A Balanced Sparse Matrix Convolution Accelerator for Efficient CNN Training

Yuechen Chen, *Member, IEEE*, Ahmed Louri, *Fellow, IEEE*, Shanshan Liu, *Senior Member, IEEE*, and Fabrizio Lombardi, *Life Fellow, IEEE*

***Abstract*— Sparse Convolutional Neural Network (CNN) training is well known to be time-consuming due to significant off-chip memory traffic. To effectively deploy sparse training, existing accelerators store matrices in a compressed format to eliminate memory accesses for zeros; hence, accelerators are designed to process compressed matrices to avoid zero computations. We have observed that the compression rate is greatly affected by the sparsity in the matrices with different formats. Given the varying levels of sparsity in activations, weights, errors, and gradients matrices throughout the sparse training process, it becomes impractical to achieve consistently high compression rates using a singular compression method for the entire duration of the training. Moreover, random zeros in the matrices result in irregular computation patterns, further increasing execution time. To address these issues, we propose a balanced sparse matrix convolution accelerator design for efficient CNN training. Specifically, a dual matrix compression technique is developed that seamlessly combines two widely used sparse matrix compression formats with a control algorithm for lower memory traffic during training. Based on this compression technique, a two-level workload balancing technique is then designed to further reduce the execution time and energy consumption. Finally, an accelerator is implemented to support the proposed techniques. The cycle-accurate simulation results show that the proposed accelerator reduces the execution time by 34% and the energy consumption by 24% on average compared to existing sparse training accelerators.**

***Index Terms*— Convolutional neural network, training, sparse matrix compression, memory traffic, load balancing.**

I. Introduction

CONVOLUTIONAL Neural Networks (CNNs) have emerged as a fundamental technique for solving complex Machine Learning (ML) problems across a wide range of domains, including image processing, natural language processing, and autonomous vehicles [1], [2]. To efficiently deploy this tool, the sparse CNN model training is a quintessential step for solving ML problems accurately during model inference. Compared to conventional training, the sparse training process results in a smaller model with no degradation in accuracy; this allows for a faster and more efficient inference during deployment. Thus, the sparse training scheme is widely deployed in many real-world scenarios, such as training models for IoT and mobile devices [3]. Existing sparsification processes used in training can be organized into two categories, namely, model and ephemeral sparsification [4].

- The model sparsification process sets selected weights to zeros [5], [6], [7]. For example, the iterative model pruning process selects weight based on the value magnitude and gradually increases the sparsity of the weight matrix; thus, a substantial number of zero values is observed in weight matrices after this process.
- The ephemeral sparsification process sets selected activations to zero during training [8], [9]. For example, the dropout function randomly selects and sets around 30% to 70% of the activation of a fully connected layer to zero. The dropout not only increases the sparsity of the activation matrices but also solves the model overfitting problem during training.

Although the sparse training process has multiple advantages, this process is also known to be time-consuming due to significant off-chip memory traffic and irregular computation patterns [3], [10], [11], [12], [13], [14], [15]. As reported in the technical literature, an average of 60% of the execution time is spent on off-chip memory access [14]. Moreover, the random location of zeros causes irregular computation patterns, which further reduces the utilization of on-chip computation resources [3], [12], [13], [16], [17], [18]. As the size of CNN models continues to increase for better problem-solving capability, new accelerator designs are needed for more efficient sparse training with less training time and energy consumption.

To reduce memory traffic for sparse training, existing sparse-training accelerators are designed to process compressed sparse matrices [12], [13], [16], [17], [18]. Two widely used compressed sparse matrix formats are the compressed sparse row (CSR) [12], [13], [16], [19], and the bitmap format (BF) [17], [18]. Both these compression formats have two parts, including non-zero elements and indices. The indices

Manuscript received 20 April 2023; revised 13 November 2023 and 9 April 2024; accepted 11 July 2024. This work was supported in part by NSF under Grant CCF-1953961, Grant CCF-1812467, Grant CCF-1812495, Grant CCF-1953980, Grant CCF-1901165, Grant CCF-2131946, Grant CCF-2311543, and Grant CCF-2324644. This article was recommended by Associate Editor C. Liu. *(Corresponding author: Yuechen Chen.)*

Yuechen Chen is with the Department of Computer Science and Information Technologies, Frostburg State University, Frostburg, MD 21532 USA (e-mail: ychen@frostburg.edu).

Ahmed Louri is with the Department of Electrical and Computer Engineering, George Washington University, Washington, DC 20052 USA (e-mail: louri@gwu.edu).

Shanshan Liu is with the School of Information and Communication Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China (e-mail: ssliu@uestc.edu.cn).

Fabrizio Lombardi is with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115 USA (e-mail: lombardi@ece.neu.edu).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSI.2024.3430831.

Digital Object Identifier 10.1109/TCSI.2024.3430831

of CSR feature two integer arrays as column and row indices to indicate the location of the non-zero elements; the indices of BF are represented by a binary bitmap. The use of these matrix formats during sparse training not only allows the accelerator to identify and eliminate multiplication operations with a product of zero, but it also eliminates memory access of the zero values from off-chip DRAM. To process the compressed matrices, the existing accelerators [12], [13], [16], [17], [18] also include a custom data flow to distribute workloads among multiple processing elements (PEs); within each PE, an index computing unit (ICU) is implemented to only allow non-zero computation for the computation units (CUs), which performs multiplication and addition operations.

However, existing accelerator designs have two major limitations.

1) Relying on either CSR or BF limits the potential savings in memory traffic during sparse training. The problem is mainly due to the relationship between the size of the indices and matrix sparsity; with a known matrix dimension and sparsity, BF has a fixed-size bitmap whose dimension is the same as the original matrix. Also, the CSR format depends on the number of non-zero elements in the matrices, so more non-zero elements result in more entries in the column index array. As the matrix sparsity changes during the training process, two different index formats have different impacts on memory traffic. When matrix sparsity is low, the CSR format requires more memory accesses for indices compared to BF due to a longer column array. When matrix sparsity is high, the CSR format achieves a smaller data size compared to BF if the size of two integer arrays is smaller than the bitmap.
2) The ICU implemented in each PE evenly distributes the computation to the CUs; however, the workload distribution between PEs is not balanced during sparse training. Existing designs [3], [12], [13], [16], [17], [18] evenly distribute data based on the dimension of the matrices rather than on the computation workload for the convolution operation. Therefore, the uneven computation workload distribution between PEs reduces the level of parallelism during the sparse training process; this causes a longer computation time and more energy consumption.

To address these issues, in this paper, we propose dual matrix compression and two-level workload balancing techniques for efficient sparse CNN training.

1) The dual matrix compression technique introduces row-partitioned CSR (RP-CSR) and BF (RP-BF) formats to unify the memory access for non-zero elements and indices. This technique not only allows seamless format switching during training, but also includes a selection algorithm to choose a compression format with the highest compression rate for lower off-chip memory traffic.
2) Based on the dual matrix compression technique, a two-level workload balancing technique is developed; it includes coarse-grain and fine-grain load balancing mechanisms. The coarse-grain load-balancing mechanism adjusts the size of each matrix partition by estimating the computation required for each partition with a probability model. The fine-grain load balancing mechanism utilizes a workload-stealing mechanism to further reduce the difference in workload between PEs.

Moreover, to support two techniques, an accelerator is developed. The proposed accelerator is capable of computing sparse matrix convolution with the two proposed compression formats. The accelerator also supports the proposed workload balancing technique with a coarse-grain load balancing mechanism implemented in the global controller and a fine-grain load balancing mechanism implemented in each PE. The main contributions of the paper are as follows.

- The proposed techniques improve the efficiency of the sparse training process by maintaining a high matrix compression rate and balancing the computation workload.
- A row-partitioned data format is proposed for seamless switching between CSR and BF during computation.
- A probability model is developed to quickly estimate the computation requirement during training for a balanced workload distribution.
- An accelerator is implemented to support the proposed dual matrix compression and two-level workload balancing techniques.
- The evaluation results show that the proposed technique reduces the execution time by 34% and the energy consumption by 24% on average compared to existing sparse training accelerators.

II. Backgrounds

A. Sparse Convolution CNN Training

1) CNN Training: The stochastic gradient descent (SGD) algorithm is widely used for CNN model training. All involved matrices are weight (W), activation (A), error (E), and gradients (G), whose superscripts indicate the layer indices (l). The training process is divided into four phases, including forward propagation (FP), backward propagation (BP), weight-gradient generation (WG), and weight update (WU). The calculation processes are as follows.

