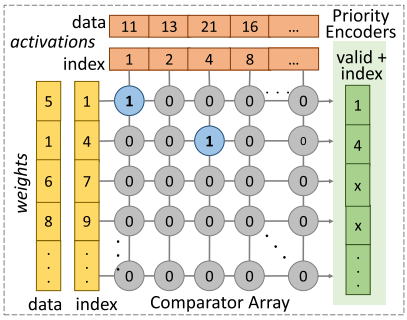


Fig. 11. ***Associative index matching in SNAP. (Figure adopted from [107].)***

multipliers within the PE. Cambricon-S [37] uses similar extraction logic, but its comparator array is just ANDing of the bits due to bitmap encoding.

3) Eliminating Extraction of Intersecting NZs: Some accelerators do not require extracting unstructured NZs.

a) Orchestrating structured computations: A few techniques targeted high sparsity of single tensor (DNN weights). With data pruning or transformations, they achieved coarse-grain sparsity so that each PE can process a dense region of NZs. ERIDANUS [126] proposed a pruning algorithm to cluster the weights [see Fig. 12(a)]. Blocks of NZ weights are streamed to PEs of systolic arrays for conventional processing [see Fig. 12(c)]. Corresponding activations are kept stationary. Partial products computed by each row of PEs are added on a separate adder tree. When block width for structured pruning can be set as the height/width of the systolic array, dot products can be accumulated linearly over the systolic array itself. So, *structured sparsity allows executing denser blocks conventionally on accelerators while requiring additional support to index and communicate the blocks.* Adaptive tiling [127] used a column-combining approach. For a sparse GEMM, NZ weights were statically combined such that each column of the systolic array could process multiple columns of IAs. So, it obviated the runtime data extraction and reduced total invocations of the systolic array by 2×–3× for processing pointwise CONVs of MobileNet. CirCNN [165] and C-LSTM [166] proposed executing DNN

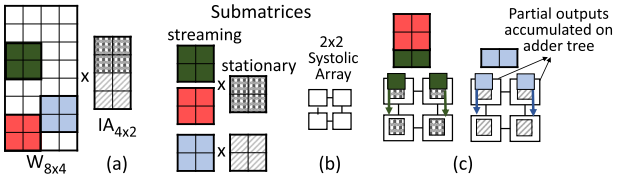


Fig. 12. ***Computation of locally dense regions in ERIDANUS (Figure adopted from [126].) (a) Matrix multiplication with block-sparse weights. (b) Submatrices for processing on a 2 × 2 systolic array. (c) Multiplication of streaming blocks (NZs) with stationary data.***

operators as fast Fourier transform (FFT) on smaller block-circulant matrices.

b) Coordinate computation unit: SCNN [115] and SqueezeFlow [65] perform unit-strided convolutions as a Cartesian product where all elements of two blocks of tensors should be multiplied together. Due to all-to-all multiplication, no special support is required for extracting matching pairs of NZs. However, index computation is still required to determine which partial-sums should be accumulated with partial products. This calculation is performed in a "coordinate computation unit" that processes metadata (indices of NZs) and determines indices of outputs. These approaches require conflict detection in hardware since it cannot be predetermined which accumulators would be accessed in any cycle. Since coordinate computation unit facilitates direct processing on compressed tensors, it may also be used for computing block-level indices for processing a *coarse-grain block-sparse* tensor.

B. Centralized Versus Distributed Management

1) Centralized: The data extraction unit can be either centralized (and shared among PEs) or within pipelines of PEs. The advantages of central mechanisms are given as follows.

1) PEs can be directly provided effective NZs for useful computations [41]. It can also be used as a preprocessing unit for a PE-array that processes structured computations, e.g., systolic arrays or near-data accelerators.
2) Centralized extraction in some architectures (e.g., Cambricon-X [41]) duplicates hardware for concurrent extractions for PEs. However, the module can be time-shared by multiple PEs (e.g., in SNAP [107]), which can reduce area and power. In fact, by leveraging structured W-sparsity, the module in Cambricon-S *shares* extracted indices among *all* PEs.
3) Centralized logic extracts work for multiple PEs, and often, it is coupled with a controller that allocates data to PEs. So, it can enable *runtime load balancing*. However, a major challenge is to maintain spatial data reuse. This is because the centralized unit mostly extracts data on a per-PE basis for communication to a unique PE. So, the common data for multiple PEs cannot be multicast. SNAP overcomes this limitation by sharing a module with a row of PEs and multicasting data to PEs. The multicast occurs first, followed by PEs communicating their metadata to the extraction unit. Then, extracted indices are streamed back to a PE, which uses them to obtain data from its local RF for computations.

2) In-PE: PEs of several accelerators, such as Cnvlutin [72], ZENA [130], and EyerissV2 [43], extract appropriate data.It allows a controller to multicast or broadcast tensor elements for spatial reuse. Then, in-PE logic extracts the data. However, challenges are given as follows: 1) in-PE logic may incur ineffectual cycles for extraction that cannot be hidden and 2) employing inter-PE load-balancing in the hardware may be infeasible or costlier, as the actual work carried out by different PEs is unknown while offloading compressed tensors to PEs (until extraction in PE datapath).

C. Optimization Opportunities

1) Sparsity-Adaptive Low-Cost Data Extraction Mechanisms: Encodings of sparse tensors are often selected with a focus on storage benefits. However, the computational overhead and hardware cost for encoding and decoding tensors should be also reduced since they affect the performance and energy consumption. When the data extraction cannot feed n pairs of NZs to n computational units of a PE at every cycle, achieved speedup can be lower from the peak. Sustaining the acceleration across various sparsities of tensors can be challenging, as different extraction schemes may be cost-effective for only a certain sparsity range and patterns. For example, for similar sparsity, extraction logic with a few comparators may easily locate a pair of NZs. However, an indexing-based mechanism may be more effective when one tensor is highly sparse and another is dense. Moreover, when positions of NZs in the two tensors are considerably distant (e.g., for diverse sparsity levels or for hypersparse tensors), the extraction logic needs to use several comparators or multiplexers for parallel lookup so that it can extract at least one pair to feed each computational unit. Therefore, the extraction module needs to be *configurable* or consists of (and select among) multiple mechanisms so that it can exploit a variety of sparsity at a modest hardware cost. For the latter, it can dynamically use partial features for desired sparsity levels/patterns (power-gated otherwise).

2) Tightening Integration With Load Balance Mechanism: The central data extraction module can enable dynamic load balancing of work among PEs (e.g., data-driven dynamic work dispatch in GraphDynS [167]). As discussed in Section X, the inter-PE imbalance can be severe due to the irregular distribution of NZs in tensor blocks that are allocated to PEs. Its mitigation by structuring the data may not always be possible (e.g., for activations/weights of some models or applications beyond deep learning). Consequently, accelerators may attain ineffective utilization of PEs and low speedup. Although some accelerators used hardware modules for dynamic balancing, further efficiency may be achieved by enhancing the centralized extraction module with additional low-cost logic. This is because it already keeps the track of the data provided to PEs, which can lead to information about the number of operations performed by different PEs.

VII. MEMORY MANAGEMENT OF COMPRESSED TENSORS

Accelerators contain *multibanked* scratchpads that are usually shared among PEs. Either a scratchpad is *unified* [23] or separate buffers store different tensors [111], [130].

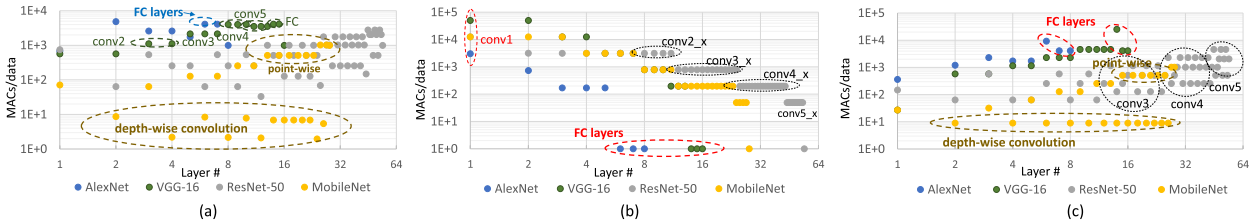


Fig. 13. ***Data reuse opportunities for executing different CNN layers (dense tensors) on hardware accelerators. (Figure inspired from [43].) (a) Reuse of IAs. (b) Reuse of weights (batch size = 1). (c) Reuse of partial summations.***

Their sizes vary from several tens of kBs [37], [43] to several MBs [22], [72]. Effective management of shared and local memory highly reuses data and hides memory access latency behind computations on PEs. This section discusses how sparsity and reduced shapes of tensors lower reuse. However, compressed tensors help to achieve better speedups and energy efficiency, as more data fit in on-chip memory, reducing off-chip accesses. This section also describes how irregular accesses (e.g., arbitrating output activations) make management of the banks challenging. Then, it discusses reusing intermediate outputs via fused-layer executions and how sparsity affects it.

A. Leveraging Data Reuse Opportunities

1) Reuse Characteristics: Depending on the functionality of layers, there can be significant reuse of tensors. Figs. 13 and 14 depict reuse opportunities for different layers (early CONV layers, later CONV layers, MLPs, DW-CONVs, PW-CONVs, expand or reduce layers, and attention mechanism). For each tensor, the data reuse is calculated as the total number of MACs per data element. For better visualization, reuse factors and layers are plotted on a logarithmic scale.

Input Activations: Reuse of IAs increases with going deeper in CNNs since the number of filters increases significantly. It is also high for "expansion" layers in bottleneck blocks (see Fig. 14). DW-CONVs are an exception and present very low reuse as there is only one filter. "Squeeze" or "reduce" layers present moderate reuse for dense tensors. Reuse in FC layers or MLPs (e.g., in encoder/decoder layers of Transformers [7]) depends on the sizes of weight matrices (i.e., sizes of output tensors).

Weights: Since 2-D feature maps in CNNs are usually much larger than 2-D weights, weight reuse can be higher by an order of magnitude. With going deeper in CNNs, feature maps shrink spatially, which lowers the reuse. There is no weight reuse for MLPs, but increasing the batch size linearly improves the weight reuse. Video processing applications use 3-D CNNs (e.g., c3d [6]), which can further increase the reuse opportunities [168] for IAs and weights due to additional processing steps on consecutive frames. For NLP models, such as Transformer [7] and BERT [8], Fig. 14 illustrates weight reuse for executing a sequence of 24 and 107 tokens, respectively. $MatMul$s in the attention-based calculation are shown for a single head.

Partial summations: Input channels are increased as we go deeper into CNNs. Similarly, "reduction" layers in bottleneck blocks involve more input channels. Both improve the reuse of partial summations. MLPs also usually provide high reuse due to larger input vectors. DW-CONVs show very low reuse because partial summations are not accumulated across input channels.

2) Impact of Sparsity on Reuse: Increase in sparsity can lead to lower reuse. To determine the impact of sparsity, we considered evaluations by Han *et al.* [25] for pruned AlexNet and VGG-16 models. For recent DNNs, such as MobileNetV2 or BERT models, we considered sparse models, as listed in Table 3. Then, we calculated the reuse as NZ MACs per NZ of a tensor. Fig. 14 plots the reuse

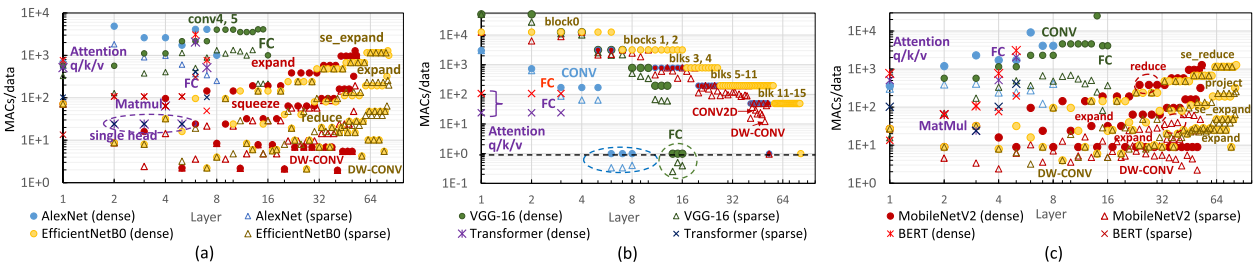


Fig. 14. ***Impact of sparsity on data reuse opportunities for accelerating CNNs and NLP models. (a) Reuse of IAs. (b) Reuse of weights. (c) Reuse of partial summations.***

opportunities for both dense and sparse tensors of CNNs and NLP models. Since execution in encoder/decoder modules of NLP models is repetitive, unique layers of a single module are only shown (sparsity averaged across all encoder/decoder modules). The figure shows that, for sparse models, reuse characteristics are preserved, but the reuse factor decreases for almost all layers and tensors compared to processing dense tensors. Primarily, this is due to the reduced number of effectual MACs. For example, for MLPs without batching, weight reuse can drop below one. It means that, even if a weight matrix consists of NZs, some of them are never used due to the unavailability of matching NZs in IAs. As an exception, reuse of weights remains the same when activation sparsity is absent (e.g., EfficientNetB0 [124] and BERT [8]). Similarly, with dense weights, low or moderate reuse of activations remains the same for DW-CONV or "excite" layers, respectively.

The reuse of partial summations also decreases since effectual MACs per partial summation decrease with sparsity. Note that each output activation element still needs to be populated or assembled before ReLU/encoding. Due to sparsity and fewer input channels, the reuse is low or moderate in "expansion" layers. Similarly, small matrices in processing individual attention heads exhibit low reuse. The reuse remains high for "reduce" layers in CNNs or query and value processing and FC layers in NLP models. To sum up, although sparsity reduces the reuse of tensors, there can be high data reuse for many layers (up to $1E+04$), which should be exploited for efficient accelerations.

3) Temporally Reusing Data Through Shared On-Chip Memory: Like CPUs, accelerators have memory hierarchies because applications have different working set sizes. Data reuse can be leveraged *temporally* (repeatedly accessing data from memory without accessing lower level memory) and *spatially* (providing the same data to multiple PEs without repeatedly accessing memory). After exploiting high temporal reuse, the highest energy is spent in upper buffers [23], [44].

B. Hiding Miss Latency Behind Computations

1) Management of Tiled Data in Double-Buffered Memory: On-chip buffers are typically not large enough to accommodate all tensors. Therefore, loops are tiled for reusing some tensors from buffers while repeatedly accessing other tensors from the off-chip memory [20], [115]. Since scratchpads are noncoherent and their management is software-directed, data are transferred by direct memory accesses (DMAs) [22], [56]. PEs are kept engaged in useful computations by interleaving computations with memory accesses. Such an objective is usually achieved by double-buffering (also known as ping-pong buffers) [37], [130]. Loop optimization techniques, such as loop tiling and ordering, can determine the sizes of tensor blocks to be managed in memories and sequence of memory accesses for high reuse and reduced data transfers [23], [56].

2) Asynchronous Communication: Some accelerators hide the latency of communicating data to the shared/local memory with an asynchronous mechanism that refills the memory after some data have been consumed (e.g., in Cambricon-X [41]). For such execution, PEs and a DMA controller may simultaneously produce/consume data either through different banks or at the granularity of small blocks in the same bank. Similarly, when accessing shared memories via configurable communication networks [43], PEs can execute in a dataflow fashion and request partial refilling of their memory with new data. Such *mechanisms for asynchronous communication and computations can alleviate work imbalance among PEs* that are caused by leveraging unstructured sparsity.

3) Impact of Sparsity on the Latency of Memory Accesses and Speedup: For memory-bounded execution (e.g., MLPs), even with effective prefetching, miss penalty may be significant. It restricts accelerators from achieving peak performance [22]. When tensors are sparse, the amount of data that need to be transferred from off-chip reduces significantly, leading to substantial performance gains. For example, Cambricon-S reported up to 59.6× speedup of FC layers for hypersparse weights. However, higher IA-sparsity did not provide such gains (speedup saturated at about 14×) since the latency of accessing weights dominated total execution time. For processing high sparsity (e.g., 90%+) and low reuse, it becomes challenging to engage functional units into effectual computations. This is because, with *low* arithmetic intensity, required data *may not be prefetched* at available bandwidth.

C. Management of Multibank Memory

1) Concurrent Accesses to Memory Banks: While single-bank memory can be easier to manage, it is infeasible to provide multiple ports for the PE-array with just one bank [169]. Moreover, multiport unified memory consumes very high power and longer latency [170]. So, on-chip memories are partitioned into smaller banks [43], [115], [171]. For mapping a layer onto the accelerator, each bank is usually allocated to only one tensor (e.g., in EyerissV2 [43]). Banked buffers provide multiple read and write ports, allowing simultaneous accesses to different tensors stored in different banks [20], [172]. Sometimes, a data layout reorganization is required before loading into memory banks. Such a transformation is done after loading it from DRAM or before writing outputs to DRAM, which consumes additional execution time and energy. For compressed tensors, such transformation can be done along with the data encoding [111] at alleviated overheads.

2) Arbitration and Conflict Management: Depending on the indexing logic and interconnect between memory and PEs, managing application data may require additional compilation support or hardware logic for data arbitration and conflict management [115], [164]. For regular memory accesses (e.g., dense or block-sparse data), allocation

and accesses to banks can be determined for mappings of layers. However, computations on unstructured sparse data can lead to accessing *arbitrary* banks and require special support. For example, outputs from PEs may need to be written to different banks. Moreover, accelerators contain accumulator buffers [115], where PEs or their functional units are connected with memory banks via a crossbar. The crossbar arbitrates write-back of outputs to the appropriate bank [65], [115]. Since these partial outcomes can correspond to noncontiguous elements in an output tensor, bank conflicts are possible during arbitration, i.e., multiple outputs need to be simultaneously handled by the same bank [115], [164]. To obviate conflicts, the buffer contains more banks (e.g., $2 \times N$ banks for storing outputs from N sources in SCNN [115]). It alleviates collisions in hashing irregular outputs into different memory banks. Consequently, the crossbar may require higher bandwidth and significant on-chip area (e.g., 21% for a 16×32 crossbar in each SCNN's PE).

D. Reusing Intermediate Tensors

1) Reusing Intermediate Tensors From Large On-Chip Memory: Intermediate feature map in DNNs is an output of a layer that serves as input to later layers. It can be kept stationary and reused from on-chip memory to reduce off-chip traffic. Such reuse is amplified when input is the same for multiple layers due to residual connections [2] or high cardinality (e.g., ResNeXt [95]). Leveraging it can be important for latency-bounded real-time applications. *Sparsity-encoding and quantization significantly make such reuse opportunities more feasible* due to reduced storage requirements. Accelerators with large memories (hundreds of kBs), such as SCNN [115] and Cnvlutin [72], can leverage such reuse.

2) Overcoming Static Bank Assignment: Many accelerators process models layer-by-layer and do not leverage cross-layer reuse, i.e., write outputs for layer L in DRAM and load them back later as inputs for layer $L + 1$. It is more prevalent among accelerators with small memories. Moreover, bank assignment for each tensor is often fixed at design time [172], which enforces write-back of outputs and reloading them later in other banks as inputs while processing next layers. So, in both cases, output activations are not reused on-chip, causing excessive off-chip memory traffic. To address this problem and exploit cross-layer reuse, shortcut-mining [172] used a flexible architecture with decoupled physical-logical buffers.

For preknown sparsity, prior techniques for statically determining the data allocation to memory banks may work well by estimating sizes of encoded tensors. However, for dynamic sparsity, conservative estimations may lead to inefficient utilization of banks, and efficient banking for nonconflicting accesses can also be challenging.

3) Fused-Layer Execution: Fused-layer CNNs [173] leveraged cross-layer reuse by processing a small tile of activations such that outputs for few layers can be computed alongside while retaining the corresponding data in the on-chip memory. Fig. 15 shows an example for processing an input tile of 5×5 activations (C_L input channels) for layer L and, finally, obtaining 1×1 output activations (M_L + 1 output channels) for layer $L + 1$. Apart from reusing intermediate outputs for obtaining the output tile, corresponding tiles of intermediate activations and filters are maintained in the memory and reused partially for processing the next tiles (striding execution in the spatial direction). Alwani *et al.* [173] reported reducing off-chip transfers of input feature maps by 28% for the first two layers of AlexNet and 95% for the first five layers of VGG-19. Since the cascading by storing all the filters and input channels (dense tensors) requires high memory, Alwani *et al.* [173] applied it to only early layers. However, encoded sparse tensors and further tiling across filters/channels allow fitting tensors for multiple layers in the small memory, making such reuse opportunities more feasible. The tile size and number of layers that can be fused are bounded by memory capacity. So, fusion parameters depend on the actual/anticipated sparsity levels. For efficient executions, fusion parameters need to be explored systematically with sparsity-aware dataflows.

