Journal of Systems Architecture 178 (2026) 103896

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

SparsePIM+: Accelerating SpMV on HBM-based PIM via logic-die accumulators with opportunistic TSV utilization

Taewoon Kang, Namhun Kim, Geonwoo Choi, Taeweon Suh, Gunjae Koo *

Department of Computer Science and Engineering, Korea University, Seoul, 02841, Republic of Korea

ARTICLE INFO

Keywords:
HBM-PIM
SpMV
Sparse matrix
Processing in memory
Logic-die accumulator

ABSTRACT

Sparse matrix–vector multiplication (SpMV) is a fundamental kernel in many applications, yet its performance is severely limited by irregular memory accesses and the accumulation of partial results on conventional architectures. While processing-in-memory (PIM) architectures mitigate data-movement overhead using high internal bandwidth, efficiently handling sparse data layouts and reducing partial results across distributed memory banks remain key challenges.

This paper proposes SparsePIM+, an HBM-based PIM architecture that accelerates SpMV through software–hardware co-design. SparsePIM+ introduces a software optimization that clusters matrix columns based on row-index similarity while balancing non-zero element distribution across bank groups to improve accumulation parallelism. It also employs a DRAM row-aligned sparse format (DRAF) that aligns sparse operands with DRAM row-buffer granularity to reduce indirect memory accesses and redundant data movement. On the hardware side, SparsePIM+ integrates bank-group accumulators (BGAs) for local accumulation within each bank group, where a bank group consists of four banks, and a logic-die global accumulator (GA) that aggregates partial results across bank groups via a lightweight arbitrator. The accumulated results are stored in an on-die buffer accessible by host memory controllers, enabling concurrent PIM execution and accumulation. Experimental results show that SparsePIM+ significantly improves accumulation efficiency and achieves an average speedup of 1.38× over prior HBM-based PIM designs for SpMV and 6.16× over an NVIDIA RTX 3080 GPU baseline, with a maximum speedup of up to 15.48×.

1. Introduction

Sparse matrix vector multiplication (SpMV) is a foundational kernel in scientific computing, graph analytics, circuit simulations, and machine learning workloads [1–9]. However, its performance is typically restricted by memory behavior rather than arithmetic computations since SpMV kernels generate highly irregular memory accesses, leading to poor compute-core utilization and inefficient use of the memory hierarchy. As a result, even highly optimized CPU/GPU implementations remain bottlenecked by off-chip bandwidth, access granularity, and index-driven accesses.

Processing-in-Memory (PIM) architectures, especially HBM-based PIM, have emerged as a promising approach that can mitigate data-movement costs in memory-intensive kernels by pushing compute cores closer to DRAM [10–12]. Prior studies demonstrate that in-memory accumulation and carefully orchestrated dataflows in HBM-based PIM can reduce data movement overhead and improve energy efficiency [10–15]. In particular, SparsePIM [15] introduced a DRAM row-aligned format (DRAF) together with bank-group accumulators (BGAs) to improve SpMV efficiency by reducing redundant row activations and enabling accumulation within each bank group (BG), where each BG consists of four banks.

Nevertheless, today's PIM approaches still face a fundamental scalability bottleneck when executing SpMV. Although BGAs can reduce partial results across banks within the same BG, the merged partial results must still be written back to the corresponding banks. These partial results are then fetched by the host memory controllers for global accumulation. Our analysis shows that this host-driven global accumulation incurs substantial data movement overhead, resulting in 1.56×–11.81× higher data fetch time than in-memory computation, thereby inflating end-to-end latency and energy consumption.

The prior research for SpMV executions on HBM-based PIM addresses several performance hurdles of SpMV in-memory computations by aligning sparse data with DRAM row-buffer granularity and performing local accumulation within BGs to curtail redundant activations and

* Corresponding author.
E-mail addresses: taewoon_kang@korea.ac.kr (T. Kang), skagns3571@korea.ac.kr (N. Kim), hnts03@korea.ac.kr (G. Choi), suhtw@korea.ac.kr (T. Suh), gunjaekoo@korea.ac.kr (G. Koo).

https://doi.org/10.1016/j.sysarc.2026.103896
Received 25 January 2026; Received in revised form 11 May 2026; Accepted 14 June 2026
Available online 19 June 2026

bank conflicts [15]. However, inter-BG accumulation remains an open problem that fundamentally limits performance and energy efficiency of SpMV on HBM-PIM, especially for matrices with heavy inter-bank row collisions. Another inefficiency arises from the fact that during PIM execution column commands do not utilize through-silicon vias (TSVs) for data movement. Instead, data transfers occur only between the DRAM subarray and the internal processing units. As a result, a substantial portion of TSV bandwidth remains idle throughout most PIM operations. Prior PIM architectures have not exploited these unused TSV cycles, leaving valuable vertical bandwidth underutilized during active computation.

This work builds upon SparsePIM [15] by extending its architectural and execution model to address remaining scalability bottlenecks. The prior work demonstrates that employing a DRAM row-aligned format (DRAF) together with bank-group-level accumulation through bank-group accumulators (BGAs) effectively reduces redundant row activations and improves in-memory execution efficiency. However, overall performance remains constrained by inefficient inter-BG partial-result handling and the underutilization of vertical bandwidth. In order to further tackle the performance hurdles of SpMV computations in HBM-based PIM, we propose SparsePIM+ in this work. SparsePIM+ preserves the core design principles of the prior research, namely, row-buffer-aware data compression and near-bank parallel execution. SparsePIM+ augments this framework with new logic-die-level computation and execution-flow mechanisms. Specifically, by exploiting global accumulators on the logic die of an HBM stack and an opportunistic TSV-aware execution scheme, SparsePIM+ reduces the dominant data movement cost for inter-BG reduction and enables scalable SpMV execution across all banks without reintroducing read amplification.

We evaluate SparsePIM+ using a modified DRAMSim3 [16] simulator. Our evaluation results show that SparsePIM+ achieves up to 15.48× speedup compared to SpMV kernels using the cuSPARSE library on a GPU. In addition, the proposed DRAM row-based compression format reduces memory usage by up to 29.82% compared to the conventional coordinate (COO) compression format. The computation engines in SparsePIM+ can efficiently parallelize the computations of non-zero elements with the proposed DRAM row-based data allocation format. We also estimate the power and area overhead of SparsePIM+. The estimated dynamic power consumption of a single BGA is 31.85 μW, which meets the thermal design power (TDP) requirements of the existing HBM-PIM.

The key contributions of SparsePIM+ are summarized as follows:

- We propose global accumulators (GAs) for performing inter-BG reduction on the logic die of an HBM stack. Beyond BGAs working within BGs, GAs ingest partial sums from all banks and perform parallel reduction using a merge/buffer-tree organization. By aligning streams on row indices (index-sorted merging), GAs eliminate the dominant DRAM re-read path for inter-bank merges, substantially reducing partial-result read time and off-chip energy.
- SparsePIM+ exploits underutilized TSVs for transferring partial results to GAs. Because PIM column commands do not occupy TSV resources, TSVs remain idle during subarray-level in-memory computation. We propose leveraging the underutilized vertical bandwidth to opportunistically forward partial sums from banks to GAs without modifying HBM timing or command protocols. This mechanism converts idle TSV cycles into effective reduction bandwidth, mitigating global-merge stalls and improving inter-bank accumulation efficiency.
- We propose software-assisted preprocessing and execution flow for SparsePIM+. Host-side preprocessing converts inputs to a DRAF variant tailored for PIM execution (row-index alignment and column grouping). Execution proceeds with BGA-based and GA-based inter-BG merging running in parallel without requiring protocol changes or timing assumptions in HBM.

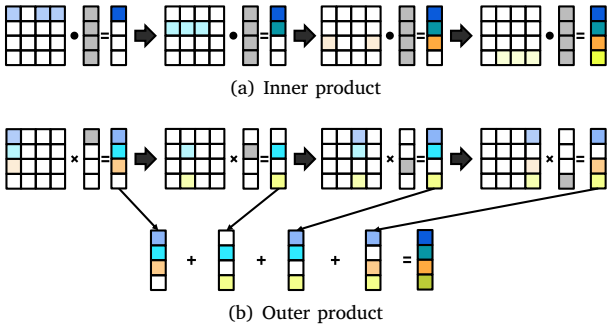


Fig. 1. Comparison of SpMV computation.

The remainder of this paper is organized as follows. Section 2 reviews SpMV formats, the baseline HBM-based PIM architecture, and relevant background for SpMV execution on PIM systems. Section 3 discusses prior PIM-based approaches proposed for accelerating SpMV. Section 4 analyzes the performance overhead of partial-result reduction and motivates the design of SparsePIM+. Section 5 presents the SparsePIM+ architecture and its execution flow. Section 6 describes the evaluation methodology and experimental results, including performance and hardware overhead analysis. Section 7 reviews additional related work on PIM architectures and SpMV accelerators. Finally, Section 8 concludes the paper.

2. Background

2.1. Sparse compression formats

Sparse compression formats are widely adopted to reduce the storage overhead of datasets that contain a large proportion of zero-valued elements. However, while these formats significantly decrease memory footprint, they often introduce non-trivial performance challenges when deployed on parallel processing architectures. In large-scale workloads dominated by sparse matrix vector multiplication (SpMV), sparse matrices typically include vast numbers of zero entries, making naïve dense representations highly inefficient.

To avoid unnecessary storage and computation on zero values, sparse matrices are commonly represented by storing only non-zero elements along with their corresponding indices using compression formats. Among these, coordinate (COO), compressed sparse row (CSR), and compressed sparse column (CSC) formats are the most widely used in modern applications [17]. COO explicitly records two-dimensional coordinates for each non-zero element, whereas CSR and CSC employ pointer arrays to indicate the starting positions of non-zero elements along rows or columns, respectively.

Although these representations are storage-efficient, they fundamentally rely on indirect memory accesses driven by index lookups. As a result, processing units must fetch non-zero values together with their indices to perform SpMV computations. This access pattern leads to highly irregular memory behaviors, which severely limit effective utilization of both compute resources and memory bandwidth in conventional processors [18–20].

To mitigate these inefficiencies, prior studies have proposed alternative compression formats tailored to specific hardware targets, such as GPUs [21–25] and domain-specific accelerators [26–29]. These approaches improve performance by reorganizing sparse data to enhance memory coalescing, reduce control divergence, or increase parallelism. Nevertheless, sparse compression formats that explicitly consider the architectural characteristics of PIM systems — particularly those based on two-dimensional DRAM cell arrays — remain relatively underexplored.

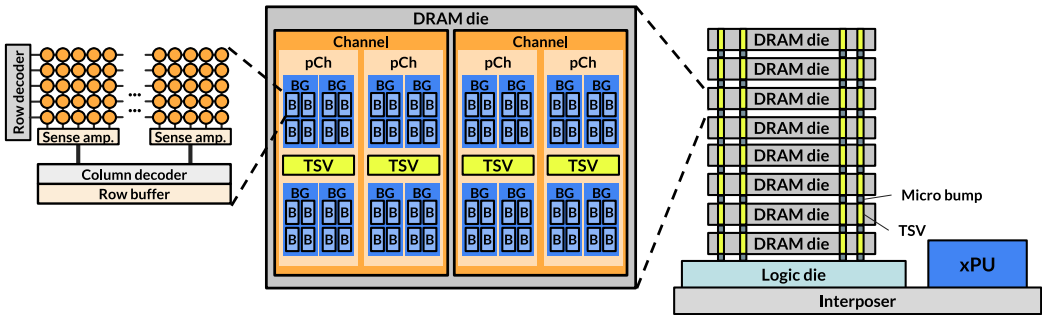


Fig. 2. HBM architecture and organization.

2.2. SpMV computation methods

SpMV can be implemented using two fundamentally different strategies: inner product and outer product, as illustrated in Fig. 1. The inner product approach computes each output element as a dot product between a matrix row and the input vector. In this scheme, partial products generated from non-zero elements within a single row are accumulated to produce one element of the result vector, as shown in Fig. 1(a).

For dense matrices, the inner product method can effectively exploit data-level parallelism and reuse input vector elements via local buffering. However, when applied to sparse matrices, this approach incurs substantial overhead due to index matching between sparse matrix entries and the input vector. Furthermore, because the number of non-zero elements (NZEs) varies across rows, the computational workload per output element becomes highly imbalanced. Such irregularity degrades performance on architectures that favor uniform and predictable execution patterns [30–32].

In contrast, the outer product approach decomposes SpMV into a sequence of vector–scalar multiplications between individual matrix columns and corresponding elements of the input vector, as shown in Fig. 1(b). Each multiplication generates a partial result vector indexed by row positions, and the final output vector is obtained by accumulating these partial results across all columns. This approach avoids explicit index matching during multiplication, as each operation simply scales a sparse column by a scalar input value.

While the outer product method enables simpler multiplication logic and immediate generation of partial results, it introduces additional challenges related to intermediate storage. Specifically, partial results associated with the same row index must be accumulated across different columns, requiring extra buffering and accumulation mechanisms. Therefore, efficient dataflows and storage management strategies are essential to control the overhead of intermediate data in outer-product-based SpMV implementations [13,33].

2.3. High bandwidth memory

High bandwidth memory (HBM) is a three-dimensional stacked memory technology designed to provide substantially higher bandwidth and improved energy efficiency compared to conventional DRAM architectures [34–45]. HBM achieves these benefits by vertically stacking multiple DRAM dies on top of a logic die and interconnecting them through TSVs, as illustrated in Fig. 2 [36,46]. The logic die, also referred to as the base die, contains peripheral components such as data buffers, I/O circuitry, and physical interfaces (PHYs), and serves as the communication interface between the stacked memory dies and external processors via a silicon interposer. Each stacked memory die functions similarly to a conventional DRAM chip and contains arrays of DRAM cells organized into banks.

As shown on the left side of Fig. 2, an HBM stack is hierarchically organized into multiple channels, each supporting 128-bit-wide data transfers. To increase memory-level parallelism, each channel is further divided into two independent 64-bit pseudo-channels (pCh). While the two pChs share the same physical data bus, they maintain separate command and address interfaces. Each pCh contains multiple bank groups (BG), and each BG consists of several banks that operate independently. Within a bank, data cells are arranged into rows, which define the fundamental access granularity. During a read operation, a row is activated, and its contents are transferred into the row buffer via sense amplifiers. Column selection logic then extracts the requested data from the sense amplifiers. In HBM2, each row typically spans 1 KB of data [35], making row-level activation a critical factor in memory access efficiency.

2.4. HBM-based PIM

Processing-in-memory (PIM) is an architectural paradigm that embeds computation capabilities directly within memory devices to alleviate the overhead associated with frequent data transfers between processors and memory. Recently, a commercial HBM-based PIM architecture, referred to as HBM-PIM (a.k.a., Aquabolt-XL [12], FIM-DRAM [10]), has been introduced [10–12]. As illustrated in Fig. 3, HBM-PIM integrates SIMD-style processing units and register files into the memory die itself.

To accommodate these processing elements, a portion of the DRAM rows within each bank is replaced by logic circuits on the PIM-enabled die. Consequently, the storage capacity of a PIM die is reduced compared to a conventional DRAM die. HBM-PIM operates in two modes: single-bank (SB) mode and all-bank (AB) mode. In SB mode, the HBM-PIM behaves similarly to standard DRAM, where commands target a specific bank for conventional read/write operations. In contrast, AB mode is designed to enable PIM execution by broadcasting memory commands to all banks simultaneously, effectively ignoring bank address fields. Under this mode, read and write commands — particularly column commands — are interpreted as triggers for PIM operations, causing the programmed PIM execution in command register files to be executed across all activated banks.

