SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator

Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, Yuan Xie
Department of Electrical and Computer Engineering, UCSB, Santa Barbara, USA
Email: {xinfeng,liangzheng,peng_gu,abasak,leideng,
lingliang,xinghu,yuanxie}@ucsb.edu

***Abstract*—Sparse matrix-vector multiplication (SpMV) is an important primitive across a wide range of application domains such as scientific computing and graph analytics. Due to its intrinsic memory-bound characteristics, the performance of SpMV on throughput-oriented architectures such as GPU is bounded by the limited bandwidth between processors and memory. Processing-in-memory (PIM) architectures, made feasible by advances in 3D stacking, provide new opportunities to utilize ultra-high bandwidth by integrating compute-logic into memory.**

In this paper, we develop an SpMV accelerator, named as SpaceA, based on PIM architectures. SpaceA integrates compute-logic near memory banks to exploit bank-level bandwidth. SpaceA contains both hardware and data-mapping design features to alleviate irregular memory access patterns which hinder full utilization of high memory bandwidth. In terms of hardware design features, SpaceA consists of two unique features: (1) it utilizes the capability of outstanding memory requests to hide the memory access latency to data located in non-local memory banks; (2) it integrates Content Addressable Memory (CAM) at the bank level to exploit data reuse of the input vectors. In addition, we develop a mapping scheme that partitions the sparse matrix into different memory banks, to maximize the data locality of the input vector and to achieve workload balance among processing elements (PEs) near each bank. Overall, SpaceA together with the proposed mapping method achieves 13.54x speedup and 87.49% energy saving on average over the GPU baseline on SpMV computation. In addition to SpMV primitives, we conduct a case study on graph analytics to demonstrate the benefits of SpaceA for applications built on SpMV. Compared to Tesseract and GraphP, state-of-the-art graph accelerators, SpaceA obtains better performance due to its higher effective bandwidth provided by near-bank integration.

***Keywords*—SpMV, Accelerator, Processing-in-memory**

I. Introduction

Sparse matrix-vector multiplication (SpMV) is a special case of matrix-vector multiplication, where the input matrix contains a large number of zero elements. In many real-world application domains such as scientific computing [19], [51] and graph analytics [14], [72], [74], algorithms can be formulated as iterations of matrix-vector multiplication where the matrix is sparse and is reused across multiple runs. Multiple iterations of SpMV run until the convergence of the output vector form the primary bottleneck in these algorithms. The zero elements in the sparse matrix provide new opportunities for efficient storage and computation by skipping them, allowing them to overcome the bottleneck in these algorithms.

Because of the importance of SpMV across these application domains, efficient SpMV computing has been well-studied in traditional multi-core and many-core architectures over the last several decades. These studies mainly focus on efficient sparse matrix storage methods and corresponding parallel algorithm designs to balance workloads across processors, to exploit the locality of the input vector, and to reduce atomic operation overheads in output vector [47], [59], [60], [71]. In addition, several offline preprocessing techniques [37], [38], [54] have been developed to improve the performance of SpMV through optimized storage formats and partitioning methods. In the application scenario where the same sparse matrix is reused over multiple iterations, the overhead of offline preprocessing is well-amortized. In this paper, we first study state-of-the-art SpMV implementations in the vendor-provided library on NVIDIA GPU. Our profiling results reveal a high DRAM utilization, which indicates that SpMV has been well-optimized on GPU and the memory bandwidth becomes the bottleneck.

To provide a higher effective memory bandwidth, processing in memory (PIM) and near data processing (NDP) architectures introduce new opportunities by integrating compute-logic near memory. The recent advancement in integrated circuit (IC) process technology makes these architectures feasible, especially 3D stacking process technology [65]. Hybrid memory cube (HMC) [1] and high bandwidth memory (HBM) [2] are two promising memory designs that leverage the benefits of 3D stacking. Memory banks are stacked onto layers above the base logic die where memory control logic is fabricated, and different layers communicate using through-silicon vias (TSV). Such a stacked memory organization presents opportunities for PIM at two levels: 1) base logic die level and 2) memory bank level with minimal changes to the circuit design of bank groups. In comparison with the integration of compute-logic on the base die, compute-logic near memory banks is closer to data. Thus, access to local data has lower latency, higher bandwidth, and higher energy efficiency. Therefore,

This work was supported in part by NSF 1725447 and 1719160.

PIM architectures, especially bank-level logic integration, are promising for designing SpMV accelerators to overcome the bandwidth bottleneck in traditional multi-core and many-core designs. In this work, we develop SpaceA, an accelerator for SpMV based on near bank data processing. In SpaceA, we distribute the non-zero elements across the memory banks and operate on them using processing elements (PEs) near the banks.

Although PIM architectures provide higher effective bandwidth compared to the traditional memory interface between processors and memory, there are several challenges in accelerator designs. First, the memory latency to access data in other banks is much higher than in the local bank. The PE design should hide such a high latency for fully utilizing the bank level bandwidth. Second, since interconnect bandwidth is much smaller than that of the memory bank, memory access should be kept as local as possible to ease the burden on the interconnect. Third, PEs near the memory banks have a strict area budget, which requires the compute logic to remain fairly simple, but effective. In addition, challenges of workload balancing and locality exploitation of the input vector also exist when distributing non-zero elements across PEs.

Our accelerator, SpaceA, is designed to overcome these challenges. To overcome the first challenge, each PE near the memory bank possesses a queue to hold the non-zero elements for processing and memory requests to input vector according to the column index of non-zero elements in this queue. Memory requests are non-blocking to hide the memory access latency to other banks by exploiting memory-level parallelism (MLP). To address the second challenge, content addressable memory (CAM) is integrated at the bank level to cache elements from the input vector so that the amount of memory access to other banks is reduced by exploiting the locality. This helps alleviate the bandwidth pressure on the TSVs. The third challenge related to the strict area budget is tackled by the fact that our PE design only includes a queue and a floating-point unit (FPU). Therefore, our PE occupies a very small area overhead, which makes it practical to be integrated near the memory banks. In addition to these design options to overcome hardware challenges, we develop a mapping scheme for SpaceA to distribute the non-zero elements of the sparse matrix across different memory banks to achieve workload balance among PEs and to exploit the locality of data from the input vector.

In summary, our contributions are as follows:

- We design an accelerator, named SpaceA, to leverage outstanding memory requests to hide the memory access latency to non-local banks. To reduce the memory traffic to non-local banks, we integrate CAM buffers in SpaceA to exploit the locality of input vectors.
- We develop a mapping scheme for SpaceA to distribute the non-zero elements across different banks to achieve workload balance among PEs and to exploit the data locality of the input vector.
- Our evaluation of SpaceA with the proposed mapping scheme on matrices [19] from real-world applications reveals 13.5x speedup and 87.49% energy saving on average over the GPU baseline with only 4.86% area overhead. Additionally, our case study on graph applications demonstrates a better performance than state-of-the-art graph accelerators, Tesseract [4] and GraphP [76], because of the higher effective bandwidth provided by near-bank integration instead of placing compute-logic on the base die.

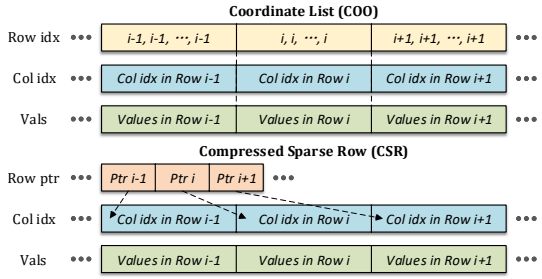

Fig. 1. The compressed sparse row (CSR) format of a sparse matrix.

II. Background and Motivation

A. SpMV Workloads

SpMV is a widely used operation in many applications that use algorithms based on a large amount iterations of matrix-vector multiplication in which the coefficient matrix is sparse. We denote an SpMV operation as $Y = Y + AX$ where X is the input vector, A is the input matrix, and Y is the output vector. We denote the dimensions of the input matrix as m and n, which indicate that the matrix has m rows and n columns. Additionally, we denote nnz as the number of non-zero elements. Each component of the output vector can be computed as $Y_i = Y_i + \sum_{j=0}^{n} A_{ij}X_j$ where Y_i is the i-th component of vector Y, X_j is the j-th component of vector X, and A_{ij} is the element located in i-th row and j-th column of matrix A. For a sparse matrix, the computation of $A_{ij}X_j$ can be skipped for locations where $A_{ij} = 0$.

For highly sparse matrices, compressed storage formats such as Coordinate List (COO) and Compressed Sparse Row (CSR) store the non-zero elements efficiently and remove ineffective computation for the zero elements. The COO format is composed of three lists of length nnz. These three lists store the row index, the column index, and the value, respectively, for each non-zero element. The CSR format, on the other hand, consists of three arrays: 1) row ptr, 2) col idx, and 3) vals. Each entry in the row ptr array points to an entry in the col ids array which represents the beginning of the list of column ids containing non-zero elements in that row. The row ptr entry simultaneously points to the entry in the vals array which records the value of the non-zero elements. Figure 1 demonstrates how non-zero elements of i-th row are stored.

