SPLIM: Bridging the Gap Between Unstructured SpGEMM and Structured In-situ Computing

Huize Li, *Member, IEEE,* Dan Chen, *Member, IEEE,* Tulika Mitra, *Member, IEEE*

***Abstract*—Sparse matrix-matrix multiplication (SpGEMM) is a critical kernel widely employed in machine learning and graph algorithms. However, high sparsity of real-world matrices makes SpGEMM memory-intensive. In-situ computing offers the potential to accelerate memory-intensive applications through high bandwidth and parallelism. Nevertheless, the irregular distribution of non-zeros renders software SpGEMM computation unstructured. In contrast, in-situ hardware platforms follow a fixed computation pattern, making them structured. The mismatch between unstructured software and structured hardware leads to sub-optimal performance of current solutions.**

In this paper, we propose SPLIM, a novel in-situ computing SpGEMM accelerator. SPLIM involves two innovations. First, we present a novel computation paradigm that converts SpGEMM into structured in-situ multiplication and unstructured accumulation. Second, we develop a unique coordinates alignment method utilizing in-situ search operations, effectively transforming unstructured accumulation into highly parallel search operations. Our experimental results demonstrate that SPLIM achieves 276× performance improvement and 687× energy saving compared to NVIDIA RTX A6000 GPU.

I. Introduction

Sparse matrix-matrix multiplication (SpGEMM) is an essential operation for diverse applications, such as graph algorithms [14], [38] and machine learning [23], [27]. Real-world sparse matrices are often large scale, lack structure, and exhibit high sparsity. The significant sparsity entails numerous "zeros", substantially increasing SpGEMM's computational complexity. To mitigate computing complexity, researchers propose diverse compression methods to identify and skip unnecessary computation with zeros. Moreover, the unstructured nature of sparse matrices leads to irregular distributions of non-zeros, resulting in substantial random memory access. In response to this issue, various software-oriented optimizations are proposed, such as compression formats [33].

There is growing interest in hardware-oriented solutions, driven by the potential to improve execution efficiency through architecture-based algorithm optimization. Prominent architectures for accelerating SpGEMM include GPU [31], [32], *Field Programmable Gate Array* (FPGA) [1], [11], *Application Specific Integrated Circuit* (ASIC) [42], [43], and *Processing In Memory* (PIM) [5], [7]. These solutions outperform software-oriented approaches by designing dedicated architectures and dataflows that efficiently support SpGEMM. Nonetheless, SpGEMM involves numerous random access in the whole memory space (as Figure 3 shows). Conventional hardwares suffer from substantial access overhead when searching the memory space [39]. PIM-based solutions reduce the off-chip access overhead by integrating *processing elements* (PEs) near the memory banks. However, on-chip PEs can efficiently access only their local banks, leading to longer access times for searching data stored in other banks, known as cross-bank transfers [44]. Additionally, the use of on-chip PEs may reduce memory density and increase thermal concerns.

Processing-using-memory (PUM) [15] performs in-situ computation, processing tasks directly in memory cells where the data is stored. PUM-based solutions [15], [21] demonstrate exceptional performance and energy efficiency, particularly in processing *general matrix multiplication* (GEMM), a typical structured kernel. However, using structured PUM platforms to accelerate unstructured SpGEMM may lead to sub-optimal performance. The *Sparse Matrix-vector Multiplication* (SpMV) computation paradigm employed by GraphR [38] serves as an example. GraphR involves three steps: sparse matrices → compression for storage → decompression for computing. Nevertheless, this paradigm does not fully exploit the potential of in-situ computing, as the decompression phase introduces matrix remapping with significant transmission overhead. Moreover, the decompressed matrix reintroduces zeros, reducing the utilization of in-situ computing hardware (as shown in Figure 5).

In response to the current landscape, we present SPLIM, a novel PUM-based SpGEMM accelerator. SPLIM introduces two innovations. First, we introduce a new computation paradigm for SpGEMM, namely *ELLPACK-based Computation Paradigm* (ECP). ECP can perform in-situ structured vector multiplication with fewer zeros and higher hardware utilization, while the accumulation phase remains unstructured. Second, we adopt in-situ search operations for coordinates alignment, converting unstructured accumulation to highly parallel search operations. We compare SPLIM with state-of-the-art GPU, ASIC, PIM, and PUM-based SpGEMM accelerators. The experimental results show that SPLIM achieves 276× (GPU A6000), 11.2× (SAM [13]), 19.7× (SpaceA [39]), and 3.9× (ReFlip [14]) performance improvement.

Figure 1 offers a comprehensive overview of the insights behind SPLIM, depicting the transformation of unstructured SpGEMM into structured PUM-friendly kernels. In Figure 1 (a), the commonly used computation paradigm [38] used in current PUM platforms is depicted, revealing an abundance of zeros that diminish hardware utilization. Figure 1 (b) entails the adoption of structured vector multiplication utilizing the ECP method, showing huge hardware utilization gains. Figure 1 (c) showcases the accumulation of intermediate results

The authors are with the School of Computing, National University of Singapore, Singapore (e-mail: huizeli@nus.edu.sg, danchen@nus.edu.sg, tulika@comp.nus.edu.sg).

This research is partially supported by the National Research Foundation, Singapore under its Competitive Research Program Award NRF-CRP23-2019-0003. The correspondence of this paper should be addressed to Dan Chen.

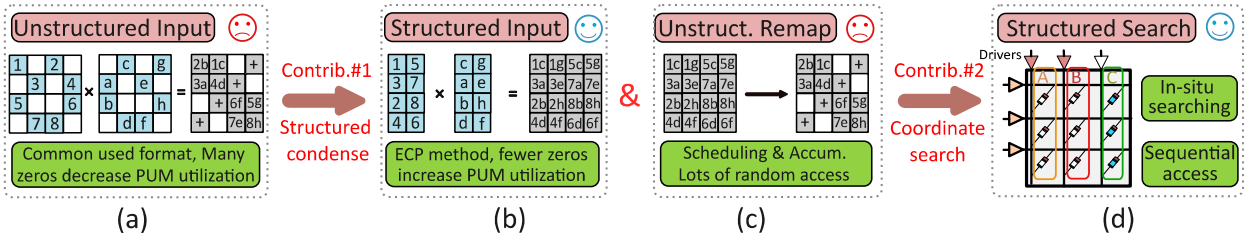

Fig. 1. (a) PUM-based SpGEMM with commonly used compression formats, (b) Structured vector multiplication based on ECP method, (c) Unstructured accumulation based on decompression, (d) Structured in-situ computing hardware

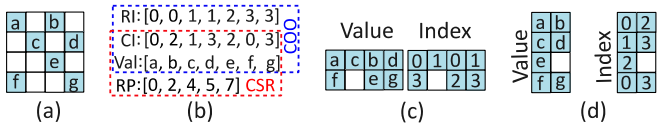

Fig. 2. (a) An example of sparse matrix, (b) The COO and CSR formats of the example, (c) The row-wise ELLPACK format, (d) The column-wise ELLPACK format

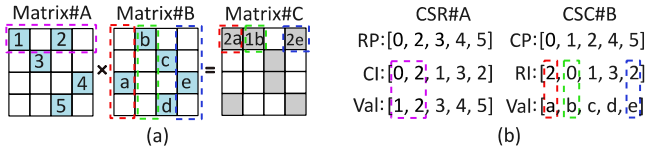

Fig. 3. (a) Matrix multiplication between sparse matrices A and B, (b) Random access of CSR#A and CSC#B

through decompression, which introduces a multitude of on-chip scheduling and random access overhead. To address this, we introduce a search-based method for merging intermediate results, capitalizing on the in-situ computing capability of the PUM platform, as depicted in Figure 1 (d).

II. Background and Motivation

A. Sparse Matrix-matrix Multiplication

Figure 2 (a) illustrates an example of sparse matrix where white cells represent zeros, and blue cells represent non-zeros. To efficiently process sparse matrices, compression formats are employed for storage and processing, allowing the skipping of zeros and reducing computation complexity. Some widely-used compression formats include *coordinates format* (COO), *compressed sparse row* (CSR), *compressed sparse column* (CSC), *diagonal format* (DIA), and ELLPACK [8].

Figure 2 (b) illustrates the COO format (blue dashed rectangle), consisting of three vectors: the *row index* (RI), *column index* (CI), and the *values* (Val). RI and CI store the row and column coordinates of Val. The CSR and CSC formats are modifications of the COO format, organized row-wise and column-wise, respectively. We present the CSR format in the red dashed rectangle, replacing the RI of COO format with a *row pointer* (RP) while CI and Val vectors remain unchanged. RP stores the number of non-zeros before each row, thus saving more storage than RI. The DIA format compresses non-zeros in each diagonal, performing well for good diagonal locality sparse matrices [23]. The ELLPACK format comprises two vectors: the Val vector stores non-zeros, and the index vector records the index of Val. As shown in Figure 2 (c), the row-wise ELLPACK format condenses non-zeros to the topmost rows, while the index vector retains the original row index. Figure 2 (d) presents the column-wise ELLPACK format, compressing non-zeros to the leftmost columns.