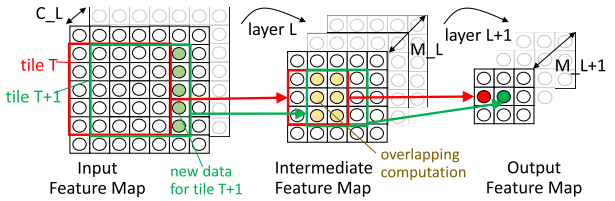


Fig. 15. ***Fusing the execution of layers can significantly reuse intermediate activations [173]. (Figure adopted from [173].)***

E. Techniques for Further Energy Efficiency

1) Look-Ahead Snoozing: Depending on the sparsity, encoding of tensors, and mapping of the layer, several banks can be unused or inactive for certain time intervals. Accelerators achieve further energy efficiency by power gating unused or inactive banks. For example, look-ahead snoozing in CompAct [111] targeted reducing the leakage power of large on-chip SRAMs. Each bank of its activation SRAM can be power-gated. Banks unutilized during the execution of a layer were put in the deep sleep mode (maximal savings in leakage power, while not preserving any data in unused banks). Furthermore, the period of active cycles for each bank was determined based on the data movement schedule. Then, inactive banks were snoozed during execution (i.e., connecting to the data retention voltage for consuming lower leakage power).

2) Skipping Memory Hierarchy: Some layers do not provide significant reuse. Data reuse is also lowered due to sparsity and architectural choice for extracting or communicating NZs. Therefore, a few accelerators (e.g., EIE [42] and Cambricon-X [41]) obviate storing nonreusable data

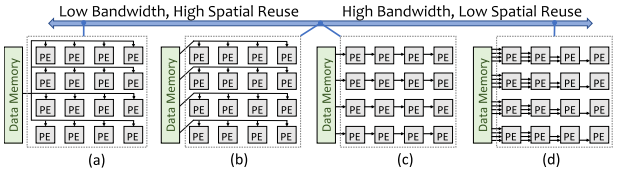


Fig. 16. ***Common NoC designs. (Figure adopted from [43].) (a) Broadcast. (b) 1-D Multicast. (c) 1-D Systolic. (d) Unicast.***

in the shared memory and directly feed it to appropriate PEs (weights for MLPs).

F. Optimization Opportunities

Managing both data and metadata in unified memory: Accelerators often contain separate buffers for metadata (positions of NZs and indirection tables for shared values). Although such designs are easy to manage for processing tensors of some models encoded in a specific format, they may not work well across different levels of sparsity and value similarity, as storage requirements vary significantly. So, designers can explore unified memory architectures for managing both data and metadata (including memory partitioning and bank management) and their tradeoffs. It can also be leveraged to tailor efficient designs for programming FPGAs.

VIII. INTERCONNECTS FOR DISTRIBUTING NONZEROS AND REDUCING PARTIAL OUTPUTS

Network-on-chip (NoC) is required to efficiently distribute data to PEs, exchange data between PEs (for reducing partial outputs), and collect distinct outputs back from PEs. To process data-intensive ML models, accelerators employ multiple high-bandwidth interconnects for simultaneous communication of different tensors between PEs and buffers. First, this section describes NoCs for the distribution of operands, which varies in terms of bandwidth and spatial reuse of the data. With efficient NoC design, PEs can be engaged in processing data from input FIFOs or local memory, which gets interleaved with communication of another set of data via NoC. This section also discusses *configurable NoC* designs that can support various bandwidth requirements and spatial reuse opportunities due to variations in sparsity and tensor shapes. In processing sparse tensors, *unstructured reduction of partial outputs among PEs can be challenging*. This section describes different mechanisms for *accumulating* the outputs *temporally* or *spatially* at PE level and PE-array level. It also discusses configurable mechanisms for *asymmetric* accumulation of *variable-sized* partial outputs.

A. Mechanisms for Distribution of Operands

Fig. 16 shows some common NoC designs for distributing the operands, their bandwidth, and achievable spatial reuse [43]. Data can be reused spatially by distributing it to multiple PEs or functional units. For layers with high reuse opportunities (see Fig. 13), it lowers communication and helps to hide the communication latency. Most accelerators leverage spatial reuse with multicast or broadcast NoC. They consist of configurable buses or trees that multicast the data to PEs (often in a single cycle) [20], [105]. In contrast, the mesh interconnect (e.g., in systolic arrays [22]) or 1-D buses communicate the data and reuse spatially with a store-and-forward mechanism. Low-reuse tensors are distributed with unicast NoCs. Table 8 lists common interconnect topologies used by previous accelerators for data distribution.

Communication requirements vary significantly depending on the sparsity of tensors, available reuse, and adopted dataflow mechanism. Prior work [174] provides a detailed analysis of different NoC topologies, and the work [175] characterizes the NoC bandwidth required for different dataflows. Similarly, analytical tools, including [57], model implications of different dataflows on communication requirements and execution time.

1) Broadcast: Accelerators, including Cnvlutin [72], EIE [42], and Cambricon-S [37], use broadcast NoC to reuse activations for processing CONVs or MLPs. Similarly, in SCNN [115], weights are broadcast to PEs for executing unit-strided convolutions with input stationary dataflow. For sparsity-encoded tensors, their NZs (and positions) can be broadcast for spatial reuse, as long as the NZs are indexed or extracted afterward (e.g., in-PE extraction in Cnvlutin and EIE). In Cambricon-S, positions of intersecting NZs are extracted centrally before broadcast, but, due to structured sparsity, the same extracted positions are used by all PEs. So, NZ activations are broadcast to all PEs.

2) Multicast: Eyeriss [20], ZENA [130], and SNAP [107] use multicast NoC to reuse multiple operands spatially. For example, Eyeriss processed tensors with row-stationary dataflow where PEs of a row processed the same spatial rows of filters, and diagonal PEs processed the same spatial row of feature maps. Eyeriss facilitated such multicasting through its configurable NoCs, which consisted of rowwise and columnwise controllers for 2-D PE-array. Each controller could be configured with a predetermined tag value, which was compared with the row or column tag of a packet. Upon matching the tags, a rowwise controller forwarded the packet to associated columnwise controllers, and a columnwise

Table 8 NoC Designs for Distribution of Sparse Tensors

Topology	Unicast	[41], [65], [72], [93], [113]–[115], [121]–[123]
	Multicast	[93], [107], [121]
	Broadcast	[37], [42], [60], [65], [66], [68], [72], [107], [112]–[117], [122], [123], [130]
	Mesh	[111], [126], [127], [134]
	Configurable	[43], [105], [175]
Spatial Reuse	Activations	[37], [42], [43], [60], [66], [68], [72], [105], [107], [112]–[114], [116]–[118], [121]–[123], [130], [164]
	Weights	[43], [65], [105], [107], [115], [117], [130], [164]

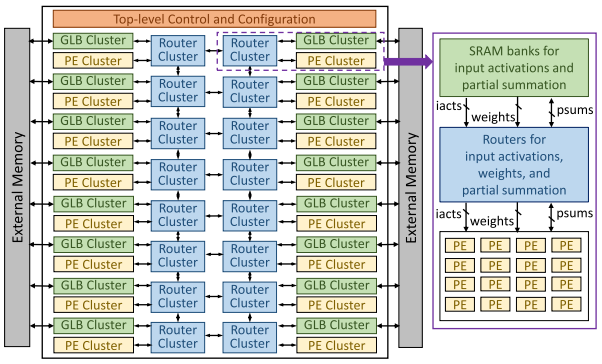


Fig. 17. ***EyerissV2 accelerator architecture [43]. (Figure adopted from [43].)***

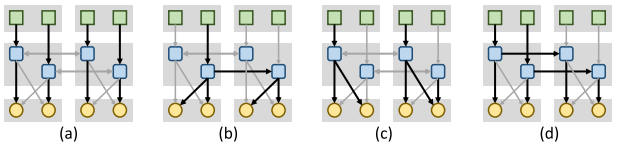


Fig. 18. ***Different configuration modes of hierarchical mesh network in the EyerissV2 architecture [43]. (Figure adopted from [43].) (a) High BW. (b) High reuse. (c) Grouped-multicast. (d) Interleaved-multicast.***

controller forwarded it to the associated PE. Similarly, for processing bitmap-coded tensors in ZENA [130], a block of activations was broadcast to a row of PEs, and a block of weights was multicast to PEs of the same column.

3) Mesh: A few accelerators, including Compact [111], ERIDANUS [126], and [127], use systolic arrays with mesh interconnects. Since the same data are forwarded among PEs in the same row or column, such NoCs achieve the same amount of spatial reuse as multicast NoCs. However, for sparse tensors, efficient and correct processing becomes challenging. Hence, *preprocessing* is needed to cluster appropriate NZs or index appropriate block of structured-sparse tensor before feeding PEs of the systolic array [105], [126], [136].

4) Unicast: SCNN [115], Cambricon-X [41], and SqueezeFlow [65] use unicast NoC or point-to-point links. Such NoCs concurrently feed different elements to various PEs. They are used when spatial reuse of a tensor is infeasible (e.g., weights in MLPs and NZs are extracted beforehand due to dataflow requirements), or outputs are collected simultaneously (see Section XI-A). With high bandwidth, they reduce communication latency [41] but can incur high area and power.

5) Configurable: Communication requirements vary with different dataflows that are effective for only some DNN layers (see Section IX-B and Table 25). Furthermore, while communication may consist of gather, scatter, forward, or reduction patterns [175], [176], efficient execution may demand their combination or even nonuniform patterns, including multihop communications among PEs [23]. Therefore, configurable NoC designs are required, which can support various communication patterns that are amenable to different reuse and sparsity. Recent designs including EyerissV2 [43], microswitch-NoC [175], and SIGMA [105] address some of these challenges.

EyerissV2 [43] uses a novel *hierarchical-mesh* NoC, which is illustrated in Fig. 17. EyerissV2 contains 16 clusters (8×2 array) of PEs and global buffers (GLBs). Each PE-cluster contains 3×4 PEs, and each 12-kB GLB-cluster contains seven banks for input and output activations. At the top level, router clusters are connected through a 2-D mesh, and they enable communication among different PE-clusters and GLB-clusters. For local communication among each PE-cluster and GLB-cluster, a router-cluster with ten routers is used. Each router connects PEs with a port of the GLB cluster for accessing GLB bank or off-chip memory (three, three, and four routers for managing IAs, weights, and partial summations). Locally, an all-to-all NoC connects all PEs of a PE-cluster to the routers for each data type. As illustrated in Fig. 18(a)–(d), it facilitates multiple communication patterns, including multicast, broadcast, and unicast of the tensors. The 2-D mesh topology enables intercluster communications, allowing an interleaved-multicast or broadcast to all clusters.

For an N-PE accelerator, an array of microswitches [see Fig. 19(a)] contains $\log_2 N + 1$ levels with N microswitches at each level. Each microswitch contains a small combinational logic for configuration and up to two FIFOs for buffering the data during routing conflict. With small logic and storage, data traverses through several microswitches within each cycle [175]. All microswitches contain *gather* and *scatter* units, and bottom microswitches (level $\log_2 N$) also contain local units for inter-PE communication. In top microswitches (level 0), the scatter unit connects to memory banks, and the gather unit uses round-robin-based priority logic for arbitrating the incoming data in a pipelined manner. In middle microswitches, scatter units forward data to desired lower level links, and gather

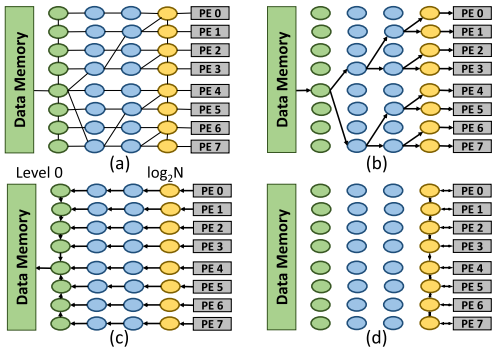


Fig. 19. ***(a) Microswich network [175]. NoC configurations: (b) multicast, (c) gather, and (d) local communication. (Figure adopted from [175].)***

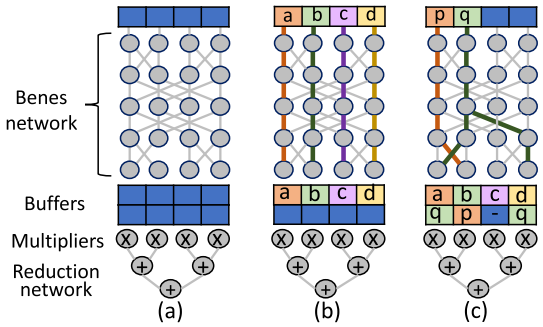


Fig. 20. ***(a) Flexible dot product engine in SIGMA accelerator [105] features a data distribution NoC with configurable switches interconnected via Benes topology. (b) and (c) Configuration of the interconnect facilitates different unicast and multicast communication patterns. (Figure adopted from [105].)***

units stream the data back. In bottom microswitches, scatter and gather units stream the data, and local units connect adjacent PEs. Fig. 19(b)–(d) shows how configurable microswitches can enable various communication patterns.

SIGMA [105] used *Benes topology* with *configurable* switches [see Fig. 20(a)]. For N source and destinations, the interconnect contains $2\log_2 N + 1$ levels, each with N number of 2×2 switches. Each switch receives two control signals to determine whether to forward data vertically and/or diagonally. After combining communication requirements for distributing all elements to desired multipliers, switches can be configured to forward the data, as shown in Fig. 20(b) and (c).

B. Mechanisms for Reduction of Partial Outputs

Computation primitives of ML models require reduction (accumulation) of partial outputs from PEs or their functional units. It can be done temporally and/or spatially (see Table 9).

1) Temporal: All the reductions for computing an output scalar are performed on a *single* PE (or a functional unit) during *different* cycles. Accumulations are done temporally when different PEs compute distinct outputs, e.g., for output stationary dataflow. The temporal reduction makes the processing of sparse tensors *simple* since PEs update partial outputs in their private memory or registers *without communicating* to other PEs via an additional interconnect. Therefore, it is adopted by many accelerators, including EIE [42], ZENA [130], and SparTen [116]. However, temporal accumulation requires *indexing* the buffer for reading/writing partial outputs or accumulating computations in the output register (of MAC unit). So, it involves register/memory read and write operations that consume higher energy than integer arithmetic [23], [44]. Besides, using local accumulator buffers for vector/SIMD functional units (e.g., in SCNN [115]) requires support for arbitration of partial outputs.

2) Spatial: Partial outputs can be reduced spatially for an output scalar. It can be either done by functional units within a PE (e.g., adder trees in Cambricon-X/S to sum up partial products) or inter-PE communication via a *separate interconnect* (e.g., forwarding in systolic arrays). Inter-PE spatial reduction usually requires communication among neighboring PEs and is typically achieved through a mesh or similar topology [20], [127]. Spatial reduction obviates buffer accesses and improves energy efficiency (e.g., by 2×–3× [129] compared to the temporal reduction on scalar PEs). These linear or tree-based reductions are typically symmetric. However, a major challenge is to enable *asymmetric and asynchronous reductions* of a variable number of partial outputs, for adapting to high sparsity, tensor shape, or target functionality (e.g., DW-CONV). This is because an efficient dataflow may require some of the interconnected functional units or PEs to process partial outputs for distinct output elements (e.g., different depthwise groups); all partial outputs cannot be reduced altogether. Hence, configurable interconnects are needed. Otherwise, for high sparsity or hypersparsity, functional units cannot be fed enough NZs and are poorly utilized. Note that structured sparsity can alleviate imbalance by inducing patterns such that all PEs process the same NNZs. However, configurable mechanisms are still required to support different dataflows for the variations in functionalities or tensor shapes.

3) Spatiotemporal: Partial outputs can be reduced spatially and temporally and *locally* (within PEs) and *globally* (across PEs). Spatial and temporal reductions of outputs depend on the mapping of computation graph onto PEs [56]. In spatiotemporal reduction, different PEs or their functional units compute partial outputs at every cycle or a few, which are, at first, reduced spatially. The resultant partial output is then reduced temporally by updating the previous partial output in the memory. For example, when data stream through PEs of a systolic array, there is an inter-PE spatial reduction of partial outputs (via PEs of each column). Then, the bottom PE-row provides the reduced partial outputs to accumulator buffers (CompAct [111] and TPU [22]). PEs of SNAP [107] perform spatiotemporal accumulation locally, where partial products are first spatially accumulated through a configurable adder tree and then accumulated in PE's memory over time.

4) Temporospatial: In temporospatial reduction, PEs compute partial outputs and reduce them locally over time.

Table 9 Mechanisms for Accumulations of Partial Outputs

Temporal	[42], [66], [68], [93], [113], [114], [116], [118], [121]–[123], [130], [133], [134]
Spatial (intra-PE)	[37], [41], [105]
Spatial (inter-PE)	[65], [66], [105], [177]
Spatio-temporal	[107], [111], [129]
Temporo-spatial	[20], [43], [107], [115]
Configurable	[105], [107], [177]

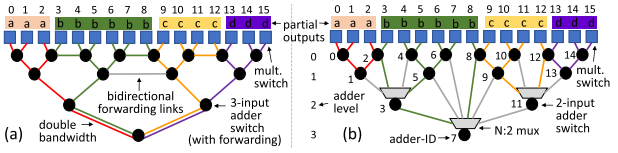


Fig. 21. ***Configurable spatial reduction trees. (a) Augmented reduction tree in MAERI. (Figure adopted from [177].) (b) Forwarding adder network in SIGMA. (Figure adopted from [105].)***

Then, they are collected later and accumulated spatially via interconnect before further processing (e.g., write-back, encoding). For example, PEs of a cluster in EyerissV2 [43] first locally accumulate partial summations. Then, partial outputs can be accumulated across vertically connected clusters. SCNN [115] PEs compute output tiles corresponding to distinct input feature maps stored in their buffers. Outputs are temporally reduced by indexing the accumulator buffers. Then, overlapping fractions of incomplete outputs are exchanged among neighboring PEs for reduction. SNAP [107] also performs temporospatial reduction at PE-array (core) level. Its PEs accumulate outputs locally over time, which are reduced spatially across horizontal/diagonal PEs by a core-level reducer.

5) Configurable: MAERI [177] and SIGMA [105] employ configurable reduction trees for efficient and asymmetric spatial reduction of partial outputs. So, it can be useful for spatial processing of unstructured sparsity and variable-sized vectors for dot products. The augmented reduction tree in MAERI [see Fig. 21(a)] allows an asymmetric reduction of partial outputs with configurable adder switches and bidirectional forwarding links. Each three-input adder switch can receive two partial outputs from the previous level and one via a forwarding link, and it can add and forward them. Plus, upper levels of the tree (near root) have double bandwidths than lower levels, allowing simultaneous collection of multiple reduced outputs. The forwarding adder network in SIGMA [see Fig. 21(b)] enables similar configurable reduction but at reduced area and power. Instead of three-input adders, it uses two-input adders and $N:2$ mux for selecting the inputs. Also, adders at the zeroth level allow bypassing of partial products to the next level.

C. Optimization Opportunities

1) Low-Cost Flexible Interconnects for Accommodating Spatial Reuse Opportunities, Dynamic Communication Requirements, Various Sparsities, and Different Precision: Variations in data reuse (see Fig. 14) are caused by the tensor size, functionality (e.g., stride and separable convolution), batch size, and sparsity of tensors. The communication mechanism needs to leverage available reuse by supporting various multicast and unicast patterns [43], [175]. Moreover, the distribution, inter-PE communication, and collection of the outputs can be done *asynchronously* and *concurrently*. These require the interconnect switches to support *dynamic* management (priority arbitration and congestion) at a low cost. Furthermore, communication among distant PEs may be required (e.g., for store-and-forward or exchanging outputs during sparse computations). Finally, depending on sparsity and precision, the bit-width of the metadata and NZ value can differ significantly. Communicating different sizes of data and metadata can be facilitated by configurable interconnect buses and their interfacing with PEs and memory. For instance, in EyerissV2 [43], a 24-bit bus can supply PEs either three 8-b uncompressed values or two pairs of 8-b NZ and 4-b metadata. So, configurable interconnect topologies should be explored for effectively serving various communication requirements. FPGAs can also be leveraged for designing accelerators with tailored interconnects.

2) Programming of Configurable Interconnects and Design Exploration: Configurable interconnections can support various communication patterns and dynamic data movement for sparse computations. However, compilation support is needed to program them as they often contain parameterized multilevel switches and switches with many-to-many links between source and destination (e.g., [105] and [175]). Depending on the interconnect topology and optimized dataflow, the compiler may need to select efficient paths for distributing data from source to destination switches. In addition, the underlying topology (e.g., lack of multihop connectivity) *may not* support some dataflows (e.g., spatiotemporal accumulation of partial outputs from distant PEs in the absence of multihop connectivity). Furthermore, a systematic methodology for mapping communication onto interconnect topology can enable design space exploration of interconnects needed for accelerating target ML models, allowing minimum overhead of *runtime reconfiguration* of the interconnect to support various dataflows.