1) FP: A training sample is fed into the model, and the activations of each layer are calculated for the next layer. For a convolution and fully connected layer, Eq. (1) describes the computation process, which is a convolution operation between the activation of the previous layer (A^{l-1}) and the weight of the current layer (W^l). At the end of the FP, the activation of the final layer is compared with the ground truth, and the error (E) for this FP is computed with a loss function.
$$A^l = A^{l-1} * W^l \tag{1}$$
2) BP: The computation procedure of BP is similar to FP, with a weight matrix rotated 180 degrees using the R(x) function. Eq. (2) describes the process for the convolution layer.
$$E^{l-1} = E^l * R(W^l) \tag{2}$$
3) WG: Weight gradients are the derivatives of the loss with respect to weights for a training sample. Eq. (3) shows

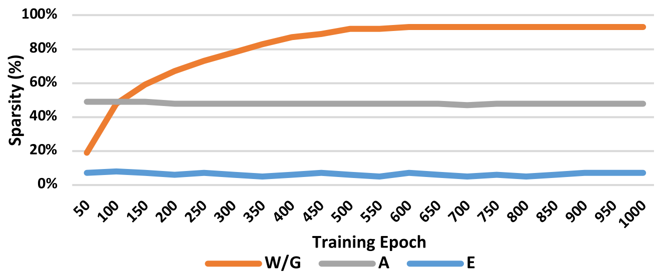


Fig. 1. Matrix sparsity for VGG11 under iterative pruning.

the gradient computation for the convolution layer and fully connected layer.

$$G^l = A^{l-1} * E^l \tag{3}$$

4) WU: One training epoch contains many training samples. Once the gradients for all the training samples are calculated, the gradients for the current training epoch (G^l_{t+1}) are the sum of all gradients calculated for each training sample. Then, the weights for the next training epoch are calculated using Eq. (4), where η is the learning rate.

$$W^l_{t+1} = W^l_t - \eta G^l_{t+1} \tag{4}$$

In a training epoch, the entire training dataset is divided into mini-batches, and then the weights are adjusted after observing each batch of training samples through this process. A typical training process contains thousands of training epochs to reach the desired accuracy. Since on-chip SRAM has a limited capacity, all matrices (*W, A, E, G*) need to be stored in off-chip memory (e.g., DRAM), resulting in heavy off-chip memory traffic and a long training time.

2) Existing Sparsification Processes: As CNN models have significant redundancy of weights and activations, sparsification processes are widely used for efficient training [3], [4], [5], [6], [7], [10], [18]. For model sparsification, this paper mainly focuses on the iterative model pruning method [5], [10], which is widely used due to its high accuracy and low training cost. In such a scheme, the pruned weights are selected by comparing each weight to a threshold, which is determined based on a user-defined sparsity for this epoch. After pruning, the values of the pruned weights are kept at zero in future training epochs. Since pruning results in accuracy model loss, more training epochs are needed to reach designated accuracy with desired sparsity [5], [10]. Fig. 1 shows the distribution of zeros in these matrices when iterative model pruning is applied to VGG11 using ImageNet [1]. The target pruning rate of 92% is achieved at 500 epochs, but training ends at 1000 epochs to ensure 69.04% accuracy.

In terms of ephemeral sparsification, this paper mainly focuses on the dropout and ReLU functions [10], [11]. The dropout function randomly nullifies activations during the training process, whereas ReLU changes the negative activations to zero. Considering a CNN model with multiple hidden layers, the dropout function randomly sets the activations for a layer to zero for each training sample. The ReLU function sets the negative activation for a layer to zero during FP.

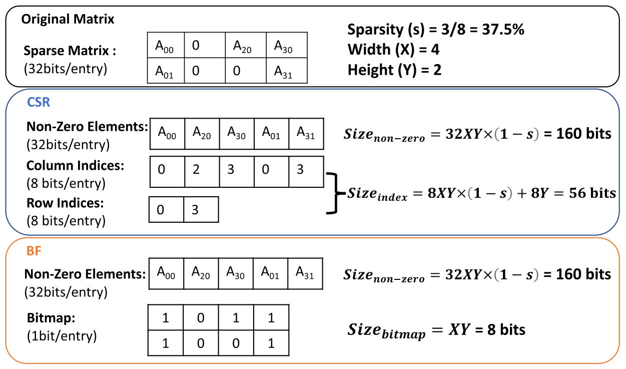


Fig. 2. CSR and BF used by existing sparse-training accelerators.

The existing sparsification processes result in both random zeros and dynamic sparsity. As shown in Fig. 1, the W/G, A, and E matrices are at different sparsity levels during the sparse training. Due to iterative model pruning, W/G sparsity changes according to the iterative model pruning method [5], [10]. The weight matrices are iteratively pruned at the beginning of the training (i.e., 1-500 epochs) to achieve the target pruning rate. Since the values of the pruned weights are kept at zero, the gradient matrices have the same sparsity as the weight matrices. After the target sparsity is reached, the rest of the training epochs (i.e., 500-1000 epochs) are used for the fine-tuning of the model to reach the designated accuracy. Due to the dropout and ReLU functions [10], [11] in the model, the sparsity of A and E is constant with a random zero location.

B. Existing Sparse Training Architecture

Consider the complexity of training CNN models; conventional homogeneous architectures (e.g., multicore CPU/GPU) are constrained due to the significant computation time and off-chip memory traffic. To efficiently process training workloads, multiple accelerator architectures with sparse matrix computation support have been proposed [11], [12], [13], [15], [16], [17], [18]. CSR and BF are the two most widely used sparse matrix compression formats for these sparse training accelerators. As shown in Fig. 2, the CSR format contains non-zero elements, row indices, and column indices. The column indices represent the column location of each non-zero element. The row indices indicate the index of the start of each row for non-zero elements. The BF format contains non-zero elements and a bitmap; the bitmap uses a 1 bit for each element in the matrix to indicate zero or non-zero elements at a specific position.

The process for a convolutional operation during sparse training on existing accelerators is described as follows.

1) The global controller initiates the convolutional operation by loading the compressed matrices from the off-chip memory to the global buffer.
2) The global controller distributes the matrices to the PEs according to the data flow of the accelerator and the mapping logic.
3) The PEs execute the convolution operation with matrix sparsification.

4) The computation results are sent back to the off-chip memory after the computation.

Therefore, existing accelerators typically consist of the following components to accomplish convolution operations for compressed sparse matrices:

1) **Computation Units (CUs).** CUs are implemented for multiplication and addition computation for each non-zero element during convolution operations.
2) **Index Computation Units (ICUs).** To compute the compressed matrixes without processing zeros, the ICUs are designed to take the indices of two input matrices and calculate the indices for the output matrices.
3) **Sparsification Unit.** The sparsification unit applies user-defined sparsification methods after the matrix-to-matrix convolution operation.
4) **On-Chip SRAM**. The on-chip SRAM loads and stores compressed matrices from and to the off-chip DRAM. To meet the need for the computation of compressed matrix format, the indices and non-zero elements are stored in two different fixed-size SRAM modules.
5) **Processing Element (PE).** Each PE contains multiple CUs, one ICU, one sparsification unit, and local SRAMs. Each PE computes the data in the local SRAM under a predetermined procedure.
6) **Global Controller.** The global controller manages the memory access for the global buffer and the computation process for all the PEs.

III. Motivations

1) **Existing compression formats limit the on-chip SRAM utilization.** Since the non-zero elements and indices are stored in separate tensors in existing compression formats, existing sparse-training accelerators implement two on-chip SRAMs to store non-zero elements and indices separately [13], [16], [17]. The ratio between the size of two SRAMs is fixed and cannot be changed during the training. Moreover, since the ratio of non-zero element size to index size depends on the matrix sparsity for a compression format, both on-chip SRAMs can be fully utilized only under a specific matrix sparsity in the current designs. Therefore, the fixed compression format reduces on-chip SRAM utilization during computation and increases off-chip memory traffic.
2) **Existing accelerator designs cannot efficiently handle irregular computation patterns.** In existing accelerator designs [3], [12], [13], [16], [17], [18], the ICUs are implemented in each PE. The computation workload for each PE is not analyzed when the global controller distributes the workload. Therefore, such an unbalanced workload reduces the level of parallelism during the convolution operation and increases the execution time. Some existing accelerator designs address this issue by introducing structured pruning methods [3], [15], [18]; even though these methods improve load balancing between PEs, structured pruning methods always result in the degradation of training accuracy.

IV. A Balanced Sparse Matrix Convolution Accelerator for Efficient CNN Training

A. Overview

The proposed accelerator design targets execution time and energy reduction for efficient sparse training. This is accomplished by the following two novel features.

1) RP-CSR and RP-BF formats are developed for unified on-chip SRAM design for non-zero elements and indices to solve the SRAM underutilization issue in existing accelerator designs. Also, the row-partitioned format combines CSR and BF to achieve a better compression rate under different matrix sparsity for lower memory traffic.
2) A two-level load-balancing technique is developed to distribute the computation workload evenly among PEs and solve the workload imbalance issue in existing accelerator designs. The proposed two-level workload balancing technique includes a coarse-grain and a fine-grain load balancing mechanism. The coarse-grain load balancing mechanism is implemented in the global controller for an even distribution of the computation workload. A fine-grain load balancing mechanism is implemented in the PEs to dynamically balance workloads during the execution of a convolution operation (i.e., Step 3 of a convolution operation, as discussed in Section II-B).