Figure 3 (a) depicts the SpGEMM operation $C = A \times B$. The calculation of three non-zeros in the first row of matrix C is highlighted using red, green, and blue dashed rectangles. In Figure 3 (b), we present the CSR format of matrices A and CSC format of matrix B, with the dashed rectangles representing the same meaning as in Figure 3 (a). As the Val vectors of the CSR and CSC formats do not contain index information for SpGEMM, we must realign their coordinates using the index vectors. Taking the example of red dashed rectangles in Figure 3 (b), only one multiplication will occur, as the coordinates alignment skips zeros. The results of coordinates alignment are shown by the three dashed rectangles in CSC#B of Figure 3 (b), irregularly distributed in the entire Val vector. This irregular memory access of Val vectors renders SpGEMM a typical unstructured kernel.

B. In-situ Computing

There are two types of in-situ computing: analog [37] and digital in-situ computing [15]. Analog in-situ computing is less robust to noise due to the accumulation of analog signals. As SpGEMM is commonly used in neural networks and graph processing, requiring high precision, this paper focuses on the more noise-robust digital in-situ computing, which employs memristor switching to implement logic [10]. In digital in-situ computing, a memristor cell has two states: high resistance (logic '0') and low resistance (logic '1'). By applying a proper voltage V_0, the memristor cell can switch from low resistance state '1' to high resistance state '0'. Researchers design digital in-situ computing with NOR logical gates [15].

Fig. 4 (a) shows NOR operation $Out = NOR(In_0, In_1, In_2)$. The output memristor cell is initialized to '1', while the *word-line* (WL) is initialized to $\frac{V_0}{2}$. The *bit-lines* (BLs) of all input cells are activated with voltage V_0, while the bit-line of the output cell is linked to the *ground* (GND). When all the input cells (In_0, In_1, In_2) are in the high resistance state ('0'), no current flows from the BLs to the WL. Consequently, the voltage difference of the output cell remains at $\frac{V_0}{2}$, which is insufficient for state switching. On the other hand, if at least one input cell is in the low resistance state (blue cell), the bit-line current flows from this cell to the word-line. As a

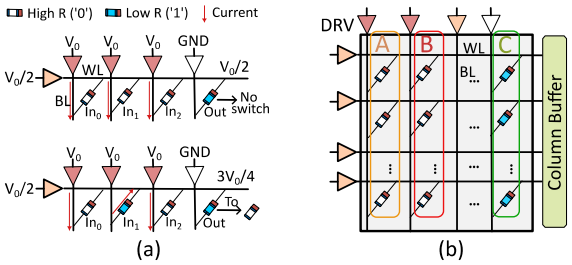

Fig. 4. (a) NOR operation of memristor switching, (b) Structured digital in-situ computing array

result, the voltage difference of the output cell changes to $\frac{3V_0}{4}$, providing enough voltage for state switching from '1' to '0'.

Figure 4 (b) presents the array-level digital in-situ computing. Initially, the output vector C is initialized to '1'. The bit-lines of input vectors A and B are set to V_0, while the word-lines are set to $\frac{V_0}{2}$. By following the same procedure as shown in Figure 4 (a), we can efficiently obtain $C = NOR(A, B)$ in a highly parallel manner. Due to the Turing completeness of NOR operation, we can perform numerous arithmetic/logic operations through a series of NOR operations [15]. The logical cells of a memristor array are coupled in a row and column manner, creating a structured architecture.

C. Motivation

PIM vs. PUM. PIM platforms [44] integrate PEs into memory, enabling in-memory computing that reduces off-chip transfers. However, due to area and thermal constraints, PIM platforms usually integrate light-weight processors near memory [44], restricting computing resources. Integrating powerful processing units can significantly reduce memory density, such as Samsung HBM-based NMP systems [17]. Moreover, significant data transfers occur between the on-chip logic units and memory banks, leading to conflicts in the *control and address* (C/A) buses shared by all memory banks. These conflicts arise when PIM platforms process large scale sparse matrix multiplication [39]. On the other hand, PUM platforms [15] show promise in reducing integration and transmission overhead. As depicted in Figure 4 (b), each memristor cell in PUM platforms can function as a logic cell without integrating on-chip PEs, exposing parallelism of million rows. Furthermore, PUM platforms perform calculations directly in where the data is stored, thus reducing data transmissions.

PUM + SpGEMM. Figure 4 (b) illustrates that all memristor cells are arranged in a regular row/column manner, providing abundant row parallelism, i.e., *the latency of processing one row is the same as processing millions of rows*, which efficiently handles structured GEMM. To utilize the high parallelism of PUM platforms, GraphR [38] proposes in-situ *sparse matrix-vector multiplication* (SpMV) with COOs format, which is widely used in PUM platforms [3], [14], [28]. Performing SpGEMM in Figure 5 (a) involves multiple iterations of GraphR's SpMV kernel. As Figure 5 (b) shows, the compressed Val vector of the COOs format cannot be directly used for SpGEMM due to unmatched coordinates. Therefore, GraphR decompresses the Val vector back to the

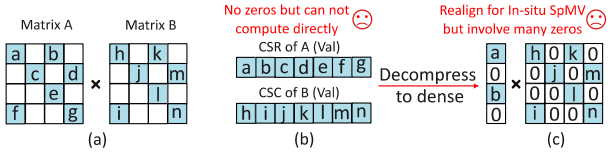

Fig. 5. (a) SpGEMM between matrices A and B, (b) The Val of CSR#A and CSC#B, (c) In-situ SpMV in GraphR [38]

structured dense matrix, reintroducing many zeros. Although PUM platform has abundant row parallelism, if too many zeros are involved, the number of valid computations decreases (as shown in Figure 5 (c)). In Figure 5 (c), the white cells (zeros) waste compute resources, referred to as invalid rows.

Our goal: PUM platforms offer excellent hardware performance, holding promise for high parallelism in accelerating SpGEMM. Nevertheless, current PUM-based SpGEMM accelerators rely on decompression to process unstructured COOs format. The decompression approach reintroduces zeros, substantially reducing PUM utilization. To this end, we aim to design a more efficient method to bridge the gap between unstructured SpGEMM and structured PUM platforms.

III. Structured In-situ SpGEMM

A. ELLPACK-based Computation Paradigm (ECP)

High-level motivation. To perform SpGEMM with PUM platforms, the Val vector of COO/CSR/CSC format must be realigned with index vectors. Decompression is the commonly used method to align Val vectors in PUM platforms, as depicted in Figure 5. However, the decompression phase reintroduces zeros and hampers the utilization of PUM platforms. In Figure 5 (c), the white cells (zeros) waste computation resources, referred to as invalid rows, i.e., *more zeros in PUM* $\rightarrow$ *less valid computing* $\rightarrow$ *lower utilization of PUM*. We seek a potential solution to tackle the above issues by adopting a novel method, which can perform SpGEMM without decompressing the Val vector, i.e., computing directly using the compressed Val vector without decompression to introduce zeros. To this end, the number of valid calculations in PUM can be greatly increased without introducing extra zeros.

Opportunities from ELLPACK format. As depicted in Figure 5, the primary objective of the decompression operation is to align the coordinates of the two input Val vectors. *If the coordinates of two input Val vectors are naturally aligned, then we can eliminate the decompression for coordinates alignment. We identify that the SpGEMM coordinates alignment can be effectively divided into two distinct parts.* First, the *vector multiplication* only requires aligning the column coordinates of the left matrix with the row coordinates of the right matrix. Second, the *accumulation phase* merges the intermediate results generated by the vector multiplication, necessitating alignment of the row coordinates of the left matrix with the column coordinates of the right matrix. This observation presents an opportunity for computing *vector multiplication* directly using the compressed Val vectors. Specifically, the row-wise ELLPACK format retains the column coordinates of left Val, while the column-wise ELLPACK format preserves the row

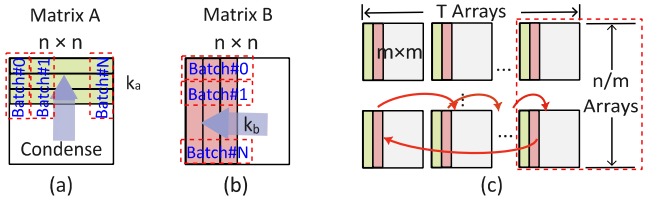

Fig. 6. (a) ELLPACK format of matrix A, (b) ELLPACK format of matrix B, (c) Matrices mapping of A and B

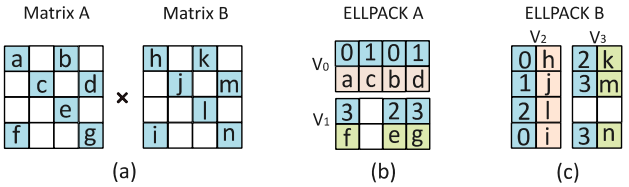

Fig. 7. (a) An example of SpGEMM $A \times B$, (b) ELLPACK format of matrix A, (c) ELLPACK format of matrix B

coordinates of right Val. Consequently, the ELLPACK's Val vector of input matrices can be directly used to perform in-situ *vector multiplication* without the need for decompression.