IX. PE ARCHITECTURE DESIGN

PE architecture consists of functional units, local memory (RFs or SRAMs), and local control [instruction buffer or finite state machine (FSM)]. Fig. 22 shows pipeline stages for processing sparse and value-approximated tensors. Depending upon PE's interface, it either gets data from the interconnect (typical) or directly accesses off-chip memory via DMA transfer. At every cycle or few, a PE: 1) processes an instruction or events based on its state [128], [164], [178]; 2) fetches data from local memory or interconnect;

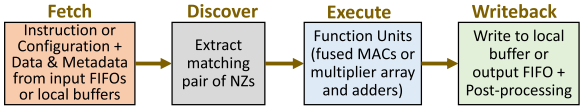


Fig. 22. ***Overview of the PE pipeline for processing sparse and value-approximated tensors. (Figure adopted from [107].)***

Table 10 PE Architectures for Sparse Tensor Computations

Scalar	[20], [30], [42], [65], [66], [68], [93], [109], [113], [114], [116], [117], [119], [121], [122], [128], [130], [133]
SIMD / Vector	[37], [41], [43], [60], [72], [105], [107], [112], [115], [118], [123], [125], [129]

3) computes tensor elements via functional unit; and 4) writes the intermediate result to local memory or interconnect. PEs may contain *special-function* modules (e.g., for ReLU or sigmoid computations [68], [115]).

Processing compressed tensors can impose significant maneuvering efforts for PE design. For example, reusing tensors temporally through local memory (e.g., in EyerissV2 [43] and SNAP [107]) alleviates overheads of repeatedly accessing compressed tensors via memory hierarchy and decoding them. However, it requires communicating data to PEs before extracting NZs. So, the PE may require additional hardware for *extracting* or *correctly indexing* NZs (see Section VI). In addition, *the selection of functional units is affected by the NNZs that can be fed for various sparsities of tensors, support for mixed-precision, and functionality of the layers.* In such various scenarios, a single dataflow may not always be effective [43], [179] and can lead to significant acceleration loss. So, PE datapath needs to be *adaptive* for supporting multiple dataflows optimized for different layers and sparsity. Furthermore, techniques for leveraging computation reuse due to *value similarity* often require enhancements in the design. PEs may also postprocess outputs or generate additional metadata for communicating outputs. So, an efficient pipeline needs to hide preprocessing and postprocessing latency.

A. Functional Units

1) Scalar PEs: Table 10 lists accelerators based on their functional units for scalar, SIMD, or vector processing. Many architectures contain an array of scalar PEs; PE datapath contains a pipelined MAC unit (e.g., EIE [42] and SparTen [116]).

2) SIMD/Vector PEs: PEs of Cnvlutin [72] and Cambricon-S [37] contain multiplier arrays and adder trees. By performing dot products at every cycle, they can deliver high throughput. Moreover, accumulation through adder trees reuses data spatially, which lowers energy consumption (by 2×–3× [129]) compared to temporal accumulation on scalar PEs by reading and writing partial summations via local memory. However, a major challenge is the inefficient utilization of multipliers and adders, which often leads to ineffectual computation cycles and acceleration loss. This is because, for high sparsity, enough NZs may not be extracted to feed all multipliers at every cycle. For example, a sensitivity analysis for Cambricon-X [41] determined that, for hyper-W-sparsity, it accelerated CONVs by about 8× (out of 16× peak speedup). The utilization may be improved by employing larger indexing or extraction modules (increased on-chip area and power). Alternatively, PEs can be designed with fewer multipliers to sustain the scalability and efficiency over a wide sparsity range.

While SIMD or vector PEs achieve spatial reuse, due to fixed designs, they are utilized poorly when executing some functionalities, such as DW-CONV. The efficiency of SIMD PEs is further affected by high sparsity, as functional units of the PE require synchronization, and there may not be enough effectual NZs to feed all of them. *Configurable functional units* can overcome such limitations. For example, PEs of SNAP architecture [107] use a configurable adder tree. It processes inputs from three multipliers and computes different combinations of partial summations. With multiple adders and multiplexers, the PE can concurrently process different partial summations (versus gather in adder tree) without high-bandwidth crossbars. Such configurable designs can support different DNN operators (e.g., DW-CONVs).

3) Multiplier-Free PEs: Accelerators, such as ZENA [113], [130], use multiplier-free PEs for high energy efficiency. These PEs process tensors of very low-precision (binary or ternary values) or logarithmic quantization. So, they replace multipliers with *simpler arithmetic*, such as 2's complement (inverters and adders or subtractors) [113], [180] or bitwise shift and additions [103], [181]. However, one challenge is to *maintain* the accuracy of DNNs, as aggressive quantization often drops top-1 and top-5 accuracy, e.g., by 0.1% [181]–5% [103]. By trading off the flexibility with simple hardware, supporting various models can be challenging.

4) Bit-Adaptive Computing: Precision requirements for targeted accuracy can vary for different models [108], [182], which can be supported by PEs with bit-adaptive computing.

a) Bit-serial computing: Albericio *et al.* [108] showed that *zero bits in NZ activations* (8- or 16-b precision) can be more than 50% and proposed the Pragmatic accelerator to leverage sparsity of activation bits. Fig. 23(b) shows the *bit-serial* computation of an inner product with AND gates, adder tree, and bitwise shift of partial output. AND gates are serially fed 1-b activations (variable precision) and bit-parallel 16-b weights (fixed precision). Fig. 23(c) shows the processing of only NZ activations in Pragmatic

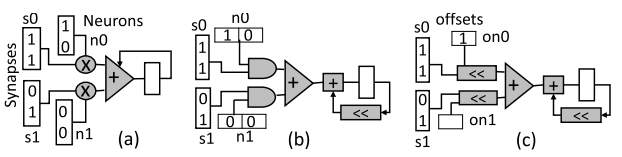


Fig. 23. ***Bit-serial processing of sparse activations in Pragmatic [108]. (a) Bit-parallel unit. (b) Bit-serial unit. (c) Bit-serial unit in Pragmatic for processing only essential bits. (Figure adopted from [108].)***

Table 11 Precision of Sparse Tensors Supported by Accelerators

binary/ternary	[113], [134]
int8	[111], [116]
int16	[20], [37], [41], [42], [65], [68], [72], [107], [112], [114], [115], [121], [123], [125], [127], [133]
logarithmic	[130], [132]
bit-adaptive	[108]–[110], [183], [185]
FP8	[66]
FP16	[66], [122], [134]
FP32	[30], [105], [134]
FP64	[93], [119]

(essential bits indicated by their positions). Laconic [183] achieved further accelerations by processing only NZ bits of both activations and weights.

b) Bit-composable computing: Bit-fusion [184] employed fusion units consisting of an array of BitBricks. The fusion units can be configured for processing multiplications of 2-, 4-, 8-, or 16-b operands. For processing NZs, PEs of CNN accelerator Envision [109] used a single-cycle N-subword-parallel multiplier, followed by an $N \times$ 48-b/N reconfigurable adder. The subword-parallel design allowed the configuration of MAC units for processing the data of 4, 8, or 16 b. SPU architecture [110] employed DGRA, a decomposable CGRA, for efficiently processing stream-join accesses. The DGRA PE and interconnect switches enabled decomposing up to four 16-b subword operands. DGRA also supported accessing subword data from the scratchpad. For DNN training with mixed-precision and sparse tensors, PEs of LNPU contained configurable MAC units that can process FP8 or FP16 tensors. Table 11 lists precisions of sparse tensors that are supported by different accelerators. The precision indicates the bit-width of input operands (activations and weights). For MAC operations, accumulators usually produce high-precision output, which can be down-scaled or truncated afterward.

5) Clock-Gated PEs: PEs can be clock-gated when *not used* for executing a layer and for ineffectual computations. For example, Eyeriss [20], Thinker [128], and Minerva [74] use zero detection for clock gating the datapath in the PE pipeline. PEs check whether the value being read is zero (or compare with a threshold, e.g., in MNN-Fast [131]). Based on the comparator output, their clock gating logic prevent the MAC datapath from switching in the consecutive cycle, which reduces energy (e.g., it saved power consumption of Eyeriss PE by 45%). Zero-skipping through flags in Envision [109] and Sticker [164] achieved similar savings.

6) Optimization Opportunities: Exploring efficient designs of functional units for various sparsity ranges/patterns and functionality: The utilization of vector/SIMD units can drop significantly due to unstructured sparsity [107] and functionality beyond structured dot products (e.g., DW-CONV). So, for exploring design hyperparameters, such as the number of functional units, designers need to consider the impacts of sparsity, data extraction mechanism, required synchronization among computational units, and configurations required to support various functionalities. Moreover, for low sparsity, designs should deliver performance *at par* with a sparsity-oblivious design. For example, for processing dense tensors, SCNN [115] achieved 79% of the performance and consumed 33% higher energy compared to the baseline accelerator for processing dense tensors. So, designers may ensure that additional features for exploiting sparsity and configurable components do not increase the critical path latency and are power-gated if not used.

B. Dataflow Mechanisms

1) Background: The efficiency of executing a layer onto a hardware accelerator depends on the computational, communication, and memory access patterns, which are commonly referred to as dataflow mechanisms [44], [56]. A dataflow refers to the spatiotemporal execution of a model layer (nested loop) on architectural resources [23], [56]. Here, spatial execution corresponds to how PEs exploits parallelism in the computation graph and processes different subsets of tensors. Temporal execution drives the data accessed throughout memory hierarchy and data communication via interconnects. So, depending on the functionality and tensor dimensions, dataflow can significantly impact the utilization of resources, data reuse, and latency hiding for memory accesses and data communication, and, consequently, the execution time and energy consumption [43], [44], [56], [57], [186].

One way to classify dataflows is by what data are kept "stationary" in registers or local memory of PEs (and reused fully before eviction), while other data are being iterated over. Some commonly used dataflow mechanisms are output stationary, weight stationary, input stationary, row stationary, and no local reuse. Fig. 24 shows an example of convolution and the layout of the stationary data for mapping the convolution with these dataflows. In weight stationary dataflow, each weight (of a 2-D filter)

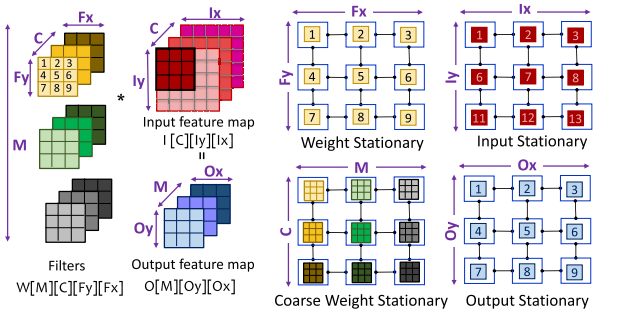


Fig. 24. ***Commonly used dataflow mechanisms for executing convolution layers on hardware accelerators.***

remains stationary on a unique PE, and reused many times, during processing activations (corresponding to the same input channel C). By processing a unique weight, each PE produces partial summations for output activations, which are communicated by PEs and accumulated before outputs are written back to the memory hierarchy. So, input and output activations are accessed from off-chip memory (via shared scratchpad and PE's local memory) several times, while weights are continuously reused. After reuse, a new set of weights is loaded from memory, and the execution repeats. Weight reuse is higher in processing larger feature maps (CNNs) and multifolded for processing data in a batch (e.g., images for CNNs and tokens of sentences for NLP models). Fig. 25 lists such characteristics of different layers.

Dataflows can be applied at a coarser level, where PEs process a data block or plane (1-D/2-D/3-D). In a coarse weight stationary approach [23], each PE processes weights of an entire 2-D filter (dimensions C and/or M are laid out spatially on PEs). Rows and columns of PEs process the data corresponding to unique input and output channels, respectively. So, activations need to be multicast to the PEs of a row, different weights need to be provided to each PE, and partial summations for output channels can be accumulated vertically [23]. Similarly, in an input stationary dataflow, unique activations (or blocks of input feature maps) remain stationary and are reused. In an output stationary dataflow, each PE produces a unique output activation (corresponding to the same or different output channel) [44]. By processing spatial data and input channels first, partial summations are accumulated and reused in the memory of each PE. With the temporal accumulation of outputs on PEs, the output stationary dataflow does not need to reduce partial outputs spatially by collecting them from appropriate PEs, which is otherwise challenging for unstructured sparse data (see Section VIII-B). Therefore, many accelerators opt for such dataflow. In no local reuse dataflow, input operands are streamed to PEs, but they are not stored in PE's memory [22], [44]. In row stationary dataflow, PEs of the same row process the same weights (a row of a filter), diagonal PEs process the same row of IAs, and partial summations for rows of the output feature map are accumulated through vertically connected PEs [20]. So, different dataflows uniquely exploit the spatial parallelism and reuse of different tensors.

Dataflow optimization: As dimensions of tensors are often large, many ways exist for spatiotemporally executing a layer onto the computational and memory resources of an accelerator. The optimization of the dataflow is important as it can significantly impact the performance and energy consumption [23], [56], [57]. For instance, mappings with similar performance can consume an order of magnitude higher energy [186] or vice versa. Furthermore, as shown in Fig. 25, reuse characteristics, tensor dimensions, functionality, and sparsity can vary significantly for different DNN layers. Hence, a single dataflow may not always be effective for acceleration. Fig. 26 provides two such examples that lead to low PE-array utilization. The coarse weight stationary dataflow processes different 2-D filters on different PEs. So, it is inefficient for DW-CONV. Similarly, output-stationary or input-stationary dataflows can result in low utilization of PEs for processing later layers of deep CNNs. With the vast space of execution methods and the growing development of new models (with variations in tensor dimensions), it becomes hard for nonexperts to figure out optimized execution methods and designs. Therefore, many optimization tools have been proposed recently including Timeloop [186], dMazeRunner [56], MAESTRO [57], and Interstellar [23]. They analytically model the execution of accelerators to estimate execution metrics and evaluate a set of mappings from the pruned space of various dataflows.

Layers	Examples	Characteristics	Implications on Execution
Early CONV	AlexNet / MobileNet CONV1	Large spatial feature maps	High weight reuse
		Less filters	Low input reuse
		Less channels	Low reuse of partial sums
		Low W and IA sparsity	Higher data movement
Last CONV	ResNet-50 CONV4/5_x	Small spatial feature maps	Low weight reuse
		More filters	High input reuse
		More channels	High reuse of partial sums
		High/Moderate W/IA sparsity	Usually compute-bounded
MLP	VGG-16 FC, FC0 in encoders of BERT	Smaller activation vectors	No weight reuse w/o batching
		Large weight matrix	Memory/comm. bounded
		High/low W/IA sparsity	Reduced data movement
Depth-wise CONV	MobileNets d/w, Xception, EfficientNetB0	Single filter	Low reuse of inputs
			Low reuse of partial sums
		No channel-wise reduction	Low computation, high communication req.
		Unpruned, fewer parameters	Inefficient PE-array utilization w/o group-parallel processing
Point-wise CONV	MobileNet s/1, Fire module in SqueezeNet	1x1 convolution kernel	Reduced reuse of W, IA
		Moderate sparsity	structured sparsity limited to channel and filter direction
Residual Layers	ResNet-50, U-Net	Concatenation of outputs	additional off-chip accesses
		additional inputs from previous layers	opportunity for reusing intermediate feature maps
Group CONV	ResNeXt Aggre--gation Blocks	Parallel paths due to cardinality	opportunity for input reuse with fused executions
3D CNN	C3D	Temporal processing across frames	heavy computations, increased data reuse
Attention	Encoders in Pruned Transformer, BERT	Highly/hyper sparse weights in query, value processing	Reduced computations and data movement
		Multiplications of dense, small matrices for each head	High collective data movement for all heads

Fig. 25. ***Characteristics of different DNN layers pertaining to hardware execution. (Figure inspired from [43] and [57].)***

2) Sparsity-Aware Dataflows: Dataflows for processing sparse tensors are typically similar to those for dense tensors while processing the data in compressed format. For correct functionality, dataflow executions are facilitated by extraction/orchestration of NZs, which is done either in PEs [43], [115], on a different module [37], or by a separate controller. For example, SCNN [115] used PT-IS-CP dataflow. It processed planar tiles of feature maps with input stationary dataflow. SCNN's PT-IS-CP-sparse dataflow extended the PT-IS-CP. It processed only NZ activations and weights in the compressed format while accessing them from memory and performing computations. The coordinate computation module in each PE ensured that

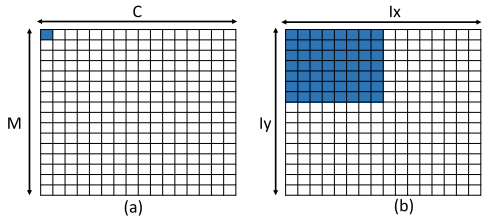


Fig. 26. ***Low utilization of a 16 × 16 PE-array in (a) coarse weight stationary dataflow when executing depthwise layers and (b) input stationary dataflow for executing later layers of deep CNN models. (Figure inspired from [43].)***

partial products generated by all-to-all multiplications of NZ inputs and weights were accumulated correctly and stored in appropriate buffers. Table 12 lists sparsity-aware dataflow mechanisms used by accelerators.

EyerissV2 [43] used an enhanced row-stationary dataflow. By using statically known sparsity of weights, more NZ weights were allocated in local memories and global scratchpads. For example, each PE can store up to 192 NZ weights. Mappings of CONV and FC layers of AlexNet with row-stationary dataflow allocated 64–174 NZ weights, which corresponded to a total of 132–480 weights in the dense format. With in-PE data extraction logic, each PE only processed NZ values from CSC-encoded data. So, sparsity-aware dataflow can be optimized with the preknown (or expected bounds of) sparsity and value similarity.

3) Optimization Opportunities:

a) Dataflow optimizations accounting for storage and computational overheads for metadata and codebooks: Sparse and value-shared tensors are processed along with metadata (indicates positions of NZs) and codebook (common values shared among tensor elements), respectively. It requires additional processing, e.g., buffer management, communication via interconnects, and indexing the appropriate values. Depending on the dataflow, such processing can amplify the execution costs, which needs to be optimized. Existing tools for optimizing dataflows target dense tensor computations. Accelerators EIE, SCNN, and Cambricon-S process sparse tensor computations but with customized dataflows. Hence, frameworks for mapping and design explorations need to consider the sparsity and value similarity of tensors and their variations across layers/models. Such tools can include additional costs corresponding to storage, communication, and extraction in their analytical models. Explorations supporting multiple dataflows can help to achieve efficient designs for handling different functionality and variations in sparsity, shapes, and quantizations of tensors.

Table 12 Dataflow Mechanisms of Accelerators

Input Stationary	[105], [113], [115]
Output Stationary	[30], [37], [41], [42], [60], [65], [66], [68], [72], [93], [107], [109], [112], [116]–[118], [122], [123], [133], [134]
Weight Stationary	[105], [126], [127]
Coarse Weight Stationary	[72], [114]
Row Stationary	[20], [43]

Table 13 Techniques for Leveraging Value Similarity

Value sharing	Weights	[37], [42], [98], [117], [189]
	Activations	[99], [190]
Computation reuse and memoization	Partial	[98], [99], [125], [189]–[192]
	Full	[149], [188], [193]
Early termination of computations		[194]–[196]

b) Sparsity-aware resource partitioning: Acceleration of deep learning models is scaled by simultaneously processing multiple layers. It is done either by partitioning resources [171] of a scaled-up accelerator or on multiple accelerators (scale-out) by leveraging model- or data-parallelism [187]. Techniques for resource partitioning aim to highly reuse data from the on-chip memory of accelerators. It involves evaluating many-to-many mappings between layers and accelerators. Such optimizations can be crucial for several applications that require low latency, real-time processing, or high frame rates (e.g., processing the frames for multiple object detection models of an autonomous vehicle's perception system). Exploiting sparsity can provide further opportunities due to fewer computations, communications, and storages.

C. Leveraging Value Similarity

Several techniques have leveraged value similarity for accelerating DNNs by value sharing and computation reuse (see Table 13). Video frames exhibit high similarity spatially (among neighboring pixels) and temporally (over consecutive frames) [99], [188]. After precision lowering, values of limited range repeat frequently [27], [98], which are further compressed by maintaining a codebook of unique values [42]. With repetition of values, computation (outputs) can be reused, either partially during processing a layer [98], [99] or by skipping processing of a whole layer [149]. This section describes such techniques and corresponding hardware enhancements.

1) Weight Similarity: Prior studies have shown that weights can be approximated with a small set of values. Hegde *et al.* [98] showed that, for 8-b weights of DNNs, each NZ value mostly repeated more than ten times and even more than 100 times in later layers of AlexNet and ResNet-50 models. Han *et al.* [27] pruned weights of DNNs with k-means clustering for value sharing. Shared unique values were represented with 4 or 5 bits without dropping classification accuracy. Local quantization (applying clustering separately over different subtensors) can achieve even smaller codebooks [37]. Leveraging the weight similarity can compress pruned models further by up to an order of magnitude [27], [37].

