SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Systems

CHRISTINA GIANNOULA, ETH Zürich, Switzerland and National Technical University of Athens, Greece
IVAN FERNANDEZ, ETH Zürich, Switzerland and University of Malaga, Spain
JUAN GÓMEZ-LUNA, ETH Zürich, Switzerland
NECTARIOS KOZIRIS, National Technical University of Athens, Greece
GEORGIOS GOUMAS, National Technical University of Athens, Greece
ONUR MUTLU, ETH Zürich, Switzerland

Several manufacturers have already started to commercialize near-bank Processing-In-Memory (PIM) architectures, after decades of research efforts. Near-bank PIM architectures place simple cores close to DRAM banks. Recent research demonstrates that they can yield significant performance and energy improvements in parallel applications by alleviating data access costs. Real PIM systems can provide high levels of parallelism, large aggregate memory bandwidth and low memory access latency, thereby being a good fit to accelerate the Sparse Matrix Vector Multiplication (SpMV) kernel. SpMV has been characterized as one of the most significant and thoroughly studied scientific computation kernels. It is primarily a memory-bound kernel with intensive memory accesses due its algorithmic nature, the compressed matrix format used, and the sparsity patterns of the input matrices given.

This paper provides the first comprehensive analysis of SpMV on a real-world PIM architecture, and presents *SparseP*, the first SpMV library for real PIM architectures. We make three key contributions. First, we implement a wide variety of software strategies on SpMV for a multithreaded PIM core, including (1) various compressed matrix formats, (2) load balancing schemes across parallel threads and (3) synchronization approaches, and characterize the computational limits of a single multithreaded PIM core. Second, we design various load balancing schemes across multiple PIM cores, and two types of data partitioning techniques to execute SpMV on thousands of PIM cores: (1) 1D-partitioned kernels to perform the complete SpMV computation only using PIM cores and (2) 2D-partitioned to strive a balance between computation and data transfer costs to PIM-enabled memory. Third, we compare SpMV execution on a real-world PIM system with 2528 PIM cores to an Intel Xeon CPU and an NVIDIA Tesla V100 GPU to study the performance and energy efficiency of various devices, i.e., both memory-centric PIM systems and conventional processor-centric CPU/GPU systems for the SpMV kernel. *SparseP* software package provides 25 SpMV kernels for real PIM systems supporting the four most widely used compressed matrix formats, i.e., CSR, COO, BCSR and BCOO, and a wide range of data types. *SparseP* is publicly and freely available at [49]. Our extensive evaluation using 26 matrices with various sparsity patterns provides new insights and recommendations for software designers and hardware architects to efficiently accelerate the SpMV kernel on real PIM systems.

Key Words: high-performance computing, HPC, sparse matrix-vector multiplication, SpMV, SpMV library, multicore, processing-in-memory, near-data processing, memory systems, data movement bottleneck, DRAM, benchmarking, real-system characterization, workload characterization

1 Introduction

Sparse Matrix-Vector Multiplication (SpMV) is a fundamental linear algebra kernel for important applications from the scientific computing, machine learning, and graph analytics domains. In commodity systems, it has been repeatedly reported to achieve only a small fraction of the peak performance [33, 35, 47, 64, 70, 93, 146–148, 154] due to its algorithmic nature, the employed compressed matrix storage format, and the sparsity pattern of the input matrix. SpMV

performs indirect memory references as a result of storing the matrix in a compressed format, and irregular memory accesses to the input vector due to sparsity. The matrices involved are very sparse, i.e., the vast majority of elements are zeros [33, 35, 46, 57, 69, 87, 136, 149]. For example, the matrices that represent Facebook's and YouTube's network connectivity contain 0.0003% [69, 87] and 2.31% [69, 136] non-zero elements, respectively. Therefore, in processor-centric systems, SpMV is a memory-bandwidth-bound kernel for the majority of real sparse matrices, and is bottlenecked by data movement between memory and processors [31, 33–35, 45–47, 53, 64, 70, 78, 79, 93, 121, 146–148, 153].

One promising way to alleviate the data movement bottleneck is the Processing-In-Memory (PIM) paradigm [1, 4, 30, 38, 41–45, 53, 72, 81, 82, 86, 107–109, 119, 135]. PIM moves computation close to application data by equipping memory chips with processing capabilities [107, 109]. Prior works [1, 14, 15, 23, 28, 32, 38, 41, 42, 44, 61, 62, 74, 85, 98, 99, 111, 165] propose PIM architectures wherein a processor logic layer is tightly integrated with DRAM memory layers using 2.5D/3D-stacking technologies [54, 68]. Nonetheless, the 2.5D/3D integration itself might not always be able to provide significantly higher memory bandwidth for processors than standard DRAM [4, 86]. To provide even higher bandwidth for the in-memory processors, *near-bank* PIM designs have been explored [4, 21, 22, 30, 45, 50, 53, 80, 82, 86, 110, 122, 144]. *Near-bank* PIM designs tightly couple a PIM core with each DRAM bank, exploiting bank-level parallelism to expose high on-chip memory bandwidth of standard DRAM to processors. Moreover, manufacturers of near-bank PIM architectures avoid disturbing the key components (i.e., subarray and bank) of commodity DRAM to provide a cost-efficient and practical way for silicon materialization. Two *real* near-bank PIM architectures are Samsung's FIMDRAM [82, 86] and UPMEM PIM [30, 45, 53, 143].

Most near-bank PIM architectures [4, 21, 22, 30, 45, 50, 53, 80, 82, 86, 110, 122, 144] support several PIM-enabled memory chips connected to a host CPU via memory channels. Each memory chip comprises multiple PIM cores, which are low-area and low-power cores with relatively low computation capabilities [45, 53], and each of them is located close to a DRAM bank [4, 21, 22, 30, 45, 50, 53, 80, 82, 86, 110, 122, 144]. Each PIM core can access data located on their local DRAM banks, and typically there is no direct communication channel among PIM cores. Overall, near-bank PIM architectures provide high levels of parallelism and very large memory bandwidth, thereby being a very promising computing platform to accelerate memory-bound kernels. Recent works leverage near-bank PIM architectures to provide high performance and energy benefits on bioinformatics [45, 53, 84] and neural network kernels [21, 45, 50, 53, 86]. However, there is no prior work to thoroughly study the widely-used, memory-bound SpMV kernel on a real PIM system.

Our work is the first to efficiently map the SpMV execution kernel on near-bank PIM systems, and understand its performance implications on a real PIM system. Specifically, our **goal** in this work is twofold: (i) design efficient SpMV algorithms to accelerate this kernel in current and future PIM systems, while covering a wide variety of sparse matrices with diverse sparsity patterns, and (ii) provide an extensive characterization analysis of the widely used SpMV kernel on a real PIM architecture. To this end, we provide a wide variety of SpMV implementations for real PIM architectures, and conduct a rigorous experimental analysis of SpMV kernels in the UPMEM PIM system, the first publicly-available real-world PIM architecture.

We present the *SparseP* library [49] that includes 25 SpMV kernels for real PIM systems, supporting various (1) data types, (2) data partitioning techniques of the sparse matrix to PIM-enabled memory, (3) compressed matrix formats, (4) load balancing schemes across PIM cores,

(5) load balancing schemes across threads of a multithreaded PIM core, and (6) synchronization approaches among threads within PIM core. We support a wide range of data types, i.e., 8-bit integer, 16-bit integer, 32-bit integer, 64-bit integer, 32-bit float and 64-bit float data types to cover a wide variety of real-world applications that employ SpMV as their underlying kernel. We design two types of well-crafted data partitioning techniques: (i) the 1D partitioning technique to perform the complete SpMV computation only using PIM cores, and (ii) the 2D partitioning technique to strive a balance between computation and data transfer costs to PIM-enabled memory. In the 1D partitioning technique, the matrix is horizontally partitioned across PIM cores, and the *whole* input vector is copied into the DRAM bank of *each* PIM core, while PIM cores directly compute the elements of the final output vector. In the 2D partitioning technique, the matrix is split in 2D tiles, the number of which is equal to the number of PIM cores, and a *subset* of the elements of the input vector is copied to the DRAM bank of each PIM core. However, in the 2D partitioning technique, PIM cores create a large number of partial results for the elements of the output vector which are gathered and merged by the host CPU cores to assemble the final output vector. We support the most popular compressed matrix formats, i.e., CSR [11, 125], COO [125, 131], BCSR [63], BCOO [125], and for each compressed format we implement various load balancing schemes across PIM cores to provide efficient SpMV execution for a wide variety of sparse matrices with diverse sparsity patterns. Finally, we design several load balancing schemes and synchronization approaches among parallel threads within a PIM core to cover a variety of real PIM systems that provide multithreaded PIM cores.

We conduct an extensive characterization analysis of *SparseP* kernels on the UPMEM PIM system [30, 53, 53, 144] analyzing the SpMV execution using (1) one single multithreaded PIM core, (2) thousands of PIM cores, and (3) comparing it with that achieved on conventional processor-centric CPU and GPU systems. First, we characterize the limits of a single multithreaded PIM core, and show that (i) high operation imbalance across threads of a PIM core can impose high overhead in the core pipeline, and (ii) fine-grained synchronization approaches to increase parallelism cannot outperform a coarse-grained approach, if PIM hardware serializes accesses to the local DRAM memory bank. Second, we analyze the end-to-end SpMV execution of 1D and 2D partitioning techniques using thousands of PIM cores. Our study indicates that the performance (i) of the 1D partitioning technique is limited by data transfer costs to *broadcast* the whole input vector into *each* DRAM bank of PIM cores, and (ii) of the 2D partitioning technique is limited by data transfer costs to *gather* partial results for the elements of the output vector from PIM-enabled memory to the host CPU. Such data transfers incur high overheads, because they take place via the narrow memory bus. In addition, our detailed study across a wide variety of compressed matrix formats and sparse matrices with diverse sparsity patterns demonstrates that (i) the compressed matrix format determines the data partitioning strategy across DRAM banks of PIM-enabled memory, thereby affecting the computation balance across PIM cores with the corresponding performance implications, and (ii) there is *no one-size-fits-all* solution. The load balancing scheme across PIM cores (and across threads within a PIM core) and data partitioning technique that provides the best-performing SpMV execution depends on the characteristics of the input matrix and the underlying PIM hardware. Finally, we compare the SpMV execution on a state-of-the-art UPMEM PIM system with 2528 PIM cores to state-of-the-art CPU and GPU systems, and observe that SpMV on UPMEM PIM achieves a much higher fraction of machine's peak performance compared to that of CPU and GPU systems. Our extensive evaluation provides programming recommendations for software designers, and suggestions and hints for hardware and system designers of future PIM systems.

Our most significant recommendations for PIM software designers are:

(1) Design algorithms that provide high load balance across threads of PIM core in terms of computations, loop control iterations, synchronization points and memory accesses.
(2) Design compressed data structures that can be effectively partitioned across DRAM banks, with the goal of providing high computation balance across PIM cores.
(3) Design *adaptive* algorithms that trade off computation balance across PIM cores for lower data transfer costs to PIM-enabled memory, and adapt their configuration to the particular patterns of each input given, as well as the characteristics of the PIM hardware.

Our most significant suggestions for PIM hardware and system designers are:

(1) Provide low-cost synchronization support and hardware support to enable concurrent memory accesses by multiple threads to the local DRAM memory bank to increase parallelism in a multithreaded PIM core.
(2) Optimize the broadcast collective operation in data transfers from main memory to PIM-enabled memory to minimize overheads of copying the input data into all DRAM banks in the system.
(3) Optimize the gather collective operation *at DRAM bank granularity* for data transfers from PIM-enabled memory to host CPU to minimize overheads of retrieving the output results.
(4) Design high-speed communication channels and optimized libraries for data transfers to/from thousands of DRAM banks of PIM-enabled memory.

Our *SparseP* software package is freely and publicly available [49] to enable further research on SpMV in current and future PIM systems. The main contributions of this work are as follows:

- We present *SparseP*, the first open-source SpMV software package for real PIM architectures. *SparseP* includes 25 SpMV kernels, supporting the four most widely used compressed matrix formats and a wide range of data types. *SparseP* is publicly available at [49], and can be useful for researchers to improve multiple aspects of future PIM hardware and software.
- We perform the first comprehensive study of the widely used SpMV on the UPMEM PIM architecture, the first real commercial PIM architecture. We analyze performance implications of SpMV PIM execution using a wide variety of (1) compressed matrix formats, (2) data types, (3) data partitioning and load balancing techniques, and (4) 26 sparse matrices with diverse sparsity patterns.
- We compare the performance and energy of SpMV on the state-of-the-art UPMEM PIM system with 2528 PIM cores to state-of-the-art CPU and GPU systems. SpMV achieves less than 1% of the peak performance on processor-centric CPU and GPU systems, while it achieves on average 51.7% of the peak performance on the UPMEM PIM system, thus better leveraging the computation capabilities of underlying hardware. The UPMEM PIM system also provides high-energy efficiency on the SpMV kernel.

2 Background and Motivation

2.1 Sparse Matrix Vector Multiplication (SpMV)

The SpMV kernel multiples a sparse matrix of size $M \times N$ with a dense input vector of size $1 \times N$ to compute an output vector of size $M \times 1$. The SpMV kernel is widely used in a variety of applications including graph processing [9, 16, 69], convolutional neural networks [92, 163], machine learning [52, 91, 113, 160], and high performance computing [12, 22, 31, 36, 37, 58]. These applications involve matrices with very high sparsity [33, 35, 46, 57, 69, 87, 136, 149], i.e., a large fraction of zero elements. Thus, using a compression scheme is a straightforward

approach to avoid unnecessarily storing zero elements and performing computations on them. For general sparse matrices, the most widely-used storage format is the Compressed Sparse Row (CSR) format [11, 125]. Figure 1 presents an example of a compressed matrix using the CSR format (left), and the CSR-based SpMV execution (right), assuming an input vector x and an output vector y.