B. Mathematical Models

1) Compression Format Analysis: Since the size of the compressed matrix directly impacts memory traffic, the analysis of the relationship between compression rate and sparsity is critical to the efficiency of the dual matrix compression technique. Eq. (5) illustrates the total number of bits for a CSR format with a given sparsity $s \in [0, 1]$ and height (Y) / width (X) of a matrix. Assuming each non-zero element is 32 bits and 8 bits for each entry in the column index and row index arrays.

$$\begin{aligned} Size_{CSR} &= Size_{non-zero} + Size_{col\_index} \\ &\quad + Size_{row\_index} \end{aligned} \tag{5}$$

$$Size_{non-zero} = 32 \times XY \times (1 - s) \tag{6}$$

$$Size_{col\_index} + Size_{row\_index} = 8XY \times (1 - s) + 8Y \tag{7}$$

Eq. (8) illustrates the total number of bits for a BF.

$$Size_{BF} = Size_{non-zero} + Size_{bitmap} \tag{8}$$

$$Size_{bitmap} = XY \tag{9}$$

From Eqs. (6), (7), and (9), the ratios of the non-zero element size (Eq. (6)) to index size (Eqs. (7) and (9)) for both formats are based on matrix sparsity. Therefore, a unified on-chip SRAM design, which stores both non-zero elements and indices, is preferred because the matrices are not limited by either the size of the indices or the size of the non-zero elements. With a unified on-chip SRAM, the accelerator is able to achieve higher SRAM utilization and lower off-chip memory access.

From Eqs. (5) and (8), the size of a compressed sparse matrix depends on the dimension of the matrix and sparsity.

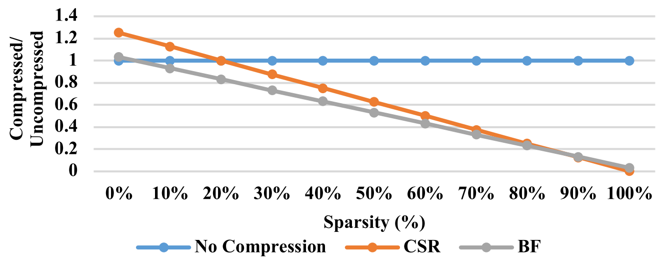


Fig. 3. Matrix size reductions under different sparsity for CSR and BF.

To establish which compression format achieves a smaller size under a given sparsity, the equation ($Size_{CSR} = Size_{BF}$) is solved to find the threshold sparsity. Eq. (10) shows the solution for this equation.

$$S_{threshold} = 0.875 + \frac{1}{X} \approx 88\% \tag{10}$$

As the width of a matrix (X) is very large during training, then the threshold for sparsity is approximated to 88%. Fig. 3 shows the matrix size reduction for CSR and BF of a matrix with 500 widths and 500 heights under different sparsity. When the matrix sparsity is less than 88%, BF is smaller in matrix size.

Eq. (10) also implies that the threshold is not affected by the height (Y) of the matrix. This indicates that if the matrix is row-partitioned, each row can determine its own compression format by comparing the sparsity of the row with the threshold, specifically when the sparsity of a row is lower than 88% ($S_{row} \leq S_{threshold}$), the RP-BF format is selected, otherwise ($S_{row} > S_{threshold}$) the RP-CSR format is selected for higher compression rate. Since the location of zeros in all the matrices is dynamically changing during sparse training, this technique further increases the compression rate.

2) Computation Workload Estimation: To distribute workloads evenly among multiple PEs before the execution of the convolution operation, the estimate of the non-zero multiplication workload is critical. A simple solution for estimating the computation is to process the indices with a convolution operation before loading the PEs. However, this requires additional on-chip SRAM space to store all indices for both matrices (e.g., the accelerator design in [15]). Since on-chip SRAM size is limited, storing all indices is expensive for workload estimate purposes.

For a quick estimate of the non-zero multiplication workload before computation, the proposed technique estimates the amount of non-zero multiplication for the matrix convolution operation by using a probability model. In a convolution operation, the probability of a non-zero multiplication is found from the elements from both the input matrix (i.e., A^{l-1} in Eq. (1) and (3), E^{l-1} in Eq. (2)) and the kernel matrix (i.e., W^l in Eq. (1), $R(W^l)$ in Eq. (2), E^l in Eq. (3) when they are non-zero. Eq. (11) illustrates the probability of getting a non-zero element in a kernel matrix during a convolution operation.

$$P_{non-zero\ Ele.in\ kernel} = \frac{Number\ of\ Non-zero\ Elements}{Number\ of\ Elements} \tag{11}$$

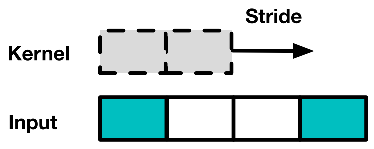


Fig. 4. $P_{non-zero\ in\ I}$ calculation example. The white box in the input matrix indicates the matrix value is zero.

Eq. (12) illustrates the probability of getting a non-zero element in an input matrix during a convolution operation.

$$P_{non-zero\ in\ I} = \frac{\sum_{window=0}^{n} Number\ of\ Non-zero\ Elements}{\sum_{window=0}^{n} Number\ of\ Elements} \tag{12}$$

For each output of a convolution operation, the kernel operates as a moving window and selects a set of elements in the input matrix according to the stride. Since the stride may be smaller than the size of the kernel, overlapping in inputs during convolution operation is common; thus, the number of elements and the number of non-zero elements in all windows must be added to account for the additional multiplications needed due to the overlapping in the inputs. Fig. 4 shows a simple example to show the process of calculating $P_{non-zero\ Ele.\ in\ I}$ and how the proposed model deals with overlapping in the convolution operation; so when the stride is 1, $P_{non-zero\ Ele.in\ I} = (1+0+1)/(2+2+2) = 0.33$ and when the stride is 2, $P_{non-zeroEle.in\ I} = (1+1)/(2+2) = 0.5$.

Since the kernel and input matrices are independent during the convolution operation, Eq. (13) illustrates the probability for non-zero multiplication during the convolution operation.

$$P_{non-zero\ Multi.} = P_{non-zero\ Ele.in\ kernel} \cdot P_{non-zero\ Ele.in\ I} \tag{13}$$

Eq. (13) only needs the number of non-zero elements in the kernel and the inputs, the stride, and the size of each matrix to estimate the computation workloads of a convolution operation. Thus, storing the indices is not required for the proposed mechanism while providing an accurate estimate for the coarse-grain load-balancing process.

C. Row-Partitioned Compression Format

According to Section IV-B.1, the dual matrix compression technique achieves a higher compression rate across all sparsity levels compared to SCR or BF compression formats. To implement the proposed technique, seamlessly switching between the two data formats is a major challenge. The ICU needs to compute the indices from both formats efficiently during the matrix convolution operation. By analyzing the CSR, it can be seen that it is naturally partitioned by row according to the row index array. During the matrix convolution operation, the indices for the output matrix can be easily computed with a row-partitioned workflow. The indices of BF are not partitioned; therefore, a row-partitioned workflow is not applicable to the conventional BF, and to calculate the indices for mixed format (e.g., BF $*$ CSR), BF needs to be row partitioned to match the data format of CSR for a row-partitioned workflow.

Fig. 5 shows the proposed row-partitioned CSR (RP-CSR) and row-partitioned BF (RP-BF) with an example that is based on the same sparse matrix shown in Fig. 2. Compared to the existing BF (Fig. 2), RP-BF puts the bitmap at the beginning of each row with non-zero elements. Compared to the existing CSR (Fig. 2), RP-CSR places the row index at the beginning of each row, then the column indices, followed by the non-zero elements. The proposed row-partitioned compression format provides the following benefits compared to conventional compression formats (i.e., CSR or BF).

1) The row-partitioned compression formats only change the sequence of the data, and it does not require additional memory.
2) The row-partitioned compression formats allow a unified on-chip SRAM design, as the indices and non-zero data are fused together to fully utilize on-chip SRAM.
3) The row-partitioned compression formats allow mixed RP-CSR and RP-BF for the same matrix because every row is independent. This feature not only allows seamless format switching, but also further increases the compression rate with less off-chip memory access.

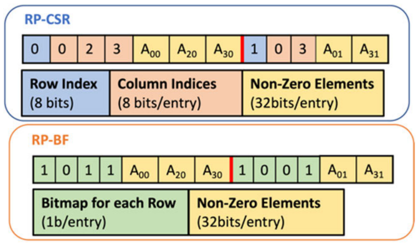

RP-CSR
0 0 2 3 A00 A20 A30 1 0 3 A01 A31
Row Index (8 bits)
Column Indices (8 bits/entry)
Non-Zero Elements (32bits/entry)
RP-BF
1 0 1 1 A00 A20 A30 1 0 0 1 A01 A31
Bitmap for each Row (1b/entry)
Non-Zero Elements (32bits/entry)

Fig. 5. Proposed RP-CSR and RP-BF formats with examples.