Value-shared weights are processed by augmenting the PE datapath with a weight decoder (e.g., in EIE [42]). For processing NZ weights, the PE provides the encoded index

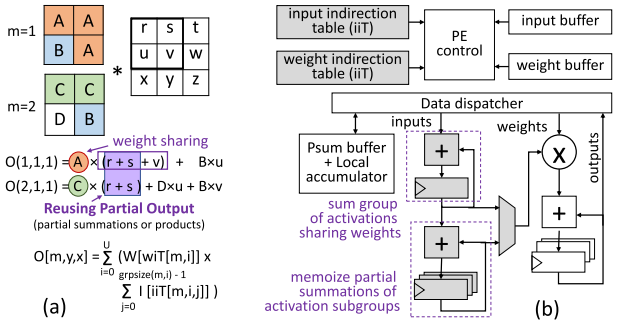


Fig. 27. ***(a) Leveraging weight similarity and reuse of partial outputs [98]. (b) Modifications in UCNN PE architecture (shaded blocks) for buffering indirection tables, partial summations of activation groups, and memoization of partial outputs. (Figure adopted from [98].)***

of the weight to the decoder and obtains shared value. Depending on the lookup mechanism and total bits to be extracted at every cycle, the decoder can incur considerable area and power costs (e.g., for Cambricon-S [37], 32.56% and 3.98% of the total on-chip area and power, respectively).

2) Input Similarity: Audio or video frames can contain high similarity spatially or temporally. This is because a speech signal can be quasi-stationary for a short interval. Also, successive executions of DNNs process overlapping windows of audio frames for context extraction [99]. Feature maps for CNNs exhibit high spatial correlation [190]. High input similarity enables only storing unique values and reusing computations by *differential computing* over nonsimilar data.

Riera *et al.* [99] showed that, after uniform linear quantization of inputs of DNNs (e.g., C3D [6], EESEN [197], and CNN for self-driving cars [198]), about 61% of IAs are the same as previous execution, and 66% computations can be avoided. Their accelerator maintains centroids of quantized inputs and the index corresponding to each input element. Then, consecutive frames are processed layerwise with differential computing. For example, for each activation of an FC layer (of a new frame), the accelerator calculates centroid and index, and then, it compares calculated centroid to memoized centroid. If the difference is zero, then the output from the previous execution is reused, and the next activation is processed. Otherwise, a new value of the index is updated in the buffer, and new values for output activations are computed by accumulating multiplications of weights with the difference.

3) Computation Reuse (Partial During Processing a Layer): UCNN [98] leverages the repetition of weights by forming activation groups (summations of activations) that share the same weight. It also reuses activation subgroups, i.e., memoizes partial summations of activations that can repeatedly appear across different filters. Fig. 27(a) illustrates an example. Weights A and C can be shared among corresponding activation groups. For producing activation groups, subgroups, such as $(r + s)$, can be reused with memoization. So, an output activation is calculated by indexing a unique weight value and corresponding activation groups. Indirection tables provide indices of the unique weight and grouped activations. Fig. 27(b) shows corresponding modifications in the PE datapath. UCNN reported up to 24% area overhead for a PE and 1.8× speedup for CNNs compared to execution on a baseline accelerator without exploiting weight repetition.

Silfa *et al.* [191] showed that, for RNNs (e.g., DeepSpeech2 [199] and EESEN [197]), the relative difference between the output activations over consecutive frames was about 23%. Leveraging temporal similarity of outputs saved about 24% computations with negligible accuracy loss. For predicting whether an output activation leads to a value similar to the previous output, their technique extended each RNN layer with a binary neural network (BNN). With BNN outputs correlating to actual outputs, execution of much smaller BNN layers led to an efficient prediction of the temporal output similarity.

4) Computation Reuse (Skip Processing of Entire Layer): A few techniques predict outputs based on previous computations and skip heavy computations of some layers. Gonçalves *et al.* [188] showed that 18%–81% of computations in AlexNet CONV layers could be reused due to spatial (intraframe) and temporal (interframe) redundancy of the inputs. They leveraged such reuse with memory look-ups and avoided executing CONVs. For YOLO-v3 [4], it processed only 22%–32% frames while incurring negligible accuracy loss. Buckler *et al.* [149] proposed skipping heavy processing of some CNN layers for several frames (predicted) and executing precise computations periodically for remaining (key) frames. For predicted frames, their algorithm estimated motion in the input frame. It used results for incrementally updating the output saved from the last key frame. Unlike other value similarity techniques that incur changes in PE datapath, such techniques can be efficiently executed on a separate module (e.g., EVA^2 [149]) or coprocessor, while other modules of the same or different accelerator process sparse tensors of DNN layers. EVA^2 identified 78%–96% of the frames for AlexNet and 40%–71% of the frames for Faster-RCNN as predicted frames while processing the YouTube-BoundingBoxes dataset [200].

5) Early Termination of Computations by Predicting Outputs: SnaPEA [194], SparseNN [195], and CompEND [196] reduce ineffectual computations by early prediction of the usefulness of outputs. They check whether computations contribute to the effective inputs for the subsequent layer (e.g., ReLU or max-pooling). If not, their PEs terminate such computations. To reduce computations corresponding to output sparsity, Akhlaghi *et al.* [194] statically reordered weights based on their signs. PEs of its SnaPEA architecture contained prediction activation units (with each MAC), which checked the sign-bit of the partial

summation and raised a termination signal to notify the controller as the sign-bit was set.

6) Optimization Opportunities:

a) Joint exploration of spatial and temporal similarity of inputs, weights, and outputs: Depending on the model's configurations (depth, layer dimensions, and cardinality) and domain-specific data, opportunities for value sharing and computation reuse (at both fine-grain and coarse-grain levels) in processing activations and weights can vary considerably. A joint exploration for different tensors can help to identify storage-Ops-accuracy tradeoffs for efficient acceleration.

b) Leveraging value similarity through separate processing: Determining value similarity and leveraging computation reuse often demands modifications in PE-array, increasing the accelerator's area, latency, or power. Designers may obviate it by providing a separate and compact module for differential computing that handles necessary preprocessing or postprocessing and can be interfaced with the PE-array and on-chip memory. Upon requirement, it can trigger execution on PE-array for structured computations. Furthermore, algorithms expressing the functionality of ML layers/models may be defined in terms of *differential computing* (i.e., execution is conditional to the input mismatch, reused otherwise). With efficient accelerator/model codesigns for differential computing of tensors, accelerators may attain structured effectual computations with fewer overheads of metadata or memoization.

X. LOAD BALANCING OF EFFECTUAL COMPUTATIONS

Depending on the distribution of zeros, the inter-PE or intra-PE imbalance can cause low utilization of PEs or their functional units, which increases execution time and energy consumption. This section summarizes sources of such imbalance, and then, it discusses different software-directed techniques or hardware designs for balancing the computations. Table 14 categorizes these techniques. Software-based techniques facilitate structured computations by forming local regions of dense elements, sorting the data by combining same-sparsity tensor blocks, or regularizing models with structured pruning. Although requiring low/no additional hardware, they are often limited to static W-sparsity. Accelerators dynamically balance computations by prefetching work in FIFOs or memory, obviating fine-grained synchronization of computations on PEs. Some accelerators achieve further runtime balance across PEs by a central hardware module for work sharing.

A. Sources and Impact of Imbalance

1) Inter-PE Imbalance: Zeros in different tensors can be scattered, and their positions may not be determined statically (e.g., unstructured IA-sparsity). For most accelerators, work to be performed concurrently by PEs is fixed statically. Also, executions with conventional dataflows usually require synchronization among PEs (e.g., in SCNN [115] and Cnvlutin [72]), which is achieved by barriers implemented in software via instructions or in hardware via PE architecture or controller logic. Consequently, computations per PE during each execution pass can vary drastically (inter-PE load imbalance). So, many PEs may finish their computations early, get stalled, and wait for the next set of data due to synchronized execution, while other PEs still process the previously allocated data. It increases execution time and energy consumption. Kim *et al.* [130] analyzed the distribution of NZ weights in AlexNet CONV3 filters and showed that, in an execution pass, NZs processed by the leading and trailing PEs differed by up to 6.5×. Similarly, up to 40% cycles were idle for executions of PEs in SCNN architecture [115]. The sensitivity analysis for EIE showed that, without any load balance, about 47% of the cycles were idle for the 64-PE accelerator [42].

Table 14 Classification of Load Balancing Techniques

Software Directed	Data Clustering	[65], [126], [127], [136]
	Data Reorganization	[116], [123], [130]
	Model Regularization	[37], [60]–[62], [68], [118], [137]
Hardware Module	Work Prefetching	[41], [42], [68]
	Work Sharing	[66], [89], [130], [137]

2) Intra-PE Imbalance: For SIMD or vector PEs, intra-PE load imbalance can also contribute to a significant acceleration loss. With unstructured sparsity of one or more tensors, enough NZs may not be extracted to feed all the functional units within PEs, which causes intra-PE load imbalance. The sensitivity analysis for the SNAP accelerator showed that, with moderate sparsity, the utilization of multipliers falls below 80% and up to 20% for 90% sparsity [107]. Similarly, SCNN [115] reported below 80% utilization of multipliers for all GoogLeNet [96] layers and 20% for the last two inception modules. Moreover, a few architectures use PE designs with multiple subunits in each PE. For SIMD processing, a subunit works in synchronization with other subunits of the same PE, e.g., in Cnvlutin [60], [72], [112]. With unstructured sparsity, multipliers and accumulators in some subunits can often be idle, while trailing subunits process computations.

B. Software Directed Load Balance

1) Clustering of NZs for Populating Dense Data Regions: As described in Section VI-A, a few techniques targeted high W-sparsity. They used structured pruning or data combining approaches for clustering the tensor elements in locally dense regions that can be dispatched to PEs for processing in a conventional manner [126], [127]. So, they achieve high PE utilization and lower invocations to accelerator resources. However, such techniques may not be effective when algorithms cannot generate or pack structured sparse data (e.g., dynamic unstructured sparsity).

Concise convolution rules (CCRs) [65] partitioned sparse convolutions into effective and ineffective subconvolutions for processing locally dense regions of filters and

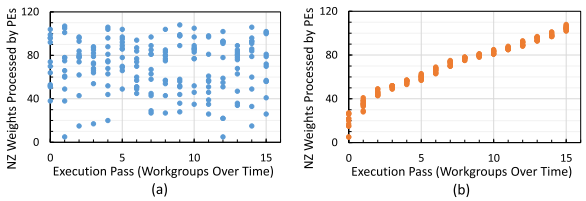


Fig. 28. ***Distribution of NZ weights for executing CONV1 of AlexNet [201] with coarse weight stationary dataflow on a 4 × 3 PE-array. Distribution shows NZ weights in different workgroups (each workgroup contains NZs for 12 PEs): (a) without load balance and (b) after sorting. (Figure inspired from [130].)***

input feature maps. It eliminated a majority of ineffectual computations and their storage (for VGG-16, achieving the reduction of about 79% and 51%, respectively) [65]. Subconvolutions after CCR transformation were executed on the SqueezeFlow accelerator [65]. However, with PEs performing only all-to-all multiplications, it may not support generic tensor operations; it can be challenging to extend CCR methodology for other algorithms.

2) Data Reorganization Before Work Allocation: In ZENA [130], each PE processed a different set of filters for processing a subworkgroup. For balancing computations among these PEs, filters were sorted by sparsity and allocated to PEs such that all PEs executed filters of similar sparsity.

To determine the efficacy of such sorting, we considered AlexNet [201] for ImageNet classification. We obtained the pruned model through the neural network distiller [202] with a pruning algorithm similar to [25]. For accelerating AlexNet [201] CONV1 layer with coarse weight stationary dataflow, Fig. 28 presents distributions of NZs in filters before and after reorganization. For processing 64 filters of size 3 × 11 × 11 on 4 × 3 PEs, we consider execution through 16 different workgroups. Each workgroup contains NZ weights for concurrent processing of four filters and three channels on 12 PEs (up to 11 × 11 on a PE). The next workgroup is initiated once all PEs entirely use previously allocated weights. Fig. 28(a) shows that, before data reorganization, the total NZ weights allocated to PEs within workgroups differed by up to 21.4× (5 versus 107 for 11 × 11 filters) and 6.09× on average. Fig. 28(b) shows that, after sorting the weights (both filterwise and input channelwise), it leads to an almost equal NNZs for computations onto 12 PEs during each workgroup. The total allocated NZ weights differed by only 1.36×.

After static sorting, ZENA achieved about 20%–32% more acceleration for CONV layers of AlexNet and VGG-16 [130]. Depending on the sparsity, distribution of NZs, and achievable sorting granularity, the work allocated to PEs may differ considerably even after sorting. Moreover, such transformations are usually feasible only statically. So, ZENA also used dynamic work sharing, which we discuss in Section X-C.

3) Accelerator-Aware Regularization of the Model: Recent accelerators, including [60] Sparse Tensor Cores in NVIDIA Ampere architecture [61], [118], execute models pruned with k:n block-sparsity (e.g., 2:4 sparsity supported by Ampere [61]). Their PEs contain multiplexers that use indices of k NZ weights to select k out of n dense activations. Then, functional units process extracted values. Like k:n block-sparsity, ESE [68] used a load-balance aware pruning for RNNs. It considered submatrices to be processed by different PEs and induced the same sparsity into all submatrices.

In some architectures, all PEs receive the same set of NZ activations. They process them with their unique weights and produce distinct output activations. One such architecture is Cambricon-S [37] that used a coarse-grained pruning of weights. The block size for pruning depends on the total number of PEs (16). Over local regions, the pruning removed all connections between an IA and all (16) output activations. So, when PEs processed output neurons, they processed the same NNZ IAs and weights for computing MACs.

C. Load Balancing With Hardware Structures

1) Facilitating Asynchronous Computations by Prefetching Allocated Work: One way to improve PE utilization (in the presence of load imbalance) is to prefetch the allocated work for PEs and avoid their fine-grain synchronization. So, even if there is a different amount of work (e.g., MACs per IA), all the PEs may perform effectual computations at the same time (e.g., work on different activations). So, each PE can be engaged in performing some computations, before it runs out of the available data. This can be achieved by offloading more data into the FIFO or memory of each PE. For example, in EIE [42], activations are broadcast to FIFOs of all PEs. Once a PE finishes multiplying an activation to corresponding NZ weights or does not find any matching weights, it processes the next activation from its queue. The FIFO size of 8 or higher ensured each PE almost having an NZ activation to process (during 87%–91% of computation cycles) and lowered idle time of PEs from about 47% to 13% [42].

Cambricon-X [41] allows asynchronous communication of weights to PEs. A centralized data extraction mechanism provides NZ activations to each PE via a unicast network, and compressed weights are loaded in the memory of each PE (2 kB). The memory access port is assigned to each PE for a short period, where it fetches several chunks of weights via DMA transfer. Depending on the prefetching interval and unstructured sparsity, each PE may asynchronously work on useful computations in most of the execution cycles.

While asynchronous execution improves the utilization of PEs, the work allocated to PEs is still fixed. Plus, in-PE data fetching mechanisms may restrict PEs from finding the pending work in other PEs and sharing it. For highly imbalanced computations, straggling PEs can still be the bottleneck.

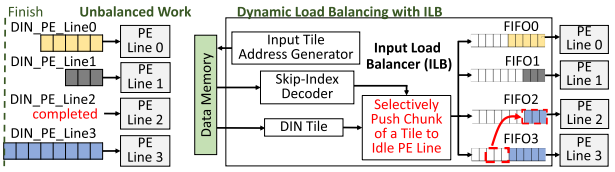


Fig. 29. ***Load balance mechanism in LNPU [66]. (Figure adopted from [66].)***

2) Centralized Load Balance: In some accelerators, data are multicast to one or more rows (or columns) of PEs. A central logic processes the metadata (indices) of the tensor tiles to be distributed along with control signals from PE-rows and finds out work distribution. Then, it feeds the fast-acting rows/lanes of PEs and facilitates work sharing. For instance, ZENA [130] allocates work dynamically through down counters. Different PE-groups (e.g., PE-rows) process the same filters with different activation tiles. A central distribution mechanism contains down counters that store the number of remaining activation tiles for each PE-group. When a leading PE-group finishes its work (counter is zero), it obtains an activation tile from a straggling group (has the biggest count value) and then continues processing output activations. The work sharing improved the acceleration by about 10% for CONV layers of AlexNet and VGG-16 [130]. Memory port contention may occur when multiple leading groups simultaneously attempt to fetch the same set of IA tiles. ZENA's execution mechanism overcomes this problem by reassigning only one activation tile at a time (to the leading group) and performing reassignments only during bus idle time.

LNPU [66] uses an input load balancer (ILB), which is shared among PE-rows. As shown in Fig. 29, ILB contains address generator units to determine the indices of the compressed elements that need to be fetched. Once ILB fetches them, the skip-index decoder unit determines the appropriate indices for data extraction. It pushes them along with the NZ values into the FIFO of a PE-row. It also calculates bitmaps, which are used for pushing the data (indices and NZs) selectively into FIFOs of PE-rows at run time. Due to ILB, PE utilization in LNPU was increased by 2%–26% for 10%–90% sparsity of the inputs (activations or their gradients) [66]. So, centralized load balancing mechanisms can leverage the information about data allocation for PEs and provide equal work to PEs or feed the fast-acting PEs during runtime

D. Optimization Opportunities

Software-level or hardware/software/model codesign optimizations for low-cost load balance: Most accelerators lack special support to balance computations among PEs, e.g., to avoid area and power overheads due to special hardware logic.

1) One technique is to reorganize the data [116], [130]. However, it can mostly exploit only static W-sparsity for inference at no/low hardware cost. So, we may require additional codesign optimizations for regularizing dynamic sparsity.
2) For preknown or estimated sparsity, sparsity-aware mapping optimizations for accelerators may identify efficient dataflows that sustain high PE utilization.
3) When sparsity may be regularized at modest accuracy loss (e.g., for several DNNs), accelerator/model codesigns can induce the balance. It can be either done via structured pruning of activations or refactoring the operators (nonlinear activations, batch normalization [77], or quantization of outputs). Consequently, the codesigns may achieve structured computations over both activations and weights to an extent, leading to further accelerations.

XI. WRITE-BACK AND POSTPROCESSING

Once PEs process allocated data, they write (partial) outputs back via interconnect. For unstructured sparsity, managing *write-backs (WBs)* can be challenging because different PEs can produce outputs of different sizes at different times. Moreover, operations, such as ReLU, pooling, and batch-normalization, need to be performed on outputs. They are usually not performance-critical, such as CONV or MLP. So, they can be either executed on PEs before WBs of outputs (in SCNN [115], Cambricon-S [37], and EIE [42]) or postprocessed on central modules (in MAERI [177], ZENA [130], and SqueezeFlow [65]). Central modules often assemble the outputs collected from PEs, transform data for the next layer, and encode sparse outputs on the fly.

A. Write-Back From PEs

1) Simultaneous WB: Cambricon-X [41] and SCNN [115] use fat-tree networks or point-to-point links, which allows simultaneous WBs from multiple PEs. Whenever ready, PEs can execute in a dataflow manner and immediately write outputs back after computations. This is important for processing unstructured sparsity because different PEs may process different NNZs and produce different amounts of output values for WB at different time intervals. With such high bandwidth, communication time can be reduced and interleaved with computations, which is important for processing models with low arithmetic intensity. These PEs write to a central module for postprocessing (e.g., in Cambricon-X [41]), the on-chip memory [41], or off-chip memory (e.g., in SCNN [115]). Although simultaneous WBs are faster, such a fat-tree network can incur considerable overhead due to increased bandwidth and inefficient bandwidth utilization in some scenarios. So, accelerator designs can instead use a common bus that is time-shared among multiple PEs; PEs can write the data back turnwise or asynchronously.

2) Sequential WB: PEs in several accelerator designs operate in a lock-stepped manner, where data blocks common to PEs are broadcast to them, and all PEs synchronize for processing the outputs (idle when done). Synchronized execution can allow WB in a specific sequence (e.g., a PE with the lowest PE-index writes the data first and so forth). It makes the programming of the accelerator easier. It also obviates overheads of specialized hardware/software support, which is required otherwise for asynchronous WB.

3) Asynchronous WB: With unstructured sparsity, PEs process a different amount of data and can asynchronously request WB during the execution. For facilitating such support, accelerator designs can employ additional hardware logic. For example, ZENA [130] used a common bus for multicasting blocks of filters and feature maps to PEs and collecting the output. Output buffers of PEs were flushed to the memory during the idle period of the bus, which avoided bus contention between broadcasting activations from memory and WB of partial summations. For prioritizing the requests from PEs to access the bus for WB, it determined the PE groups with a high number of pending output tiles.

B. Data Assembling

PEs often process large output tiles. So, they perform fine-grained assembling of outputs locally. For example, SCNN [115] PEs use a coordinate computation unit that determines appropriate indices for arbitrating partial outputs to the local accumulator buffer. In other accelerators, PEs produce metadata and supplies it with outputs for correctly indexing the memory (e.g., in ZENA [130]) or assembling outputs on a central module (e.g., in Cambricon-X [41] and CoNNA [114]). The central module uses the metadata (e.g., output coordinates) from PEs or preknown indices of PEs to assemble collected outputs before WB or postprocessing. In some designs, data assembling is done by global accumulators that reduce partial summations and update outputs into appropriate memory banks (e.g., SNAP [107]). The data assembling logic typically also handles data layout transformation (e.g., in [111] and [114]), which is required for processing the subsequent layer.

