SPSA: Exploring Sparse-Packing Computation on Systolic Arrays From Scratch

Minjin Tang, Mei Wen, Jianchao Yang, Zeyu Xue, and Junzhong Shen

***Abstract*—Sparse matrix-matrix multiplication (SpMM) and Generalized SpMM (SpGEMM) are essential computational kernels in domains, such as graph analytics and scientific computation. While systolic arrays have traditionally been employed as specialized architectures for complex computing problems like matrix multiplication, they exhibit inefficiency when dealing with sparse matrices. This inefficiency arises from the unnecessary operations performed by processing elements (PEs) that contain zero-valued entries, which do not contribute to the final result. To address this issue, we propose SPSA, a framework that leverages a sparse-packing algorithm suitable for systolic arrays to accelerate sparse matrix computations. Our approach achieves significant reduction of zero-valued items and improves matrix density by packing the rows or columns of the sparse matrix. Furthermore, we have introduced for the first time a data representation format tailored to systolic arrays, called CSXD, which further enhances storage and computational efficiency. Importantly, our adaptation scheme enables acceleration benefits even with limited resources. Through sparse packing, SPSA achieved a 5.2× performance improvement compared to the dense baseline, and further reached a 6.4× enhancement via CSXD. Simultaneously, CSXD realized an average storage efficiency improvement of 15.0×. Through extensive evaluations, SPSA outperforms previous designs on CPU, GPU, and ASIC platforms. Finally, in end-to-end evaluations, SPSA achieved a performance improvement of 3.9 times across the workloads of BERT, VGG19, and ResNet50.**

***Index Terms*—Sparse computation, sparsity, systolic arrays.**

I. Introduction

MATRIX multiplication find extensive applications in diverse domains, such as scientific computation [1], [2] and large language models [3], [4]. However, performing calculations on a sparse matrix as if it were a dense matrix results in inefficient utilization of computing power and storage, as the computations involving zero values are redundant and wasteful. Exploiting the inherent sparsity of the matrix offers the opportunity to reduce computational overhead and accelerate the overall computation process.

Systolic arrays [5] offer significant potential for accelerating matrix multiplication. Notably, the Google team has leveraged systolic arrays as the core computing architecture in the design of the tensor processing unit (TPU) [6]. Furthermore, several studies [6], [7], [8], [9], [10] have demonstrated the effectiveness of systolic arrays in accelerating matrix multiplication, affirming their viability as a powerful method for enhancing computational efficiency in various applications. In systolic arrays, the processing elements (PEs) are not only responsible for executing multiplication and accumulation operations but also actively participate in cyclically transmitting data to neighboring PEs. This tightly integrated data transmission structure plays a pivotal role in enabling efficient parallel computing, reducing data transfer delays, and enhancing overall computational efficiency.

However, when accelerating sparse matrix multiplication, some approaches [11], [12], [13] decouple systolic arrays by separating multiplication and addition operations within PEs. In other words, the array is responsible only for the multiplication part, while the addition part is handled by external accumulation units, such as SIGMA [12], which completes the accumulation through an adder tree. Moreover, this loosely coupled structure breaks the coupling relationships within and between the PEs, as data within the array can be directly routed by dedicated routing without the need to pulsate horizontally or vertically through the adjacent PEs. While the loosely coupled structure offers more flexibility, it sacrifices the inherent high concurrency of tightly coupled structures that enables systolic arrays to efficiently execute regular computations. In addition, the loosely coupled structure introduces additional routing and communication network overhead.

Conversely, some approaches [14], [15], [16] maintain the tightly coupled structure of systolic arrays. They achieve support for sparse computations by initially condensing sparse matrices and then making hardware modifications. Common preprocessing methods involve compression techniques based on conflict pruning [8], [16], which, while achieving higher-matrix compression rates, compromise computational precision and accuracy. In general, enabling sparse matrix multiplication on tightly coupled systolic arrays proves to be a challenge.

Developing sparse matrix encoding formats tailored for systolic arrays that are efficient in terms of both storage and computation poses a challenge. To minimize the storage cost of sparse matrices, compression-based storage methods, which exclusively store nonzero elements, have been devised. Formats, such as COO, CSR/CSC, ELL, and so on, extract nonzero elements in different ways, catering to specific use cases. While these formats efficiently represent

Manuscript received 29 February 2024; revised 1 June 2024; accepted 23 July 2024. Date of publication 26 July 2024; date of current version 22 January 2025. This work was supported in part by the Natural Science Foundation of Hunan Province under Grant 2024JJ6470; in part by the National Natural Science Foundation of China under Grant U23A20301; and in part by NUDT Foundation under Grant ZK2023-16. This article was recommended by Associate Editor G. Ansaloni. *(Corresponding author: Mei Wen.)*

The authors are with the Key Laboratory of Advanced Microprocessor Chips and Systems, National University of Defense Technology, Changsha 410073, China (e-mail: meiwen@nudt.edu.cn).

Digital Object Identifier 10.1109/TCAD.2024.3434359

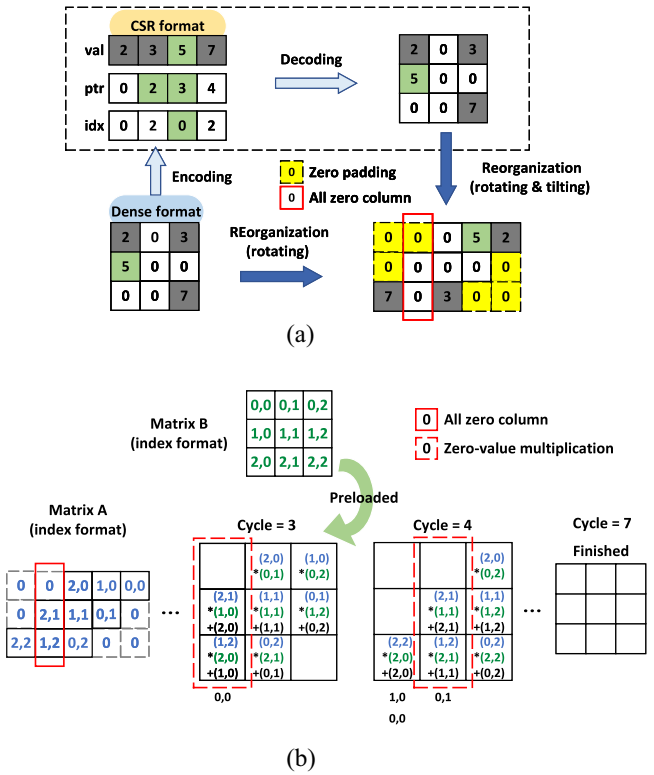


Fig. 1. (a) When a dense format matrix is encoded using the CSR format, it needs to undergo decoding and reorganization (rotation and skewing) before being mapped to systolic arrays. The reorganized matrix will exhibit new fully zeroed columns (highlighted by the red solid box). (b) Matrix from (a) introduces new zero-value computations (highlighted by the red dashed box).

sparse matrices in terms of storage, their compatibility with systolic arrays is limited (detailed in Fig. 1). For instance, the CSR format requires data decoding and reorganization to align with systolic dataflow, introducing additional overhead. Furthermore, computational efficiency during the calculation process remains a challenge. In particular, the occurrence of zero-column multiplications can exceed 50% of the total computations at high sparsity. Therefore, there is a need to design novel sparse data representation formats that are optimized for systolic arrays, taking into account their unique characteristics, to enhance computation efficiency.

To address the aforementioned challenges, we introduces SPSA, a framework based on systolic arrays, for a combined software and hardware design aimed at accelerating sparse matrix multiplication. Within SPSA, we employ two novel key technologies: 1) sparse packing and 2) CSXD encoding. By combining these core technologies that involve both software and hardware, SPSA achieves efficient sparse computation on systolic arrays. In summary, the contributions of this article can be summarize as follows.

1) We present a novel sparse packing method, suitable for systolic arrays, to compress sparse matrices. Additionally, we innovatively introduce graph coloring problems on the software side to determine the optimal sparse-packing strategy, while on the hardware side, we ensure the continuity and correctness of matrix computation by lightweight PE reconstruction of systolic arrays.
2) To the best of our knowledge, the proposed CSXD is the first data storage format highly tailored to the characteristics of systolic arrays, balancing storage efficiency and computational efficiency. Complementing this on the hardware side, we design fast paths to ensure the release of acceleration potential.
3) We have equipped SPSA with adaptation schemes, allowing it to operate efficiently in resource-constrained scenarios through coordinated interaction between hardware and software.

By conducting tests with a large number of sparse matrices, SPSA has demonstrated commendable performance. The obtained speedup ratios are 1.6× for Sputnik and 6× for cuSparse on the GPU platform; 5.7× for J_stream and 7.1× for math kernel library (MKL) on the CPU platform; and 12.4× for TPU-like, 1.4× for Mentha, 5.3× for SIGMA, and 2.7× for MatRaptor on the ASIC platform. Finally, in end-to-end evaluations, SPSA achieved a performance improvement of 3.9× across the workloads of BERT, VGG19, and ResNet50.

II. Background and Related Works

A. Taxonomy of Dataflow

One method of classifying dataflow is to identify which data remains "stationary" within PE throughout the entire computation, being fully reused before eviction. We categorize the dataflow into the following classes.

1) Output stationary (OS) entails that the partial sums (intermediate results) remain unchanged within PE, while the two input dataflow between PEs. The aim is to simplify the accumulation operation and reduce the number of partial sum transfers. Some sparse accelerators like [13], [17], and [18] demand results to be processed within a single PE, and OS serves their needs effectively.
2) Input stationary (IS) involves maintaining one of the inputs unchanged within PE, while the other input and output flow between PEs. The aim is to maximize the reuse of input data, reduce the number of input data transfers, and consequently decrease memory bandwidth requirements. Depending on the specific application, further subdivisions can be made; for instance, in CNNs, it can be divided into weight stationary and IS, while in matrix multiplication ($C = A \times B$), it is divided into A stationary and B stationary. For sparse matrix multiplication, the sparsity of the input matrix can be effectively leveraged to reduce the index of nonzero elements and the number of data accesses [11], [12].
3) Row stationary (RS) involves holding the row data of a matrix within the processing unit to maximize the reuse of row data. Initially proposed by Eyeriss [19] and continued in Eyeriss v2 [20], RS is primarily used to accelerate various convolution operations rather than matrix multiplication.

