SparsePIM: An Efficient HBM-Based PIM Architecture for Sparse Matrix-Vector Multiplications

Taewoon Kang
Computer Science and Engineering
Korea University
Seoul, Republic of Korea
taewoon_kang@korea.ac.kr

Geonwoo Choi
Computer Science and Engineering
Korea University
Seoul, Republic of Korea
hnts03@korea.ac.kr

Taeweon Suh
Computer Science and Engineering
Korea University
Seoul, Republic of Korea
suhtw@korea.ac.kr

Gunjae Koo
Computer Science and Engineering
Korea University
Seoul, Republic of Korea
gunjaekoo@korea.ac.kr

Abstract

Sparse matrix-vector multiplication (SpMV) is a fundamental operation across diverse domains, including scientific computing, machine learning, and graph processing. However, its irregular memory access patterns necessitate frequent data retrieval from external memory, leading to significant inefficiencies on conventional processors such as CPUs and GPUs. Processing-in-memory (PIM) presents a promising solution to address these performance bottlenecks observed in memory-intensive workloads. However, existing PIM architectures are primarily optimized for dense matrix operations since conventional memory cell structures struggle with the challenges of indirect indexing and unbalanced data distributions inherent in sparse computations.

In order to address these challenges, we propose SparsePIM, a novel PIM architecture designed to accelerate SpMV computations efficiently. SparsePIM introduces a DRAM row-aligned format (DRAF) to optimize memory access patterns. SparsePIM exploits K-means-based column group partitioning to achieve a balanced load distribution across memory banks. Furthermore, SparsePIM includes bank group (BG) accumulators to mitigate the performance burdens of accumulating partial sums in SpMV operations. By aggregating partial results across multiple banks, SparsePIM can significantly improve the throughput of sparse matrix computations. Leveraging a combination of hardware and software optimizations, SparsePIM can achieve significant performance gains over cuSPARSE-based SpMV kernels on the GPU. Our evaluation demonstrates that SparsePIM achieves up to 5.61× speedup over SpMV on GPUs.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3735111

CCS Concepts

• **Computer systems organization** → *Special purpose systems.*

Keywords

Processing-in-Memory, SpMV, Near-Data Processing

ACM Reference Format:
Taewoon Kang, Geonwoo Choi, Taeweon Suh, and Gunjae Koo. 2025. SparsePIM: An Efficient HBM-Based PIM Architecture for Sparse Matrix-Vector Multiplications. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3721145.3735111

1 Introduction

Sparse matrix-vector multiplication (SpMV) is a fundamental kernel used in a wide range of applications, including scientific computing, machine learning, graph analytics, and circuit simulation [6, 24, 29, 34, 50, 54, 82, 86, 87]. SpMV kernels exhibit extremely low efficiency on general parallel processor architectures and matrix processing engines since those kernels handle large matrices with many zero elements that unnecessarily occupy processing units and storage space. Researchers have proposed specific processor architectures and data compression formats for sparse matrices. However, existing solutions often provoke heavy irregular accesses to the memory hierarchy, thus, the performance of SpMV kernels is significantly restricted by limited memory bandwidth.

Processing-in-memory (PIM) is an emerging approach that can mitigate the performance hurdles caused by constrained memory bandwidth. PIM architectures are implemented by integrating computing units near or within memory cell arrays, thus PIM can exploit high internal bandwidth within a memory package. Moreover, PIM can reduce data movement between processors and off-chip memory modules by offloading computations to memory. Several memory vendors presented PIM solutions, such as HBM-PIM [31, 39, 44]

and GDDR6-AiM [37, 38, 45], demonstrating that PIM solutions are effective for general matrix operations frequently observed in neural network applications. However, the existing PIM solutions cannot handle sparse matrix operations effectively since those solutions rely on regularly structured 2-dimensional memory cell arrays. Note that general representations of sparse matrices include many zero elements, wasting memory space significantly. Furthermore, existing PIM solutions cannot handle compressed sparse matrices efficiently due to the indirect indexing and unbalanced data allocations observed in compressed sparse matrices.

In this paper, we propose SparsePIM, an efficient PIM architecture for accelerating SpMV kernels on 3D-stacked high-bandwidth memory (HBM). SparsePIM features a hardware architecture tailored for SpMV and incorporates software-level optimizations to achieve a significant speedup in SpMV operations on PIM. First, SparsePIM employs a data compression and allocation format aligned with the DRAM row structure to leverage row-buffer locality and reduce the overhead of indirect indexing. SparsePIM also applies K-means-based column grouping to evenly distribute non-zero elements across multiple memory banks and mitigate bank-level load imbalance. Additionally, the software optimization groups matrix columns that include the same row indices, thus SparsePIM can improve computational efficiency by exploiting data parallelism more efficiently. By combining these hardware and software techniques, SparsePIM can significantly improve the performance of SpMV computations on HBM-based PIM.

We evaluate SparsePIM using a modified DRAMSim3 [48] simulator. Our evaluation results exhibit that SparsePIM achieves up to 5.61× 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) 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 SparsePIM is 31.85 μW, which meets the thermal design power (TDP) requirements of the existing HBM-PIM.

The following are the contributions of our work.

- We propose SparsePIM, an efficient software/hardware approach for accelerating SpMV computations on HBM-based PIM architecture. SparsePIM's software optimizations and hardware design are specifically tailored to the hierarchical structure of an HBM stack.
- We present efficient software optimization techniques that ensure load balancing in PIM operations and enable effective accumulation of partial results.
- We propose an effective sparse compression format tailored to the memory cell structures of DRAM.
- We propose an efficient and lightweight hardware architecture and instruction set to support SpMV in HBM-based PIM architecture.
- We implement SparsePIM using a cycle-accurate DRAM simulator to evaluate the effectiveness of the proposed software/hardware co-design approach.

2 Background

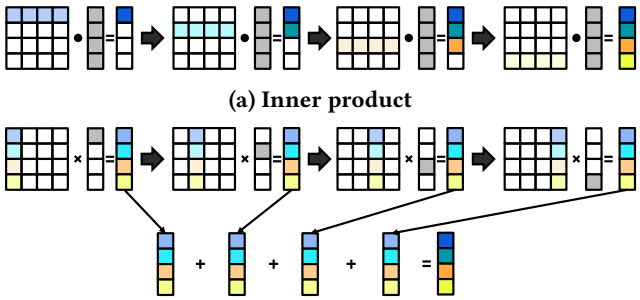
2.1 Sparse compression formats

Although compression formats are widely deployed for reducing the memory footprint of sparse datasets containing a large fraction of zero values, compressed data structures work as critical performance hurdles in parallel architectures. For large-scale applications that rely on SpMV computations, sparse data are represented as large matrices where most elements are zero. Since zero elements waste storage space and computational resources for multiplications, sparse matrices can be efficiently represented by storing only non-zero values and their corresponding indices using compression formats. Modern applications that handle sparse data typically employ compression formats such as coordinate (COO), compressed sparse row (CSR), and compressed sparse column (CSC) formats to represent sparse matrices [71]. Using these compression formats, only non-zero elements are stored along with two-dimensional coordinates in a matrix (i.e., COO) or index pointers to the starting positions of column/row indices (i.e., CSR/CSC formats). Note that processors fetch non-zero elements using the corresponding indices for computing sparse data structures. Hence, computing kernels that handle compressed sparse matrices often create heavy irregular memory accesses to provoke extremely low utilization in processing units and memory systems [36, 90, 91].

In order to handle such performance hurdles by the commonly used sparse data structures, researchers have proposed more efficient compression formats specifically tailored for target processors such as GPUs [4, 14, 51, 60, 79] and domain-specific architectures [47, 52, 53, 68]. Such compression formats can offer more optimized memory access patterns and/or enhanced parallel computations. However, sparse compression formats are not well-explored for PIM architectures that rely on two-dimensional data cell structures.

2.2 SpMV computation methods

SpMV computation can be performed using two different approaches, inner product and outer product, as depicted in Figure 1. The inner product approach performs a conventional dot-product operation between each matrix row and an input vector. By the inner product method, each element



(a) Inner product

(b) Outer product

Figure 1: Comparison of SpMV computation

of a result vector is computed by accumulating partial products computed from elements in a single matrix row and the corresponding elements in an input vector, as shown in Figure 1a. For dense matrix operations, the inner product approach can exploit efficient data-level parallelism in a matrix row and an input vector, and the input vector elements can be reused in local buffers. However, when sparse data structures are involved, the inner product method requires complex index matching with the input vector. Furthermore, since each row of a sparse matrix contains a different number of non-zero elements, the number of multiplications and accumulations varies for each output element. Such irregular processing in SpMV computations can lead to significant performance degradation in regularly organized processing engines [8, 15, 61].

On the other hand, the outer product approach performs a vector-scalar multiplication between a matrix column and a single element of an input vector to compute a partial result vector as depicted in Figure 1b. The partial output vectors computed from the matrix columns are accumulated to generate the final result vector. For SpMV operations, the outer product approach can perform multiplications between non-zero elements in a matrix column and a single element in an input vector to generate partial results associated with the row indices of the corresponding non-zero elements. Then, the partial results associated with the same row indices are accumulated to generate the element in the final result vector. Unlike the inner product approach, which requires index matching between non-zero elements in a row and an input vector, the outer product approach performs simple element-wise multiplications between a non-zero vector and a scalar. The generated partial results associated with row indices can be accumulated immediately once generated. However, the outer product approach requires additional memory space for storing the intermediate partial results. Thus, efficient dataflows and buffer management mechanisms are essential to mitigate the storage overhead of the outer product method [25, 65].

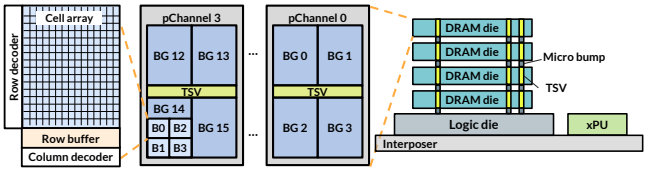


Figure 2: HBM architecture and organization

2.3 High bandwidth memory

High bandwidth memory (HBM) is an advanced memory technology that employs 3D-stacked memory dies mounted on a silicon interposer and vertically connected data channels to provide higher data bandwidth and energy efficiency compared to traditional DRAM [7, 9, 23, 40–43, 56, 57, 63, 67, 76]. As illustrated on the right side of Figure 2, the multiple memory dies in HBM are interconnected using through-silicon vias (TSVs), which allow high-speed data communication between layers [23, 32]. In HBM, multiple memory dies are stacked on a logic die (also called a base die) that includes peripheral circuits such as I/O drivers, data buffers, and a physical interface (PHY). The logic die interfaces with processors (xPU in the figure) through the silicon interposer by managing data transfers from the stacked memory dies. A memory die includes DRAM cells that store data. Each memory die works like a traditional DRAM chip.