C. Data Layout Transformations

1) Data Reorganization: Accelerators are often designed for efficient vector or matrix multiplications. So, for processing convolutions, they (e.g., [72] and [111]) require data layout in *NHWC (channel-first)* format [203], which is also used for processing on CPUs and GPUs. Fig. 30(b) shows data reorganization for striding execution of the convolution of Fig. 30(a). It shows iterative processing of the spatial data with channel-first processing. For example, an output activation $1A$ can be processed by fetching a block containing all channels of the first filter and ifmap. Vectors corresponding to channels can be processed iteratively. Sparse data blocks are also processed similarly but with computations on appropriate NZs.

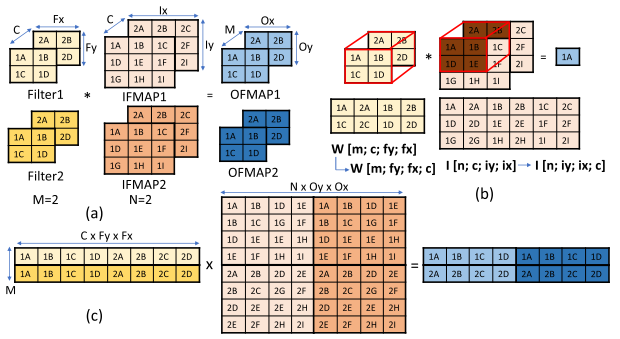


Fig. 30. ***Data layout transformation for executing convolution. (a) Convolution of two 2 × 3 × 3 feature maps with two 2 × 2 × 2 filters. (b) Reorganizing data for striding execution. (c) Transforming feature maps into the Toeplitz matrix.***

2) Transformations to Toeplitz Matrix: Processing with NHWC format allows executing CONVs as iterative vector–vector multiplications, but it requires hardware support to fetch appropriate blocks. So, for processing CONVs as sparse GEMMs, a few accelerators, including ERIDANUS [126] and [127], transform sparse feature maps into Toeplitz matrix with *im2col* transformation [204]. Once transformed matrices are obtained and sparsity-encoded, accelerators compute sparse matrix multiplication. Fig. 30(c) illustrates the transformation for tensors of Fig. 30(a). It shows that neighborhood values for computing an output with a 2-D CONV are combined as a vector. For multiple input channels of ifmaps or filters, corresponding elements are stacked in column-vectors or row-vectors. However, transforming ifmap into Toeplitz matrix duplicates neighborhood data and yields storage overhead (Fy×Fx times higher memory for unit-strided CONV).

D. On-the-Fly Encoding

Several accelerators, such as SparTen [116], SqueezeFlow [65], Eyeriss [20], and CompAct [111], use an encoding module. Such a module encodes blocks of output tensor on the fly and typically before WB. It reduces access to off-chip memory significantly [20], [65]. On-the-fly encoding allows efficient processing of dynamically sparsified tensors, i.e., sparse activations for DNN inference and tensors in the training of models. It typically consumes low on-chip area and power, e.g., 2.5% and 3.06% for the RLC encoder–decoder unit in SqueezeFlow [65] and 0.3% of the total on-chip area for the RLC unit in Eyeriss. The complexity of the hardware logic required for encoding depends on the coding format (see Section V). For example, single-step processing for bitmap, RLC, or COO-1D incurs low overhead. A central bitmap-encoder in SparTen consisted of comparators (XNOR gates) for determining NZs and additional logic for shifting NZs to populate data

vector. The encoding overhead may be lowered for block-sparse tensors, which requires indicating only positions of blocks of NZs.

Sticker [117] facilitates sparsity-aware encoding. It uses three modes to encode DNN tensors of high, medium, or low sparsity with COO, bitmap, and dense format. The three modes are controlled by two threshold values. Since weights can be processed offline for DNN inference, they are preencoded in appropriate formats. To encode activations online, Sticker uses a sparsity adaptor module. It consists of a sparsity detector, a 4-kB buffer, an encoder, and a controller. The sparsity detector contains counters that count zeros in activations of consecutive 16 channels. After the detector processes output activations (obtained after ReLU), they are stored in the buffer. Then, the controller determines the encoding mode with which the encoder can encode the data in the buffer.

XII. COMPILER SUPPORT

This section provides an overview of the compiler support for sparse deep learning accelerators. It focuses on the following.

1) *IRs:* They determine what type of code the compiler can support and what kind of compiler transformations it can perform.
2) *Support for sparse tensors:* This section discusses compilation challenges in supporting sparse deep learning and compilers developed to overcome these challenges.
3) *Compiler optimizations:* This section provides an overview of state-of-the-art techniques that allow the compiler to apply advanced optimizations and generate the most efficient code from high-level neural network descriptions.
4) *Accelerator ISAs and code generation:* This section focuses on accelerator ISAs (e.g., instruction set for high-level tensor operations) and the compiler support for machine code generation for accelerators.

A. Intermediate Representations

IR determines which types of code can be represented by the compiler, whether it can support sparse tensor computations, the types of code transformations that can be done, and even the scalability of the compiler.

1) Need for High-Level Representations: A common example of low-level IR is LLVM IR, which is well suited for low-level code optimizations, such as register allocation, but not for many high-level code optimizations needed for optimizing sparse deep learning. This is mainly because low-level IRs do not preserve information about loop structures and data layouts, and reconstructing such information is not trivial [205]. That is why many deep learning compilers, such as TVM [206], Tiramisu [205], and Halide [207] apply many code optimizations on a high-level IR (an IR that has loops and represents multi-dimensional tensors). This is also one of the motivations for creating MLIR [208], which serves as a high-level IR for low-level compilers like LLVM.

2) Mathematical Abstractions of Code: While previous IRs have focused on representing program statements and program structure, many compilers use an additional mathematical representation (abstraction) to represent iteration domains[1] and array accesses of statements. These mathematical representations are usually used in conjunction with the IR to simplify iteration domain and array access transformations. This subsection presents two major families of mathematical representations and compares their strengths and weaknesses.

a) Polyhedral representation: It is a unified mathematical representation for the iteration domains of statements, code transformations, and dependencies. It relies on two main concepts: *integer sets* and *maps*. *Integer sets* represent iteration domains. *Maps* are used for representing memory accesses and transforming iteration domains and memory accesses.

An integer set is a set of integer tuples described using affine constraints. An example of a set of integer tuples is

$$\{(1,1);(2,1);(1,2);(2,2);(1,3);(2,3)\}.$$

Instead of listing all the tuples, we can describe the set by using affine constraints over loop iterators and symbolic constants as follows:

$$\{S(i,j) : 1 \leq i \leq 2 \wedge 1 \leq j \leq 3\},$$

where i and j are the dimensions of the tuples in the set.

A map is a relation between two integer sets. For example,

$$\{S1(i,j) \rightarrow S2(i+1, j+1) : 1 \leq i \leq 2 \wedge 1 \leq j \leq 3\}$$

is a map between tuples in the set S1 and tuples in the set S2. More details about the polyhedral model and formal definitions can be found in [209]–[211].

Polyhedral compilers: Notable polyhedral compilers for deep learning include Tiramisu [205], Tensor Comprehensions [212], Diesel [213], and TensorFlow XLA [214] (through affine MLIR dialect [208]). General-purpose compilers that support deep learning include PENCIL [211], Pluto [215], Polly [216], PolyMage [217], AlphaZ [218], and CHiLL [219].

Strengths of the polyhedral representation are as follows.

1) *Unified representation:* It eliminates friction within compiler IRs and greatly simplifies design of code transformations.
2) *Instancewise representation:* The representation granularity is instances of statement executions where each instance is a single execution of a statement during one loop iteration. Instancewise representation includes iteration domains, data dependencies, data accesses, and code transformations, which allows the compiler to have a precise representation.
3) *Support for the whole class of affine transformations:* It allows applying any affine transformation on iteration

[1]The iteration domain of loop iterators in a loop is all possible values that loop iterators can take.

domain and data accesses. An example of a complex affine transformation is iteration space skewing, which allows extracting parallelism from multilayer RNNs to increase hardware occupancy.

4) *Nonrectangular iteration domains:* The representation allows compilers to naturally express nonrectangular iteration domains (i.e., iteration domains with an affine conditional).

Weaknesses of the polyhedral representation are as follows.

1) *Limited support for nonaffine code:* The polyhedral model mainly represents code and transformations using sets and maps described using affine constraints. So, it does not naturally support code that leads to nonaffine constraints. This includes code with nonaffine loop bounds, nonaffine array accesses, and nonaffine conditional. While the classical polyhedral model does not support nonaffine constraints, recent work has extended the polyhedral representation to support nonaffine array accesses, nonaffine loop bounds, nonaffine conditionals [220], and parametric tiling [221]. The efficiency of these techniques has been demonstrated in practice by PENCIL [222] and Tiramisu [205].
2) *Slower compilation:* While polyhedral operations are precise, they are computationally expensive. So, polyhedral compilers are slower than nonpolyhedral compilers. Recent techniques reduce the number of statements by clustering groups of statements into macrostatements and scheduling macrostatements instead of individual statements [223], reducing the compilation time notably.

b) Nonpolyhedral representation: A common nonpolyhedral representation used in deep learning compilers is *interval*-based representation. It uses intervals and interval arithmetic to represent iteration domain and code transformations, respectively. Using intervals, N-dimensional loops are represented with N-dimensional boxes, e.g., iteration domain of a loop nest can be represented as $(i, j) \in$ ([0, N],[2, M-2]).

Nonpolyhedral DNN compilers: Their examples include TVM [206], Halide [207], DLVM [224], and Latte [225].

Strengths of interval-based representations are as follows.

1) *Better support for nonaffine code:* Nonpolyhedral compilers can naturally support nonaffine code transformations, such as parametric tiling (loop tiling with parametric tile size). This is because the interval-based representation does not rely on using affine sets and affine relations to represent the code or dependencies. However, nonpolyhedral compilers also have limited support for nonaffine code (e.g., indirect memory accesses) and code transformations.
2) *Faster compilation:* Operations on intervals are computationally less expensive than polyhedral equivalent operations on sets of integer points, which yields faster compilation.

Weaknesses of interval-based representations are as follows.

1) *Limited expressiveness:* Interval-based nonpolyhedral compilers cannot naturally represent nonrectangular iteration spaces (e.g., when bounds of loop iterators depend on a condition). It is also hard to perform certain complex affine transformations, such as iteration space skewing.
2) *Lack of support for programs with cyclic data-flow graphs:* To simplify checking the legality of a schedule, many interval-based compilers assume that the program has an acyclic dataflow graph. This prevents users from expressing many programs with cyclic dataflow. For example, when a value produced by a loop is read by another loop, Halide [207] does not allow fusion of the two loops (with `compute_with` command). While it avoids illegal fusion, it prevents legal loop fusions in common cases. Polyhedral compilers avoid these overconservative constraints by using dependence analysis [226] to check for the correctness of code transformations, which enables more schedules. While interval-based compilers can also implement nonpolyhedral dependence analysis (by computing dependence distance vectors [227]), it is not as precise as polyhedral dependence analysis [226].

B. Support for Sparse Tensors

1) Challenges in Supporting Sparse Tensors: While compiler support is needed in general for targeting ML hardware accelerators with diverse features, sparse tensor computations with various dataflows especially need further support. The code for manipulating sparse tensors exhibits *nonstatic loop bounds, nonstatic array accesses, and conditionals*, which are difficult to analyze at compile time. The following pseudocode shows one example of a direct convolution with sparse tensors (bounds of j and accesses of in are nonstatic).

```
for each output channel c_o
  for j in (W.row_ptr[c_o], W.row_ptr[c_o
  + 1])
  {
    coeff = W.value[j]
    offset = W.col_idx[j]
    for y in (0, out_H)
      for x in (0, out_W)
        out[c_o][y][x] += coeff*in[y*out_W
        +x+offset]
  }
```

2) DNN Compilers Supporting Sparsity: Their examples include Tiramisu [205], Acorns [81], and Taichi [228].

Tiramisu supports W-sparsity by extending the polyhedral model in a way similar to [220]. For example,

a nonaffine conditional is transformed into a predicate that is attached to computation. The list of accesses of the computation is the union of the accesses of the computation in the two branches of the conditional, which is an overapproximation. During code generation, a preprocessing step inserts the conditional back into generated code. Nonstatic loop bounds and tensor accesses are represented as parameters in the polyhedral model. Statements that define those parameters are inserted just before the original statements that have nonstatic code. These techniques introduce approximations in the compiler. Their efficiency was demonstrated by [220] and confirmed by PENCIL [222] and Tiramisu [205].

Acorns [81] optimizes the CNNs with *IA*-sparsity. It fuses operators in a computation graph of a deep CNN, followed by sparse layout conversion (which ensures that dense/sparse tensors produced by each operator are compatible with the next operation), followed by code optimization and code generation. Acorns introduces a data layout for exploiting the sparsity structure of input data in certain domains (face detection, LiDAR, and so on) where only certain data regions are NZs. For code optimization and generation, the compiler processes a set of template codes for CNN operators (e.g., convolution and pooling) and applies optimizations, such as loop tiling, vectorization, and weight packing. It does not implement advanced loop-nest optimizations, such as iteration space skewing.

TACO [229] uses a specific representation (iteration graphs) to generate code for sparse tensor operations and uses a scheduling language to guide the code optimization.

C. Compiler Optimizations

To generate efficient code for NN operators, a compiler has to apply a large set of complex code optimizations. It includes operator fusion; multilevel tiling, and register blocking that improve data reuse; loop reordering, array packing [230], and data prefetching; loop skewing that enables the extraction of wavefront parallelism from multilayer RNNs; parallelization; loop unrolling; vectorization; full/partial tile separation; and tuning optimization parameters to the target architecture (e.g., tile sizes or loop unrolling factors). There are two major families of optimizing compilers: compilers that allow semiautomatic code optimization and fully automatic compilers.

1) Compilers With Semiautomatic Code Optimization (Scheduling Languages): The main idea in these compilers is to separate the algorithm from optimizations. A program, in this case, has two parts. The first part specifies the algorithm without specifying how it is optimized. The second part specifies how the algorithm is optimized (transformed). This is done through a set of high-level scheduling commands for common optimizations. Halide [207], Tiramisu [205], and TVM [206] are examples of compilers that allow semiautomatic optimization. The main advantage of this approach is it allows a user to have full control over how code should be optimized. This is important because fully automatic optimization techniques do not always succeed in providing the best performance.

Semiautomatic deep learning compilers usually provide a library of highly optimized deep learning operators. The compiler then only needs to decide automatically whether to apply certain optimizations, such as operator fusion. All other optimizations are encoded manually in the library using scheduling commands. This minimizes the number of decisions that the compiler needs to make and, thus, guarantees the best possible performance. Note that semiautomatic compilers usually also have automatic optimization modules, but such modules can be disabled if necessary.

2) Fully Automatic Compilers: Tensor Comprehensions [212] and Diesel [213] are examples of fully automatic compilers for deep learning. Other examples of fully automatic compilers include PENCIL [211], [222], Pluto [215], and Polly [216]. All of them use Pluto [215] algorithm to automatically optimize code (choosing the schedule of statements). The main idea of the Pluto algorithm is to use integer linear programming to model the problem of automatic code optimization where constraints are dependencies of the program, and the objective function is the minimization of the distance between producer and consumer statements. Other compilers, such as PolyMage [217], use a custom algorithm for automatic optimization.

All these compilers do not have a scheduling language and, therefore, do not allow the user to have fine-grain control over optimizations. Although fully automatic compilers provide productivity, they may not always obtain the best performance. Performance can be suboptimal because they do not have a precise cost model to decide which optimizations are profitable. For instance, the Pluto [215] algorithm does not consider the redundant computations, data layout, or the complexity of the control flow of generated code.

Cost models for automatic code optimization: The goal of an automatic code optimization pass in a compiler is to find the best combination of code optimizations that minimize the execution time. This can be modeled as a search problem where the search space is a set of combinations of code optimizations. Then, the compiler needs a search technique and a cost model to evaluate each combination. Classical compilers use hand-tuned cost models [231], while others use ML to build cost models [232]. Both of these models do not precisely capture hardware complexity (different memory hierarchies, out-of-order execution, hardware prefetching, communication latency, and so on). Instead, state-of-the-art models are built using deep learning for better accuracy [233], [234]. For example, Ithemal [234] is a cost model that predicts the throughput of a basic block of x86 instructions and gets less than half the error of state-of-the-art hand-tuned models (llvm-mca in LLVM [235] and Intel's IACA).

D. Accelerator ISAs and Code Generation

Accelerators, such as Cambricon-X [41], Scaledeep [236], Thinker [128], and DnnWeaver [184], expose a high-level ISA where some instructions perform tensor operations (e.g., dot product, matrix–matrix multiplication, convolution, pooling, and sigmoid). They simplify the compiler's mission because it can invoke high-level operations instead of generating and optimizing a low-level code. However, it still has to manage data copies automatically. This section describes such high-level ISAs used by accelerators and machine code generation.

1) Instruction Sets: For tensor computations on hardware accelerators, ISAs typically feature instructions for arithmetic, logical, and data transfer operations with matrix, vector, and scalar data. Layers of ML models feature loops iterating thousands of times; dynamic instances of repetitive instructions can significantly increase the bandwidth requirements for delivering them to PEs at each cycle and the energy consumption. To mitigate such overheads, accelerators are designed with an array of vector or SIMD PEs. It allows PEs to process a single instruction for performing multiple computations on the blocks of tensors. Alternatively, PEs contain additional control logic such that they process an instruction once but repeatedly perform the sequence of execution for a certain interval.

Cambricon ISA for ML [237] contains instructions for matrix and vector processing with arithmetic and logic operations, control (conditional branch and jump), and data transfer. Each operand of the instruction is either an immediate value or one of the 64 32-b general-purpose registers. The registers are used for temporarily storing scalars or register-indirect addressing of the on-chip scratchpad memory. The tensor blocks are communicated between computational units from the on-chip scratchpad that is transparent to the compiler and programmers. The instructions support commonly used primitives in various ML models, e.g., multiplication, addition, subtraction, and division operations on matrices and vectors. It also supports max-pooling with a vector-greater-than-merge instruction and provides dedicated instruction for random vector generation with uniform distribution of values within [0, 1]. For supporting weight update during the training of DNNs, Cambricon provides additional instructions such as outer product, scalar–matrix multiplication, and matrix–matrix addition. However, it lacks support for managing data in the local memory of PEs and configuring NoC for communication in various dataflows. Moreover, it does not provide specific instructions for handling sparsity, e.g., predicated execution of encoded sparse data.

The instruction set for Sticker [164] consists of instructions for high-level operations. For processing each layer, one of the instructions is executed only once. It configures instruction registers and common control signals that correspond to the sparsity levels and tensor dimensions. Then, at a certain time interval, a dynamic 32-b instruction executes for computing convolution over data blocks on PE-array. Meanwhile, the accelerator controller distributes the next instruction if there is no collision between the current and the next instruction. It allows hiding the execution of other dynamic instructions including the write-back and encoding of outputs and transferring data between on-chip and off-chip memories.

2) FSMs: Some accelerators use FSMs for PE executions. The parameters of FSMs or PE's control logic correspond to tensor shapes and target functionality, and they are configured once (e.g., through bit-streams [20]) before executing a model or a layer. Accelerator controllers (which usually initiate the data movement between on-chip and off-chip memories and configure PEs and NoC) can also contain FSMs. For example, in Thinker architecture [128], a finite-state controller is used for configuring the accelerator at three levels, i.e., the PE-array level, the model layer level, and the PE level. Configuration word for PE-array level handles partitioning of the PE-array, and it points to the memory address of configurations for model layers. Each configuration word for a layer contains information about tensor dimensions and their memory addresses. Finally, layer configurations for PEs correspond to PE functionality and the interval (loop iterations) of computations and idle time.

3) Library Support and Code Generation: The instructions for cycle-level executions or primitives are usually obtained offline. Accelerator system designers often provide users a template library that defines high-level primitives, such as model layers, or low-level primitives, such as vector/matrix operations. Using these primitives, users can construct the model of their interest. Then, the low-level code is obtained automatically by the compiler or using the predefined optimized code [236], [238]. For example, Zhang *et al.* [41] programed Cambricon-X accelerator with a set of library functions (written in C/C++) for primitives, such as convolution and matrix/vector multiplication and addition. Chen *et al.* [237] proposed a programming framework consisting of assembly language, an assembler, and runtime support for executing ML models with their Cambricon ISA. For executing common layers, it also replaced the primitives with predefined code blocks.