Once AB mode is enabled, PIM instructions are triggered by column commands and executed concurrently across banks. HBM-PIM provides a RISC-style instruction set that includes arithmetic, control, and data movement operations [11]. However, since the architecture is primarily optimized for dense matrix–vector workloads, it does not natively support indirect indexing operations, which are pervasive in sparse matrix processing.

3. Previous work

SpMV is a memory-intensive kernel due to its irregular memory access patterns originating from indirect indexing and sparse data

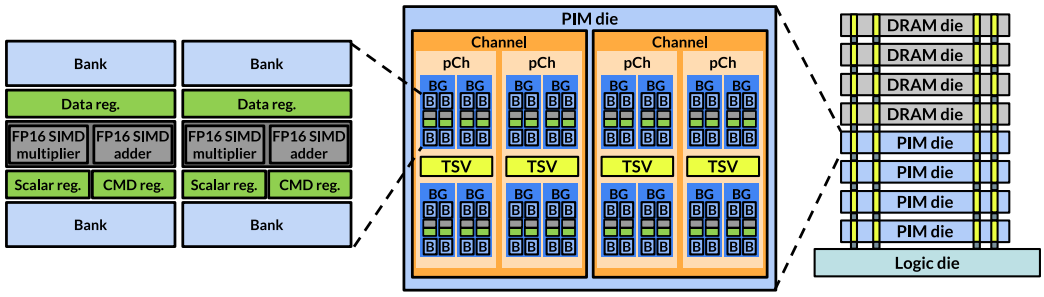


Fig. 3. HBM-PIM architecture.

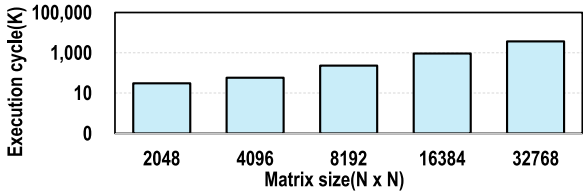
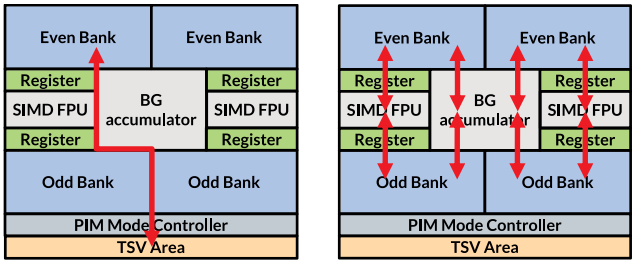


Fig. 4. Execution cycle of SpMV on HBM-PIM by the size of a sparse matrix.



(a) SB mode (DRAM mode) (b) AB mode (PIM mode)

Fig. 5. Data movement paths in SB mode and AB mode.

structures. These characteristics often limit performance by generating a large number of fine-grained and uncoalesced memory transactions. To address this challenge, several studies have explored PIM-based approaches that accelerate SpMV by exploiting the high internal bandwidth and massive parallelism available within memory systems.

SpaceA proposes a PIM architecture for accelerating SpMV on hybrid memory cube (HMC) [47]. SpaceA integrates a two-level content-addressable memory (CAM) hierarchy to accelerate cache tag matching inside processing engines and applies software-level optimizations to distribute non-zero elements evenly across memory banks. In addition, SpaceA maps sparse matrix row data to physical memory rows to improve row-level data locality and leverages internal data movement capabilities supported by HMC. However, since HMC is not a standardized memory technology, the applicability of SpaceA is limited to specific memory configurations.

pSyncPIM is an HBM-based PIM approach designed to accelerate SpMV and sparse triangular solve (SpTRSV) workloads [14]. pSyncPIM introduces a partially synchronous execution model that enables semi-independent bank-level operations, reducing idle cycles caused by irregular computation patterns. While pSyncPIM effectively mitigates inefficiencies stemming from fully synchronous execution in HBM-PIM, it does not address the indexing overheads imposed by conventional sparse matrix compression formats.

SpDRAM [48] presents a DRAM-based SpMV acceleration framework that exploits in-DRAM bit-serial operations [49]. By leveraging bitwise computation primitives inside DRAM, SpDRAM performs arithmetic operations without transferring data to external compute units. SpDRAM also proposes a data allocation strategy tailored for bit-serial execution. Nevertheless, fully exploiting bit-serial computation requires complex bit-level control and orchestration mechanisms, which can introduce additional design complexity.

4. Motivation

The performance overhead of SpMV kernels becomes increasingly pronounced as modern applications adopt large-scale sparse data structures. To reduce the storage cost associated with a large number of zero-valued elements, SpMV kernels typically rely on compressed sparse matrix formats. However, as discussed in Section 2.1, compressed representations store index metadata together with NZEs, introducing indirect memory accesses during computation. Such indirect indexing significantly degrades efficiency in both processing units and memory systems [50,51]. In addition, NZEs are often distributed unevenly across rows or columns in sparse matrices, resulting in substantial load imbalance during parallel execution. This imbalance is particularly evident in graph analytics and neural network workloads, where sparse matrices frequently exhibit power-law distributions in the number of NZEs per row or column. In such cases, the overall execution time of SpMV is dominated by a small subset of rows or columns containing a large number of NZEs, leading to severe performance degradation when rows or columns are processed in parallel [52,53].

Load imbalance poses an even greater challenge when SpMV kernels are executed on PIM-based architectures, where processing units are tightly coupled with individual banks or bank groups within a memory package. In these architectures, row or column data from large sparse matrices are distributed across multiple banks or bank groups, while inter-bank communication is typically limited. Consequently, non-uniform distributions of NZEs across banks lead to imbalanced workloads among bank-level processing engines, significantly reducing overall performance. To alleviate this issue, software-level optimizations are required to balance the distribution of NZEs across banks or bank groups associated with processing units. Such techniques aim to mitigate workload skew but cannot fully eliminate the fundamental challenges arising from irregular sparsity patterns.

To execute SpMV efficiently on PIM platforms, specialized architectural support is required to handle the irregular access patterns and sparsity inherent in sparse matrix computations. Nevertheless, existing PIM solutions such as HBM-PIM and GDDR6-AiM are primarily optimized for general matrix–vector operations rather than large-scale sparse workloads. To examine the performance implications of executing SpMV on such architectures, we evaluate the execution latency of SpMV kernels represented in a dense matrix format using a DRAM-Sim3-based HBM-PIM simulator [16], as illustrated in Fig. 4. Our results show that the execution cycles of SpMV increase rapidly with matrix size, indicating poor scalability. This analysis highlights that current HBM-PIM architectures exhibit significant inefficiencies when processing large-scale sparse matrices.

Another overlooked inefficiency in existing HBM-based PIM architectures lies in the underutilization of TSVs during PIM execution. In prior HBM-based PIM designs, including HBM-PIM, SparsePIM, and pSyncPIM, in-memory computation is performed using only the internal DRAM bandwidth between the row buffer and the near-bank processing units. During PIM execution, data does not traverse the TSV interface, and therefore, no off-die data movement occurs through the TSV links. Fig. 5 illustrates the data movement paths in SparsePIM under conventional DRAM mode and all-bank (AB) PIM mode. In DRAM mode, data flows vertically through the TSV area between DRAM dies and the logic die to serve host read/write requests. In contrast, under AB mode for PIM execution, arithmetic operations are performed within bank-local processing units, and data movement is confined to intra-bank and intra-BG paths. As a result, the TSV interface is not actively utilized during most PIM computation phases.

This lack of TSV utilization represents a missed opportunity for overlapping data reduction with computation. While bank-level SIMD units perform multiplications and local accumulation, the vertical bandwidth provided by TSVs remains unused. Consequently, inter-bank partial-result reduction must either wait for computation to complete or rely on additional DRAM read/write operations, increasing latency and energy overhead. This observation motivates the need for an architecture that can exploit otherwise idle TSV bandwidth during PIM execution.

5. SparsePIM+

In this paper, we propose *SparsePIM+*, an HBM-based PIM architecture designed to execute large-scale SpMV efficiently. As discussed in Section 4, practical PIM execution of SpMV requires (i) software support to mitigate load imbalance across banks/bank groups and (ii) hardware support to reduce the overhead of sparse indexing and partial-result handling. SparsePIM+ follows the outer-product formulation of SpMV and combines three key components. First, SparsePIM+ applies a lightweight software preprocessing step to partition matrix columns such that NZEs are distributed evenly across bank groups while preserving row-index similarity within each group. Second, SparsePIM+ adopts a DRAM-aware compression format, *DRAM row-aligned format (DRAF)*, which aligns sparse data layout with the DRAM row buffer to minimize inefficient indirect accesses. Third, SparsePIM+ provides in-memory partial-result accumulation using a two-level hierarchy: (i) *bank-group accumulators (BGAs)* for intra-BG merging and (ii) a *global accumulator (GA)* for inter-BG merging. A lightweight *bank-side arbitrator* coordinates local transfers of partial results to GAs, enabling efficient reduction across bank groups without introducing a centralized controller.

SparsePIM+ performs SpMV operations using an outer-product approach, as described in Section 2.2. Execution follows the hierarchy of an HBM stack (see Section 2.3). A SIMD floating-point unit (FPU) associated with two neighboring banks (i.e., even/odd banks within a bank group) computes vector–scalar multiplications using NZEs from a matrix column and a single input-vector element. Within each bank group, a BGA merges partial results produced by the banks within the group. To further reduce the performance overhead of reduction across multiple bank groups, SparsePIM+ forwards selected partial sums to GA through TSV links during available transfer windows to enable inter-BG accumulations.

5.1. Software optimizations

SparsePIM+ executes vector–scalar multiplications for the outer-product SpMV using processing engines distributed across multiple banks. Hence, as described in Section 4, NZEs should be allocated evenly across banks or bank groups to avoid load imbalancing. In addition, SparsePIM+ benefits when NZEs that share the same row indices are placed within the same bank group, because intra-BG partial results can be merged locally by a local BGA. Accordingly, SparsePIM+ employs a software preprocessing step that jointly targets (i) load balancing across bank groups and (ii) increased row-index similarity within each group.

Fig. 6 illustrates the column-grouping procedure using a 10×10 sparse matrix, where colored blocks represent NZEs. The preprocessing first selects a reference (centroid) column (e.g., *column 0* in the figure). For each remaining column, it computes *row-index similarity*, defined as the number of NZEs whose row indices overlap with those of the reference column. In the example, *column 1* and *column 2* have similarities of 1 and 3, respectively, with respect to *column 0*. As a result, *column 2* is grouped with the centroid column and assigned to the same bank group. Grouping columns with higher row-index similarity increases the likelihood that partial results generated by the outer-product computation share identical row indices, thereby improving the effectiveness of BGA accumulation.

We use K-means as the base clustering algorithm for sparse matrices [54] due to its relatively low preprocessing cost compared to more complex clustering methods [55–57]. Since SparsePIM+ uses a fixed number of bank groups for in-memory computations, the number of clusters is set to the number of bank groups on PIM dies. However, naïve K-means does not guarantee a balanced distribution of NZEs across clusters because columns can have highly varying numbers of NZEs. To address this limitation, SparsePIM+ applies a two-stage preprocessing flow consisting of *bounded cap K-means* followed by a *refinement* step.

Algorithm 1 Bounded cap K-means

Input: col.nz[] (each c has nNZE(c), fmap(c)), k, maxIter, delta
Output: col.bg[] (initial assignment)

$totalN = \sum_c nNZE(c)$
$minCap = (totalN/k) \cdot (1 - delta)$
$maxCap = (totalN/k) \cdot (1 + delta)$
Initialize centroids[] ← random columns
for each column **c** in col.nz[] **do**
  for each **bgIdx** $= 0 \ldots k-1$ **do**
    if $nNZE(col.bg[\textbf{bgIdx}]) + nNZE(c) \le maxCap$ **then**
      $cost \leftarrow distance(fmap(c),\ centroids[\textbf{bgIdx}])$
      if $nNZE(bg) < minCap$ **then**
        $cost \leftarrow 0.5 \times cost$ (discount cost)
      end if
    end if
    assign **c** to the col.bg[**bgIdx**] with the smallest cost
  end for
  if no suitable cluster found **then**
    assign *c* to the smallest $nNZE(col.bg[\textbf{bgIdx}])$
  end if
end for
Update centroids (avg. each cluster's fmap(c))
return col.bg[]

Algorithm 1 describes the first stage, *bounded cap K-means*, of the software optimization. This algorithm extends standard K-means by explicitly constraining the number of NZEs assigned to each cluster using upper and lower bounds, defined by *maxCap* and *minCap*. During assignment, a column is considered for a cluster only if adding its NZEs does not exceed *maxCap*. To prevent clusters from remaining underfilled, a cost discount is applied when the current NZE count of a cluster is below *minCap*. This stage produces an initial assignment that balances row-index similarity and NZE distribution across bank groups.

Algorithm 2 presents the second stage, *refinement*, which further improves load balancing across bank groups. While bounded-cap K-means enforces capacity constraints, residual imbalance may still remain due to the discrete nature of columns. The refinement step identifies a cluster with excessive NZEs (*big cluster*) and a cluster with insufficient NZEs (*small cluster*), and then attempts to migrate a column from the former to the latter. A migration is allowed only when the resulting

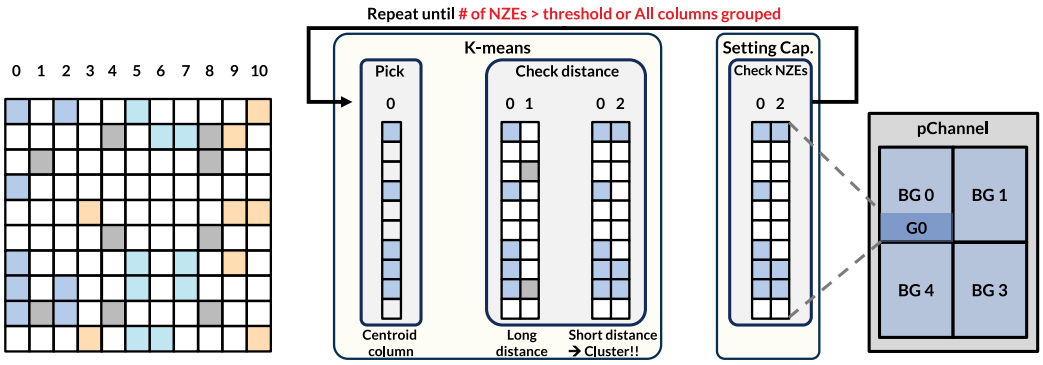


Fig. 6. Simplified execution flow of grouping columns based on row index similarity.

Algorithm 2 Refinement

Input: col.bg[], col.nz[], refineIter, th
Output: col.bg[] (refined assignment)
for $iter = 1 \ldots refineIter$ **do**
  Identify **"big cluster"** and **"small cluster"** in col.bg[]
  for each column **c do**
    if moving **c** from **big cluster** to **small cluster and** distance change < th **then**
      Move **c** to **small cluster**
    end if
  end for
  if no improvement **then**
    break
  end if
end for
return col.bg[]

increase in distance, reflecting loss of row-index similarity, is smaller than a predefined threshold *th*. This process is repeated for a bounded number of iterations, yielding a final assignment that improves load balance while preserving row-index similarity.