The left side of Figure 2 depicts the hierarchical structure of HBM. A single HBM stack includes multiple channels, and each channel supports 128-bit wide data transfers. Each channel is logically divided into two 64-bit pseudo-channels (denoted as pChannel in the figure) to increase parallelism in memory operations. The two pseudo-channels within a physical channel have separate command and address buses but share the same 128-bit data channel. Each pseudo-channel covers several bank groups (BG in the figure), and each bank group includes multiple banks (B in the figure).

A bank is a basic data cell unit that operates independently by memory read and write commands. Each bank includes data cells organized into multiple rows as shown in Figure 2. During a read operation, a specific row is activated using a row address, and then the data in the activated row is loaded into the row buffer (i.e., sense amplifiers) via bit lines. The data in the row buffer is further multiplexed by a column address, then the selected column data is read out. Namely, a row represents the basic access granularity within a bank. In HBM2, the size of a single row is 1 KB [57].

2.4 HBM-based PIM

Processing-in-memory (PIM) is an architectural idea that integrates computation capabilities into memory components to reduce the performance overhead caused by data

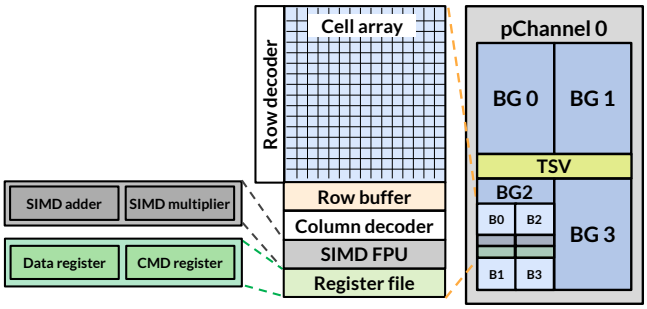


Figure 3: HBM-PIM architecture

transfers from external memory. Recently, a memory vendor demonstrated an HBM-based PIM solution, called HBM-PIM [31, 39, 44]. In order to implement a PIM-enabled die, HBM-PIM integrates SIMD-like processing units and register files into a memory die as shown in Figure 3. Note that these processing elements occupy the die area, thus, half of the rows in a DRAM bank are replaced with the logic elements on a PIM die.

HBM-PIM supports two distinct operation modes, a single-bank (SB) mode and an all-bank (AB) mode. In SB mode, HBM-PIM behaves like conventional memory devices, namely, memory commands are issued to only a single target bank specified by a bank address. In contrast, AB mode is designed to facilitate PIM operations in HBM-PIM to exploit parallel operations by activating multiple banks. In AB mode, a bank address is ignored, thus, memory commands are broadcast to all banks simultaneously. To initiate PIM operations of HBM-PIM, a host processor transfers a special command sequence to change the operation mode of HBM-PIM to AB mode. Then, the PIM dies in HBM-PIM can concurrently execute PIM instructions triggered by column commands.

HBM-PIM supports several RISC-type instructions to perform PIM operations. The instruction set of HBM-PIM includes control, arithmetic, and data movement instructions [44]. Since HBM-PIM is primarily designed to support general matrix-vector operations, HBM-PIM does not include instructions that can support indirect indexing, which is observed frequently in sparse data processing.

3 Related work

SpMV on PIM: SpMV operations are memory-intensive kernels since SpMV can create many irregular memory transactions from its indirect indexing mechanisms. Several researchers have explored PIM-based solutions that can accelerate SpMV computations by leveraging high internal bandwidth and parallel computation capabilities of PIM architectures.

SpaceA is a PIM solution for accelerating SpMV operations on hybrid memory cube (HMC) [88]. SpaceA employs two-level content-addressable memories (CAMs) to perform cache tag matching more efficiently in processing engines. SpaceA also utilizes software optimizations to distribute non-zero elements evenly across multiple memory banks. SpaceA assigned the row data of sparse matrices to rows in a bank to optimize row data accesses. SpaceA exploits a unique feature of HMC that allows internal data movement across memory dies. However, currently, HMC is not a standardized memory device, thus, the impacts of SpaceA may be limited.

pSyncPIM is an HBM-based approach that can accelerate SpMV and sparse triangular solve (SpTRSV) operations [5]. pSyncPIM proposes a partial synchronous execution mode that can accommodate irregular accesses and computations more efficiently. pSyncPIM employs semi-independent bank operations to minimize idle times of PIM operations. pSyncPIM tackles the synchronous executions in HBM-PIM, however, pSyncPIM does not address the inefficient indexing caused by conventional compression formats.

SpDRAM [27] is a DRAM-based SpMV acceleration approach that exploits bit-serial operations [74]. SpDRAM leverages in-DRAM bitwise computations to perform arithmetic operations. SpDRAM presents a data allocation method that can support bit-serial operations efficiently. However, SpDRAM requires complex bit-level control mechanisms to fully exploit the bit-serial operations.

PIM architectures: PIM is an emerging research area actively explored by both industry and academia. Major memory vendors have demonstrated their PIM solutions based on existing memory technology. Samsung presented HBM-based PIM solutions [31, 39, 44]. AiM is a PIM architecture implemented on GDDR6 [16, 37, 38, 45]. UPMEM is a commercialized PIM solution based on DRAM [10, 64]. Academic researchers have also presented PIM architectures based on 3D-stacked memory technology [1–3, 11, 21, 25, 58, 69, 81, 89, 92–96]. PIM solutions have been further investigated for modern large-scale AI systems [18, 64, 73].

SpMV computations: As SpMV is a core compute kernel for a wide range of applications, researchers have presented efficient hardware/software solutions for SpMV. In order to exploit data locality in SpMV kernels, researchers have proposed software optimizations such as tiling [8, 12, 15, 46, 59, 61] and reordering [12]. Several researchers have worked on hardware accelerator architectures that can handle irregular computations and memory accesses in SpMV kernels more efficiently [13, 19, 19, 28, 33, 47, 49, 53, 65, 68, 72, 77, 78, 83].

4 Motivation

The performance overhead of SpMV kernels becomes more critical as modern applications employ large-scale sparse data structures. In order to handle the storage overhead of numerous zero elements, SpMV kernels typically employ

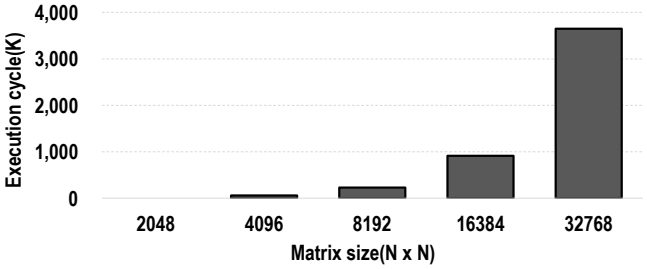


Figure 4: Execution cycle of SpMV on HBM-PIM by the size of a sparse matrix

sparse compression formats. However, as described in Section 2.1, compressed sparse data structures include index information along with non-zero elements (NZEs), thus, the indirect indexing in compressed sparse matrices causes significant inefficiency in both processing units and memory systems [26, 35]. Furthermore, non-zero elements are not uniformly distributed across columns/rows in a sparse matrix, leading to significant load imbalance in computations in columns/rows. For example, graph data structures used in graph analytics and neural network applications often follow power-law distributions in the number of non-zero elements per row or column. Note that the overall performance of SpMV kernels is usually determined by the row/column that includes a large number of non-zero elements. As a result, such applications experience serious performance degradation when processing rows or columns in parallel [22, 30]

In particular, the performance issues caused by load imbalance in SpMV should be addressed seriously when SpMV kernels are executed on PIM-based architectures since the processing units in PIM architectures are tightly coupled with banks or bank groups in a memory package. Note that the row/column data in a large sparse matrix are allocated across multiple banks (or bank groups) in a memory package and data communications across banks (or bank groups) are typically restricted. Hence, if non-zero elements are non-uniformly allocated in multiple banks, it causes significant performance drops due to imbalanced computations among bank-level processing engines. In order to mitigate such issues, software-based optimizations are required to balance the distribution of non-zero elements across banks (or bank groups) associated with the processing units.

To accommodate SpMV computations efficiently on PIM platforms, the hardware architectures in PIM or logic dies are specifically designed to support sparse data structures. However, commodity PIM solutions such as HBM-PIM and GDDR6-AiM are primarily tailored for general matrix-vector operations. In order to investigate the performance burdens by large-scale sparse matrix operations, as shown in Figure 4 we measure the performance of SpMV formatted in a general matrix structure using a DRAMsim3-based HBM-PIM simulator [48]. Our experiment results exhibit that the execution cycles of the SpMV kernel increase exponentially as the matrix size grows. Our analysis reveals that the current HBM-PIM architecture is significantly inefficient for computing large-scale sparse matrices.

5 SparsePIM

In this paper, we propose *SparsePIM*, an efficient PIM architecture optimized for SpMV computations. As discussed in the previous section, PIM solutions require software-based optimizations and specific hardware architectures to tackle the performance overhead of large-scale SpMV kernels. SparsePIM employs software approaches that can distribute non-zero elements evenly across multiple banks and bank groups. For this purpose, SparsePIM utilizes a K-means-based simple partitioning algorithm. We also propose a new sparse data compression format, called a DRAM row-aligned format (DRAF), tailored to the data cell structures of DRAM. Our proposed compression format minimizes data movement between DRAM data cells and processing engines to improve processing efficiency for generating partial results of outer product computation for SpMV. Moreover, SparsePIM incorporates partial result accumulators associated with bank groups to reduce the number of partial products quickly. By combining both software and hardware approaches, SparsePIM can perform large-scale SpMV computations efficiently within an HBM stack.

SparsePIM performs SpMV operations using an outer product approach, as described in Section 2.2. The outer product-based SpMV computations by SparsePIM are executed hierarchically based on the hierarchical structures in 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 non-zero elements in a matrix column and a single element from an input vector. The corresponding bank group includes a bank group accumulator (BGA) that accumulates partial results generated from the banks within the bank group.