B. Mapping Matrices on Systolic Arrays

When mapping matrix multiplication onto systolic arrays, it is imperative to restructure the initial matrix. Fig. 1(a) illustrates this process where the input matrix undergoes data reorganization operations, such as rotation and tilt before being fed into systolic arrays. The dense format is employed for storing and computing the sparse matrix in systolic arrays, as depicted in the lower half of the figure. In contrast, the upper half of the figure shows the usage of the CSR format. It represents the matrix using three arrays: 1) array *val* containing all nonzero elements of the matrix stored in row-major order; 2) array *idx* storing the column indices corresponding to each nonzero element in the values array; and 3) array *ptr* indicates the starting index of each row in the values array, with the last element pointing to the end of array *val*.

Although the CSR format reduces storage and bandwidth requirements for highly sparse matrices, it cannot be directly utilized in the computation. Prior to being employed in systolic arrays, the CSR format must be decoded and reorganized. Following the reorganization process, zero padding (indicated by yellow dashed squares) is introduced to achieve data skew, resulting in the creation of new all-zero columns (represented by red solid line boxes). Subsequently, as depicted in Fig. 1(b), we represent matrices A and B in an indexed format to describe the matrix multiplication computation on systolic arrays. We utilize the IS dataflow. Initially, the B matrix is preloaded onto systolic arrays, and then the matrix A flows from left to right, while the calculation results flow out from top to bottom until the computation is completed. Notably, due to the presence of the all-zero column in matrix A (denoted by the red solid box), zero-column multiplications (highlighted by the red dashed box) arise during the computation. It is important to note that zero-column multiplication is an inefficient operation as it yields a result of zero and does not contribute to the final outcome.

C. Sparse Accelerators

Sparsification has been recognized as an effective approach for reducing computations, leading to the development of various architecture designs aimed at supporting sparse computation [13], [18], [21], [22], [23], [24], [25]. Prior works have focused on efficient software-hardware co-design methods [23], [24], [25], where special sparse patterns are designed to coordinate with the accelerator. However, these sparse accelerators primarily target static sparsity, such as structure pruning, resulting in limited applicability to scenarios involving dynamic sparsity, such as in deep learning training. In contrast, SPSA is capable of handling sparsity without relying on predefined patterns.

In recent years, sparse architectures that leverage dual-side sparsity have gained attention. Several accelerators [11], [12], [17], [22], [26], [27] utilize the indices of two multiplicative matrices in tandem to determine memory accesses and computations. Some of these works [11], [27], [28], [29], [30] specifically propose sparse acceleration techniques for convolutional computations in neural networks. SIGMA [12], for instance, utilizes a collection of adaptable dot-product engines to create a flexible dot-product unit, thereby augmenting the intricacy of PEs. SIGMA adopts the Benes topology for uninterrupted data distribution and introduces a novel tree-based reduction network. Nevertheless, the intricate nature of the interconnection networks within SIGMA results in notable increases in both area and energy overhead. Conversely, the SPSA system capitalizes on the inherent dataflow of systolic arrays, necessitating only minimal adjustments to fulfill specific requirements.

It is worth mentioning that several works [7], [8], [14], [16], [31] have proposed unique design ideas based on systolic architecture to accelerate sparse CNNs and BERT. FSA [7] introduces a fine-grained systolic array tailored for CSR-encoded data. The CSR format does not integrate well with the systolic array compared to the CSXD used by SPSA, failing to maximize the elimination of zero-value computations. Sanger [8] proposed a hardware/software co-design that leverages reconfigurable systolic arrays to accelerate BERT. In Sanger, structured pruning was applied to the attention matrices, resulting in structured sparsity. Meanwhile, SPSA aims to deal with unstructured sparsity. Additionally, Kung et al. [16] proposed an approach to pack sparse CNNs into a denser format for efficient implementations using systolic arrays. But the Kung's work only supports integer matrix multiplication with losses due to pruning and fine-tuning. STPU [14] presents an algorithm for efficient matrix packing by merging columns and a systolic array-based design that utilizes conditional execution to handle sparse matrices. STPU is designed for sparse matrix-vector multiplication, by splitting a vector into consecutive multiple inputs while keeping the PE's input interface constant. To ensure the correctness of the multiplication, STPU's PEs have added hold and latch functionality, which, however, may lead to potential stalling of partial sums. These factors limit STPU's support for sparse matrix multiplication. Moreover, Mentha [15] is a systolic array accelerator based on packing technology, which further enhances the efficiency of sparse matrix multiplication on the systolic array. However, it lacks a more detailed analysis of dataflow and does not address the creation of new all-zero rows/columns due to matrix reorganization during computation, which could be further eliminated, a task accomplished by SPSA. The aim of SPSA is to address unstructured sparsity and achieve accelerated lossless sparse matrix multiplication. Moreover, SPSA further enhances overall execution efficiency by improving storage efficiency and computational efficiency through the use of new sparse data formats.

III. SPSA Overview

We present SPSA, a collaborative framework that combines software and hardware to efficiently perform sparse matrix multiplication on systolic arrays. Fig. 2 provides an overview of SPSA's execution process. The software component handles sparse packing, encoding, and mapping onto the hardware design based on systolic arrays. Our design emphasizes the interaction between software and hardware to achieve optimal performance. The software component receives configuration information and constraints from the hardware side,

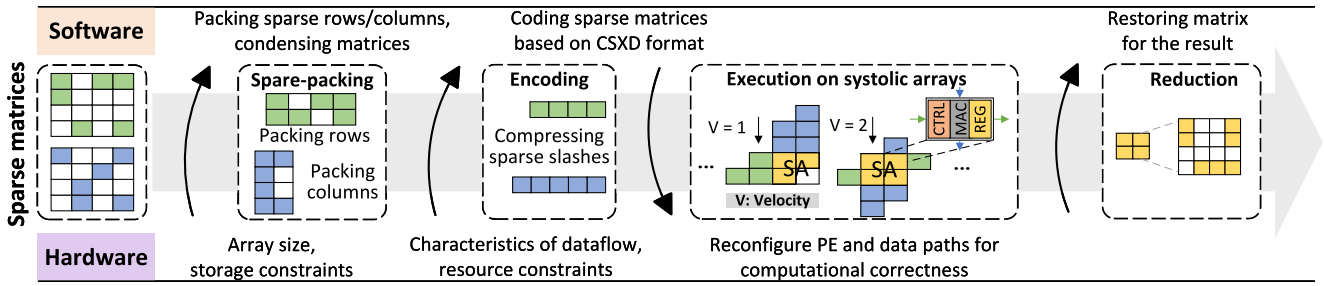


Fig. 2. Overview of SPSA framework.

controlling the compression level of the compression algorithm (explained in Section IV). The packed matrix is then encoded and stored based on hardware's dataflow characteristics and resource constraints (explained in Section V). On the hardware side, PEs and the data path are reconfigured while preserving dataflow characteristics (explained in Section VI). This allows accurate computation with moderate additional overhead. The reconfigured PEs support operations like multiplier and adder matching, and skipping zero-value computations. The data path enables flexible dataflow into the array at varying speeds. Finally, the software component handles result restoration.

By combining software and hardware, we efficiently compute sparse matrix multiplication on systolic arrays. Our collaborative design methodology ensures optimal performance and efficiency while adhering to hardware constraints.

IV. Packing Sparse Rows/Columns

A. Basic Concept

Considering the calculation rules for matrix multiplication, it is observed that the common dimensions (columns of matrix A and rows of matrix B) have a one-to-one correspondence. This implies that adjusting only the noncommon dimension without altering the common dimension does not affect the correctness of the multiplication. Consequently, we can exploit this property to compress matrices along the noncommon dimensions, aiming to obtain more condensed matrices. To illustrate this concept, we present an example as depicted in Fig. 3(a). In this example, we calculate $C = A \times B$. Note that the different colors represent different rows and columns in the matrices; for example blue represents the first row of matrix A and the first column of matrix B. After applying compression, matrix A (4×2) is transformed into matrix A^* (2×2) by row-dimension compression, while matrix B (2×4) becomes matrix B^* (2×2) by column-dimension compression. In more detail, the first and fourth rows of A combine to form the first row of A^*, the second row of A becomes the second row of A^*, and the third row of A is removed as it is made up entirely of zeros. Subsequently, C^* can be obtained from matrix multiplication with SPSA between A^* and B^*. We can then restore C^* to C. It is crucial to note that a single position in the C^* matrix may encompass multiple partial sums, such as the two elements "2 and 3" in C^*_{00}, which are not to be accumulated together since "2" corresponds to the result in C_{00}, while "3" pertains to the result in C_{32}. These partial sums must be preserved meticulously to ensure the accurate reconstruction of the result matrix.

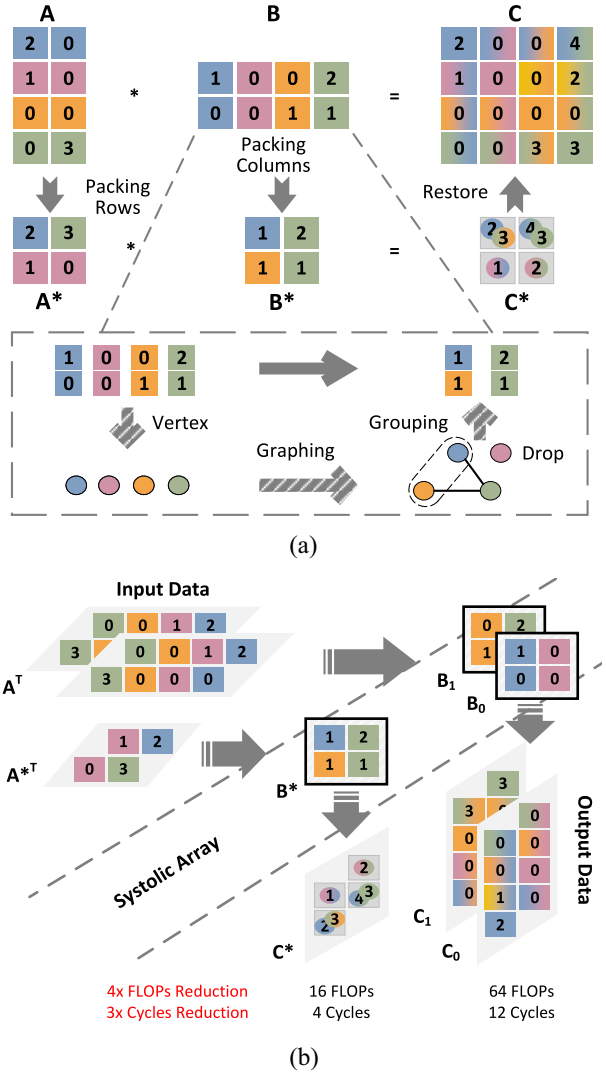


Fig. 3. Matrix multiplication within SPSA. (a) Packing of sparse matrices is accomplished by treating the sparse packing problem as a graph coloring problem, allowing for packing in either the row or column direction. (b) Packed matrix multiplication results in a 3× reduction in cycles and a 4× reduction in FLOPs compared to the unpacked baseline. (a) Packing rows and columns in sparse matrices. (b) Multiplication between packed matrices on SPSA.