TVM [206] supports defining custom back ends for accelerators, which was demonstrated using a vanilla accelerator with a matrix–multiply engine. For executing primitives on accelerators, TVM enables Tensorization [206], i.e., decoupling the target hardware intrinsic from the schedule while mapping ML operators. To demonstrate code generation for the vanilla accelerator, TVM enabled a driver library and runtime support that constructs the instructions and offloads them to the accelerator. Its code generation module translated the program into appropriate function calls of the runtime API. Moreau *et al.* [239] leveraged the TVM stack and proposed a JIT compiler and a runtime system to generate code for a programmable VTA accelerator.

It is important that the accelerator can support multiple front ends corresponding to different ML frameworks, such as TensorFlow [49], PyTorch [48], and MXNet [240]. Integration of the programming, compilation, and runtime environment with the common frameworks for ML application development is necessary for supporting different compact ML models. Leveraging the existing system stack (e.g., TVM) can provide such opportunities to accelerator system developers. Note that, although TVM supports defining custom accelerator back ends and can lower optimized mappings to accelerator-specific code, it currently does not provide support for sparse tensors.

XIII. TRENDS AND FUTURE DIRECTIONS

A. Hardware/Software/Model Codesigns

1) Hardware-Aware Compression Techniques: The framework for exploring efficient model compression (either of quantization, pruning, and size reduction) should be aware of hardware features and provide directed search accordingly. For example, bit-widths of tensors that can be efficiently processed by different hardware platforms vary considerably (e.g., from multiples of 8-bits to arbitrary bit-widths). Accelerators typically support only uniform widths of tensors (activations and weights), and many accelerators do not support value sharing. Also, when hardware only supports widths that are multiple of 4 or 8 bits, quantization with other bit-widths requires zero paddings, which incurs inefficient storage and processing. Instead, the compression algorithm can opt for improving the accuracy, increasing sparsity, or trading off the bit-widths among layers for achieving higher compression and acceleration. Similarly, depending on the hardware support for fine-grained or block-sparsity, hardware-aware pruning can better achieve the compression objectives (model exploration time, performance, and energy efficiency while meeting target accuracy). Efficiency can be further improved when compression techniques leverage execution models of hardware accelerators (e.g., energy-aware pruning [40]). Relatively simple logic modules of hardware accelerators have enabled recent techniques to estimate execution metrics through analytical cost models. Accommodating such cost models (including for different sparsity levels/patterns and precisions) enables the compression algorithms to select effective pruning ratios/structures, tensor shapes, and tensor precisions, which can help to achieve desired accelerations.

2) Joint and Automated Exploration of Sparsity, Precision, and Value Similarity: Recent compression techniques typically employ structured or fine-grained data pruning during training with a fixed precision of tensors. Techniques for adaptive quantization often do not explore pruning. Joint explorations of pruning and quantization may achieve high compression due to the interplay of these compression mechanisms. For instance, quantization can increase sparsity considerably [121], as more values can be represented as zero after compressing the range [31]. Likewise, pruning may reduce bit-widths further since fewer NZ values in the pruned model may be expressed with a much lower numeric range and precision. Moreover, such compression techniques do not leverage temporal and spatial value similarity in inputs, outputs, or weights. So, joint exploration algorithms may be developed that use multiple compression strategies during training and automatically explore combinations that compress the model further. Recent techniques for automated explorations include CLIP-Q [58], [241], [242]. Exploring a wide range of compression combinations during the training may not be feasible. Therefore, model designers may reduce the space of compression choices by limiting effective options before beginning resource-extensive training and, if required, further limiting the search space by quick evaluations with a pretrained model and fine-tuning.

Compression benefits achieved through joint explorations need to be translated into efficient hardware accelerations. So, the exploration heuristic should not preclude experts from expressing a directed search for hardware-friendly executions, e.g., specifying pruning with 1-D or k:n block-sparsity, constraints for bit-widths, and tolerable accuracy loss. Moreover, the heuristic should also provide automated optimization/exploration of hyperparameters (including using cost models of accelerators). This is because the compression algorithm needs to adjust the strategy of pruning or quantization and its hyperparameters. For instance, the pruning algorithm needs to find out the pruning ratio for each iteration (epoch); pruning mechanism (which values to prune, e.g., below a certain threshold); pruning pattern (fine-grain, block size); and bit-widths of tensors (quantization). All such hyperparameters or strategies need to be adjusted automatically (to an extent) such that the memory footprint or computations are greatly reduced, with no or tolerable accuracy loss.

3) Value-Aware Neural Architecture Search (NAS) and Accelerator/Model Codesigns: Techniques for NAS or AutoML can automatically obtain efficient models that surpass the accuracy of models devised by human developers. However, there remains scope for considerably improving NAS for obtaining highly compact models. Recent techniques [243]–[246] have explored accelerator/model codesigns that support quantized models and layers of different shapes. However, the efficiency can be further amplified by including the sparsity and adaptive bit-widths of model layers and analytically considering their implications on hardware accelerators.

A major challenge faced by the model search techniques and automated accelerator/model codesigns is the vast search space. As shown in Fig. 31, explorations can be performed for: 1) ML models (i.e., NAS) [31]; 2) compression strategies (e.g., automated pruning and quantization) [247]; 3) mappings of models on accelerators [179], [186]; and 4) specifications of hardware accelerators [57], [179]. The explorations of 1) and 2) directly impact

compression and accuracy, while search optimizations for 3) and 4) affect the performance and energy efficiency of the accelerator for given models. Among these exploration spaces, NAS can be significantly time-consuming (several GPU days [31]), followed by automated model compression (e.g., [247]). Therefore, the resultant joint space for value-aware NAS and accelerator/model codesigns is many-folded. So, it may require notable efforts for developing automated exploration of codesigns that can obtain extremely efficient and accelerator-friendly compact models.

4) Facilitating Structured Computations of Sparse Tensors: Designers may opt for accelerators that are effective for structured computations of dense tensors, e.g., systolic arrays (as near-data accelerators or coupled to processor cores) and in-memory processing with resistive crossbars. While sparsity or size reduction of tensors may need to be leveraged, significant design modifications are often infeasible due to design requirements (area/power budgets) or the increase in complexity of the system stack. So, techniques for preprocessing can be developed, which can arrange structured dense regions for feeding underlying engines or achieve structured data through sparsification/reorganization at almost no accuracy loss. Such preprocessing can be done on additional hardware modules or the host processor that handles the nonperformance-critical tasks. Such disjoint mechanisms can obviate heavy design modifications in systolic arrays (e.g., [127]) or in-memory/near-data processing engines (e.g., ReCom [248] and SNNrram [249]) while leveraging various sparsities and value similarity opportunities across different models.

B. Design Tools and Frameworks

1) Framework for Analyzing Performance Gains of Accelerators Due to Sparsity: Given that tensors of several ML models are sparse, it is important to design accelerator systems that can exploit performance gains for multiple models through low-cost hardware modules and enhanced software support. As we discussed in Sections V–XII, each enhancement presents multiple implementation choices at the hardware or software level. Although crafting a cycle-level simulation infrastructure for such a wide design space may be infeasible, a data-driven quantitative model can be significantly helpful for design explorations. It can process the actual data (or discover distributions of zeros), provide high-level modeling of common choices, and estimate the performance gains for each combination of the implementation choices. For newer models or functionality, hardware designers can run through a set of implementation choices in an early design phase. They can explore the implications of sparsity for the desired choice of encoding, data extraction logic, functional units, NoC, load balancing, and dataflow mechanism.

2) Accelerator Design Frameworks for Compact Models: Several frameworks for developing and simulating

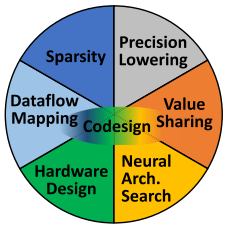


Fig. 31. ***Codesigns can enable efficient accelerations of compact models.***

FPGA- or ASIC-based accelerators have recently been proposed, including DNNWeaver [184], DNNBuilder [250], T2S-Tensor [251], and HeteroCL [252] for FPGAs and NVDLA [120], VTA [239], MAGNet [253], MAERI [177], and AutoDNNChip [176] for specialized accelerators. Similarly, hardware construction languages or representations, such as Chisel [254] and μIR [255], enable expressing microarchitectural features through high-level primitives. Such infrastructures are key for the community since they can serve as a good learning resource for training the new professionals and provide a kick-starter baseline for developing new design features.

However, most frameworks support designs for dense tensors of fixed bit-widths and lack support for sparsity-tailoring features. Such frameworks can provide some prebuilt modules for encoding/extracting sparse data (e.g., with common formats, such as RLC, bitmap, or for block-sparsity), dynamic load balancing or data reorganization, configurable functional units, configurable interconnects for sparse and bit-adaptive computing, and so on. Even with limited features, they may serve as reusable logic that can be leveraged by designers for quick prototyping and design explorations. Furthermore, abstractions and specifications for constructing sparsity-tailored hardware and dataflows can enable automated and efficient design explorations and easier programming of accelerators.

C. Accelerating Training of ML Models

While there have been significant advances in performing inference on hardware accelerators, efficient training of the models on hardware accelerators has received relatively little attention. Training has been done in high-performance computing environments containing CPU and GPU platforms and recently on FPGAs and TPU accelerators. Hardware accelerators can offer significant benefits to the model training in both edge and datacenter-scale computing environments, and they can notably improve performance and energy efficiency, respectively. In particular, they are promising for enabling online learning on edge devices through compact models.

Accelerators, such as [21], ScaleDeep [236], and HyPar [187], have been proposed for efficiently training the models. However, they either do not leverage sparsity

or may not be efficiently utilized for irregular shaped tensors or lack support for various precisions of weights, activations, gradients, and weight updates. This presents further opportunities for performance gains and energy efficiency. In addition, designers can leverage cross-layer optimizations (e.g., by reusing the data of gradients during backpropagation) and support mixed-precision of tensors during the training of compact models.

D. Applying Techniques for Sparsity to Other Domains

In this work, we considered a wide variety of techniques that leverage sparsity for the ML domain, which represents an enormous research effort. Many other domains face similar challenges in exploiting sparsity, and accelerators have been proposed for some of the more processing-intensive domains; this includes graph processing [256], [257], database operations [258], genomics [259], [260], and compression [261]. In some cases, computation primitives even extend across domains. For example, finding intersecting NZs is analogous to joins in a database context [110]. Applying the lessons learned from extensive research on sparsity in an ML context can likely speed innovation in a broader context.

XIV. RELATED WORK

A. Deep Learning Models and Their Applications

Surveys [45], [46] described different deep learning models along with different frameworks and datasets. Gu *et al.* [262] discussed applications of CNNs in CV and language processing. Recent surveys have also discussed applications of deep learning in medical image analysis [13], biomedical applications [263], wireless and networking [16], and embedded systems [15]. Elsken *et al.* [264] surveyed techniques for NAS.

B. Compact Models

Cheng *et al.* [265] surveyed techniques for parameter pruning and low-rank factorization. Wang *et al.* [266] surveyed techniques for pruning, precision lowering, weight sharing, low-rank factorization, and knowledge distillation. Deng *et al.* [31] described techniques to obtain compact models, including sparsification, quantization, tensor decomposition, and joint-way compression.

C. Hardware Accelerators for Dense ML Models

Shawahna *et al.* [267] surveyed FPGA accelerators for processing dense tensor computations of deep learning applications. Venieris *et al.* [268] discussed different CNN-to-FPGA toolchains and described their hardware architectures, design space exploration techniques, and support for different precisions of tensors. They also compared execution metrics of the designs obtained with various toolchains and those with the previously proposed FPGA accelerators for CNNs. Sze *et al.* [44] presented a survey about efficiently executing DNNs on hardware accelerators. It described different DNNs, different compression techniques for compact models, and optimized dataflows for spatial architectures. Reuther *et al.* [269] benchmarked executions of different ML accelerators. Li *et al.* [270] discussed different ML frameworks and compilers for deep learning models.

D. Hardware Accelerators for Compact ML Models

Mittal [271] surveyed executing compact models, including BNNs, on FPGAs. It also discussed processing convolutions with the Winograd algorithm and executions on multiple FPGAs. Deng *et al.* [31] surveyed hardware accelerators that support bit-adaptive computing and the data extraction modules for leveraging the sparsity of inputs, weights, or outputs. Du *et al.* [272] recently proposed MinMaxNN system for dynamically switching NN models. They surveyed techniques for designing self-aware NN systems (which can continuously sense information from the environment and dynamically react), including leveraging sparsity and tensor quantization. Wang *et al.* [266] surveyed hardware implementations for processing tensors of lower precisions (binary, ternary, and logarithmic quantizations). Ignatov *et al.* [273] benchmarked executions of quantized deep learning models on mobile AI accelerators.

In contrast to the above surveys, this work highlights sources of sparsity and size reduction of tensors in ML models and challenges in efficiently executing them on hardware accelerators. Then, it surveys and discusses the corresponding hardware and software support, including encodings and extraction of sparse data, sparsity-aware dataflows, memory management, and on-chip communication of sparse tensors while leveraging data reuse, load balancing of computations, and compiler support. It also discusses techniques for computations of mixed-precision and value-shared sparse tensors.

XV. SUMMARY

For efficient and hardware-friendly processing, compact deep learning models have been designed. They consume less storage and computations and consist of tensors with considerable sparsity, asymmetric shapes, and variable precisions. While these compact models can be accelerated on hardware accelerators efficiently, it requires special hardware and software support. We have highlighted challenges in efficiently accelerating their sparse and irregular tensor computations. Leveraging sparsity, especially unstructured, requires a significant redesign to store, extract, communicate, compute, and load-balance only NZs. Moreover, the sparsity levels and patterns due to various sources lead to unique challenges and solutions in hardware/software/model codesigns.

In this article, we have discussed how exploiting sparsity effectively depends on tailoring the data encoding

and extraction, dataflow, memory bank structure, interconnect design, and write-back mechanisms. We provided an overview of corresponding enhancements in accelerator designs and their effectiveness in exploiting sparsity. Categorization of different techniques informs how they leveraged structured or unstructured sparsity of weight or activations during learning or inference of ML models (see Tables 1 and 2). For recent DNNs, we analyzed achievable accelerations for a few popular accelerators (see Section IV-B). The analysis showed that accelerators exploit moderate sparsity and achieve high speedups as sparsity increases. However, exploiting high sparsity or hypersparsity can further provide considerable opportunities, which would also need efficient mechanisms for data extraction and load balancing. Also, configurable architectures for NoCs, functional units, and buffers are required for catering to various functionalities and metadata management.

Our analysis of sparsity-encodings describes their storage efficiency for various sparsities and the decoding requirements. While bitmaps and RLC/CSC formats are commonly used for moderate and high sparsity, respectively, storage efficiency can be improved with block-sparse tensors (especially at hypersparsity). We have introduced a taxonomy for NZ extraction techniques that are used for feeding the functional units of PEs. Existing data extraction mechanisms (e.g., in EIE [42], EyerissV2 [43], and Cambricon-X/S [37], [41]) exploit moderate sparsity. However, they may not extract enough NZs at high sparsity or hypersparsity of large tensors (e.g., sparse BERT [71]), achieving lower speedups. We also discuss how block-sparsity can simplify data extraction and facilitate balanced computations. For exploiting diverse sparsity across tensors of different models, designers can explore multiple or configurable mechanisms for decoding and extraction of NZs.

Data reuse opportunities in processing common DNNs vary significantly and sparsity lower the reuse due to fewer effectual computations. However, compressed tensors allow fitting and reusing more data in on-chip memory, which reduces access to off-chip memory and overall latency. We have discussed techniques for memory bank management to support unstructured accesses for sparse computations and hiding the memory access latency. At high sparsity or hypersparsity, execution may become *bandwidth-bounded*, as enough data may not be prefetched always. Hence, techniques for efficient data management (e.g., cross-layer and on-chip data reuse) and exploiting high bandwidths need to be explored. Different accelerator designs have used various interconnects for the distribution of operands, reduction of partial outputs, and collecting the outputs. They vary in terms of the bandwidth requirement and exploiting spatial data reuse. *Configurable interconnects* (e.g., in EyerissV2 [43] and SIGMA [105]) are required for accelerating different DNNs of diverse sparsity, functionality, and tensor shapes since they can support a mix of communication patterns. They are important for enabling *asymmetric* spatial accumulations of partial outputs (for sparse tensor computations) and concurrent spatial processing of different groups, e.g., for DW-CONV.

Processing compressed tensors can impose significant maneuvering efforts in the PE architecture design. We discuss further opportunities including configurable designs of functional units for efficient vector processing and flexible sparsity-aware dataflows for high utilization across variations in sparsity and functionality of different layers. We also surveyed techniques for approximated computing through multiplier-free PEs and leveraging temporal and spatial similarity of values, which improves execution efficiency further. Sparse tensor computations over different PEs can be highly imbalanced. We have surveyed different techniques that sustain the acceleration by balancing the work through hardware modules for asynchronous computations or work sharing (e.g., EIE [42] and ZENA [130]). Software-directed regularization, such as structured sparsity, eliminates load imbalance, e.g., in leveraging weight/activation sparsity for Cambricon-S [37] and 50% weight sparsity for NVIDIA A100 [61]. Techniques including data transformations and refactoring of DNN operators may achieve low-cost load balance, including for dynamic sparsity. We have also surveyed mechanisms for asynchronous write-backs of outputs and sparsity-aware encodings on the fly. Compilation for the accelerators requires the ability to efficiently express sparsity in IRs, flexibly apply different compiler optimizations, and emit efficient accelerator-specific code. The survey has discussed techniques that can enable such support and open challenges.

Accelerator/model codesigns can efficiently leverage various precision and value similarities of different tensors and induce sparsity for accelerator-friendly executions. Automated and joint explorations of accelerator-aware compression algorithms can advance acceleration opportunities further. We have highlighted future directions for such codesigns and the system stack development (see Section XIII). In individual sections, we have also discussed further opportunities for tailoring different hardware or software enhancements for sparsity. While our discussions focused on leveraging sparsity for ML models, exploiting diverse sparsity can also aid the efficient processing of applications of other domains [92], [93].

In conclusion, while different accelerators and compression algorithms have been proposed for efficiently processing compact ML models, it remains an active research frontier. In particular, hardware/software/model codesigns and automated and joint explorations of tensor sparsity, adaptive quantization, shape reductions, and dataflow will likely provide further opportunities for innovations across the system. With a boost in energy-efficient accelerations of the learning and inference at the cloud and edge, they can be anticipated to further improve the intelligence of various systems or applications.