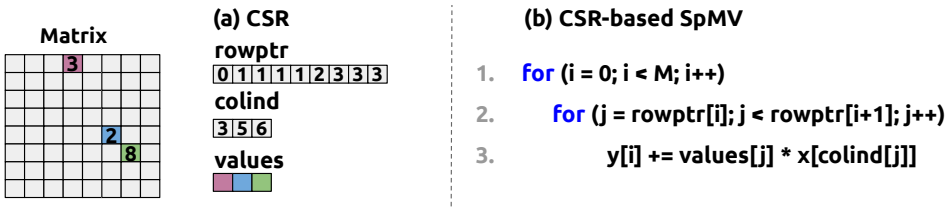


Fig. 1. (a) CSR-based representation of a sparse matrix. (b) CSR-based SpMV implementation.

2.1.1 *Compressed Matrix Storage Formats*

Several prior works [5, 11, 17, 59, 63, 76, 78, 78, 79, 83, 95, 97, 101, 102, 105, 124, 125, 127, 129, 131, 148, 154, 157] propose compressed storage formats for sparse matrices, which are typically of two types [69]. The first approach is to design general purpose compressed formats, such as CSR [11, 125], CSR5 [95], COO [125, 131], BCSR [63], and BCOO [125]. Such encodings are general in applicability and are highly-efficient in storage. The second approach is to leverage a certain known structure in a given type of sparse matrix. For example, the DIA format [5] is effective in matrices where the non-zero elements are concentrated along the diagonals of the matrix. Such encodings aim to improve performance of sparse matrix computations by specializing to particular matrix patterns, but they sacrifice generality. In this work, we explore with the four most widely used, *general* compressed formats (Figure 2), which we describe in more detail next.

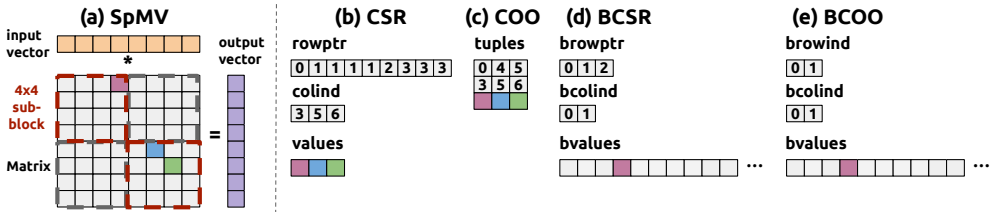


Fig. 2. (a) SpMV with a dense matrix representation, and (b) CSR, (c) COO, (d) BCSR, (e) BCOO formats.

Compressed Sparse Row (CSR) [11, 125]. The CSR format (Figure 2b) sequentially stores values in row order. A column index array (`colind[]`) and a value array (`values[]`) store the column index and value of each non-zero element, respectively. An array, named `rowptr[]`, stores the location of the first non-zero element of each row within the `values[]` array. The adjacent pair `rowptr[i]`, `rowptr[i+1]` represents a slice of the `colind[]` and `values[]` arrays. The corresponding slice of the `colind[]` and `values[]` arrays stores the column indices and the values of the non-zero elements, respectively, for the i-th row.

Coordinate Format (COO) [125, 131]. The COO format (Figure 2c) stores the non-zero elements as a series of tuples (`tuples[]` array). Each tuple includes the row index, column index, and value of the non-zero element.

Block Compressed Sparse Row (BCSR) [63]. The BCSR format (Figure 2d) is a block representation of CSR. Instead of storing and indexing single non-zero elements, BCSR stores and indexes $b \times b$ sub-blocks with at least one non-zero element. The original matrix is split in $b \times b$ sub-blocks. Figure 2d shows an example of BCSR assuming 4×4 sub-blocks. The original matrix of Figure 2a is split in four sub-blocks, and two of them (highlighted with red color) contain at least one non-zero element. The `bvalues[]` array stores the values of all the non-zero sub-blocks of the original matrix. Each non-zero sub-block is stored in the `bvalues[]` array with a dense representation, i.e., padding with zero values when needed. The `bcolind[]` array stores the block-column index of each non-zero sub-block. The `browptr[]` array stores pointers to the first non-zero sub-block of each block row within the `bcolind[]` array, assuming a block row represents b consecutive rows of the original matrix.
Block Coordinate Format (BCOO) [125]. The BCOO format is the block counterpart of COO. The `browind[]`, `bcolind[]` and `bvalues[]` arrays store the row indices, column indices and values of the non-zero sub-blocks, respectively. Figure 2e shows an example of BCOO assuming 4×4 sub-blocks.

2.1.2 *SpMV in Processor-Centric Systems*

Many prior works [33, 35, 47, 64, 70, 93, 146–149, 154] generally show that SpMV performs poorly on commodity CPU and GPU systems, and achieves a small fraction of the peak performance (e.g., 10% of the peak performance [147]) due to its algorithmic nature, the employed compressed matrix storage format and the sparsity pattern of the matrix.

The SpMV kernel is highly bottlenecked by the memory subsystem in processor-centric CPU and GPU systems due to three reasons. First, due to its algorithmic nature there is *no* temporal locality in the input matrix. Unlike traditional algebra kernels like Matrix Matrix Multiplication or LU decomposition, the elements of the matrix in SpMV are used only *once* [46, 47]. Second, due to the sparsity of the matrix, the matrix is stored in a compressed format (e.g., CSR) to avoid unnecessary computations and data accesses. Specifically, the non-zero elements of the matrix are stored contiguously in memory, while additional data structures assist in the proper traversal of the matrix, i.e., to discover the positions of the non-zero elements. For example, CSR uses the `rowptr[]` and `colind[]` arrays to discover the positions of the non-zero elements of the matrix. These additional data structures cause additional memory access operations, memory bandwidth pressure and contention with other requests in the memory subsystem. Third, due to the sparsity of the input matrix, SpMV causes irregular memory accesses to the elements of the input vector x. The memory accesses to the elements of the input vector are input driven, i.e., they follow the sparsity pattern of the input matrix. This irregularity results to poor data locality on the elements of the input vector and expensive data accesses, because it increases the average access latency due to a high number of cache misses on commodity systems with deep cache hierarchies [46, 47]. As a result, memory-centric near-bank PIM systems constitute a better fit for the widely-used SpMV kernel, because they provide high levels of parallelism, large aggregate memory bandwidth and low memory access latency [45, 53].

2.2 Near-bank PIM Systems

Figure 3 shows the baseline organization of a near-bank PIM system that we assume in this work. The PIM system (Figure 3) consists of a host CPU, standard DRAM memory modules, and PIM-enabled memory modules. PIM-enabled modules are connected to the host CPU using one or more memory channels, and include multiple PIM chips. A PIM chip (Figure 3 right)

tightly integrates a low-area PIM core with a DRAM memory bank. We assume that each PIM core can additionally include a small private instruction memory and a small data (scratchpad or cache) memory. PIM cores can access data located on their local DRAM bank, and typically there is no direct communication channel among PIM cores. The DRAM banks of PIM chips are accessible by the host CPU for copying input data and retrieving results via the memory bus.

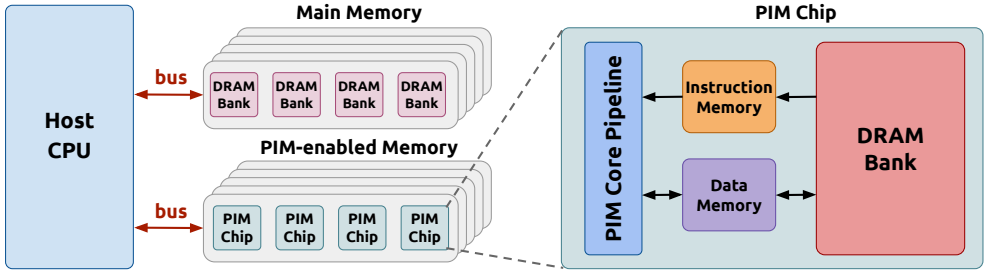


Fig. 3. High-level organization of a near-bank PIM architecture.

2.2.1 *UPMEM PIM Architecture*

The UPMEM PIM system [30, 53, 53] includes the host CPU with standard main memory, and UPMEM modules. An UPMEM module is a standard DDR4-2400 DIMM [67] with 2 ranks. Each rank contains 64 PIM cores, which are called DRAM Processing Units (DPUs). In the current UPMEM PIM system, there are 20 double-rank PIM DIMMs with 2560 DPUs.[1]

DPU Architecture and Interface. Each DPU has exclusive access to a 24-KB instruction memory, called **IRAM**, a 64-KB scratchpad memory, called **WRAM**, and a 64-MB DRAM bank, called **MRAM**. A DPU is a multithreaded in-order 32-bit RISC core that can potentially reach 500MHz [144]. The DPU has 24 hardware threads, each of which has 24 32-bit general purpose registers. The DPU pipeline has 14 stages, and supports a single cycle 8x8-bit multiplier. Multiplications on 64-bit integers, 32-bit floats and 64-bit floats are not supported in hardware, and require longer routines with a large number of operations [45, 53, 144]. Threads share the IRAM and WRAM, and can access the MRAM by executing transactions at 64-bit granularity via a DMA engine, i.e., data can be accessed from/to MRAM as a multiple of 8 bytes, up to 2048 bytes. MRAM transactions are serialized in the DMA engine. The ISA provides DMA instructions to move instructions from MRAM to IRAM, or data between MRAM and WRAM. The DPU accesses the WRAM through 8-, 16-, 32- and 64-bit load/store instructions. DPUs use the *Single Program Multiple Data* programming model, where software threads, called **tasklets**, execute the same code, but operate in different pieces of data, and can execute different control-flow paths during runtime. Tasklets can synchronize using mutexes, barriers, handshakes and semaphores provided by UPMEM runtime library.

CPU-DPU Data Transfers. Standard main memory and PIM-enabled memory have different data layouts. The UPMEM SDK [145] has a transposition library to execute necessary data shuffling when moving data between main memory and MRAM banks via a programmer-transparent way. The CPU-DPU and DPU-CPU data transfers can be performed in parallel, i.e., concurrently across multiple MRAM banks, having the limitation that *the transfer sizes from/to all MRAM banks need to be the same*. The UPMEM SDK provides two options: (i) perform

[1]There are thirty two faulty DPUs in the system where we run our experiments. They cannot be used and do not affect the correctness of our results, but take away from the system's full computational power of 2560 DPUs.

parallel transfers to all MRAM banks of all ranks, or (ii) iterate over each rank to perform parallel transfers to MRAM banks of the same rank, and serialize data transfers across ranks.

3 *SparseP* Library

This section describes the parallelization techniques that we explore for SpMV on real PIM architectures, and presents the SpMV implementations of our *SparseP* package. Section 3.1 describes SpMV execution on a real PIM system. Section 3.2 presents an overview of the data partitioning techniques that we explore. Section 3.3 and Section 3.4 describe in detail the parallelization techniques across PIM cores, and across threads within a PIM core, respectively. Section 3.5 describes the kernel implementation for all compressed formats.

3.1 SpMV Execution on a PIM System

Figure 4 shows the SpMV execution on a real PIM system, which is broken down in four steps: (1) the time to load the input vector into DRAM banks of PIM-enabled memory (**`load`**), (2) the time to execute the SpMV kernel on PIM cores (**`kernel`**), (3) the time to retrieve from DRAM banks to the host CPU results for the output vector (**`retrieve`**), and (4) the time to merge partial results and assemble the final output vector on the host CPU (**`merge`**). In our analysis, we omit the time to load the matrix into PIM-enabled memory, since this step can typically be hidden in real-world applications (it can be overlapped with other computation performed by the application or amortized if the application performs multiple SpMV iterations on the same matrix).

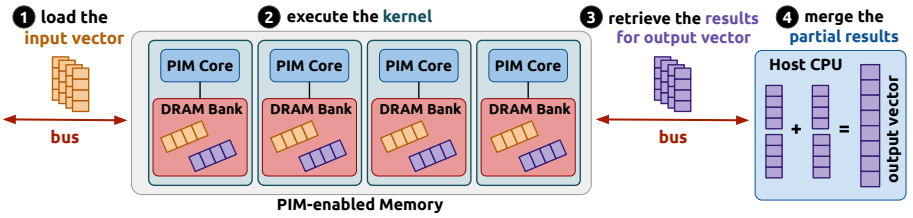


Fig. 4. Execution of the SpMV kernel on a real PIM system.

3.2 Overview of Data Partitioning Techniques

To parallelize the SpMV kernel, we implement well-crafted data partitioning schemes to split the matrix across multiple DRAM banks of PIM cores. *SparseP* supports two general types of data partitioning techniques, shown in Figure 5.