Compared to COO, CSR is more compact since it saves the memory space of row index array from the length of nnz to the length of $m + 1$. Therefore, CSR is the most widely used sparse matrix format and $csrmv()$ [3] is supported in almost all libraries on multi-core and many-core architectures

to compute SpMV. SpaceA is designed to perform SpMV based on the CSR format.

B. SpMV on GPU

The poor reuse opportunity in the sparse matrix and irregular memory access patterns make SpMV memory-bound on multi-core and many-core processors. Compared to the CPU, GPU provides higher memory bandwidth through the GDDR memory bus and exploits memory-level parallelism to hide long memory access latency. To understand the state-of-the-art implementation of SpMV on GPU, we profiled SpMV computation with a collection of real-world matrices from the University of Florida sparse matrix collection [19]. The names, application domains, and characteristics of these matrices are elaborated upon Table I. For the implementation of SpMV on GPU, we use the library routine $csrmv()$ from the vendor-provided library cuSPARSE [3], which is a library optimized for sparse linear algebra operations on NVIDIA GPU. We measured the performance and profiled the DRAM metrics of SpMV on NVIDIA GPU, Titan Xp. DRAM read throughput is collected by $nvprof$. In addition, we measured effective read throughput which is computed as nnz times the size of a non-zero element over the measured execution time. Moreover, we compute the achieved GFLOPs of SpMV as nnz over the execution time, and the ALU utilization as the achieved GFLOPs over the maximum GFLOPs provided by GPU. Compared to the maximum DRAM bandwidth of Titan Xp, which is 547.8 GB/s, Figure 2 shows that the current average bandwidth utilization (as represented by the mean orange bar) of SpMV on GPU is 27.08% and 43.39% when excluding matrices 12, 13, and 14[1]. In addition, Figure 2 shows that the ALU utilization is only 2.68%. Figure 2 provides two important starting points for our work. First, the small ALU utilization compared to the much larger DRAM bandwidth utilization demonstrates the memory-bound behavior of SpMV, motivating our PIM-based architecture. Second, the effective bandwidth utilization (represented by the blue bar) is close to the actual bandwidth utilization (represented by the orange bar), which indicates that actual hardware innovation (rather than algorithmic innovation to eliminate redundant DRAM accesses) is required for higher performance SpMV.

C. 3D Memory

3D stacked memory such as HMC (the focus of this paper) and HBM are promising in terms of PIM architectures [65]. As shown in Figure 3(b), 3D stacking involves a base logic die with layers of DRAM dies stacked on top of it. A memory cube can be partitioned into vertical slices called vaults, each with private vertical connections through all layers physically realized with TSVs. Memory banks on a given layer are partitioned into bank groups (usually one bank group per vault per layer), and each bank group shares the same TSVs

[1]Exceptions represent social networks and web graphs which show relatively poorer utilization of the DRAM bandwidth, in agreement with prior studies [9], [10], [12].

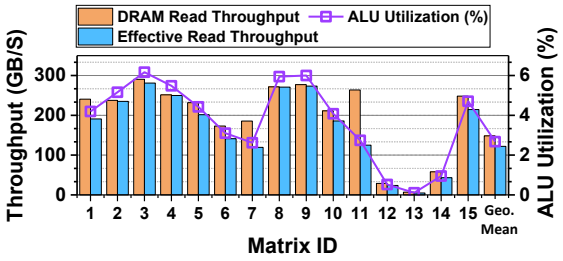

Fig. 2. Profiling results of SpMV on GPU (The details of each matrix are listed in Table I).

allowing them to communicate among the vault layers. Intra-vault communication occurs through the TSVs, and inter-vault communication occurs through Network-on-Chip (NoC) routers. The external interface for a memory cube is composed of four SerDes links.

In a typical HMC specification [1], there are 1024 TSVs in the same memory cube running at the bit rate of 2Gbps. Thus the TSVs of a cube provide bandwidth up to 256 GB/s. Bank-level bandwidth offers greater potential than the TSV bandwidth. Each memory bank has the interface to read or write 256 bits when data is in the row buffer for t_{CCD} cycles. Without bus turn-around overhead, t_{CCD} can be as small as 4 cycles. Therefore, 8GB/s bandwidth with a 1 GHz clock can be provided per bank-level interface. A memory cube with 16 vaults where each vault controls a stack of memory banks with 8 layers and 2 banks in a bank group has 256 memory banks. Thus, this memory cube can provide 2 TB/s internal bandwidth at the bank-level, which is 8 times than the internal bandwidth provided by TSV.

III. SpaceA Architecture

A. Overview

The architecture design of SpaceA is demonstrated in Figure 3. As shown in Figure 3(a) and 3(b), SpaceA is composed of several 3D stacked memory cubes connected in a memory network. To exploit bank-level memory bandwidth, SpaceA integrates a PE near every memory bank. The input/output vectors are evenly partitioned and stored in memory banks on the DRAM layer just above the base logic die, whereas the sparse matrix is statically distributed by the mapping algorithm (Section IV) on all the other DRAM layers. The separation of the storage allows each PE to process the sparse matrix in a streaming manner to maximize the read bandwidth. In addition, on the DRAM die for vectors, the elements with the same index from input and output vectors are stored in the same memory bank. This is because, in an iteration of SpMV, the output of i-th iteration is the input of $i+1$-th iteration. Therefore, this storage scheme can eliminate data movement between iterations for input and output vectors.

B. PE Design

In SpaceA design, there is a PE dedicated for each memory bank. Since matrix and vector data are separated into memory

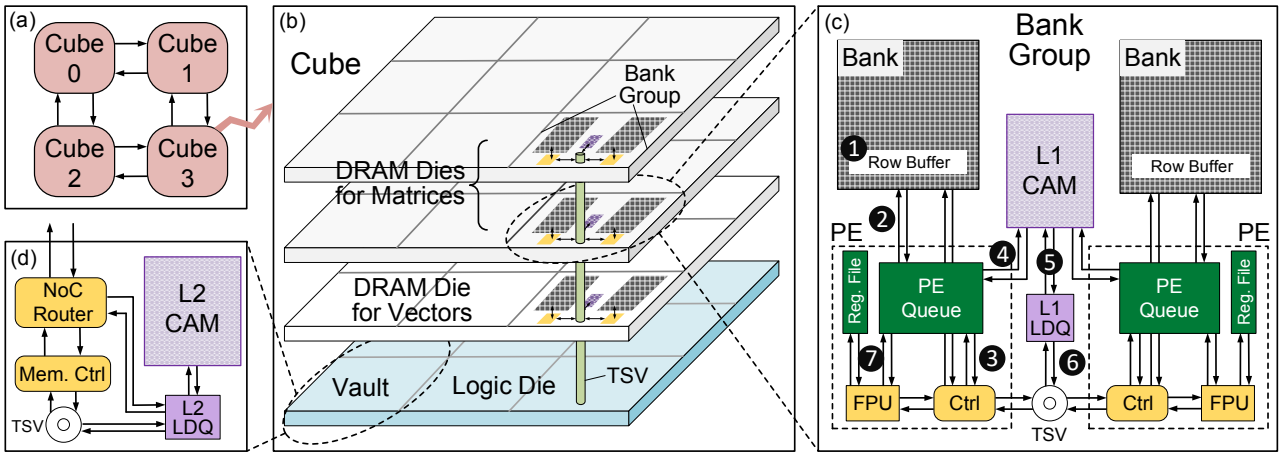

Fig. 3. SapceA architecture design: (a) memory cubes connected through memory network, (b) the overview of a cube, (c) components in a bank group, and (d) components in a vault controller.

banks on different dies, PEs attached to these memory banks have different functionalities. The PE of memory banks storing the sparse matrix computes partial dot-product results, while the PE of memory banks storing vectors accumulates the partial results which is finally stored into the output vector. We denote the first type of PE as **Product-PE** and the second type of PE as **Accumulation-PE**. Although these two types of PEs have different functionalities, they can be realized by the same set of hardware components. The hardware components of a bank group are shown in Figure 3(c). Following is the description of how these components are designed for the Product-PE and how they are used for the Accumulation-PE.

Product-PE: Product-PE is responsible for processing non-zero elements of the sparse matrix in its local memory bank. After t_{RAS} cycles, a DRAM row will be loaded into the row buffer of the memory bank (Figure 3(c)-❶). Using the similar idea of CSR matrix format, when distributing non-zero elements into memory banks, the mapping algorithm aligns the number of non-zero elements of a row into the size of a DRAM row. This alignment causes non-zero elements of the same DRAM row to end in the same row index in the original sparse matrix. As a result, when storing non-zero elements in a DRAM row, the leading 4 bytes are used to indicate the row index of non-zero elements in this DRAM row, and the rest of the space is used to store pairs of column index and value.