APPENDIX
HARDWARE ACCELERATORS CAN EXPLOIT SPARSITY BETTER

Exploiting acceleration opportunities due to sparsity (especially unstructured) is relatively hard for execution on CPUs and GPUs [37], [41], [105]. The performance of ML models can even degrade compared to the execution with dense data (e.g., for a GEMM, when unstructured W-sparsity is below 70% [274]). For executing AlexNet layers on GPUs, Wen *et al.* [28] analyzed speedup for processing CSR-encoded matrices with cuSPARSE and dense matrices with cuBLAS. Their experiments showed obtaining *limited* speedups (below 1.4×) or even slowdowns for high sparsity. This is because unstructured sparsity may yield poor data locality for scattered effectual NZs. Plus, it is challenging to skip ineffectual computations and equally distribute the work among multiple threads or computational units of processor cores. Zhang *et al.* [41] analyzed performance benefits of executing sparse models (LeNet, AlexNet, and VGG-16) on CPU (with sparse BLAS) and GPU (with cuSPARSE) platforms compared to processing dense models (with Caffe [204]). For average sparsity of 90.94%, they reported geomean speedup of only 23.34% for GPU and 110% more time on CPU. In their sparsity–sensitivity analysis, CPU and GPU showed marginal speedup only at moderate or high sparsity due to nontrivial costs of sparse data processing. However, for Cambricon-X [41], performance gains were reported for 5% or more sparsity due to its design tailored for sparse tensor computations. For hypersparsity, it achieved high speedups (e.g., 15.5× for CONV and 48.5× for FC layer) compared to executing dense tensors [41]. So, with special support for sparse and irregular tensor computations, hardware accelerators can achieve notable speedups and efficiency. ■

Acknowledgment

The authors thank anonymous reviewers for providing valuable feedback.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
[3] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 6105–6114.
[4] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018, *arXiv:1804.02767*. [Online]. Available: http://arxiv.org/abs/1804.02767
[5] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," 2017, *arXiv:1706.05587*. [Online]. Available: http://arxiv.org/abs/1706.05587
[6] 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)*, Dec. 2015, pp. 4489–4497.
[7] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, vol. 1, 2019, pp. 4171–4186.
[9] T. B. Brown *et al.*, "Language models are few-shot learners," 2020, *arXiv:2005.14165*. [Online]. Available: http://arxiv.org/abs/2005.14165
[10] I. Goodfellow *et al.*, "Generative adversarial nets," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 1–9.
[11] J. Park *et al.*, "Deep learning inference in Facebook data centers: Characterization, performance optimizations and hardware implications," 2018, *arXiv:1811.09886*. [Online]. Available: http://arxiv.org/abs/1811.09886
[12] M. Naumov *et al.*, "Deep learning recommendation model for personalization and recommendation systems," 2019, *arXiv:1906.00091*. [Online]. Available: http://arxiv.org/abs/1906.00091
[13] G. Litjens *et al.*, "A survey on deep learning in medical image analysis," *Med. Image Anal.*, vol. 42, pp. 60–88, Dec. 2017.
[14] T. Kurth *et al.*, "Exascale deep learning for climate analytics," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2018, pp. 649–660.
[15] N. M. Rezk, M. Purnaprajna, T. Nordström, and Z. Ul-Abdin, "Recurrent neural networks: An embedded computing perspective," *IEEE Access*, vol. 8, pp. 57967–57996, 2020.
[16] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 3, pp. 2224–2287, 3rd Quart., 2019.
[17] C. R. Banbury *et al.*, "Benchmarking TinyML systems: Challenges and direction," 2020, *arXiv:2003.04821*. [Online]. Available: http://arxiv.org/abs/2003.04821
[18] J. Dean, "1.1 The deep learning revolution and its implications for computer architecture and chip design," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2020, pp. 8–14.
[19] K. Olukotun, "Designing computer systems for software 2.0," presented at the Conf. Neural Inf. Process. Syst., 2018.
[20] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.
[21] B. Fleischer *et al.*, "A scalable multi-TeraOPS deep learning processor core for AI trainina and inference," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2018, pp. 35–36.
[22] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 1–12.
[23] X. Yang *et al.*, "DNN dataflow choice is overrated," 2018, *arXiv:1809.04070*. [Online]. Available: https://arxiv.org/abs/1809.04070
[24] D. Amodei, D. Hernandez, G. Sastry, J. Clark, G. Brockman, and I. Sutskever. (May 2018). *AI and Compute*. [Online]. Available: https://openai.com/blog/ai-and-compute/
[25] 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.*, 2015, pp. 1135–1143.
[26] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.
[27] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*. [Online]. Available: http://arxiv.org/abs/1510.00149
[28] 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.*, 2016, pp. 2074–2082.
[29] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *Proc. Int. Conf. Learn. Represent.*, 2018, pp. 1–42.
[30] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr, "Fine-grained accelerators for sparse machine learning workloads," in *Proc. 22nd Asia South Pacific Design Automat. Conf. (ASP-DAC)*, Jan. 2017, pp. 635–640.
[31] B. L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.
[32] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4510–4520.
[33] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size," 2016, *arXiv:1602.07360*. [Online]. Available: http://arxiv.org/abs/1602.07360
[34] A. Cichocki *et al.*, "Tensor decompositions for signal processing applications: From two-way to multiway component analysis," *IEEE Signal Process. Mag.*, vol. 32, no. 2, pp. 145–163, Mar. 2015.
[35] L. Moroney. (2018). *Introducing Ragged Tensors*. [Online]. Available: https://blog.tensorflow.org/2018/12/introducing-ragged-tensors.html
[36] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018, *arXiv:1806.08342*. [Online]. Available: http://arxiv.org/abs/1806.08342
[37] X. Zhou *et al.*, "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 15–28.

[38] 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. Archit. Support Program. Lang. Oper. Syst.*, Apr. 2019, pp. 925–938.

[39] S. Narang, E. Elsen, G. Diamos, and S. Sengupta, "Exploring sparsity in recurrent neural networks," 2017, *arXiv:1704.05119*. [Online]. Available: http://arxiv.org/abs/1704.05119

[40] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 5687–5695.

[41] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, p. 20.

[42] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 243–254.

[43] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019.

[44] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[45] S. Pouyanfar *et al.*, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 1–36, Sep. 2018.

[46] M. Z. Alom *et al.*, "A state-of-the-art survey on deep learning theory and architectures," *Electronics*, vol. 8, no. 3, p. 292, 2019.

[47] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," 2017, *arXiv:1709.05584*. [Online]. Available: http://arxiv.org/abs/1709.05584

[48] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035.

[49] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2016, pp. 265–283.

[50] E. Wang *et al.*, "Intel math kernel library," in *High-Performance Computing on the Intel Xeon Phi*. Cham, Switzerland: Springer, 2014, pp. 167–188.

[51] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2014, pp. 609–622.

[52] K. Khan, "Xilinx DNN processor (xDNN), accelerating AI in datacenters," Xilinx, Frankfurt, Germany, Tech. Rep., 2018. [Online]. Available: https://www.xilinx.com/publications/events/developer-forum/2018-frankfurt/accelerating-ai-in-datacenters-xilinx-ml-suite.pdf

[53] J. Fowers *et al.*, "A configurable cloud-scale DNN processor for real-time AI," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 1–14.

[54] N. Suda *et al.*, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2016, pp. 16–25.

[55] K. E. Fleming, K. D. Glossop, and S. C. Steely, "Apparatus, methods, and systems with a configurable spatial accelerator," U.S. Patent 10 445 250, Oct. 15, 2019.

[56] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, "DMazeRunner: Executing perfectly nested loops on dataflow accelerators," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5s, pp. 1–27, Oct. 2019.

[57] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 754–768.

[58] G. Srivastava, D. Kadetotad, S. Yin, V. Berisha, C. Chakrabarti, and J.-S. Seo, "Joint optimization of quantization and structured sparsity for compressed deep neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2019, pp. 1393–1397.

[59] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 548–560, 2017.

[60] H.-J. Kang, "Accelerator-aware pruning for convolutional neural networks," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 30, no. 7, pp. 2093–2103, Jul. 2020.

[61] R. Krashinsky, O. Giroux, S. Jones, N. Stam, and S. Ramaswamy. (2020). *NVIDIA Ampere Architecture In-Depth*. [Online]. Available: https://devblogs.nvidia.com/nvidia-ampere-architecture-in-depth/

[62] Z.-G. Liu, P. N. Whatmough, and M. Mattina, "Systolic tensor array: An efficient structured-sparse GEMM accelerator for mobile CNN inference," *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 34–37, Jan. 2020.

[63] D. T. Vooturi, D. Mudigere, and S. Avancha, "Hierarchical block sparse neural networks," 2018, *arXiv:1808.03420*. [Online]. Available: http://arxiv.org/abs/1808.03420

[64] S. Cao *et al.*, "SeerNet: Predicting convolutional neural network feature-map sparsity through low-bit quantization," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 11216–11225.

[65] J. Li *et al.*, "SqueezeFlow: A sparse CNN accelerator exploiting concise convolution rules," *IEEE Trans. Comput.*, vol. 68, no. 11, pp. 1663–1677, Nov. 2019.

[66] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-J. Yoo, "7.7 LNPU: A 25.3 TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8-FP16," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2019, pp. 142–144.

[67] E. Elsen, M. Dukhan, T. Gale, and K. Simonyan, "Fast sparse ConvNets," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 14629–14638.

[68] S. Han *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2017, pp. 75–84.

[69] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," 2017, *arXiv:1710.01878*. [Online]. Available: http://arxiv.org/abs/1710.01878

[70] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," 2019, *arXiv:1902.09574*. [Online]. Available: http://arxiv.org/abs/1902.09574

[71] T. Wolf *et al.*, "Transformers: State-of-the-art natural language processing," in *Proc. Conf. Empirical Methods Natural Lang. Process., Syst. Demonstrations*. Stroudsburg, PA, USA: Association for Computational Linguistics, Oct. 2020, pp. 38–45.

[72] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 1–13, 2016.

[73] Q. Yang, J. Mao, Z. Wang, and H. Li, "DASNet: Dynamic activation sparsity for neural network efficiency improvement," 2019, *arXiv:1909.06964*. [Online]. Available: http://arxiv.org/abs/1909.06964

[74] B. Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 267–278.

[75] G. Georgiadis, "Accelerating convolutional neural networks via activation map compression," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 7085–7095.

[76] S. Shi and X. Chu, "Speeding up convolutional neural networks by exploiting the sparsity of rectifier units," 2017, *arXiv:1704.07724*. [Online]. Available: http://arxiv.org/abs/1704.07724

[77] U. Gupta *et al.*, "MASR: A modular accelerator for sparse RNNs," in *Proc. 28th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Sep. 2019, pp. 1–14.

[78] H. Wang, Z. Zhang, and S. Han, "SpAtten: Efficient sparse attention architecture with cascade token and head pruning," 2020, *arXiv:2012.09852*. [Online]. Available: http://arxiv.org/abs/2012.09852

[79] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "GANAX: A unified MIMD-SIMD acceleration for generative adversarial networks," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 650–661.

[80] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2015, *arXiv:1511.06434*. [Online]. Available: http://arxiv.org/abs/1511.06434

[81] X. Dong *et al.*, "Acorns: A framework for accelerating deep neural networks with input sparsity," in *Proc. 28th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Sep. 2019, pp. 178–191.

[82] M. Ren, A. Pokrovsky, B. Yang, and R. Urtasun, "SBNet: Sparse blocks network for fast inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 8711–8720.

[83] M. Engelcke, D. Rao, D. Z. Wang, C. H. Tong, and I. Posner, "Vote3Deep: Fast object detection in 3D point clouds using efficient convolutional neural networks," in *Proc. IEEE Int. Conf. Robot. Automat. (ICRA)*, May 2017, pp. 1355–1361.

[84] Y. Lin, S. Han, H. Mao, Y. Wang, and B. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," in *Proc. Int. Conf. Learn. Represent.*, 2018, pp. 1–14.

[85] V. Gupta *et al.*, "Training recommender systems at scale: Communication-efficient model and data parallelism," 2020, *arXiv:2010.08899*. [Online]. Available: http://arxiv.org/abs/2010.08899

[86] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," in *Proc. 26th Int. Conf. World Wide Web*, 2017, pp. 173–182.

[87] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood, "Understanding training efficiency of deep learning recommendation models at scale," 2020, *arXiv:2011.05497*. [Online]. Available: http://arxiv.org/abs/2011.05497

[88] M. Yan *et al.*, "HyGCN: A GCN accelerator with hybrid architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 15–29.

[89] T. Geng *et al.*, "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 922–936.

[90] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2020, pp. 255–265.

[91] S. Liang *et al.*, "EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Trans. Comput.*, early access, Aug. 6, 2020, doi: 10.1109/TC.2020.3014632.

[92] I. S. Duff, "A survey of sparse matrix research," *Proc. IEEE*, vol. 65, no. 4, pp. 500–535, Apr. 1977.

[93] K. Hegde *et al.*, "ExTensor: An accelerator for sparse tensor algebra," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 319–333.

[94] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2014, pp. 10–14.

[95] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proc. IEEE Conf. Comput. Vis.*

Pattern Recognit. (CVPR), Jul. 2017, pp. 1492–1500.
[96] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.
[97] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 2818–2826.
[98] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "UCNN: Exploiting computational reuse in deep neural networks via weight repetition," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 674–687.
[99] M. Riera, J.-M. Arnau, and A. Gonzalez, "Computation reuse in DNNs by exploiting input similarity," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 57–68.
[100] P. Warden. (2015). *Why are Eight Bits Enough for Deep Neural Networks?* [Online]. Available: https://petewarden.com/2015/05/23/why-are-eight-bits-enough-for-deep-neural-networks/
[101] D. Kalamkar *et al.*, "A study of BFLOAT16 for deep learning training," 2019, *arXiv:1905.12322*. [Online]. Available: http://arxiv.org/abs/1905.12322
[102] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6869–6898, 2017.
[103] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong, "LogNet: Energy-efficient neural networks using logarithmic computation," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Mar. 2017, pp. 5900–5904.
[104] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 269–284, Apr. 2014.
[105] E. Qin *et al.*, "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 58–70.
[106] M. Adelman, K. Y. Levy, I. Hakimi, and M. Silberstein, "Faster neural network training with approximate tensor operations," 2018, *arXiv:1805.08079*. [Online]. Available: http://arxiv.org/abs/1805.08079
[107] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, "SNAP: A 1.67–21.55 TOPS/W sparse neural acceleration processor for unstructured sparse deep neural network inference in 16 nm CMOS," in *Proc. Symp. VLSI Circuits*, 2019, pp. C306–C307.
[108] J. Albericio *et al.*, "Bit-pragmatic deep neural network computing," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2017, pp. 382–394.
[109] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 Envision: A 0.26-to-10 TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28 nm FDSOI," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2017, pp. 246–247.
[110] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 924–939.
[111] J. J. Zhang, P. Raj, S. Zarar, A. Ambardekar, and S. Garg, "CompAct: On-chip compression of activations for low power systolic array based CNN acceleration," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5s, p. 47, 2019.
[112] P. Judd, A. Delmas, S. Sharify, and A. Moshovos, "Cnvlutin2: Ineffectual-activation-and-weight-free deep neural network computing," 2017, *arXiv:1705.00125*. [Online]. Available: http://arxiv.org/abs/1705.00125
[113] S. Zheng, Y. Liu, S. Yin, L. Liu, and S. Wei, "An efficient kernel transformation architecture for binary- and ternary-weight neural network inference," in *Proc. 55th Annu. Design Automat. Conf.*, Jun. 2018, p. 137.
[114] R. Struharik, B. Vukobratović, A. Erdeljan, and D. Rakanović, "CoNNA—Compressed CNN hardware accelerator," in *Proc. 21st Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2018, pp. 365–372.
[115] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 27–40.
[116] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 151–165.
[117] Z. Yuan *et al.*, "Sticker: A 0.41–62.1 TOPS/W 8 bit neural network processor with multi-sparsity compatible convolution arrays and online tuning acceleration for fully connected layers," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2018, pp. 33–34.
[118] S. Chole, R. Tadishetti, and S. Reddy, "SparseCore: An accelerator for structurally sparse CNNs," in *Proc. SysML Conf.*, 2018, pp. 1–3.
[119] L. Yavits and R. Ginosar, "Accelerator for sparse machine learning," *IEEE Comput. Archit. Lett.*, vol. 17, no. 1, pp. 21–24, Jan. 2018.
[120] NVIDIA Corporation. *NVIDIA Deep Learning Accelerator (NVDLA)*. Accessed: Nov. 5, 2018. [Online]. Available: http://nvdla.org
[121] A. Aimar *et al.*, "NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 3, pp. 644–656, Mar. 2019.
[122] A. Page, A. Jafari, C. Shea, and T. Mohsenin, "SPARCNet: A hardware accelerator for efficient deployment of sparse convolutional networks," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, pp. 1–32, May 2017.
[123] L. Lu and Y. Liang, "SpWA: An efficient sparse winograd convolutional neural networks accelerator on FPGAs," in *Proc. 55th ACM/ESDA/IEEE Design Automat. Conf. (DAC)*, Jun. 2018, pp. 1–6.
[124] 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)*, Apr. 2019, pp. 17–25.
[125] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "DeltaRNN: A power-efficient recurrent neural network accelerator," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 21–30.
[126] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "ERIDANUS: Efficiently running inference of DNNs using systolic arrays," *IEEE Micro*, vol. 39, no. 5, pp. 46–54, Sep. 2019.
[127] H. T. Kung, B. McDanel, and S. Q. Zhang, "Adaptive tiling: Applying fixed-size systolic arrays to sparse convolutional neural networks," in *Proc. 24th Int. Conf. Pattern Recognit. (ICPR)*, Aug. 2018, pp. 1006–1011.
[128] S. Yin *et al.*, "A high energy efficient reconfigurable hybrid neural network processor for deep learning applications," *IEEE J. Solid-State Circuits*, vol. 53, no. 4, pp. 968–982, Apr. 2018.
[129] C.-E. Lee *et al.*, "Stitch-X: An accelerator architecture for exploiting unstructured sparsity in deep neural networks," in *Proc. SysML Conf.*, 2018, pp. 1–3.
[130] D. Kim, J. Ahn, and S. Yoo, "ZeNA: Zero-aware neural network accelerator," *IEEE Design Test*, vol. 35, no. 1, pp. 39–46, Feb. 2018.
[131] 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. Archit.*, Jun. 2019, pp. 250–263.
[132] B. McDanel, S. Q. Zhang, H. T. Kung, and X. Dong, "Full-stack optimization for accelerating CNNs using powers-of-two weights with FPGA validation," in *Proc. ACM Int. Conf. Supercomput.*, Jun. 2019, pp. 449–460.
[133] P. N. Whatmough, S. K. Lee, D. Brooks, and G.-Y. Wei, "DNN engine: A 28-nm timing-error tolerant sparse deep neural network processor for IoT applications," *IEEE J. Solid-State Circuits*, vol. 53, no. 9, pp. 2722–2731, Sep. 2018.
[134] G. Venkatesh, E. Nurvitadhi, and D. Marr, "Accelerating deep convolutional networks using low-precision and sparsity," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Mar. 2017, pp. 2861–2865.
[135] T. J. Ham *et al.*, "A^3: Accelerating attention mechanisms in neural networks with approximation," 2020, *arXiv:2002.10941*. [Online]. Available: https://arxiv.org/abs/2002.10941
[136] X. He *et al.*, "Sparse-TPU: Adapting systolic arrays for sparse matrices," in *Proc. 34th ACM Int. Conf. Supercomput.*, Jun. 2020, pp. 1–12.
[137] R. Shi *et al.*, "CSB-RNN: A faster-than-realtime RNN acceleration framework with compressed structured blocks," in *Proc. 34th ACM Int. Conf. Supercomput.*, Jun. 2020, pp. 1–12.
[138] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2016, pp. 2383–2392.
[139] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, pp. 1–30, Oct. 2018.
[140] S. Smith *et al.* (2017). *Frostt File Format*. [Online]. Available: http://frostt.io/tensors/file-formats.html
[141] National Institute of Standards and Technology. (2013). *Matrix Market Exchange Formats*. [Online]. Available: https://math.nist.gov/MatrixMarket/formats.html
[142] E. Jones, T. Oliphant, and P. Peterson, "SciPy: Open source scientific tools for Python," Tech. Rep., 2001. [Online]. Available: http://www.scipy.org/
[143] Y. Saad, "SPARSKIT: A basic tool kit for sparse matrix computations," Res. Inst. Adv. Comput. Sci., NTRS, Moffett Field, CA, USA, Tech. Rep. NASA-CR-185876, 1990.
[144] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Apr. 2008, pp. 1–11.
[145] E.-J. Im and K. Yelick, "Model-based memory hierarchy optimizations for sparse matrices," in *Proc. Workshop Profile Feedback-Directed Compilation*, vol. 139, 1998, pp. 1–10.
[146] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proc. 5th Workshop Irregular Appl., Archit. Algorithms*, 2015, pp. 1–7.
[147] P. A. Tew, "An investigation of sparse tensor formats for tensor libraries," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, MA, USA, 2016.
[148] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*. [Online]. Available: http://arxiv.org/abs/1409.1556
[149] M. Buckler, P. Bedoukian, S. Jayasuriya, and A. Sampson, "EVA^2: Exploiting temporal redundancy in live computer vision," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 533–546.
[150] K. Guo, S. Han, S. Yao, Y. Wang, Y. Xie, and H. Yang, "Software-hardware codesign for efficient neural network acceleration," *IEEE Micro*, vol. 37, no. 2, pp. 18–25, Mar./Apr. 2017.
[151] X. Ma *et al.*, "Non-structured DNN weight pruning—Is it beneficial in any platform?" 2019, *arXiv:1907.02124*. [Online]. Available: http://arxiv.org/abs/1907.02124
[152] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proc. 21st Annu. Symp. Parallelism Algorithms Archit. (SPAA)*, 2009,