First, we provide a 1D partitioning technique (Figure 5a), where the matrix is horizontally partitioned across PIM cores, and the whole input vector is copied into the DRAM bank of each core. With the 1D partitioning technique, almost the entire SpMV computation is performed using only PIM cores, since the `merge` step in the host CPU is negligible: a very small number of partial results are created, i.e., only for a few rows that are split across neighboring PIM cores. Thus, the number of partial elements of the output vector is at most equal to the number of PIM cores used. Second, we provide a 2D partitioning technique (Figure 5b), where the matrix is partitioned into 2D tiles, the number of which is equal to the number of PIM cores. With the 2D partitioning technique, we aim to strive a balance between computation and data transfer costs, since only a subset of the elements of the input vector is copied into the DRAM bank of each PIM core. However, PIM cores assigned to tiles that horizontally overlap, i.e., tiles that

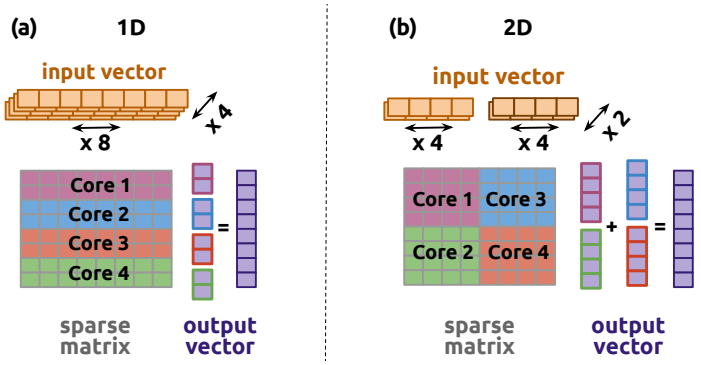


Fig. 5. Data partitioning techniques of the *SparseP* package.

share the same rows of the original matrix (rows that are split across multiple tiles), produce *many* partial results for the elements of the output vector. These partial results are transferred to the host CPU, and merged by CPU cores, which assemble the final output vector. In the *SparseP* library, the `merge` step performed by the CPU cores is parallelized using the OpenMP API [27].

In both data partitioning schemes, matrices are stored in a row-sorted way, i.e., the non-zero elements are sorted in increasing order of their row indices. Therefore, each PIM core computes results for a *continuous* subset of elements of the output vector. This way we minimize data transfer costs, since we only transfer necessary data to the host CPU, i.e., *the values* of the elements of the output vector produced at PIM cores. If PIM cores computed results for *non-continuous* elements of the output vector, an additional array *per core*, that would store *the indices* of the *non-continuous* elements within the output vector, would need to be transferred to the host CPU, causing additional data transfer overheads. Finally, for both data partitioning techniques, *SparseP* supports various parallelization schemes across PIM cores and across threads within a multithreaded PIM core, which we describe next.

3.3 Parallelization Techniques Across PIM Cores

3.3.1 ***1D Partitioning Technique*** To efficiently parallelize SpMV across multiple PIM cores via the 1D partitioning technique, *SparseP* provides various load balancing schemes for each supported compressed format. Figure 6 presents an example of parallelizing SpMV across multiple PIM cores using load balancing schemes for CSR and COO formats. For CSR and COO formats, we balance either the rows, such that each PIM core processes almost the same number of rows, or the non-zero elements, such that each PIM core processes almost the same number of non-zero elements. In CSR format, since the matrix is stored in row-order, i.e., the `rowptr[]` array stores the index pointers of the non-zero elements of *each row*, balancing the non-zero elements across PIM cores is performed at row granularity. In COO format, the matrix is stored in non-zero order using the `tuples[]` array, thus balancing the non-zero elements can be performed either at row granularity, or by splitting a row across two neighboring PIM cores to provide a near-perfect non-zero element balance across cores. In the latter case, as mentioned, a small number of partial results for the output vector is merged by the host CPU: if the row is split between two neighboring PIM cores at most one element needs to be accumulated at the host CPU cores.

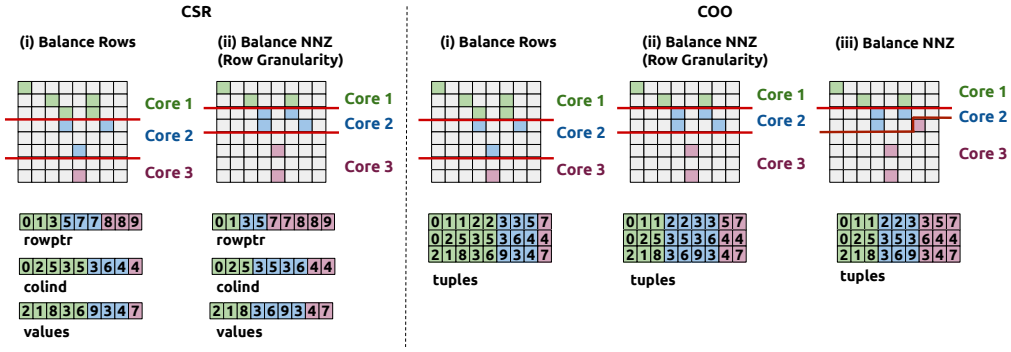


Fig. 6. Load balancing schemes across PIM cores for CSR (left) and COO (right) formats with the 1D partitioning technique. The colored cells of the matrix represent non-zero elements.

Figure 7 presents an example of parallelizing SpMV across multiple PIM cores using load balancing schemes of BCSR and BCOO formats. In Figure 7, the cells of the matrix represent sub-blocks of size 4x4: the *grey* cells represent sub-blocks that do not have *any* non-zero element, and the *colored* cells represent sub-blocks that have k non-zero elements, where k is the number shown inside the colored cell. In BCSR and BCOO formats, since the matrix is stored in sub-blocks of non-zero elements, we balance either the blocks, such that each PIM core processes almost the same number of blocks, or the non-zero elements, such that each PIM core processes almost the same number of non-zero elements. Similarly to CSR, in BCSR format, the matrix is stored in block-row order, i.e., the `browptr[]` array stores the index pointers of the non-zero blocks of *each block row* (recall that a block row represents b consecutive rows of the original matrix, where b is the vertical dimension of the sub-block), and thus balancing the blocks or the non-zero elements across cores is limited to be performed at block-row granularity. In BCOO format, given that a block-row might be split across two PIM cores, a small number of partial results for the output vector is merged by the host CPU: between two neighboring PIM cores at most block size b elements might need to be accumulated at the host CPU cores.

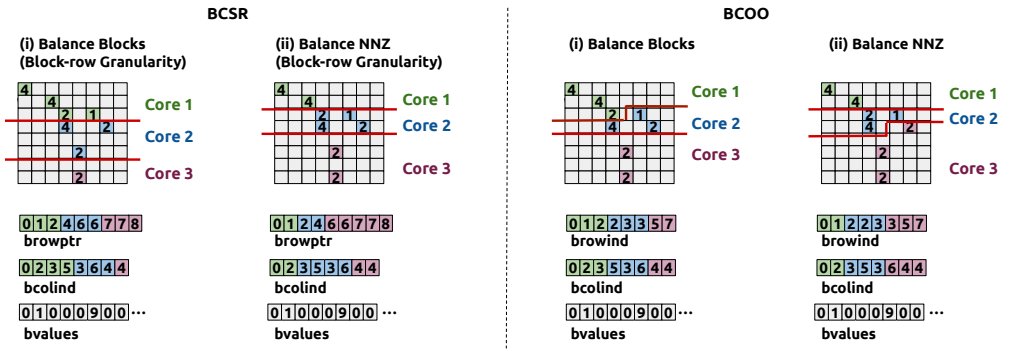


Fig. 7. Load balancing schemes across PIM cores for BCSR (left) and BCOO (left) formats with the 1D partitioning technique. The cells of the matrix represent sub-blocks of size 4x4. The colored cells of the matrix represent non-zero sub-blocks, and the number inside a colored cell describes the number of non-zero elements of the corresponding sub-block.

3.3.2 ***2D Partitioning Technique*** *SparseP* includes three 2D partitioning techniques, shown in Figure 8:

(1) ***equally-sized*** (Figure 8a): The 2D tiles are statically created to have the same height and width. This way the subsets of the elements for the input and output vectors have the same sizes across all PIM cores.

(2) ***equally-wide*** (Figure 8b): The 2D tiles have the same width and variable height. This way the subset of the elements for the input vector has the same size across PIM cores, while the subset of the elements for the output vector varies across PIM cores. We balance the non-zero elements across the tiles of the *same* vertical partition, such that we can provide high non-zero balance across PIM cores assigned to the same vertical partition.

(3) ***variable-sized*** (Figure 8c): The 2D tiles have both variable width and height. We balance the non-zero elements both across the vertical partitions and across the tiles of the *same* vertical partition. This way we can provide high-non zero balance across all PIM cores.

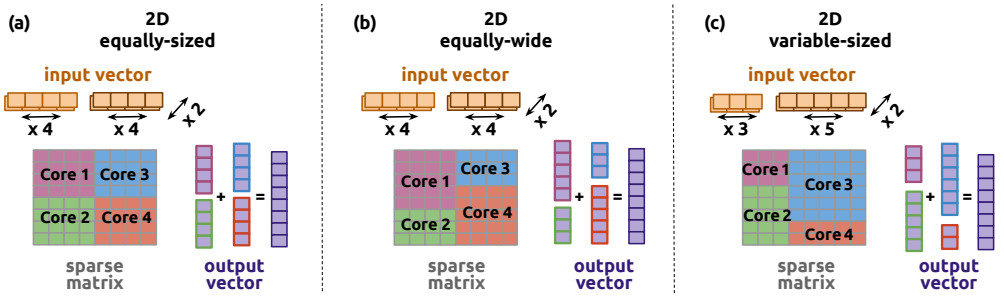


Fig. 8. The 2D partitioning techniques of *SparseP* package assuming 4 PIM cores and 2 vertical partitions.

SparseP provides various load balancing schemes across PIM cores in the *equally-wide* and *variable-sized* techniques. In the *equally-wide* technique, for CSR and COO formats, we balance the non-zero elements across the tiles of the same vertical partition. Load balancing in the CSR format is performed at row-granularity, i.e., splitting the `rowptr[]` array across PIM cores. For BCSR and BCOO formats, we balance either the blocks or the non-zero elements across the tiles of the same vertical partition. Load balancing in the BCSR format is performed at block-row granularity, i.e., splitting the `browptr[]` array across PIM cores. In the *variable-sized* technique, we first balance the non-zero elements across the vertical partitions, such that the vertical partitions include the same number of non-zero elements. Then, across the tiles of the same vertical partition, we balance the non-zero elements for CSR (at row-granularity) and COO formats, and either the blocks or the non-zero elements for BCSR (at block-row granularity) and BCOO formats.

Table 1 summarizes the parallelization approaches across PIM cores. Please also see Appendix C for all SpMV kernels provided by the *SparseP* software package. All kernels support a wide range of data types, i.e., 8-bit integer (**int8**), 16-bit integer (**int16**), 32-bit integer (**int32**), 64-bit integer (**int64**), 32-bit float (**fp32**), and 64-bit float (**fp64**) data types.

3.4 Parallelization Techniques Across Threads within a PIM Core

PIM cores can support multiple hardware threads to exploit high memory bank bandwidth [45, 53]. To parallelize SpMV across multiple threads within a multithreaded PIM core *SparseP*

Partitioning Technique	Compressed Format	Load Balancing Across PIM Cores
1D	CSR	rows (**CSR.row**) nnz* (**CSR.nnz**)
	COO	rows (**COO.row**) nnz* (**COO.nnz-rgrn**) nnz (**COO.nnz**)
	BCSR	blocks† (**BCSR.block**) nnz† (**BCSR.nnz**)
	BCOO	blocks (**BCOO.block**) nnz (**BCOO.nnz**)
2D *equally-sized*	CSR (**DCSR**)	-
	COO (**DCOO**)	-
	BCSR (**DBCSR**)	-
	BCOO (**DBCOO**)	-
2D *equally-wide*	CSR (**RBDCSR**)	nnz*
	COO (**RBDCOO**)	nnz
	BCSR	blocks† (**RBDBCSR**) nnz†
	BCOO	blocks (**RBDBCOO**) nnz
2D *variable-sized*	CSR (**BDCSR**)	nnz*
	COO (**BDCOO**)	nnz
	BCSR	blocks† (**BDBCSR**) nnz†
	BCOO	blocks (**BDBCOO**) nnz

Table 1. Parallelization techniques across PIM cores of *SparseP*. *: row-granularity, †: block-row-granularity

supports various load balancing schemes for each compressed format, and three synchronization approaches to ensure correctness among threads of a PIM core.

3.4.1 ***Load Balancing Approaches.*** In a similar way as explained in Figure 6, for CSR and COO formats, we balance either the rows, such that each thread processes almost the same number of rows, or the non-zero elements, such that each thread processes almost the same number of non-zero elements. In CSR format, matrix is stored in row-order, thus load balancing across threads is performed at row granularity. In the UPMEM PIM system, elements of the output vector are accessed at 64-bit granularity in DRAM memory. Thus, when balancing is performed at row granularity, we assign rows to threads in chunks of $8/sizeof(data\_type)$ to ensure 8-byte alignment on the elements of the output vector. In COO format, balancing the non-zero elements can be performed either at row-granularity or by splitting the row between threads, i.e., providing an almost perfect non-zero balance across threads. In the latter case, synchronization among threads for write accesses on the elements of the output vector can be implemented with three synchronization approaches described in the Section 3.4.2.

For BCSR and BCOO formats, we balance either the blocks, such that each thread processes almost the same number of blocks, or the non-zero elements, such that each thread processes

almost the same number of non-zero elements. In BCSR format, the matrix is stored in block-row order, thus load balancing across threads is performed at block-row granularity. For both formats, the block sizes are *configurable* in *SparseP*. In our evaluation, we use block sizes of 4x4, since these are the most common dimensions to cover various sparse matrices [2, 35, 70]. In the UPMEM PIM architecture, elements of the output vector are accessed at 64-bit granularity. Therefore, for BCSR format, with an 8-bit integer data type and small block sizes (4x4 or smaller), threads use synchronization primitives to ensure correctness when writing the elements of the output vector. This is because different threads may write to the same 64-bit-aligned DRAM memory location. Synchronization among threads for writes to the elements of the output vector is necessary for all configurations of BCOO format, and can be implemented with three approaches described next.