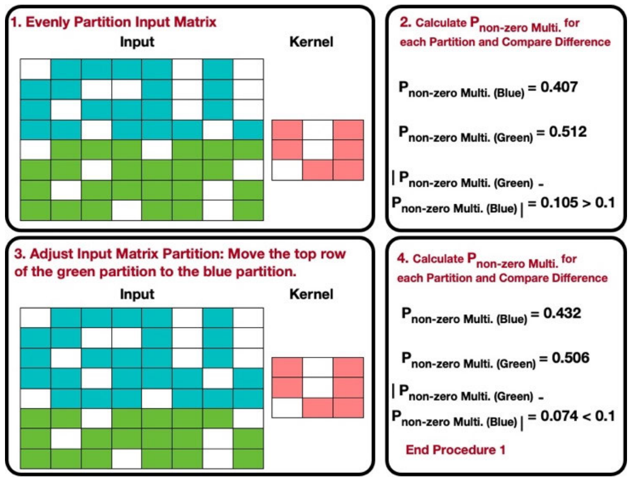

1. Evenly Partition Input Matrix
Input
Kernel
2. Calculate P non-zero Multi. for each Partition and Compare Difference
P non-zero Multi. (Blue) = 0.407
P non-zero Multi. (Green) = 0.512
| P non-zero Multi. (Green) - P non-zero Multi. (Blue) | = 0.105 > 0.1
3. Adjust Input Matrix Partition: Move the top row of the green partition to the blue partition.
Input
Kernel
4. Calculate P non-zero Multi. for each Partition and Compare Difference
P non-zero Multi. (Blue) = 0.432
P non-zero Multi. (Green) = 0.506
| P non-zero Multi. (Green) - P non-zero Multi. (Blue) | = 0.074 < 0.1
End Procedure 1

Fig. 6. Simple example for coarse-grain load balancing with two PE array partitions. The white box in both the input and kernel matrices indicates the matrix value is zero. The stride is 1 for the convolution operation.

D. Two-Level Workload Balancing Technique

Based on the row-partitioned compression format, a two-level workload balancing technique is developed to distribute the computation workload between PEs evenly during sparse training. The proposed workload balancing technique includes two parts, namely, coarse-grain and fine-grain load balancing mechanisms. The coarse-grain load balancing mechanism tries to achieve an even workload distribution by changing the mapping of the matrices according to the computation workload estimation. The fine-grain load balancing mechanism further reduces the workload difference between PEs with a workload stealing mechanism.

1) Coarse-Grain Load Balancing Mechanism: The goal of the coarse-grain load-balancing mechanism is to balance the workload based on the workload estimation described in Eq. (13). As an accelerator contains many PEs and the coarse-grain load-balancing process must be completed before distributing data to PE, a reduction in the complexity of the mechanism is critical. Thus, the proposed design divides the PE array into multiple partitions; the goal of the coarse-grain load balancing mechanism is to divide the input matrix based on the estimated workload for an even computation workload between PE partitions. Then, each PE partition is loaded with a subset of the input matrix and the kernel for the convolution operation.

Procedure 1 describes the process for the coarse-grain load balancing mechanism with the proposed row-partition compression format. The procedure uses the basic operation of balancing the probability of non-zero multiplications between two partitions. Fig. 6 shows an example of the proposed coarse-grain load balancing process, assuming the accelerator has two PE partitions. The procedure first evenly partitions the input matrices according to the total number of PE partitions in the accelerator. Each matrix partition is mapped to the corresponding PE partition. Procedure 1 shows a total of $n + 1$ partitions for the input matrix. The example in Fig. 6 assumes that the accelerator contains two PE array partitions; thus, the input matrix is evenly partitioned into two parts (i.e., the blue and green partitions).

Procedure 1 Coarse-Grain Load Balancing

```
1  Evenly divide the input matrix based on # of PE partitions.
2  While (1)
3    Calculate $P_{\text{non-zero Multi.}}$ for matrix partition [0, n]
4    If $|P^{i}_{non-zero\ Multi.} - P^{i+1}_{non-zeroMulti.}| \leq 0.1$
5      Break
6    For (i = 0; i < n; i++ )
7      If $(P^{i}_{non-zero\ Multi.} > P^{i+1}_{non-zero\ Multi.})$
8        Move one row from partition $i$ to $i+1$.
9      Else
10       Move one row from partition $i+1$ to $i$.
11 End Procedure
```

Then, $P_{\text{non-zero Multi.}}$ is calculated for each partition. Step 2 in Fig. 6 shows the calculation results for both the blue and green partitions with the assumption that the stride is 1. Then, the procedure calculates the absolute value of the differences between the $P_{\text{non-zero Multi}}$ of two consecutive partitions (i.e., $|P^{i}_{non-zero\ Multi.} - P^{i+1}_{non-zero\ Multi.}|$ in procedure 1). If the value is larger than 0.1, the elements in the two matrix partitions are adjusted based on the $P_{\text{non-zero Multi}}$.

TABLE I
COMPUTATION AND COMMUNICATION COST COMPARISON FOR FINE-GRAIN LOAD BALANCING

	Multiplication Time		**Communication Time**	
	Float	**Int**	**Same Partition**	**Different Partition**
Neighbor PE	10 Cycles /Batch	7 Cycles / Batch	8 Cycles	16 Cycles
Local PE	10 Cycles /Operation	7 Cycles / Operation	0 Cycles	0 Cycles

If $P^{i}_{non-zero\ Multi.} > P^{i+1}_{non-zero\ Multi.}$, one row is moved from partition i to $i+1$. Otherwise, one row is moved from partition $i+1$ to i. Specifically, only the row adjacent to the partition is moved when adjusting the matrix partition. For example, in Fig. 6, step 3, the top row of the green partition is moved to the blue partition. Then $\mathrm{P}_{\mathrm{non-zero\ Multi.}}$ is recalculated after adjusting the matrix partition. This process continues until the difference between all $\mathrm{P}_{\mathrm{non-zero\ Multi.}}$ for matrix partitions i and $i+1$ is less than 0.1. Step 4 in Fig. 6 shows the difference between the two $\mathrm{P}_{\mathrm{non-zero\ Multi.}}$ is less than 0.1. Thus, the procedure completes, and the current mapping is selected when loading the PEs in the accelerator. There are two reasons for choosing 0.1 as the threshold when designing procedure 1.

1) The threshold is directly related to the number of times that the FOR loop executes. Since this procedure is executed with the memory store (i.e., Step 4 in the convolution operation of the previous layer), the matrix partition of the current layer must be decided prior to loading the current layer (i.e., Step 2 in the convolution operation). Relaxing the requirement on the computation load difference reduces the time needed for the coarse-grain load balancing and ensures the procedure is completed prior to loading the PE.
2) The fine-grain load balancing mechanism further balances the workload during the execution of the convolution operation (i.e., Step 3 in the convolution operation). Fine-grain load balancing is capable of distributing the workload across the partition, and then the computation workload difference of 0.1 is handled at the next level.

2) Fine-Grain Load Balancing Mechanism: The goal of the fine-grain load balancing mechanism is to reduce the difference in workload between PEs during the convolution computation. It is achieved by Procedure 2 with a workload-stealing mechanism in PE. Each PE first finishes processing the local workload; then, the PE checks the neighbor's workload and selects the neighbor PE with the largest workload. If the neighbor shares the same set of inputs and kernel, the PE acquires more than two multiplications for the floating-point values (float) or more than three multiplications for integer values (int) from the neighbor PE. If the neighbor does not share the same set of inputs and kernel, the PE acquires more than three multiplications for floating-point value (float) or more than four multiplications for integer value (int) from the neighbor PE.

The number of multiplications that need to be acquired from the neighbor is based on the computation and communication cost during the convolution operation. Table I shows the comparison of computation and communication costs from a workload-balancing perspective. Based on the parameters provided in the SMAUG simulator [14], the cycles needed for multiplication operation for floating-point and integer values are ten cycles and seven cycles, respectively. Since a PE has multiple multipliers to perform multiplication operations, a PE with waiting workloads indicates insufficient multipliers for the operations and longer computation time. To capture this difference in computation time, the local computation is set to 10 cycles and seven cycles for one floating-point or integer multiplication operation in Table I, assuming a local PE with insufficient available multipliers. If the computation is performed by a PE with available multipliers (i.e., a neighbor PE in Table I), it takes ten cycles and seven cycles for one batch of floating-point or integer multiplication operations. The batch size depends on the number of multipliers in a PE.

To handle irregular traffic generated by the fine-grain load balancing mechanism, the on-chip communication is accomplished with a packet-based network-on-chip (NoC) with a 4-stage router. For data to be sent from one PE to another, the packet must go through two routers (i.e., one local router and one neighbor's router). It takes eight cycles for data to traverse the NoC and reach the neighbor. Therefore, if the two PEs are in the same partition and share the same set of inputs and kernel, only the product needs to be sent back to the original PE after the computation. Assuming a PE with unprocessed multiplication workloads, each multiplication takes ten cycles for float and seven cycles for int. When a neighbor PE tries to acquire workloads, a batch of multiplications takes ten cycles for float and seven cycles for int, with an additional eight cycles to send the results back to the original PE. Thus, moving the multiplication workload is beneficial only when a neighbor PE acquires more than two float multiplications (i.e., 18 cycles for neighbor PE computation $<$ 20 cycles for the local PE computation) or three int multiplications (i.e., 15 cycles for neighbor PE computation $<$ 21 cycles for local PE computation) within the same partition. If the two PEs are not in the same partition, both the variables and products need to traverse the NoC. Thus, moving multiplication workload is beneficial only when a neighbor PE acquires more than three float multiplications (i.e., 26 cycles for neighbor PE computation $<$ 30 cycles for local PE computation) or four int multiplications (i.e., 23 cycles for neighbor PE computation $<$ 28 cycles for local PE computation).

E. Architecture Design

1) Architecture Overview: An architecture is developed to support the proposed dual matrix compression and the two-level workload balancing technique. Fig. 7 shows an overview of the proposed accelerator design; it has a global controller, a global buffer, a data distributor/collector, and multiple PEs. To support the proposed fine-grain load balancing mechanism, a network-on-chip (NoC) is implemented to connect the PEs and the data distributor/collector. To support the proposed coarse-grain load balancing mechanism and row-partitioned compression format, each PE in the accelerator calculates one row of output for a convolution operation.