We summarize the two steps of the software optimization as follows:

Bounded cap K-means: The clustering objective is to group columns with high row-index similarity while keeping each cluster within the NZE capacity bounds. SparsePIM+ computes distances using feature maps and iteratively updates centroid columns. Unlike traditional K-means, bounded-cap K-means enforces NZE capacity constraints (*minCap*, *maxCap*) to prevent workload skew across bank groups. The process terminates when the maximum number of iterations is reached or when the centroids stabilize.

Refinement: The refinement step further reduces residual imbalance across clusters. In each iteration, SparsePIM+ identifies a *big cluster* (above-average NZEs) and a *small cluster* (below-average NZEs), then attempts to move columns from the big to the small cluster if the similarity loss is within *th*. The process continues until *refineIter* is reached or no further improvements are observed.

We compare the complexity of our software optimization approach with SpaceA [47]. SpaceA clusters rows to enable data reuse by organizing partitioned rows such that their internal NZEs share similar column indices, and its preprocessing includes assigning rows to logical PEs and mapping logical PEs to physical PEs. The first stage dominates its execution time, resulting in $O(N_{PE} \times N_{NZE} \times \log N_{NZE})$ complexity. In contrast, SparsePIM+ assigns columns during bounded-cap K-means by comparing n data points (proportional to N_{NZE}), over k clusters with average per-column NZE dimension d, yielding $O(nkd)$ per iteration and $O(nkdt)$ for t iterations. The refinement step evaluates clusters and potentially reassigns columns for r iterations, resulting in $O(nr)$. Since t and r are typically much smaller than n, the overall cost is dominated by $O(nkd)$. Depending on sparsity distributions (i.e., d vs. $\log(N_{NZE})$), the two approaches can exhibit different preprocessing overhead trends.

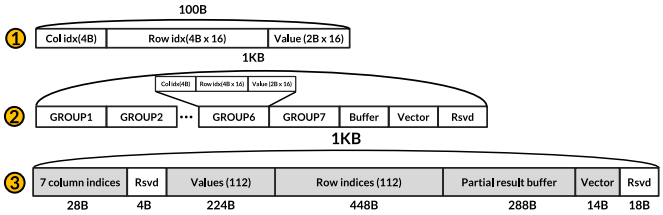


Fig. 7. DRAM row-aligned format for a DRAM row.

5.1.1. Parameter sensitivity

We analyze the sensitivity of the preprocessing parameters, including the capacity slack parameter *delta*, the number of K-means iterations, and the number of refinement iterations. The results reveal consistent trends across the evaluated workloads. Decreasing *delta* tightens the NZE capacity bounds of clusters and generally improves load balance by reducing the standard deviation of NZEs across bank groups. In contrast, larger *delta* values allow greater flexibility during clustering and tend to increase row-index similarity within clusters. The number of K-means iterations mainly affects clustering quality: increasing the iteration count slightly improves row-index similarity but provides limited benefit to load balance. On the other hand, the number of refinement iterations has the most direct impact on load balancing, as additional refinement steps progressively migrate columns from overloaded clusters to underutilized ones. Overall, these observations indicate that K-means iterations primarily control clustering quality, refinement iterations regulate the distribution of NZEs across clusters, and *delta* acts as a tuning parameter that adjusts the trade-off between row-index similarity and load balance.

5.2. DRAM row-aligned format

SparsePIM+ adopts a column-oriented sparse compression format aligned with DRAM row organization, referred to as DRAF (DRAM row-aligned format). Conventional sparse formats rely on index metadata to locate non-zero elements (NZEs), which introduces indirect memory accesses that are inefficient for DRAM's two-dimensional cell organization. To reduce this overhead under an outer-product formulation, DRAF organizes column data and the required indices so that PIM operations can directly consume operands from the DRAM row buffer with minimal data movement.

Fig. 7 illustrates the construction and organization of DRAF within a DRAM row, where bracketed numbers indicate the number of elements stored in each field. We assume FP16 operands (2 bytes) for matrix

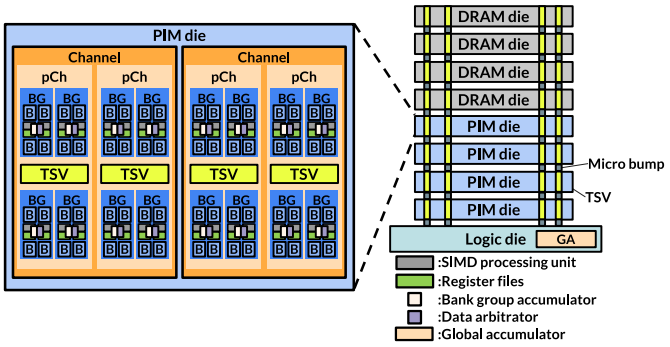


Fig. 8. Overview of proposed SparsePIM+.

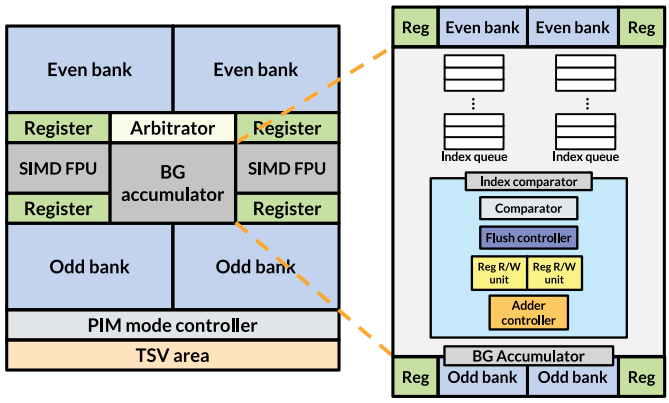


Fig. 9. Hardware architecture of processing units within a bank group.

and vector values and 4-byte entries for row and column indices. Since SparsePIM+ targets HBM2, DRAF is constructed to fit within a 1 KB DRAM page [35]. The construction process begins by selecting a set of matrix columns assigned to the same bank group. For each selected column, non-zero elements are grouped in blocks of up to 16 values while preserving sorted row-index order, and the corresponding row indices are packed alongside the values.

As shown in the figure, DRAF packs multiple data fields into a fixed layout within a DRAM row, including column indices, NZE values, row indices, a partial result buffer, and the corresponding input vector elements. Under the assumed configuration, a single DRAM row can encapsulate up to seven groups of matrix columns, each group containing up to 16 NZEs, resulting in a maximum of 112 values and 112 row indices per DRAM row. If a column contains more than 16 NZEs, the remaining elements are placed into an additional column data group that reuses the same column index and follows the same packing procedure.

DRAF also reserves a dedicated field for storing partial results, referred to as the partial result buffer in Fig. 7 at the very bottom. Intermediate partial results generated during PIM execution are temporarily stored in the DRAM row buffer, eliminating the need for separate on-die storage. Because row indices for NZEs are already included in DRAF, no additional row-index metadata is required for partial results. Finally, the corresponding input vector elements are stored in the vector field. Since each DRAM row contains seven matrix columns, seven vector elements are stored accordingly. Reserved fields (Rsvd) are included to satisfy the 32-byte DRAM column access granularity. In particular, after packing the seven column indices and the corresponding vector elements, reserved space is inserted to align each logical field to a 32-byte boundary. By enforcing this 32B alignment, DRAF preserves the native DRAM column access granularity, enabling each column read to retrieve the required operands in a single DRAM access without additional column commands or misaligned fetch overhead.

An additional advantage of DRAF is that it enables efficient reuse of the sparse matrix layout across multiple SpMV executions. Once the DRAF-formatted rows are constructed and stored in memory, subsequent SpMV runs with a different input vector do not require reconstructing the sparse matrix data. The indices and NZE fields remain unchanged, and only the vector field needs to be updated. Specifically, by reading the column indices field and updating the corresponding vector elements in the vector field, SparsePIM+ can immediately initiate a new SpMV computation using the same DRAF layout. This design allows repeated SpMV operations to be executed efficiently without modifying the stored sparse matrix structure.

5.3. Hardware architecture

SparsePIM+ includes processing engines for vector–scalar multiplication and bank-group accumulator (BGA) for accumulation of partial results to support outer-product-based SpMV computation. Similar to existing HBM-PIM designs [10–12], SparsePIM+ includes register files and a 16-lane SIMD FPU for FP16 arithmetic. The SIMD width matches the 32-byte DRAM column access granularity. As shown in Fig. 8, a single SIMD FPU is associated with two banks (even/odd banks) within a bank group. SparsePIM+ stores microkernel-generated PIM instructions in command registers and triggers execution using standard DRAM commands to guarantee compatibility with conventional HBM command sequences.

5.3.1. PIM instructions

SparsePIM+ adds two new PIM instructions, *BMOV* and *BACC*, building on the baseline HBM-PIM instruction set [11]. Table 1 presents their formats and target components. *BMOV* moves a selected element from the *vector* field of the row buffer (Fig. 7) into the scalar register to enable vector–scalar multiplications between a matrix column and a single input-vector element. *BACC* initiates partial-result accumulation by transferring row indices (from the *row indices* field in the row buffer) and corresponding partial results (from the data registers) into the BGA queues. Once the queues are populated, the BGA begins merging entries that share identical row indices.

5.3.2. Bank group accumulator (BGA)

The right side of Fig. 9 depicts the hardware architecture of BGA. BGA aggregates partial results that share identical row indices within a bank group to reduce the number of partial results and improve write-back efficiency. Since the banks within the same bank group share a single BGA, partial results computed by those banks can be merged locally. In addition, each pseudo-channel (pCh) is equipped with a dedicated global accumulator (GA), resulting in a total of 16 GAs in an HBM stack. The software preprocessing in Section 5.1 increases the likelihood of row-index matches within a bank group, improving the BGA hit rate.

The BGA includes two index queues, where each entry contains a row index and its associated partial result. When executing *BACC*, row indices from the *row indices* field in the row buffer and partial results from the data registers are enqueued. Since a DRAM column access is 32 bytes, a single *BACC* fills eight queue entries. In this paper, we set the depth of each index queue to 16, considering the power constraint of an HBM stack.

The BGA performs accumulation using an index comparator unit consisting of a comparator, a flush controller, register read/write units, and an adder controller. To merge partial results, the comparator inspects the head entries of the queues and compares their row indices [58–61]. If the indices match, the corresponding partial results are accumulated, one data register is updated with the merged value, and the other is cleared. If the indices differ, the entry with the smaller

Table 1
PIM instructions added in SparsePIM+.

(a) SparsePIM instructions

Operation	Source A	Source B	Destination
BACC	Row buffer	Data register	BGA
BMOV	Row buffer	-	Scalar register

(b) Instruction format

	31	30	29	28	27	26	25	24	23	22	21	...	8	7	6	5	4	3	2	1	0
BACC	OPCODE				Unused			SRC0			Unused			SRC0 Idx				Unused			
BMOV	OPCODE				Unused																

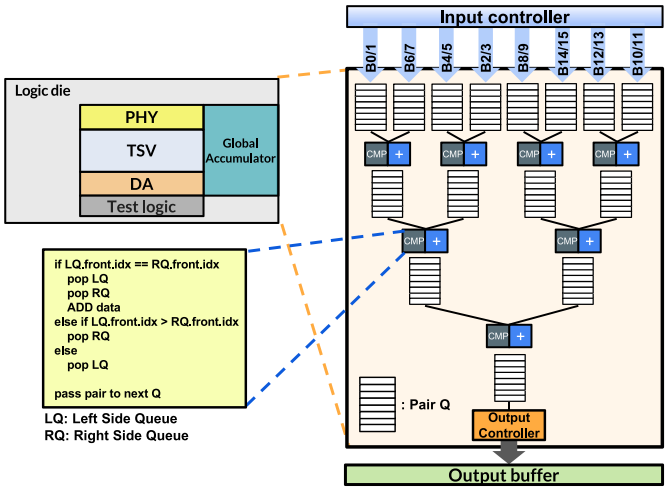


Fig. 10. Global accumulator.

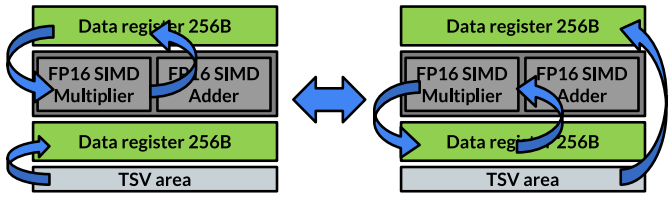


Fig. 11. Data register double buffering scheme.

row index is dequeued. This process repeats until the queues are empty. To preserve sorted ordering and prevent queue overflow, the flush controller clears the queues when necessary (e.g., when a *BACC* arrives while a queue exceeds a validity threshold). The adder controller leverages otherwise-idle SIMD adders during outer-product execution by routing accumulation operands through the SIMD addition datapath.

5.3.3. Global accumulator (GA)

While BGAs reduce partial-result traffic within each bank group, their accumulation scope is limited to intra-BG partial results. In large sparse matrices, partial results often span multiple bank groups, requiring additional reduction steps beyond bank-group boundaries. To address this limitation, SparsePIM+ employs a *global accumulator (GA)* located in the logic die, as illustrated in Fig. 10. GA performs inter-BG accumulation by collecting partial sums generated by multiple BGAs and merging them according to their row indices using a merge-based accumulation structure.

As shown in Fig. 10, partial results forwarded from different bank groups enter the GA through TSV interfaces and are processed by a hierarchical comparison-and-accumulation pipeline. The GA compares row indices of incoming partial sums and incrementally merges matching entries while propagating unmatched entries to subsequent stages. To minimize interference with ongoing bank-level computation, SparsePIM+ schedules partial-result transfers to the GA by exploiting available TSV transfer windows. This design improves TSV utilization and enables efficient inter-BG accumulation without stalling bank-local SpMV computations.

5.3.4. Bank-side arbitrator

SparsePIM+ also places a lightweight *bank-side arbitrator* adjacent to each bank group to coordinate partial-result forwarding to the GA. The arbitrator serializes competing local requests (e.g., BGA-to-GA transfers vs. other bank-group activities that share the same interface resources) and ensures conflict-free partial-result transfers. Unlike centralized controllers, the arbitrator operates locally with minimal control state, preserving a simple execution model while enabling efficient inter-BG accumulation.

5.4. Execution flow

We now describe the overall execution flow of SparsePIM+. Initially, an HBM stack operates in single-bank (SB) mode to function as a conventional memory device. SparsePIM+ first applies software preprocessing to (i) partition matrix columns into bank groups and (ii) balance the NZE distribution across bank groups, as described in Section 5.1. Next, SparsePIM+ compresses the sparse matrix and the input vector using the DRAF format (Section 5.2) and stores the formatted data into target bank groups in the HBM stack.

To perform PIM operations, SparsePIM+ switches the HBM stack to all-bank (AB) mode. It then programs the compiled PIM instructions into the command registers of the bank groups and triggers execution using standard DRAM commands. Before computation, the DRAM rows formatted in DRAF are activated (via ACT commands) so that the row-buffer contents, including column indices, non-zero elements, row indices, and vector elements, become accessible to the in-memory processing units. Because AB mode synchronizes bank-group operations, PIM instructions across the stack proceed in lockstep while leveraging bank-level parallelism.

SparsePIM+ performs outer-product vector–scalar multiplication as follows. Using *BMOV*, a bank loads one element from the *vector* field in the row buffer into the scalar register. Then, the SIMD multipliers compute vector–scalar products between the scalar and the NZEs in the *NZEs* field, producing 16 partial results that are stored in data registers.