As shown in Fig. 3(b), the matrix multiplication in Fig. 3(a) is mapped to a 2×2-scale systolic array. Initially, we partition

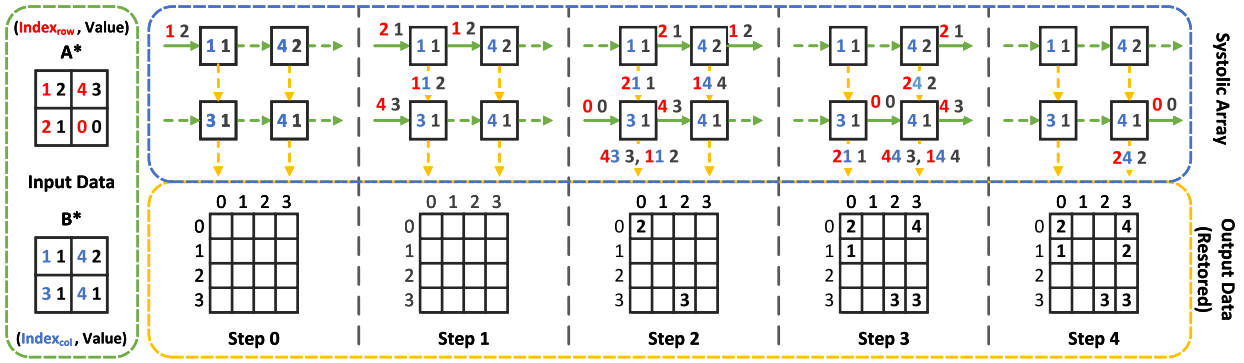


Fig. 4. Computation dataflow of matrix multiplication in SPSA.

the matrix B into block matrices B_0 and B_1, with a block size of 2×2, then load the B_0 matrix into systolic arrays. Subsequently the transposed and tilted matrix A flows into systolic arrays in a step-by-step manner and is calculated with B_0 to obtain the partial result C_0. Next, B_1 is loaded and calculated with A to yield another partial result C_1. Finally, C is acquired by splicing C_0 and C_1. Similarly, the matrix B^* is loaded initially, after which the matrix A^* is transposed and flows obliquely into systolic arrays. The reconfigured PEs in SPSA (further details provided in Section VI) guarantee the validity of C^*. Overall, SPSA achieves a reduction of $4\times$ in terms of floating-point operations (FLOPs) and $3\times$ in terms of time in this example.

B. Computation Dataflow

Within the computational flow of SPSA, data is encoded in the (index, value) format, facilitating storage, transmission, and computation among PEs. To illustrate, Fig. 4 delineates the matrix multiplication process of A^* and B^* in SPSA (corresponding to Fig. 3). First, the input matrices are represented in the format (index, data), as shown in the green frame. Additionally, the red is the row number of A^*, the blue is the column number of B^*, and the black is the value. Note that the index begins numbering from 1, while the index of zero-value data is set to 0 to skip the zero-value calculation in SPSA. At step 0, A^* is tilted and flows into SPSA after B^* is loaded at the PEs (in the blue frame). Subsequently, the calculation process progresses from step 1 to step 4. At step 2, the first set of the output in C^* is "43 3, 11 2," meaning that the value of the third column in the fourth row is 3, while the value of the first column in the first row is 2. Accordingly, we can restore C^* to C (in the yellow frame). At step 4, the calculation is finally completed; as expected, the result is equal to matrix C in Fig. 3.

C. Sparse-Packing Algorithm

The goal is to achieve optimal sparse-packing performance while avoiding excessive resource. In particular, we abstract a sparse matrix A into an undirected graph G. Noncommon dimensions are abstracted as a vertex set V, where an individual vertex represents a row or a column of matrix A, while the relationship between two rows/columns is abstracted as an edge set E; we define an edge as valid when both rows/columns have a nonzero value at the same column/row. We then divide V into K groups, of which forms an independent set where there are no adjacent vertices. Our goal is to obtain the minimum K value. Accordingly, at this point, we have transformed our issue into an optimal graph coloring problem. To illustrate, inside the gray line box in Fig. 3(a) is the process of packing the B matrix by graph coloring. The four columns in the B matrix are first abstracted to four vertices, and then the relation graph is generated based on the correspondence between them. The nonadjacent blue vertex (the first column of the B matrix) and the orange vertex (the third column of the B matrix) are grouped together, and the purple vertex is all zero so it is discarded. Finally, the compressed matrix B^* is obtained.

Algorithm 1: Overview of Sparse-Packing Algorithm

Input: sparse matrix A, compression type, threshold
Output: compressed matrix A^*
1 **Function:**
2 $Mask = get\_Mask(A)$;
3 $Vertex = get\_Vertex(A, compression\ type)$;
4 $Edge = get\_adjMatrix(Vertex, Mask)$;
5 $grpID = Grouping(Vertex, Edge, threshold)$;
6 $A^* = compress\_Matrix(A, grpID)$;
7 **return** A^*;

Algorithm 1 outlines the execution flow of the sparse-packing algorithm. First, we mark the zero and nonzero values with 0 and 1, respectively, to obtain the mask for the sparse matrix (line 2). Meanwhile, a row or column of the matrix is abstracted as a vertex in the graph, so we generate a collection of vertices based on the compression type (row or column dimension) (line 3). Subsequently, we use a 2-D adjacency matrix to represent the relationship between vertices [the set of edges (line 4)]. It should be noted that the edge between two conflicting rows or columns will be set to one. In the next step, the group number of each vertex can be obtained from the vertices and edges through a sparse-packing function (line 5); this is a flexible solution that can be implemented in different ways as needed. Finally, we filter and compress the data of A according to the group number to obtain the compression matrix A^* (line 6).

Algorithm 2: Grouping Algorithm

```
   Input: Vertex, Edge, threshold (from Alg. 1)
   Output: grpID is the group number of vertices
1  Function:
2      for k in range(Vertex.size()) do
3          D = max_degree(Vertex);
4          for p in D do
5              if p not in grpdVertex then
6                  gNum = gNum + 1;
7                  grpID[gNum].append(p);
8                  grpdVertex.append(p);
9                  nAdjVertex.update(Edge, p) ;
10                 for i in nAdjVertex do
11                     if grpID[gNum].size() != threshold and i not
                         in grpdVertex then
12                         grpID[gNum].append(i);
13                         grpdVertex.append(i);
14                         nAdjVertex.update(Edge, i) ;
15     return grpID;
```

Algorithm 2 describes the implementation of the algorithm we use to obtain group number, corresponding to line 5 in Algorithm 1. In the first step, we add the ungrouped vertex with the largest degree to the group G_i (lines 3–8). Then in the second step, we generate a vertex set that does not conflict with the vertices in G_i (line 9). Next, in the third step, we add an ungrouped vertex in this set to G_i if the number of vertices in G_i does not exceed the threshold (detailed in Section IV-D), and update the nonconflicting vertex set at the same time (lines 10–14). In the fourth step, we repeat the operations in the third step until the nonconflicting vertex set is empty. Finally, we return to the first step for the next grouping ($i = i + 1$) after updating the ungrouped vertex set, until all vertices are grouped.

D. Adaptation Scheme

To address the challenge of striking a balance between efficiency and resources, we introduce a new input parameter *threshold*, which controls the number of vertices in each group, allowing us to adjust the matrix multiplication process in the SPSA to meet different hardware resource constraints while ensuring correctness. The complexity of the grouping algorithm increases as the number of vertices in a group grows, leading to a greater demand for resources within the SPSA. In our experiments, we utilize the Welch-Powell vertex coloring algorithm (implemented in Algorithm 2) to achieve the highest-compression ratio while ensuring sufficient resource availability. Furthermore, when faced with limited resources, it becomes necessary to set a lower threshold to reduce the algorithm's complexity and resource requirements. This adaptive approach allows us to optimize the performance of the SPSA matrix multiplication while efficiently managing available resources.

V. Compressing Sparse Slashes

A. Coded Sparse Matrix

Upon analyzing the execution characteristics of matrix multiplication on systolic arrays, we have observed that reading data in the direction of slashes aligns well with the data consumption pattern in real-time computing. Notably, slashes with all zero values do not contribute to the computation, allowing us to encode only the nonzero values when dealing with sparse matrices. Building upon these characteristics, we propose two encoding directions for sparse matrices.

1) *Compress Sparse 45-Degree Slashes (CS45D):* CS45D format involves traversing and encoding data with slashes from the bottom left to the top right. The first part in Fig. 5(a) illustrates how a 3×3 sparse matrix in dense format is encoded into a compressed matrix using the CS45D format. The red dashed box represents a slash, and in this particular matrix, there are five slashes, four of which contain nonzero values. To the right of the blue arrow, we can see five slashes that correspond to the rearrangement matrix's five columns.
2) *Compress Sparse 135-Degree Slashes (CS135D):* CS-135D format traverses and encodes data along the slash from the top left to the bottom right. The second part in Fig. 5(a) demonstrates how a 3×3 sparse matrix in dense format is encoded into a compressed matrix using the CS135D format. Similarly, there are five slashes in the sparse matrix, corresponding to the five columns of the rearranged matrix, three of which contain nonzero values. In general, CS45D and CS135D formats share the same underlying concept, differing only in the direction of data traversal. Therefore, we refer to these two encoding formats collectively as CSXD format.

Encoding: During the encoding process, we utilize four arrays to store the nonzero elements of the sparse matrix.

1) The *val* array stores the nonzero values in the order of slash traversal;
2) The *nr* array indicates the column of the rearranged matrix that each slash corresponds to (more details will be provided in a later section);
3) The *ptr* array contains information about the total number of nonzero values in each slash. This is calculated as $ptr[i+1] - ptr[i]$. The last element of *ptr* represents the total number of nonzero values in the sparse matrix.
4) The *idx* array stores the column indices of each nonzero value.

To illustrate the encoding process, let's consider an example where we encode a sparse matrix in dense, CSR, CS45D, and CS135D formats. In the figure, the green squares represent how the nonzero values are stored and encoded.