3.4.2 ***Synchronization Approaches*** *SparseP* provides three synchronization approaches.

(1) **Coarse-grained Locking (lb-cg).** One global mutex protects the elements of the entire output vector.

(2) **Fine-grained Locking (lb-fg).** Multiple mutexes protect the elements of the output vector. *SparseP* associates mutexes to the elements of the output vector in a round-robin manner. The UPMEM API supports up to 56 mutexes [145]. In our evaluation, we use 32 mutexes such that we can find the corresponding mutex for a particular element of the output vector only with a shift operation on the MRAM address, avoiding costly division operations.

(3) **Lock-free (lf).** Since the formats are row-sorted or block-row sorted, race conditions in the elements of the output vector arise *only in a few elements*, i.e., either when a row (or a block row for BCSR/BCOO) is split across threads, or when continuous elements of the output vector processed by different threads belong to the same 64-bit aligned MRAM location in the UPMEM PIM system. In our proposed lock-free approach, threads temporarily store partial results for these few elements in the data (scratchpad) memory (i.e., WRAM in the UPMEM PIM system), and later one single thread merges the partial results, and writes the final result for the corresponding element of the output vector to the DRAM bank.

Table 2 summarizes the parallelization techniques across threads of a PIM core. All kernels support a wide range of data types, i.e., 8-bit integer (**int8**), 16-bit integer (**int16**), 32-bit integer (**int32**), 64-bit integer (**int64**), 32-bit float (**fp32**), and 64-bit float (**fp64**) data types.

Compressed Format	Load Balancing Across Threads	Synchronization Approach
CSR	rows (**CSR.row**)	-
	nnz* (**CSR.nnz**)	-
COO	rows (**COO.row**)	-
	nnz* (**COO.nnz-rgrn**)	-
	nnz (**COO.nnz**)	lb-cg / lb-fg / lf
BCSR	blocks† (**BCSR.block**)	lb-cg / lb-fg (only for int8 and small block sizes)
	nnz† (**BCSR.nnz**)	lb-cg / lb-fg (only for int8 and small block sizes)
BCOO	blocks (**BCOO.block**)	lb-cg / lb-fg / lf
	nnz (**BCOO.nnz**)	lb-cg / lb-fg / lf

Table 2. Parallelization schemes across threads of a PIM core. *: row-granularity, †: block-row-granularity

3.5 Kernel Implementation

We briefly describe the *SparseP* implementations for all compressed matrix formats, i.e., the way that threads access data involved in the kernel from/to the local DRAM bank. The SpMV kernels include three types of data structures: (i) the arrays that store the non-zero elements, i.e., the values (`values[]`) and the positions of the non-zero elements (`rowptr[]`, `colind[]` for CSR format, `tuples[]` for COO format, `browptr[]`, `bcolind[]` for BCSR format, `browind[]`, `bcolind[]` for BCOO format), (ii) the array that stores the elements of the input vector, and (iii) the array that stores the partial results created for the elements of the output vector.

First, SpMV performs streaming memory accesses to the arrays that store the non-zero elements and their positions. Therefore, to exploit spatial locality and immense bandwidth in data (scratchpad or cache) memory, each thread reads the non-zero elements by fetching large chunks of bytes in a coarse-grained manner from DRAM memory to data memory (i.e., WRAM in the UPMEM PIM system). Then, it accesses elements through data memory in a fine-grained manner. In the UPMEM PIM system, we fetch chunks of 256-byte data to discover the non-zero elements, as suggested by UPMEM API [145], since 256-byte transfer sizes highly exploit the available local bandwidth of DRAM memory bank [45, 53]. For BCSR and BCOO formats, only for the array that stores the values of the non-zero elements (i.e., `bvalues[]` array), we fetch from DRAM to data memory block size chunks, i.e., chunks of $b \times b \times sizeof(data\_type)$ bytes, assuming that the matrix is stored in blocks of size $b \times b$.

Second, SpMV causes irregular memory accesses to the elements of the input vector (poor data locality), i.e., the accesses are input-driven, since they are determined by the column positions of the non-zero elements of each particular sparse matrix. Thus, threads of a PIM core directly access elements of the input vector through DRAM bank at fine-granularity [45, 53, 145], i.e., using the smallest possible granularity: for CSR and COO formats at 64-bit granularity, and for BCSR and BCOO formats at block size granularity, i.e., $b \times sizeof(data\_type)$ bytes, where b is the horizontal dimension of the block size.

Third, regarding the output vector, threads temporarily store partial results for the same elements of the output vector in data (scratchpad or cache) memory to exploit data locality, until all the non-zero elements of the *same* row or the *same* block row have been traversed (recall matrices are stored in a row-sorted way). Then, the produced results are written to DRAM bank at fine-granularity [45, 53, 145]: for CSR and COO formats at 64-bit granularity, and for BCSR and BCOO formats at block size granularity, i.e., $b \times sizeof(data\_type)$ bytes, where b is the vertical dimension of the block size.

4 Evaluation Methodology

We conduct our evaluation on a UPMEM PIM system that includes a 2-socket Intel Xeon Silver 4110 CPU [66] at 2.10 GHz (host CPU), standard main memory (DDR4-2400) [67] of 128 GB, and 20 UPMEM PIM DIMMs with 160 GB PIM-capable memory and 2560 DPUs.[2]

First, we evaluate SpMV execution using one single DPU and multiple tasklets (Section 5). Table 3 shows our evaluated small matrices that fit in the 64MB DRAM memory of a single DPU. The evaluated matrices vary in sparsity (i.e., NNZ / (rows x columns)), standard deviation of non-zero elements among rows (NNZ-r-std) and columns (NNZ-c-std). The highlighted

[2]There are thirty two faulty DPUs in the system where we run our experiments. They cannot be used and do not affect the correctness of our results, but take away from the system's full computational power of 2560 DPUs.

matrices in Table 3 with red color exhibit block pattern [35, 79], i.e., they include *a lot* of dense sub-blocks (almost all their non-zero elements fit in dense sub-blocks).

Matrix Name	Sparsity	NNZ-r-std	NNZ-c-std
delaunay_n13	7.32e-04	1.343	1.343
wing_nodal	1.26e-03	2.861	2.861
raefsky4	3.396e-03	15.956	15.956
pkustk08	0.006542	61.537	61.537

Table 3. Small Matrix Dataset.

Second, we evaluate SpMV execution using *multiple* DPUs of the UPMEM PIM system (Section 6). We evaluate SpMV execution using both 1D (Section 6.1) and 2D (Section 6.2) partitioning techniques, and compare them (Section 6.3) using a wide variety of sparse matrices with diverse sparsity patterns. We select 22 representative sparse matrices from the Sparse Suite Collection [29], the characteristics of which are shown in Table 4. As the values of the last two metrics increase (i.e., NNZ-r-std and NNZ-c-std), the matrix becomes very irregular [112, 141], and is referred to as *scale-free* matrix. In our evaluation, we refer to as scale-free matrices, the matrices in which NNZ-r-std is larger than 25, i.e., from `wbs` to `ask` matrices of Table 4. For the remaining matrices, i.e., from `hgc` to `bns` matrices of Table 4, we refer to as *regular* matrices. Please see Appendix D for a complete description of our dataset of large sparse matrices.

Matrix Name	Sparsity	NNZ-r-std	NNZ-c-std
hugetric-00020 (**hgc**)	4.21e-07	0.031	0.031
mc2depi (**mc2**)	7.59e-06	0.076	0.076
parabolic_fem (**pfm**)	1.33e-05	0.153	0.153
roadNet-TX (**rtn**)	1.98e-06	1.037	1.037
rajat31 (**rjt**)	9.24e-07	1.106	1.106
af_shell1 (**ash**)	6.90e-05	1.275	1.275
delaunay_n19 (**del**)	1.14e-05	1.338	1.338
thermomech_dK (**tdk**)	6.81e-05	1.431	1.431
memchip (**mem**)	2.02e-06	2.062	1.173
amazon0601 (**amz**)	2.08e-05	2.79	15.29
FEM_3D_thermal2 (**fth**)	1.59e-04	4.481	4.481
web-Google (**wbg**)	6.08e-06	6.557	38.366
ldoor (**ldr**)	5.13e-05	11.951	11.951
poisson3Db (**psb**)	3.24e-04	14.712	14.712
boneS10 (**bns**)	6.63e-05	20.374	20.374
webbase-1M (**wbs**)	3.106e-06	25.345	36.890
in-2004 (**in**)	8.846e-06	37.230	144.062
pkustk14 (**pks**)	6.428e-04	46.508	46.508
com-Youtube (**cmb**)	4.639e-06	50.754	50.754
as-Skitter (**skt**)	7.71e-06	136.861	136.861
sx-stackoverflow (**sxw**)	5.352e-06	137.849	65.367
ASIC_680k (**ask**)	8.303e-06	659.807	659.807

Table 4. Large Matrix Dataset. Matrices are sorted by NNZ-r-std, i.e., based on their irregular pattern.

Third, we compare the performance and energy consumption of SpMV execution on the UPMEM PIM system to those on a state-of-the-art Intel Xeon Silver 4110 CPU [66] and a state-of-the-art NVIDIA Tesla V100 GPU [118] (Section 7).

In Section 8, we summarize our key takeaways and provide programming recommendations for software designers, and suggestions and hints for hardware and system designers of future PIM systems.

5 Analysis of SpMV Execution on One DPU

This section characterizes SpMV performance with various load balancing schemes and compressed formats using multiple tasklets in a single DPU. Section 5.1 compares load balancing schemes of each compressed matrix format, and Section 5.2 compares the scalability of various compressed matrix formats.

5.1 Load Balancing Schemes Across Tasklets of One DPU

We compare the parallelization schemes of each compressed matrix format supported by *SparseP* library (presented in Table 2) across multiple threads of a multithreaded PIM core. Figure 9 compares the load balancing schemes of each compressed matrix format using 16 tasklets in a single DPU. For BCSR and BCOO formats, we omit results for the fine-grained locking approach, since it performs similarly with the coarse-grained locking approach: as we explain in Appendix A.1, fine-grained locking does not increase parallelism over coarse-grained, since in the UPMEM PIM hardware, DRAM memory accesses of the critical section are serialized in the DMA engine of the DPU [53, 53, 145].

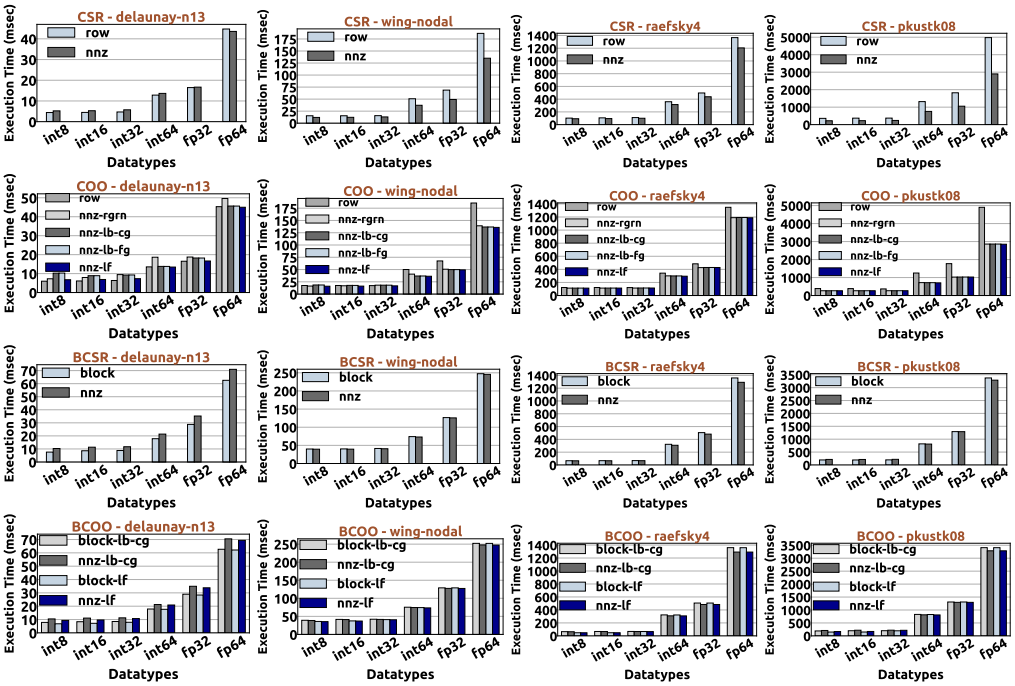


Fig. 9. Execution time achieved by 16 tasklets of a single DPU for various load balancing schemes.

We draw four findings from Figure 9. First, we find that SpMV execution using int8, int16, and int32 data types achieves similar execution times across them. This is because the multiplication operation of these data types is sufficiently supported by hardware [45]. In contrast, execution time sharply increases when using more heavyweight data types, i.e., int64 and floating point data types, in which multiplication is emulated in software using the 8-bit support in the UPMEM PIM hardware [45, 53, 145].

Second, we observe that balancing the non-zero elements across tasklets typically outperforms balancing the rows for CSR/COO formats or blocks for BCSR/BCOO formats, since the non-zero element multiplications are computationally very expensive and can significantly affect load balance across tasklets. However, in `delaunay_n13 matrix`, balancing the non-zero elements causes high row/block imbalance across tasklets, since one tasklet processes a significantly higher number of rows/blocks over the rest, thereby incurring high operation imbalance across tasklets within the DPU core pipeline. As a result, balancing the rows/blocks outperforms balancing the non-zero elements due to the particular pattern of `delaunay_n13 matrix`. In addition, performance benefits of balancing the blocks over balancing the non-zero elements are significant in BCSR/BCOO formats, because they operate at block granularity and incur high loop control costs.