Matrices mapping strategy. In SpGEMM, processing matrices with millions of rows/columns needs to store the sparse matrices into multiple memristor arrays due to limited capacity of one array. Figure 6 (a) presents the $n \times n$ sparse matrix A, where all non-zeros are condensed to the top, resulting in an ELLPACK format with k_a row vectors. Similarly, Figure 6 (b) shows the $n \times n$ sparse matrix B, with non-zeros condensed to the left, yielding an ELLPACK format with k_b column vectors. To map the ELLPACK format of matrices A and B, we utilize multiple $m \times m$ memristor arrays, as presented in Figure 6 (c). Specifically, each memristor array stores a row vector of A and a column vector of B. Since $n \gg m$, we use $\frac{n}{m}$ memristor arrays to store the same vector (red dotted rectangle).

To accommodate k_a row vectors of matrix A and k_b column vectors of matrix B, we employ T memristor arrays. Each memristor array is responsible for storing $\frac{k_a}{T}$ row vectors of matrix A and $\frac{k_b}{T}$ column vectors of matrix B. When $k_a = k_b = T$, the storage configuration of matrices A and B is depicted in Figure 6 (c). To manage large input matrices and accommodate intermediate results, SPLIM processes input matrices in batches. Assuming that SPLIM has $K\times$ ReRAM rows and the input matrices have $NK\times$ rows. We will partition input matrices into $N\times$ batches as shown in Figure 6 (a) (parallel execution within batch and serial execution between batches). Accordingly, the $NK\times$ rows can be processed with $N\times$ iterations of batches.

ELLPACK-based in-situ vector multiplication. We employ matrices A and B shown in Figure 7 (a) as an example. Figure 7 (b) presents the row-wise ELLPACK format of matrix A with $k_a = 2$ row vectors, namely V_0 and V_1. Similarly, Figure 7 (c) shows the column-wise ELLPACK format of matrix B comprising $k_b = 2$ column vectors, denoted as V_2 and V_3. Each row/column vector is composed of two parts: the index vector (numbers) and the Val vector (alphabets).

To map the ELLPACK format of matrices A and B, we

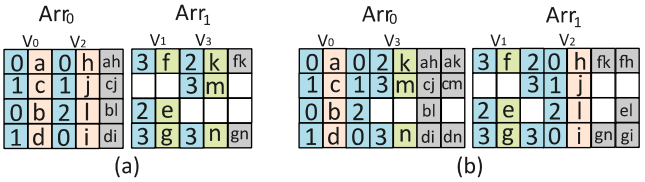

Fig. 8. (a) Data mapping and vector multiplication of matrices A and B, (b) Vector multiplication after ring-wise transfer

follow the matrix mapping approach depicted in Figure 6 (c). In our example, we use two memristor arrays, namely Arr_0 and Arr_1, to store the row vectors of A and the column vectors of B. Specifically, Arr_0 stores V_0 and V_2, while Arr_1 stores V_1 and V_3. Arr_0 and Arr_1 will conduct the in-situ vector-vector multiplication directly using the Val vector of ELLPACK formats (no decompression for coordinates alignment), leading to the generation of intermediate results (indicated by grey columns) shown in Figure 8 (a).

The SpGEMM kernel involves iterative vector multiplication among different vectors. However, storing sparse matrices in separate memristor arrays necessitates cross-array data transfers, leading to increased random access. To mitigate this issue, we employ the ring-wise transfer method, illustrated by the red arrows in Figure 6 (c). Specifically, each column vector of matrix B is transferred to the next memristor array, i.e., $Arr_i \rightarrow Arr_{i+1}$. In our example, we transfer V_1 to Arr_1 and V_3 to Arr_0. Arr_0 and Arr_1 then perform the in-situ vector-vector multiplication again, generating the next intermediate results indicated by the grey columns in Figure 8 (b). To optimize memory usage, we overwrite the previous Val vector with the newly received one. Because all the index vectors are essential for coordinate alignment, we retain the previous index vector instead of rewriting it.

Latency analysis. Figure 5 shows the in-situ SpMV using the decompressed COOs format, which requires $4\times$ SpMV iterations for SpGEMM. Figure 8 depicts our ELLPACK-based computation paradigm, achieving the same vector multiplication in only $2\times$ iterations. Compared to the COOs computation paradigm, our ELLPACK computation paradigm shows $2\times$ iterations saving due to 50% reduction of zeros. In realistic sparse matrices with $< 1\%$ non-zeros (over 99% are zeros), our ELLPACK-based computation paradigm can save over $99\times$ iterations (latency) by eliminating the zeros.

Transmission analysis. We adopt ring-wise transfer to perform the vector multiplication among Val vectors stored in different arrays, totalling $T\times$ ring-wise transfer for $T\times$ arrays. Fortunately, the cross-array transmission is more efficient than cross-bank transmission. Pinatubo [24] shows that we can perform high performance RowClone [36] between different memristor arrays using the column buffer. Specifically, we numbered the memristor arrays as odd $(2i-1)$ and even $(2i)$, $i = 1, 2, ..., \frac{T}{2}$. We can finish one ring-wise transfer with two steps RowClone. *In the first RowClone*, we read arrays $2i-1$ to their column buffer. Then, we transfer the data from column buffer $2i-1$ to $2i$. Finally, we write data from column buffer to array $2i$. *In the second RowClone*, we perform the above operation from $2i$ to $2i+1$. We need $2T\times$ RowClone to finish

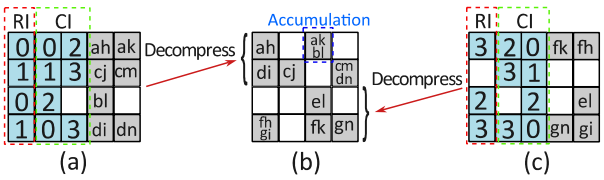

Fig. 9. (a) Intermediate results generated by Arr_0, (b) The decompressed intermediate results and accumulation operation, (c) Intermediate results generated by Arr_1

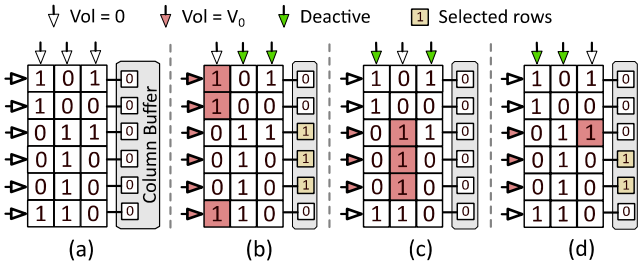

Fig. 10. (a) Int3 unsigned vector for in-situ search operation, (b) DRVs status of bit-2, (c) *Drivers* (DRVs) status of bit-1, (d) DRVs status of bit-0

all ring-wise transfers without C/A bus conflicts.

Memory analysis. ELLPACK-based computation paradigm performs $T\times$ iterations of vector multiplication to generate an intermediate vector in each iteration, totalling $T\times$ intermediate vectors for each memristor array. Thus, one memristor array is unable to store all intermediate results. To scale memristor array, we adopt the *Block Size Scalability* (BSS) in FloatPIM [15], forming a bigger array by transferring data between neighbor memory sub-blocks. For example, 32 $1k\times 1k$ arrays can be treated as a $1k\times 32k$ array with $< 4\%$ extra overhead.

B. In-situ Search for Accumulation

High-level motivation. The proposed ELLPACK-based computation paradigm exhibits high PUM utilization when processing in-situ vector multiplication. However, accumulating the intermediate results generated by this paradigm requires huge scheduling overhead, because the Val of ELLPACK format does not hold the row (column) index of left (right) matrix. Therefore, we still need to restore the coordinates using the index vectors before performing accumulation. Decompression is commonly employed to restore coordinates and accumulate intermediate results. Figure 9 (a) and (c) depicts the intermediate results from Section III-A, where the *row index* (RI) and *column index* (CI) are marked with red and green dashed rectangles, respectively. Figure 9 (b) shows the SpGEMM output matrix obtained through accumulating the decompressed intermediate results with RI and CI.

However, decompression-based methods introduce two challenges: substantial scheduling overhead and storage overhead. First, the decompression operation typically relies on the on-chip scheduler, which utilizes RI and CI of the intermediate results to generate control signals for decompression. Given the large number of randomly distributed intermediate results, the on-chip scheduling overhead can become substantial. Second, the decompression operation converts the ELLPACK format of intermediate results to a decompressed dense matrix, imposing considerable storage overhead for zeros. These challenges underscore the need for more efficient methods that can mitigate the scheduling and storage overheads.