1) *Dense Format:* The nonzero value is located in the second row and second column of the sparse matrix.
2) *CSR Format:* The nonzero value is located in the second row ($i = 2$) of the sparse matrix. There is one nonzero value ($ptr[3] - ptr[2]$) on this row, which corresponds to the 0th column of the sparse matrix ($idx[ptr[2]]$) with a value of 5 ($val[ptr[2]]$).
3) *CS45D Format:* The nonzero value is located on the second nonzero slash ($i = 1$), which corresponds to the second column of the rearranged matrix ($nr[1] = 1$). There is one nonzero value ($ptr[2] - ptr[1]$) on this slash, which corresponds to the 0th column of the sparse matrix ($idx[ptr[1]]$) with a value of 5 ($val[ptr[1]]$).

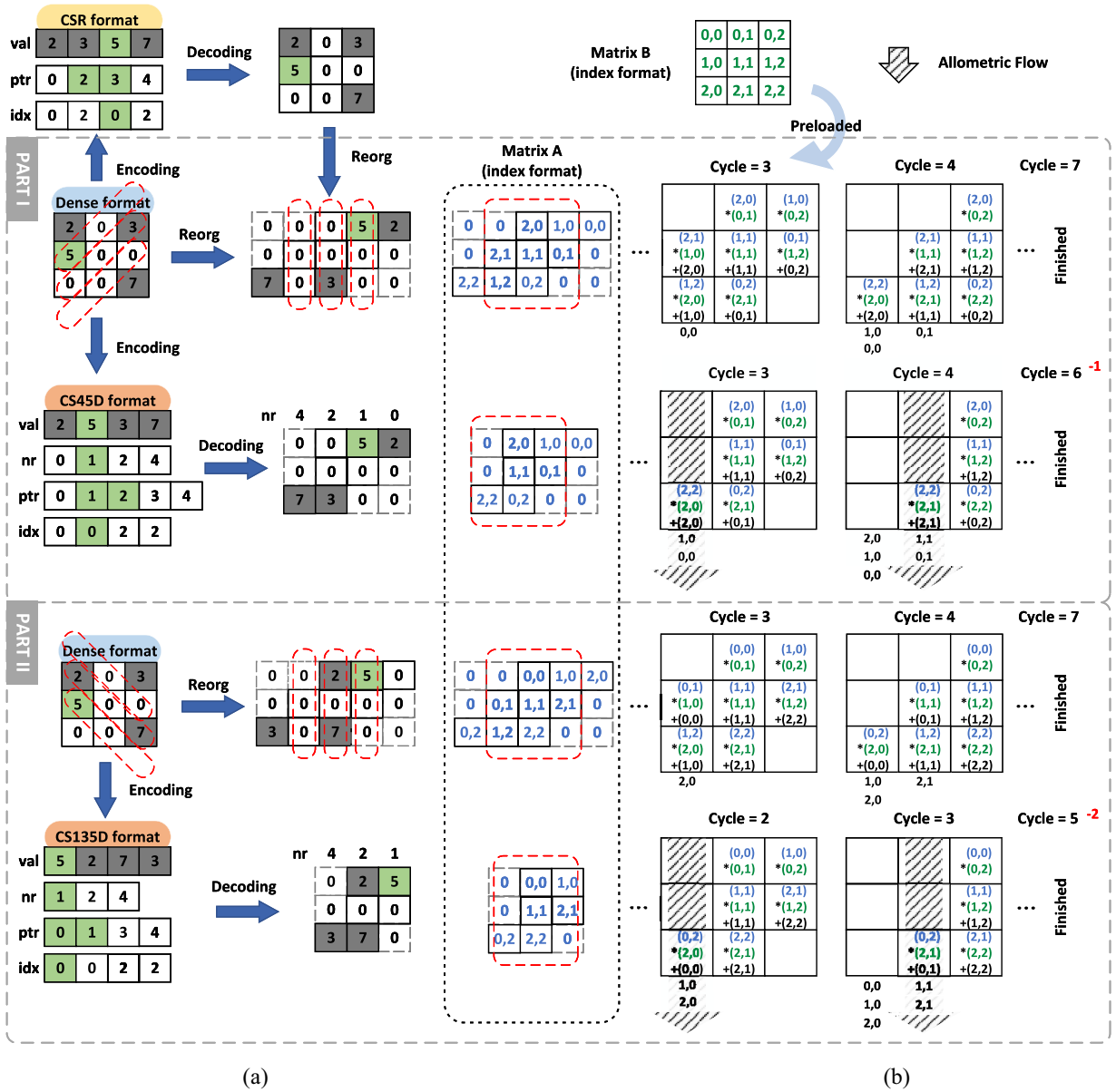


Fig. 5. CSXD format expression and execution. The first and second parts represent the encoding and mapping processes of CS45D and CS135D, respectively, contrasting them with the dense format. CSXD eliminates the newly generated fully zeroed columns in the reorganized matrix and skips the corresponding zero-value computations during calculation. (a) Sparse matrix encoded in different formats. (b) Execution flow of different formats based on systolic arrays.

4) *CS135D Format:* The nonzero value is located on the first nonzero slash ($i = 0$), which corresponds to the second column of the rearranged matrix ($nr[0] = 1$). There is one nonzero value ($ptr[1] - ptr[0]$) on this slash, which corresponds to the 0th column of the sparse matrix ($idx[ptr[0]]$) with a value of 5 ($val[ptr[0]]$).

Decoding: Decoding is the reverse process of encoding. The encoding process, along with the provided example, has made the decoding process quite intuitive. Fig. 5(a) illustrates that compared to the dense format, the decoded matrix in CS45D is reduced by one column, while the decoded matrix in CS135D is reduced by two columns. It is important to note that the computation time in systolic arrays is proportional to the matrix size. By encoding and decoding in CSXD format, the computation time can be reduced for sparse matrices. Furthermore, the decoded matrix can be directly fed into systolic arrays for calculations, eliminating the need for data reorganization operations, such as rotation and tilt.

During the encoding of sparse-packed data, the *val* array stores metadata, consisting of indexes and values, as shown in Fig. 7. It is worth emphasizing that CSXD and DIA [32] have different encoding methods. The encoding and decoding approach of DIA is not well-suited for systolic arrays, whereas CSXD is specifically designed for systolic arrays and can be used directly after decoding without any data rearrangement.

B. Execution Flow of Sparse Matrix Multiplication

Fig. 5(b) illustrates the execution of sparse matrix multiplication using systolic arrays. The matrices involved in the multiplication are represented by blue and green data.

The green matrix is preloaded into systolic arrays, while the blue matrix flows into the arrays from left to right for computation. The computed partial sums are denoted by black data and are emitted from systolic arrays in each cycle. In the CSXD format, the decoded matrix is stripped of all zero columns. During execution, the flow velocity of the partial sums is influenced by the removal of zero columns in the blue matrix. This results in a faster flow of data relative to the original matrix. To ensure accurate computation, it is crucial to align the vertical data with the horizontal data. The modified encoding scheme of the blue matrix reduces the incoming data volume, while the horizontal speed remains unchanged. Therefore, it is imperative to adjust the vertical velocity to ensure proper data alignment, resulting in accurate partial sums. It is worth noting that each column of the blue matrix has a different flow velocity, which can be obtained from the array nr. For column i in the matrix, its flow velocity is given by $nr[i] - nr[i-1]$. A flow velocity of 1 represents the traditional systolic dataflow, while velocities greater than 1 are referred to as allometric flow.

In the first part of Fig. 5(b), we compare the CS45D and dense formats, where the matrix values are represented in an indexed format. In the dense format, each column of the blue matrix has a flow velocity of 1, and the computation completes in 7 cycles. For the CS45D format, the fourth column of the matrix flows into the first column of systolic arrays in the third cycle. Since this column has a flow velocity of 2 ($nr[3] - nr[2]$), a fast vertical flow is required [indicated by the shaded arrow in Fig. 5(b)]. Subsequent arrays perform the same fast flow operation, completing the computation in 6 cycles. The correctness of the output is maintained after these operations. In the second part of Fig. 5(b), we change the inflow order of the blue matrix. By comparing it with the first part, we observe that alterations in the sequence of data inflow also impact the outflow sequence of outcomes. However, the correctness of the computation is not affected. The order is adjusted to match the CS135D format. In this scenario, the dense format still requires 7 cycles for computation, while the CS135D format completes the calculation in 5 cycles.

In summary, the CS45D and CS135D formats differ in their storage and computation efficiency for the same sparse matrix. When encoding sparse matrices, we prioritize formats with higher-execution efficiency.

C. Adaption Scheme

Velocity Control: The CSXD format not only enhances storage and computation efficiency but also imposes requirements on hardware design. Generally, the faster streams we aim to support, the more hardware logic we need to incorporate. However, it is essential to strike a balance between the additional hardware overhead and the resulting speedup. Blindly increasing the hardware overhead in exchange for speedup is not desirable. Similarly, we control the maximum velocity by introducing the threshold parameter *maxFlow*. Specifically, we impose constraints during the encoding process such that each column i satisfies $nr[i] - nr[i-1] \leq maxFlow$. If this condition cannot be satisfied, we fill in a slash with all zeros at the current position. This approach allows us to achieve the storage and computation efficiency of the CSXD format while considering limited resource constraints.

VI. Implementation

A. Overview

The overall hardware architecture of SPSA is based on systolic arrays, where each PE has extra register files to store the results and the control logic for the new dataflow. Moreover, all PEs share an on-chip global buffer, while an off-chip high-bandwidth memory (HBM) completes the memory hierarchy. PEs are interconnected by two vertical and horizontal 1-D traffic flows. In addition, on the PE of each column, we add a fast path for allometric dataflow. Fig. 6 presents an overview of the SPSA system architecture. We devise a 8×8-scale systolic array that interacts with the on-chip buffer, which in turn interacts with the off-chip HBM. Different from the baseline systolic array, we add two off-chip modules for preprocessing and post-processing between HBM and on-chip buffer, and one on-chip module for decoding CSXD format before the data flows into systolic arrays.

B. Implementation Details

Decoder Module: The decoder module comprises registers, multiplexers, and shifters, and we employ pipeline techniques to optimize the decoding process. Specifically, we divide the decoding process into four stages: 1) reading the arrays nr and ptr; 2) reading the arrays idx and val; 3) flowing into systolic arrays; and 4) initiating the calculation. In Fig. 6(a), we illustrate the pipeline for decoding the CS135D format, as depicted in Fig. 5(a). The four stages are color-coded, where the y-axis represents the column number of the decoded matrix, and the x-axis represents the cycle. The figure clearly shows the decoding phases of different columns within each cycle. In Fig. 6(b), we highlight the components involved in the different stages using dashed boxes of corresponding colors. During the third stage, the array ptr determines the validity of each group (idx, val). Additionally, in the shifter, the valid val values are shifted and stored according to idx. Finally, data conforming to the scale of systolic arrays is generated and flows into systolic arrays.