5.1 Software optimizations

SparsePIM performs vector-scalar multiplications for the outer product operation of SpMV using processing engines in multiple banks. Hence, as described in Section 4, non-zero elements in a sparse matrix should be loaded evenly across banks. Furthermore, SparsePIM's efficiency can be improved if non-zero elements associated with the same row indices are grouped within a single bank group. Note that SparsePIM performs accumulations of partial results using bank group

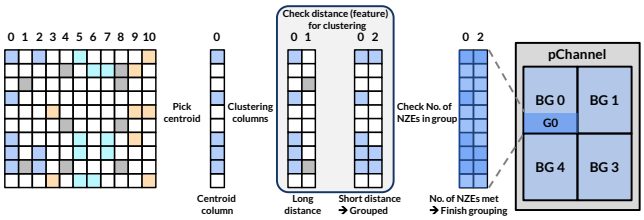


Figure 5: Simplified execution flow of grouping columns based on row index similarity.

accumulators. Consequently, SparsePIM employs software-based optimizations to achieve higher computational efficiency.

Figure 5 depicts how SparsePIM's software-based optimization clusters the columns of a sparse matrix. In this example, we assume the size of the sparse matrix is 10×10 and the colored blocks represent non-zero elements (NZEs) in the matrix. First, SparsePIM selects a reference column randomly as a centroid column (*column 0* in the figure). Then, SparsePIM computes the *row-index similarity* (i.e., the number of NZEs that share the same row indices with NZEs in the reference column) of a different column based on the centroid column. In this example, *column 1* and *column 2* have row-index similarities of 1 and 3, respectively, when compared to *column 0*. As a result, *column 2* is grouped with the centroid column (i.e. *column 0*) and assigned to the same bank group. Note that the grouped columns include more NZEs that share the same row indices, thus, more partial results generated by the outer product operation can be efficiently accumulated by a bank group accumulator (BGA).

We choose K-means as the base clustering algorithm for sparse matrices [75]. K-means clustering exhibits low computation cost compared to other clustering algorithms [20, 70, 85]. Since SparsePIM incorporates a fixed number of bank groups associated with processing units, we set the number of clusters equivalent to the number of PIM bank groups when K-means clustering is applied. However, simple K-means clustering cannot guarantee an even distribution of non-zero elements across bank groups. In order to address this issue, SparsePIM employs a modified clustering algorithm, referred to as *bounded cap K-means*, and an additional *refinement* step for balancing the distribution of non-zero elements further.

SparsePIM's software optimization algorithm is described in Algorithm 1. The algorithm receives the data structures, *nNZE(c)* and *fmap(c)*, of each column in the sparse matrix. *nNZE(c)* denotes the number of non-zero elements (NZEs) in the column *c*, and *fmap(c)* is a feature map that represents the mapping information of non-zero elements. The behavior of the algorithm is controlled using the hyperparameters.

Algorithm 1 Software optimizations based on K-means

Input: col.nz[] (each c has nNZE(c), fmap(c)), k, maxIter, refineIter, delta, th
Output: col.bg[] (where col.bg[c] = assigned bg index (0...k-1))

$totalN = \sum_c nNZE(c)$
$minCap = (totalN/k) \cdot (1 - delta)$
$maxCap = (totalN/k) \cdot (1 + delta)$

Bounded cap K-means (iteration no. = maxIter):
Initialize centroids[] ← random columns
for each column **c** in col.nz[] **do**
  for each **bgIdx** = $0 \ldots k-1$ **do**
    if $nNZE(col.bg[\textbf{bgIdx}]) + nNZE(c) \leq maxCap$ **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))

Refinement (iteration no. = refineIter):
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

return col.bg[]

k specifies the number of partitions, corresponding to the total number of bank groups for PIM operations in the HBM stack. *delta* adjusts the min/max boundary condition (i.e., *minCap* and *maxCap*) of the number of NZEs in each cluster. Namely, *delta* adjusts the deviations from the average number of NZEs in each cluster. Note that all clusters should contain the same number of NZEs to ensure balanced PIM operations across clusters (i.e., bank groups). However, enforcing this constraint may reduce the row-index similarity

within a cluster. SparsePIM utilizes *delta* to trade off the load balancing of PIM operations and row-index similarity. *th* sets the distance threshold for balancing the distribution of NZEs across clusters. Finally, *maxIter* and *refineIter* define the number of iterations for the *bounded cap K-means* and *refinement* steps, respectively.

Bounded cap K-means: SparsePIM exploits a K-means clustering algorithm to group sparse matrix columns that share the same row index values. Note that a traditional K-means algorithm clusters elements based on Euclidean distance between elements. SparsePIM's *bounded cap K-means* algorithm computes the Euclidean distance between columns using the feature maps (i.e., *fmap*), which represent the row indices of NZEs in the column. The clustering process begins by randomly selecting a centroid column for each cluster. Similar to traditional K-means, SparsePIM iteratively updates the centroid of each cluster based on the computed Euclidean distances from the centroid. SparsePIM uses the row-index similarity calculated using the feature maps of the sparse matrix columns as *distances*. However, traditional K-means cannot guarantee the even distribution of non-zero elements across clusters since each column includes a divergent number of non-zero elements, and the algorithm only considers the distances between feature maps. Hence, SparsePIM sets the constraints (i.e., *minCap* and *maxCap*) that define the allowable range of NZEs per cluster. The bounded-cap K-means process terminates either when the maximum number of iterations is reached or when the centroids no longer change.

Refinement: SparsePIM further leverages the refinement process to guarantee load balancing in PIM operations per bank group. Since the K-means-based clustering algorithm primarily considers the row-index similarity from the centroids, the clusters may exhibit significant variation in the number of NZEs. SparsePIM applies the refinement step to reduce imbalances in NZE distribution among bank groups. In each iteration, SparsePIM picks a *big cluster* (i.e., the cluster containing NZEs more than the average) and a *small cluster* (i.e., the cluster containing fewer NZEs). SparsePIM tries to move an arbitrary column from the *big cluster* to the *small cluster* and evaluate the distance change. If the distance change is less than the threshold (*th*), SparsePIM allows the move of the selected column. The refinement process continues until the maximum number of iterations (*refineIter*) is reached, or no further improvements can be made.

One of the main reasons SparsePIM employs a K-means-based clustering approach is its relatively fast execution time compared to other methods. In order to evaluate this advantage, we compare our proposed software optimization methodology with the preprocessing technique presented in SpaceA [88]. Whereas SparsePIM performs the column-based grouping, SpaceA clusters rows of a sparse matrix.

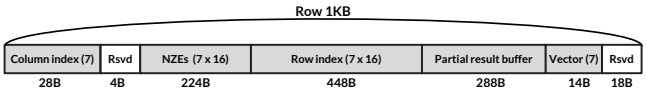


Figure 6: DRAM row-aligned format for a DRAM row

SpaceA proposes a preprocessing strategy that improves computational efficiency through data reuse by organizing partitioned rows such that their internal non-zero elements share similar column indices. This preprocessing consists of two stages: *assignment of rows to logical PEs* and *assignment of logical PEs to physical PEs*. Among these, the first stage dominates the overall execution time, resulting in a time complexity of $O(N_{PE} \times N_{NZE} \times \log N_{NZE})$.

In contrast, SparsePIM's optimization technique assigns columns to clusters during the *bounded cap K-means* process by comparing all data points (n) based on the total average number of non-zero elements (same as N_{NZE}), (d) based on the average number of non-zero elements per column, and iterating over the k clusters. This results in a time complexity of $O(nkd)$. Since the clustering process is repeated for t iterations, the overall complexity becomes $O(nkdt)$. For the *refinement* process, all clusters are evaluated to identify a *big cluster* and a *small cluster*, and the columns within the selected *big cluster* are traversed to determine whether they should be reassigned. Repeating this process for *refineIter* iterations (r) yields a complexity of $O\left(k \times \frac{n}{k} \times r\right) = O(nr)$. Since both the K-means iterations (t) and the refinement iterations (r) are much smaller than n (i.e., $t, r \ll n$), the final time complexity is dominated by $O(nkd)$. In conclusion, the time complexities of the preprocessing strategies in SpaceA and SparsePIM depend on the relative magnitudes of d and $\log(N_{NZE})$, and may vary depending on the distribution of non-zero elements in the matrix. As a result, the two approaches exhibit different performance characteristics across varying sparsity patterns.

5.2 DRAM row-aligned format

SparsePIM employs a column-oriented sparse compression format designed to align with DRAM's row structure. Note that the existing sparse compression formats include index information that represents the positions of non-zero elements in a sparse matrix. However, such indexing mechanisms require indirect accesses, which are significantly inefficient for DRAM's two-dimensional data cell structures. In order to address such issues, SparsePIM adopts the outer product approach for SpMV and an efficient compression format aligned with DRAM rows. We call this sparse compression format DRAF (DRAM row-aligned format).

Figure 6 exhibits the data organization formatted by DRAF. In each data field, the number in brackets indicates the number of elements. We assume that the size of a single element in matrices and vectors is 2 bytes (i.e., FP16 precision) and a single column/row index occupies 4 bytes. Since SparsePIM implements its PIM architecture on HBM2, we assume that DRAF is applied based on 1 KB of DRAM row size [57]. Based on DRAF, SparsePIM assigns necessary operands and indices within the data field of a single DRAM row as shown in the figure. Note that SparsePIM performs an outer product operation between non-zero elements (NZEs) in a matrix column and a single element in a vector (see Section 2.2). In order to support the outer product operations using the data in a row buffer, SparsePIM encapsulates a column index, NZEs, and the corresponding row indices from a matrix column. Under the HBM2 configuration, SparsePIM can store up to seven matrix columns within a single DRAM row, and each column can contain a maximum of 16 NZEs. If a single matrix column includes more than 16 NZEs, the remaining NZEs are assigned to a new column data group that retains the same column index. Following the DRAF structure, SparsePIM also allocates a data field for storing the computed partial results (i.e., partial result buffer in Figure 6). Unlike the prior outer product-based SpMV accelerators, SparsePIM utilizes the DRAM row buffer to store the partial results, thus, SparsePIM does not require additional storage space. Note that DRAF already includes the row indices field, thus, SparsePIM does not store row indices associated with the partial results. The partial result field can also hold up to 112 partial results. The scalar elements from the input vector are stored in the vector field. Since a single DRAM row includes seven columns in the matrix, SparsePIM also stores the corresponding seven vector elements with DRAF. DRAF includes reserved fields (marked as Rsvd) to align with the 32-byte DRAM column access granularity.