Procedure 2 Workload-stealing Mechanism in PE

```
1  Finish process local workload.
2  Check all neighbor PEs' workloads.
3  Select the neighbor PE with the largest workload.
4  **If** the neighbor PE shares the same set of input and kernel
5      **If** the values are *float*
6          Acquire _TWO or MORE_ multiplication workloads.
7      **If** the values are *int*
8          Acquire _THREE or MORE_ multiplication workloads.
9  **If** the neighbor PE does not share the same set of input and
   kernel
10     **If** the values are *float*
11         Acquire _THREE or MORE_ multiplication workloads.
12     **If** the values are *int*
13         Acquire _FOUR or MORE_ multiplication workloads.
14 Repeat until all the workload is completed.
```

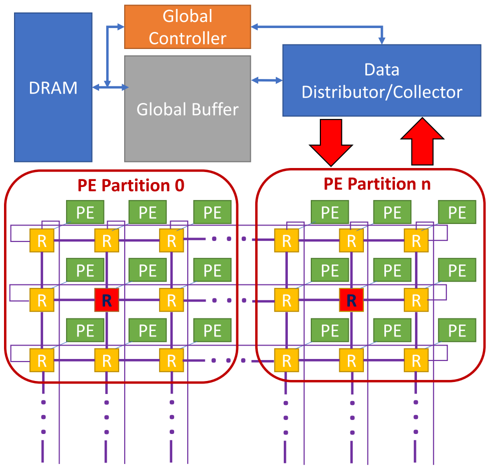


Fig. 7. Overview of the proposed accelerator design. The center router (highlighted in red) of each PE partition is connected to the data distributor/collector.

2) NoC Design: The NoC in the proposed accelerator serves two purposes. When the data distributor/collector loads the PEs or gathers results from the PEs, the NoC is responsible for connecting the PEs and the data distributor/collector. During the PE calculation process, the NoC is responsible for handling the communication for the load balancing mechanism. As shown in Fig. 7, the NoC is implemented with the network links, which are highlighted in purple, and 4-stage routers (i.e., R in Fig. 7). Each PE is connected to a router through a network interface, which is shown in Fig. 8.

For the fine-grain load balancing mechanism, the proposed NoC uses a 2D-torus topology to ensure that all PEs have the same number of neighbors. Compared to other topologies (e.g., bus, tree, or 2D-mesh), this design allows all PEs to have access to the same number of neighbors when applying the fine-grain load balancing mechanism, resulting in better workload balancing during computation. Since the coarse-grain load balancing mechanism allows a maximum of 10% in workload difference when loading the PEs, the network in each partition should be small enough to allow the fine-grain load balancing mechanism to further reduce the difference in workload. Thus, as shown in Fig. 7, the proposed design divides the PE array into multiple partitions; each partition contains six PEs. To load all PEs efficiently, the center router of each partition is connected to the data distributor for fast distribution and gathering of data before and after the convolution operation during sparse training.

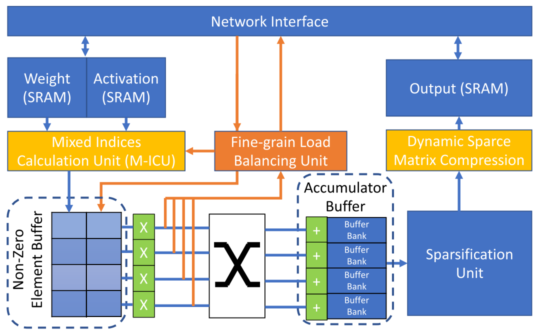


Fig. 8. Proposed PE design to support the dual matrix compression and fine-grain load balancing techniques.

3) Global Controller: The global controller is responsible for managing the whole convolution operation described in Section II-B. The global controller monitors the traffic between DRAM and the global buffer and controls the data distributor/collector to load PEs or gather results from PEs. Thus, the coarse-grain load balancing mechanism is implemented in the global controller. After the PE array completes its computation, the global controller monitors the results by registering the number of non-zeros in each row. When the global controller loads the global buffer with the kernel for the next layer, Procedure 1 is executed to partition and map the input matrix to PE partitions according to the estimated workload. Then, the partitioned inputs and kernel matrices are loaded to the SRAM in the PEs for calculation.

4) PE Design: Fig. 8 shows the proposed PE design. In this figure, the data path for processing the sparse matrices is in blue. The data path for the fine-grain load balancing mechanism is highlighted in orange. Different from existing accelerators, the PE contains two unified input SRAMs and one unified output SRAM, which is directly connected to the network interface of the NoC. The two input SRAMs store the input and kernel matrices. During training, these SRAMs store the matrices (i.e., E, $R(W)$) during BP and the matrices (i.e., A, E) during WG. The mixed index calculation unit (M-ICU) processes indices from both RP-CSR and RP-BF, as well as loads the non-zero element buffers as per the indices of the input matrices. The non-zero element buffer contains two columns. Each multiplier is connected to a row of non-zero element buffers and multiplies the elements in two columns. After the multiplication, a crossbar sends the products to the corresponding accumulator buffer to compute the non-zero output of the convolution operation. The sparsification unit has the ReLU, pruning, and random drop functions. The dynamic sparse matrix compression unit checks the sparsity of each row and selects the appropriate compression format for each row in the output matrix.

5) Fine-Grain Load Balancing Unit: The fine-grain load-balancing unit is implemented based on Procedure 2, described

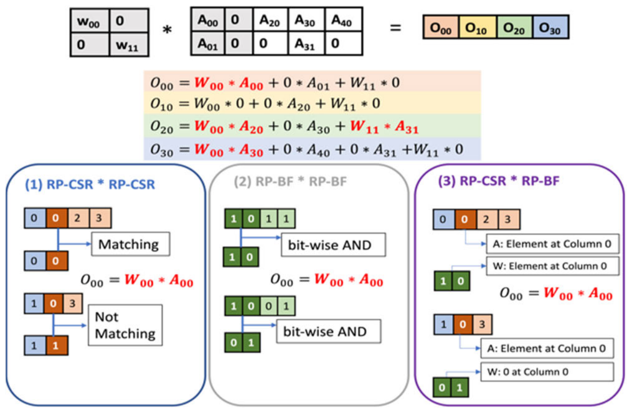


Fig. 9. Index convolution on the two compression formats.

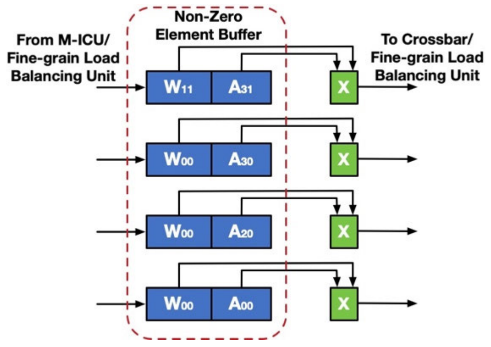

Non-Zero Element Buffer
From M-ICU/ Fine-grain Load Balancing Unit
To Crossbar/ Fine-grain Load Balancing Unit
W11
A31
W00
A30
A20
A00
X

Fig. 10. Non-zero element buffer design.

in Section IV-D.2. Dedicated wires are connected to the fine-grain load-balancing unit in the neighbor PEs to monitor and acquire the workload. For clarity, these wires are not shown in Fig. 8. If the local and neighbor PE share the same set of input and kernel, the load balancing unit controls the M-ICU to acquire the required data from the local SRAM and perform the multiplication operation; then, the products are sent back to the neighbor PE via NoC. If the local and neighbor PEs do not share the same set of input and kernel, the neighbor sends the required data through the NoC. The load balancing unit loads the non-zero element buffer with the data and performs the multiplication operations. After the multiplication operations, the products are sent back to the neighbor PE via NoC.

6) Mixed Index Calculation Unit (M-ICU): Since both input matrices are partitioned in rows, the M-ICU processes the indices one row at a time. For two input rows, the M-ICU first performs the convolution on the indices to estimate the number of multiplications needed for each partial sum; then, the non-zero buffers are loaded with the pairs of values for multiplication.

There are three combinations of index format for the two input matrices. Fig. 9 shows an example of the three combinations; this example shows a convolution operation between a sparse weight matrix and a sparse activation matrix with a stride of 1.

1) RP-CSR $*$ RP-CSR: The index convolution for two RP-CSR matrices is accomplished by checking the indices for two non-zero elements. If the non-zero column index from the weight (e.g., W_{00}, column index $= 0$) equals the non-zero column index from the activation (e.g., A_{00}, index $= 0$) plus the column number of the output (e.g., O_{00}, index $= 0$), then the product of this multiplication is non-zero. The same condition also applies to the row indices. The product of this multiplication is non-zero only if both the column and row indices match.
2) RP-BF $*$ RP-BF: The index convolution for two RP-BF matrices is accomplished by the AND operation of two bitmaps. If the result is 1, the product of this multiplication is non-zero; otherwise, the product is zero.
3) RP-CSR $*$ RP-BF: The index convolution for RP-CSR and RP-BF matrices is accomplished by first checking the row and column indices in CSR. Since the indices in CSR only contain the location for non-zero elements, to find the non-zero multiplications, the unit only needs to check the corresponding bit in the BF bitmap. If the bit is 1, the product of this multiplication is non-zero; otherwise, the product is zero.