As shown in Figure 3(c)-❷, non-zero elements from a DRAM row buffer are pushed into a PE queue if the PE queue is not full. The PE queue is physically realized with scratchpad memory while the control logic in Product-PE accesses elements inside it as a logical cyclic queue. For each non-zero element in the PE queue, it needs to compute the partial result $A_{ij}X_j$ where the row index i, column index j, and value A_{ij} are already known. The control unit scans the PE queue in a cyclic manner when it is not empty, and then processes a non-zero element every L_p cycle (Figure 3(c)-❸). For each unprocessed non-zero element, it will check whether X_j is ready in the register file.

Case I: X_j is not ready: When X_j is not ready, the Product-PE needs to read it from other memory banks because vectors are stored separately from the matrix. The access latency to a remote bank is significantly larger than the access to its local memory. To exploit the locality of the input vector, SpaceA integrates an L1 CAM for PEs in the same bank group. This L1 CAM provides a key-value store so that it can help to search the value X_j according to the column index j (Figure 3(c)-❹). When the access to L1 CAM misses, it will return a miss signal which indicates that the remote access is unavoidable. To hide the latency of remote access, the control unit will continue to process the next element in the PE queue instead of waiting for the value X_j. Since this is a logical cyclic queue, the control unit will access this non-zero element again after scanning the rest of non-zero elements in the PE queue. Meanwhile, L1 CAM will send the requested column index j to the load queue (LDQ) (Figure 3-❺) to remove the duplication of data requests. If this column index has not been requested yet, it will send out the request through TSV (Figure 3(c)-❻). When the requested value X_j comes back, it will be written into both L1 CAM and register file. The corresponding load request j will be removed from the load queue. Since the control unit repeatedly iterates through all elements in the PE queue, X_j will become ready when the control unit accesses it again after the requested value comes back.

Case II: X_j is ready: When X_j is ready in the register file, it will send A_{ij}, X_j, and the current partial result Y_i to Floating-point Unit (FPU) for computing $Y_i = Y_i + A_{ij}X_j$ (Figure 3(c)-❼). After accumulating the partial result into Y_i, the non-zero element is labelled as processed. When all of non-zero elements from the same DRAM row in the front of the PE queue are processed, they are popped out of the queue and the control unit moves the front pointer of the queue. The granularity for popping non-zero elements is the same size of a DRAM row buffer so that the whole row buffer of new data

can be pushed into the PE queue, and checked as to whether the row index of this new row is different from the existing row index. The partial result Y_i is flushed out through TSV when the new row index is different from the existing row index.

Accumulation-PE: Bank groups with Accumulation PE serve two purposes. First, since the memory banks of this bank group store some elements of the input vector, it will respond the value X_j according to the requested column index j. For this purpose, the request first goes to L1 CAM, and then goes to the memory bank if X_j is not in the L1 CAM (CAM miss). This part only needs the help of the memory bank, L1 CAM, and control unit. Second, since the memory banks of this bank group store some elements of the output vector, they need to accumulate partial results Y_i. To achieve this purpose, the SRAM of the PE queue is used to realize an update buffer where the elements of the output buffer are stored. When Y_i comes, it will first look up the output buffer by the row index stored in the register file. If corresponding output elements are not in the update buffer, it will be loaded into the update buffer from the memory bank. Then existing Y_i and new partial result Y_i will be accumulated with the help of the FPU. When the update buffer is full, it will write the logical first row back to the memory bank, and load a new row containing Y_i from the memory bank.

C. Vault Controller

The components of a vault controller on the base die are shown in Figure 3(d). In addition to the existing NoC router for inter-vault communication and the memory controller to read and write memory banks attached to the same TSVs, SpaceA integrates a L2 CAM and a corresponding load queue to exploit the locality of the input vector in the communication path between bank groups. There are three types of packets a vault controller could potentially process.

Type I: X_j request. When the vault controller receives the request for the value of X_j, it will first look up the L2 CAM according to the column index j. If X_j exists in L2 CAM, the vault controller will generate a response packet with the value of X_j, and send it back to the source of the request packet, either by NoC router to other vaults or TSV to bank groups attached to the same TSV. If X_j does not exist in L2 CAM, it will look up the load queue (LDQ) to remove duplicated requests for X_j. The vault controller will forward this request to the bank group storing X_j according to the column index j by either the NoC router or the TSV.

Type II: X_j response. When the vault controller receives the response for the value of X_j, the vault controller will have the same logic as the Product-PE hearing back the value of X_j. Besides forwarding this packet to its destination, the vault controller will write the value X_j into its L2 CAM and remove the corresponding entry in the load queue.

Type III: Y_i partial result: The vault controller for partial result Y_i will forward it to the corresponding vault storing Y_i according to the row index i. If Y_i is stored in the same vault, it will forward it to the bank groups on the bottom of DRAM dies by TSVs so that the partial result of Y_i can be accumulated with the help of the Accumulation-PE.

Algorithm 1 Row assignment to logical PEs.

```
Init nnz = nnz / #PEs
Init K_p = a large constant value
for pid = 0 to #PEs do
    Init the set of assigned rows, R_pid = ∅
    Init the set of unique column indexes, COL_pid = ∅
    Init the number of assigned non-zero elements, W_pid = 0
end for
for i = 0 to m do
    N_i: the number of non-zero elements in i-th row
    C_i: the set of column indexes in i-th row
    for pid = 0 to #PEs do
        if W_pid + N_i > nnz then
            Score_pid = −(W_pid + N_i − nnz) × K_p
        else
            Overlap = |C_i ⋂ COL_pid|
            Score_pid = max{Overlap/N_i, 1/(W_pid + N_i)}
        end if
    end for
    maxID = the pid with highest Score_pid
    R_maxID = R_maxID ⋃ {i}
    COL_maxID = COL_maxID ⋃ C_i
    W_maxID = W_maxID + N_i
end for
```

IV. Mapping Method

A. Overview

The proposed mapping method distributes the non-zero elements of a sparse matrix into the memory banks of SpaceA for Product-PE to process. There are two overall metrics to efficiently use SpaceA hardware: 1) workload balance and 2) locality. First, since all PEs process non-zero elements in parallel, the performance is bounded by the slowest PE, which requires workload balance among PEs. Second, since SpaceA integrates L1 CAM at the bank group level and L2 CAM at the base die of each vault to mitigate the latency of accessing the data of input vector, non-zero elements assignment should consider the column index locality of non-zero elements to leverage L1 and L2 CAM for keeping access local.

To achieve workload balance and leverage the locality of the input vector, we design the overall mapping pipeline shown in Figure 4, which takes the hardware configuration of SpaceA and the sparse matrix as the input. As shown in Figure 4, the first phase assigns different rows to PEs, which are logical PEs without any physical location information. In this phase, we exploit the intra-PE locality by assigning the rows of non-zero elements with similar column index pattern to the same PE. Meanwhile, this phase also balances the number of non-zero elements assigned to PEs. The algorithm used in this phase is further introduced in Section IV-B. In the second phase, we place all logical PEs into the physical location in SpaceA. This phase clusters PE workloads with similar sets of column index from the non-zero elements, and minimizes

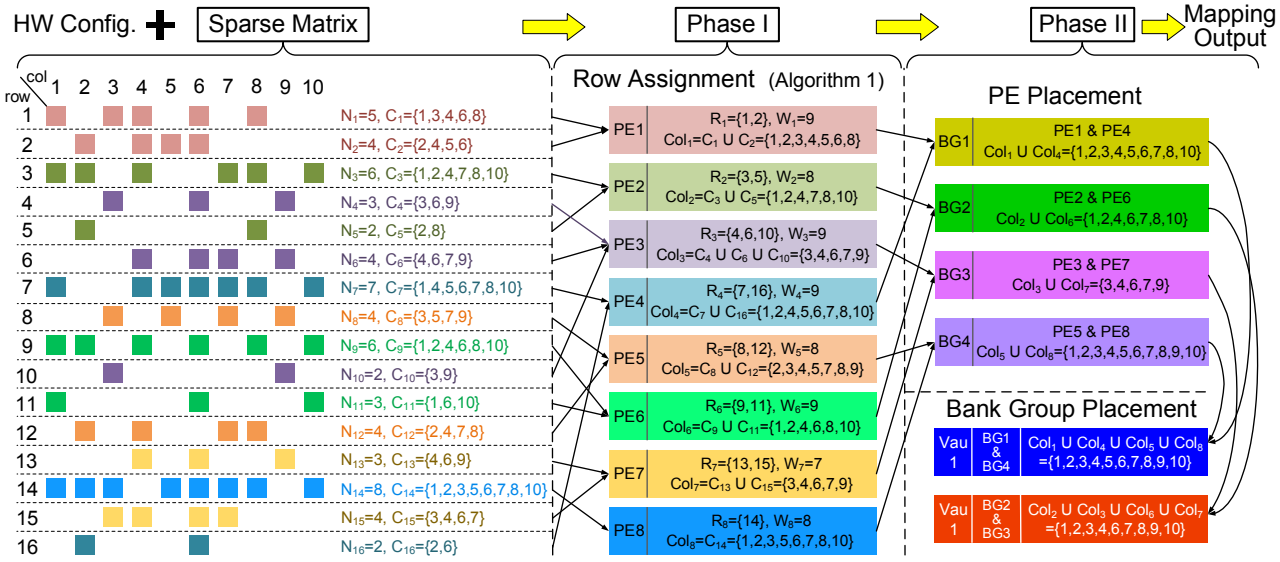

Fig. 4. The flow of our mapping algorithm which is composed of two phases: row assignment to logical PEs (Phase I) and PE placement to bank groups and vaults (Phase II).

the maximal number of unique column indexes across bank groups and vaults to achieve workload balance. The formulated optimization problem is detailed in Section IV-C.