5.3 Hardware architecture

In order to support the vector-scalar multiplications and the accumulations of partial results, SparsePIM incorporates processing units such as SIMD FPU and bank group accumulators (BGAs). The left side of Figure 7 describes the hardware components added to a bank group in an HBM stack. As explained in Section 2.3, a single bank group of HBM2 consists of four banks. Similar to prior HBM-PIM, SparsePIM relies on a register file and a 16-lane SIMD FPU that supports FP16 multiplications and additions associated with a bank [31, 39, 44]. Note that the number of lanes in the SIMD FPU is designed to align with the 32-byte DRAM column access granularity. As shown in the figure, a single SIMD FPU is shared by two banks (i.e., even and odd banks). SparsePIM stores the PIM instructions generated from a microkernel in

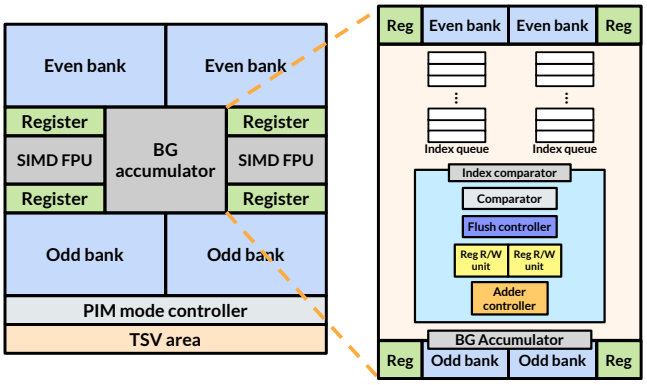


Figure 7: Hardware architecture of processing units in a bank group

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																

command registers, and these instructions are triggered by memory commands like HBM-PIM. As a result, SparsePIM can operate using conventional DRAM command sequences.

5.3.1 SparsePIM instructions.

SparsePIM adds two PIM instructions, *BMOV* and *BACC*, to the base instruction set of HBM-PIM [44]. Table 1 presents the instruction formats and target components of the newly added instructions. *BMOV* moves the selected element from the *vector* field of the row buffer data (see Figure 6) into the scalar register. Note that SparsePIM performs vector-scalar multiplications for outer product operations using a column of a matrix and an input vector element. *BMOV* reads the target vector element selected by *SRC0Idx* from the row buffer and stores the read element in the scalar register. *BACC* is used for accumulating partial results with the bank group accumulator (BGA). When *BACC* is executed, SparsePIM reads eight index values from the *row indices* field in the row buffer, then transfers the index values to the row index queue in BGA. *BACC* selects the target index data using *SRC0Idx*. The corresponding partial results in the data register are also transferred to the index queue simultaneously. Once the index queue is filled, BGA can initiate the accumulation process.

5.3.2 Bank group accumulator (BGA).

The right side of Figure 7 depicts the architecture of a BGA. SparsePIM exploits BGAs to accumulate the partial results associated with the same row index, thus, SparsePIM can reduce the number of partial results effectively. Since all banks within the same bank group share a single BGA, the partial results computed by these banks are aggregated in the BGA. Note that SparsePIM relies on software optimizations to allocate columns with high row-index similarity to the same group, as described in Section 5.1. Hence, the BGA within a bank group can find the partial products associated with the same row index with a higher probability.

A BGA incorporates two index queues where each entry contains a partial result along with the corresponding row index. When SparsePIM executes the *BACC* instruction, the row index values in the *row indices* field in a row buffer and the partial results in the data registers are transferred to the entries in the index queue. Note that the size of a single DRAM column is 32 bytes, thus, SparsePIM fills eight entries per *BACC* instruction. In this paper, we set the depth of each index queue to 16, considering the power constraint of an HBM stack.

The BGA performs the accumulation process using an index comparator unit composed of a comparator, a flush controller, register read/write units, and an adder controller. In order to accumulate partial results within a bank group, the BGA matches row index values one by one from the index queues and executes the corresponding accumulation operations [17, 55, 72, 84]. The index comparator inspects the head entries of the index queues to compare the row index values. If the index values match, BGA accumulates the partial results associated with the row index. Then, the accumulated result is assigned to one of the register read/write units, and zero is allocated to another register read/write unit. The register read/write units write the accumulated result and zero back to the target data registers, thus one of the data registers can contain the merged partial result. If the index values in the head entries are not equivalent, BGA dequeues the head entry that has a smaller index value. Note that row indices are pre-sorted by SparsePIM's software optimization while the elements of a sparse matrix are compressed and encapsulated using the DRAM row-aligned format (DRAF). This accumulation process is repeated until the index queues are empty.

To prevent overflows of the index queues, the BGA includes a flush controller that can clear the index queues. Since the accumulation process of BGA performs the index matching one-by-one, consecutive *BACC* instructions may cause the index queues to overflow. To address this, the BGA flushes the index queues if one of the index queues includes more than nine valid entries and a *BACC* instruction is executed. This flushing mechanism is required to maintain the sorted order of row indices within the index queues. The adder controller in the BGA is designed to utilize the SIMD adder in HBM-PIM more efficiently. The adder controller transfers operands from the register read/write units to the FP adders. Note that the SpMV microkernel of SparsePIM uses only the SIMD multiplier for vector-scalar multiplications of the outer product approach, thus, the SIMD adder remains idle. The adder controller utilizes this idle resource by allocating partial results to the SIMD adder when the *BACC* instruction is executed.

5.4 Execution flow

We now describe the overall execution flow of SparsePIM. Initially, HBM is set to single-bank (SB) mode to operate the HBM stack as a standard memory device. SparsePIM performs the software optimizations to partition the columns of a sparse matrix and balance the number of non-zero elements across the bank groups in the target HBM stack. SparsePIM compresses the sparse matrix and the input vector using the DRAM row-aligned format (DRAF) as described in Section 6.5. Then, SparsePIM stores the formatted data to the target bank groups on PIM dies of the HBM stack.

In order to enable the PIM operations of HBM-PIM, SparsePIM switches the operation mode to all-bank (AB) mode. Then, SparsePIM programs the compiled PIM instructions for SpMV operations to the command registers in the bank groups. SparsePIM initiates the execution of the PIM instructions by issuing DRAM commands for PIM operations. Since the HBM stack is set to AB mode, all PIM operations of the bank groups on PIM dies are synchronized.

SparsePIM performs the vector-scalar multiplications of the outer product SpMV computation as follows. Using the *BMOV* instructions, SparsePIM reads one of the vector elements in the *vector* field in the row buffer of the bank and stores it in the scalar register. Then, SparsePIM performs vector-scalar multiplications with the SIMD multipliers using the 16 non-zero elements from the *NZEs* field in the row buffer and the vector element in the scalar register. The 16 partial results are stored in the data register.

Next, SparsePIM performs the partial result accumulations using the bank group accumulator (BGA). SparsePIM initiates the accumulation process using the *BACC* instruction. Then, the row index values in the *row index* field in a row buffer are transferred to the index queues in the BGA. Since the size of a single index value is 4 bytes, SparsePIM issues two DRAM column commands to load 64 bytes of the row index data. The BGA operates in a pipelining manner as explained in the previous section. The accumulated partial results are then written back to the data register.

SparsePIM repeats the vector-scalar multiplication and accumulation process seven times since a DRAF encapsulates seven columns of the sparse matrix. Finally, the accumulated

Table 2: Configurations of SparsePIM and baseline

Component	Configuration
SparsePIM	
Memory type	HBM2
No. of pChannels	16
No. of BG / pChannel	4
No. of banks / BG	4
No. of DRAM rows	16,384
No. of DRAM columns	64
No. of SIMD FPU / BG	2
No. of BGA / BG	1
Clock frequency	1 GHz
Timing parameters	$t_{CCDS} = 1$, $t_{CCDL} = 2$, $t_{RAS} = 34$, $t_{RP} = 14$, $t_{RCDRD/WR} = 14$, $t_{RRDS} = 4$, $t_{RRDL} = 6$
Baseline (NVIDIA RTX3090)	
No. of CUDA cores	10,496
Clock frequency	1,395 MHz / 1,695 MHz
Memory bandwidth	935.8 GB/s
Device memory	24 GB GDDR6X

Table 3: Workload

Workload	Domain	Size	Ratio of NZEs
(w1) cant	SE	62,451	5.218E-04
(w2) crankseg_2	SE	63,838	1.744E-03
(w3) lhr71	Chem	70,304	3.092E-04
(w4) pdb1HYS	CB	36,417	1.652E-03
(w5) rma10	CFD	46,835	1.082E-03
(w6) soc-sign-epinions	SNA	131,828	4.841E-05
(w7) Stanford	WGA	281,903	2.910E-05
(w8) bcsstk32	SE	44,609	5.174E-04
(w9) consph	SE	83,334	4.387E-04
(w10) ct20stif	SE	52,329	5.023E-04
(w11) ohne2	SDS	181,343	3.364E-04
(w12) pwtk	SE	217,918	1.248E-04
(w13) shipsec1	SE	140,874	2.004E-04
(w14) ASIC_100k	CS	99,340	9.669E-05
(w15) xenon2	MS	157,464	1.559E-04
(w16) webbase-1M	WGA	1,000,005	3.106E-06

partial results are stored in the *partial result buffer* field in the row buffer using the *MOV* instruction.

6 Evaluation

6.1 Methodology

For performance evaluation, we employ a cycle-accurate simulator modified from DRAMSim3 [48]. Detailed system parameters are summarized in Table 2. In order to align with the thermal design power (TDP) and logic area assumptions of prior work [31, 39, 44], processing logic is placed on only four DRAM dies, while the remaining four dies contain no logic circuits. The simulator is configured to accurately measure the number of execution cycles consumed by both memory transactions and computation.

To evaluate SpMV performance, we implement a micro-kernel in assembly based on the custom instruction set described in Section 5.3.1. The kernel is loaded through memory write commands into the command register, enabling the simulator to model realistic instruction execution. Since each vector-scalar multiplication produces 16 partial results per one *MUL* command, the kernel is written to issue two consecutive *BACC* instructions, each processing eight row indices from the DRAM row buffer.

For comparison, we also evaluate SpMV execution on an NVIDIA RTX3090 GPU using cuSPARSE [62] as the baseline. The RTX3090, based on the Ampere architecture, is used to ensure architectural consistency with the RTX3080 evaluated in pSyncPIM [5]. RTX3090's specifications are also included in Table 2. GPU performance is measured using *cudaEvent*, recording the elapsed time from host-to-device data transfer to host-side retrieval of computation results. The reported execution time is the average of five independent runs conducted on a real system to account for execution time variability inherent in real-world environments. For SparsePIM, we measure the time from kernel programming into the command register and initialization to computation completion and result retrieval.

For power and area analysis, we synthesize the SystemVerilog code using Synopsys Design Compiler. Given that DRAM manufacturing nodes such as 1ynm, 1znm, and 1anm are fabricated using sub-14nm technology [66, 80], we adopt the SAED 14nm FinFET process to evaluate the hardware characteristics of our design.