Third, we observe that the lock-free approach (`COO.nnz-lf`) outperforms the lock-based approaches (`COO.nnz-lb-cg`, `COO.nnz-lb-fg`) in `delaunay_n13 matrix`, especially in data types where the multiplication operation is supported directly in hardware. In `delaunay_n13 matrix`, one tasklet processes a much higher number of rows than the rest, i.e., it performs a much higher number of critical sections than the rest. In other words, one tasklet performs a much higher number of lock acquisitions/releases and memory instructions than the rest. Thus, lock-based approaches cause high operation imbalance in the DPU core pipeline with significant performance costs. Instead, lock-free and lock-based approaches in BCOO format perform similarly, since lock acquisition/release costs can be hidden due to BCOO's higher loop control costs and larger critical sections. Overall, based on the second and the third findings, we conclude that in matrices and formats, where the load balancing and/or the synchronization scheme used cause *high* disparity in the number of non-zero elements/blocks/rows processed across tasklets or the number of lock acquisitions/lock releases/memory accesses performed among tasklets, the DPU core pipeline can incur significant performance overheads.

OBSERVATION 1:
High operation imbalance in computation, control, synchronization, or memory instructions executed by multiple threads of a PIM core can cause *high performance overheads* in the compute-bound and area-limited PIM cores.

Fourth, we find that the fine-grained locking approach (`COO.nnz-lb-fg`) performs similarly with the coarse-grained locking approach (`COO.nnz-lb-cg`). This is because the critical section includes memory accesses to the local DRAM bank, which, in the UPMEM PIM hardware, are serialized in the DMA engine of the DPU. Therefore, fine-grained locking does not increase execution parallelism over coarse-grained locking, since concurrent accesses to MRAM are not supported in the UPMEM PIM hardware. Fine-grained locking does not improve performance over coarse-grained locking, also when using block-based formats (e.g., BCSR/BCOO formats), as we demonstrate in Appendix A.1. Therefore, we recommend PIM hardware designers to provide lightweight synchronization mechanisms [44] for PIM cores, and/or enable concurrent

accesses to local DRAM memory, e.g., supporting sub-array level parallelism [55, 75, 132–134] or multiple DRAM banks per PIM core.

OBSERVATION 2:
Fine-grained locking approaches to parallelizing critical sections that perform memory accesses to different DRAM memory locations cannot improve performance over *coarse-grained* locking, when the PIM hardware does not support *concurrent accesses to DRAM memory.*

5.2 Analysis of Compressed Matrix Formats on One DPU

We compare the scalability and the performance achieved by various compressed matrix formats. Figure 10 compares the supported compressed formats for int8 (top graphs) and fp64 (bottom graphs) data types when balancing the non-zero elements across tasklets of a DPU.

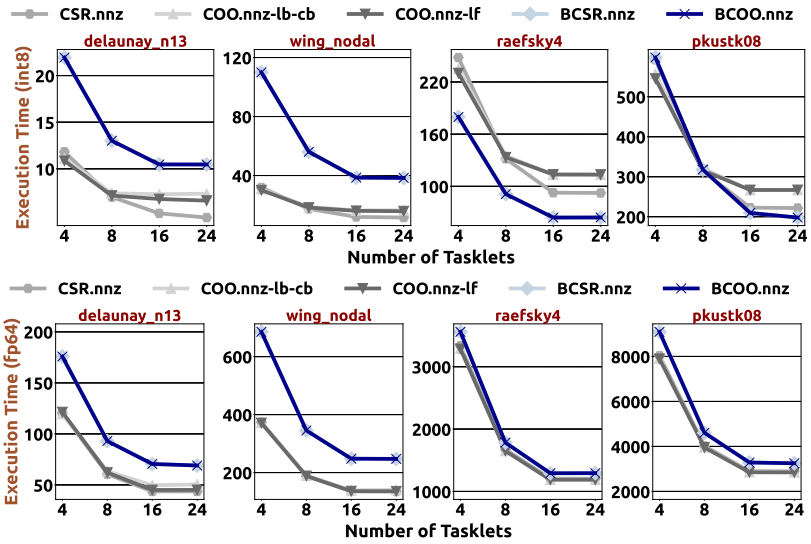


Fig. 10. Scalability of all compressed formats for int8 (top graphs) and fp64 (bottom graphs) data types as the number of tasklets of a single DPU increases.

We draw three findings. First, we find that even though a DPU supports 24 tasklets, SpMV execution typically scales up to 16 tasklets, since the DPU pipeline is fully utilized. In `delaunay_n13` matrix, `CSR.nnz` scales up to 24 tasklets. In this matrix, when using 16 tasklets, performance of `CSR.nnz` scheme is limited by memory accesses: *only* one tasklet processes 6 × more rows than the rest, i.e., it performs 6 × more memory accesses to fetch elements from the `rowptr[]` array. Thus, as we increase the number of tasklets from 16 to 24, the disparity in the number of rows across tasklets decreases, and `CSR.nnz` scheme improves performance by exploiting higher memory bandwidth. Second, we observe that for data types with hardware-supported multiplication operation (e.g., int8 data type), CSR achieves the highest scalability, since it provides a better balance between memory access and computation. In contrast, in floating point data types (e.g., fp64 data type), the DPU is significantly bottlenecked by the expensive software-emulated multiplication operations, and thus all formats scale similarly. Third, we

observe that BCSR and BCOO formats outperform CSR and COO formats in matrices that exhibit a block pattern (`raefsky4` and `pkustk08` matrices), only when multiplication is supported by hardware (e.g., int8 data type). This is because they exploit spatial and temporal locality in WRAM memory in the accesses of the elements of the input vector. Instead, in fp64 data type, performance is severely bottlenecked by computation, thus BCSR/BCOO formats perform worse than CSR/COO formats, since they incur higher indexing costs to discover the non-zero elements [2, 69].

OBSERVATION 3:
Blocked formats (e.g., BCSR/BCOO) and can provide high performance gains over non-blocked formats (e.g., CSR/COO) in matrices that exhibit a block pattern, if the multiplication operation is supported by hardware. Otherwise, the state-of-the-art CSR and COO formats can provide high performance and scalability.

6 Analysis of SpMV Execution on Multiple DPUs

This section analyzes SpMV execution using multiple DPUs in the UPMEM PIM system using the large matrix data set of Table 4.

Section 6.1 evaluates the 1D partitioning schemes. Section 6.1.1 evaluates the actual kernel time of SpMV by comparing (a) all load balancing schemes of each compressed matrix format, and (b) the performance of all compressed matrix formats. Section 6.1.2 characterizes end-to-end SpMV execution time of the 1D partitioning technique including the data transfer costs for the input and output vectors.

Section 6.2 evaluates the 2D partitioning techniques. Section 6.2.1 presents three characterization studies on (a) performing fine-grained data transfers to transfer the elements of the input and output vectors to/from PIM-enabled memory, (b) the scalability of 2D partitioning techniques to thousands of DPUs, and (c) the number of vertical partitions to perform on the matrix. Section 6.2.2 compares the end-to-end performance of all compressed matrix formats for each of the three types of 2D partitioning techniques. Section 6.2.3 compares the best-performing SpMV implementations of all three types of 2D partitioning techniques.

Section 6.3 compares the best-performing (on average across all matrices and data types) SpMV implementations of the 1D and 2D partitioning techniques.

6.1 Analysis of SpMV Execution Using 1D Partitioning Techniques

We evaluate the 1D partitioning techniques highlighted in bold in Table 1. Specifically, for `COO.nnz`, we present the coarse-grained locking (`COO.nnz-lb`) and lock-free (`COO.nnz-lf`) approaches, since the fine-grained locking approach performs similarly with the coarse-grained locking approach, as shown in the previous section (Section 5.1). Similarly, for BCSR (int8 data type) and BCOO formats, we present only the coarse-grained locking approach, since all synchronization approaches perform similarly (Section 5.1). Finally, in all experiments presented henceforth, we use 16 tasklets and load-balance the non-zero elements across tasklets within the DPU, since this load balancing scheme provides the highest performance benefits on average across all matrices and data types, according to our evaluation shown in Section 5.

6.1.1 ***Analysis of*** `Kernel` ***Time*** We compare the `kernel` time of SpMV achieved by various load balancing schemes for each particular compressed format, and then we compare the `kernel` time of the compressed formats.

Analysis of Load Balancing Schemes Across DPUs. Figure 11 compares load balancing techniques for each compressed format using 2048 DPUs and the int32 data type.

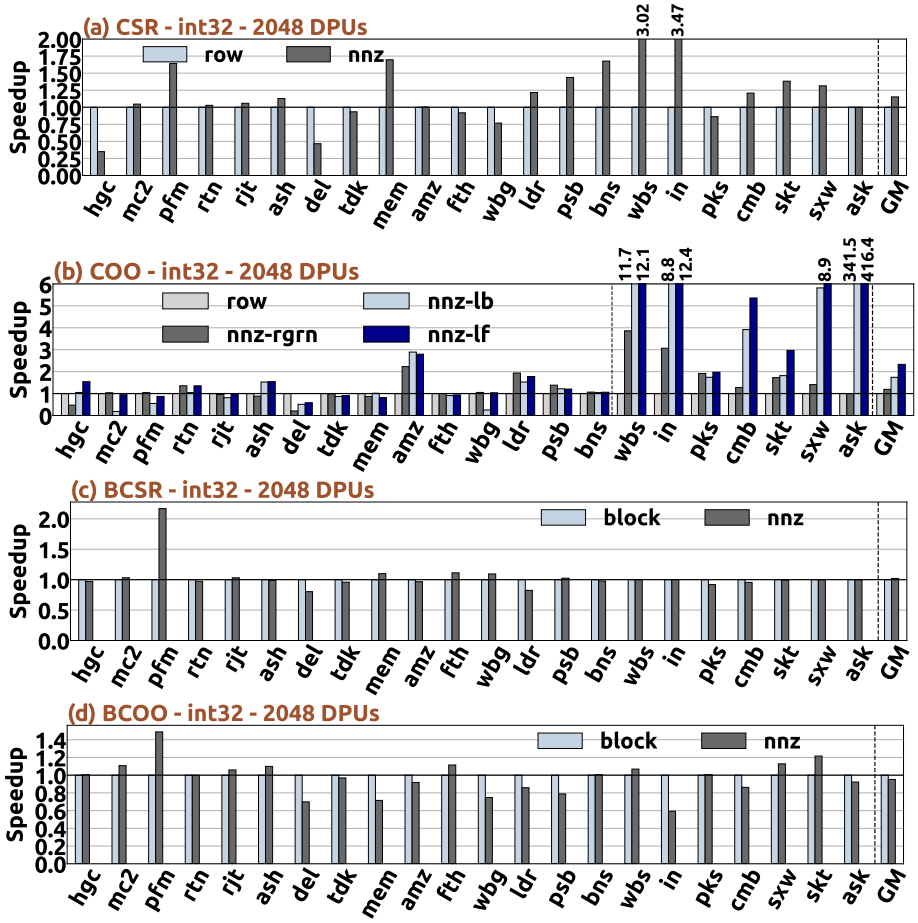


Fig. 11. Performance comparison of load balancing techniques for each particular compressed format using 2048 DPUs and the int32 data type.

We draw four findings. First, we observe that `CSR.nnz` and `COO.nnz-rgrn`, i.e., balancing the non-zero elements across DPUs (at row granularity), either outperform or perform similarly to `CSR.row` and `COO.row`, respectively, i.e., balancing the rows across DPUs, except for `hgc` and `del` matrices. In these two matrices, `CSR.nnz` and `COO.nnz-rowgrn` incur a high disparity in rows assigned to DPUs, i.e., only one DPU processes 4× and 11× more rows than the rest, for `hgc` and `del` matrices, respectively. This in turn creates a high disparity in the elements of the output vector processed across DPUs, causing performance to be limited by the DPU that processes the largest number of rows. Thus, adaptive load balancing approaches and selection

methods based on the characteristics of each input matrix need to be developed to achieve high performance across all matrices.

OBSERVATION 4:
Adaptive load balancing schemes and selection methods for the balancing scheme on rows/blocks/non-zero elements based on the characteristics of each input matrix need to be developed to provide best performance across all matrices.

Second, we find that `COO.nnz-lb` and `COO.nnz-lf`, which provide an almost perfect non-zero element balance across DPUs, significantly outperform `COO.row` and `COO.nnz-rgrn` in *scale-free* matrices (i.e., from `wbs` to `ask` matrices) by on average 6.73×. Scale-free matrices have only a few rows, that include a much larger number of non-zero elements compared to the remaining rows of the matrix. Therefore, perfectly balancing the non-zero elements across DPUs provides high performance gains.

OBSERVATION 5:
Perfectly balancing the non-zero elements across PIM cores can provide significant performance benefits in *highly irregular, scale-free matrices.*

Third, we find that the lock-free `COO.nnz-lf` scheme outperforms the lock-based `COO.nnz-lb` scheme by 1.34× on average, and it provides high performance benefits when there is a high row imbalance across tasklets within the DPU. When one tasklet processes a much higher number of rows versus the rest, it executes a much larger number of critical sections. As a result, the core pipeline incurs high imbalance in lock acquisitions/releases, causing the lock-based approach to incur high performance overheads in relatively compute-bound DPUs [45, 53].

OBSERVATION 6:
Lock-free approaches can provide high performance benefits over *lock-based* approaches in PIM architectures, because they minimize synchronization overheads in PIM cores.

Finally, in BCSR and BCOO formats, balancing the blocks performs similarly (on average across all matrices) to balancing the non-zero elements.