B. Logical PE Workload

The first phase assigns multiple rows of the sparse matrix into PEs, each of which is considered equivalent. In this phase, we balance the workload among PEs and maximize the intra-PE locality. Algorithm 1 shows the scheme, which iterates through all rows and determines which PE is the best to be assigned for a specific row according to the current assignment of previous rows. The metric used to determine the best PE for processing this row is designed according to the following two principles. First, if the current row i assigned to the current PE pid makes the current PE process the number of elements larger than $\overline{nnz}$, we add a penalty for the number of elements exceeding this budget. The budget $\overline{nnz}$ is computed as nnz over the total number of logical PEs. When each PE processes $\overline{nnz}$ elements, the workloads of PEs are perfectly balanced. Second, the column index overlap between non-zero elements of the current row i and the non-zero elements of existing rows assigned to the current PE pid is computed. In case of an overlap, the overlap ratio is taken as the score, which is the number of overlap non-zero elements over the number of non-zero elements of row i. When there is no overlap, the factor one over the number of non-zero elements assigned to the current PE is taken as the score. This score rating metric means we optimize locality first and the workload balance when the number of non-zero elements does not exceed the given budget $\overline{nnz}$. After computing the score of each PE, the row i is assigned to the PE with the highest score.

Although mapping the sparse matrix optimally to logical PEs is an NP-hard problem, our mapping heuristic is feasible in terms of time complexity. We denote P as the number of PEs, W_{pid} as the number of non-zero elements assigned to PE pid, and N_i as the number of non-zero elements in the row i. Each row needs the time complexity of $O(N_i \sum_{pid=1}^{P} logW_{pid})$. Since W_{pid} is always smaller than nnz (the total number of non-zero elements), the upper bound of the time complexity of a row assignment can be simplified as $O(N_i Plognnz)$. Summing up the time complexity of assigning all rows, since $\sum_{i=1}^{m} N_i = nnz$, the time complexity for finishing all row assignments has an upper bound $O(P \times nnzlognnz)$. This time complexity is scalable in terms of the number of PEs and the number of non-zero elements. Therefore, the algorithm of this phase is practical enough, and its effectiveness is further demonstrated in Section V-C.

C. Logical PE Placement

In this phase, each logical PE is placed into the position of a physical PE. We decouple this phase into two stages. First, logical PEs are clustered into bank groups. Second, bank groups are clustered into vaults. To achieve locality and workload balance, this phase minimizes the maximal number of unique column indexes across bank groups and vaults when clustering banks and bank groups. Both stages, clustering logical PEs and bank groups, represent similar problems in terms of problem structure and optimization target. Therefore, we abstract the problem of both stages as follows. Given p sets $S_1, S_2, ..., S_p$, we divide them evenly into q groups, and each group has the same number of sets k where $p = kq$. Therefore, we denote C_{gw} as the w-th set assigned to the group g. The value of C_{gw} should be one of the values between 1 and p. To optimize locality, we want sets assigned to the same group to have a larger overlap. Locality indicates the preference to assign sets with a larger number of overlap while

workload balance implies that the maximal number of unique elements should be minimized. The problem is formulated as Formula 1 where $F(C)$ stands for the maximum number of unique elements across all groups under the assignment C.

$$\begin{aligned} \underset{C}{\text{minimize}} \quad & F(C) \\ \text{subject to} \quad & F(C) = \max_{1 \le g \le q} \{ | \bigcup_{w=1}^{k} S_{C_{gw}} | \}, \\ & C_{gw} \in \{1, 2, ..., p\},\ \forall 1 \le g \le q,\ 1 \le w \le k \\ & C_{g_1 w_1} \neq C_{g_2 w_2},\ \forall (g_1, w_1) \neq (g_2, w_2) \end{aligned} \tag{1}$$

In the first stage, p equals the number of logical PEs and q equals the number of bank groups while in the second stage, p equals the number of bank groups and q equals the number of vaults. The formulated problem is also an NP-hard problem, thus we use a heuristic algorithm similar to Algorithm 1 to solve it. The effectiveness of the mapping algorithm is quantitatively shown in Section V-C.

V. Evaluation

In this section, we first introduce the experimental setup in Section V-A. Next, we detail the overall performance, power, and area results of our design compared to state-of-the-art SpMV implementations on GPU in Section V-B. Section V-C demonstrates the advantages of our proposed mapping methods. Section V-D shows the sensitivity studies of SpaceA performance to hardware configurations. Section V-E studies the scalability of SpaceA design. Finally, we conduct a case study of using SpaceA to accelerate graph analytics to show its potential for benefiting applications built on SpMV.

A. Evaluation Methodology

Workload. We evaluate SpaceA by executing SpMV using fifteen real-world matrices from various application domains including scientific computing and graph analytics. These matrices come from the University of Florida collection [19], and they are used in prior studies for accelerating SpMV on GPU [60] and Intel Xeon Phi processors [61]. In terms of the distribution of non-zeros, these matrices cover both structural patterns (i.e. a smaller standard deviation of the number of non-zeros in each row) and non-structural patterns (i.e. a larger standard deviation of the number of non-zeros in each row). The details of these matrices are listed in Table I.

Hardware Configuration. We adopt an HMC-like [1] design to realize the architecture design of SpaceA. The rest of the evaluation results assume an HMC-like architecture, a detailed further discussion between HMC and HBM technology can be found in Section VII. We use an HMC configuration specified in the prior HMC characterization study [28]. Specifically, a memory cube has 16 vaults that use 1024 TSVs running at the bit rate of 2 Gbps to communicate with 8 stacked DRAM die layers. Each bank group has 2 banks; each bank has a capacity of 128 Mb with a 2 Kb row buffer. Therefore, there are 256 memory banks in a memory cube with a total of 4 GB capacity, and a memory cube has a footprint of $48mm^2$. We use NVIDIA Titan Xp as a representative of GPU architecture for comparison which has processors with a die size 471 mm^2, an area equivalent to that of 10 cubes. We assume that the area of GPU DRAM dies is comparable to processors in Titan Xp, thus the default configuration of SpaceA uses 16 cubes, occupying 768 mm^2 – a similar area footprint as Titan Xp. Inside each PE, there is a 16 Kb scratchpad memory for the PE queue, which enables the PE to process non-zero elements from 8 DRAM rows concurrently. Register file has the same size as the number of non-zero elements stored in a PE queue. To support double-precision SpMV in scientific computing, each PE includes a floating-point unit (FPU). PEs from the same bank group share an L1 CAM with 32 sets and 4 ways per set. Each way in L1 CAM has 32 bytes, which is equivalent to the size of 4 input vector elements. The configuration of the number of ways per set and the size of each way in L2 CAM is the same as L1 CAM for simplicity, whereas L2 CAM has a larger number of sets, which is 2048 by default. The size of L1 and L2 CAM are 4 KB and 256 KB respectively. The load queues for L1 and L2 CAM are used to remove duplicate requests, and they are realized with fully associated CAM which have the sizes of 512 and 8192 elements respectively. The default configuration of L1 and L2 CAM is an intuitive design point; a detailed sensitivity for L1 and L2 CAM will be demonstrated in Section V-D to further justify our design point.

Simulation Method. We develop an event-based in-house simulator for the performance and power simulation. The performance simulation is based on triggering events according to the behavior of each hardware component described in Section III. The triggered events are simulated to happen after a deterministic latency of the event triggering it which is based on the latency model of each hardware component. The events in the performance simulator cover FPU computation, the read/write to DRAM banks, on-chip SRAM (register file, PE queue, L1 CAM, L1 load queue, L2 CAM, and L2 load queue), TSV, and NoC packet transfer. Additionally, our simulator maintains a data structure tracking values stored in DRAM banks and on-chip SRAM when simulating each event, and some events will modify this data structure. At the end of the simulation, the correctness of the event triggering mechanism is validated by the values of the output vector.