6.2 Workload

The sparse matrix suite used for evaluation is summarized in Table 3. Each matrix is categorized by its application domain, number of dimensions, and the Ratio of non-zero elements (Ratio of NZEs), which is computed as the ratio of the number of non-zero elements to the total number of matrix elements. For compact representation, domain names are abbreviated as follows: SE denotes structural engineering, Chem refers to chemical process simulation, WGA corresponds to web graph analysis, CB to computational biology, SNA to social network analysis, CFD to computational fluid dynamics, SDS to semiconductor device simulation, CS to circuit simulation, and MS to material science. Using this benchmark suite, we evaluate the performance of the proposed architecture, highlighting the benefits of applying software optimization and employing the DRAF compression format.

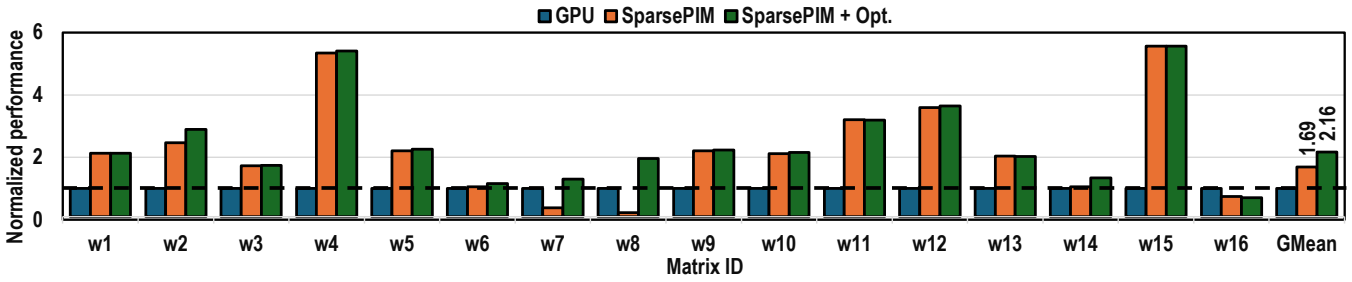


Figure 8: Performance

6.3 Performance

Figure 8 presents the performance evaluation results in terms of execution time, normalized to the baseline performance of the GPU using cuSPARSE. The results indicate that standard deviation, Jaccard similarity score, and DRAF-related overhead jointly influence the observed performance. SparsePIM without software optimization (SparsePIM + Opt.) achieves a geometric mean speedup of 1.69× over the GPU baseline, and this increases to 2.16× when software optimization is applied. The maximum speedup observed is 5.61× when both SparsePIM and software optimization are used. However, performance degrades slightly in the case of workload w16.

For workloads such as w6, w7, w14, and w16, which exhibit a fraction of non-zero elements smaller than 10^{-5}, the performance gain compared to the GPU is relatively limited. This is because the computational cost associated with zero-padding in DRAF, illustrated in Figure 11, becomes less dominant as the number of non-zero elements increases. In particular, workload w16 shows improved standard deviation, as seen in Figure 10, and improved Jaccard similarity score, shown in Figure 9. However, performance degradation is observed regardless of software optimization. This is attributed to the relatively high overhead of converting the compressed format to DRAF, which increases both memory usage and the number of operations due to the characteristics of SparsePIM. Since SparsePIM performs synchronous column access across all banks, an increase in padded zero elements directly translates to increased computational load and memory latency.

An analysis of the sparse matrix structure shows that, except for workloads w1, w6, w7, w9, w14, and w16, all other workloads contain more than one non-zero element per column. Among these, w6 has 30.05% of its columns with only one non-zero element, while in w1, w7, w9, and w14, approximately 3.75% of columns contain a single non-zero element. In contrast, w16 exhibits an extreme case in which 91.94% of the columns contain only one non-zero element. This structure significantly increases the number of padded zeros required to align the number of values per group to 16 in DRAF, thereby causing substantial overhead in both memory and computation.

In the case of w7 and w8, the application of software optimization results in a notable performance improvement compared to their unoptimized performance. This improvement arises because software optimization significantly reduces memory usage when converting the matrix to DRAF, in contrast to the conventional tiling-based approach. Since the amount of zero padding in DRAF depends on the placement of matrix columns, the memory footprint and access time are reduced when column placement is optimized.

For other workloads, SparsePIM outperforms the GPU baseline, but the additional performance gains from software optimization are negligible or even negative. This is mainly due to increased latency when the number of rows in a cluster is not divisible by four, requiring all bank groups to wait until the remaining rows are processed. In such cases, the benefits of optimization diminish. Moreover, for workloads where DRAF memory usage is already low or lower than that of COO, the added benefit of software optimization is minimal. Ultimately, SparsePIM performance is highly sensitive to the number of non-zero elements and the extent of zero padding, as these factors significantly affect both the number of memory accesses and the time required to retrieve partial results.

6.4 Evaluation of software optimization

6.4.1 *Row Index Similarity.*

In this section, we describe the effects of applying the software optimization methodology introduced in Section 5.1 to input matrix clustering. To evaluate the performance improvement, we use as our baseline a naive clustering method that partitions the original sparse matrix into K clusters by dividing the column vectors sequentially such that each cluster contains an equal number of columns. This baseline approach resembles the 1D-partitioning scheme proposed in [15].

Due to the architectural characteristics of SparsePIM, higher row index similarity among the non-zero elements processed within each BG accumulator leads to improved parallelism and, consequently, enhanced computational efficiency. To verify this effect, we evaluated the row index similarity of columns within each cluster using a score based

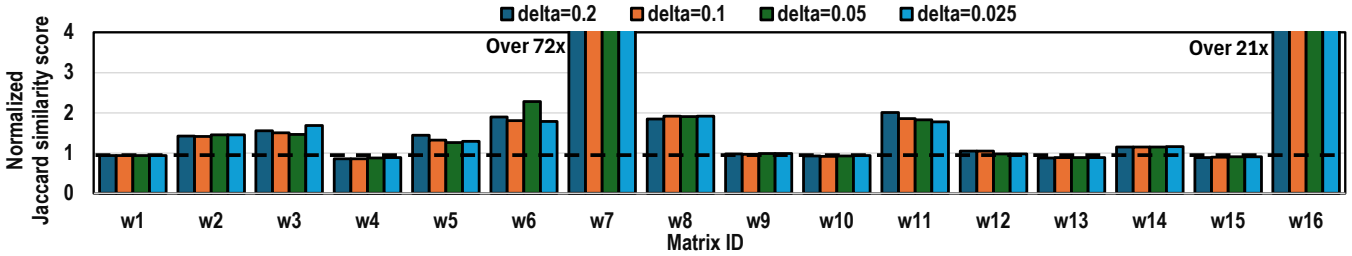


Figure 9: Jaccard similarity

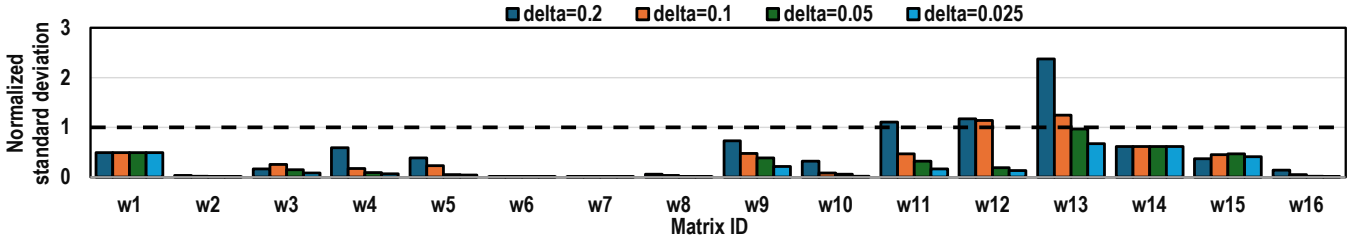


Figure 10: Standard deviation

on Jaccard similarity. The Jaccard similarity between two columns A and B is defined as $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$, where A and B represent the sets of row indices containing non-zero elements in each column within the same cluster. To assess overall similarity within a cluster, the Jaccard similarity is computed for all possible pairs of columns, and the average of these values is used. This per-cluster average is then further averaged across all clusters to obtain a final Jaccard similarity score.

Figure 9 shows the Jaccard similarity score after applying software optimization, normalized to the unoptimized baseline. Based on preliminary experiments, the hyperparameters *maxIter*, *refineIter*, and *threshold* were heuristically set to 30, 5, and 0.2, respectively. We varied the *delta* parameter, which is an essential hyperparameter controlling load imbalance, and visualized its impact on the Jaccard similarity score.

Experimental results show that, across nearly all workloads and *delta* values, row index similarity significantly improves compared to the baseline. In particular, workloads w7 and w16 exhibit up to 72× and 21× increases, respectively, in Jaccard similarity score across all *delta* values. These results indicate that the clustering methodology effectively groups columns with highly overlapping row indices. Consequently, when each cluster is mapped to a BG and accumulation is performed using the BGA, the throughput of accumulation operations can be substantially improved.

6.4.2 Load Imbalance.

Our software optimization methodology reduces the imbalance in the number of non-zero elements across clusters by first applying *bounded cap K-means* and then refining the distribution through an additional *Refinement* phase. We

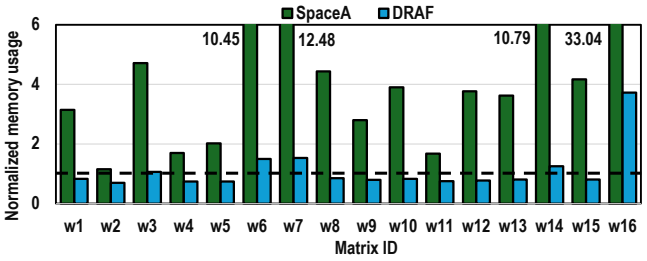


Figure 11: Memory usage of COO, SpaceA's method, and DRAF

use the standard deviation of the number of non-zero elements across clusters as the evaluation metric to quantify the effectiveness of this approach. A lower standard deviation indicates that the clusters' number of non-zero elements is closer to the mean, signifying a more uniform data distribution. Figure 10 illustrates the results normalized against the baseline for the case where *Refinement* is applied. Across all of the sparse suite workload inputs, the software optimization shows a normalized standard deviation of less than 1 when hyperparameter *delta* is smaller than 0.05, indicating that each cluster's number of non-zero elements is more evenly distributed compared to the baseline. As the number of non-zero element balanced clusters is fed into the BGAs for accumulation, the runtime discrepancy among accumulators diminishes, enhancing resource utilization and reducing idle time overall.