To further investigate the performance of the various load balancing schemes, Figure 12 compares them using all data types. We present the geometric mean of all matrices using 2048 DPUs. In CSR and COO formats, balancing the non-zero elements across DPUs on average outperforms balancing the rows across DPUs by 1.18× and 1.20×, respectively. We observe that in COO format almost perfectly balancing the non-zero elements across DPUs provides significant performance benefits (2.55×, averaged across all data types), compared to balancing the rows, especially when multiplication is not supported by hardware (e.g., for floating point data types). In contrast, in BCSR and BCOO formats, balancing the blocks across DPUs performs (averaged across all data types) only slightly better (on average 2.7%) than balancing the non-zero elements.

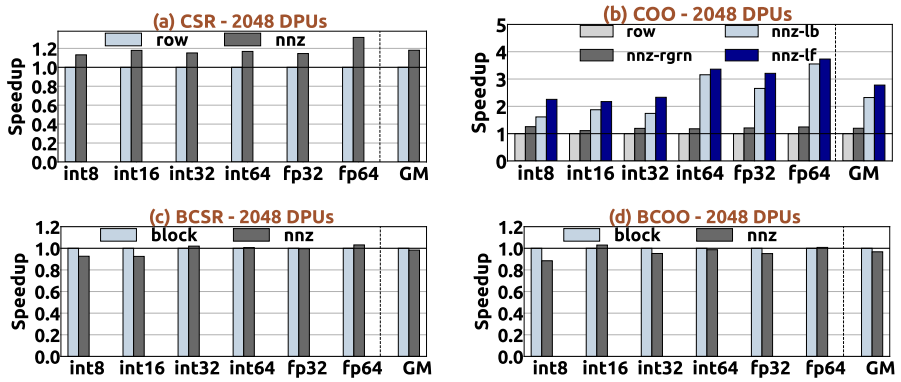


Fig. 12. Performance comparison of load balancing techniques for each data type using 2048 DPUs.

Comparison of Compressed Matrix Formats. Figure 13 compares the throughput (in GOperations per second) achieved by various compressed formats using 2048 DPUs and the int32 data type. For CSR and COO formats, we select balancing the non-zero elements, and for BCSR and BCOO formats, we select balancing the blocks, since these are the best-performing schemes for each format averaged across all matrices and data types (Figure 12).

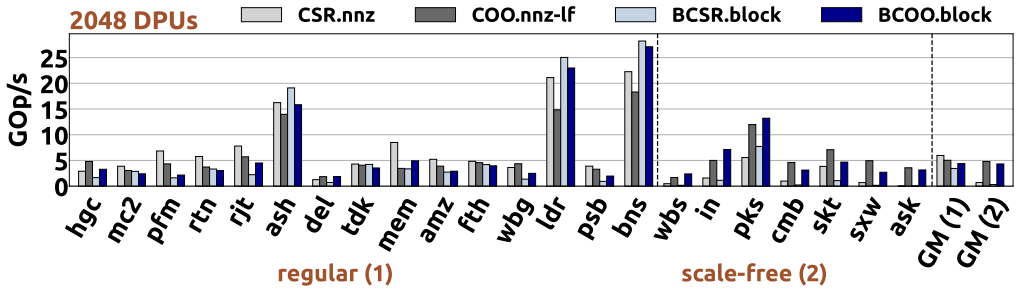


Fig. 13. Performance comparison of compressed formats using 2048 DPUs and the int32 data type.

We draw four findings. First, matrices that exhibit block pattern (almost all non-zero elements of the matrix fit in dense sub-blocks), i.e., `ash`, `ldr`, `bns`, `pks` matrices, have the highest throughput, since they leverage higher data locality compared to matrices with non-block pattern. Second, in scale-free matrices, COO and BCOO formats significantly outperform CSR and BCSR formats by 6.94× and 13.90×, respectively. This is because they provide better non-zero element balance across DPUs. In CSR and BCSR formats, the non-zero element balance is limited to be performed at row and block-row granularity, respectively, causing performance to be limited by the DPU that processes the largest number of non-zero elements. Third, we observe that BCOO format can outperform CSR even in *non-blocked* scale-free matrices. Fourth, we find that when CSR and BCSR formats provide sufficient non-zero element balance across DPUs, i.e., in many regular matrices such as `rtn`, `tdk`, `amz`, and `fth`, they can outperform COO and BCOO formats, respectively.

OBSERVATION 7:
In *scale-free* matrices, COO and BCOO formats significantly outperform CSR and BCSR formats, because they provide higher non-zero element balance across PIM cores.

6.1.2 ***Analysis of End-To-End SpMV Execution*** Figure 14 shows the end-to-end execution time of 1D-partitioned kernels using 2048 DPUs and the int32 data type. The times are broken down into (i) the time for CPU to DPU transfer to load the input vector into DRAM banks (`load`), (ii) the kernel time on DPUs (`kernel`), (iii) the time for DPU to CPU transfer to retrieve the results for the output vector (`retrieve`), and (iv) the time to merge partial results on the host CPU cores (`merge`).

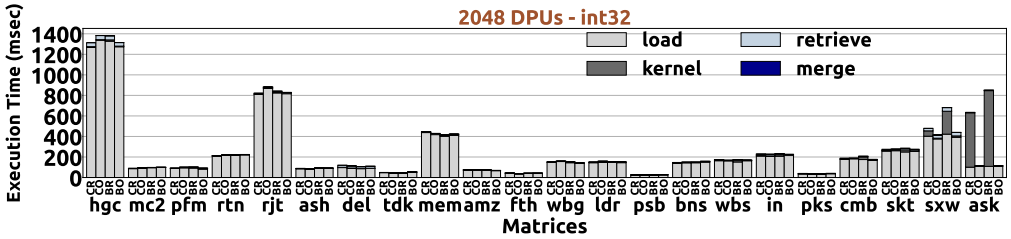


Fig. 14. Total execution time when using 2048 DPUs and the int32 data type for CR: `CSR.nnz`, CO: `COO.nnz-lf`, BR: `BCSR.block` and BO: `BCOO.block` kernels.

We draw four findings. First, the `load` data transfers constitute more than 90% of the total execution time, because the input vector is replicated and broadcast to each DPU, causing a large number of bytes to be transferred through the narrow off-chip memory bus. An exception is in CSR and BCSR formats for `sxw`, `ask` matrices, which include one very dense row, and thus `kernel` time is highly bottlenecked by one DPU that processes a significantly larger number of non-zero elements than the rest. Second, the `kernel` time constitutes on average only 4.3% of the total execution time, since SpMV is effectively parallelized to thousands of DPUs. Third, the `retrieve` data transfers constitute on average 3.4% of the total execution time, because the output vector is split across DPUs. Fourth, the `merge` time on the host CPU is negligible (less than 1% of the total execution time), since only a few partial results for the elements of the output vector are merged by the host CPU cores in the 1D partitioning techniques.

OBSERVATION 8:
The end-to-end performance of the 1D partitioning techniques is severely bottlenecked by the data transfer costs to replicate and broadcast the whole input vector to *each* DRAM bank of PIM core, which takes place through the narrow off-chip memory bus.

To further investigate on the costs to the load input vector into all DRAM banks of PIM-enabled memory, we present in Figure 15 the total execution time achieved by `COO.nnz-lf` when varying (a) the data type using 2048 DPUs (normalized to the experiment for the int8 data type), and (b) the number of DPUs for the int32 data type (normalized to 64 DPUs).

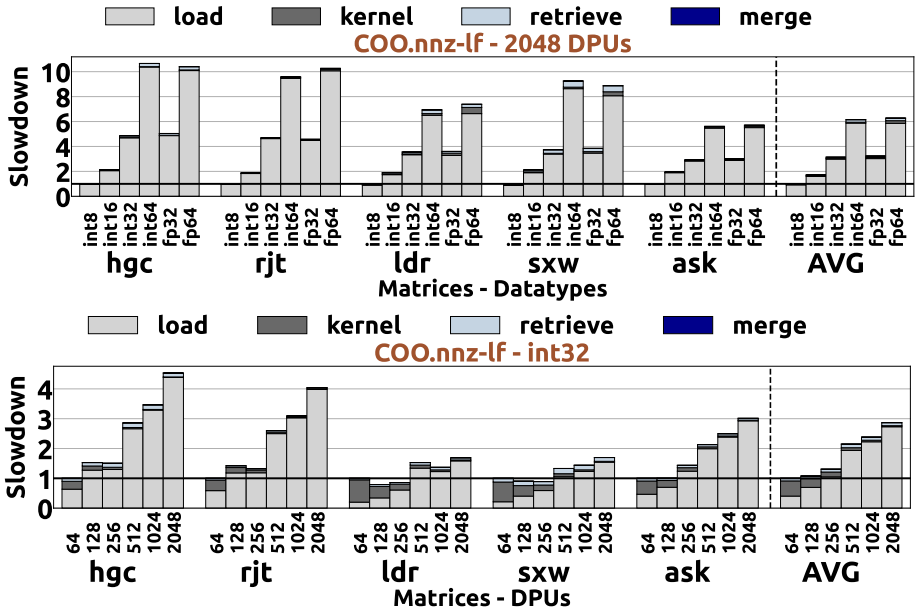


Fig. 15. End-to-end execution time breakdown achieved by `COO.nnz-lf` when varying (a) the data type using 2048 DPUs (normalized to the experiment for the int8 data type), and (b) the number of DPUs for the int32 data type (normalized to 64 DPUs).

We draw two conclusions. First, the `load` data transfer costs increase proportionally to the number of bytes of the data type, and still dominate performance even for the data type with the smallest memory footprint (int8). Second, the `load` data transfer costs and the associated memory footprint for the input vector increase proportionally to the number of DPUs used, and thus the best end-to-end performance is achieved using only a small portion of the available DPUs on the system.

OBSERVATION 9:
SpMV execution of the 1D-partitioned schemes cannot scale up to a high number of PIM cores due to high data transfer overheads to copy the input vector into *each* DRAM bank of PIM-enabled memory.

6.2 Analysis of SpMV Execution Using 2D Partitioning Techniques

We evaluate the 2D-partitioned kernels highlighted in bold in Table 1. Specifically, for COO format we use the lock-free approach, and for BCSR (in the int8 data type) and BCOO formats we use the coarse-grained locking approach. In *equally-wide* and *variable-sized* techniques, for BCSR and BCOO formats we balance the blocks across DPUs of the same vertical partition, since doing so performs slightly better than balancing the non-zero elements, as explained in Section 6.1.1. In all experiments, we balance the non-zero elements across 16 tasklets within a single DPU.

6.2.1 ***Sensitivity Studies on 2D Partitioning Techniques*** We present three characterization studies on the 2D partitioning techniques. First, we evaluate the performance of fine-grained data transfers from/to PIM-enabled memory for the input and output vectors. Second, we explore performance implications on the number of vertical partitions used in the 2D-partitioned kernels. Finally, we evaluate the scalability of the 2D partitioning techniques to thousands of DPUs.

Analysis of Fine-Grained Data Transfers. The UPMEM API [145] has the limitation that *the transfer sizes from/to all DRAM banks involved in the same parallel transfer need to be the same.* The UPMEM API provides *parallel data transfers* either to all DPUs of all ranks (henceforth referred to as *coarse-grained* transfers), or at rank granularity, i.e., to 64 DPUs of the same rank (henceforth referred to as *fine-grained* transfers). In the first case, parallel data transfers are performed to all DPUs used at once, padding with empty bytes at the granularity of *all* DPUs used, e.g., 2048 DPUs in Figure 16. In the latter case, programmers iterate over the ranks of PIM-enabled DIMMs, and for *each* rank perform parallel data transfers to the 64 DPUs of the same rank padding with empty bytes at the granularity of 64 DPUs. Therefore, in the *equally-wide* and *variable-sized* techniques, where the heights and widths of 2D tiles vary, padding with empty bytes is necessary for the `load` and `retrieve` data transfers of the elements of the input and output vector, respectively. Figure 16 compares coarse-grained data transfers, i.e., performing parallel transfers to all 2048 DPUs at once, with fine-grained data transfers, i.e., iterating over the ranks and for each rank performing parallel transfers to the 64 DPUs of the same rank. We evaluate both *equally-wide* and *variable-sized* techniques using the COO format and with 2 and 32 vertical partitions. Please see Appendix A.2 for all matrices.

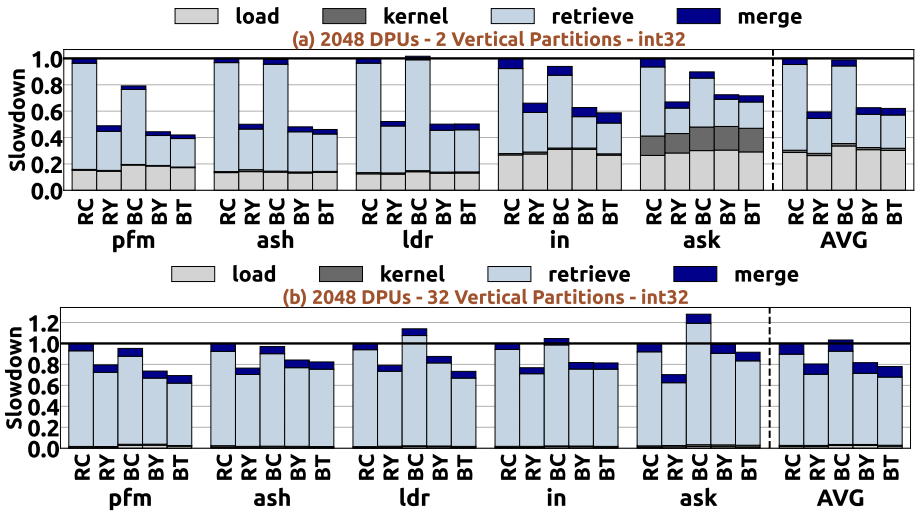


Fig. 16. Performance comparison of RC: RBDCOO with coarse-grained transfers, RY: RBDCOO with fine-grained transfers in the output vector, BC: BDCOO with coarse-grained transfers, BY: BDCOO with fine-grained transfers only in the output vector, and BT: BDCOO with fine-grained transfers in both the input and the output vector using the int32 data type, 2048 DPUs and having 2 (left) and 32 (right) vertical partitions. Performance is normalized to that of the RC scheme.