Opportunities from in-situ search. We find that the primary objective of decompression is to extract intermediate results with identical row and column coordinates, as depicted in Figure 9. A novel approach that extracts all intermediate results with matched coordinates, without relying on a scheduler or data remapping, could significantly reduce the scheduling and storage overhead. In this context, the in-situ search operation of the memristor array emerges as a promising solution for highly parallel in-situ coordinates alignment.

Algorithm 1 In-situ Minima Search

Require: Unsigned 32-bit integer vector $V \in \mathbb{R}^n$.
Ensure: Minimum value of V.
1: Mapping vector V to n-rows ReRAM (Figure 10 (a)).
2: Initializing i to highest bit with $i = 32$.
3: Activating all row DRVs with $V = V_0$ (Figure 10 (b)).
4: **while** $i > 0$ **do**
5: Initializing column buffer to '0'.
6: Activating i-th column DRV with $V = 0$ (Fig. 10(b)).
7: Column buffer (CB) stores '1' signals (Figure 10 (b)).
8: Activating DRVs of row's CB stores '1' (Fig. 10 (c)).
▷ If no row's CB stores '1', row DRVs' activation remains the same as previous iteration (Figure 10 (d))
9: $i = i - 1$.
10: **end while**
11: Return all rows' CB stores '1' (Figure 10 (d)).

Introduction of in-situ search. The in-situ search operation of memristor arrays is illustrated in Figure 10. For visualization, we use six 3-bit unsigned integers (RI and CI in Figure 11 are also unsigned integers) as an example, shown in Figure 10 (a) (Alg. 1 line 1). Three distinct colors are used to indicate the voltage *driver* (DRV) status: white for $V = 0$, red for $V = V_0$, and green for deactivated status. During each iteration, the column buffer is set to all zeros (Alg. 1 line 5). The in-situ search starts by setting all row DRVs to V_0 (Alg. 1 line 3), resulting in high voltage word-lines, as depicted in Figure 10 (b). Next, we deactivate all column DRVs except the highest bit (bit-2) while setting its DRV to zero (Alg. 1 line 6). The voltage difference enables the word-lines' current flowing from the low resistance ('1') cells to the bit-line, represented by the red cells in Figure 10 (b). Meanwhile, the high resistance ('0') cells prevent current flow, maintaining high voltage on the word-lines, which is recorded by the column buffer (yellow cells in Figure 10 (b) and Alg. 1 line 7).

The above process is iterated for the next bit. As shown in Figure 10 (c), the row DRVs are set based on the '1' signals stored in the column buffer (Alg. 1 line 8). We deactivate all column drivers except for bit-1 while setting its DRV to zero. All memristor cells are in low resistance ('1') states, so all word-lines' current flow to the bit-line. *Consequently, the column buffer records no high voltage ('1'), and the same row*

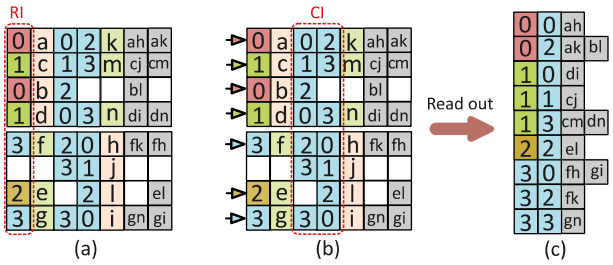

Fig. 11. (a) In-situ search operation for RI, (b) In-situ search operation for CI, (c) Sorted list of output matrix

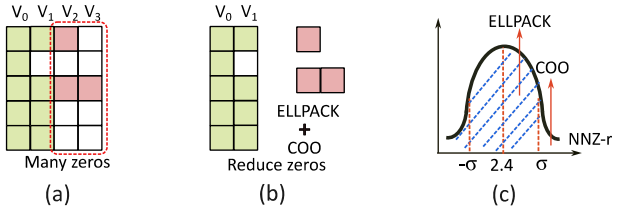

Fig. 12. (a) ELLPACK format with zeros, (b) Hybrid ELLPCAK and COO formats [2], (c) Normal distribution of NNZ-r

DRVs are activated in the subsequent iteration (bit-0). Figure 10 (d) provides details of the lowest bit (bit-0) processing. Following the same principle of leakage current, the column buffer records word-lines with high voltage, representing the storage of minimal value.

To find the next minimal value, we need to deactivate the rows storing the current minimal value. Repeating the above iterations allows us to identify those rows holding the subsequent minimal values. ReSQM [22] introduces an in-situ sorting algorithm for structured database queries. Their method draws inspiration from the bit-wise minimal algorithm for *Resistive Content Addressable Memory* (ReCAM). However, it is worth noting that ReSQM's method requires double hardware resources compared to our approach due to the fact that one ReCAM cell is composed of two memristor cells.

Detailed designs. In Figure 11 (a), we present the intermediate results obtained from Section III-A. The *row index* (RI) of matrix A is indicated by the red dotted rectangle. First, we perform an in-situ search operation on the RI to locate all rows with the smallest row index (RI#0 marked in red cells). Subsequently, we activate only these rows with RI#0 (red DRVs in Figure 11 (b)) and conduct another in-situ search operation on the *column index* (CI). In our example, the smallest CI is CI#0. Consequently, we retrieve the intermediate results of RI#0 and CI#0, i.e., 'ah' in Figure 11 (c). We then store RI#0, CI#0, and 'ah' to the off-chip memory. We modify the sign bit of CI#0 to '1', rendering this CI as invalid. Next, we iterate to search for the next minimal CI of RI#0, which in our example is CI#2. Then, we add the intermediate results of RI#0 and CI#2, i.e., 'ak' and 'bl' using an on-chip accumulator to obtain the Val of RI#0 and CI#2.

Once we complete the CI iterations of RI#0, we set the sign bit of RI#0 to '1', making RI#0 invalid. We proceed to search for the next minimal RI, which is RI#1 in our example (green cells in Figure 11 (a)). We then activate rows of RI#1 (green DRVs in Figure 11 (b)) and conduct an in-situ search on the CI of RI#1. Following the same procedure to all RI, we can obtain a sorted list as Figure 11 (c) shows. After each row's iteration, we add all grey cells using an on-chip accumulator to obtain the COO format of the output matrix. To maintain the continuity of ELLPACK format, we recommend converting the COO format of output matrix to ELLPACK format for further matrix multiplication using SPLIM, which will introduce an extra format converting overhead. Compared to $\mathbf{O}(n^2)$ time complexity matrix multiplication, format converting of linear complexity is negligible when the input matrices are large.

Memory/Transmission/Latency analysis. *Memory*: Apart from storing the output COO format in off-chip memory, we do not need additional memory, thanks to the use of in-situ search rather than decompression. *Transmission*: After the in-situ search, we can read data with exact access. Therefore, only one iteration of full memory space exact access can obtain the final results. *Latency*: For two $n \times k$ ELLPACK matrix, we can obtain all rows of the output matrix with only $n\times$ RI iterations. In each RI iteration, we need $k\times$ CI iterations, resulting in an $\mathbf{O}(nk)$ complexity. For highly sparse matrices, $n \gg k$, and we have $\mathbf{O}(n)$ time complexity. In comparison to the decompression-based method, our approach eliminates the need for an on-chip scheduler. Additionally, our method allows us to generate the COO format of the output matrix without requiring decompression, thereby reducing the significant random access and on-chip storage overhead.

C. Hybrid ELLPACK and COO Format

Real-world matrices often do not exhibit complete randomness and may contain rows or columns with good locality. In Figure 12 (a), the column-wise ELLPACK format of a sparse matrix comprises four column vectors labeled from V_0 to V_3. We use two colors to illustrate the impact of non-zero values' distributions. The green color vectors, like V_0 and V_1, demonstrate a high compression ratio with only a few zeros. On the other hand, the red color vectors, like V_2 and V_3, exhibit a low compression ratio with many zeros. There are two methods to increase the compression rate of ELLPACK format, i.e., hybrid ELLPCAK&COO and reordering. However, utilizing the reordering method for SPLIM's in-situ computing can not generate the correct result. The reasons is as follows:

Using reordering on CPU, GPU, FPGA, and ASIC platforms to accelerate SpGEMM can get good performance. We do not claim that reordering produces wrong results on the above platforms. In the above von-Neuman architectures, the storage of data is separate from the computation. The common computation procedure for SpGEMM is: (1) sparse matrix $\rightarrow$ (2) reordering for storage $\rightarrow$ (3) load data and realignment coordinates $\rightarrow$ (4) computing for results. The coordinates are realigned to get the correct results in the (3) phase.

In-situ computing platforms calculate SpGEMM directly in where the data are stored. If we utilize reordering for SPLIM, the computation procedure for SpGEMM in SPLIM is: (1) sparse matrix $\rightarrow$ (2) reordering for storage $\rightarrow$ (4) computing without data load and realignment. The above procedure will get wrong results because the coordinates are not aligned.