6.5 Evaluation of DRAF

Figure 11 compares the memory usage of the proposed DRAF format with the conventional COO format and the mapping

method previously proposed by SpaceA [88]. The sparse suites are arranged in the same order as Table 3. For DRAF, we include the memory space used for storing vectors and sparse matrices, including the empty area, while excluding only the space reserved for partial result storage. In most cases, except for w3, w6, w7, w14, and w16, DRAF reduces memory usage by an average of 21.28% compared to COO, with the most significant reduction observed in w2, where memory usage decreases by up to 29.82%. However, in the case of w16, DRAF increases memory usage by up to 3.7× compared to COO. This overhead occurs when many columns contain fewer than 16 non-zero values or when the number of values per column is not divisible by 16. Nonetheless, since DRAF aligns data according to the access granularity required for computation, it can tolerate such overhead in a limited number of cases.

In contrast, the mapping method proposed by SpaceA stores matrix values by mapping each DRAM row to one matrix row, and under the configuration assumed in this work (1KB row buffer, 2B data, and 4B for each row and column index), this approach can waste up to 1,014 bytes of space per row. In the worst case, only a single data element can reside in a DRAM row, resulting in severe inefficiency. As a result, SpaceA's mapping method increases memory usage by between 1.2× and 33.0× compared to COO. In particular, w6, w7, w14, and w16 exhibit more than 10× overhead, which correlates with their high fraction of non-zero elements. In contrast, DRAF achieves significantly better efficiency and reduces overhead by a geometric mean of 4.34× compared to SpaceA, demonstrating its practicality and suitability for DRAM-based sparse matrix storage.

6.6 Power and area

We measured the power consumption and area of the BGA using Synopsys Design Compiler. Specifically, we evaluated the area of the newly added components, such as the index queue pair, comparator, register read/write unit, adder controller, and flush controller, while excluding the SIMD FPU from the measurement. The result shows that the BGA consumes 31.85 μW of dynamic power. For comparison, we designed a 16-lane SIMD multiplier identical to the one embedded in HBM-PIM, using the same 14nm process. The measurement result shows that the SIMD multiplier consumes 34.51 μW, which is 8% higher than the BGA. Based on these results and existing HBM-PIM power data, we conclude that SparsePIM can operate without exceeding the thermal design power (TDP) limit of HBM2.

For the area comparison, we used a 14nm process for the BGA, in contrast to the different technology node used in HBM-PIM. Therefore, instead of comparing physical cell area, we compared gate counts with the HBM-PIM design. In SparsePIM, a bank group consisting of four banks shares one processing unit, whereas HBM-PIM assigns one processing unit to every two banks. As a result, the BGA increases the gate count by only 2.2% compared to the original HBM-PIM processing unit. In contrast, pSyncPIM [5] increases the area by 35.8% over HBM-PIM. SparsePIM achieves SpMV acceleration while occupying less area and staying within thermal constraints, demonstrating both area and power efficiency.

7 Conclusion

In this paper, we propose SparsePIM, an HBM-based PIM architecture for accelerating SpMV kernels. SparsePIM includes a software optimization approach, a new sparse matrix compression format, and an efficient processing in-memory architecture to perform SpMV operations. By applying the proposed software optimization, SparsePIM improves the parallel computations by increasing the row index similarity within a column group to solve the load imbalance issues. In addition, we propose a new compressed format, called DRAF, to facilitate operations according to data access granularity. In the case of the proposed SparsePIM, we enable SpMV, a sparse operation, in existing PIMs that only support dense BLAS operations, such as existing matrix-vector or vector-scalar multiplication, through the BGA. The standard deviation results show that the load imbalance problem is improved through SW optimization, and the Jaccard similarity scores show that the row index similarity is increased for most workloads, which increases parallelism when accumulating partial results. According to the experimental results, the performance was improved by an average of 2.16× and up to 5.61× speedup compared to the cuSPARSE-based GPU. In addition, the TDP and logic area met the requirements of HBM2 despite efficiently processing SpMV operations. Our approach lays a foundation for future PIM architectures capable of efficiently handling irregular data patterns.

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) (IITP-2025-RS-2020-II201819, ICT Creative Consilience Program / No. RS-2024-00459774, RISC-V based system software development for open ecosystem of SDR). The EDA tool was supported by the IC Design Education Center (IDEC), Korea. Gunjae Koo is the corresponding author.

References

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 105–117. https://doi.org/10.1145/2749469.2750386

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 336–348. https://doi.org/10.1145/2749469.2750385

[3] Berkin Akin, Franz Franchetti, and James C. Hoe. 2015. Data reorganization in memory using 3D-stacked DRAM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 131–143. https://doi.org/10.1145/2749469.2750397

[4] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P. Sadayappan. 2015. A model-driven blocking strategy for load balanced sparse matrix–vector multiplication on GPUs. *J. Parallel and Distrib. Comput.* 76 (2015), 3–15. https://doi.org/10.1016/j.jpdc.2014.11.001 Special Issue on Architecture and Algorithms for Irregular Applications.

[5] Daehyeon Baek, Soojin Hwang, and Jaehyuk Huh. 2024. 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)*. 354–367. https://doi.org/10.1109/ISCA59077.2024.00034

[6] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems* 30, 1 (1998), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X Proceedings of the Seventh International World Wide Web Conference.

[7] Kwanyeob Chae, Jaegeun Song, Yoonjae Choi, Jiyeon Park, Billy Koo, Jihun Oh, Shinyoung Yi, Won Lee, Dongha Kim, Kyeongkeun Kang, Eunsu Kim, Juyoung Kim, Sanghune Park, Sungcheol Park, Mijung Noh, Hyo Gyuem Rhew, and Jongshin Shin. 2024. A 4-nm 1.15 TB/s HBM3 Interface With Resistor-Tuned Offset Calibration and In Situ Margin Detection. *IEEE Journal of Solid-State Circuits* 59, 1 (2024), 231–242. https://doi.org/10.1109/JSSC.2023.3330485

[8] Yuedan Chen, Guoqing Xiao, Fan Wu, Zhuo Tang, and Keqin Li. 2020. tpSpMV: A two-phase large-scale sparse matrix-vector multiplication kernel for manycore architectures. *Information Sciences* 523 (2020), 279–295. https://doi.org/10.1016/j.ins.2020.03.020

[9] Jin Hee Cho, Jihwan Kim, Woo Young Lee, Dong Uk Lee, Tae Kyun Kim, Heat Bit Park, Chunseok Jeong, Myeong-Jae Park, Seung Geun Baek, Seokwoo Choi, Byung Kuk Yoon, Young Jae Choi, Kyo Yun Lee, Daeyong Shim, Jonghoon Oh, Jinkook Kim, and Seok-Hee Lee. 2018. 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)*. 208–210. https://doi.org/10.1109/ISSCC.2018.8310257

[10] Fabrice Devaux. 2019. The true Processing In Memory accelerator . In *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–24. https://doi.org/10.1109/HOTCHIPS.2019.8875680

[11] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. 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)*. 283–295. https://doi.org/10.1109/HPCA.2015.7056040

[12] Xiang Fei and Youhui Zhang. 2021. Regu2D: Accelerating Vectorization of SpMV on Intel Processors through 2D-partitioning and Regular Arrangement. In *Proceedings of the 50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP '21)*. Association for Computing Machinery, New York, NY, USA, Article 77, 11 pages. https://doi.org/10.1145/3472456.3472479

[13] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S. Chung, and Greg Stitt. 2014. A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 36–43. https://doi.org/10.1109/FCCM.2014.23

[14] Jianhua Gao, Weixing Ji, Jie Liu, Senhao Shao, Yizhuo Wang, and Feng Shi. 2021. AMF-CSR: Adaptive Multi-Row Folding of CSR for SpMV on GPU. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*. 418–425. https://doi.org/10.1109/ICPADS53394.2021.00058

[15] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 21 (Feb. 2022), 49 pages. https://doi.org/10.1145/3508041

[16] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and T. N. Vijaykumar. 2020. Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 372–385. https://doi.org/10.1109/MICRO50266.2020.00040

[17] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275

[18] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. 2024. NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 722–737. https://doi.org/10.1145/3620666.3651380

[19] Mohammad Hosseinabady and Jose Luis Nunez-Yanez. 2020. A Streaming Dataflow Engine for Sparse Matrix-Vector Multiplication Using High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 6 (2020), 1272–1285. https://doi.org/10.1109/TCAD.2019.2912923

[20] Anil K. Jain. 2010. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters* 31, 8 (2010), 651–666. https://doi.org/10.1016/j.patrec.2009.09.011 Award winning papers from the 19th International Conference on Pattern Recognition (ICPR).

[21] Hai Jin, Dan Chen, Long Zheng, Yu Huang, Pengcheng Yao, Jin Zhao, Xiaofei Liao, and Wenbin Jiang. 2023. Accelerating Graph Convolutional Networks Through a PIM-Accelerated Approach. *IEEE Trans. Comput.* 72, 9 (2023), 2628–2640. https://doi.org/10.1109/TC.2023.3257514

[22] Yong-Yeon Jo, Myung-Hwan Jang, Sang-Wook Kim, and Sunju Park. 2019. RealGraph: A Graph Engine Leveraging the Power-Law Distribution of Real-World Graphs. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 807–817. https://doi.org/10.1145/3308558.3313434

[23] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. HBM (High Bandwidth Memory) DRAM Technology and Architecture. In *2017 IEEE International Memory Workshop (IMW)*. 1–4. https://doi.org/10.1109/IMW.2017.7939084

[24] E.F. Kaasschieter. 1988. Preconditioned conjugate gradients for solving singular systems. *J. Comput. Appl. Math.* 24, 1 (1988), 265–275. https://doi.org/10.1016/0377-0427(88)90358-5

[25] Hongju Kal, Chanyoung Yoo, and Won Woo Ro. 2023. 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* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 815–827. https://doi.org/10.1145/3613424.3614314

[26] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. 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* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 600–614. https://doi.org/10.1145/3352460.3358286

[27] Jieui Kang, Soeun Choi, Eunjin Lee, and Jaehyeong Sim. 2024. SpDRAM: Efficient In-DRAM Acceleration of Sparse Matrix-Vector Multiplication. *IEEE Access* 12 (2024), 176009–176021. https://doi.org/10.1109/ACCESS.2024.3505622

[28] Srinidhi Kestur, John D. Davis, and Eric S. Chung. 2012. Towards a Universal FPGA Matrix-Vector Multiplication Architecture. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 9–16. https://doi.org/10.1109/FCCM.2012.12