After validating the event triggering mechanism, the fidelity of our performance simulation relies on the latency of each event. Therefore, we use an existing well-validated simulator CACIT-3DD [15] and tape-out FPU design [23] to provide the latency model of each hardware component. Specifically, CACTI-3DD provides the access latency for DRAM banks, on-chip SRAM, and data transfer via TSV. Prior work [23] provides the latency of the FPU design.

Our event-based simulator logs a detailed event trace including read/write transactions to DRAM banks and on-chip SRAM, TSV data transfer, and FPU computation. Meanwhile, CACTI-3DD provides the energy consumption for each read/write transaction, TSV data transfer, and the static power of these components. FPU design [23] provides both dynamic and static power. Finally, we estimate the total energy con-

TABLE I

THE INFORMATION OF SPARSE MATRICES USED TO EVALUATE SPMV ON GPU AND SPACEA. THE NUMBER OF NON-ZERO ELEMENTS (nnz), AVERAGE NUMBER OF NON-ZERO ELEMENTS PER ROW (μ), AND THE STANDARD DEVIATION OF THE NUMBER OF NON-ZERO ELEMENTS IN EACH ROW (σ) ARE SHOWN TO REFLECT THE PATTERN OF NON-ZERO ELEMENTS DISTRIBUTION.

ID	Matrix	Domain	Dimensions	nnz	μ	σ
1	bcsstk32	Structural Problem	44609 x 44609	2014701	45.16	15.48
2	cant	2D/3D Problem	62451 x 62451	4007383	64.17	14.06
3	consph	2D/3D Problem	83334 x 83334	6010480	72.13	19.08
4	crankseg_2	Structural Problem	63838 x 63838	14148858	221.64	95.88
5	ct20stif	Structural Problem	52329 x 52329	2600295	51.57	16.98
6	lhr71	Chemical Process Simulation Problem	70304 x 70304	1494006	21.74	26.32
7	ohne2	Semiconductor Device Problem	181343 x 181343	6869939	61.01	21.09
8	pdb1HYS	Weighted Undirected Graph	36417 x 36417	4344765	119.31	31.86
9	pwtk	Structural Problem	217918 x 217918	11524432	53.39	4.74
10	rma10	Computational Fluid Dynamics Problem	46835 x 46835	2329092	50.69	27.78
11	shipsec1	Structural Problem	140874 x 140874	3568176	55.46	11.07
12	soc-sign-epinions	Directed Weighted Graph	131828 x 131828	841372	6.38	32.95
13	Stanford	Directed Graph	281903 x 281903	2312497	8.20	166.33
14	webbase-1M	Weighted Directed Graph	1000005 x 1000005	3105536	3.11	25.35
15	xenon2	Materials Problem	157464 x 157464	3866688	24.56	4.07

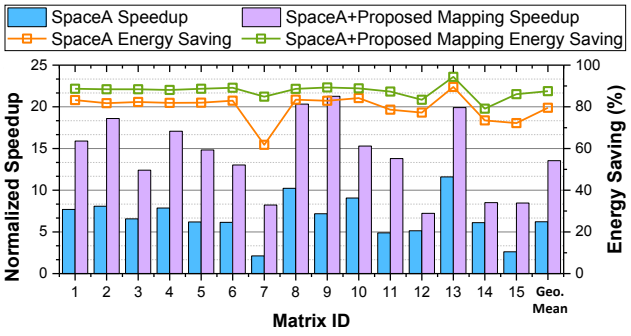

Fig. 5. Overall speedup and energy savings w.r.t GPU.

sumption by accumulating the energy needed for each activity and the energy spent in the static power.

B. Overall Performance, Power, and Area

Figure 5 shows the performance and energy efficiency for both our architecture design and the proposed mapping algorithm. As shown in Figure 5, the architecture design of SpaceA obtains 6.22x speedup and reduces the energy consumption by 4.89x (79.55% energy saving) on average compared to the GPU baseline. The results of SpaceA shown in Figure 5 uses a naive mapping which randomly assigns rows from the sparse matrix to PEs, so the performance and energy efficiency benefits mainly come from the advance of the architecture design. Figure 5 also demonstrates the overall performance energy efficiency with the proposed mapping method. SpaceA with the proposed mapping achieves 13.54x speedup and reduces 7.99x energy consumption (87.49% energy saving) on average compared to the GPU baseline. The comparison between the results of SpaceA using two mapping methods reveals that our proposed mapping method contributes 2.18x speedup and saves 1.63x energy consumption over the naive mapping method.

We estimate the area of the hardware components needed by SpaceA in addition to the existing HMC memory with

TABLE II

THE AREA AND POWER DENSITY OF COMPONENTS IN A BANK GROUP.

Component	Area	Power Density
PE Queue (x2)	0.0048 mm^2	43.75 mW/mm^2
Register File (x2)	0.0058 mm^2	49.66 mW/mm^2
PE Logic (x2)	0.0994 mm^2	28.21 mW/mm^2
L1 CAM (4 KB)	0.0286 mm^2	66.56 mW/mm^2
L1 Load Queue	0.0072 mm^2	56.29 mW/mm^2
Total / Peak	0.1458 mm^2	66.56 mW/mm^2

CACTI-3DD [15] and an existing FPU design [23]. These hardware components are assumed to be fabricated in the 22 nm technology. According to prior studies [73], the area of compute-logic fabricated in the DRAM process could be up to 2x larger than the one fabricated in the CMOS process due to the less number of metal layers. Thus we multiply all area results from CACTI-3DD and existing FPU design by 2x to estimate the area of these components in the DRAM process.

The area of hardware components in a bank group is shown in Table II. As shown in Table II, SpaceA only has an area overhead of 0.1458 mm^2 on the bank group level, which is only 4.86% of the area of a bank group and 5.96% of the area of memory banks. Thus the design of SpaceA has very little area overheads when integrating PE with memory banks. We estimate the area of L2 CAM and L2 load queue which reside on the base die in a similar way. In the default configuration, the area of an L2 CAM is 0.1898 mm^2 and the area of an L2 load queue is 0.0760 mm^2. The area of these two components is 0.2658 mm^2 in total, which is 8.86% area of a vault. The base die in the vanilla HMC memory has 10% to 30% area budget where other prior work integrates compute-logic [4], [24]. As long as the area of components on the base die does not exceed this budget, these components do not introduce any area overhead. In our work, we conservatively assume that the area budget on the base die is only 10%, thus the area of our L2 CAM and load queue is still within such a conservative area budget.

Recent research studies [73] and industrial prototypes [62]

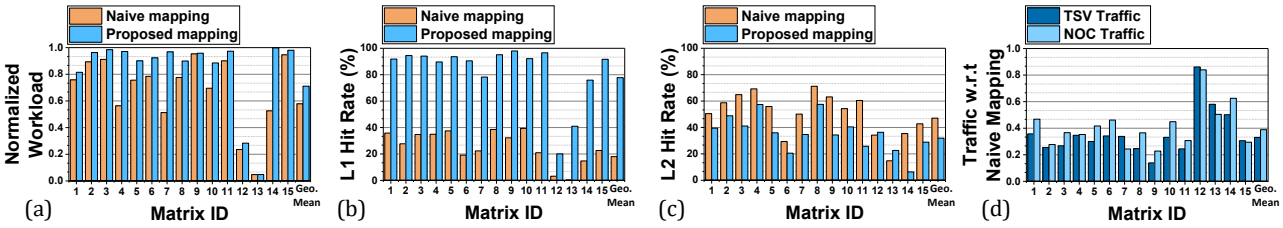

Fig. 6. Performance metric comparisons between the naive mapping and our proposed mapping: (a) normalized workload, (b) L1 CAM hit rate, (c) L2 CAM hit rate, and (d) traffic between bank groups and vaults (TSV and NoC) with respect to that of the naive mapping.

have demonstrated the feasibility of fabricating compute-logic in the DRAM process. However, the thermal issue is still a well-known challenge for PIM architecture based on 3D memory [20], [34]. We demonstrate the power density of components on DRAM dies in Table II. As shown in Table II, the peak power density per footprint is 532.48 mW/mm^2 (66.56 mW/mm^2 x 8 layers), which is under the constraint of power density from both commodity server active cooling [46] (706 mW/mm^2) and high-end server active cooling [20] (1214 mW/mm^2).

C. Mapping

In this section, we discuss performance metrics and power breakdown in detail in order to gain a better understanding of the source of these performance and energy efficiency benefits from our proposed mapping method.