Next, SparsePIM+ initiates intra-BG accumulation via *BACC*. Row index values in the *row indices* field are transferred to the BGA queues together with the corresponding partial results from the data registers. The BGA merges partial results that share identical row indices and writes merged values back to the data registers to reduce the number of partial results produced within the bank group.

After intra-BG accumulation, SparsePIM+ manages the movement of partial results using a double-buffered data register organization, as illustrated in Fig. 11. Each bank group maintains two data registers that alternate their roles between computation and data transfer. While one data register is actively used as the operand storage for SIMD execution, the other data register temporarily holds previously

Table 2
Configurations of SparsePIM+ and baseline.

Component	Configuration
SparsePIM+	
No. of SIMD FPU/BG	2
No. of BGA/BG	1
No. of total GA	16
Memory type	HBM2
No. of pCh	16
No. of BG/pCh	4
No. of banks/BG	4
No. of rows	16,384
No. of columns	32
Total capacity	6GB
Clock frequency	1 GHz
Timing parameters	$t_{CCDS} = 1$, $t_{CCDL} = 2$, $t_{RAS} = 34$, $t_{RP} = 14$, $t_{RCDRD/WR} = 14$, $t_{RRDS} = 4$, $t_{RRDL} = 6$, $t_{RL} = 14$
Baseline (NVIDIA RTX 3080)	
No. of CUDA cores	8704
Clock frequency	1440 MHz/1710 MHz
Memory bandwidth	760.3 GB/s (peak bandwidth)
Device memory	10GB GDDR6X

computed partial results and streams them through the TSV interface. Once the TSV transfer completes, the roles of the two data registers are swapped. This ping-pong switching enables continuous SIMD execution while overlapping TSV data movement, effectively hiding TSV transfer latency.

To further perform reduction across multiple bank groups, SparsePIM+ forwards selected partial sums to the GA for inter-BG accumulation. Specifically, the arbitrator grants TSV access in a round-robin manner across banks within a pseudo-channel, sequentially allowing Bank 0 (bank0 in BG0) through Bank 15 (bank3 in BG3) to transmit partial sums from their bank-side registers to the GA. This scheduling ensures fair and orderly utilization of TSV bandwidth while preventing contention among banks. The transfer is synchronized with the DRAM read command, such that partial sums are forwarded during column read operations without introducing additional command overhead. The GA merges incoming partial sums by row index and stores the accumulated results in an on-die *output buffer* located in the logic die alongside the GA. Since the output buffer is directly accessible by the host, the host can retrieve the accumulated results while PIM execution continues. This design enables overlap between in-memory computation (i.e., multiplication and local accumulation) and accumulation of the remaining partial sums (i.e., global accumulation) to effectively reduce end-to-end execution latency.

SparsePIM+ repeats the vector–scalar multiplication and accumulation flow for the column groups encapsulated in each DRAF row (up to seven columns per DRAM row under the assumed configuration). This hierarchical reduction (i.e., BGA for intra-BG merging and GA for inter-BG merging) significantly cuts down the volume of partial results transferred across the memory hierarchy. By overlapping SIMD computation, TSV transfers, and host-side consumption of GA outputs, SparsePIM+ further improves overall execution efficiency while maintaining compatibility with the HBM command protocol.

6. Evaluation

In order to evaluate SparsePIM+, we use a cycle-accurate simulator derived from DRAMSim3 [16]. The detailed system configuration is listed in Table 2. To remain consistent with the thermal design power (TDP) and logic-area assumptions adopted in prior HBM-PIM designs [10–12], processing logic is integrated into only four dies, while the remaining four dies operate without logic circuits. Namely, we assume that the HBM-based PIM for SparsePIM+ includes four PIM dies out of eight dies of an HBM stack. The simulator is extended to precisely account for execution cycles consumed by both memory commands and in-memory computation.

Table 3
Sparse matrix workloads.

ID	Workload	Domain	Size	Ratio of NZEs
w1	rma10	CFD	46,835	1.082E–03
w2	pdb1HYS	CB	36,417	1.652E–03
w3	cranksegs_2	SE	63,838	1.744E–03
w4	pwtk	SE	217,918	1.248E–04
w5	xenon2	MS	157,464	1.559E–04
w6	shipsec1	SE	140,874	2.004E–04
w7	lhr71	Chem	70,304	3.092E–04
w8	ohne2	SDS	181,343	3.364E–04
w9	consph	SE	83,334	4.387E–04
w10	ct20stif	SE	52,329	5.023E–04
w11	bcsstk32	SE	44,609	5.174E–04
w12	cant	SE	62,451	5.218E–04
w13	Stanford	WGA	281,903	2.910E–05
w14	soc-sign-epinions	SNA	131,828	4.841E–05
w15	webbase-1M	WGA	1,000,005	3.106E–06

To evaluate SpMV performance, we implement a micro-kernel in assembly using the custom instruction set described in Section 5.3.1. The kernel is loaded into the command register via memory write commands, allowing the simulator to faithfully model instruction fetch and execution behavior. As each vector–scalar multiplication generates 16 partial results per *MUL* instruction, the kernel issues two consecutive *BACC* instructions, each accumulating eight row indices of matrix from the DRAM row buffer.

For comparison, we evaluate SpMV execution on NVIDIA RTX 3080 GPU using cuSPARSE [62]. We chose RTX 3080 to ensure direct comparability with the prior study, pSyncPIM [14], which adopts the same GPU platform. The specification of the RTX 3080 is also listed in Table 2. GPU execution time is measured using *cudaEvent*, capturing the elapsed time from host-to-device data transfer through kernel execution to host-side result retrieval. Each experiment is repeated five times on a real system, and the reported performance corresponds to the average execution time to mitigate run-to-run variability. For SparsePIM+, we measure the end-to-end execution time of an SpMV kernel, including kernel programming, PIM initialization, in-memory computations, and result retrieval at the host.

To estimate the area cost and power consumption of SparsePIM+, we implement the processing units and buffers of SparsePIM+ in SystemVerilog. We synthesize the RTL model of SparsePIM+ using Synopsys Design Compiler. Although modern DRAM processes such as 1ynm, 1znm, and 1anm employ sub-14 nm technology [63,64], we adopt the SAED 14 nm FinFET process for hardware evaluation to ensure consistency with prior PIM studies and to provide a conservative estimation of area and power overheads.

6.1. Workload

Table 3 lists the sparse matrix datasets used in our evaluation. Each matrix is characterized by its application domain, matrix dimension (size), and the ratio of non-zero elements (ratio of NZEs), defined as the number of non-zero elements divided by the total number of matrix elements. For brevity, application domains are abbreviated as follows: SE for structural engineering, Chem for chemical process simulation, WGA for web graph analysis, CB for computational biology, SNA for social network analysis, CFD for computational fluid dynamics, SDS for semiconductor device simulation, and MS for material science. Using the datasets, we evaluate the performance of the proposed architecture and analyze the impact of software preprocessing and the DRAF compression format on SpMV execution.

SparsePIM+ primarily targets workloads with a low ratio of non-zero elements (NZEs), where the proposed sparse-aware data layout and accumulation mechanisms can effectively reduce indirect accesses and

Table 4
Effective bytes per non-zero element (NZE) of sparse compression formats.

Workload	NNZ	COO	CSR/CSC	DRAF
w1	2,374,001	10.00	6.08	7.38
w2	2,190,591	10.00	6.07	7.39
w3	7,106,348	10.00	6.04	7.02
w4	5,926,171	10.00	6.15	7.79
w5	3,866,688	10.00	6.16	8.12
w6	3,977,139	10.00	6.14	8.13
w7	1,528,092	10.00	6.18	10.62
w8	11,063,545	10.00	6.07	7.59
w9	3,046,907	10.00	6.11	7.97
w10	1,375,396	10.00	6.15	8.36
w11	1,029,655	10.00	6.17	8.54
w12	2,034,917	10.00	6.12	8.30
w13	2,312,497	10.00	6.49	15.31
w14	841,372	10.00	6.63	15.00
w15	3,105,536	10.00	7.29	37.23

improve computational efficiency. In such workloads, sparse matrices contain a large number of zero elements, and organizing only the NZEs in a DRAM-aware format allows the architecture to exploit row-buffer locality while reducing irregular memory accesses.

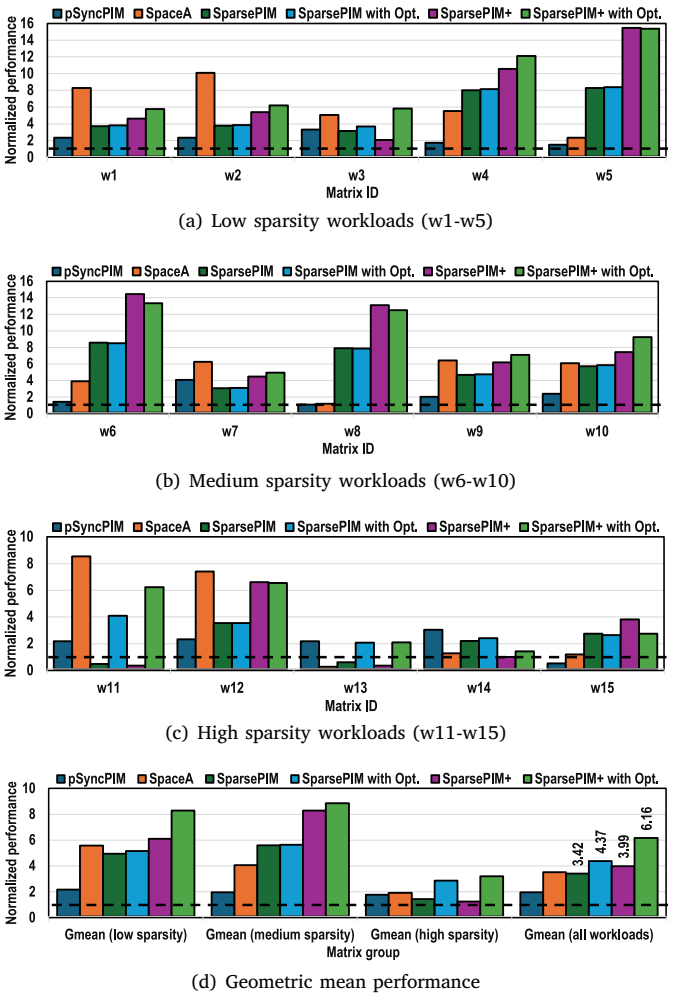
When the ratio of NZEs in a matrix becomes relatively high (i.e., the matrix is relatively dense), the benefit of sparse compression becomes limited because the overhead of storing indices and managing sparse structures can outweigh the reduction in memory footprint. In such cases, the matrix can be processed using the conventional dense matrix–vector multiplication (GEMV) execution model supported by existing HBM-PIM architectures. Specifically, the matrix can be stored in a dense layout and executed using the baseline PIM instructions without relying on the DRAF representation. This design allows SparsePIM+ to remain compatible with existing HBM-PIM execution models while focusing on accelerating workloads with a low ratio of NZEs.

In order to further quantify the compression efficiency of each compression data format, we compute the *effective bytes per NZE* as shown in Table 4. We compare the effective bytes/NZE for COO, CSR/CSC, and DRAF. For the COO compression format, each NZE stores a 4-byte row index, a 4-byte column index, and a 2-byte FP16 data, resulting in 10 bytes/NZE. The CSR/CSC formats store a single row/column index for NZEs that share the same row/column, thus the effective bytes/NZE of CSR/CSC is obviously lower compared to COO. For DRAF, we compute effective bytes/NZE using the memory footprint excluding the partial-result buffer and vector fields, while still including row-aligned *unused* space. Note that DRAF stores the fixed number (i.e. 112) of elements as explained in Section 5.2. Across the evaluated workloads, CSR/CSC exhibits 6.04–7.29 bytes/NZE, while DRAF requires 7.02–37.23 bytes/NZE.

For extremely sparse workloads, DRAF exhibits high effective bytes/NZE since many columns cannot fully utilize DRAF's packing granularity (i.e. the reserved space for NZEs per column). In particular, DRAF requires 15.31 bytes/NZE for Stanford, 15.00 bytes/NZE for soc-sign-epinions, and 37.23 bytes/NZE for webbase-1M. These results reveal that DRAF does not always minimize storage footprint for extremely sparse matrices. However, DRAF provides DRAM-row-aligned and direct accesses for PIM operations instead of irregular index-driven indirect accesses created by the existing sparse compression formats. Consequently, DRAF trades the efficiency of data accesses and PIM operations with the storage efficiency of the existing index-based sparse compression formats.

6.2. Performance

Fig. 12 presents the performance of the evaluated systems, normalized to the GPU baseline implemented using cuSPARSE on an NVIDIA RTX 3080. Figs. 12(a)–(c) report normalized performance for matrices categorized by the ratio of NZEs (low, medium, and high sparsity



(a) Low sparsity workloads (w1-w5)

(b) Medium sparsity workloads (w6-w10)

(c) High sparsity workloads (w11-w15)

(d) Geometric mean performance

Fig. 12. Performance comparison (categorized by the ratio of NZEs).

groups), while Fig. 12(d) summarizes the geometric mean across the groups. We compare SparsePIM+ with two prior studies, SpaceA and pSyncPIM. SpaceA accelerates SpMV kernels using an HMC-based PIM architecture with a two-level CAM hierarchy [47]. pSyncPIM is an HBM-based PIM solution that reduces idle cycles of irregular operations through a partially synchronous execution model [14]. SparsePIM is our previously proposed HBM-based PIM architecture for SpMV acceleration [15]. SparsePIM with Opt. combines SparsePIM with the software optimizations described in Section 5.1. SparsePIM+ further integrates the global accumulators into SparsePIM as described in Section 5.3.3, and the label *with Opt.* indicates the software optimizations are applied.

Across the evaluated workloads, the proposed SparsePIM+ consistently improves performance over prior PIM-based designs. In particular, incorporating inter-bank-group accumulation significantly enhances execution efficiency by reducing redundant partial-result handling. When software optimization is enabled, the extension achieves the highest overall performance, demonstrating a geometric mean speedup of 6.16× over the GPU baseline, with a maximum speedup of up to 15.48× across the evaluated workloads. This improvement indicates that hierarchical accumulation combined with optimized data placement effectively mitigates the memory and synchronization overheads inherent in large-scale SpMV execution.

Workloads with highly skewed sparsity patterns exhibit distinct performance trends. For matrices with extremely low non-zero ratios,

such as w13, w14, and w15, the benefits of aggressive in-memory accumulation are limited by sparse column structures. In these cases, a large fraction of columns contains only a small number of non-zero elements, increasing the amount of zero padding required by DRAF. As a result, the relative impact of padded computation and memory accesses becomes more pronounced, reducing the achievable speedup despite improved row-index similarity and balanced clustering. Moreover, under such sparse conditions, the accumulation overhead in GAs increases due to the reduced effectiveness of partial-result consolidation, further limiting performance gains. In particular, for w14, this additional accumulation burden outweighs the benefits of hierarchical accumulation, leading to a slight performance degradation compared to SparsePIM.

Although the performance benefit becomes smaller for extremely sparse workloads, SparsePIM+ still exhibits the performance uplifts over the GPU baseline for these workloads. The benefit is not determined by the ratio of NZEs alone, but also by the absolute number of NZEs, the column-wise distribution of NZEs, and row-index overlap. For example, webbase-1M has an NZE ratio of only 3.106×10^{-6}, but still contains approximately 3.11 million NZEs. However, when many columns contain only one or two NZEs, DRAF rows become underutilized and the relative cost of padded operations increases. Therefore, SparsePIM+ remains beneficial when the reduction in irregular accesses and partial-result movement outweighs DRAF padding overhead, but its advantage becomes limited for matrices with extremely low column occupancy and low row-index overlap.