7) Load Non-Zero Element Buffer: Fig. 10 shows the detailed connection between non-zero buffers and the multipliers with an example. After the M-ICU determines the non-zero multiplication for a row, the buffer is loaded with non-zero elements for multiplication. The M-ICU loads all non-zero element buffers in the PE; if the buffer is not filled, the non-zero multiplication for the next row is loaded into the buffer. For example, the multiplication of W_{11} and A_{31} is the non-zero multiplication for the second row. As the example has four multipliers, the last buffer is loaded with these two elements to fully utilize all multipliers in a PE.

8) Dynamic Sparse Matrix Compression: After the matrix sparsification unit, the compression unit monitors the sparsity of each row. If the sparsity of a row is less than $S_{threshold}$ (i.e., 88% according to Eq. (10)) the row is compressed in RP-BF. Otherwise, the row is compressed in RP-CSR.

V. Evaluation

A. Simulation Setup

In this section, the performance of the dual matrix compression and two-level load-balancing techniques are evaluated by using the SMAUG [14] simulator. The cycle-accurate SMAUG simulation model is modified to implement the proposed accelerator design, SCNN [13], [16], SIGMA [17] and SparseTrain [12]. Table II shows the settings for the SMAUG simulator. PyTorch is used with the SMAUG simulator to control the entire sparse training process. The proposed accelerator is compared against SCNN (CSR with partitioned SRAM), SIGMA (BF with partitioned SRAM), and SparseTrain (CSR with unified SRAM) on off-chip memory traffic, load balancing, execution time, energy consumption, and area. All accelerators are implemented with the same number of CUs as well as the same size of SRAM, DRAM, and DRAM bandwidth. The SCNN and SIGMA reserve 20% of each on-chip SRAM for the indices. Memory traffic, execution time, and dynamic power consumption are captured during the execution of the sparse training workloads on SMAUG. The accelerators are also synthesized using

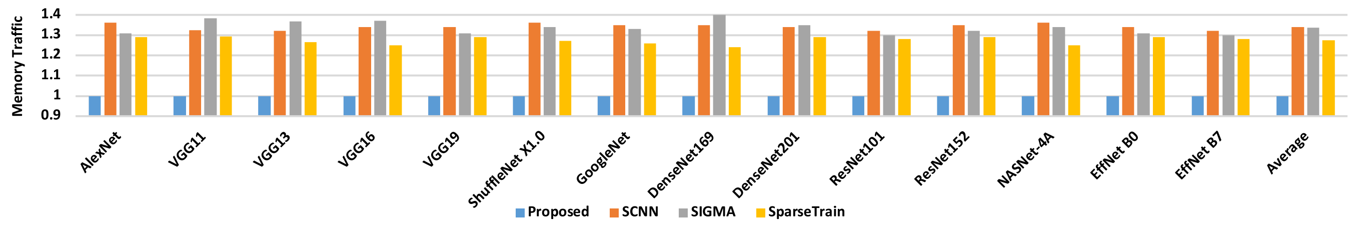


Fig. 11. Off-chip memory traffic during sparse training. Results are normalized to the proposed design.

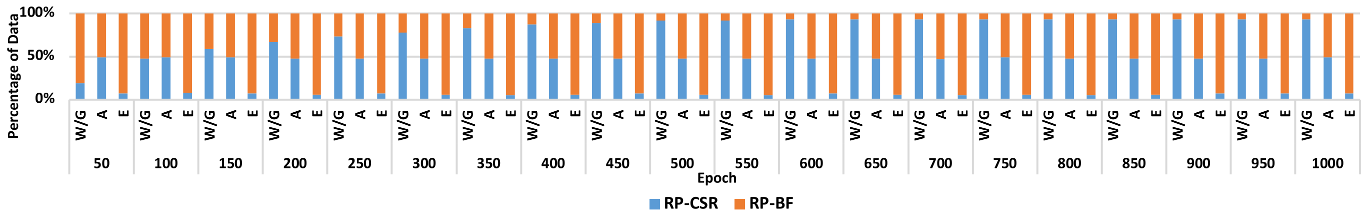


Fig. 12. Average proportion of RP-CSR and RP-BF for W/A/G/E matrices during training for the proposed design.

TABLE II
SIMULATION SETUP

PE Parameter	**Value**
Multiplier Width	32 bits
Accumulator Width	32 bits
Number of Multiplier	16
Accumulator Banks	32
SRAM Size/PE	3 MB
Accelerator Parameters	**Value**
Number of PEs	64
Number of PE Partitions	3*3
NoC Topology	9*9 2D Torus
Global Buffer Size	256 MB
DRAM Bandwidth	1024 GB/s
DRAM Size	16 GB

TABLE III
CNN TRAINING WORKLOADS

CNN Models	**Sparse Training Accuracy**	**Conventional Training Accuracy**
AlexNet	56.55%	56.57%
VGG11	69.02%	69.04%
VGG13	69.93%	69.95%
VGG16	71.59%	71.61%
VGG19	72.38%	72.39%
ShuffleNet X1.0	67.60%	67.63%
GoogleNet	69.78%	69.79%
DenseNet169	77.20%	77.23%
DenseNet201	77.65%	77.67%
ResNet101	77.37%	77.37%
ResNet152	78.31%	78.32%
NASNet-4A	74.00%	74.01%
EfficientNet_B0	76.30%	76.32%
EfficientNet_B7	84.40%	84.43%

Synopsys Design Compiler with TSMC 16nm and clock speed of 1GHz to obtain area and static power consumptions.

Table III shows the executed sparse training workloads with their classification accuracy after training. A 95% pruning rate is achieved during the sparse training process for all models. ImageNet [1] is used for training and accuracy testing. The batch size for the sparse training of all the models is 128, and all the models are trained with 1000 epochs, which includes 500 epochs for pruning and 500 epochs for fine-tuning to reach a similar accuracy as conventional training. Since the same sparse training workloads are executed on the proposed design, SCNN, SIGMA, and SparseTrain, exactly the same sparse training accuracy is achieved at the end of training. To illustrate the potential accuracy loss of sparse training, the accuracy of the models with conventional training is also shown in Table III.

B. Off-Chip Memory Traffic

Fig. 11 shows the off-chip memory traffic normalized to the proposed design. The proposed dual matrix compression technique reduces memory traffic by 32% on average compared to the existing accelerators. Compared to SCNN and SparseTrain, the proposed technique reduces memory traffic by 34 % and 27%, respectively. Compared to SIGMA, the proposed design achieves a 34% reduction in memory traffic. This achievement is mainly due to the dynamic utilization of both the RP-CSR and RP-BF during the training. Fig 12 shows the average proportion of RP-CSR and PR-BF for W/A/G/E matrices during the training of the models shown in Table II. The proposed accelerator seamlessly switches between RP-CSR and RP-BF to take advantage of both data formats with lower memory traffic under different matrix sparsity when executing sparse training workloads.

Fig. 13 shows the contribution of memory traffic savings of compressing W, A, E, and G with the dual matrix compression technique compared to SparseTrain. SparseTrain is implemented with a unified SRAM and CSR; thus, the memory traffic saving is directly related to the sparsity of the matrix, as shown in Fig. 14. The lower sparsity results in higher savings compared to the proposed design due to the utilization of RP-BF under low sparsity. Compared to SparseTrain, the dual matrix compression technique achieves an average of 5% reduction in memory traffic for both W and G. As matrices A and E are less sparse compared to W and G, A and E contribute to a 6% and 11% memory

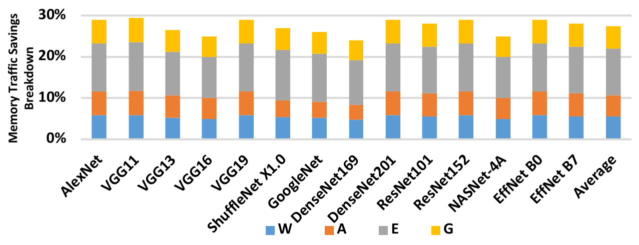


Fig. 13. Memory traffic savings breakdown compared to SparseTrain.

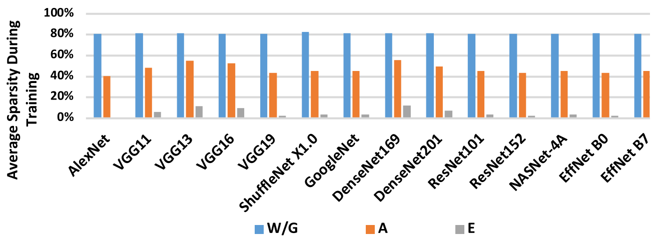


Fig. 14. Average matrix sparsity during training.

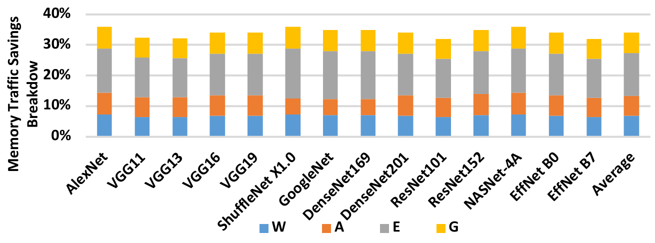


Fig. 15. Memory traffic savings breakdown compared to SCNN.