We draw two findings. First, when the number of vertical partitions is small, e.g., 2 vertical partitions, the disparity in widths across tiles in the *variable-sized* scheme is low. Thus, BT only slightly outperforms BY by 1% on average, since in BY *only* a small amount of padding

is added on the load data transfers of the input vector. In contrast, the disparity in heights across tiles in the *equally-wide* and *variable-sized* schemes is high. Thus, RY and BY significantly outperform RC and BC by an average of 1.68× and 1.60×, respectively. This is because fine-grained transfers to retrieve the elements of the output vector significantly decrease the amount of bytes transferred from PIM-enabled memory to host CPU over coarse-grained transfers. Second, when the number of vertical partitions is large, e.g., 32 vertical partitions, the disparity in heights across tiles in the *equally-wide* and *variable-sized* schemes is lower compared to when the number of vertical partitions is small. Thus, RY and BY provide smaller performance benefits over RC and BC (on average 1.24× and 1.22×, respectively), respectively, compared to a small number of vertical partitions. In contrast, the disparity in heights across tiles in the *equally-wide* and *variable-sized* schemes is higher compared to when the number of vertical partitions is small. Thus, BT outperforms BY by 4.7% on average. Overall, we conclude that fine-grained data transfers (i.e., at rank granularity in the UPMEM PIM system) can significantly improve performance in *equally-wide* and *variable-sized* schemes.

OBSERVATION 10:
Fine-grained parallel transfers in *equally-wide* and *variable-sized* 2D partitioning techniques, i.e., minimizing the amount of padding with empty bytes in parallel data transfers to/from PIM-enabled memory, can provide large performance gains.

Scalability of the 2D Partitioning Techniques. We analyze scalability with the number of DPUs for the 2D partitioning techniques. Figures 17, 18 and 19 compare the performance of *equally-sized*, *equally-wide* and *variable-sized* schemes, respectively, using the COO format and the int32 data type, as the number of DPUs increases.

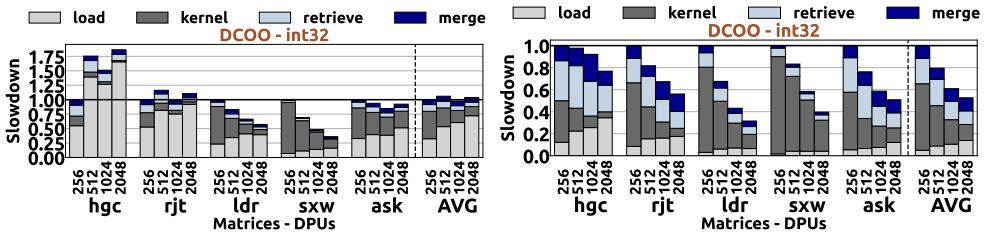


Fig. 17. Execution time breakdown of *equally-sized* partitioning techniques of the COO format using 4 (left) and 16 (right) vertical partitions when varying the number of DPUs used for the int32 data type. Performance is normalized to that with 256 DPUs.

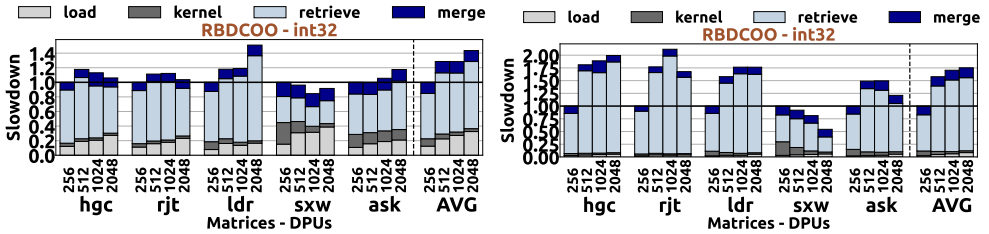


Fig. 18. Execution time breakdown of *equally-wide* partitioning techniques of the COO format using 4 (left) and 16 (right) vertical partitions when varying the number of DPUs used for the int32 data type. Performance is normalized to that with 256 DPUs.

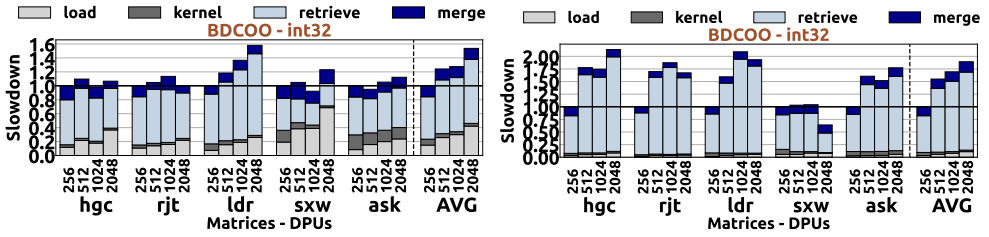


Fig. 19. Execution time breakdown of *variable-sized* partitioning techniques of the COO format using 4 (left) and 16 (right) vertical partitions when varying the number of DPUs used for the int32 data type. Performance is normalized to that with 256 DPUs.

We draw two findings. First, the *equally-sized* scheme achieves high scalability with a large number of vertical partitions. The `kernel` time of *equally-sized* scheme is mainly limited by the DPU (or a few DPUs) that processes the largest number of non-zero elements. With a large number of *static* vertical partitions, the non-zero element disparity across DPUs is high, i.e., the `kernel` time is highly bottlenecked by the DPU that processes the largest number of non-zero elements. As a result, increasing the number of DPUs improves performance by decreasing the `kernel` time via better non-zero element balance across DPUs.

OBSERVATION 11:
The `kernel` time in *equally-sized* schemes is limited by the PIM core (or a few PIM cores) assigned to the 2D tile with the largest number of non-zero elements.

Second, we observe that the *equally-wide* and *variable-sized* schemes are severely bottlenecked by `retrieve` data transfer costs (a large number of partial results is created on PIM cores), and thus they are difficult to scale up to thousands of DPUs. Moreover, when the number of vertical partitions is high, the disparity in heights of the tiles is high. Thus, as the number of DPUs increases, the amount of padding needed in `retrieve` data transfers becomes very high, causing significant performance degradation.

OBSERVATION 12:
The scalability of the *equally-wide* and *variable-sized* schemes to a high number of PIM cores is severely limited by large data transfer overheads to retrieve partial results for the elements of the output vector from the DRAM banks to the host CPU via the narrow memory bus.

Effect of the Number of Vertical Partitions. In all experiments presented henceforth, we perform fine-grained data transfers (at rank granularity, i.e., 64 DPUs in the UPMEM PIM system) in the 2D partitioning schemes. Figure 20 evaluates performance implications of the number of vertical partitions performed in 2D-partitioned kernels. We use the COO format and vary the number of vertical partitions from 1 to 32, in steps of multiple of 2. We draw four findings.

First, in the *equally-sized* scheme, as the number of vertical partitions increases, `kernel` time increases, if there is *no* dense row in the matrix. This is because the disparity in the number of non-zero elements across tiles increases as the number of vertical partitions increases. Thus, performance is limited by one DPU or a few DPUs that process the largest number of non-zero elements.

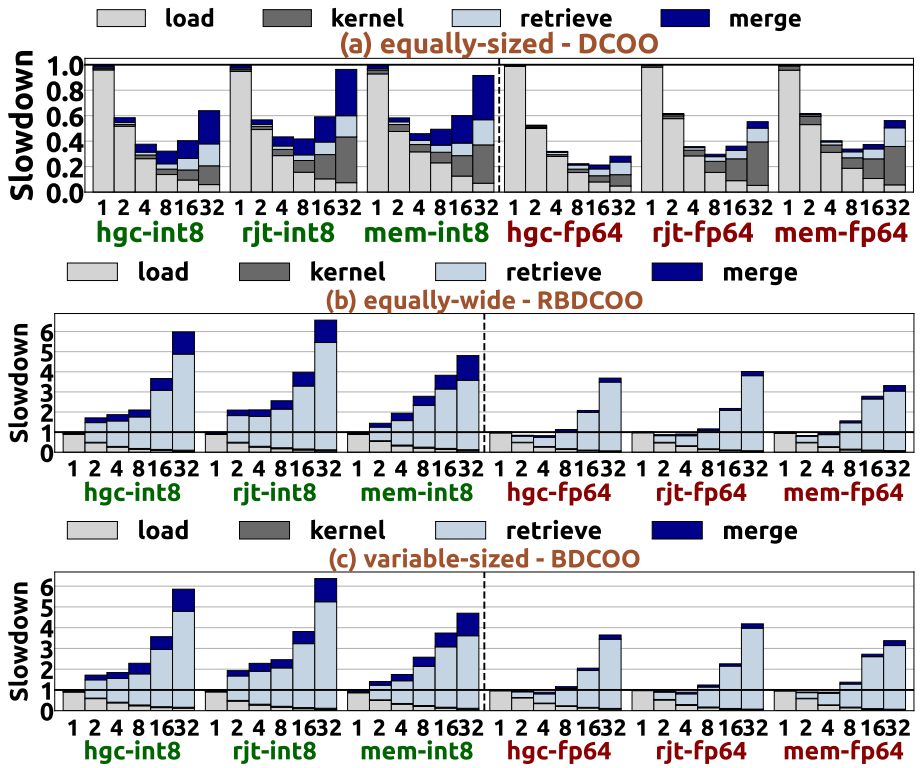


Fig. 20. Execution time breakdown of 2D partitioning schemes using the COO format and 2048 DPUs when varying the number of vertical partitions from 1 to 32 for the int8 and fp64 data types. Performance is normalized to the performance of the experiment with 1 vertical partition.

OBSERVATION 13:

As the number of vertical partitions increases, the *equally-sized* 2D scheme typically increases the non-zero element disparity across PIM cores (unless there is one dense row on the matrix), thereby increasing the `kernel` time.

Second, as the number of vertical partitions increases, `retrieve` data transfer costs and `merge` time increase. This is because the partial results created for the output vector increase proportionally with the number of vertical partitions. The performance overheads of `retrieve` data transfer costs are highly affected by the characteristics of the underlying hardware (e.g., the bandwidth provided on I/O channels of the memory bus between host CPU and PIM-enabled DIMMs). Similarly, the performance cost of the `merge` step depends on the hardware characteristics of the host CPU (e.g., number of CPU cores, the available hardware threads, microarchitecture of CPU cores).

Third, we find that in the *equally-wide* and *variable-sized* schemes, there is high disparity in heights of 2D tiles, and as a result on the number of partial results created across DPUs. Even with fine-grained parallel `retrieve` data transfers at rank granularity, the amount of padding needed in the *equally-wide* and *variable-sized* schemes is at 88.6% and 88.0%, respectively, causing high bottlenecks in the narrow memory bus. Therefore, in PIM systems that do not

support very fine-grained parallel transfers to gather results from PIM-enabled DIMMs *at DRAM bank granularity*, execution is highly limited by the amount of padding performed in `retrieve` data transfers, which can be very large in irregular workloads like the SpMV kernel.

OBSERVATION 14:
The *equally-wide* and *variable-sized* 2D schemes require fine-grain parallel transfers *at DRAM bank granularity* to be supported by the PIM system (i.e., *zero* padding in *parallel* `retrieve` data transfers), to achieve high performance.

Fourth, we find that the number of vertical partitions that provides the best performance depends on the sparsity pattern of the input matrix, the data type, and the underlying hardware parameters (e.g., number of PIM cores, off-chip memory bus bandwidth, transfer latency costs between main memory and PIM-enabled memory, characteristics and microarchitecture of the host CPU cores that perform the `merge` step). For example, with the int8 data type, `DCOO` performs best for `hgc` and `mem` matrices with 8 and 4 vertical partitions, respectively. Instead, with the fp64 data type, `DCOO` performs best for `hgc` and `mem` matrices with 16 and 8 vertical partitions, respectively. We leave for future work the exploration of selection methods for the number of vertical partitions that provide best SpMV execution. Overall, based on our analysis we conclude that the parallelization scheme that achieves the best performance in SpMV depends on both the input sparse matrix and the hardware characteristics of the PIM system.

OBSERVATION 15:
There is *no one-size-fits-all* parallelization approach for SpMV in PIM systems, since the performance of each parallelization scheme depends on the characteristics of the input matrix and the underlying PIM hardware.

6.2.2 ***Analysis of Compressed Formats*** We compare the performance achieved by various compressed matrix formats for each of the three types of the 2D partitioning technique. The goal of this experiment is to find the best-performing compressed format for each 2D partitioning technique. Figures 21, 22, and 23 compare the performance of compressed matrix formats for the *equally-sized*, *equally-wide* and *variable-sized* 2D partitioning techniques, respectively. We use 2048 DPUs and the int32 data type having 4 vertical partitions. See Appendix A.3 for the complete evaluation on all large matrices.