For workloads with moderate to high overlap in row indices across columns, the proposed extension delivers significant gains. By enabling inter-BG accumulation, partial results are merged in an HBM stack during the execution pipeline. This effect is particularly evident in large matrices with power-law degree distributions, where a significant fraction of partial sums span multiple bank groups. The extended architecture with GAs can perform reduction across bank groups efficiently to curtail both memory traffic and synchronization overhead.

For workloads w11 and w13, enabling software optimization leads to a clear performance improvement over the unoptimized configuration. This gain is primarily attributed to more efficient column placement during the construction of DRAF, which reduces unnecessary memory consumption. Because the degree of zero padding in DRAF is sensitive to how matrix columns are grouped, optimized column assignment lowers the amount of padded data within DRAM rows. Consequently, both the effective memory footprint and the number of DRAM accesses are reduced, resulting in improved execution efficiency.

Overall, the results demonstrate that performance is primarily governed by the interaction between sparse matrix structure and accumulation locality. By combining software-based clustering with hierarchical accumulation across bank groups and the logic die, the proposed extension improves execution efficiency across a wide range of sparsity patterns while preserving compatibility with existing HBM-based PIM execution models.

We also compare SparsePIM+ with the prior PIM-based solutions for SpMV kernels. Our evaluation results exhibit that SparsePIM+ outperforms the prior PIM architectures for the majority of the evaluated workloads. Compared to pSyncPIM, which primarily focuses on mitigating synchronization overhead among banks, SparsePIM+ reduces execution overhead by aggressively eliminating partial results through hierarchical accumulation, yielding clear performance advantages for matrices with significant inter-BG overlap. SpaceA proposes a SpMV acceleration solution based on HMC by exploiting near-memory processing and internal memory-level parallelism. In contrast, the proposed SparsePIM+ consistently achieves higher execution efficiency across diverse workloads by aligning sparse data with DRAM row access granularity and minimizing redundant data movement. As a result, the proposed SparsePIM+ outperforms prior PIM-based designs in most cases, demonstrating that the hierarchical reduction of partial results in SparsePIM+ is a particularly effective approach for accelerating SpMV on HBM-based PIM architectures.

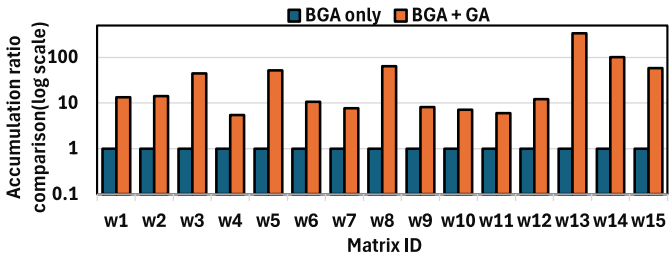


Fig. 13. Accumulation ratio comparison.

These results also indicate that the dominant bottleneck differs between conventional processors and SparsePIM+. On conventional processors, the performance of SpMV operations is mainly limited by irregular memory accesses caused by index-based sparse compression formats. Such accesses reduce cache efficiency and make it difficult to fully utilize the available external memory bandwidth. In contrast, SparsePIM+ alleviates this bottleneck by using the DRAM-friendly compression format to provide DRAM-row-aligned operands that can be directly accessed by the PE datapath. Therefore, matrix values, row indices, and vector elements are accessed from the activated row buffer without additional pointer chasing or scattered memory lookups. Moreover, multiplication and local accumulation are performed inside the HBM stack using internal DRAM bandwidth, while the GA exploits otherwise idle TSV cycles during AB-mode PIM execution for inter-BG reduction. As a result, SparsePIM+ is not primarily limited by external memory bandwidth or host-side irregular access latency. Instead, after irregular accesses are removed, the remaining overhead is mainly determined by PIM-side computation, DRAF-induced padded operations, and BGA/GA accumulation. This effect is more evident in extremely sparse workloads, where underutilized DRAF rows increase the relative cost of padded operations and limit the accumulation opportunities across bank groups.

6.3. Impact of global accumulators

To evaluate the effectiveness of the proposed global accumulators (GAs), we compare accumulation behavior between a baseline configuration that employs only bank-group accumulators (BGA-only) and the proposed configuration that integrates both BGAs and GAs (BGA+GA). We use the *accumulation ratio*, defined as the reduction factor of partial results achieved through in-memory accumulation, normalized to the BGA-only case.

Fig. 13 presents the accumulation ratios across the evaluated workloads on a logarithmic scale. With only BGAs enabled, accumulation is limited to intra-BG partial results. On the other hand, when GA-based inter-BG accumulation is enabled, the accumulation ratio increases significantly, ranging from several times to more than two orders of magnitude, depending on the workload.

To evaluate the effectiveness of GA in reducing host-side accumulation overhead, we analyze the normalized host accumulation burden derived from the accumulation-ratio results. Compared to the BGA-only baseline, the proposed GA reduces the host accumulation burden by 93.13% on average across 15 workloads, leaving only 6.87% of the original host-side reduction work. This reduction ranges from 84.43% to 99.70%, depending on the degree of inter-BG row-index overlap. These results show that GA effectively absorbs inter-BG partial-result reduction within the HBM stack, thereby alleviating host-side data movement and accumulation overhead.

In the BGA-only design, all remaining partial results need to be written back to DRAM banks and retrieved by the host after PIM executions are complete. In contrast, with GA enabled, partial results

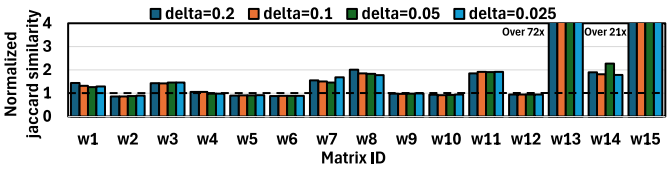


Fig. 14. Jaccard similarity.

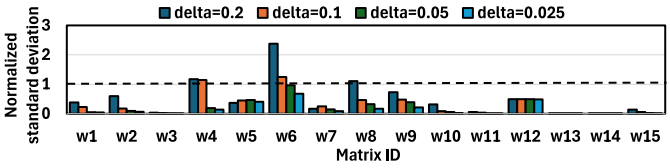


Fig. 15. Standard deviation.

accumulated across bank groups are forwarded to the GA located at the logic die and stored in on-die output buffers. The host can access this buffer while PIM execution is still in progress, allowing host-side accumulation of the remaining partial sums to overlap with in-memory computation.

Furthermore, because the host retrieves results directly from the output buffers associated with each pCh in the GA, the access latency is comparable to a DRAM row-hit–like access rather than a full DRAM read that may incur row-miss or refresh delays. In addition, since many partial sums have already been accumulated within the GA, the host only needs to process a significantly smaller amount of data. These characteristics allow the host to benefit from both lower access latency and reduced partial sum work compared to retrieving all partial results from DRAM banks.

As a result, GA not only reduces the volume of partial results transferred from DRAM banks but also shortens the critical path of SpMV execution by enabling concurrent PIM-side and host-side accumulation. This cooperative accumulation model is particularly effective for workloads with high inter-BG overlap in row indices, as reflected by the large accumulation ratios observed in Fig. 13, which reports accumulation ratios measured with software optimization enabled.

6.4. Evaluation of software optimization

6.4.1. Row index similarity

We evaluate the effectiveness of the software optimization described in Section 5.1 in increasing row-index similarity among clustered matrix columns. As a baseline, we consider a naive clustering strategy that partitions the sparse matrix into K clusters by sequentially grouping columns such that each cluster contains the same number of columns. This approach is similar to the one-dimensional column partitioning scheme used in the prior work [32].

Due to the architectural characteristics of SparsePIM+, higher overlap in row indices among non-zero elements processed within the same bank-group accumulator (BGA) improves accumulation parallelism and overall efficiency. To quantitatively analyze this effect, we measure row-index similarity using the Jaccard similarity metric. Given two columns A and B, the Jaccard similarity is defined as $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$, where A and B denote the sets of row indices corresponding to non-zero elements in each column.

For each cluster, we compute the Jaccard similarity across all possible column pairs and take the average value. The final Jaccard similarity score is obtained by averaging these per-cluster values across all clusters. Fig. 14 reports the Jaccard similarity after software optimization, normalized to the baseline clustering scheme.

Based on preliminary exploration, the hyperparameters *maxIter*, *refineIter*, and *threshold* are set to 30, 5, and 0.2, respectively. We vary the *delta* parameter, which controls the allowable imbalance in non-zero element distribution, and evaluate its impact on row-index similarity.

Across nearly all workloads and *delta* values, the optimized clustering consistently achieves higher row-index similarity than the baseline. In particular, workloads w13 and w5 show up to 72× and 21× increases in Jaccard similarity, respectively. These results indicate that the proposed clustering strategy effectively groups columns with highly overlapping row indices, enabling BGAs to merge partial results more efficiently and improving accumulation throughput.

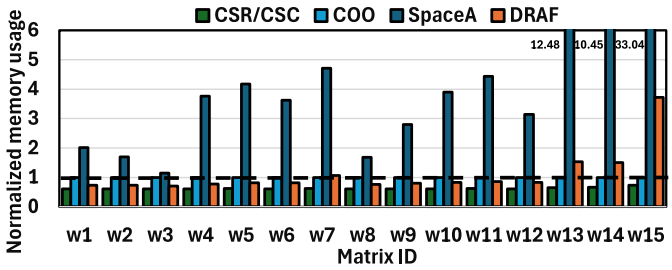


Fig. 16. Memory usage of CSR/CSC, SpaceA's method, and DRAF normalized to the COO format.

6.4.2. Load imbalance

The proposed software optimization also targets the imbalance in the distribution of non-zero elements across clusters. To quantify this effect, we measure the standard deviation of the number of non-zero elements per cluster. A smaller standard deviation indicates a more uniform distribution and, consequently, better load balance among bank groups.

Fig. 15 presents the normalized standard deviation results with the *Refinement* phase enabled. When the hyperparameter *delta* is set below 0.05, the normalized standard deviation remains below 1 for all evaluated workloads, indicating improved balance compared to the baseline. As the optimized clusters are mapped to BGAs, this balanced distribution reduces execution time variation across accumulators, improving overall resource utilization and reducing idle cycles during accumulation.

6.5. Evaluation of DRAF

Fig. 16 compares the memory usage of the proposed DRAM row-aligned format (DRAF) with the conventional COO format and the row-mapping method proposed by SpaceA [47]. The workloads are presented in the same order as Table 3. For DRAF, we account for the memory footprint of both sparse matrices and input vectors, including unused space within DRAM rows, while excluding only the storage reserved for partial-result buffers.

We also measure the memory usage of the conventional CSR/CSC formats. Because these formats store sparse matrices in a compact structure without row-aligned constraints, they achieve the smallest memory footprint among the evaluated formats, requiring approximately 0.60×–0.73× the storage of COO across the evaluated workloads. While this compact representation minimizes storage overhead, it requires indirect indexing and irregular memory accesses during SpMV execution.

Compared to CSR/CSC, DRAF introduces additional storage overhead due to replicated column indices and zero padding required to align sparse data with DRAM row granularity. Across the evaluated workloads, DRAF increases memory usage by approximately 1.55× on average compared to CSR/CSC. This overhead primarily arises from explicitly embedding column indices and corresponding vector elements within each DRAM page, as well as padding when the number of non-zero elements per column does not fully utilize the SIMD width. Despite

this moderate increase in storage, the row-aligned layout of DRAF enables regular memory accesses and eliminates indirect indexing during SpMV execution, making it more suitable for efficient PIM-based computation.

Across most workloads, DRAF reduces memory usage compared to COO, achieving an average reduction of 21.28%. The largest reduction is observed in w3, where memory usage decreases by up to 29.82%. However, in the case of w5, w7, w13, w14, and w15, DRAF incurs additional memory overhead. In particular, w15 exhibits up to a 3.7× increase compared to COO. This behavior arises when many columns contain fewer than 16 non-zero elements or when the number of non-zero elements per column is not a multiple of the SIMD width, resulting in underutilized DRAM row capacity.

Despite this overhead in certain cases, DRAF aligns sparse data with the access granularity required for computation, enabling efficient in-memory processing and tolerating limited storage inefficiency. In contrast, SpaceA's row-mapping method assigns one sparse matrix row to a DRAM row, which can lead to severe internal fragmentation. Under the assumed configuration (1 KB DRAM page, 2 B data elements, and 4 B row and column indices), this approach can waste nearly an entire DRAM row when a matrix row contains only a few non-zero elements.

As a result, SpaceA's method increases memory usage by between 1.2× and 33.0× compared to COO, with workloads such as w5, w13, w14, and w15 exhibiting more than 10× overhead. Overall, DRAF reduces memory overhead by a geometric mean of 4.34× compared to SpaceA, demonstrating that DRAM-aware row-aligned compression is a practical and efficient approach for storing sparse matrices in PIM-based systems.

Beyond memory footprint reduction, DRAF also improves computational efficiency by restructuring sparse data to eliminate indirect accesses inherent in conventional formats such as COO. Although DRAF introduces limited data duplication by explicitly embedding column indices and corresponding vector elements within each DRAM page, and may incur zero padding when the number of non-zero elements per column does not fully utilize the column access granularity, this controlled redundancy enables direct, index-free access during outer-product execution. As a result, the processing units can consume operands directly from the row buffer without additional pointer chasing or scattered memory lookups. Therefore, compared to COO, DRAF not only achieves lower memory usage in most workloads but also transforms irregular, index-driven memory accesses into aligned and computation-friendly accesses, making it a highly efficient format for SpMV execution on HBM-based PIM architectures. Although CSR/CSC provides the most compact representation, its irregular access pattern limits its suitability for in-memory processing. DRAF, on the other hand, reorganizes sparse data to match DRAM row access granularity, enabling efficient PIM execution while maintaining reasonable storage efficiency.

6.6. Hardware overhead

We evaluate the hardware overhead of the additional logic introduced by SparsePIM+ using a synthesis-based methodology consistent with prior HBM-PIM studies. All logic components are synthesized using Synopsys Design Compiler with a SAED 14 nm FinFET standard-cell library. To approximate the DRAM peripheral logic process used in HBM-PIM implementations, we scale the synthesized area results from 14 nm to a 28 nm process using DeepScaleTool [65]. We adopt a larger node than 20 nm because DRAM peripheral circuits are typically fabricated using more relaxed process technologies compared to logic-oriented standard-cell processes. Therefore, scaling to 28 nm provides a more conservative estimate that better reflects the characteristics of DRAM-based PIM implementations.

Table 5 summarizes the measured area of each component. The bank-group accumulator (BGA) and the associated arbitrator together occupy 0.0854 mm^2 per instance. For comparison, the SIMD floating-point unit (FPU) and register files used for PIM computation occupy 0.1886 mm^2 and 0.2882 mm^2, respectively. Note that a single BGA is shared by four banks within a bank group (BG), unlike the SIMD FPU and register files, which are allocated per processing unit. These results indicate that the additional BGA logic introduces only a modest hardware overhead compared to the main compute data-path components of the PIM processing unit.

A global accumulator (GA) together with its on-die data buffer requires 0.845 mm^2 per instance. Since SparsePIM+ deploys one GA per pseudo-channel, a total of sixteen GA instances are integrated on the logic die, resulting in an aggregate GA area of 13.52 mm^2. Based on prior reports on HBM-PIM logic-die area [10–12], the baseline processing unit (PU) occupies approximately 0.94 mm^2. Relative to this baseline, the BGA and arbitrator introduce only a 3.66% additional area overhead. Furthermore, the sixteen GA instances and their data buffers collectively account for 11.35% of the total logic-die area.