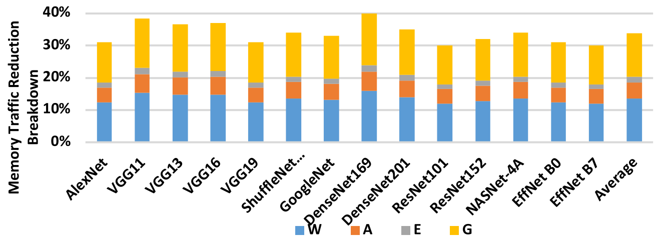


Fig. 16. Memory traffic savings breakdown compared to SIGMA.

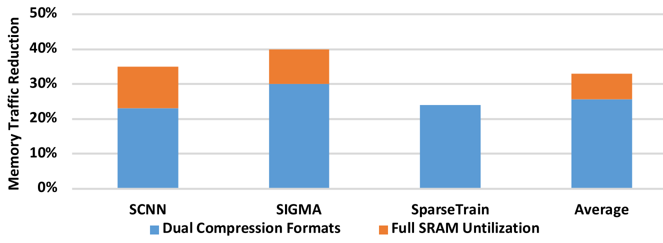


Fig. 17. Off-chip memory traffic reduction breakdown in existing accelerators normalized to the proposed design for Densnet169.

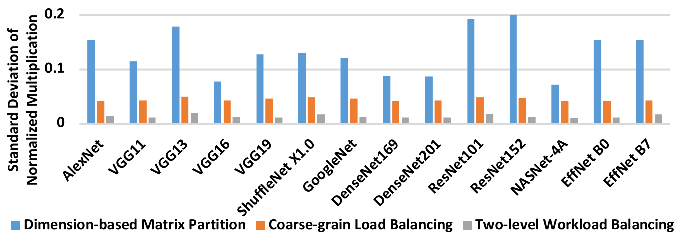


Fig. 18. Standard deviation of normalized multiplication workloads for all the PEs.

traffic reduction on average, respectively. Figs. 15 and 16 illustrate the memory traffic savings with the dual matrix compression technique compared to SCNN and SIGMA. Since both SCNN and SIGMA have partitioned SRAM, the memory traffic saving with SCNN is larger compared to the savings with SparseTrain. This is mainly due to the underutilization of the SRAM as the sparsity level changes during training. Compared to SCNN, the dual matrix compression technique achieves an average of 7% reduction in memory traffic for both W and G. A and E contribute to 7% and 14% memory traffic reduction on average, respectively.

Compared to SIGMA, the dual matrix compression technique achieves a better reduction for the matrices with higher sparsity due to the utilization of RP-CSR under high sparsity. Moreover, since SIGMA only supports matrix multiplication, an Im2Col operation is required to extend the input and kernel matrices. The extension of the matrices further increases memory traffic for SIGMA. As shown in Fig. 16, the dual matrix compression technique achieves an average of 14% reduction in memory traffic for both W and G; A and E contribute to 5% and 2% memory traffic reduction on average, respectively.

Fig. 17 shows the breakdown of the off-chip memory traffic reduction for Densnet169. The average memory traffic reduction is 33% compared to SCNN, SIGMA, and SparseTrain. Compared to SCNN and SIGMA, an average of 7% reduction is achieved due to the unification of indices and non-zero data that fully utilize the on-chip SRAM. The combined effects of both the row-partitioned matrix compression format and unified SRAM result in a significant reduction of off-chip memory access. SparseTrain, which has unified SRAM, can fully utilize the on-chip SRAM; therefore, the memory traffic reduction for SparseTrain (i.e., a 24% reduction) is due to the higher compression rate achieved by the proposed dual matrix compression technique.

C. Load Balancing

Fig. 18 shows the standard deviation of normalized multiplication workloads for all the PEs during sparse training. Three workload balancing techniques, including dimension-based matrix partition, the proposed coarse-grain load-balancing, and the proposed two-level load-balancing, are evaluated. SCNN, SIGMA, and SparseTrain use dimension-based matrix partition, which evenly divides the matrix based on the number of PEs in the accelerator. The results in Fig. 18 are generated based on the following steps. First, the multiplication workloads for each PE are observed and normalized to the average number of multiplications. Then, the standard deviation of normalized multiplication workloads is calculated.

Since only the multiplication with a non-zero product is calculated during execution, the workload is not evenly distributed for all the PEs. In Fig. 18, the results closer to

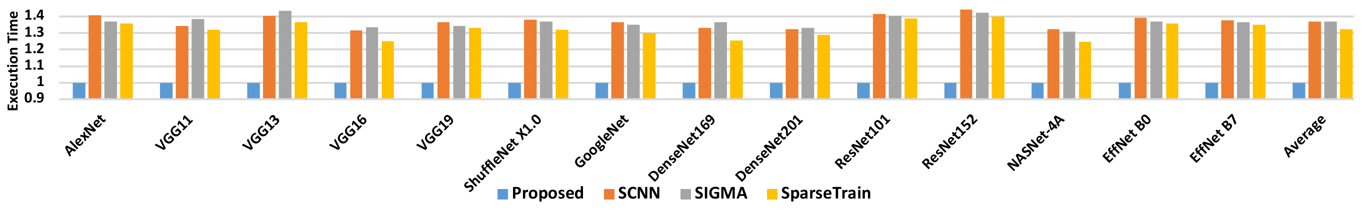


Fig. 19. Execution time of sparse training. Results are normalized to the proposed design.

0 indicate a more even workload distribution. Compared to coarse-grain and two-level workload balancing techniques, simply dividing the matrices evenly based on the number of PEs shows a significant deviation in multiplication workloads. The largest deviation is for ResNet152, which is 0.19. This indicates the variation of multiplication workloads for all the PEs is significant. After applying the coarse-grain load balancing mechanism, the difference in workload for any two neighbor partitions is less than 10%, as accomplished by the proposed procedure. Thus, it is observed that the average standard deviation for coarse-grain load balancing is 0.045. After applying both the coarse-grain and fine-grain load balancing mechanisms, the difference in workload for any two PEs is further reduced. Therefore, the average deviation for the two-level workload balancing technique is 0.013. For ResNet152, the standard deviation of normalized multiplication workloads is reduced to 0.012. This indicates that the proposed technique achieves a more balanced computation compared to existing accelerators. This increases the utilization of computing resources and further reduces the execution time.

TABLE IV
AREA AND STATIC POWER EVALUATION

	SCNN	SIGMA	SparseTrain	Proposed
Area (mm^2)	19.62	22.55	19.78	20.16
Static Power (W)	0.289	0.332	0.291	0.295

TABLE V
AREA BREAKDOWN FOR PROPOSED ACCELERATOR

Accelerator Components	**Area (mm^2)**
Global Controller	1.28
NoC + Data Distributor/Collector	9.41
PEs	9.47
Accelerator Total	**20.16**
PE Components	**Area (mm^2)**
Multiplier	0.009
M-ICU	0.002
Fine-Grain Load Balancing Unit	0.001
Dynamic Sparse Matrix Compression	0.004
Accumulator Buffer	0.045
Others	0.024
PE Total	**0.148**

D. Execution Time

Off-chip memory access and irregular computation patterns are the two main causes of a long execution time during sparse training. Since the proposed technique significantly reduces memory traffic, a reduction in execution time is expected for the sparse training workloads. Moreover, the proposed load-balancing technique further reduces the execution time by evenly distributing the workloads among all the PEs in the accelerator. Fig 19 shows the execution time reduction normalized to the proposed design. The proposed accelerator reduces execution time by 34% on average compared to the existing accelerators. Specifically, compared to SCNN, SIGMA, and SparseTrain, the proposed technique reduces execution time by 37%, 36%, and 32% on average, respectively. Compared to the reduction in memory traffic, the proposed accelerator achieves significantly more reduction in execution time. This is mainly due to the implementation of the proposed workload balancing technique, which increases parallelism for multiplication operations during training. The combined effect of both memory traffic reduction and balanced workload results in significant savings in execution time. For example, the highest reduction in training time is achieved by ResNet152. The proposed accelerator design reduces execution time by 44%, 42%, and 40% compared to SCNN, SIGMA, and SparseTrain, respectively. This significant achievement is achieved by an average of 32% reduction in memory traffic and 0.18 reduction in the standard deviation of multiplication workloads for all the PEs compared to existing accelerators.

E. Energy Consumption

Fig. 20 shows the evaluation results for the energy consumption of the proposed accelerator, SCNN, and SIGMA. All results are normalized to the proposed design. The energy is the product of execution time and power dissipation. Power dissipation includes two parts: static power and dynamic power. The execution time and dynamic power are obtained from the SMAUG simulator, and static power is obtained from the Synopsys Design Compiler.

The proposed accelerator reduces energy consumption by 24% on average compared to the existing accelerators. Specifically, compared to SCNN, SIGMA, and SparseTrain, the proposed technique reduces energy consumption by 25%, 27%, and 18% on average, respectively. Compared to the reduction in execution time, the energy reduction is less; this is mostly due to the extra power consumed by packet switching NoC in the proposed accelerator for workload balancing and communication purposes. The SCNN and SparseTrain have simplified on-chip networks with the predetermined data flow; however, fixed data flow in existing designs results in an unbalanced workload distribution during the sparse training process, which incurs additional energy consumption.