Workload Balance. Since the performance of SpMV in SpaceA is bounded by the slowest PE, one goal of the proposed mapping method is to balance workloads among PEs. To quantify the workload balance, we do the following. First, we define the amount of work done by a PE to be the number of non-zero elements processed by it. Next, we define *normalized workload*, which indicates the ratio of the average amount of work done across all the PEs and the maximum amount of work done by any single PE. We use the normalized workload to represent the quantitative metric for workload balance (higher the better). The choice of the denominator in this ratio calculation is explained by the fact that workload balance is bottlenecked by the slowest PE, i.e., the PE which does the largest amount of work. In the ideal case where non-zero elements are evenly distributed among PEs, the normalized workload should equal one due to the equivalence between the average and the maximum PE workloads. The difference of normalized workload between the naive mapping and the proposed mapping is shown in Figure 6(a). Figure 6(a) shows that the normalized workload of the naive mapping is only 81% of that of the proposed mapping on average, which indicates that the maximum PE workload in the proposed mapping is only 81% of that in the naive mapping. The smaller maximum PE workload demonstrates a better workload balance in the proposed mapping.

Locality Improvement. In the flow of our proposed mapping method, we consider locality optimization. To demonstrate the locality improvement in the proposed mapping, we profile the hit rate of both L1 CAM and L2 CAM, the traffic on TSV for intra-vault communication, and the traffic on NoC for inter-vault communication. Since intra-vault communication through TSVs has a uniform latency while inter-vault communication through NoC has non-deterministic latency, we define the traffic of TSV as the amount of data transferred through TSVs and the traffic of NoC as the size of a packet multiplied by the distance between the source and the destination of the packet. Figure 6(b)-(d) demonstrate these profiling results. Overall, Figure 6(d) shows that the traffic on TSV and NoC is only 33.11% and 38.89% with respect to that of the naive mapping, which indicates a significant amount of communication savings resulted from the improvement of the locality. In details, Figure 6(b) shows that the proposed mapping improves the average L1 CAM hit rate of all L1 CAMs significantly from 18% and 78% on average while Figure 6(c) shows that the L2 CAM hit rate decreases in the proposed mapping from 47.09% to 31.93%. The main reason for the decreasing L2 CAM hit rate comes from the reduction of requests to L2 CAM with the same amount of cold miss. As a result, the saving of NoC traffic is less than the saving of TSV traffic.

Energy Breakdown. To understand the energy efficiency between the naive mapping and the proposed mapping method, we demonstrate the energy consumption breakdown for these two mapping methods. We normalize the energy consumption of different parts into the energy consumption of DRAM dynamic power mapped by the naive mapping. We divide the overall energy consumption into four parts. The first part is the DRAM dynamic power. The second part is the dynamic power of PE, L1 CAM with its load queue, and L2 CAM with its load queue. The third part is the dynamic power of interconnect, which includes TSV and NoC. The last part is the static power of the whole chip. The energy breakdown of these four parts for the naive mapping and the proposed mapping is shown in Figure 8. We have several observations from Figure 8. First, the dynamic power of hardware components added by SpaceA design is negligible (PE & L1 & L2 dynamic). Second, 65.55% on average of the dynamic power of interconnect is saved by the proposed mapping, which is the result of a reduced traffic amount on TSV and NoC shown in Figure 6(d). Finally, the proposed mapping method saves 54.05% energy consumption of the static power part: the result of improved performance. The static power dominates the overall energy consumption in

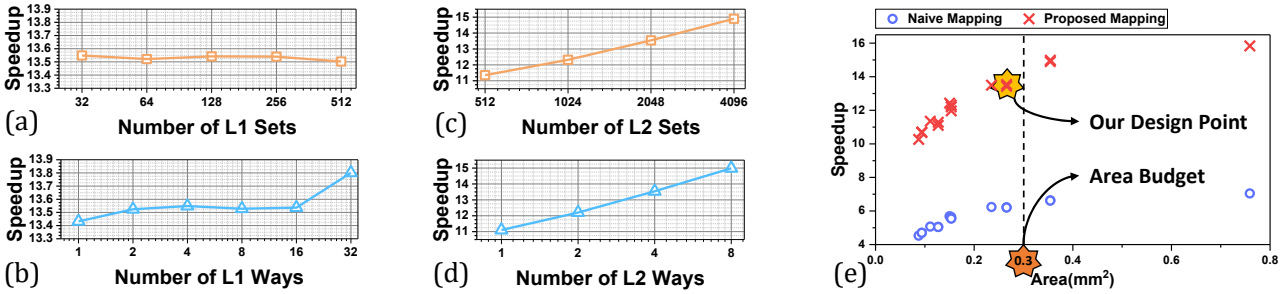

Fig. 7. The sensitivity of performance to (a) the number of L1 sets, (b) the number of L1 ways, (c) the number of L2 sets, and (d) the number of L2 ways. (e) The trade-off between performance and area in L2 CAM design.

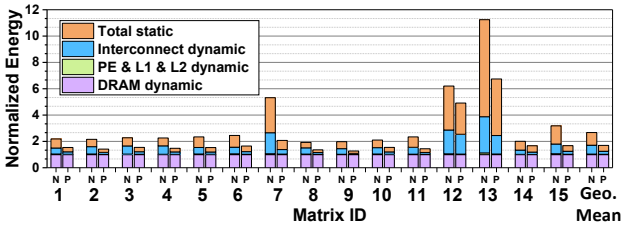

Fig. 8. The energy consumption breakdown of SpaceA for the naive mapping (denoted as N) and our proposed mapping (denoted as P).

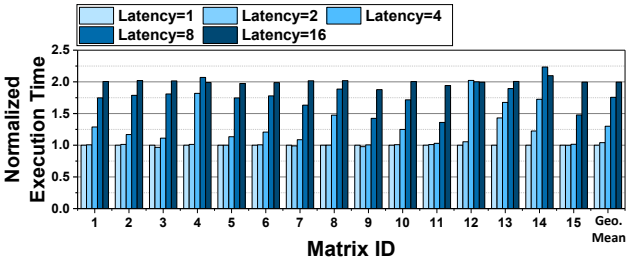

Fig. 9. The sensitivity of performance to TSV transfer latency.

matrix 7, 12, and 13. The energy consumption of static power is saved in the proposed mapping method of the matrix 7 due to a 3.87x speedup over the naive mapping. Matrix 12 and 13 have a relatively poor access pattern, thus pushing heavy traffic in the interconnect and resulting in a long execution time while DRAM banks and PEs are idle in most of the cycles.

D. Sensitivity Study

We conduct sensitivity studies for L1 CAM, L2 CAM, and TSV transfer latency to justify the selected design points in SpaceA architecture.

L1 and L2 CAM Sensitivity Study. We study the performance sensitivity of L1 and L2 CAM by varying either the number of sets or the number of ways. The average speedups compared to GPU for different numbers of sets and different numbers of ways in L1 and L2 CAM are shown in Figure 7(a)-(d). As shown in Figure 7(a) and (b), the performance of SpaceA is not sensitive to the size of L1 CAM. Although varying the number of ways could help the average speedup from 13.43 to 13.80, the benefit from such a large number of ways is relatively insignificant. Therefore, we keep the number of sets as small and the number of ways as large, resulting in the design with an L1 CAM composed of 32 sets and 4 ways per set. As shown in Figure 7(c) and (d), the performance is moderately sensitive to L2 CAM settings. Although the changes of speedup are not significant, these speedup changes are still noticeable, from 11x to 15x, among different CAM settings. Since L2 CAM can be as large as within the area budget, we study the trade-off between the performance and L2 CAM area as shown in Figure 7(e). Figure 7(e) shows that a larger L2 CAM usually result in a better speedup. Thus we select the largest one under our area budget, 10% area of a vault. Figure 7(e) also shows that our proposed mapping algorithm can leverage a smaller L2 CAM while achieving a better performance compared to the naive mapping. The naive mapping with an L2 CAM as large as 0.76 mm^2 achieves only a 68.61% speedup of the proposed mapping with an L2 CAM as small as 0.09 mm^2. The results further demonstrate the advantage of our proposed mapping method in terms of efficient hardware resource usage.

TSV Sensitivity Study. Most of PIM architecture design based on 3D memory technology leverages the low latency of TSV data transfer. We conduct a sensitivity study for TSV latency by varying the latency setting in our performance simulator. Figure 9 shows the performance slowdown of different TSV data transfer latency. Figure 9 shows that there is little difference between the latency of 1 cycle or 2 cycles for most of matrices. For the scenario where TSV transfer is 4 cycles, some matrices are not affected significantly (within 10% performance slowdown) while some matrices exhibit significant performance slowdown up to 2x. Thus the average slowdown of the performance is 1.3x, a factor which can hardly be ignored. When the TSV transfer latency is increased to 16 cycles, the performance incurs a 2x slowdown on average. In summary, our design is not sensitive to the TSV latency when it is low enough while the performance of design will start to degrade when the TSV latency is large enough, which justifies the reason for a design based on 3D memory technology bringing the low latency of TSV transfers.

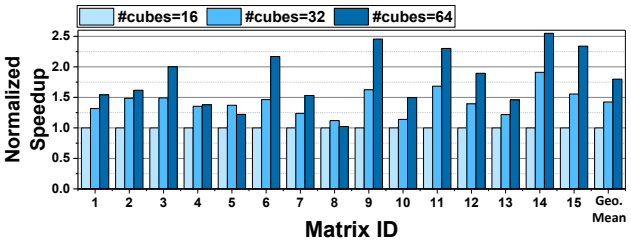

Fig. 10. The scalability of SpaceA with the increase of the number of cubes.