To further justify the physical feasibility of integrating the proposed logic-die accumulators, we compare the GA area with a floorplan-level HBM2 logic-die analysis reported in prior work [66]. The prior work implements custom PIM unit clusters and a control unit inside the HBM2 logic die and shows that more than 48.52% of the logic die can be occupied by PIM/control logic while still leaving approximately 46 mm^2 of unoccupied area after considering the TSV-centered floorplan and the physical separation of PIM unit clusters. Our area estimation shown in Table 5 reports that the sixteen GAs and on-die data buffers in SparsePIM+ require 13.52 mm^2 in total. This corresponds to only 29.4% of the reported 46 mm^2 unoccupied-area budget, leaving 32.48 mm^2 of margin for routing resources, power-grid overhead, clock/control wiring, placement spacing, and floorplan inefficiency. The available free-area budget is 3.40× larger than the synthesized GA area. Even if the effective footprint increases by 2× or 3× due to floorplan-aware placement and routing around TSV/PHY-dominated regions, the resulting areas of 27.04 mm^2 and 40.56 mm^2 remain below the 46 mm^2 reference budget.

We also estimate the dynamic power overhead using the same synthesis flow without scale-up. To isolate the incremental cost beyond the baseline HBM-PIM processing unit, we synthesize only the newly introduced BGA components, including the index-queue pair, comparator, register read/write unit, adder controller, and flush controller. The synthesis results indicate that a single BGA consumes 31.85 μW of dynamic power. For comparison, a 16-lane FP16 SIMD multiplier synthesized under the same 14 nm library consumes 34.51 μW, which is 8% higher than the BGA. These results indicate that the additional accumulation logic introduces modest dynamic power overhead relative to existing PIM compute units. Considering the number of BGAs deployed per stack and previously reported HBM-PIM power envelopes, the proposed design operates within the thermal design power (TDP) constraints assumed for HBM-based PIM architectures.

We further evaluate the impact of increasing the index-queue depth from 16 entries to 32 entries. The synthesis results show that a 32-entry configuration increases the dynamic power consumption of a single BGA to 61.58 μW, nearly doubling the power overhead. This level of consumption becomes comparable to, or exceeds, the power of baseline HBM-PIM compute units [10–12], leading to a significantly higher aggregate stack-level power when multiple BGAs are instantiated. Increasing the queue depth reduces the frequency of queue flush events and increases the likelihood of successful intra-BGA accumulation, thereby decreasing the reduction burden on the GA. However, when measured across the evaluated workloads, the execution-time improvement remains marginal and does not provide a meaningful performance benefit relative to the substantial power increase. Considering the TDP constraints of HBM-based PIM systems, such a configuration would impose excessive thermal pressure on the logic die. Therefore, we adopt a 16-entry queue depth as a balanced design point that achieves sufficient accumulation capacity while maintaining power feasibility within the HBM stack.

Table 5
Area overhead.

Component	Area (mm^2)
BGA + Arbitrator (per instance)	0.0854
SIMD FPU	0.1886
Register files	0.2882
GA + Data Buffer (per instance)	0.845

7. Related work

PIM architectures: Processing-in-memory has been widely studied as an architectural approach to mitigate the memory bandwidth and data movement bottlenecks observed in modern computing systems. Both industry and academia have actively explored PIM designs by augmenting existing memory technologies with near-data compute capabilities. Major memory vendors have demonstrated practical PIM solutions integrated into commodity memory devices. For instance, Samsung introduced HBM-based PIM architectures that embed SIMD-style processing units within 3D-stacked memory to exploit high internal bandwidth [10–12].

AiM proposes a PIM architecture implemented on GDDR6 memory, targeting bandwidth-intensive workloads through near-memory computation [67–70]. UPMEM presents a commercially deployed DRAM-based PIM platform that integrates lightweight processing cores directly inside memory chips [71,72]. Beyond commercial systems, a large body of academic work has investigated PIM and near-data processing architectures based on 3D-stacked memory technologies [13,73–87]. More recently, PIM architectures have also been explored as an enabling substrate for large-scale AI and data-intensive workloads [72,88–90].

Recent studies have further demonstrated the feasibility of integrating substantial logic components within the logic die of 3D-stacked HBM memory systems. Prior work has presented PIM accelerators for inline data deduplication, graph processing, and quantized neural networks that incorporate compute logic inside stacked memory architectures, validating the practicality of logic-die integration [91–95]. Moreover, DRAM vendors have discussed the possibility of adopting custom base-die logic in future HBM generations, indicating industrial support for incorporating programmable logic within stacked memory devices [96].

SpMV computations: Sparse matrix–vector multiplication (SpMV) is a fundamental kernel in a wide range of scientific and data analytics applications, yet its performance is often limited by irregular memory accesses and low arithmetic intensity. To alleviate these challenges, prior work has proposed various software-level optimizations to improve data locality, including tiling [30–32,97–99] and reordering techniques [99]. These approaches aim to reduce memory traffic and improve cache utilization but remain constrained by the inherent irregularity of sparse data structures.

In parallel, researchers have developed specialized hardware accelerators designed to better tolerate irregular memory access patterns and unbalanced computations in SpMV workloads [26,28,29,33,59,100–107]. These architectures employ diverse techniques such as streaming dataflows, outer-product formulations, and customized memory hierarchies. However, efficiently managing partial results and memory traffic remains a key challenge, particularly for scalable SpMV execution.

8. Conclusion

In this paper, we propose SparsePIM+, an HBM-based PIM architecture that extends SparsePIM to more efficiently execute large-scale SpMV workloads. While SparsePIM improves SpMV execution by combining software-based column grouping, a DRAM-aware sparse compression format, and bank-group accumulators (BGAs), its accumulation scope is limited to intra-BG partial results. SparsePIM+ addresses this limitation by introducing global accumulators (GAs) and a lightweight arbitration mechanism that enables efficient inter-BG accumulation. By hierarchically reducing partial results—first within bank groups using BGAs and then across bank groups using the GA—SparsePIM+ significantly reduces redundant partial-result read/write operations and improves TSV bandwidth utilization. Furthermore, exporting accumulated results to an output buffer accessible by the host allows in-memory computation and host-side processing to overlap, mitigating the performance overhead caused by inter-BG synchronization. Our experimental results show that these architectural extensions consistently improve SpMV execution efficiency compared to SparsePIM. In particular, SparsePIM+ achieves an average speedup of 1.38× over SparsePIM and 6.16× over an RTX 3080 GPU baseline by reducing inter-BG partial results and increasing effective parallelism during hierarchical accumulation.

CRediT authorship contribution statement

Taewoon Kang: Writing – original draft, Validation, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Namhun Kim:** Validation, Methodology, Investigation, Data curation. **Geonwoo Choi:** Methodology, Investigation. **Taeweon Suh:** Methodology, Investigation, Conceptualization. **Gunjae Koo:** Writing – review & editing, Writing – original draft, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) funded by Korea government (MSIT) (NRF-2021R1C1C1012172), and in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2025-02304483 Chiplet-Based Hub SoC Development Optimized for On-Device Al, RS-2024-00459774 RISC-V Based System Software Development for Open Ecosystem of SDR, IITP-2026-RS-2020-II201819 ICT Creative Consilience Program). The EDA tool was supported by the IC Design Education Center (IDEC), Korea.

Data availability

Data will be made available on request.

References

[1] E. Kaasschieter, Preconditioned conjugate gradients for solving singular systems, J. Comput. Appl. Math. 24 (1) (1988) 265–275, http://dx.doi.org/10.1016/0377-0427(88)90358-5, URL https://www.sciencedirect.com/science/article/pii/0377042788903585.

[2] M.F. Khairoutdinov, D.A. Randall, A cloud resolving model as a cloud parameterization in the NCAR community climate system model: Preliminary results, Geophys. Res. Lett. 28 (18) (2001) 3617–3620, http://dx.doi.org/10.1029/2001GL013552.

[3] B. Liu, M. Wang, H. Foroosh, M. Tappen, M. Pensky, Sparse convolutional neural networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2015.

[4] S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine, Comput. Netw. ISDN Syst. 30 (1) (1998) 107–117, http://dx.doi.org/10.1016/S0169-7552(98)00110-X, URL https://www.sciencedirect.com/science/article/pii/S016975529800110X, Proceedings of the Seventh International World Wide Web Conference.

[5] J.M. Kleinberg, Authoritative sources in a hyperlinked environment, J. ACM 46 (5) (1999) 604–632, http://dx.doi.org/10.1145/324133.324140.

[6] H. Tong, C. Faloutsos, J.-Y. Pan, Random walk with restart: fast solutions and applications, Knowl. Inf. Syst. 14 (2008) 327–346, http://dx.doi.org/10.1007/s10115-007-0094-2.

[7] A. Maringanti, V. Athavale, S.B. Patkar, Acceleration of conjugate gradient method for circuit simulation using CUDA, in: 2009 International Conference on High Performance Computing, HiPC, 2009, pp. 438–444, http://dx.doi.org/10.1109/HIPC.2009.5433184.

[8] S.-H. Weng, Q. Chen, N. Wong, C.-K. Cheng, Circuit simulation via matrix exponential method for stiffness handling and parallel processing, in: Proceedings of the International Conference on Computer-Aided Design, ICCAD '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 407–414, http://dx.doi.org/10.1145/2429384.2429469.

[9] S.-H. Weng, Q. Chen, C.-K. Cheng, Time-domain analysis of large-scale circuits by matrix exponential method with adaptive control, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 31 (8) (2012) 1180–1193, http://dx.doi.org/10.1109/TCAD.2012.2189396.

[10] Y.-C. Kwon, S.H. Lee, J. Lee, S.-H. Kwon, J.M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S.Y. Kim, Y. Cho, J.G. Kim, J. Choi, H.-S. Shin, J. Kim, B. Phuah, H. Kim, M.J. Song, A. Choi, D. Kim, S. Kim, E.-B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. Song, J. Youn, K. Sohn, N.S. Kim, 25.4 a 20nm 6GB function-in-memory DRAM, based on HBM2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications, ISSCC, in: 2021 IEEE International Solid-State Circuits Conference, vol. 64, 2021, pp. 350–352, http://dx.doi.org/10.1109/ISSCC42613.2021.9365862.

[11] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, N.S. Kim, Hardware architecture and software stack for PIM based on commercial DRAM technology : Industrial product, in: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture, ISCA, 2021, pp. 43–56, http://dx.doi.org/10.1109/ISCA52012.2021.00013.

[12] J.H. Kim, S.-h. Kang, S. Lee, H. Kim, W. Song, Y. Ro, S. Lee, D. Wang, H. Shin, B. Phuah, J. Choi, J. So, Y. Cho, J. Song, J. Choi, J. Cho, K. Sohn, Y. Sohn, K. Park, N.S. Kim, Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ml accelerators and beyond, in: 2021 IEEE Hot Chips 33 Symposium, HCS, 2021, pp. 1–26, http://dx.doi.org/10.1109/HCS52781.2021.9567191.

[13] H. Kal, C. Yoo, W.W. Ro, AESPA: Asynchronous execution scheme to exploit bank-level parallelism of processing-in-memory, in: Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 815–827, http://dx.doi.org/10.1145/3613424.3614314.

[14] D. Baek, S. Hwang, J. Huh, pSyncPIM: Partially synchronous execution of sparse matrix operations for all-bank PIM architectures, in: 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture, ISCA, 2024, pp. 354–367, http://dx.doi.org/10.1109/ISCA59077.2024.00034.

[15] T. Kang, G. Choi, T. Suh, G. Koo, SparsePIM: An efficient HBM-based PIM architecture for sparse matrix-vector multiplications, in: Proceedings of the 39th ACM International Conference on Supercomputing, ICS '25, Association for Computing Machinery, New York, NY, USA, 2025, pp. 495–512, http://dx.doi.org/10.1145/3721145.3735111.

[16] S. Li, Z. Yang, D. Reddy, A. Srivastava, B. Jacob, DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator, IEEE Comput. Archit. Lett. 19 (2) (2020) 106–109, http://dx.doi.org/10.1109/LCA.2020.2973991.

[17] Y. Saad, Numerical Methods for Large Eigenvalue Problems: Revised Edition, SIAM, 2011.

[18] K. Kourtis, G. Goumas, N. Koziris, Optimizing sparse matrix-vector multiplication using index and value compression, in: Proceedings of the 5th Conference on Computing Frontiers, CF '08, Association for Computing Machinery, New York, NY, USA, 2008, pp. 87–96, http://dx.doi.org/10.1145/1366230.1366244.

[19] C. Zhang, P. Scheffler, T. Benz, M. Perotti, L. Benini, Near-memory parallel indexing and coalescing: Enabling highly efficient indirect access for SpMV, in: 2024 Design, Automation & Test in Europe Conference & Exhibition, DATE, 2024, pp. 1–6, http://dx.doi.org/10.23919/DATE58400.2024.10546797.

[20] S. Yesil, A. Heidarshenas, A. Morrison, J. Torrellas, Speeding up SpMV for power-law graph analytics by enhancing locality & vectorization, in: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, 2020, pp. 1–15, http://dx.doi.org/10.1109/SC41405.2020.00090.

[21] B. Neelima, P.S. Raghavendra, CSPR: Column only SPARSE matrix representation for performance improvement on GPU architecture, in: D. Nagamalai, E. Renault, M. Dhanuskodi (Eds.), Advances in Parallel Distributed Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 581–595.

[22] W.T. Tang, W.J. Tan, R.S.M. Goh, S.J. Turner, W.-F. Wong, A family of bit-representation-optimized formats for fast sparse matrix-vector multiplication on the GPU, IEEE Trans. Parallel Distrib. Syst. 26 (9) (2015) 2373–2385, http://dx.doi.org/10.1109/TPDS.2014.2357437.

[23] A. Ashari, N. Sedaghati, J. Eisenlohr, P. Sadayappan, A model-driven blocking strategy for load balanced sparse matrix–vector multiplication on GPUs, J. Parallel Distrib. Comput. 76 (2015) 3–15, http://dx.doi.org/10.1016/j.jpdc.2014.11.001, URL https://www.sciencedirect.com/science/article/pii/S0743731514002081, Special Issue on Architecture and Algorithms for Irregular Applications.

[24] M. Maggioni, T. Berger-Wolf, Optimization techniques for sparse matrix–vector multiplication on GPUs, J. Parallel Distrib. Comput. 93–94 (2016) 66–86, http://dx.doi.org/10.1016/j.jpdc.2016.03.011, URL https://www.sciencedirect.com/science/article/pii/S0743731516300028.

[25] J. Gao, W. Ji, J. Liu, S. Shao, Y. Wang, F. Shi, AMF-CSR: Adaptive multi-row folding of CSR for SpMV on GPU, in: 2021 IEEE 27th International Conference on Parallel and Distributed Systems, ICPADS, 2021, pp. 418–425, http://dx.doi.org/10.1109/ICPADS53394.2021.00058.

[26] S. Li, D. Liu, W.L. Liu, Efficient FPGA-based sparse matrix–Vector multiplication with data reuse-aware compression, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 42 (12) (2023) 4606–4617, http://dx.doi.org/10.1109/TCAD.2023.3281715.