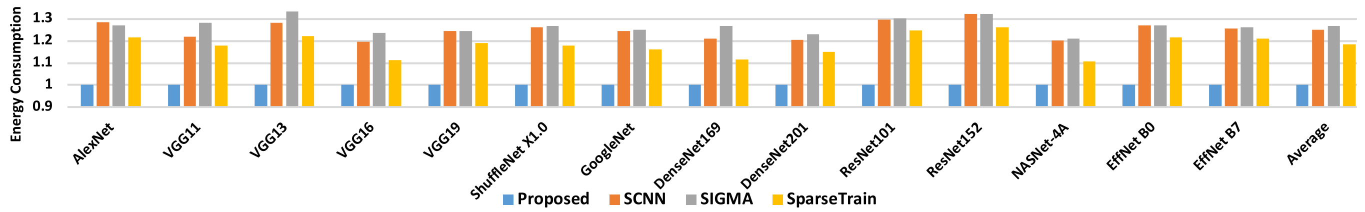


Fig. 20. Energy consumption of sparse training. Results are normalized to the proposed design.

F. Area and Static Power Analysis

Table IV summarizes the area and static power consumption for different implementations with the configuration given in Table II. Since all the accelerators are configured with the same amount of SRAM during the evaluation, the results shown in Table IV exclude the area and static power consumed by the SRAM.

The proposed design occupies less area compared to SIGMA despite adding support for dual matrix compression. This is mainly due to the implementation of flexible interconnects that consume a large amount of on-chip area for zero-product multiplication avoidance in SIGMA. Compared to SIGMA, the proposed accelerator reduces area by 10%. Compared to SCNN and SparseTrain, the proposed accelerator increases the area by 3% and 2%, respectively. This is mainly due to the extra hardware needed for two-level load-balancing and dual matrix compression techniques. The added hardware also increases static power consumption. According to the report generated by Synopsys Design Compiler, the proposed global controller with coarse grain load balancing method consumes 0.02 W static power. Each PE consumes 2.76 mW static power in the proposed accelerator, of which 2% is consumed by M-ICU and dynamic sparse matrix compression, and 0.52% is consumed by the fine-gran load balancing unit.

However, considering the significant reduction in execution time and memory traffic, the increase in area is minimal when implementing the proposed accelerator compared to SCNN and SparseTrain.

VI. Conclusion

In this paper, we have proposed dual matrix compression and two-level workload balancing techniques for efficient sparse CNN training. The dual matrix compression technique introduces a row-partitioned data format (i.e., RP-CSR and RP-BF) for unified on-chip SRAM of both non-zero elements and indices with lower off-chip memory traffic. Based on the proposed dual matrix compression technique, the two-level workload balancing technique, which includes coarse-grain and fine-grain load balancing mechanisms, has been developed. The coarse-grain load balancing mechanism adjusts each matrix partition by estimating the computation required for the partition with a probability model. The fine-grain load balancing mechanism utilizes a workload-stealing mechanism to further reduce the difference in workload between PEs. To support both dual matrix compression and workload balancing techniques, an accelerator has been developed and implemented. Our detailed evaluation shows that the proposed accelerator reduces the execution time by 34% and the energy consumption by 24% compared to existing sparse training accelerators.

Acknowledgment

The authors would like to thank the anonymous reviewers for their invaluable feedback.

References

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, May 2015, Art. no. 7553.

[3] M. Asadikouhanjani, H. Zhang, L. Gopalakrishnan, H.-J. Lee, and S.-B. Ko, "A real-time architecture for pruning the effectual computations in deep neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 5, pp. 2030–2041, May 2021.

[4] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *J. Mach. Learn. Res.*, vol. 22, no. 1, pp. 1–124, Jan. 2021.

[5] 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*.

[6] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, vol. 1, Cambridge, MA, USA, Dec. 2015, pp. 1135–1143.

[7] B. N. G. Koneru, N. Chandrachoodan, and V. Vasudevan, "A smoothed LASSO-based DNN sparsification technique," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 10, pp. 4287–4298, Oct. 2021.

[8] 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.

[9] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus, "Regularization of neural networks using DropConnect," in *Proc. 30th Int. Conf. Mach. Learn.*, May 2013, pp. 1058–1066.

[10] M. A. Raihan and T. Aamodt, "Sparse weight activation training," in *Proc. Adv. Neural Inf. Process. Syst.* Curran Associates, 2020, pp. 15625–15638. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/hash/b44182379bf9fae976e6ae5996e13cd8-Abstract.html

[11] S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, "Hardware acceleration of sparse and irregular tensor computations of ML models: A survey and insights," *Proc. IEEE*, vol. 109, no. 10, pp. 1706–1752, Oct. 2021, doi: 10.1109/JPROC.2021.3098483.

[12] P. Dai et al., "SparseTrain: Exploiting dataflow sparsity for efficient convolutional neural networks training," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.

[13] J. S. Lew, Y. Liu, W. Gong, N. Goli, R. D. Evans, and T. M. Aamodt, "Anticipating and eliminating redundant computations in accelerated sparse training," in *Proc. 49th Annu. Int. Symp. Comput. Architecture*. New York, NY, USA: Association for Comput. Machinery, Jun. 2022, pp. 536–551.

[14] S. L. Xi, Y. Yao, K. Bhardwaj, P. Whatmough, G.-Y. Wei, and D. Brooks, "SMAUG: End-to-end full-stack simulation infrastructure for deep learning workloads," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, pp. 1–26, Nov. 2020.

[15] S. Kim, J. Lee, S. Kang, D. Han, W. Jo, and H.-J. Yoo, "TSUNAMI: Triple sparsity-aware ultra energy-efficient neural network training accelerator with multi-modal iterative pruning," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 69, no. 4, pp. 1494–1506, Apr. 2022.

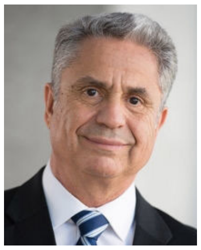
[16] A. Parashar et al., "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Architecture (ISCA)*. New York, NY, USA: Association for Comput. Machinery, Jun. 2017, pp. 27–40.

[17] 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. Architecture (HPCA)*, Feb. 2020, pp. 58–70.

[18] Y. Wang, Y. Qin, L. Liu, S. Wei, and S. Yin, "SWPU: A 126.04 TFLOPS/W edge-device sparse DNN training processor with dynamic sub-structured weight pruning," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 69, no. 10, pp. 4014–4027, Oct. 2022.

[19] E. Trommer, B. Waschneck, and A. Kumar, "DCSR: A memory-efficient sparse matrix representation for parallel neural network inference," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2021, pp. 1–9.

Yuechen Chen (Member, IEEE), received the Ph.D. degree in computer engineering from George Washington University, Washington, DC, USA, in 2024. He is currently an Assistant Professor with the Department of Computer Science and Information Technologies, Frostburg State University. His research interests include approximate computing and NoCs.



Ahmed Louri (Fellow, IEEE) received the Ph.D. degree in computer engineering from the University of Southern California, Los Angeles, CA, USA, in 1988. From 1988 to 2015, he was a Professor of electrical and computer engineering with The University of Arizona. From 2010 to 2013, he was the Program Director with the National Science Foundation's (NSF) Directorate for Computer and Information Science and Engineering. In August 2015, he joined as the David and Marilyn Karlgaard Endowed Chair Professor of electrical and computer engineering with George Washington University, Washington, DC, USA. His research interests include interconnection networks and network-on-chips for multicores and the use of machine learning techniques for energy-efficient, reliable, high-performance, and secure many-core architectures and accelerators. In 2024, he is serving on the Computer Society Publication Board Executive Committee. He is also the Chair of the Transactions Operations Committee. He was the Editor-in-Chief of IEEE TRANSACTIONS ON COMPUTERS from 2019 to 2023.

Shanshan Liu (Senior Member, IEEE) received the Ph.D. degree in microelectronics and solid-state electronics from Harbin Institute of Technology, Harbin, China, in 2018. She was a Post-Doctoral Researcher with Northeastern University, Boston, MA, USA, from 2018 to 2021, and an Assistant Professor with New Mexico State University, Las Cruces, NM, USA, from 2021 to 2023. She is currently a Professor with the University of Electronic Science and Technology of China, Chengdu, China. Her research interests include fault tolerance design in high-performance computer systems, VLSI design, dependable machine learning, stochastic computing, and error correction codes.

Fabrizio Lombardi (Life Fellow, IEEE) received the B.Sc. degree (Hons.) in electronic engineering from the University of Essex, U.K., in 1977, the master's degree in microwaves and modern optics and the Diploma degree in microwave engineering from the Microwave Research Unit, University College London, in 1978, and the Ph.D. degree from the University of London in 1982. He is currently the International Test Conference (ITC) Endowed Chair Professorship with Northeastern University, Boston, MA, USA. His research interests include bio-inspired and nano manufacturing/computing, VLSI design, testing, and fault/defect tolerance of digital systems. He was the President of the IEEE Nanotechnology Council from 2022 to 2023.