SPLIM can also add the (3) phase to get the correct results. However, SPLIM is designed to full utilize the advantages of in-situ computing. If we introduce the (3) phase to SPLIM for reordering method, SPLIM will re-introduce off-chip data loading and are not in-situ computing anymore. As shown in Figure 12 (b), we utilize the COO format to store the red vectors, providing a hybrid approach [2] that improves overall compression efficiency. We employ three metrics, NNZ-r, NNZ-a, and σ, to establish the boundary between the ELLPACK format and the COO format. NNZ-r represents the number of non-zeros in each row, NNZ-a denotes the average non-zeros across all rows, and σ represents the standard deviation of NNZ-r. In Figure 12 (c), we illustrate the normal distribution of NNZ-r. Rows with NNZ-r $\leq$ NNZ-a + σ are stored using the ELLPACK format, while rows with NNZ-r $>$ NNZ-a + σ are stored using a hybrid combination of the ELLPACK and COO formats.

IV. SPLIM

A. Overview Architecture

Figure 13 illustrates the architecture of SPLIM, comprising a group of *processing elements* (PEs), a *controller* (CTRL), and *input and output* (I/O) interfaces. Each PE houses *RowClone interfaces* (RC), *Accumulators* (ACC), and multiple memristor arrays. The functions of RC (for ring-wise transfer), and ACC are same as described in Section III. The memristor arrays consist of numerous row DRVs and column DRVs. The CTRL sends identical control signals to the row/column DRVs of all PEs, enabling parallel processing across the memristor arrays. Each memristor array operates in two states: memory status and computation status. In memory status, the data for read/write operations are stored in the *column buffer* (CB). In computation status, PEs will perform in-situ calculations introduced in Section III by applying appropriate voltages.

B. SPLIM Dataflow

To present the ELLPACK-based SpGEMM dataflow, we assume that the two input matrices are pre-stored in the PEs of SPLIM. We use the hybrid ELLPACK and COO formats in Section III-C to store input matrices. We segregate the storage of ELLPACK format and COO format in different PEs. Those storing ELLPACK format are referred to as ELL-PEs, while those storing COO format are called COO-PEs.

Initially, all ELL-PEs perform the in-situ vector multiplication, generating and storing intermediate results. Subsequently, each ELL-PE transmits the vector of the right matrix to its neighboring ELL-PE, following the two steps RowClone illustrated in Figure 6 (c). The process is then repeated, with all ELL-PEs performing the in-situ vector multiplication again to generate further intermediate results. The above RowClone operation and in-situ vector multiplication iterate until all intermediate results are obtained. Finally, all PEs execute the in-situ search operation to serially send the COO format of the output matrix to the accumulator for merging, the result of which is stored to the off-chip memory using I/O interface.

On the other hand, the COO-PEs will process these good locality rows stored with the COO format. All ELL-PEs are working in memory status, and the COO-PEs can access data from the ELL-PEs. This type of random memory access does not increase on-chip bandwidth pressure, because the involved rows have good row locality. The calculation of the COO format part follows the procedure introduced in Figure 5.

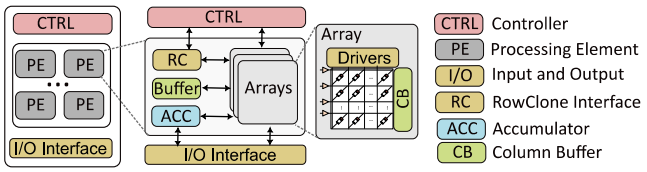

Fig. 13. SPLIM architecture

C. Comparison of Memory and Time Complexity

To quantitatively analyze the superiority of the proposed computation paradigm, we configure COO/CSR/CSC-SPLIM (short for COO-SPLIM), implementing the COOs computation paradigm shown in Figure 5 on SPLIM. For the sake of brevity, we assume that both the left and right matrices are randomly distributed $N \times N$ sparse matrices, with a standard deviation of $\sigma = 0$. Additionally, for both the left and right matrices, we assume that their NNZ-a = NNZ-r = K, indicating that the ELLPACK format comprises K vectors. In real-world matrics, N is usually million-level while K is hundred-level.

Memory complexity comparison. The memory complexity can be divided into two parts: the storage of input matrices and intermediate results. The output matrix, stored in the off-chip memory with COO format, is not considered in the memory complexity analysis. Both COO-SPLIM and SPLIM require $2NK$ memory space to store the compression formats of their Val vectors, resulting in an $\mathbf{O}(N)$ memory complexity. During calculations, COO-SPLIM decompresses COO formats to dense matrices, performing the vector multiplication between two $N \times N$ dense matrices and generating $N \times N$ intermediate results, i.e., $\mathbf{O}(N^2)$ for intermediate results. SPLIM adopts the ELLPACK-based computation paradigm, enabling direct SpGEMM using the compressed ELLPACK format. In each iteration of ELLPACK-based vector multiplication, $K\times$ PEs generate $N \times K$ intermediate elements, resulting in a total of $N \times K^2$ intermediate elements for $K\times$ iterations. Thus, SPLIM requires NK^2 memory space to store the intermediate matrices, resulting in an $\mathbf{O}(NK^2)$ memory complexity, where $N \gg K$.

Time complexity comparison. COO-SPLIM and SPLIM differ in their computation paradigm, with COO-SPLIM following *alignment* $\rightarrow$ *calculation* and SPLIM following *calculation* $\rightarrow$ *alignment*, as shown in Figure 5 and Figure 8. Let's first consider the *coordinates alignment* phase. In COO-SPLIM, the coordinates alignment requires $NK \times NK$ search operations, where each COO format of two input matrices contains $N \times K$ non-zero values. Hence, the coordinates alignment has a time complexity of $\mathbf{O}(N^2K^2)$. Similarly, in SPLIM, the vector multiplication generates NK^2 intermediate results, and it takes $N\times$ in-situ search iterations on these intermediate results to generate all rows of the output matrix, resulting in the same coordinates alignment time complexity of $\mathbf{O}(N^2K^2)$.

TABLE I
MATRICES FROM SUITESPARSE MATRIX COLLECTION [4], **Dim.** IS THE NUMBER OF ROWS/COLUMNS AND nnz_{av} IS THE AVERAGE NUMBER OF NON-ZERO VALUES PER ROW, σ IS THE STARDARD DEVIATION OF nnz_{av}.

ID	Matrix	Plot	Dim. *nnz*	nnz_{av} σ	ID	Matrix	Plot	Dim. *nnz*	nnz_{av} σ	ID	Matrix	Plot	Dim. *nnz*	nnz_{av} σ	ID	Matrix	Plot	Dim. *nnz*	nnz_{av} σ
#1	pdb1 HYS		36K 4.3M	119.3 31.86	#2	rma10		47K 2.3M	49.7 27.78	#3	bcss tk32		45K 2M	45.2 15.48	#4	ct20 stif		52K 2.6M	49.7 16.98
#5	cant		62K 4M	64.2 14.06	#6	crank seg_2		64K 14M	222 95.88	#7	lhr71		70K 1.5M	21.3 26.32	#8	consph		83K 6M	72.1 19.08
#9	soc-sign- epinions		132K 841K	6.4 32.95	#10	ship sec1		141K 3.6M	25.3 11.07	#11	xenon2		157K 3.9M	24.6 4.07	#12	ohne2		181K 6.9M	37.9 21.09
#13	pwtk		218K 11.5M	52.9 4.74	#14	stan ford		282K 2.3M	8.2 166.33	#15	cage14		1.5M 27.1M	18.0 5.37	#16	webba se-1M		1M 3.1M	3.1 25.35

Let's examine the *vector multiplication* phase. In COO-SPLIM, one SpMV iteration in Figure 5 (c) requires $N \times N$ scalar multiplications. With $N\times$ SpMV iterations for SpGEMM, the overall time complexity is $\mathbf{O}(N^3)$. In contrast, SPLIM's vector multiplication in one ELLPACK iteration only involves $N \times K$ scalar multiplications. With $K\times$ ELLPACK iterations to generate all intermediate matrices, the time complexity in SPLIM is $\mathbf{O}(NK^2)$. Comparing to $\mathbf{O}(N^3)$ complexity of COO-SPLIM, SPLIM has a lower complexity of $\mathbf{O}(NK^2)$.

V. EVALUATION METHODOLOGY

Datasets. We assess SPLIM's performance by conducting $A \times A^{\mathsf{T}}$ computations on 16 real-world sparse matrices obtained from the SuiteSparse Matrix Collection [4]. These matrices represent diverse application domains, including graph processing, scientific computing, and circuit simulation. A comprehensive overview of these matrices is provided in Table I, where we list their names, along with corresponding plots, and designate them with unique matrix IDs ranging from #1 to #16. Additionally, we furnish the number of rows (**Dim.**), the count of non-zero values (*nnz*), the average number of *nnz* per row (nnz_{av}), and the standard deviation of nnz_{av} (σ).