[27] S. Malik, P.A. Golnari, Sparse matrix to matrix multiplication: A representation and architecture for acceleration, ASAP, in: 2019 IEEE 30th International Conference on Application-Specific Systems, Architectures and Processors, vol. 2160-052X, 2019, pp. 67–70, http://dx.doi.org/10.1109/ASAP.2019.00-28.

[28] M. Pligouroudis, R.A.G. Nuno, T. Kazmierski, Modified compressed sparse row format for accelerated FPGA-based sparse matrix multiplication, in: 2020 IEEE International Symposium on Circuits and Systems, ISCAS, 2020, pp. 1–5, http://dx.doi.org/10.1109/ISCAS45731.2020.9181266.

[29] U. Mandal, A. Deb, ReMCOO: An efficient representation of sparse matrix-vector multiplication, in: 2023 IEEE Guwahati Subsection Conference, GCON, 2023, pp. 01–06, http://dx.doi.org/10.1109/GCON58516.2023.10183488.

[30] Y. Chen, G. Xiao, F. Wu, Z. Tang, K. Li, tpSpMV: A two-phase large-scale sparse matrix-vector multiplication kernel for manycore architectures, Inform. Sci. 523 (2020) 279–295, http://dx.doi.org/10.1016/j.ins.2020.03.020, URL https://www.sciencedirect.com/science/article/pii/S0020025520302024.

[31] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, G. Tan, TileSpMV: A tiled algorithm for sparse matrix-vector multiplication on GPUs, in: 2021 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2021, pp. 68–78, http://dx.doi.org/10.1109/IPDPS49936.2021.00016.

[32] C. Giannoula, I. Fernandez, J.G. Luna, N. Koziris, G. Goumas, O. Mutlu, SparseP: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures, Proc. ACM Meas. Anal. Comput. Syst. 6 (1) (2022) http://dx.doi.org/10.1145/3508041.

[33] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, R. Dreslinski, OuterSPACE: An outer product based sparse matrix multiplication accelerator, in: 2018 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2018, pp. 724–736, http://dx.doi.org/10.1109/HPCA.2018.00067.

[34] JEDEC Solid State Technology Association, Jesd235a: high bandwidth memory (hbm) dram, JEDEC, 2015, https://www.jedec.org/standards-documents/docs/jesd235a. JEDEC Standard.

[35] JEDEC Solid State Technology Association, Jesd235d: high bandwidth memory (hbm) dram, JEDEC, 2021, https://www.jedec.org/standards-documents/docs/jesd238. JEDEC Standard.

[36] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, K. Kim, HBM (high bandwidth memory) DRAM technology and architecture, in: 2017 IEEE International Memory Workshop, IMW, 2017, pp. 1–4, http://dx.doi.org/10.1109/IMW.2017.7939084.

[37] D.U. Lee, K.W. Kim, K.W. Kim, H. Kim, J.Y. Kim, Y.J. Park, J.H. Kim, D.S. Kim, H.B. Park, J.W. Shin, J.H. Cho, K.H. Kwon, M.J. Kim, J. Lee, K.W. Park, B. Chung, S. Hong, 25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV, in: 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers, ISSCC, 2014, pp. 432–433, http://dx.doi.org/10.1109/ISSCC.2014.6757501.

[38] J.C. Lee, J. Kim, K.W. Kim, Y.J. Ku, D.S. Kim, C. Jeong, T.S. Yun, H. Kim, H.S. Cho, Y.O. Kim, J.H. Kim, J.H. Kim, S. Oh, H.S. Lee, K.H. Kwon, D.B. Lee, Y.J. Choi, J. Lee, H.G. Kim, J.H. Chun, J. Oh, S.H. Lee, 18.3 A 1.2V 64Gb 8-channel 256GB/s HBM DRAM with peripheral-base-die architecture and small-swing technique on heavy load interface, in: 2016 IEEE International Solid-State Circuits Conference, ISSCC, 2016, pp. 318–319, http://dx.doi.org/10.1109/ISSCC.2016.7418035.

[39] J.H. Cho, J. Kim, W.Y. Lee, D.U. Lee, T.K. Kim, H.B. Park, C. Jeong, M.-J. Park, S.G. Baek, S. Choi, B.K. Yoon, Y.J. Choi, K.Y. Lee, D. Shim, J. Oh, J. Kim, S.-H. Lee, A 1.2V 64Gb 341GB/S HBM2 stacked DRAM with spiral point-to-point TSV structure and improved bank group data control, in: 2018 IEEE International Solid-State Circuits Conference, ISSCC, 2018, pp. 208–210, http://dx.doi.org/10.1109/ISSCC.2018.8310257.

[40] D.U. Lee, H.S. Cho, J. Kim, Y.J. Ku, S. Oh, C. Dae Kim, H.W. Kim, W.Y. Lee, T.K. Kim, T.S. Yun, M.J. Kim, S. Lim, S.H. Lee, B.K. Yun, J.I. Moon, J.H. Park, S. Choi, Y.J. Park, C.K. Lee, C. Jeong, J.-S. Lee, S.H. Lee, W.S. We, J.C. Yun, D. Lee, J. Shin, S. Kim, J. Lee, J. Choi, Y. Ju, M.-J. Park, K.S. Lee, Y. Hur, D. Shim, S. Lee, J. Chun, K.-W. Jin, 22.3 A 128Gb 8-High 512GB/s HBM2E DRAM with a pseudo quarter bank structure, power dispersion and an instruction-based at-speed PMBIST, in: 2020 IEEE International Solid-State Circuits Conference, ISSCC, 2020, pp. 334–336, http://dx.doi.org/10.1109/ISSCC19947.2020.9062977.

[41] M.-J. Park, J. Lee, K. Cho, J. Park, J. Moon, S.-H. Lee, T.-K. Kim, S. Oh, S. Choi, Y. Choi, H.S. Cho, T. Yun, Y.J. Koo, J.-S. Lee, B.-K. Yoon, Y.-J. Park, S. Oh, C.K. Lee, S.-H. Lee, H.-W. Kim, Y. Ju, S.-K. Lim, K.Y. Lee, S.-H. Lee, W.S. We, S. Kim, S.M. Yang, K. Lee, I.-K. Kim, Y. Jeon, J.-H. Park, J.C. Yun, S. Kim, D.-Y. Lee, S.-H. Oh, J.-H. Shin, Y. Lee, J. Jang, J. Cho, A 192-Gb 12-High 896-GB/s HBM3 DRAM with a TSV auto-calibration scheme and machine-learning-based layout optimization, IEEE J. Solid-State Circuits 58 (1) (2023) 256–269, http://dx.doi.org/10.1109/JSSC.2022.3193354.

[42] J. Lee, K. Cho, C.K. Lee, Y. Lee, J.-H. Park, S.-H. Oh, Y. Ju, C. Jeong, H.S. Cho, J. Lee, T.-S. Yun, J.H. Cho, S. Oh, J. Moon, Y.-J. Park, H.-S. Choi, I.-K. Kim, S.M. Yang, S.-Y. Kim, J. Jang, J. Kim, S.-H. Lee, Y. Jeon, J. Park, T.-K. Kim, D. Ka, S. Oh, J. Kim, J. Jeon, S. Kim, K.T. Kim, T. Kim, H. Yang, D. Yang, M. Lee, H. Song, D. Jang, J. Shin, H. Kim, C. Baek, H. Jeong, J. Yoon, S.-K. Lim, K.Y. Lee, Y.J. Koo, M.-J. Park, J. Cho, J. Kim, 13.4 A 48GB 16-High 1280GB/s HBM3E DRAM with all-around power TSV and a 6-phase RDQS scheme for TSV area optimization, in: 2024 IEEE International Solid-State Circuits Conference, ISSCC, 67, 2024, pp. 238–240, http://dx.doi.org/10.1109/ISSCC49657.2024.10454440.

[43] K. Sohn, W.-J. Yun, R. Oh, C.-S. Oh, S.-Y. Seo, M.-S. Park, D.-H. Shin, W.-C. Jung, S.-H. Shin, J.-M. Ryu, H.-S. Yu, J.-H. Jung, H. Lee, S.-Y. Kang, Y.-S. Sohn, J.-H. Choi, Y.-C. Bae, S.-J. Jang, G. Jin, A 1.2 V 20 nm 307 GB/s HBM DRAM with at-speed wafer-level IO test scheme and adaptive refresh considering temperature distribution, IEEE J. Solid-State Circuits 52 (1) (2017) 250–260, http://dx.doi.org/10.1109/JSSC.2016.2602221.

[44] C.-S. Oh, K.C. Chun, Y.-Y. Byun, Y.-K. Kim, S.-Y. Kim, Y. Ryu, J. Park, S. Kim, S. Cha, D. Shin, J. Lee, J.-P. Son, B.-K. Ho, S.-J. Cho, B. Kil, S. Ahn, B. Lim, Y. Park, K. Lee, M.-K. Lee, S. Baek, J. Noh, J.-W. Lee, S. Lee, S. Kim, B. Lim, S.-K. Choi, J.-G. Kim, H.-I. Choi, H.-J. Kwon, J.J. Kong, K. Sohn, N.S. Kim, K.-I. Park, J.-B. Lee, 22.1 A 1.1V 16GB 640GB/s HBM2E DRAM with a data-bus window-extension technique and a synergetic on-die ECC scheme, in: 2020 IEEE International Solid-State Circuits Conference, ISSCC, 2020, pp. 330–332, http://dx.doi.org/10.1109/ISSCC19947.2020.9063110.

[45] K. Chae, J. Song, Y. Choi, J. Park, B. Koo, J. Oh, S. Yi, W. Lee, D. Kim, K. Kang, E. Kim, J. Kim, S. Park, S. Park, M. Noh, H.G. Rhew, J. Shin, A 4-nm 1.15 TB/s HBM3 interface with resistor-tuned offset calibration and in situ margin detection, IEEE J. Solid-State Circuits 59 (1) (2024) 231–242, http://dx.doi.org/10.1109/JSSC.2023.3330485.

[46] J.-S. Kim, C.S. Oh, H. Lee, D. Lee, H.-R. Hwang, S. Hwang, B. Na, J. Moon, J.-G. Kim, H. Park, J.-W. Ryu, K. Park, S.-K. Kang, S.-Y. Kim, H. Kim, J.-M. Bang, H. Cho, M. Jang, C. Han, J.-B. Lee, K. Kyung, J.-S. Choi, Y.-H. Jun, A 1.2V 12.8GB/s 2Gb mobile wide-I/O DRAM with 4× 128 I/Os using TSV-based stacking, in: 2011 IEEE International Solid-State Circuits Conference, 2011, pp. 496–498, http://dx.doi.org/10.1109/ISSCC.2011.5746413.

[47] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, 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, http://dx.doi.org/10.1109/HPCA51647.2021.00055.

[48] J. Kang, S. Choi, E. Lee, J. Sim, SpDRAM: Efficient in-DRAM acceleration of sparse matrix-vector multiplication, IEEE Access 12 (2024) 176009–176021, http://dx.doi.org/10.1109/ACCESS.2024.3505622.

[49] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M.A. Kozuch, O. Mutlu, P.B. Gibbons, T.C. Mowry, Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, in: MICRO-50 '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 273–287, http://dx.doi.org/10.1145/3123939.3124544.

[50] G. Koo, H. Jeon, M. Annavaram, Revealing critical loads and hidden data locality in GPGPU applications, in: 2015 IEEE International Symposium on Workload Characterization, 2015, pp. 120–129, http://dx.doi.org/10.1109/IISWC.2015.23.

[51] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N.M. Ghiasi, T. Shahroodi, J.G. Luna, O. Mutlu, SMASH: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '52, Association for Computing Machinery, New York, NY, USA, 2019, pp. 600–614, http://dx.doi.org/10.1145/3352460.3358286.

[52] I. Kim, J. Jeong, Y. Oh, M.K. Yoon, G. Koo, Analyzing GCN aggregation on GPU, IEEE Access 10 (2022) 113046–113060, http://dx.doi.org/10.1109/ACCESS.2022.3217222.

[53] Y.-Y. Jo, M.-H. Jang, S.-W. Kim, S. Park, RealGraph: A graph engine leveraging the power-law distribution of real-world graphs, in: The World Wide Web Conference, WWW '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 807–817, http://dx.doi.org/10.1145/3308558.3313434.

[54] K.P. Sinaga, M.-S. Yang, Unsupervised K-means clustering algorithm, IEEE Access 8 (2020) 80716–80727, http://dx.doi.org/10.1109/ACCESS.2020.2988796.

[55] M.Z. Rodriguez, C.H. Comin, D. Casanova, O.M. Bruno, D.R. Amancio, L.d.F. Costa, F.A. Rodrigues, Clustering algorithms: A comparative approach, PLoS One 14 (1) (2019) 1–34, http://dx.doi.org/10.1371/journal.pone.0210236.

[56] S. Wang, Y. Sun, Z. Bao, On the efficiency of K-means clustering: evaluation, optimization, and algorithm selection, Proc. VLDB Endow. 14 (2) (2020) 163–175, http://dx.doi.org/10.14778/3425879.3425887.

[57] A.K. Jain, Data clustering: 50 years beyond K-means, Pattern Recognit. Lett. 31 (8) (2010) 651–666, http://dx.doi.org/10.1016/j.patrec.2009.09.011, URL https://www.sciencedirect.com/science/article/pii/S0167865509002323, Award winning papers from the 19th International Conference on Pattern Recognition (ICPR).

[58] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, C.W. Fletcher, Extensor: An accelerator for sparse tensor algebra, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '52, Association for Computing Machinery, New York, NY, USA, 2019, pp. 319–333, http://dx.doi.org/10.1145/3352460.3358275.

[59] F. Sadi, J. Sweeney, T.M. Low, J.C. Hoe, L. Pileggi, F. Franchetti, Efficient SpMV operation for large and highly sparse matrices using scalable multi-way merge parallelization, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '52, Association for Computing Machinery, New York, NY, USA, 2019, pp. 347–358, http://dx.doi.org/10.1145/3352460.3358330.

[60] T. Usui, T. Van Chu, K. Kise, A cost-effective and scalable merge sorter tree on FPGAs, in: 2016 Fourth International Symposium on Computing and Networking, CANDAR, 2016, pp. 47–56, http://dx.doi.org/10.1109/CANDAR.2016.0023.

[61] S. Mashimo, T. Van Chu, K. Kise, Cost-effective and high-throughput merge network: Architecture for the fastest FPGA sorting accelerator, SIGARCH Comput. Arch. News 44 (4) (2017) 8–13, http://dx.doi.org/10.1145/3039902.3039905.

[62] C. Nvidia, Cusparse Library, NVIDIA Corporation, Santa Clara, California, 2025.

[63] J. Park, J.H. Lee, S.-K. Park, K.C. Chun, K. Sohn, S. Kang, An in-DRAM BIST for 16 Gb DDR4 DRAM in the 2nd 10-nm-class DRAM process, IEEE Access 9 (2021) 33487–33497, http://dx.doi.org/10.1109/ACCESS.2021.3061349.

[64] Y. Tang, Z. Liu, W. Shang, F. Zhang, B. Wu, Z. Kong, H. Li, H. Ma, K. Cao, Pitch device design in 10nm-class DRAM process through DTCO, in: 2021 IEEE 14th International Conference on ASIC, ASICON, 2021, pp. 1–4, http://dx.doi.org/10.1109/ASICON52560.2021.9620445.

[65] S. Sarangi, B. Baas, DeepScaleTool: A tool for the accurate estimation of technology scaling in the deep-submicron era, in: 2021 IEEE International Symposium on Circuits and Systems, ISCAS, 2021, pp. 1–5, http://dx.doi.org/10.1109/ISCAS51556.2021.9401196.