PE Unit: To perform the matrix multiplication of A^* and B^* as shown in Fig. 4, we utilize a 2×2-scale systolic array for the computation. However, when obtaining the result C^*, there may be multiple partial sums within a single PE, which cannot be handled by a traditional PE. Therefore, several modifications are necessary in the PE to ensure the correctness of the accumulation operation. Taking the PE in Fig. 6(c) as an example, when it calculates $C_1^* = A^* \times B^* + C_0^*$, the control logic first selects the appropriate value from the collection C^* that needs to be accumulated based on the indexes of A^* and B^*. After the addition, the result is output to the next adjacent PE along with the other C^* values that did not participate in the current calculation. Additionally, the control logic serves another purpose of eliminating the overhead of zero-valued multiplications and additions. When the input dataflow into

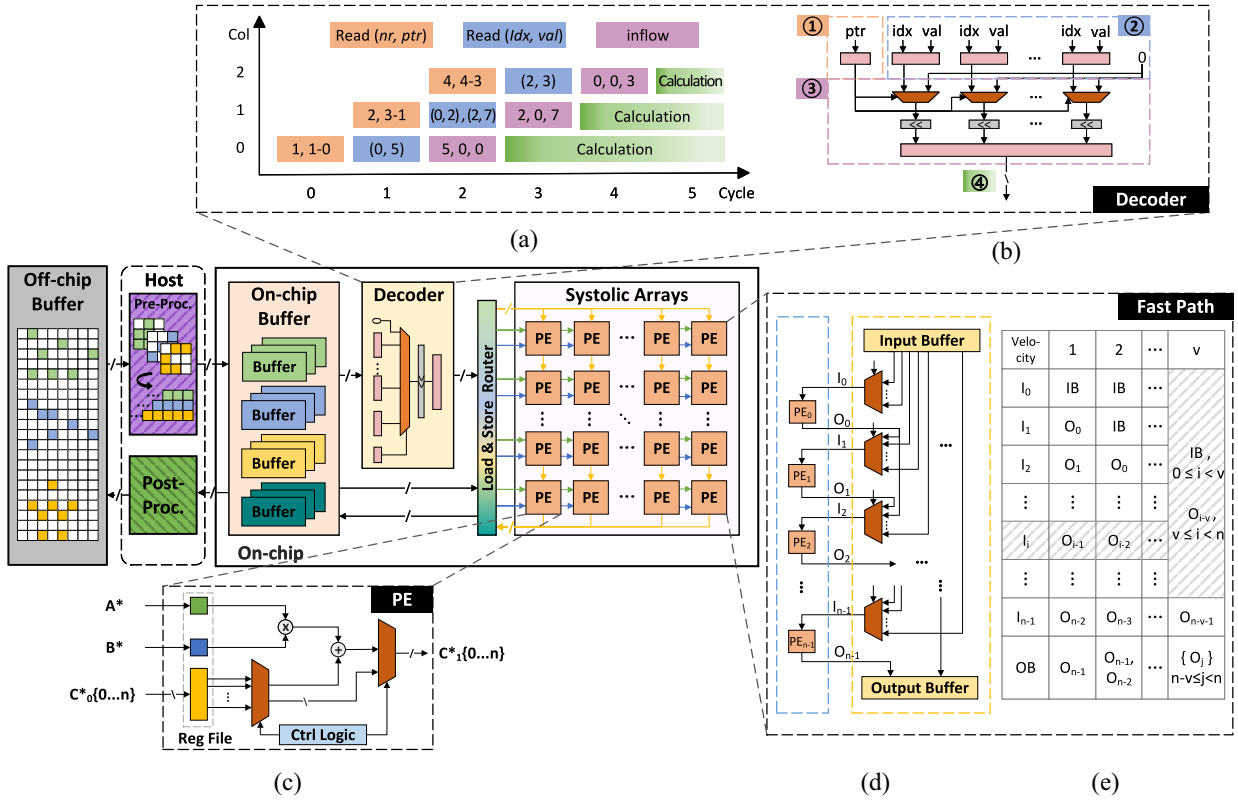


Fig. 6. SPSA system architecture. (c) PE unit. (d) Hardware design of fast path. (e) Value table.

the PE, if any of the indexes is 0, the data can bypass the multiplication step and directly flow out to the adjacent PE.

Fast Path: The fast path ensures that allometric dataflows are not stalled, enabling smooth movement within systolic arrays to complete computations. In order to ensure that data transfer between PEs is completed within a single cycle, fast paths are constructed using combinational logic to meet timing requirements. Each fast path encompasses an entire column of systolic arrays, enabling PEs within the same column to accommodate allometric dataflows through fast path. Fig. 6(d) depicts the integration of fast paths and PEs [highlighted by yellow and blue dashed boxes, respectively, in Fig. 6(d)]. Notably, PEs can receive inputs not only from adjacent PEs but also from nonadjacent PEs via the fast paths. To achieve this, the array *nr* in the CSXD format serves as the control signal for the multiplexer within the fast path, enabling the selection of the appropriate data as the input for the PE based on the velocity. Fig. 6(e) illustrates the input states corresponding to the maximum velocity values of 1, 2, and v on the right side, providing detailed insights into the input configurations. Here, IB/OB denotes the input/output buffer, I_i/O_i represents the input/output of the ith PE, and n denotes the number of PEs within a column. Based on different maximum velocity, we configure different multiplexers for fast paths. Specifically, if the maximum velocity is 4, then we will set up 4-to-1 multiplexers, meaning the current PE's multiplexer can receive outputs from the preceding four adjacent PEs. Subsequently, depending on the runtime velocity, one of them is selected as the current PE's input. For instance, if the runtime velocity is 2, then the PE_i selects the output of the PE_{i-2} as its input. Similarly, its output will be transmitted to the PE_{i+2}. If it exceeds the range of PEs, it will obtain input from the input buffer, or the output will be transmitted to the output buffer. In Section VII-B1, we conduct experiments on setting the maximum velocity and based on the experimental results, we configure a 4-to-1 multiplexer for each PE.

Host Processing Modules: This section encompasses two distinct modules: 1) preprocessing and 2) post-processing. It is customary to store sparse matrices in HBM using a dense format. The preprocessing module assumes the responsibility of compressing and encoding these matrices before their transfer onto the chip. Subsequently, the compressed data finds its place in the on-chip buffer, employing CSXD format. Upon the completion of computation, the post-processing module undertakes the task of restoring the result to either a dense format or an alternative compressed format, contingent upon the specific requirements at hand. It is a widely adopted convention to offload the computational tasks performed by these modules to the host side, where they can be co-processed by CPUs or DSPs.

C. Dataflow Within PEs

Actually, sparse matrices are challenging to compact into fully dense matrices. Therefore, the compressed matrix retains increased density while containing zero-valued data. To minimize power consumption in PEs when involving zero-valued elements in computations, dataflow into the compute unit is prevented. In such cases, the PE skips the calculation and passes the data to the next adjacent PE. Fig. 7 details the

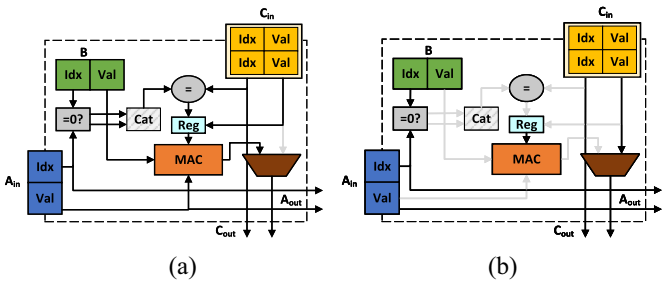
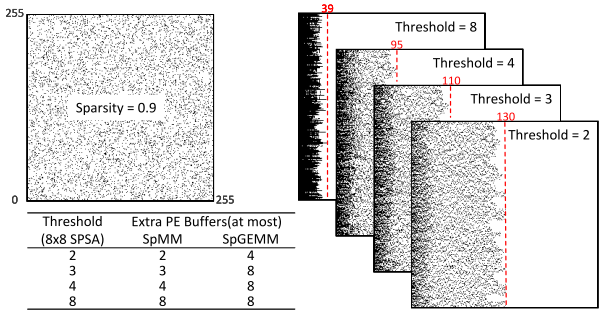


Fig. 7. Micro-architecture and dataflow within a PE in IS dataflow mode. (a) Calculation of $C^*_{\text{out}} = A^* \times B^* + C^*_{\text{in}}$. (b) Skip zero-value calculations and pass the input.



Threshold (8x8 SPSA)	Extra PE Buffers(at most) SpMM	SpGEMM
2	2	4
3	3	8
4	4	8
8	8	8

Fig. 8. Compression visualization at different thresholds.

dataflow of $C^*_{\text{out}} = A^* \times B^* + C^*_{\text{in}}$ in IS mode. Specifically, B^* remains stationary in the register file, while A^* and C^* flow horizontally and vertically, respectively. Simultaneous pass and compute functions are performed by the PE when both A^* and B^* are nonzero values. This involves concatenating the indices of A^* and B^*, matching the items in C^*, and subsequently sending the three matched values to the multiply–add calculation (MAC) unit to obtain C^*_{out}. Conversely, when either A^* or B^* is zero, the computation is bypassed, and A^* and C^* are directly passed to the next PE, as shown in Fig. 7(b). It is important to note that dataflow within PEs remains uninterrupted, with only the computation path disabled based on specific conditions. This design ensures efficient data processing without blocking the dataflow within PEs.

VII. Evaluations

A. Experimental Setup

We built cycle-accurate simulators to explore the design space and measure performance for SPSA. Furthermore, we generated RTL design of SPSA and synthesized the design with the ASAP 7nm open source PDK [33] to estimate the chip area and total power.

Baselines: We compared SPSA with designs on three different platforms.