Baseline and comparison platforms. We compare SPLIM to the NVIDIA RTX A6000 GPU, equipped with 46GB memory, 300W TDP, and running on CUDA v11.6. For the GPU baseline, we utilize the cuSPARSE [30] library to conduct SpGEMM computations. We also compare SPLIM with state-of-the-art ASIC-based sparse matrix accelerator SAM [13], using the SpGEMM kernel of their open source project. Furthermore, we benchmark SPLIM against two contemporary PIM and PUM-based sparse matrix multiplication accelerators: SpaceA (PIM) [39] and ReFlip (PUM) [14]. We extract the SpMV kernel of SpaceA and ReFlip as descripted in their paper. Then, we extend their methodologies to support SpGEMM through multiple SpMV iterations. The results of SpaceA is obtained with the Ramulator-PIM [18]. ReFlip is state-of-the-art PUM-based SpMV accelerator using the same compression format to GraphR, so as to the same advantages and disadvantages shown in Figure 5.

SPLIM configurations. SPLIM is configured with 32 *Processing Elements* (PEs) to store the column vectors of input matrices. In cases where the number of column vectors exceeds 32, multiple vectors are stored in a single PE. Each PE consists of 1000 1024×1024 memristor arrays. Utilizing 32 memristor cells to store one float32 number, a 1024×1024 memristor array can hold 32 32-bit column vectors (32 columns per vector). Cross-array transfers of RowClone (Figure 6 red arrows) use the H-tree architecture [26].

TABLE II
SPLIM CONFIGURATIONS

Component	Area (mm^2)	Power (W)	Params.	Spec
ELL-PE properties				
ReRAM Arrays	3.45	6.14	Bit per Cell	1
			Size	1024×1024
			Numbers	1000
Buffers	0.16	0.079	Size	128KB
Accumu.	0.00024	0.02	Numbers	1
PE total	3.62	6.22	Size	128.2MB
PEs	115.85	199.12	Numbers	32
			Size	4.1GB
Controller	0.013	0.21	Numbers	1
SPLIM	115.85	199.34	Size	4.1GB

Table II presents the details of one memristor array. We utilize 1GHz *one transistor and one memristor* (1T1M) ReRAM arrays for digital in-situ computing. For the accelerator design, we use HSPICE for circuit-level simulations to measure the energy consumption and performance of all the SPLIM operations in 28nm technology. The area and power of on-chip buffers and I/O interfaces are obtained using CACTI 6.5 [29]. The DRVs of memristor array is implemented using 1-bit DAC from [34]. To evaluate SPLIM's latency and energy consumption, we modify ZSim [35] based on the mathematical proof for PUM-based cycle-accurate simulation [41]. We use two separate phases to obtain simulation results. In the first phase, we use HSPICE to obtain the parameters of all peripheral circuits. We also use CACTI 6.5 to obtain the parameters of all on-chip buffers and I/O interfaces. In the second phase, we use ZSim to combine all the above components together to obtain the system-level latency and energy.

Declaration of iso-area. All non-PUM platforms [13], [39] have two components of area, i.e., (i) on-chip logic area and (ii) DRAM area. PUM solution [14] and SPLIM have only one area because computation and storage are both in ReRAM (iii). It is unfair to SPLIM if we keep (i) and (iii) the same. It is also unfair to other comparison platforms if we keep (ii) the same as (iii) because ReRAM has higher memory density. For a fair comparison, we configure equal-capacity for non-PUM platforms and iso-area for PUM platforms. Specifically, we

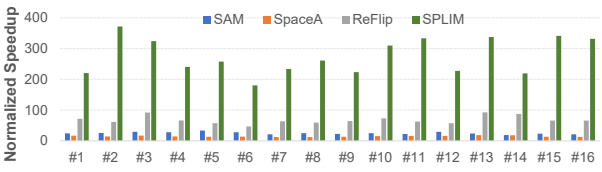

Fig. 14. Performance comparison between GPU baseline, SAM, SpaceA, ReFlip, and SPLIM (normalized to GPU)

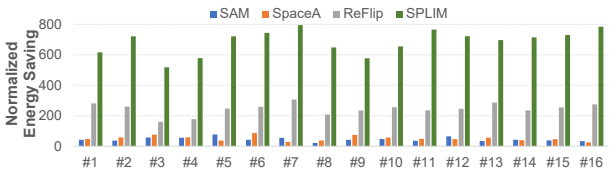

Fig. 15. Energy saving comparison between GPU baseline, SAM, SpaceA, ReFlip, and SPLIM (normalized to GPU)

keep the memory capacity the same as SPLIM for SAM [13] and SpaceA [39], while using 3 ReFlip chips [14] to maintain the same area with SPLIM.

VI. Evaluation Results

A. Performance and Energy Efficiency

We devise experiments to evaluate SPLIM's overall performance and energy efficiency in contrast to GPU baseline, SAM [13], SpaceA [39], and ReFlip [14]. The execution time and energy outcomes are depicted in Figure 14 and Figure 15, respectively, with normalization to the GPU. The point-to-point comparison and analysis are detailed below.

SPLIM vs. GPU. SPLIM demonstrates a remarkable performance advantage over the GPU baseline, achieving an average performance improvement of 276× and energy savings of 687× across all 16 input matrices. The superiority of SPLIM over the GPU baseline can be attributed to several factors. First, SPLIM effectively mitigates off-chip data movement issues, which are recognized bottlenecks in conventional hardware like GPUs [39]. Second, the utilization of the in-situ computing platform exposes SPLIM with million-level row parallelism. Finally, SPLIM adopts ELLPACK-based computation paradigm and in-situ search for SpGEMM, eliminating significant random access overhead compared to GPU baseline. The heightened energy efficiency of SPLIM, as compared to the GPU, can be attributed to its avoidance of energy consumption through off-chip transfer and random data access. Additionally, SPLIM's in-situ computing strategy substantially reduces data transfers between the storage and computation units, thus leading to reduced energy consumption. Finally, SPLIM eliminate on-chip scheduler, saving lots of random scheduling energy compared to GPU.

SPLIM vs. SAM. SPLIM demonstrates an average performance improvement of 11.2× and energy saving of 15.4× when compared to the state-of-the-art ASIC-based SpGEMM accelerator, SAM [13]. As a memory-processor separated architecture, SAM takes long latency to random access non-zero values from the off-chip DRAM. These off-chip data transfers significantly impede SAM performance. Second, SAM still relies on the on-chip scheduler for coordinates alignment. Therefore, scheduling large amount of irregular data makes the scheduler a performance bottleneck for SAM. Conversely, SPLIM's exceptional performance can be attributed to two point-to-point pivotal factors. First, SPLIM operates as a PUM-based accelerator, negating the need for extensive off-chip DRAM access, reducing latency and energy consumption at the same time. Second, SPLIM adopts ELLPACK-based computation paradigm and in-situ search rather than on-chip scheduler for SpGEMM, removing the random on-chip scheduling bottleneck with PUM-friendly in-situ operations.

SPLIM vs. SpaceA. SPLIM demonstrates an average performance improvement of 19.7× and achieves energy savings of 13.4× when compared to the PIM-based SpGEMM accelerator, SpaceA [39]. SPLIM outperforms SpaceA in terms of both performance and energy efficiency, benefiting from its PUM-based design that inherently offers enhanced computational parallelism compared to the PIM-based SpaceA. SpaceA requires cross-bank scheduling for input matrices involving many C/A buses conflicts and on-chip PE idleness. In contrast to SpaceA, SPLIM adopts PUM-friendly approaches for intermediate result storage and processing, effectively reducing on-chip memory access overhead. We further design ring-wise transfer to release C/A bus conflicts. The reduction in memory access overhead enhances both computational and energy efficiency. The innovative utilization of in-situ search operations for coordinate alignment further enhances SPLIM's efficiency, enabling exact read/write operations and mitigating inefficient scheduling for intermediate results.

SPLIM vs. ReFlip. SPLIM showcases an average performance improvement of 3.9× and energy savings of 2.8× when comparing to the PUM-based accelerator ReFlip [14]. Although ReFlip also adopts PUM-based platform, it does not fully utilize the potential of in-situ computing for the following reasons. ReFlip employs a novel data mapping strategy for their SpMV kernel, i.e., storing dense vectors rather than sparse matrices to PUM platform. Their data mapping method performs well for SpMV kernel, because the dense vectors stored in PUM do not introduce any zeros. However, their mapping method fails for processing SpGEMM because there is no dense vectors any more. To this end, ReFlip suffers the same problem as GraphR (Figure 5). First, ReFlip needs an additional decompression phase that converts COOs format (for storage) into a dense decompressed format (for computation). This decompression incurs extra data read/write operations between storage and computation formats. Furthermore, ReFlip conducts matrix multiplication in a decompressed dense format, reintroducing zeros and thereby diminishing the array utilization of the PUM platform. In contrast, SPLIM directly employs the compressed ELLPACK format for both storage and computation, eliminating the need for remapping and significantly reducing data read/write operations.