[29] Marat F. Khairoutdinov and David A. Randall. 2001. A cloud resolving model as a cloud parameterization in the NCAR Community Climate System Model: Preliminary results. *Geophysical Research Letters* 28, 18 (2001), 3617–3620. https://doi.org/10.1029/2001GL013552

[30] Inje Kim, Jonghyun Jeong, Yunho Oh, Myung Kuk Yoon, and Gunjae Koo. 2022. Analyzing GCN Aggregation on GPU. *IEEE Access* 10 (2022), 113046–113060. https://doi.org/10.1109/ACCESS.2022.3217222

[31] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, Jihyun Choi, Jinin So, YeonGon Cho, JoonHo Song, Jangseok Choi, Jeonghyeon Cho, Kyomin Sohn, Youngsoo Sohn, Kwangil Park, and Nam Sung Kim. 2021. Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–26. https://doi.org/10.1109/HCS52781.2021.9567191

[32] Jung-Sik Kim, Chi Sung Oh, Hocheol Lee, Donghyuk Lee, Hyong-Ryol Hwang, Sooman Hwang, Byongwook Na, Joungwook Moon, Jin-Guk Kim, Hanna Park, Jang-Woo Ryu, Kiwon Park, Sang-Kyu Kang, So-Young Kim, Hoyoung Kim, Jong-Min Bang, Hyunyoon Cho, Minsoo Jang, Cheolmin Han, Jung-Bae Lee, Kyehyun Kyung, Joo-Sun Choi, and Young-Hyun Jun. 2011. 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*. 496–498. https://doi.org/10.1109/ISSCC.2011.5746413

[33] Whijin Kim, Hana Kim, Jihye Lee, Hyunji Kim, and Ji-Hoon Kim. 2023. Multi-Mode SpMV Accelerator for Transprecision PageRank With Real-World Graphs. *IEEE Access* 11 (2023), 6261–6272. https://doi.org/10.1109/ACCESS.2023.3237079

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

[35] Gunjae Koo, Hyeran Jeon, and Murali Annavaram. 2015. Revealing Critical Loads and Hidden Data Locality in GPGPU Applications. In *2015 IEEE International Symposium on Workload Characterization*. 120–129. https://doi.org/10.1109/IISWC.2015.23

[36] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. 2008. Optimizing sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 5th Conference on Computing Frontiers* (Ischia, Italy) *(CF '08)*. Association for Computing Machinery, New York, NY, USA, 87–96. https://doi.org/10.1145/1366230.1366244

[37] Daehan Kwon, Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gi-Moon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Vladimir Kornijcuk, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Guhyun Kim, Byeongju An, Jaewook Lee, Donguc Ko, Younggun Jun, Ilwoong Kim, Choungki Song, Ilkon Kim, Chanwook Park, Seho Kim, Chunseok Jeong, Euicheol Lim, Dongkyun Kim, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. 2023. 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 Journal of Solid-State Circuits* 58, 1 (2023), 291–302. https://doi.org/10.1109/JSSC.2022.3200718

[38] Yongkee Kwon, Kornijcuk Vladimir, Nahsung Kim, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Guhyun Kim, Byeongju An, Jeongbin Kim, Jaewook Lee, Ilkon Kim, Jaehan Park, Chanwook Park, Yosub Song, Byeongsu Yang, Hyungdeok Lee, Seho Kim, Daehan Kwon, Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Myeongjun Lee, Minyoung Shin, Minhwan Shin, Jaekyung Cha, Changson Jung, Kijoon Chang, Chunseok Jeong, Euicheol Lim, Il Park, Junhyun Chun, and Sk Hynix. 2022. System Architecture and Software Stack for GDDR6-AiM. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. 1–25. https://doi.org/10.1109/HCS55958.2022.9895629

[39] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phuah, HyoungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 2021. 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. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 64. 350–352. https://doi.org/10.1109/ISSCC42613.2021.9365862

[40] Dong Uk Lee, Ho Sung Cho, Jihwan Kim, Young Jun Ku, Sangmuk Oh, Chul Dae Kim, Hyun Woo Kim, Woo Young Lee, Tae Kyun Kim, Tae Sik Yun, Min Jeong Kim, SeungGyeon Lim, Seong Hee Lee, Byung Kuk Yun, Jun Il Moon, Ji Hwan Park, Seokwoo Choi, Young Jun Park, Chang Kwon Lee, Chunseok Jeong, Jae-Seung Lee, Sang Hun Lee, Woo Sung We, Jong Chan Yun, Doobock Lee, Junghyun Shin, Seungchan Kim, Junghwan Lee, Jiho Choi, Yucheon Ju, Myeong-Jae Park, Kang Seol Lee, Youngdo Hur, Daeyong Shim, Sangkwon Lee, Junhyun Chun, and Kyo-Won Jin. 2020. 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)*. 334–336. https://doi.org/10.1109/ISSCC19947.2020.9062977

[41] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Hongjung Kim, Ju Young Kim, Young Jun Park, Jae Hwan Kim, Dae Suk Kim, Heat Bit Park, Jin Wook Shin, Jang Hwan Cho, Ki Hun Kwon, Min Jeong Kim, Jaejin Lee, Kun Woo Park, Byongtae Chung, and Sungjoo Hong.

2014. 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)*. 432–433. https://doi.org/10.1109/ISSCC.2014.6757501

[42] Jinhyung Lee, Kyungjun Cho, Chang Kwon Lee, Yeonho Lee, Jae-Hyung Park, Su-Hyun Oh, Yucheon Ju, Chunseok Jeong, Ho Sung Cho, Jaeseung Lee, Tae-Sik Yun, Jin Hee Cho, Sangmuk Oh, Junil Moon, Young-Jun Park, Hong-Seok Choi, In-Keun Kim, Seung Min Yang, Sun-Yeol Kim, Jaemin Jang, Jinwook Kim, Seong-Hee Lee, Younghyun Jeon, Juhyung Park, Tae-Kyun Kim, Dongyoon Ka, Sanghoon Oh, Jinse Kim, Junyeol Jeon, Seonhong Kim, Kyeong Tae Kim, Taeho Kim, Hyeonjin Yang, Dongju Yang, Minseop Lee, Heewoong Song, Dongwook Jang, Junghyun Shin, Hyunsik Kim, Changki Baek, Hajun Jeong, Jongchan Yoon, Seung-Kyun Lim, Kyo Yun Lee, Young Jun Koo, Myeong-Jae Park, Joohwan Cho, and Jonghwan Kim. 2024. 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)*, Vol. 67. 238–240. https://doi.org/10.1109/ISSCC49657.2024.10454440

[43] Jong Chern Lee, Jihwan Kim, Kyung Whan Kim, Young Jun Ku, Dae Suk Kim, Chunseok Jeong, Tae Sik Yun, Hongjung Kim, Ho Sung Cho, Yeon Ok Kim, Jae Hwan Kim, Jin Ho Kim, Sangmuk Oh, Hyun Sung Lee, Ki Hun Kwon, Dong Beom Lee, Young Jae Choi, Jeajin Lee, Hyeon Gon Kim, Jun Hyun Chun, Jonghoon Oh, and Seok Hee Lee. 2016. 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)*. 318–319. https://doi.org/10.1109/ISSCC.2016.7418035

[44] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. 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)*. 43–56. https://doi.org/10.1109/ISCA52012.2021.00013

[45] Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Kornijcuk Vladimir, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Jaewook Lee, Donguc Ko, Younggun Jun, Keewon Cho, Ilwoong Kim, Choungki Song, Chunseok Jeong, Daehan Kwon, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. 2022. 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)*, Vol. 65. 1–3. https://doi.org/10.1109/ISSCC42614.2022.9731711

[46] Min Li, Yulong Ao, and Chao Yang. 2021. Adaptive SpMV/SpMSpV on GPUs for Input Vectors of Varied Sparsity. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2021), 1842–1853. https://doi.org/10.1109/TPDS.2020.3040150

[47] Shiqing Li, Di Liu, and Weichen LiuDi Liu. 2023. Efficient FPGA-Based Sparse Matrix–Vector Multiplication With Data Reuse-Aware Compression. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (2023), 4606–4617. https://doi.org/10.1109/TCAD.2023.3281715

[48] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109. https://doi.org/10.1109/LCA.2020.2973991

[49] Bowen Liu and Dajiang Liu. 2023. Towards High-Bandwidth-Utilization SpMV on FPGAs via Partial Vector Duplication. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference* (Tokyo, Japan) *(ASPDAC '23)*. Association for Computing Machinery, New York, NY, USA, 33–38. https://doi.org/10.1145/3566097.3567839

[50] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. 2015. Sparse Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[51] Marco Maggioni and Tanya Berger-Wolf. 2016. Optimization techniques for sparse matrix–vector multiplication on GPUs. *J. Parallel and Distrib. Comput.* 93-94 (2016), 66–86. https://doi.org/10.1016/j.jpdc.2016.03.011

[52] Sharad Malik and Pareesa Ameneh Golnari. 2019. Sparse Matrix to Matrix Multiplication: A Representation and Architecture for Acceleration. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Vol. 2160-052X. 67–70. https://doi.org/10.1109/ASAP.2019.00-28

[53] Uditnarayan Mandal and Arighna Deb. 2023. ReMCOO: An Efficient Representation of Sparse Matrix-Vector Multiplication. In *2023 IEEE Guwahati Subsection Conference (GCON)*. 01–06. https://doi.org/10.1109/GCON58516.2023.10183488

[54] Anirudh Maringanti, Viraj Athavale, and Sachin B. Patkar. 2009. Acceleration of conjugate gradient method for circuit simulation using CUDA. In *2009 International Conference on High Performance Computing (HiPC)*. 438–444. https://doi.org/10.1109/HIPC.2009.5433184

[55] Susumu Mashimo, Thiem Van Chu, and Kenji Kise. 2017. Cost-Effective and High-Throughput Merge Network: Architecture for the Fastest FPGA Sorting Accelerator. *SIGARCH Comput. Archit. News* 44, 4 (Jan. 2017), 8–13. https://doi.org/10.1145/3039902.3039905

[56] JEDEC Standard High Bandwidth Memory DRAM Memory. 2015. Dram specification. *Standard JESD235A* (2015).

[57] JEDEC Standard High Bandwidth Memory DRAM Memory. 2021. Dram specification. *Standard JESD235D* (2021).

[58] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 457–468. https://doi.org/10.1109/HPCA.2017.54

[59] Naveen Namashivavam, Sanyam Mehta, and Pen-Chung Yew. 2021. Variable-Sized Blocks for Locality-Aware SpMV. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 211–221. https://doi.org/10.1109/CGO51591.2021.9370327

[60] B. Neelima and Prakash S. Raghavendra. 2011. CSPR: Column Only SPARSE Matrix Representation for Performance Improvement on GPU Architecture. In *Advances in Parallel Distributed Computing*, Dhinaharan Nagamalai, Eric Renault, and Murugan Dhanuskodi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 581–595.