1) *CPU Platform:* We use Intel's MKL [34] and J_stream [1] on Intel Xeon CPU E5-2660v4@2GHz. The CPU supports a peak memory bandwidth of 76.8 GB/s. Since sparse matrix multiplication is mostly memory-bound [2], [18], and the peak bandwidth of HBM used by SPSA is 128 GB/s, we have accordingly extended the CPU performance to the same bandwidth for comparison, as in [18].
2) *GPU Platform:* We use cuSPARSE library [35] and Sputnik [36] on GeForce RTX 2080 Ti. The peak bandwidth of this GPU is 616 GB/s. Similarly to the CPU, we have quantized the bandwidth to 128 GB/s for comparison.
3) *ASIC Platform:* We compare SPSA with the STPU [14], the work of Kung et al. [16], TPU-like architectures [6], Mentha [15], SIGMA [12], MatRaptor [18].

For a fair comparison, they were all configured with the same number of PEs (64), as well as identical HBM modules with a peak bandwidth of 128 GB/s. All designs utilized a 32-bit single-precision data type. Both TPU-like and Mentha were set up with an 8 × 8 configuration, matching SPSA, at a clock frequency of 2 GHz. For SIGMA, we employed 8 flex-DPEs, each sized at 8, adjusting the number of elements distributed per cycle according to bandwidth, while increasing the frequency to 2 GHz. As for MatRaptor,we increased the number of PEs to 64 and employed a round-robin allocation strategy consistent with the original paper, along with a 2 GHz clock frequency. It should be explained that the Kung's work supports integer matrix multiplication with losses due to pruning and fine-tuning, and the STPU mainly supports SpMV according to this article, so our comparison with them mainly focuses on the performance of compression algorithms.

Datasets: We evaluated the SPSA algorithm and the aforementioned designs using three matrix datasets.

1) Randomly generated square matrices with sparsity ranging from 0.7 to 0.99.
2) Sparse matrices from the SuiteSparse collection [37], as referenced in various studies, such as [2], [18], and [36]. These matrices are representative of real-world data structures commonly encountered in computational applications.
3) Sparse transformer datasets obtained from DLMC [38] achieve sparsity levels of 0.5–0.98 through pruning algorithms, as utilized in [36], [39], and [40]. Additionally, we conducted end-to-end evaluations using three DNN workloads: a) BERT-base; b) VGG19; and c) ResNet50. All datasets are processed using 32-bit floating-point numbers.

B. Experimental Results

1) Design Space Exploration: Threshold of Sparse-Packing Algorithm: We investigated the impact of different threshold values on the compression performance of a 4096 × 4096 matrix with a block size of 8 × 256 and a sparsity level of 0.9. Specifically, we considered threshold values of [2, 3, 4, 8]. Fig. 8 provides a visualization of a 256 × 256 submatrix from the original matrix. When varying the threshold values, as shown in Fig. 8 (right), we observe that the compression ratio is approximately [2.0, 2.3, 2.7, 6.7] times. Therefore, elevating the threshold value results in enhanced density. Furthermore, the bottom left of Fig. 8 presents the number of additional PE buffers required for SPSA under different threshold settings. Sparse packing

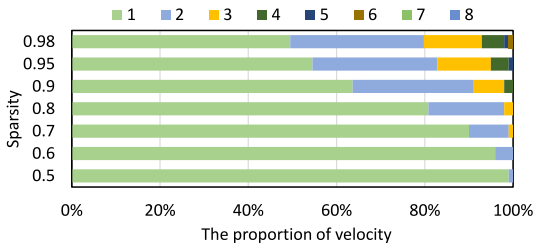


Fig. 9. Velocity distribution under different sparsity.

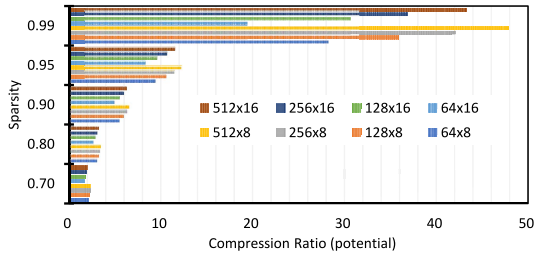


Fig. 10. Compression ratio for different sparsity and block sizes.

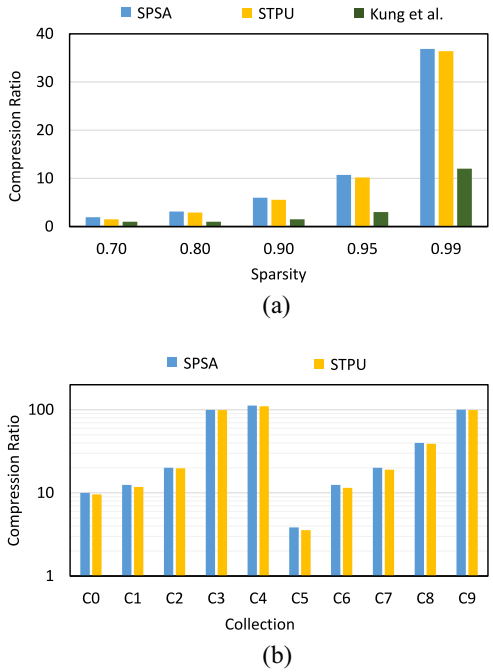


Fig. 11. Packing efficiency of SPSA compared to baseline designs. (a) Dataset 1: Square matrices with sparsity from 0.7 to 0.99. (b) Dataset 2: SuiteSparse collection.

introduces numerous partial sums in a single PE, and buffers are utilized to retain these partial sums without aggregation. Consequently, the quantity of buffers corresponds to the total of these partial sums, yet it does not exceed the dimensions of the systolic array. Sparse matrix-matrix multiplication (SpMM) entails one packed matrix, while generalized SpMM (SpGEMM) involves two packed matrices, leading to varying quantities of buffers. In particular, for an SPSA at the scale of 8×8, the additional PE buffers needed are [2, 3, 4, 8] for SpMM and [4, 8, 8, 8] for SpGEMM when utilizing threshold values of [2, 3, 4, 8]. Regardless of the threshold, SpGEMM requires more extra buffers compared to SpMM.

Velocity Configuration: Fig. 9 presents the distribution of velocity at different sparsity levels (from 0.5 to 0.98). With increasing sparsity, the proportion of allometric dataflows (velocity >1) gradually increases. For instance, at a sparsity level of 0.98, allometric flow accounts for more than 50% of the dataflows. However, it is worth noting that the maximum velocity did not exceed 8 due to systolic arrays being set to an 8×8 scale, and the majority of the allometric dataflows are characterized by low-velocity values (velocity $= 2, 3, 4$), while the proportion of high-velocity values (velocity >4) remains low. Based on these observations, it is unnecessary to cover all velocity values when constructing fast paths, as the associated costs outweigh the benefits. Similar to the sparse-packing approach, we introduce the threshold parameter *maxFlow* to control the velocity within the specified range, effectively reducing additional hardware overhead. Typically, we set the *maxFlow* to 4.

Block Size: When the matrix size exceeds the dimensions of systolic arrays, it is common practice to partition it into multiple smaller blocks for calculation, similar to the illustration in Fig. 1(b). To evaluate the compression performance under different block sizes without a threshold, we utilized randomly generated matrices of size 4096×4096, with sparsity levels ranging from 0.7 to 0.99. Fig. 10 demonstrates that the maximum potential compression ratio can be [2.4, 3.5, 6.6, 12.3, 47.9] times higher when the sparsity levels are [0.70, 0.80, 0.90, 0.95, 0.99], respectively. From the experiments, it is evident that setting the block size to $M \times N$, where N represents the size of the systolic array, results in higher-compression ratios as M increases when N is fixed. Conversely, when M is fixed, smaller systolic arrays demonstrate better performance.

2) Evaluations on SPSA (Packing Efficiency): First, we define the matrix density D as (NNZ/N), where N is the number of elements of the matrix, and NNZ is the amount of nonzero valued elements. Next, we define the compression ratio as (D^*/D), where D and D^* are the densities of the sparse matrix before and after packing, respectively. We randomly selected ten sets of data from the suitSparse dataset, including C0: *gent*113, C1: *Goodwin*_010, C2: *fs*_541_1, C3: *S*10*PI_n*1, C4: *M*20*PI_n*, C5: *lns*_511, C6: *celegansneural*, C7: *will*199, C8: *raefsky*1, and C9: *west*1505. Fig. 11 shows the compression ratio obtained from SPSA without threshold, STPU and Kung's work. From the experimental results, we can see that Kung's performance is generally low across different sparsity levels. In contrast, SPSA and STPU show superiority in both datasets. This is because Kung does not effectively handle conflicts among columns/rows in a local context. Although STPU allows for local conflicts and ensures correctness through the hold and latch operations in PEs, the resulting stalls sacrifice computational efficiency. Meanwhile, SPSA minimizes conflicts by considering matrix globally, ensuring both continuity and efficiency during calculations.

Storage Efficiency: We evaluated the storage efficiency of COO, CSR, and CSXD formats. The results in Table I demonstrate that all three formats achieve better-storage efficiency

TABLE I
STORAGE EFFICIENCY COMPARED TO DENSE FORMAT

Sparsity	0.5	0.6	0.7	0.8	0.9	0.95	0.98	Average	w/o REORG
COO	0.9	1.3	1.9	3.1	6.6	17.7	67.9	14.2	✗
CSR	1.1	1.5	2.3	3.7	7.9	21.0	78.7	16.6	✗
CSXD	1.1	1.4	2.1	3.4	7.1	18.3	71.8	15.0	✓

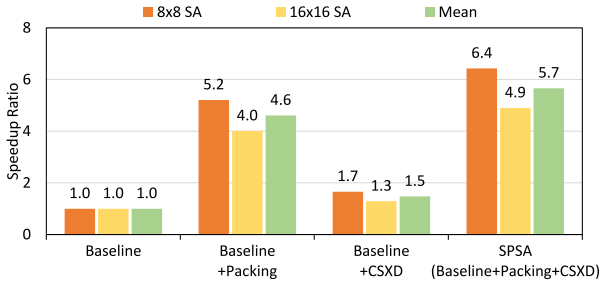


Fig. 12. Impact of different technologies on performance.

compared to the dense format. CSXD falls between COO and CSR in terms of storage efficiency for different sparsity levels. On average, COO, CSR, and CSXD achieve storage improvements of approximately [14.2, 16.6, 15.0] times, respectively. Although CSXD doesn't surpass CSR in storage efficiency, it offers advantages in computational efficiency. CSXD doesn't require data reorganization during consumption, making it more efficient. Moreover, for systolic arrays, CSXD is well-suited for sparse computation optimization, such as eliminating all-zero columns after zero padding.