B. Efficiency of Computation Paradigm

To assess the effectiveness of the proposed computation paradigm, we establish a sister comparison platform for

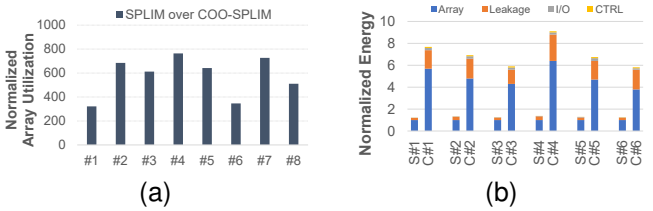

Fig. 16. (a) Array utilization comparison between SPLIM and COO-SPLIM, and (b) Energy breakdown of SPLIM and COO-SPLIM, with normalization to SPLIM's array energy. In this context, "S#1" denotes SPLIM processing matrix#1, and "C#1" signifies COO-SPLIM processing matrix#1.

SPLIM (Section IV-C), denoted as COO-SPLIM. While COO-SPLIM maintains identical configurations to SPLIM, the key distinction lies in its computation paradigm, wherein it employs the COO storage format. To mitigate storage-related challenges stemming from decompression, we configure COO-SPLIM to handle the input matrix in batches, thereby processing individual sub-matrices sequentially.

Array utilization. We define array utilization as the ratio of *the number of valid rows* to *the total number of ReRAM rows*. Figure 16 (a) illustrates the array utilization comparison between SPLIM and COO-SPLIM. SPLIM outperforms COO-SPLIM, exhibiting an average utilization enhancement of 557×. The discernable performance disparity between SPLIM and COO-SPLIM can be attributed to the following reasons. COO-SPLIM adopts an additional decompression phase to convert the COO storage format into a decompressed computation format. This decompression step involves many zeros to the computation region, thereby introducing supplementary invalid computations. Conversely, SPLIM directly employs the compressed ELLPACK format for both storage and calculation, eliminating the need for remapping between different regions. As a result, SPLIM effectively reduces introducing zeros, leading to increase in valid computations.

Energy breakdown. Figure 16 (b) depicts the energy breakdown of both SPLIM and COO-SPLIM. The energy consumption is categorized into four distinct components. The term "Array" signifies the energy expended by the memristor arrays, while "Leakage" accounts for the energy dissipation resulting from array current flowing into the *ground* (GND), attributed to high resistance cells ('0'). "I/O" and "CTRL" correspond to the energy consumption of the I/O interface and the controller, respectively. Notably, COO-SPLIM registers higher energy consumption in "Array" due to the necessity for a greater number of vector multiplication iterations, consequently requiring more frequent activation of DRVs. Additionally, COO-SPLIM exhibits increased leakage current to the GND owing to the presence of numerous zeros. Both SPLIM and COO-SPLIM demonstrate I/O and CTRL energy consumption of less than 4%, benefiting from the good energy efficiency of digital in-situ computing.

C. Sensitivity Study

The sparsity of the matrix and the distribution pattern of its non-zero values influence the performance of SpGEMM. In this context, we denote τ as the matrix's sparsity, defined as the ratio of non-zero elements to the total number of elements in the sparse matrix ($\frac{nnz}{Dim^2}$). To gauge the distribution of non-zero elements, we employ the standard deviation (σ) of the non-zero elements per row. We conduct experiments aimed at examining the impact of variations in τ and σ.

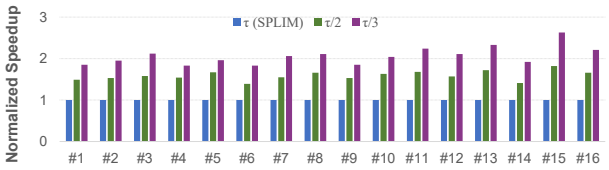

Fig. 17. Performance comparison of SPLIM processing input matrices with various sparsity (normalized to τ)

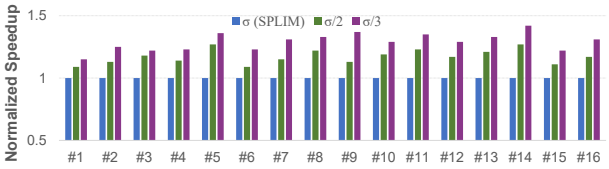

Fig. 18. Performance comparison of SPLIM processing input matrices with various standard deviation (normalized to σ)

Impact of matrix sparsity τ. We configure three levels of input matrix sparsity: τ, $\frac{\tau}{2}$, and $\frac{\tau}{3}$. The sparsity levels of $\frac{\tau}{2}$ and $\frac{\tau}{3}$ are achieved by randomly removing $\frac{1}{2}$ and $\frac{2}{3}$ of the non-zero elements from the sparse matrices, respectively. In Figure 17, we depict the normalized speedup of SPLIM when processing sparse matrices of varying sparsity. SPLIM exhibits improved performance as matrix sparsity increases, while maintaining the same matrix dimensions. Specifically, transitioning from sparsity τ to $\frac{\tau}{2}$ results in a 39.6% reduction in SPLIM's execution time. This trend can be attributed to SPLIM's utilization of the ELLPACK format, effectively condensing all non-zero values. As matrix sparsity increases, the number of ELLPACK vectors decreases, subsequently leading to fewer iterations of vector-vector multiplications.

Impact of standard deviation σ. We vary the standard deviation across three levels: σ, $\frac{\sigma}{2}$, and $\frac{\sigma}{3}$. By redistributing non-zero elements from rows with higher *nnz* to rows with lower *nnz*, sparse matrices are created with reduced standard deviations of $\frac{\sigma}{2}$ and $\frac{\sigma}{3}$. Figure 18 illustrates the normalized speedup achieved by SPLIM when processing input matrices with varying standard deviations. Notably, SPLIM demonstrates heightened performance as the standard deviation diminishes. A decrease in standard deviation signifies a narrower disparity in non-zero element counts among rows, leading to a reduction in ELLPACK format zeros. The diminished presence of zeros within the ELLPACK format contributes to an enhanced level of array utilization within SPLIM, thus reducing the number of iterations and latency.

D. Scalability

The scalability of SPLIM is evidenced through incremental augmentation of processing elements (PEs), as depicted in Figure 19. When equipped with 32 PEs, SPLIM achieves average speedups of 3.8× and 1.8× in comparison to configurations with 8 PEs and 16 PEs, respectively. This performance

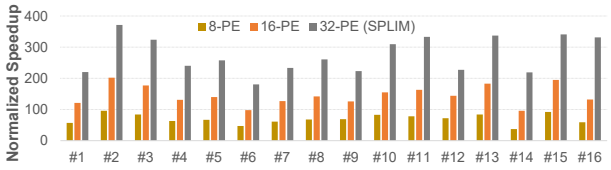

Fig. 19. Speedups comparison of different PE configurations

trend highlights SPLIM's robust scalability, a quality primarily attributed to the utilization of in-situ computing hardware. The in-situ computational memory cells within SPLIM scales proportionally with the number of PEs, leading to a significant enhancement of computational parallelism. The inclusion of additional PEs also facilitates a more granular partitioning of input matrices, resulting in a more uniform distribution of intermediate results. Importantly, SPLIM circumvents the need for cross-PE transfer of intermediate results, rendering it impervious to potential performance hindrances stemming from such transfers.

VII. Related Work

Conventional SpGEMM Solutions. Compared to CPU-based solutions [9], GPUs have higher computing parallelism and memory bandwidth, emerged as promising candidates for SpGEMM acceleration [31], [32]. The *compressed sparse row* (CSR) format has maintained its dominance as the preferred compression scheme on GPUs. In parallel, FPGA [1], [11], [16] and ASIC [12], [43] based SpGEMM accelerators have emerged to address inherent latency-bound inefficiencies associated with limited sparse data reuse. However, the performance of these conventional platforms is constrained as the size of sparse matrices expands.

PIM and PUM Accelerators. PIM and PUM platforms are potential eliminate off-chip transfers. While these solutions excel in addressing bandwidth utilization and data reuse, they grapple with issues like C/A bus conflicts and on-chip scheduling overhead. PUM platforms advantage serves as the bedrock for developing various accelerators catering to applications ranging from neural networks [19], [20], [25], [40] to graph processing [6]. While prior solutions yield remarkable performance enhancements, they have yet to unlock the full computational parallelism potential of PUM platforms. This is attributed to the inherent trade-off between array-level parallelism and hardware flexibility.

VIII. Conclusion

The objective of this study is to accelerate the commonly employed SpGEMM. PUM-based platforms offer enhanced parallelism compared to other platforms, making them a promising candidate for accelerating SpGEMM. Nevertheless, attempting to accelerate unstructured SpGEMM using structured PUM platforms results in sub-optimal performance. To bridge this gap, we introduce SPLIM, a novel co-design SpGEMM accelerator that utilizes PUM platforms. First, we develop an innovative computational paradigm by transforming SpGEMM into structured ELLPACK-based vector multiplication. Second, we introduce a search-based approach for coordinates alignment, converting unstructured accumulation into in-situ search. The experimental outcomes demonstrate that SPLIM exhibits remarkable performance and energy efficiency compared to state-of-the-art accelerators.