pp. 233–244.
[153] C. C. Chang and C. J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 1–27, 2011.
[154] D. R. Kincaid and D. M. Young, "The ITPACK project: Past, present, and future," in *Elliptic Problem Solvers*. Amsterdam, The Netherlands: Elsevier, 1984, pp. 53–63.
[155] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: SIAM, 2003.
[156] J. King, T. Gilray, R. M. Kirby, and M. Might, "Dynamic sparse-matrix allocation on GPUs," in *Proc. Int. Conf. High Perform. Comput.* Cham, Switzerland: Springer, 2016, pp. 61–80.
[157] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," in *Proc. 20th Annu. Int. Conf. Supercomput. (ICS)*, 2006, pp. 307–316.
[158] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin, "Efficient and scalable computations with sparse tensors," in *Proc. IEEE Conf. High Perform. Extreme Comput.*, Sep. 2012, pp. 1–6.
[159] B. W. Bader and T. G. Kolda, "Efficient MATLAB computations with sparse and factored tensors," *SIAM J. Sci. Comput.*, vol. 30, no. 1, pp. 205–231, Dec. 2007.
[160] R. W. Vuduc and J. W. Demmel, *Automatic Performance Tuning of Sparse Matrix Kernels*, vol. 1. Berkeley, CA, USA: Univ. California, Berkeley, 2003.
[161] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, Feb. 2019, pp. 300–314.
[162] B. W. Bader *et al.* (Feb. 2015). *MATLAB Tensor Toolbox Version 2.6*. [Online]. Available: http://www.sandia.gov/~tgkolda/TensorToolbox/
[163] NVIDIA. *Cusparse, the CUDA Sparse Matrix Library*. [Online]. Available: http://docs.nvidia.com/cuda/cusparse
[164] Z. Yuan *et al.*, "STICKER: An energy-efficient multi-sparsity compatible accelerator for convolutional neural networks in 65-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 55, no. 2, pp. 465–477, Feb. 2020.
[165] C. Ding *et al.*, "CirCNN: Accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2017, pp. 395–408.
[166] S. Wang *et al.*, "C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 11–20.
[167] M. Yan *et al.*, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 615–628.
[168] K. Hegde, R. Agrawal, Y. Yao, and C. W. Fletcher, "Morph: Flexible acceleration for 3D CNN-based video understanding," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 933–946.
[169] Y. Kim, J. Lee, A. Shrivastava, J. W. Yoon, D. Cho, and Y. Paek, "High throughput data mapping for coarse-grained reconfigurable architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 11, pp. 1599–1609, Nov. 2011.
[170] N. H. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. London, U.K.: Pearson, 2015.
[171] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 535–547.
[172] A. Azizimazreah and L. Chen, "Shortcut mining: Exploiting cross-layer shortcut reuse in DCNN accelerators," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 94–105.
[173] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.
[174] D. Vainbrand and R. Ginosar, "Network-on-chip architectures for neural networks," in *Proc. 4th ACM/IEEE Int. Symp. Netw.-Chip*, May 2010, pp. 135–144.
[175] H. Kwon, A. Samajdar, and T. Krishna, "Rethinking NoCs for spatial neural network accelerators," in *Proc. 11th IEEE/ACM Int. Symp. Netw.-Chip (NOCS)*, Oct. 2017, pp. 1–8.
[176] P. Xu *et al.*, "AutoDNNchip: An automated DNN chip predictor and builder for both FPGAs and ASICs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2020, pp. 40–50.
[177] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 461–475, 2018.
[178] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 553–564.
[179] S. Dave, A. Shrivastava, Y. Kim, S. Avancha, and K. Lee, "DMazeRunner: Optimizing convolutions on dataflow accelerators," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2020, pp. 1544–1548.
[180] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An architecture for ultralow power binary-weight CNN acceleration," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 48–60, Jan. 2018.
[181] H. Tann, S. Hashemi, R. I. Bahar, and S. Reda, "Hardware-software codesign of accurate, multiplier-free deep neural networks," in *Proc. 54th ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jun. 2017, pp. 1–6.
[182] H. Sharma *et al.*, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 764–775.
[183] S. Sharify *et al.*, "Laconic deep learning inference acceleration," in *Proc. 46th Int. Symp. Comput. Archit.*, Jun. 2019, pp. 304–317.
[184] H. Sharma *et al.*, "From high-level deep neural models to FPGAs," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, p. 17.
[185] A. D. Lascorz *et al.*, "Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Apr. 2019, pp. 749–763.
[186] A. Parashar *et al.*, "Timeloop: A systematic approach to DNN accelerator evaluation," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2019, pp. 304–315.
[187] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "HyPar: Towards hybrid parallelism for deep learning accelerator array," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 56–68.
[188] L. R. Gonçalves, R. F. D. Moura, and L. Carro, "Aggressive energy reduction for video inference with software-only strategies," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5s, pp. 1–20, Oct. 2019.
[189] H. Mahdiani, A. Khadem, A. Ghanbari, M. Modarressi, F. Fattahi, and M. Daneshtalab, "δNN: Power-efficient neural network acceleration using differential weights," *IEEE Micro*, vol. 40, no. 1, pp. 67–74, Jan./Feb. 2019.
[190] M. Mahmoud, K. Siu, and A. Moshovos, "Diffy: A Déjà vu-free differential deep neural network accelerator," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 134–147.
[191] F. Silfa, G. Dot, J.-M. Arnau, and A. Gonzàlez, "Neuron-level fuzzy memoization in RNNs," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 782–793.
[192] Y. Wang, S. Liang, H. Li, and X. Li, "A none-sparse inference accelerator that distills and reuses the computation redundancy in CNNs," in *Proc. 56th Annu. Design Automat. Conf.*, Jun. 2019, p. 202.
[193] Y. Zhu, A. Samajdar, M. Mattina, and P. Whatmough, "Euphrates: Algorithm-SoC co-design for low-power mobile continuous vision," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 547–560.
[194] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 662–673.
[195] J. Zhu, J. Jiang, X. Chen, and C.-Y. Tsui, "SparseNN: An energy-efficient neural network accelerator exploiting input and output sparsity," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 241–244.
[196] D. Lee, S. Kang, and K. Choi, "ComPEND: Computation pruning through early negative detection for ReLU in a deep neural network accelerator," in *Proc. Int. Conf. Supercomput.*, Jun. 2018, pp. 139–148.
[197] Y. Miao, M. Gowayyed, and F. Metze, "EESEN: End-to-end speech recognition using deep RNN models and WFST-based decoding," in *Proc. IEEE Workshop Automat. Speech Recognit. Understand. (ASRU)*, Dec. 2015, pp. 167–174.
[198] M. Bojarski *et al.*, "End to end learning for self-driving cars," 2016, *arXiv:1604.07316*. [Online]. Available: http://arxiv.org/abs/1604.07316
[199] D. Amodei *et al.*, "Deep speech 2: End-to-end speech recognition in English and Mandarin," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 173–182.
[200] E. Real, J. Shlens, S. Mazzocchi, X. Pan, and V. Vanhoucke, "YouTube-BoundingBoxes: A large high-precision human-annotated data set for object detection in video," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 5296–5305.
[201] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," 2014, *arXiv:1404.5997*. [Online]. Available: http://arxiv.org/abs/1404.5997
[202] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik, "Neural network distiller: A Python package for DNN compression research," 2019, *arXiv:1910.12232*. [Online]. Available: http://arxiv.org/abs/1910.12232
[203] Intel. *Understanding Memory Formats, Intel MKL-DNN*. Accessed: Mar. 3, 2020. [Online]. Available: https://intel.github.io/mkl-dnn/understanding_memory_formats.html
[204] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, Nov. 2014, pp. 675–678.
[205] R. Baghdadi *et al.*, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2019, pp. 193–205.
[206] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2018, pp. 1–17.
[207] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 1–12, Aug. 2012.
[208] C. Lattner *et al.*, "MLIR: A compiler infrastructure for the end of Moore's law," 2020, *arXiv:2002.11054*. [Online]. Available: https://arxiv.org/abs/2002.11054
[209] F. Paul and L. Christian, "The polyhedron model," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA, USA: Springer, 2011, pp. 1581–1592.
[210] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Proc. Int. Congr. Math. Softw.* Berlin, Germany: Springer, 2010, pp. 299–302.
[211] R. Baghdadi *et al.*, "PENCIL: A platform-neutral compute intermediate language for accelerator programming," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, Oct. 2015, pp. 138–149.
[212] N. Vasilache *et al.*, "Tensor comprehensions:

Framework-agnostic high-performance machine learning abstractions," 2018, *arXiv:1802.04730*. [Online]. Available: http://arxiv.org/abs/1802.04730

[213] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover, "Diesel: DSL for linear algebra and neural net computations on GPUs," in *Proc. 2nd ACM SIGPLAN Int. Workshop Mach. Learn. Program. Lang.*, Jun. 2018, pp. 42–51.

[214] C. Leary and T. Wang, "XLA: TensorFlow, compiled," TensorFlow Dev Summit, Tech. Rep., 2017. [Online]. Available: https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html

[215] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proc. PLDI*, 2008, pp. 101–113.

[216] T. Grosser, A. Groslinger, and C. Lengauer, "Polly—Performing polyhedral optimizations on a low-level intermediate representation," *Parallel Process. Lett.*, vol. 22, no. 4, 2012, Art. no. 1250010. [Online]. Available: http://dblp.uni-trier.de/db/journals/ppl/ppl22.html#GrosserGL12

[217] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic optimization for image processing pipelines," in *Proc. 20th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2015, pp. 429–443.

[218] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "AlphaZ: A system for design space exploration in the polyhedral model," in *Proc. Int. Workshop Lang. Compil. Parallel Comput.* Berlin, Germany: Springer, 2012, pp. 17–31.

[219] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," Univ. Southern California, Los Angeles, CA, USA, Tech. Rep. 08-897, 2008.

[220] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proc. 19th Joint Eur. Conf. Theory Pract. Softw., Int. Conf. Compiler Construct. (CC/ETAPS)*. Berlin, Germany: Springer, 2010.

[221] A. Hartono *et al.*, "Parametric multi-level tiling of imperfectly nested loops," in *Proc. 23rd Int. Conf. Conf. Supercomput. (ICS)*, 2009, pp. 147–157.

[222] R. Baghdadi *et al.*, "PENCIL language specification," INRIA, Paris, France, Res. Rep. RR-8706, 2015. [Online]. Available: https://hal.inria.fr/hal-01154812

[223] R. Baghdadi and A. Cohen, "Scalable polyhedral compilation, syntax vs. semantics: 1–0 in the first round," in *Proc. IMPACT Workshop Associated With HIPEAC*, 2020, pp. 1–12.

[224] R. Wei, L. Schwartz, and V. Adve, "DLVM: A modern compiler infrastructure for deep learning systems," 2017, *arXiv:1711.03016*. [Online]. Available: http://arxiv.org/abs/1711.03016

[225] L. Truong *et al.*, "Latte: A language, compiler, and runtime for elegant and efficient deep neural networks," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2016, pp. 209–223.

[226] P. Feautrier, "Dataflow analysis of array and scalar references," *Int. J. Parallel Program.*, vol. 20, no. 1, pp. 23–53, Feb. 1991.

[227] M. E. Wolf, "Improving locality and parallelism in nested loops," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 1992.

[228] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, and F. Durand, "Taichi: A language for high-performance computation on spatially sparse data structures," *ACM Trans. Graph.*, vol. 38, no. 6, pp. 1–16, Nov. 2019.

[229] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, pp. 1–29, Oct. 2017.

[230] K. Goto and R. A. V. D. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 1–25, May 2008.

[231] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-model guided loop-nest auto-vectorization," in *Proc. 18th Int. Conf. Parallel Archit. Compilation Techn.*, Sep. 2009, pp. 327–337.

[232] F. Agakov *et al.*, "Using machine learning to focus iterative optimization," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2006, p. 11.

[233] A. Adams *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Trans. Graph.*, vol. 38, no. 4, pp. 1–12, Jul. 2019.

[234] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 4505–4515.

[235] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2004, pp. 75–86.

[236] S. Venkataramani *et al.*, "ScaleDeep: A scalable compute architecture for learning and evaluating deep networks," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 13–26, 2017.

[237] Y. Chen *et al.*, "An instruction set architecture for machine learning," *ACM Trans. Comput. Syst.*, vol. 36, no. 3, pp. 1–35, 2019.

[238] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma, "Compiling KB-sized machine learning models to tiny IoT devices," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2019, pp. 79–95.

[239] T. Moreau *et al.*, "A hardware-software blueprint for flexible deep learning specialization," 2018, *arXiv:1807.04188*. [Online]. Available: http://arxiv.org/abs/1807.04188

[240] T. Chen *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015, *arXiv:1512.01274*. [Online]. Available: http://arxiv.org/abs/1512.01274

[241] F. Tung and G. Mori, "CLIP-Q: Deep network compression learning by in-parallel pruning-quantization," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 7873–7882.

[242] H. Yang, S. Gui, Y. Zhu, and J. Liu, "Automatic neural network compression by sparsity-quantization joint learning: A constrained optimization-based approach," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2019, pp. 2178–2188.

[243] K. Kwon, A. Amid, A. Gholami, B. Wu, K. Asanovic, and K. Keutzer, "Co-design of deep neural nets and neural net accelerators for embedded vision applications," in *Proc. 55th ACM/ESDA/IEEE Design Automat. Conf. (DAC)*, Jun. 2018, pp. 1–6.

[244] D. Marculescu, D. Stamoulis, and E. Cai, "Hardware-aware machine learning: Modeling and optimization," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, pp. 1–8.

[245] X. Zhang, W. Jiang, Y. Shi, and J. Hu, "When neural architecture search meets hardware implementation: From hardware awareness to co-design," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2019, pp. 25–30.

[246] M. S. Abdelfattah, Ł. Dudziak, T. Chau, R. Lee, H. Kim, and N. D. Lane, "Best of both worlds: AutoML codesign of a CNN and its hardware accelerator," 2020, *arXiv:2002.05022*. [Online]. Available: http://arxiv.org/abs/2002.05022

[247] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "AMC: AutoML for model compression and acceleration on mobile devices," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 784–800.

[248] H. Ji, L. Song, L. Jiang, H. Li, and Y. Chen, "ReCom: An efficient resistive accelerator for compressed deep neural networks," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 237–240.

[249] P. Wang, Y. Ji, C. Hong, Y. Lyu, D. Wang, and Y. Xie, "SNrram: An efficient sparse neural network computation architecture based on resistive random-access memory," in *Proc. 55th ACM/ESDA/IEEE Design Automat. Conf. (DAC)*, Jun. 2018, p. 106.

[250] X. Zhang *et al.*, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, p. 56.

[251] N. Srivastava *et al.*, "T2S-tensor: Productively generating high-performance spatial hardware for dense tensor computations," in *Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2019, pp. 181–189.

[252] Y.-H. Lai *et al.*, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2019, pp. 242–251.

[253] R. Venkatesan *et al.*, "MAGNet: A modular accelerator generator for neural networks," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2019, pp. 1–8.

[254] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proc. DAC Design Automt. Conf.*, 2012, pp. 1212–1221.

[255] A. Sharifian *et al.*, "μir—An intermediate representation for transforming and optimizing the microarchitecture of application accelerators," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 940–953.

[256] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.

[257] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, Jun. 2015, pp. 105–117.

[258] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2014, pp. 255–268.

[259] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000X acceleration on long read assembly," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2018, pp. 199–213.

[260] D. Fujiki *et al.*, "GenAx: A genome sequencing accelerator," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 69–82.

[261] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on FPGAs," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2015, pp. 52–59.

[262] J. Gu *et al.*, "Recent advances in convolutional neural networks," *Pattern Recognit.*, vol. 77, pp. 354–377, May 2018.

[263] Y. Wei *et al.*, "A review of algorithm & hardware design for AI-based biomedical applications," *IEEE Trans. Biomed. Circuits Syst.*, vol. 14, no. 2, pp. 145–163, Apr. 2020.

[264] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *J. Mach. Learn. Res.*, vol. 20, no. 1, pp. 1997–2017, 2019.

[265] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "Model compression and acceleration for deep neural networks: The principles, progress, and challenges," *IEEE Signal Process. Mag.*, vol. 35, no. 1, pp. 126–136, Jan. 2018.

[266] E. Wang *et al.*, "Deep neural network approximation for custom hardware: Where we've been, where we're going," *ACM Comput. Surv.*, vol. 52, no. 2, p. 40, 2019.

[267] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[268] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–39, Jul. 2018.

[269] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey and benchmarking of machine learning accelerators," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2019, pp. 1–9.

[270] M. Li *et al.*, "The deep learning compiler: A comprehensive survey," 2020, *arXiv:2002.03794*. [Online]. Available: http://arxiv.org/abs/2002.03794

[271] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Comput. Appl.*, vol. 32, pp. 1–31, Oct. 2018.

[272] B. Z. Du, Q. Guo, Y. Zhao, T. Zhi, Y. Chen, and Z. Xu, "Self-aware neural network systems: A survey and new perspective," *Proc. IEEE*, vol. 108, no. 7, pp. 1047–1067, Jul. 2020.

[273] A. Ignatov *et al.*, "AI benchmark: All about deep learning on smartphones in 2019," 2019, *arXiv:1910.06663*. [Online]. Available: http://arxiv.org/abs/1910.06663

[274] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse GPU kernels for deep learning," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2020, pp. 1–14.

ABOUT THE AUTHORS

Shail Dave (Graduate Student Member, IEEE) is currently working toward the Ph.D. degree at the School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ, USA.

His research interests include reconfigurable hardware accelerators, hardware/software codesign, and computer architecture. His current research focuses on developing the tools and techniques for coarse-grained programmable dataflow accelerators for machine learning, including mapping optimizations and design explorations.

Riyadh Baghdadi (Member, IEEE) received the master's and Ph.D. degrees from INRIA (Sorbonne University, Paris VI), France, in 2011 and 2015, respectively.

He is currently an Assistant Professor of computer science with the New York University at Abu Dhabi (NYU AD), Abu Dhabi, United Arab Emirates, and an Affiliated Researcher with the Massachusetts Institute of Technology (MIT), Cambridge, MA, USA, in 2002 and 2005, respectively. He works on the intersection of compilers and applied machine learning. More precisely, he works on developing compilers that take high-level code and optimize it automatically to generate highly efficient code. He uses machine learning to automate optimizations in these compilers.

Tony Nowatzki (Member, IEEE) received the Ph.D. degree from the University of Wisconsin–Madison, Madison, WI, USA, in 2016.

He joined the University of California at Los Angeles (UCLA), Los Angeles, CA, USA, in 2017. He is currently an Assistant Professor of computer science with UCLA, where he leads the PolyArch Research Group. His research interests include computer architecture, microarchitecture, hardware specialization, and compiler codesign.

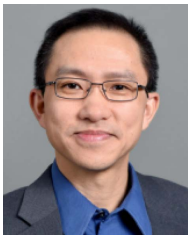
Sasikanth Avancha (Member, IEEE) received the B.E. degree in computer science and engineering from the University Visvesvaraya College of Engineering (UVCE), Bengaluru, India, in 1994, and the M.S. and Ph.D. degrees in computer science from the University of Maryland at Baltimore County (UMBC), Baltimore, MD, USA, in 2002 and 2005, respectively.

He is currently a Senior Research Scientist with the Parallel Computing Lab, Intel Labs, Intel India, Bengaluru. He has over 20 years of industry and research experience. He has three patents and over 25 articles spanning security, wireless networks, systems software, and computer architecture. His current research focuses on high-performance algorithm development and analysis for large-scale, distributed deep learning and machine learning training, and inference on different data-parallel architectures, including x86 and other accelerators, across application domains.

Aviral Shrivastava (Senior Member, IEEE) received the bachelor's degree in computer science and engineering from IIT Delhi, New Delhi, India, in 1999, and the master's and Ph.D. degrees in information and computer science from the University of California at Irvine, Irvine, CA, USA, in 2002 and 2006, respectively.

He is currently a Professor with the School of Computing, Informatics, and Decision Systems Engineering, Arizona State University (ASU), Tempe, AZ, USA, where he leads the Make Programming Simple Lab. His research interests include: 1) compilers and microarchitectures for heterogeneous, accelerated, and many-core computing; 2) protecting computation from soft errors; and 3) precise timing for cyber–physical systems.

Prof. Shrivastava was a recipient of the 2011 NSF CAREER Award and the 2012 Outstanding Junior Researcher in CSE at ASU. His works have received several best paper nominations, including at the Design Automation Conference (DAC) 2017, and the Best Student Paper Award at the International Conference on VLSI Design (VLSID) 2016. He is also the Chair of the Design and Application Track of the IEEE Real-Time Systems Symposium (RTSS) 2020 and the Conference Chair of Embedded Systems Week (ESWEEK) 2020. He also serves as the Deputy Editor-in-Chief of IEEE EMBEDDED SYSTEMS LETTERS (ESL) and an Associate Editor for *ACM Transactions on Embedded Computing Systems* (TECS) and IEEE TRANSACTIONS ON COMPUTER AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS (TCAD).



Baoxin Li (Senior Member, IEEE) received the Ph.D. degree in electrical engineering from the University of Maryland, College Park, MD, USA, in 2000.

He joined Arizona State University (ASU), Tempe, AZ, USA, in 2004, where he is currently a Professor of computer science and engineering and a Graduate Faculty Endorsed to Chair in Electrical Engineering and Computer Engineering Programs. From 2000 to 2004, he was a Senior Researcher with the SHARP Laboratories of America, Camas, WA, USA, where he was the Technical Lead in developing SHARP's HiIMPACT Sports Technologies. He was an Adjunct Professor with Portland State University, Portland, OR, USA, from 2003 to 2004. He holds 18 issued U.S. Patents. His work has been featured on NY Times, EE Times, MSNBC, Discovery News, ABC News, Gizmodo India, and other media sources. His general research interests are in visual computing and machine learning, especially their application in the context of human-centered computing. He actively works on computer vision and pattern recognition, multimedia, image/video processing, assistive technologies, human–computer interaction, and statistical methods in visual computing.

Dr. Li won twice SHARP Labs' President Awards. He also won the SHARP Labs' Inventor of the Year Award in 2002. He was a recipient of the National Science Foundation's CAREER Award. He was named a Fulton Exemplar Faculty at ASU in 2017.