Ablation Experiment: We evaluated the impact of sparsity-aware packing and CSXD on performance using matrices with a sparsity of around 0.9. Fig. 12 presents the speedup results obtained for two different scale systolic arrays. The baseline involved directly performing matrix multiplication on the systolic array, disregarding the influence of sparsity. Following this, experiments were conducted based on the baseline: using only the packing technique, using only the CSXD technique, and employing both techniques (SPSA). In the three experiments, average performance improvements of [4.6, 1.5, 5.7] times were, respectively, achieved. The performance boost in SPSA, leveraging sparsity, mainly stems from the packing technique, while CSXD, in addition to providing a friendly encoding format, further contributed to a 24% acceleration in performance. Additionally, we also observed that smaller-sized systolic arrays tend to achieve higher-acceleration ratios.

Sparsity Sensitivity: We tested the performance of SPSA at different sparsity levels. As shown in Fig. 13, which represents the speedup ratios achieved compared to the dense baseline, it can be observed that as sparsity increases, the speedup ratio also increases. This is because higher sparsity eliminates more ineffective zero calculations, leading to improved speedup ratios. Furthermore, the line in the chart represents the proportion of the achieved speedup ratio to the maximum theoretical speedup ratio, which is obtained through 1/(1-sparsity). We found that SPSA maintains an efficiency of over 80% in speedup when sparsity ranges from 0.5 to 0.8, and

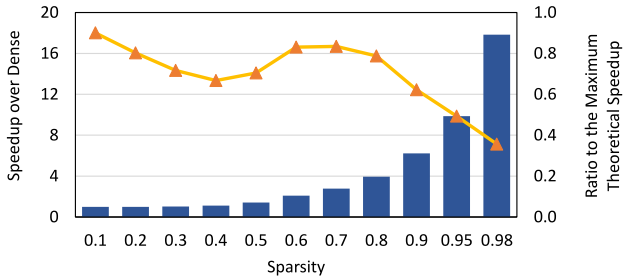


Fig. 13. Speedup ratio achieved by SPSA at different sparsities, as well as the ratio of the achieved speedup ratio to the maximum theoretical speedup ratio.

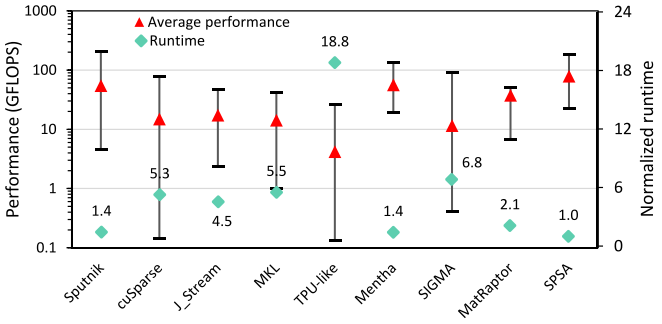


Fig. 14. Performance and normalized runtime comparisons on SuitSparse.

after 0.9, the efficiency begins to decline from 60%. Although higher-speedup ratios are more easily achieved at high-sparsity levels (greater than 0.98), the efficiency at these levels is not high, indicating room for further improvement.

3) Performance Analysis Across Diverse Platforms: When presenting throughput, we only consider operations pertaining to nonzero multiplication and accumulation to ensure consistency across designs. In order to provide a fair basis for comparison with CPUs and GPUs, we disregarded memory allocation time and solely focused on the execution time of the computational functions (memory allocation and computation being separate functions).

SuitSparse: We conducted a random selection of over 100 sparse matrices from the suitSparse dataset. In Fig. 14, we depict the maximum, minimum, and average GFLOPS achieved by each design, as well as the normalized average runtime. The performance of SPSA is [1.4, 5.3, 4.5, 5.5, 18.8, 1.4, 6.8, 2.1] times that of [Sputnik, cuSPARSE, J_stream, MKL, TPU-like, Mentha, SIGMA, MatRaptor].

DLMC: We selected a sparse transformer from the DLMC dataset, which exhibits sparsity levels of 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, and 0.98, respectively. The average performance and normalized runtime for various designs with different sparsity are presented in Fig. 15. Specifically, the average performance of SPSA across all levels of sparsity is reported to be [1.9, 6.8, 6.9, 8.7, 5.9, 1.3, 3.8, 3.3] times higher than that of the other designs [Sputnik, cuSPARSE, J_stream, MKL, TPU-like, Mentha, SIGMA, MatRaptor].

Due to the inability to handle sparsity, TPU-like exhibits poor performance. Mentha enhances the density of sparse

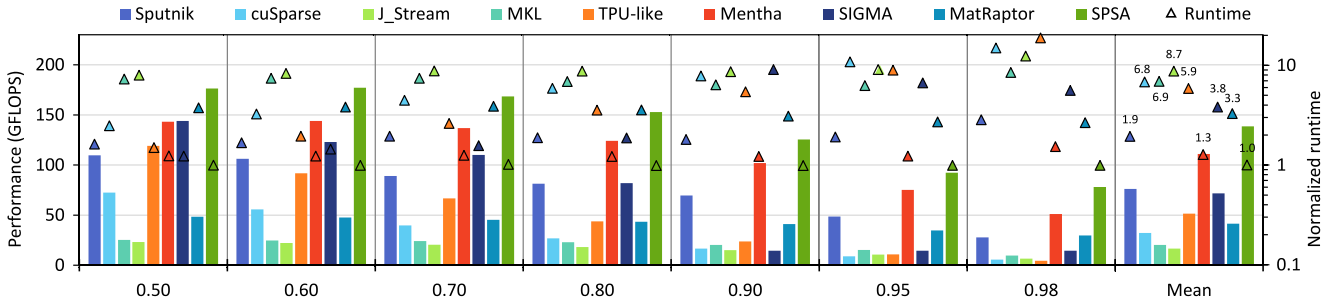


Fig. 15. Performance and normalized runtime comparisons on DLMC.

TABLE II
EVALUATIONS ON PE IN TPU-LIKE SYSTOLIC ARRAYS, KUNG ET AL., STPU AND SPSA

Datatype	INT8		FP32	
Type	Area / μm^2	Power / mW	Area / μm^2	Power / mW
TPU-like baseline	52.31 (1.00x)	0.23 (1.00x)	511.35 (1.00x)	2.08 (1.00x)
Kung et al.	141.24 (2.70x)	0.57 (2.48x)	N/A	N/A
STPU	110.79 (2.12x)	0.62 (2.69x)	585.12 (1.14x)	3.78 (1.82x)
SPSA	145.42 (2.78x)	0.66 (2.87x)	659.14 (1.28x)	2.51 (1.21x)
SPSA (Skip&Pass)	145.42 (2.78x)	0.59 (2.55x)	659.14 (1.28x)	1.26 (0.61x)

TABLE III
AREA AND POWER BREAKDOWN OF SPSA, INCLUDING THE FRACTION OF DECODER AND FAST PATHS MODULES

Scale	Type	Area / μm^2	Power / mW
8×8 (INT8)	DCD&FPs	990.55 (9.94%)	4.43 (9.79%)
	SPSA	9963.23 (100%)	45.26 (100%)
8×8 (FP32)	DCD&FPs	2360.13 (5.22%)	8.54 (4.90%)
	SPSA	45214.45 (100%)	174.35 (100%)
16×16 (INT8)	DCD&FPs	3315.62 (8.31%)	15.47 (8.58%)
	SPSA	39881.56 (100%)	180.27 (100%)
16×16 (FP32)	DCD&FPs	8132.32 (4.69%)	32.14 (4.83%)
	SPSA	173467.29 (100%)	664.99 (100%)

matrices through preprocessing, effectively reducing zero-value computations and alleviating this pain point for TPU-like. Additionally, SPSA eliminates zero values in Mentha computations, further improving the efficiency of sparse matrix multiplication on the pulsating array. SIGMA addresses sparsity by employing sophisticated allocation networks, yet the inefficiency in indexing intersection at higher levels of sparsity leads to a decline in performance. MatRaptor utilizes Gustavson dataflow and customized hardware to handle sparsity, yet strong dependencies exist in data retrieval, making the input delay difficult to overlook. Moreover, while the sorting queues within the PE are intelligent, the delay remains notable. In contrast, the superiority of SPSA is attributed to its maintenance of the systolic array dataflow, enabling it to leverage the high-density computing advantage at low-sparsity levels, while relying on packing and CSXD technology to compensate for losses at high-sparsity levels.

4) Area Overheads: Table II provides detailed information on the PE area and power overheads of SPSA and other designs in both INT8 and FP32 formats. Additionally, four extra buffers are added to each PE of SPSA. Specifically, compared to the TPU-like design in INT8 format, Kung et al., STPU, and SPSA exhibit area overheads of [2.70, 2.12, 2.78] times and power consumption overheads of [2.48, 2.69, 2.87] times, respectively. However, these multiples are significantly reduced in FP32 format. The area overheads is [N/A, 1.14, 1.28] times, and the power consumption overheads is [N/A, 1.82, 1.21] times, respectively. It should be noted that FP32 format is not supported, as stated in Kung's paper, hence denoted as N/A. Furthermore, Table II includes the overhead of SPSA when skipping zero-value calculations. In INT8 format, the power consumption of this approach is 2.55 times that of the TPU-like baseline. In FP32 format, the power

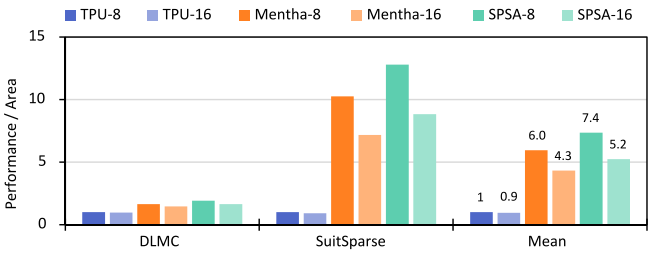


Fig. 16. Performance/area comparison of TPU-like, Mentha, and SPSA on two datasets at 8 × 8 and 16 × 16 scales.

consumption is 0.61 times that of the TPU-like baseline. The overhead in INT8 format is primarily due to the index register of the input matrix, which is comparable to an INT8 multiplier. In FP32 format, the area and power consumption of the floating-point multiplier dominate, resulting in limited additional overhead from the extra registers and logic.

Table III presents the area and power consumption of the 8 × 8-scale and 16 × 16-scale SPSA designs in INT8 and FP32 formats, respectively. Additionally, the table includes the proportions of the decoder and fast paths modules. Overall, the proportions of the decoder and fast paths modules are relatively low, with the area and power consumption not exceeding 10% in INT8 format and 5.3% in FP32 format. The higher percentage in INT8 format can be attributed to the presence of the index register of the input matrix that is comparable to an INT8 multiplier.