TABLE III
THE SPEEDUP COMPARISON AMONG TESSERACT, GRAPHP, AND SPACEA FOR PAGERANK (PR) AND SINGLE-SOURCE SHORTEST PATH (SSSP) ALGORITHMS ON WIKI (WK) AND LIVEJOURNAL (LJ) DATASETS OVER CPU BASELINE.

	Tesseract	GraphP	SpaceA
PR + WK	18.19	22.58	29.73
SSSP + WK	43.70	52.17	103.57
PR + LJ	21.09	34.08	58.34
SSSP + LJ	40.10	42.83	51.47

E. Scalability

We show the scalability of SpaceA by increasing the number of cubes in Figure 10. Figure 10 shows that SpaceA with 32 cubes achieves 1.42x speedup and SpaceA with 64 cubes achieves 1.8x speedup on average compared to the default configuration. These results reveal moderate scalability where overheads come from a more expensive inter-vault communication with an increase of cube amount. Although the scalability of SpaceA is moderate, the memory capacity of baseline design (64 GB) is able to accommodate most of the matrices from the University of Florida collection (max size about 50 GB) [19]. When the number of cubes increases, the latency of memory access to other cubes becomes larger, which makes the size of the current PE queue not large enough to hide the latency of remote memory access. Using a larger PE queue and L1 load queue to exploit larger memory-level parallelism (MLP) will introduce a larger area overhead in the bank group level.

F. Case Study: Graph Analytics

Since SpMV is a building primitive in many application domains, such as scientific computing and graph analytics, SpaceA can be used to accelerate these applications. In order to study the performance benefits of SpaceA for these applications, we conduct a case study of running graph workloads on SpaceA, and compare the performance with state-of-the-art graph accelerators, Tesseract [4] and GraphP [76]. For comparing both Tesseract and GraphP, we use algorithms and input graphs evaluated in both of them. As a result, we use PageRank (PR) and Single-Source Shortest Path (SSSP) algorithms and Wiki (WK) and LiveJournal (LJ) input graphs [36] in this case study. Then, we run the implementation of these two algorithms from the GAP benchmark [11] on NVIDIA DGX-1 server (Intel Xeon CPU E5-2698 x2) as the baseline. To obtain the performance on SpaceA, we rewrite SSSP and PR algorithms into iterations of SpMV [33], and run them on SpaceA under the same number of cubes, vaults, and memory banks as the Tesseract configuration. We assume Tesseract and GraphP can obtain the same speedup as claimed in their paper, and the speedup of Tesseract, GraphP, and SpaceA over CPU is summarized as Table III. This assumption overestimates the performance of Tesseract and GraphP because our CPU baseline is more performant than theirs. Specifically, the CPU in our baseline has more cores (40 vs. 32), the same L1 and L2 cache per core while larger L3 cache in total (100 MB vs. 32 MB), and higher memory bandwidth (153.6 GB/s vs. 102.4 GB/s). Moreover, we use a well-optimized GAP benchmark as the CPU baseline instead of in-house C++ implementations used in Tesseract [4]. The results in Table III show that SpaceA obtains better performance than Tesseract and GraphP despite the overestimation of their speedups. The performance improvement of SpaceA mainly comes from the higher bandwidth provided by the near-bank integration instead of placing compute-logic on the base die. In summary, SpaceA can significantly accelerate graph analytics and it has the potentials to benefit other workloads built on SpMV computation.

VI. RELATED WORK

SpMV workloads. The study on efficient SpMV implementation starts from the CPU platform where the exploration of the locality of SpMV computations to efficiently use the memory bandwidth plays a major role [31], [44], [49], [66]. GPU provides massive memory-level parallelism and high memory bandwidth, which makes it a promising solution when it comes to accelerating SpMV workloads [13]. Although existing studies develop efficient implementations for the widely used compressed sparse row (CSR) format on GPUs [25], [45], new matrix compression formats, such as AMB [47], BRO [60], Cocktail [59], BCSC [69], and BCCOO [71], are proposed to address the challenges of irregular memory access and workload imbalance across different processing units in more efficient and scalable manners. The road-map for SpMV on other many-core architectures, such as Intel Xeon Phi and Intel Knight Landing, is similar to GPGPU where customized matrix compression formats [43], [61], [68] are designed together with the parallel algorithms SpMV to partition workloads across cores. Although these studies exploit existing memory bandwidth very well, they can not overcome the problem of limited memory bandwidth. Unlike these prior works optimizing SpMV on multi-core (CPU) or many-core processors (GPU), we exploit PIM-based architecture for superior bandwidth to overcome the bandwidth problem in multi-core and many-core processors.

PIM and NDP accelerators. There are several studies for PIM and NDP architectures in recent years for general purpose programs on different memory technologies, such as non-volatile memory (NVM) [22], [40] and DRAM [5], [39], [56], [73], [75]. These architectures are usually equipped with compute logic designed for basic arithmetic primitives to support general purpose programs. Meanwhile, PIM ar-

chitectures are also very promising in accelerator designs, which are customized for specific application domains, such as neural networks, block-chain [67], and image processing [26] workloads. In particular, many neural network workloads are memory intensive [70], thus prior studies exploit PIM architectures for different application phases: both training [8], [16], [27], [41], [53], [57] and inference phases [6], [17], [24], [32], [35], [55], [63]. Among these application domains prior work studied for exploiting PIM architectures, graph analytics is the closest to SpMV workloads. In the vertex-centric programming model, a graph algorithm is equivalent to multiple iterations of SpMV when edges are stored in an adjacency matrix. Prior work studied graph workloads in PIM architectures for various memory technologies [4], [18], [48], [58] and efficient graph data partition methods [76]. Our case study in Section V-F shows that SpaceA achieves higher performance than prior designs placing compute-logic on the base die because of higher effective bandwidth exploited at the memory bank level. Prior studies have also exploited similar sparse linear algebra primitives, such as sparse matrix-matrix multiplication (SpGEMM) [79]. However, SpGEMM is very different from SpMV because of its poor data reuse opportunity. Other research discussing in-memory computing for the scientific workloads [21] has also been conducted. However, these studies do not use compact sparse formats leading to both storage and performance overheads. Overall, different from all of these PIM accelerators, SpaceA is the first to design lightweight compute-logic near DRAM banks for irregular workloads whose memory access pattern is highly irregular thus introducing challenges for increasing the utilization of bank-level memory bandwidth.

Sparse linear algebra primitive accelerators. There are prior studies designing accelerators for SpMV [52] or other sparse linear algebra primitives [7], [30]. In particular, because of the model compression techniques for neural network applications, such as weight pruning [29] and weight quantization [64], a lot of dense linear algebra primitives are transformed to sparse ones. As a result, these sparse neural network training and inference workloads attract intensive attention to accelerator designs for sparse linear algebra primitives [42], [50], [77], [78]. Although these studies optimize SpMV for better locality or workload balance for on-chip computation, these compute-centric hardware designs have limited memory bandwidth. SpaceA exploits a PIM-based architecture superior bandwidth to overcome the bandwidth problem in CPU, GPU, and compute-centric accelerators.

VII. Discussion

System and programming interface: Since SpaceA is designed as a standalone accelerator attached to the PCIe bus, it copies the sparse matrix and input vector from the CPU, offloads the computation of SpMV, and finally copies the output vector back to the CPU. The software support of SpaceA needs to provide APIs for memory allocation, data transfer, and SpMV computation invocation so that CPU programs can offload SpMV computation to SpaceA. Because the data format is different between sparse matrices and vectors, these two data structures need different driver APIs to support data allocation and transfer. Additionally, the sparse matrix needs to be pre-processed on the CPU for assigning different rows across PEs before it is transferred to SpaceA. This execution model has been proven practical by prior studies offloading SpMV into GPU [47], [59], [60], [71].

HMC vs. HBM: Although our architecture design of SpaceA is demonstrated and evaluated based on HMC-like configuration, SpaceA can also be realized by HBM [2] achieving similar performance and power under an equivalent configuration. The effectiveness of SpaceA architecture design mainly relies on two perspectives, near-bank logic integration and low latency communications for banks within the same channel. Although memory banks are grouped into the same channel horizontally in HBM while vertically in HMC, both of these two architectures have low latency TSV for communications among banks in the same channel. Therefore, the proposed approach would be applicable to HBM with a similar conclusion on performance and energy improvement.