[66] S. Kim, S. Kim, K. Cho, T. Shin, H. Park, D. Lho, S. Park, K. Son, G. Park, S. Jeong, Y. Kim, J. Kim, Signal integrity and computing performance analysis of a processing-in-memory of high bandwidth memory (PIM-HBM) scheme, IEEE Trans. Compon., Packag. Manuf. Technol. 11 (11) (2021) 1955–1970, http://dx.doi.org/10.1109/TCPMT.2021.3117071.

[67] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, T.N. Vijaykumar, Newton: A DRAM-maker's accelerator-in-memory (AiM) architecture for machine learning, in: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2020, pp. 372–385, http://dx.doi.org/10.1109/MICRO50266.2020.00040.

[68] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, K. Vladimir, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, J. Lee, D. Ko, Y. Jun, K. Cho, I. Kim, C. Song, C. Jeong, D. Kwon, J. Jang, I. Park, J. Chun, J. Cho, A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based accelerator-in-memory supporting 1TFLOPS MAC operation and various activation functions for deep-learning applications, in: 2022 IEEE International Solid-State Circuits Conference, ISSCC, 65, 2022, pp. 1–3, http://dx.doi.org/10.1109/ISSCC42614.2022.9731711.

[69] D. Kwon, S. Lee, K. Kim, S. Oh, J. Park, G.-M. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, V. Kornijcuk, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An, J. Lee, D. Ko, Y. Jun, I. Kim, C. Song, I. Kim, C. Park, S. Kim, C. Jeong, E. Lim, D. Kim, J. Jang, I. Park, J. Chun, J. Cho, A 1ynm 1.25V 8Gb 16Gb/s/Pin GDDR6-based accelerator-in-memory supporting 1TFLOPS MAC operation and various activation functions for deep learning application, IEEE J. Solid-State Circuits 58 (1) (2023) 291–302, http://dx.doi.org/10.1109/JSSC.2022.3200718.

[70] Y. Kwon, K. Vladimir, N. Kim, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An, J. Kim, J. Lee, I. Kim, J. Park, C. Park, Y. Song, B. Yang, H. Lee, S. Kim, D. Kwon, S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, M. Lee, M. Shin, M. Shin, J. Cha, C. Jung, K. Chang, C. Jeong, E. Lim, I. Park, J. Chun, S. Hynix, System architecture and software stack for GDDR6-aim, in: 2022 IEEE Hot Chips 34 Symposium, HCS, 2022, pp. 1–25, http://dx.doi.org/10.1109/HCS55958.2022.9895629.

[71] F. Devaux, The true Processing In Memory accelerator, in: 2019 IEEE Hot Chips 31 Symposium, HCS, IEEE Computer Society, Los Alamitos, CA, USA, 2019, pp. 1–24, http://dx.doi.org/10.1109/HOTCHIPS.2019.8875680, URL https://doi.ieeecomputersociety.org/10.1109/HOTCHIPS.2019.8875680.

[72] C. Ortega, Y. Falevoz, R. Ayrignac, PIM-AI: A novel architecture for high-efficiency LLM inference, 2024, arXiv:2411.17309, URL https://arxiv.org/abs/2411.17309.

[73] Q. Zhu, T. Graf, H.E. Sumbul, L. Pileggi, F. Franchetti, Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware, in: 2013 IEEE High Performance Extreme Computing Conference, HPEC, 2013, pp. 1–6, http://dx.doi.org/10.1109/HPEC.2013.6670336.

[74] Q. Zhu, B. Akin, H.E. Sumbul, F. Sadi, J.C. Hoe, L. Pileggi, F. Franchetti, A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing, in: 2013 IEEE International 3D Systems Integration Conference, 3DIC, 2013, pp. 1–7, http://dx.doi.org/10.1109/3DIC.2013.6702348.

[75] D. Zhang, N. Jayasena, A. Lyashevsky, J.L. Greathouse, L. Xu, M. Ignatowski, TOP-pim: throughput-oriented programmable processing in memory, in: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 85–98, http://dx.doi.org/10.1145/2600212.2600213.

[76] S.H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, F. Li, NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads, in: 2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2014, pp. 190–200, http://dx.doi.org/10.1109/ISPASS.2014.6844483.

[77] J. Ahn, S. Hong, S. Yoo, O. Mutlu, K. Choi, A scalable processing-in-memory accelerator for parallel graph processing, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 105–117, http://dx.doi.org/10.1145/2749469.2750386.

[78] J. Ahn, S. Yoo, O. Mutlu, K. Choi, PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 336–348, http://dx.doi.org/10.1145/2749469.2750385.

[79] A. Farmahini-Farahani, J.H. Ahn, K. Morrow, N.S. Kim, NDA: Near-DRAM acceleration architecture leveraging commodity dram devices and standard memory modules, in: 2015 IEEE 21st International Symposium on High Performance Computer Architecture, HPCA, 2015, pp. 283–295, http://dx.doi.org/10.1109/HPCA.2015.7056040.

[80] B. Akin, F. Franchetti, J.C. Hoe, Data reorganization in memory using 3D-stacked DRAM, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 131–143, http://dx.doi.org/10.1145/2749469.2750397.

[81] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, H. Kim, GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks, in: 2017 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2017, pp. 457–468, http://dx.doi.org/10.1109/HPCA.2017.54.

[82] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, X. Qian, Graphp: Reducing communication for PIM-based graph processing with efficient data partition, in: 2018 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2018, pp. 544–557, http://dx.doi.org/10.1109/HPCA.2018.00053.

[83] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, X. Qian, GraphQ: Scalable PIM-based graph processing, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '52, Association for Computing Machinery, New York, NY, USA, 2019, pp. 712–725, http://dx.doi.org/10.1145/3352460.3358256.

[84] H. Jin, D. Chen, L. Zheng, Y. Huang, P. Yao, J. Zhao, X. Liao, W. Jiang, Accelerating graph convolutional networks through a PIM-accelerated approach, IEEE Trans. Comput. 72 (9) (2023) 2628–2640, http://dx.doi.org/10.1109/TC.2023.3257514.

[85] B. Tian, Q. Chen, M. Gao, ABNDP: Co-optimizing data access and load balance in near-data processing, ASPLOS 2023, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 3, Association for Computing Machinery, New York, NY, USA, 2023, pp. 3–17, http://dx.doi.org/10.1145/3582016.3582026.

[86] W. Yang, Y. Yang, S. Ji, J. Jiang, N. Jing, Q. Wang, Z. Mao, W. Sheng, RecPIM: Efficient in-memory processing for personalized recommendation inference using near-bank architecture, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 43 (10) (2024) 2854–2867, http://dx.doi.org/10.1109/TCAD.2024.3386117.

[87] M. Saed, P.J. Nair, T.M. Aamodt, RayN: Ray tracing acceleration with near-memory computing, in: Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture, MICRO '25, Association for Computing Machinery, New York, NY, USA, 2025, pp. 277–291, http://dx.doi.org/10.1145/3725843.3756067.

[88] G. Heo, S. Lee, J. Cho, H. Choi, S. Lee, H. Ham, G. Kim, D. Mahajan, J. Park, NeuPIMs: NPU-PIM heterogeneous acceleration for batched llm inferencing, ASPLOS '24, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 3, Association for Computing Machinery, New York, NY, USA, 2024, pp. 722–737, http://dx.doi.org/10.1145/3620666.3651380.

[89] M. Seo, X.T. Nguyen, S.J. Hwang, Y. Kwon, G. Kim, C. Park, I. Kim, J. Park, J. Kim, W. Shin, J. Won, H. Choi, K. Kim, D. Kwon, C. Jeong, S. Lee, Y. Choi, W. Byun, S. Baek, H.-J. Lee, J. Kim, IANUS: Integrated accelerator based on NPU-pim unified memory system, ASPLOS '24, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 3, Association for Computing Machinery, New York, NY, USA, 2024, pp. 545–560, http://dx.doi.org/10.1145/3620666.3651324.

[90] W. Kim, Y. Lee, Y. Kim, J. Hwang, S. Oh, J. Jung, A. Huseynov, W.G. Park, C.H. Park, D. Mahajan, J. Park, Pimba: A processing-in-memory acceleration for post-transformer large language model serving, in: Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture, MICRO '25, Association for Computing Machinery, New York, NY, USA, 2025, pp. 292–307, http://dx.doi.org/10.1145/3725843.3756121.

[91] Y.S. Lee, K.M. Kim, J.H. Lee, J.H. Choi, S.W. Chung, A high-performance processing-in-memory accelerator for inline data deduplication, in: 2019 IEEE 37th International Conference on Computer Design, ICCD, 2019, pp. 515–523, http://dx.doi.org/10.1109/ICCD46524.2019.00077.

[92] Y. Huang, L. Zheng, P. Yao, J. Zhao, X. Liao, H. Jin, J. Xue, A heterogeneous PIM hardware-software co-design for energy-efficient graph processing, in: 2020 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2020, pp. 684–695, http://dx.doi.org/10.1109/IPDPS47924.2020.00076.

[93] Y.S. Lee, E.-Y. Chung, Y.-H. Gong, S.W. Chung, Quant-PIM: An energy-efficient processing-in-memory accelerator for layerwise quantized neural networks, IEEE Embed. Syst. Lett. 13 (4) (2021) 162–165, http://dx.doi.org/10.1109/LES.2021.3050253.

[94] J. Park, J. Choi, K. Kyung, M.J. Kim, Y. Kwon, N.S. Kim, J.H. Ahn, AttAcc! unleashing the power of PIM for batched transformer-based generative model inference, ASPLOS '24, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, Association for Computing Machinery, New York, NY, USA, 2024, pp. 103–119, http://dx.doi.org/10.1145/3620665.3640422.

[95] S. Yun, K. Kyung, J. Cho, J. Choi, J. Kim, B. Kim, S. Lee, K. Sohn, J.H. Ahn, Duplex: A device for large language models with mixture of experts, grouped query attention, and continuous batching, in: 2024 57th IEEE/ACM International Symposium on Microarchitecture, MICRO, 2024, pp. 1429–1443, http://dx.doi.org/10.1109/MICRO61859.2024.00105.

[96] J. Kim, Y. Kim, HBM: Memory solution for bandwidth-hungry processors, in: 2014 IEEE Hot Chips 26 Symposium, HCS, 2014, pp. 1–24, http://dx.doi.org/10.1109/HOTCHIPS.2014.7478812.

[97] M. Li, Y. Ao, C. Yang, Adaptive SpMV/SpMSpV on GPUs for input vectors of varied sparsity, IEEE Trans. Parallel Distrib. Syst. 32 (7) (2021) 1842–1853, http://dx.doi.org/10.1109/TPDS.2020.3040150.

[98] N. Namashivavam, S. Mehta, P.-C. Yew, Variable-sized blocks for locality-aware spmv, in: 2021 IEEE/ACM International Symposium on Code Generation and Optimization, CGO, 2021, pp. 211–221, http://dx.doi.org/10.1109/CGO51591.2021.9370327.

[99] X. Fei, Y. Zhang, Regu2D: Accelerating vectorization of SpMV on intel processors through 2D-partitioning and regular arrangement, in: Proceedings of the 50th International Conference on Parallel Processing, ICPP '21, Association for Computing Machinery, New York, NY, USA, 2021, http://dx.doi.org/10.1145/3472456.3472479.

[100] S. Kestur, J.D. Davis, E.S. Chung, Towards a universal FPGA matrix-vector multiplication architecture, in: 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, 2012, pp. 9–16, http://dx.doi.org/10.1109/FCCM.2012.12.

[101] Y. Umuroglu, M. Jahre, An energy efficient column-major backend for FPGA SpMV accelerators, in: 2014 IEEE 32nd International Conference on Computer Design, ICCD, 2014, pp. 432–439, http://dx.doi.org/10.1109/ICCD.2014.6974716.

[102] J. Fowers, K. Ovtcharov, K. Strauss, E.S. Chung, G. Stitt, A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication, in: 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom

Computing Machines, 2014, pp. 36–43, http://dx.doi.org/10.1109/FCCM.2014.23.

[103] M. Hosseinabady, J.L. Nunez-Yanez, A streaming dataflow engine for sparse matrix-vector multiplication using high-level synthesis, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (6) (2020) 1272–1285, http://dx.doi.org/10.1109/TCAD.2019.2912923.

[104] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, Z. Zhang, Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations, in: 2020 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2020, pp. 689–702, http://dx.doi.org/10.1109/HPCA47549.2020.00062.

[105] L. Song, Y. Chi, L. Guo, J. Cong, Serpens: a high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication, in: Proceedings of the 59th ACM/IEEE Design Automation Conference, DAC '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 211–216, http://dx.doi.org/10.1145/3489517.3530420.

[106] W. Kim, H. Kim, J. Lee, H. Kim, J.-H. Kim, Multi-mode SpMV accelerator for transprecision PageRank with real-world graphs, IEEE Access 11 (2023) 6261–6272, http://dx.doi.org/10.1109/ACCESS.2023.3237079.

[107] B. Liu, D. Liu, Towards high-bandwidth-utilization SpMV on FPGAs via partial vector duplication, in: Proceedings of the 28th Asia and South Pacific Design Automation Conference, ASPDAC '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 33–38, http://dx.doi.org/10.1145/3566097.3567839.

Taewoon Kang is a Ph.D. student in the Department of Computer Science and Engineering at Korea University, South Korea, under the supervision of Prof. Gunjae Koo. He received the B.S. degree in System Semiconductor Engineering from Sangmyung University in 2022. His research interests include processing-in-memory (PIM), AI accelerator architectures, and hardware–software co-design for high-performance computing systems. He is currently a visiting scholar in the Ming Hsieh Department of Electrical and Computer Engineering at the University of Southern California, where he conducts collaborative research with Prof. Murali Annavaram. His recent work focuses on HBM-based PIM architectures for accelerating sparse matrix–vector multiplication and other data-intensive workloads.

Namhun Kim received the B.S. degree in System Semiconductor Engineering from Sangmyung University, South Korea. He is currently pursuing the M.S. degree in the Department of Computer Science and Engineering at Korea University, South Korea. His research interests include DRAM reliability, RowHammer mitigation, and processing-in-memory architecture.

Geonwoo Choi received his M.S. degree in Computer Science and Engineering from Korea University, where his research focused on expanded GPU memory systems. He is currently a GPU Cluster Engineer at Moreh, an AI infrastructure company. His research interests include efficient LLM serving on largescale GPU clusters and optimizing KV cache management.

Taeweon Suh is a professor in the Department of Computer Science and Engineering, Korea University. His research interests include AI accelerators, hardware security, RISC-V custom architecture and crypto IP. Prior to joining academia, he was a systems engineer at Intel Corporation in Hillsboro, Oregon, USA. He has a BS in Electrical Engineering from the Korea University, Korea in 1993, and an MS in Electronics Engineering from the Seoul National University, Korea in 1995, and Ph.D. in Computer Engineering from the Georgia Institute of Technology, USA in 2006.

Gunjae Koo received the B.S. and M.S. degrees in Electrical and Computer Engineering from Seoul National University, in 2001 and 2003, respectively, and the Ph.D. degree in Electrical Engineering from the University of Southern California, in 2018. He is currently an associate professor with the Department of Computer Science and Engineering, Korea University. His research interests include computer system architecture and span parallel processor architecture, storage and memory systems, accelerators, and secure processor architecture. Prior to joining Korea University, he was an assistant professor with Hongik University. His industry experience includes the position of a senior research engineer with LG Electronics and also a research intern with Intel.