[61] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 68–78. https://doi.org/10.1109/IPDPS49936.2021.00016

[62] CUDA Nvidia. 2025. Cusparse library. *NVIDIA Corporation, Santa Clara, California* (2025).

[63] Chi-Sung Oh, Ki Chul Chun, Young-Yong Byun, Yong-Ki Kim, So-Young Kim, Yesin Ryu, Jaewon Park, Sinho Kim, Sanguhn Cha, Donghak Shin, Jungyu Lee, Jong-Pil Son, Byung-Kyu Ho, Seong-Jin Cho, Beomyong Kil, Sungoh Ahn, Baekmin Lim, Yongsik Park, Kijun Lee, Myung-Kyu Lee, Seungduk Baek, Junyong Noh, Jae-Wook Lee, Seungseob Lee, Sooyoung Kim, Botak Lim, Seouk-Kyu Choi, Jin-Guk Kim, Hye-In Choi, Hyuk-Jun Kwon, Jun Jin Kong, Kyomin Sohn,

Nam Sung Kim, Kwang-Il Park, and Jung-Bae Lee. 2020. 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)*. 330–332. https://doi.org/10.1109/ISSCC19947.2020.9063110

[64] Cristobal Ortega, Yann Falevoz, and Renaud Ayrignac. 2024. PIM-AI: A Novel Architecture for High-Efficiency LLM Inference. arXiv:2411.17309 [cs.AR] https://arxiv.org/abs/2411.17309

[65] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 724–736. https://doi.org/10.1109/HPCA.2018.00067

[66] Jaewon Park, Jae Hoon Lee, Sang-Kil Park, Ki Chul Chun, Kyomin Sohn, and Sungho Kang. 2021. An In-DRAM BIST for 16 Gb DDR4 DRAM in the 2nd 10-nm-Class DRAM Process. *IEEE Access* 9 (2021), 33487–33497. https://doi.org/10.1109/ACCESS.2021.3061349

[67] Myeong-Jae Park, Jinhyung Lee, Kyungjun Cho, Jihwan Park, Junil Moon, Sung-Hak Lee, Tae-Kyun Kim, Sanghoon Oh, Seokwoo Choi, Yongsuk Choi, Ho Sung Cho, Taesik Yun, Young Jun Koo, Jae-Seung Lee, Byung-Kuk Yoon, Young-Jun Park, Sangmuk Oh, Chang Kwon Lee, Seong-Hee Lee, Hyun-Woo Kim, Yucheon Ju, Seung-Kyun Lim, Kyo Yun Lee, Sang-Hoon Lee, Woo Sung We, Seungchan Kim, Seung Min Yang, Keonho Lee, In-Keun Kim, Younghyun Jeon, Jae-Hyung Park, Jong Chan Yun, Seonyeol Kim, Dong-Yeol Lee, Su-Hyun Oh, Jung-Hyun Shin, Yeonho Lee, Jieun Jang, and Joohwan Cho. 2023. A 192-Gb 12-High 896-GB/s HBM3 DRAM With a TSV Auto-Calibration Scheme and Machine-Learning-Based Layout Optimization. *IEEE Journal of Solid-State Circuits* 58, 1 (2023), 256–269. https://doi.org/10.1109/JSSC.2022.3193354

[68] Michail Pligouroudis, Rafael Angel Gutierrez Nuno, and Tom Kazmierski. 2020. Modified Compressed Sparse Row Format for Accelerated FPGA-Based Sparse Matrix Multiplication. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5. https://doi.org/10.1109/ISCAS45731.2020.9181266

[69] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. 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)*. 190–200. https://doi.org/10.1109/ISPASS.2014.6844483

[70] Mayra Z. Rodriguez, Cesar H. Comin, Dalcimar Casanova, Odemir M. Bruno, Diego R. Amancio, Luciano da F. Costa, and Francisco A. Rodrigues. 2019. Clustering algorithms: A comparative approach. *PLOS ONE* 14, 1 (01 2019), 1–34. https://doi.org/10.1371/journal.pone.0210236

[71] Yousef Saad. 2011. *Numerical methods for large eigenvalue problems: revised edition.* SIAM.

[72] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry Pileggi, and Franz Franchetti. 2019. 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* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 347–358. https://doi.org/10.1145/3352460.3358330

[73] Minseok Seo, Xuan Truong Nguyen, Seok Joong Hwang, Yongkee Kwon, Guhyun Kim, Chanwook Park, Ilkon Kim, Jaehan Park, Jeongbin Kim, Woojae Shin, Jongsoon Won, Haerang Choi, Kyuyoung Kim, Daehan Kwon, Chunseok Jeong, Sangheon Lee, Yongseok Choi, Wooseok Byun, Seungcheol Baek, Hyuk-Jae Lee, and John Kim. 2024. IANUS: Integrated Accelerator based on NPU-PIM Unified Memory System. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 545–560. https://doi.org/10.1145/3620666.3651324

[74] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 273–287. https://doi.org/10.1145/3123939.3124544

[75] Kristina P. Sinaga and Miin-Shen Yang. 2020. Unsupervised K-Means Clustering Algorithm. *IEEE Access* 8 (2020), 80716–80727. https://doi.org/10.1109/ACCESS.2020.2988796

[76] Kyomin Sohn, Won-Joo Yun, Reum Oh, Chi-Sung Oh, Seong-Young Seo, Min-Sang Park, Dong-Hak Shin, Won-Chang Jung, Sang-Hoon Shin, Je-Min Ryu, Hye-Seung Yu, Jae-Hun Jung, Hyunui Lee, Seok-Yong Kang, Young-Soo Sohn, Jung-Hwan Choi, Yong-Cheol Bae, Seong-Jin Jang, and Gyoyoung Jin. 2017. 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 Journal of Solid-State Circuits* 52, 1 (2017), 250–260. https://doi.org/10.1109/JSSC.2016.2602221

[77] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2022. Serpens: a high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 211–216. https://doi.org/10.1145/3489517.3530420

[78] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 689–702. https://doi.org/10.1109/HPCA47549.2020.00062

[79] Wai Teng Tang, Wen Jun Tan, Rick Siow Mong Goh, Stephen John Turner, and Weng-Fai Wong. 2015. A Family of Bit-Representation-Optimized Formats for Fast Sparse Matrix-Vector Multiplication on the GPU. *IEEE Transactions on Parallel and Distributed Systems* 26, 9 (2015), 2373–2385. https://doi.org/10.1109/TPDS.2014.2357437

[80] Yanzhe Tang, Zhongming Liu, Weibing Shang, Fengqin Zhang, Bernard Wu, Zhong Kong, Hongwen Li, Hong Ma, and Kanyu Cao. 2021. Pitch Device Design in 10nm-Class DRAM Process through DTCO. In *2021 IEEE 14th International Conference on ASIC (ASICON)*. 1–4. https://doi.org/10.1109/ASICON52560.2021.9620445

[81] Boyu Tian, Qihang Chen, and Mingyu Gao. 2023. ABNDP: Co-optimizing Data Access and Load Balance in Near-Data Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 3–17. https://doi.org/10.1145/3582016.3582026

[82] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. 2008. Random walk with restart: fast solutions and applications. *Knowledge and Information Systems* 14 (2008), 327–346. https://doi.org/10.1007/s10115-007-0094-2

[83] Yaman Umuroglu and Magnus Jahre. 2014. An energy efficient column-major backend for FPGA SpMV accelerators. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. 432–439. https://doi.org/10.1109/ICCD.2014.6974716

[84] Takuma Usui, Thiem Van Chu, and Kenji Kise. 2016. A Cost-Effective and Scalable Merge Sorter Tree on FPGAs. In *2016 Fourth International*

Symposium on Computing and Networking (CANDAR). 47–56. https://doi.org/10.1109/CANDAR.2016.0023

[85] Sheng Wang, Yuan Sun, and Zhifeng Bao. 2020. On the efficiency of K-means clustering: evaluation, optimization, and algorithm selection. *Proceedings of the VLDB Endowment* 14, 2 (Oct. 2020), 163–175. https://doi.org/10.14778/3425879.3425887

[86] Shih-Hung Weng, Quan Chen, and Chung-Kuan Cheng. 2012. Time-Domain Analysis of Large-Scale Circuits by Matrix Exponential Method With Adaptive Control. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 8 (2012), 1180–1193. https://doi.org/10.1109/TCAD.2012.2189396

[87] Shih-Hung Weng, Quan Chen, Ngai Wong, and Chung-Kuan Cheng. 2012. Circuit simulation via matrix exponential method for stiffness handling and parallel processing. In *Proceedings of the International Conference on Computer-Aided Design* (San Jose, California) *(ICCAD '12)*. Association for Computing Machinery, New York, NY, USA, 407–414. https://doi.org/10.1145/2429384.2429469

[88] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 570–583. https://doi.org/10.1109/HPCA51647.2021.00055

[89] Weidong Yang, Yuqing Yang, Shuya Ji, Jianfei Jiang, Naifeng Jing, Qin Wang, Zhigang Mao, and Weiguang Sheng. 2024. RecPIM: Efficient In-Memory Processing for Personalized Recommendation Inference Using Near-Bank Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 10 (2024), 2854–2867. https://doi.org/10.1109/TCAD.2024.3386117

[90] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2020. Speeding Up SpMV for Power-Law Graph Analytics by Enhancing Locality & Vectorization. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00090

[91] Chi Zhang, Paul Scheffler, Thomas Benz, Matteo Perotti, and Luca Benini. 2024. Near-Memory Parallel Indexing and Coalescing: Enabling Highly Efficient Indirect Access for SpMV. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. https://doi.org/10.23919/DATE58400.2024.10546797

[92] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing* (Vancouver, BC, Canada) *(HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/2600212.2600213

[93] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 544–557. https://doi.org/10.1109/HPCA.2018.00053

[94] Qiuling Zhu, Berkin Akin, H. Ekin Sumbul, Fazle Sadi, James C. Hoe, Larry Pileggi, and Franz Franchetti. 2013. A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing. In *2013 IEEE International 3D Systems Integration Conference (3DIC)*. 1–7. https://doi.org/10.1109/3DIC.2013.6702348

[95] Qiuling Zhu, Tobias Graf, H. Ekin Sumbul, Larry Pileggi, and Franz Franchetti. 2013. Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2013.6670336

[96] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 712–725. https://doi.org/10.1145/3352460.3358256