VIII. Conclusion

In this paper, we design an accelerator, SpaceA, based on PIM architecture by integrating compute-logic at the memory bank level to provide orders of magnitude higher effective bandwidth than GPU for SpMV computation. To exploit such a high bandwidth, our PE design is composed of a queue that holds memory requests to hide the latency of memory access to data in other memory banks. To exploit locality and to reduce traffic among memory banks, we integrate CAM buffers in SpaceA to cache data from the input vector. In addition to the architecture design, we develop a mapping scheme for SpaceA to balance workload and exploit locality among PEs. Our evaluation of 15 real-world matrices shows that SpaceA is highly competitive in terms of performance and energy-efficiency compared to the state-of-the-art GPU baseline.

References

[1] "HMC Specification 2.1," http://hybridmemorycube.org/, 2014.

[2] "JEDEC Standard. High Bandwidth Memory (HBM) DRAM. JESD25A," https://www.jedec.org/standards-documents/docs/jesd235a, 2015.

[3] "NVIDIA cuSPARSE library," https://docs.nvidia.com/cuda/cusparse/index.html, 2018.

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 105–117, 2016.

[5] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 2015, pp. 336–348.

[6] S. Angizi, Z. He, A. S. Rakin, and D. Fan, "Cmp-pim: an energy-efficient comparator-based processing-in-memory neural network accelerator," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 105.

[7] B. Asgari, R. Hadidi, T. Krishna, H. Kim, and S. Yalamanchili, "Alrescha: A lightweight reconfigurable sparse-computation accelerator," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 249–260.

[8] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes," *IEEE Transactions on Parallel & Distributed Systems*, no. 1, pp. 1–1, 2018.

[9] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2019, pp. 373–386.

[10] A. Basak, J. Lin, R. Lorica, X. Xie, Z. Chishti, A. Alameldeen, and Y. Xie, "Saga-bench: Software and hardware characterization of streaming graph analytics workloads," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 12–23.

[11] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[12] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 56–65.

[13] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM, 2009, p. 18.

[14] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 2013, pp. 1–12.

[15] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2012, pp. 33–38.

[16] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang, "Time: A training-in-memory architecture for memristor-based deep neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 26.

[17] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 27–39.

[18] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "Graphh: A processing-in-memory architecture for large-scale graph processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[19] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[20] Y. Eckert, N. Jayasena, and G. H. Loh, "Thermal feasibility of die-stacked processing in memory," 2014.

[21] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek, "Enabling scientific computing on memristive accelerators," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 367–382.

[22] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 1–14.

[23] S. Galal, O. Shacham, J. S. Brunhaver II, J. Pu, A. Vassiliev, and M. Horowitz, "Fpu generator for design space exploration," in *2013 IEEE 21st Symposium on Computer Arithmetic*. IEEE, 2013, pp. 25–34.

[24] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 751–764, 2017.

[25] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 769–780.

[26] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie, "ipim: Programmable in-memory image processing accelerator using near-bank architecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 804–817.

[27] P. Gu, X. Xie, S. Li, D. Niu, H. Zheng, K. T. Malladi, and Y. Xie, "Dlux: a lut-based near-bank accelerator for data center deep learning training workloads," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[28] R. Hadidi, B. Asgari, B. A. Mudassar, S. Mukhopadhyay, S. Yalamanchili, and H. Kim, "Demystifying the characteristics of 3d-stacked memories: A case study for hybrid memory cube," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2017, pp. 66–75.

[29] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[30] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[31] E.-J. Im and K. A. Yelick, *Optimizing the performance of sparse matrix-vector multiplication*. University of California, Berkeley, 2000.

[32] Y. Ji, Y. Zhang, X. Xie, S. Li, P. Wang, X. Hu, Y. Zhang, and Y. Xie, "Fpsa: A full system stack solution for reconfigurable reram-based nn accelerator architecture," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 733–747.

[33] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[34] M. J. Khurshid and M. Lipasti, "Data compression for thermal mitigation in the hybrid memory cube," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013, pp. 185–192.

[35] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 380–392.

[36] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[37] J. Li, G. Tan, M. Chen, and N. Sun, "Smat: an input adaptive auto-tuner for sparse matrix-vector multiplication," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 117–126.

[38] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for spmv on gpu using probabilistic modeling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 196–205, 2015.

[39] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 288–301.

[40] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 173.

[41] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2018, pp. 656–669.

[42] L. Liu, Z. Qu, L. Deng, F. Tu, S. Li, X. Hu, Z. Gu, Y. Ding, and Y. Xie, "Duet: Boosting deep neural network efficiency on dual-module architecture," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 738–750.

[43] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 273–282.

[44] J. Mellor-Crummey and J. Garvin, "Optimizing sparse matrix–vector product computations using unroll and jam," *The International Journal of High Performance Computing Applications*, vol. 18, no. 2, pp. 225–236, 2004.

[45] D. Merrill and M. Garland, "Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format," in *ACM SIGPLAN Notices*, vol. 51, no. 8. ACM, 2016, p. 43.

[46] D. Milojevic, S. Idgunji, D. Jevdjic, E. Ozer, P. Lotfi-Kamran, A. Panteli, A. Prodromou, C. Nicopoulos, D. Hardy, B. Falsari *et al.*, "Thermal characterization of cloud workloads on a power-efficient server-on-chip," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*. IEEE, 2012, pp. 175–182.

[47] Y. Nagasaka, A. Nukada, and S. Matsuoka, "Adaptive multi-level blocking optimization for sparse matrix vector multiplication on gpu," *Procedia Computer Science*, vol. 80, pp. 131–142, 2016.

[48] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frame-

works," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 457–468.

[49] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. ACM, 1999, p. 30.

[50] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 58–70.

[51] Y. Saad, *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2003, vol. 82.

[52] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 347–358.

[53] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini, "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," *arXiv preprint arXiv:1803.04783*, 2018.

[54] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 99–108.

[55] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[56] H. Shin, D. Kim, E. Park, S. Park, Y. Park, and S. Yoo, "Mcdram: Low latency and energy-efficient matrix computations in dram," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2613–2622, 2018.

[57] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 541–552.

[58] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 531–543.

[59] B.-Y. Su and K. Keutzer, "clspmv: A cross-platform opencl spmv framework on gpus," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 353–364.

[60] W. T. Tang, W. J. Tan, R. Ray, Y. W. Wong, W. Chen, S.-h. Kuo, R. S. M. Goh, S. J. Turner, and W.-F. Wong, "Accelerating sparse matrix-vector multiplication on gpus using bit-representation-optimized schemes," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 26.

[61] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huynh, X. Li, and R. S. M. Goh, "Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2015, pp. 136–145.

[62] UPMEM, "The true Processing-In-Memory accelerator," 2020. [Online]. Available: https://www.hotchips.org/hc31/HC31_1.4_UPMEM.FabriceDevaux.v2_-1.pdf

[63] P. Wang, Y. Ji, C. Hong, Y. Lyu, D. Wang, and Y. Xie, "Snrram: an efficient sparse neural network computation architecture based on resistive random-access memory," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 106.

[64] P. Wang, X. Xie, L. Deng, G. Li, D. Wang, and Y. Xie, "Hitnet: Hybrid ternary recurrent neural network," in *Advances in Neural Information Processing Systems*, 2018, pp. 604–614.

[65] C. Weis, N. Wehn, L. Igor, and L. Benini, "Design space exploration for 3d-stacked drams," in *2011 Design, Automation & Test in Europe*. IEEE, 2011, pp. 1–6.

[66] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*. IEEE, 2007, pp. 1–12.

[67] K. Wu, G. Dai, X. Hu, S. Li, X. Xie, Y. Wang, and Y. Xie, "Memory-bound proof-of-work acceleration for blockchain applications," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[68] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Cvr: efficient vectorization of spmv on x86 processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 2018, pp. 149–162.

[69] X. Xie, D. Du, Q. Li, Y. Liang, W. T. Tang, Z. L. Ong, M. Lu, H. P. Huynh, and R. S. M. Goh, "Exploiting sparsity to accelerate fully connected layers of cnn-based applications on mobile socs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 2, pp. 1–25, 2017.

[70] X. Xie, X. Hu, P. Gu, S. Li, Y. Ji, and Y. Xie, "Nnbench-x: Benchmarking and understanding neural network workloads for accelerator designs," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 38–42, 2019.

[71] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaspmv: yet another spmv framework on gpus," in *Acm Sigplan Notices*, vol. 49, no. 8. ACM, 2014, pp. 107–118.

[72] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus: implications for graph mining," *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 231–242, 2011.

[73] A. Yazdanbakhsh, C. Song, J. Sacks, P. Lotfi-Kamran, H. Esmaeilzadeh, and N. S. Kim, "In-dram near-data approximate acceleration for gpus," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–14.

[74] A. Yoo, A. H. Baker, R. Pearce *et al.*, "A scalable eigensolver for large scale-free graphs using 2d graph partitioning," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 63.

[75] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "Top-pim: throughput-oriented programmable processing in memory," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014, pp. 85–98.

[76] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 544–557.

[77] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 261–274.

[78] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 359–371.

[79] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013, pp. 1–6.