References

[1] Z. Bai, P. Dangi, H. Li, and T. Mitra, "SWAT: Scalable and Efficient Window Attention-based Transformers Acceleration on FPGAs," 2024. [Online]. Available: https://arxiv.org/abs/2405.17025

[2] S. R. Bernabeu, V. Puzyrev, M. Hanzich, and S. Fernandez, "Efficient sparse matrix-vector multiplication for geophysical electromagnetic codes on xeon phi coprocessors," in *Second EAGE Workshop on High Performance Computing for Upstream*, vol. 2015, no. 1, 2015, pp. 1–5.

[3] N. Challapalle, S. Rampalli, L. Song, N. Chandramoorthy, K. Swaminathan, J. Sampson, Y. Chen, and V. Narayanan, "GaaS-X: Graph analytics accelerator supporting sparse data representation using crossbar architectures," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 433–445.

[4] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011.

[5] S. Feng, X. He, K.-Y. Chen, L. Ke, X. Zhang, D. Blaauw, T. Mudge, and R. Dreslinski, "MeNDA: A near-Memory Multi-Way Merge Solution for Sparse Transposition and Dataflows," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, p. 245–258.

[6] S. A. Ghasemi, B. Jahannia, and H. Farbeh, "GraphA: An efficient ReRAM-based architecture to accelerate large scale graph processing," *Journal of Systems Architecture*, vol. 133, p. 102755, 2022.

[7] C. Giannoula, I. Fernandez, J. G. Luna, N. Koziris, G. Goumas, and O. Mutlu, "SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 1, feb 2022.

[8] R. G. Grimes, D. R. Kincaid, and D. M. Young, *ITPACK 2.0 user's guide*. Center for Numerical Analysis, Univ., 1979.

[9] Z. Gu, J. Moreira, D. Edelsohn, and A. Azad, "Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication Using Propagation Blocking," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, p. 293–303.

[10] S. Gupta, M. Imani, and T. Rosing, "FELIX: Fast and Energy-Efficient Logic in Memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–7.

[11] P. Haghi, T. Geng, A. Guo, T. Wang, and M. Herbordt, "FP-AMG: FPGA-Based Acceleration Framework for Algebraic Multigrid Solvers," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2020, pp. 148–156.

[12] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "ExTensor: An Accelerator for Sparse Tensor Algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, p. 319–333.

[13] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, "The Sparse Abstract Machine," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, p. 710–726.

[14] Y. Huang, L. Zheng, P. Yao, Q. Wang, X. Liao, H. Jin, and J. Xue, "Accelerating Graph Convolutional Networks Using Crossbar-based Processing-In-Memory Architectures," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 1029–1042.

[15] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, p. 802–815.

[16] E. Jamro, T. Pabiś, P. Russek, and K. Wiatr, "The algorithms for FPGA implementation of sparse matrices multiplication," *Computing and Informatics*, vol. 33, no. 3, pp. 667–684, 2014.

[17] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim, "HBM (high bandwidth memory) DRAM technology and architecture," in *2017 IEEE International Memory Workshop (IMW)*, 2017, pp. 1–4.

[18] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[19] H. Li, D. Chen, and T. Mitra, "SADIMM: Accelerating Sparse Attention using DIMM-based Near-memory Processing," *IEEE Transactions on Computers*, pp. 1–12, 2024.

[20] H. Li, H. Jin, L. Zheng, Y. Huang, X. Liao, D. Chen, Z. Duan, C. Liu, J. Xu, and C. Gui, "CPSAA: Accelerating Sparse Attention using Crossbar-based Processing-In-Memory Architecture," *arXiv preprint arXiv:2210.06696*, 2022. [Online]. Available: https://arxiv.org/abs/2210.06696

[21] H. Li, H. Jin, L. Zheng, Y. Huang, X. Liao, Z. Duan, D. Chen, and C. Gui, "ReSMA: accelerating approximate string matching using ReRAM-based content addressable memory," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, p. 991–996.

[22] H. Li, H. Jin, L. Zheng, and X. Liao, "ReSQM: Accelerating database operations using ReRAM-based content addressable memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4030–4041, 2020.

[23] H. Li, Z. Li, Z. Bai, and T. Mitra, "ASADI: Accelerating Sparse Attention Using Diagonal-based In-Situ Computing," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 774–787.

[24] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016.

[25] C. Liu, H. Liu, H. Jin, X. Liao, Y. Zhang, Z. Duan, J. Xu, and H. Li, "ReGNN: a ReRAM-based heterogeneous architecture for general graph neural networks," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, p. 469–474.

[26] F. Liu, W. Zhao, Y. Chen, Z. Wang, Z. He, R. Yang, Q. Tang, T. Yang, C. Zhuo, and L. Jiang, "PIM-DH: ReRAM-based processing-in-memory architecture for deep hashing acceleration," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, p. 1087–1092.

[27] L. Lu, Y. Jin, H. Bi, Z. Luo, P. Li, T. Wang, and Y. Liang, "Sanger: A Co-Design Framework for Enabling Sparse Attention Using Reconfigurable Architecture," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, p. 977–991.

[28] B. Lyu, M. Hamdi, Y. Yang, Y. Cao, Z. Yan, K. Li, S. Wen, and T. Huang, "Efficient Spectral Graph Convolutional Network Deployment on Memristive Crossbars," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 7, no. 2, pp. 415–425, 2023.

[29] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[30] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference*, 2010.

[31] Y. Niu, Z. Lu, H. Ji, S. Song, Z. Jin, and W. Liu, "TileSpGEMM: A Tiled Algorithm for Parallel Sparse General Matrix-Matrix Multiplication on GPUs," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, p. 90–106.

[32] M. Parger, M. Winter, D. Mlakar, and M. Steinberger, "SpECK: Accelerating GPU Sparse Matrix-Matrix Multiplication through Lightweight Analysis," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, p. 362–375.

[33] S. Park, J.-J. Kim, and J. Kung, "AutoRelax: HW-SW Co-Optimization for Efficient SpGEMM Operations With Automated Relaxation in Deep Learning," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 3, pp. 1428–1442, 2022.

[34] M. Saberi, R. Lotfi, K. Mafinezhad, and W. A. Serdijn, "Analysis of power consumption and linearity in capacitive digital-to-analog converters used in successive approximation ADCs," *IEEE Transactions on Circuits and Systems I*, vol. 58, no. 8, pp. 1736–1748, 2011.

[35] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, p. 475–486.

[36] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and Energy-Efficient in-DRAM Bulk Data Copy and Initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, p. 185–197.

[37] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with in-Situ Analog Arithmetic in Crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016, p. 14–26.

[38] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *2018 IEEE International Symposium on High Performance Computer Architecture*, 2018, pp. 531–543.

[39] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, and Y. Xie, "SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 570–583.

[40] J. Xu, H. Liu, Z. Duan, X. Liao, H. Jin, X. Yang, H. Li, C. Liu, F. Mao, and Y. Zhang, "ReHarvest: An ADC Resource-Harvesting Crossbar Architecture for ReRAM-Based DNN Accelerators," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 3, Sep. 2024.

[41] L. Yavits, A. Morad, and R. Ginosar, "Computer Architecture with Associative Processor Replacing Last-Level Cache and SIMD Accelerator," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 368–381, 2015.

[42] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, p. 687–701.

[43] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient Architecture for Sparse Matrix Multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 261–274.

[44] M. Zhou, W. Xu, J. Kang, and T. Rosing, "TransPIM: A Memory-based Acceleration via Software-Hardware Co-Design for Transformer," in *Proceedings of 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 1071–1085.

Hui-Ze Li received his Ph.D. degree from the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), in 2022. He is now working as a Research Fellow in School of Computing, National University of Singapore (NUS). His current research interests include computer architecture, emerging non-volatile memory, and processing in memory.

Dan Chen received the Ph.D. degree from the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), in 2024. He is now working as a Research Fellow in School of Computing, National University of Singapore (NUS). His research interests focus on processing-in-memory and graph neural network.

Tulika Mitra received a BE degree in computer science from Jadavpur University, Kolkata, India, in 1995, an ME degree in computer science from the Indian Institute of Science, Bengaluru, India, in 1997, and a Ph.D. degree from the State University of New York, Stony Brook, NY, USA, in 2000. She is currently Provost's Chair Professor of Computer Science at the School of Computing, National University of Singapore, Singapore. Her research interests include the design automation of embedded realtime systems with particular emphasis on software timing analysis/optimizations, application-specific processors, energy-efficient computing, and heterogeneous computing.