Fig. 16 depicts the performance/area achieved by TPU-like, Mentha, and SPSA on two datasets. The systolic array sizes of all three designs were configured to be 8 × 8 and 16 × 16. The results indicate that at an 8 × 8 scale, SPSA outperforms TPU-like and Mentha by 7.4× and 1.23×,

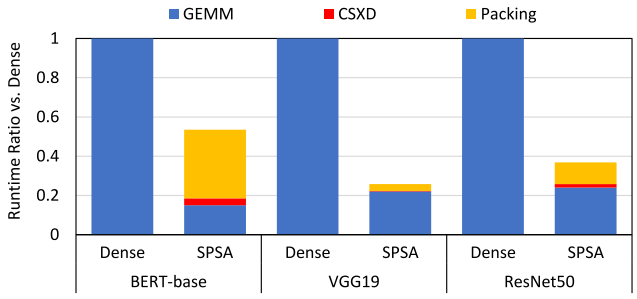


Fig. 17. End-to-end runtime evaluations using DNN workloads.

respectively. Further analysis reveals that for both Mentha and SPSA, the 8 × 8 scale consistently outperforms the 16 × 16 scale in terms of performance/area, particularly when handling highly sparse matrices. This can be attributed to the fact that the smaller-scale designs impose fewer constraints on packing and CSXD techniques, offering greater flexibility when handling sparse matrices. In conclusion, while larger-scale designs may offer superior FLOPS, smaller-scale designs demonstrate better performance/area.

5) End-to-End Evaluations for DNN Workloads: We evaluated SPSA using three DNN models, recording the inference runtime. The models included BERT-base, VGG19, and ResNet50. The matrix multiplication in BERT can be directly mapped to SPSA, whereas the convolution operations in VGG19 and ResNet50 are transformed into matrix multiplication through the im2col method before being mapped to SPSA. We utilized pruning techniques to sparsify the model's weights to approximately 0.9 and all data is in single precision. Sparse packing and CSXD encoding were implemented through software, and we recorded their runtime on the CPU. Leveraging the parallelism of matrix multiplication, we optimized the packing and encoding operations with multithreading to reduce preprocessing overhead. Finally, we compared the total runtime of running DNN workloads on SPSA with the dense baseline, which does not require preprocessing. In actual runtime, packing and encoding operations are conducted offline and overlap with online matrix multiplication. For instance, while computing the current block matrix, the sparse packing and encoding for the next block are also being executed. This approach significantly reduces latency. From Fig. 17, We observe that SPSA demonstrates varied performance across diverse workloads, attributed to disparities in the matrix dimensions engaged in the computations. In detail, SPSA achieved performance improvements of 2.9×, 4.5×, and 4.2× compared to the dense baseline across the BERT, VGG19, and ResNet50 workloads, respectively. The superior performance on VGG19 and ResNet50 is primarily attributed to the frequent reuse of weight matrices.

VIII. Conclusion

This article introduces SPSA, with the objective of accelerating sparse matrix multiplication on systolic arrays. To accomplish this, we utilize sparse packing and CSXD encoding techniques. Through the incorporation of these two innovative methods, SPSA achieved a remarkable 6.4× performance improvement compared to the dense baseline. In comparisons across various datasets with multiple designs, SPSA has demonstrated superior performance. Furthermore, in comprehensive end-to-end evaluations, SPSA achieved a notable 3.9× performance improvement across the workloads of BERT, VGG19, and ResNet50.

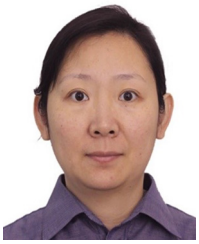
References

[1] S. E. Kurt, A. Sukumaran-Rajam, F. Rastello, and P. Sadayappan, "Efficient tiled sparse matrix multiplication through matrix signatures," in *Proc. SC IEEE/ACM*, 2020, pp. 1–14.
[2] S. Pal et al., "OuterSPACE: An outer product based sparse matrix multiplication accelerator," in *Proc. IEEE HPCA*, 2018, pp. 724–736.
[3] B. Cui, Y. Li, M. Chen, and Z. Zhang, "Fine-tune BERT with sparse self-attention mechanism," in *Proc. EMNLP/IJCNLP*, 2019, pp. 3546–3551.
[4] G. Zhao, J. Lin, Z. Zhang, X. Ren, Q. Su, and X. Sun, "Explicit sparse transformer: Concentrated attention through explicit selection," 2019, *arXiv:1912.11637*.
[5] H. T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, 1982.
[6] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. ISCA ACM*, 2017, pp. 1–12.
[7] F. Li, G. Li, Z. Mo, X. He, and J. Cheng, "FSA: A fine-grained systolic accelerator for sparse CNNs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3589–3600, Nov. 2020.
[8] L. Lu et al., "Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture," in *Proc. MICRO ACM*, 2021, pp. 977–991.
[9] Y. Cao et al., "MZ core: An enhanced matrix acceleration engine for HPC/ AI applications," in *Proc. IEEE 24th HPCC*, 2022, pp. 126–134.
[10] J. Yang et al., "BP-Im2col: Implicit Im2col supporting AI backpropagation on systolic arrays," in *Proc. ICCD*, 2022, pp. 415–418.
[11] A. Parashar et al., "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. ISCA ACM*, 2017, pp. 27–40.
[12] E. Qin et al., "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *Proc. HPCA*, 2020, pp. 58–70.
[13] K. Hegde et al., "ExTensor: An accelerator for sparse tensor algebra," in *Proc. MICRO ACM*, 2019, pp. 319–333.
[14] X. He et al., "Sparse-TPU: Adapting systolic arrays for sparse matrices," in *Proc. ICS '20 ACM*, 2020, pp. 1–12.
[15] M. Tang et al., "Mentha: Enabling sparse-packing computation on systolic arrays," in *Proc. ICPP*, 2022, pp. 1–11.
[16] H. T. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proc. ASPLOS ACM*, 2019, pp. 821–834.
[17] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proc. MICRO ACM*, 2019, pp. 151–165.
[18] N. K. Srivastava, H. Jin, J. Liu, D. H. Albonesi, and Z. Zhang, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proc. MICRO IEEE*, 2020, pp. 766–780.
[19] Y. Hsin Chen, J. S. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ACM/IEEE ISCA*, 2016, pp. 367–379.
[20] 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.
[21] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," in *Proc. IEEE HPCA*, 2020, pp. 261–274.
[22] S. Han et al., "EIE: Efficient inference engine on compressed deep neural network," in *Proc. IEEE Comput. Soc. ISCA*, 2016, pp. 243–254.
[23] S. Han et al., "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. FPGA ACM*, 2017, pp. 75–84.
[24] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs," in *Proc. MICRO ACM*, 2019, pp. 359–371.

[25] Z. Gong, H. Ji, C. Fletcher, C. Hughes, S. Baghsorkhi, and J. Torrellas, "SAVE: Sparsity-aware vector engine for accelerating DNN training and inference on CPUs," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2020, pp. 796–810.
[26] X. Zhou et al., "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proc. MICRO IEEE*, 2018, pp. 15–28.
[27] M. Mahmoud et al., "TensorDash: Exploiting sparsity to accelerate deep neural network training," in *Proc. MICRO IEEE*, 2020, pp. 781–795.
[28] Z. G. Liu, P. N. Whatmough, Y. Zhu, and M. Mattina, "S2TA: Exploiting structured sparsity for energy-efficient mobile CNN acceleration," in *Proc. HPCA IEEE*, 2022, pp. 573–586.
[29] S. Cao et al., "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," in *Proc. FPGA ACM*, 2019, pp. 63–72.
[30] D. Yang et al., "Procrustes: A dataflow and accelerator for sparse deep neural network training," in *Proc. 53rd Annu. IEEE/ACM MICRO*, 2020, pp. 711–724.
[31] M. Soltaniyeh, R. Martin, and S. Nagarakatte, "An accelerator for sparse convolutional neural networks leveraging systolic general matrix-matrix multiplication," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 3, pp. 1–26, Apr. 2022.
[32] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: Soc. Ind. Appl. Math., 2003.
[33] V. Vashishtha, M. Vangala, and L. T. Clark, "ASAP7 predictive design kit development and cell design technology co-optimization: Invited paper," in *Proc. ICCAD IEEE*, 2017, pp. 992–998.
[34] *Intel Math Kernel Library*, Intel Corp., Santa Clara, CA, USA, 2019.
[35] *CUDA CUSPARSE Library*, N. Corp., London, U.K., 2012.
[36] T. Gale, M. A. Zaharia, C. Young, and E. Elsen, "Sparse GPU kernels for deep learning," in *Proc. SC*, 2020, pp. 1–14.
[37] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, 2011.
[38] "Deep learning matrix collection." Google Research. [Online]. Available: https://github.com/google-research/google-research/tree/master/sgk
[39] Z. Chen et al., "Efficient tensor core-based GPU kernels for structured sparsity under reduced precision," in *Proc. SC'21*, 2021, pp. 1–13.
[40] S. Li, K. Osawa, and T. Hoefler, "Efficient quantized sparse matrix operations on tensor cores," in *Proc. SC'22*, 2022, pp. 1–15.

Minjin Tang received the B.S. and M.S. degrees from the National University of Defense Technology, Changsha, China, in 2018 and 2020, respectively, where he is currently pursuing the Ph.D. degree.

His research interests include computer architecture and sparse processing.



Mei Wen received the B.S., M.S., and Ph.D. degrees in computer science and technology from the National University of Defense Technology, Changsha, China, in 1995, 1999, and 2006, respectively.

She is currently a Professor with the Computer College, National University of Defense Technology. Her research interests include computer architecture, parallel programming, and scientific computing.

Jianchao Yang received the B.S. and M.S. degrees from the National University of Defense Technology, Changsha, China, in 2020 and 2022, respectively, where he is currently pursuing the Ph.D. degree.

His research interests include GPU architecture modeling and optimization.

Zeyu Xue received the B.S. degree from the National University of Defense Technology, Changsha, China, in 2022, where he is currently pursuing the M.S. degree.

His current research is focused on supporting sparsity in AI/DL on GPU and he is also interested in computer architecture and high-performance computing.

Junzhong Shen received the B.S., M.S., and Ph.D. degrees in computer science and technology from the National University of Defense Technology, Changsha, China, in 2012, 2015, and 2020, respectively.

He is currently with the Department of Computer Science and Technology, National University of Defense Technology. He does research in computer architecture, parallel computing, reconfigurable computing and programming languages.