We draw two findings. First, as already explained, `kernel` time of the *equally-sized* scheme is limited by the DPU (or a few DPUs) assigned to the 2D tile with the largest number of non-zero elements. In scale-free matrices (e.g., `in` and `ask`), the disparity in the number of non-zero elements across 2D tiles is higher than in regular matrices (e.g., `pfg` and `bns`), causing `kernel` time to be a larger portion of the total execution time. Second, we find that CSR and BCSR formats perform worse than COO and BCOO formats, especially in *equally-wide* and *variable-sized* schemes, due to higher `kernel` times. In CSR and BCSR formats, data partitioning across DPUs and/or across tasklets within a DPU is performed at row and block-row granularity, respectively. Thus, CSR and BCSR formats can cause higher non-zero element imbalance across processing units compared to COO and BCOO formats. Overall, COO and BCOO formats

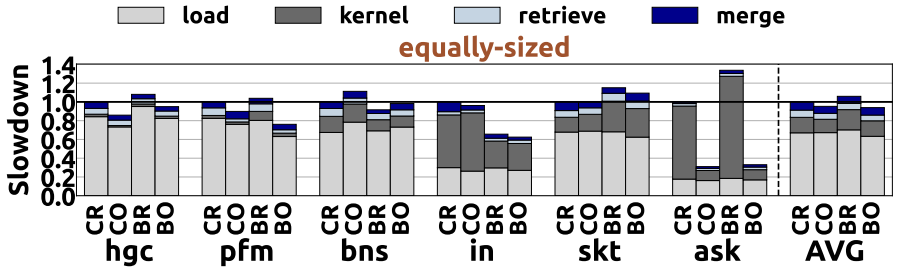


Fig. 21. End-to-end execution time breakdown of the *equally-sized* 2D partitioning technique for CR: DCSR, CO: DCOO, BR: DBCSR and BO: DBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of DCSR.

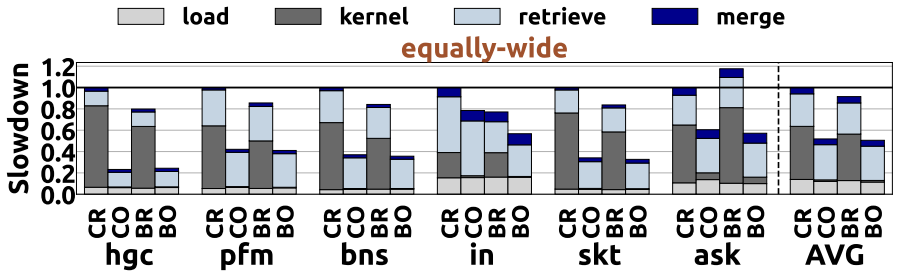


Fig. 22. End-to-end execution time breakdown of the *equally-wide* 2D partitioning technique for CR: RBDCSR, CO: RBDCOO, BR: RBDBCSR and BO: RBDBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of RBDCSR.

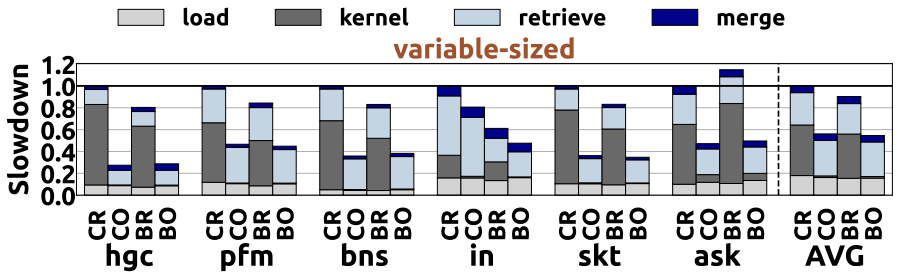


Fig. 23. End-to-end execution time breakdown of the *variable-sized* 2D partitioning technique for CR: BDCSR, CO: BDCOO, BR: BDBCSR and BO: BDBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of BDCSR.

outperform CSR and BCSR formats by 1.59 × and 1.53 × (averaged across all three types of 2D partitioning techniques), respectively.

OBSERVATION 16:

The compressed matrix format used to store the input matrix determines the data partitioning across DRAM banks of PIM-enabled memory. Thus, it affects the load balance across PIM cores with the corresponding performance implications. Overall, COO and BCOO formats outperform CSR and BCSR formats, because they provide higher non-zero element balance across PIM cores.

6.2.3 ***Comparison of 2D Partitioning Techniques*** We compare the best-performing SpMV implementations of all 2D partitioning schemes, i.e., using the COO and BCOO formats. Figures 24 and 25 compare the throughput (in GOperations per second) and the performance, respectively, of DCOO, DBCOO, RBDCOO, RBDBCOO, BDCOO, BDBCOO schemes using 2048 DPUs and the int32 data type. For each implementation, we vary the number of vertical partitions from 2 to 32, in steps of multiple of 2, and select the best-performing execution throughput. We draw two conclusions.

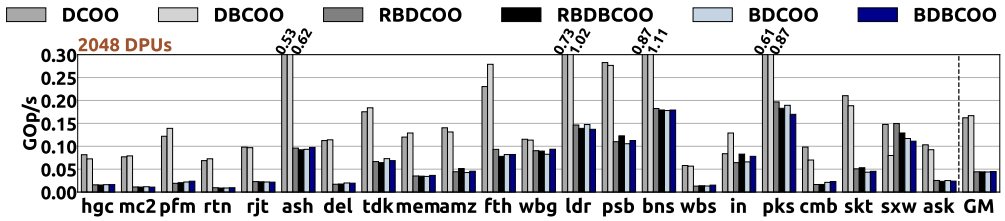


Fig. 24. Throughput achieved by 2D partitioning techniques using the COO and BCOO formats, 2048 DPUs and the int32 type.

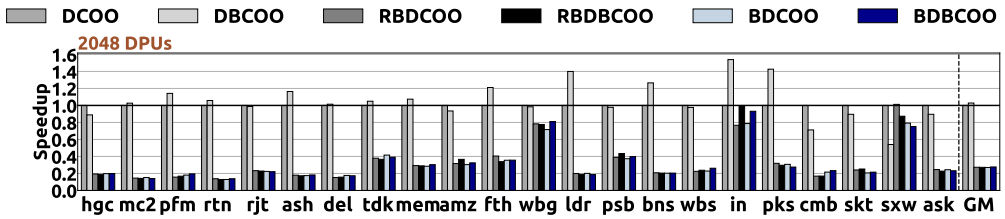


Fig. 25. Performance comparison of 2D partitioning techniques using COO and BCOO formats, 2048 DPUs and the int32 type. Performance is normalized to that of DCOO.

First, similarly to 1D-partitioned kernels, matrices that exhibit block pattern (e.g., ash, ldr, bns, pks) have the highest throughput. Second, the *equally-wide* and *variable-sized* schemes perform similarly, i.e., their performance varies only by ±1.1% on average. Even though the *variable-sized* technique can improve the non-zero element balance across DPUs, and thus kernel time, compared to the *equally-wide* technique, the total execution time does not improve. In the UPMEM PIM system, performance of both techniques is severely bottlenecked by data transfer overheads due to a large amount of padding needed to retrieve results from PIM-enabled DIMMs to the host CPU. Third, we find that the *equally-sized* technique outperforms the *equally-wide* and *variable-sized* techniques by 3.71× on average, because it achieves lower data transfer overheads. The *equally-wide* and *variable-sized* techniques provide near-perfect non-zero element balance across DPUs, but they significantly increase the retrieve data transfer costs due to the large amount of padding with empty bytes performed. As a result, we recommend software designers to explore *relaxed* load balancing schemes, i.e., schemes that trade off computation balance across PIM cores for lower amounts of data transfer.

6.3 Comparison of 1D and 2D Partitioning Techniques

We compare the throughput (in GOperations per second) and the performance achieved by the best-performing 1D- and 2D-partitioned kernels in Figures 26 and 27, respectively. For

1D partitioning, we use the lock-free COO (`COO.nnz-lf`) and coarse-grained locking BCOO (`BCOO.block`) kernels. For each matrix, we vary the number of DPUs from 64 to 2528, and select the best-performing end-to-end execution throughput. For 2D partitioning, we use the *equally-sized* COO (`DCOO`) and BCOO (`DBCOO`) kernels with 2528 DPUs. For each matrix, we vary the number of vertical partitions from 2 to 32 (in steps of multiple of 2), and select the best-performing end-to-end execution throughput. The numbers shown over each bar of Figure 26 present the number of DPUs that provide the best-performing end-to-end execution throughput for each input-scheme combination.

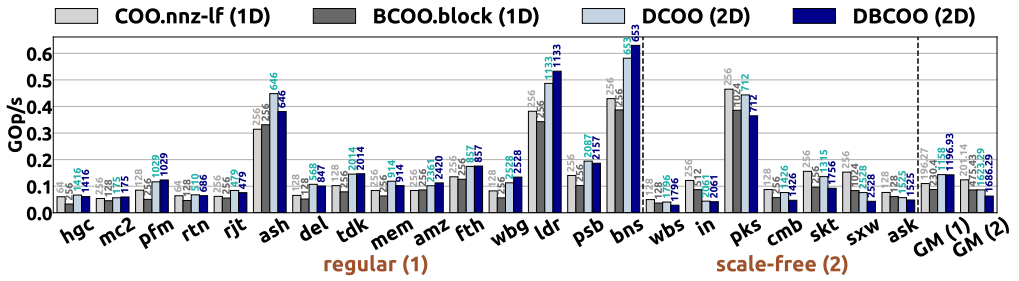


Fig. 26. Throughput achieved by the best-performing 1D- and 2D-partitioned kernels for the fp32 data type.

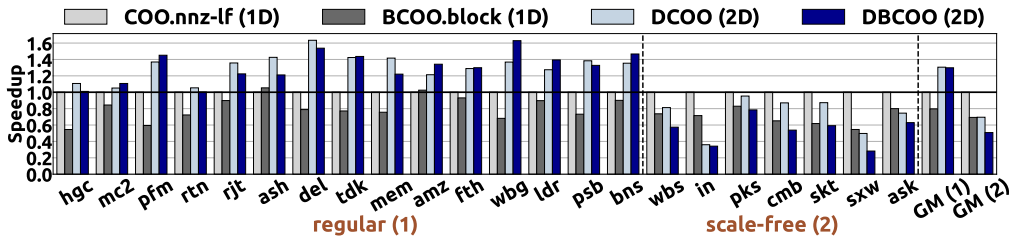


Fig. 27. Performance comparison of the best-performing 1D- and 2D-partitioned kernels for the fp32 data type. Performance is normalized to that of `COO.nnz-lf`.

We draw two conclusions. First, we find that the best performance is achieved using a much smaller number of DPUs than the available DPUs on the system. In the 1D-partitioned kernels (i.e., `COO.nnz-lf` and `BCOO.block`), replicating the input vector into a large number of DPUs significantly increases the `load` data transfer costs. Thus, best performance is achieved using 253 DPUs on average across all matrices. In the 2D-partitioned kernels (i.e., `DCOO` and `DBCOO`), creating *equally-sized* 2D tiles leads to a large disparity in non-zero element count across tiles, causing many tiles to be empty, i.e., without *any* non-zero element. Thus, best performance is achieved using 1329 DPUs on average across all matrices, since DPUs associated with empty tiles are idle.

OBSERVATION 17:
Expensive data transfers to PIM-enabled memory performed via the narrow memory bus impose significant performance overhead to end-to-end SpMV execution. Thus, it is hard to fully exploit all available PIM cores of the system.

Second, we observe that in regular matrices, the 2D-partitioned kernels outperform the 1D-partitioned kernels by 1.45× on average. This is because the 2D-partitioned kernels use a higher

number of DPUs, and thus their `kernel` times are lower. In contrast, in scale-free matrices, the 1D-partitioned kernels outperform the 2D-partitioned kernels by 1.41× on average. This because the *equally-sized* 2D technique significantly increases the non-zero element disparity across DPUs, i.e., `kernel` time is bottlenecked by only one DPU or a few DPUs that process a much larger number of non-zero elements compared to the rest.

OBSERVATION 18:
In *regular* matrices, 2D-partitioned kernels outperform 1D-partitioned kernels, since the former provide a better trade-off between computation and data transfer overheads. In contrast, in *scale-free* matrices, 2D-partitioned kernels perform worse than 1D-partitioned kernels, since the former's performance is limited by one DPU or a few DPUs that process the largest number of non-zero elements.

7 Comparison with CPUs and GPUs

We compare SpMV execution on the UPMEM PIM architecture to a state-of-the-art CPU and a state-of-the-art GPU in terms of performance and energy consumption. Our goal is to quantify the potential of the UPMEM PIM architecture on the widely-used memory-bound SpMV kernel.

We compare the UPMEM PIM system with 2528 DPUs to an Intel Xeon CPU [66] and an NVIDIA Tesla V100 GPU [118], the characteristics of which are shown in Table 5. We use peakperf [123] and stream [138] for CPU and GPU systems to calculate the peak performance, memory bandwidth, and Thermal Design Power (TDP). For the UPMEM PIM system, we estimate the peak performance as $Total\_DPUs * AT$, where the arithmetic throughput (AT) is presented in Appendix B, the total bandwidth as $Total\_DPUs * Bandwidth\_DPU$, where the Bandwidth_DPU is 700 MB/s [30, 45, 53], and TDP as $(Total\_DPUs/DPUs\_per\_chip) * 1.2W/chip$ from prior work [30, 45, 53].

System	Process Node	Total Cores	Frequency	Peak Performance	Memory Capacity	Total Bandwidth	TDP
Intel Xeon 4110 CPU [66]	14 nm	2x8 x86 cores (2x16 threads)	2.1 GHz	660 GFLOPS	128 GB	23.1 GB/s	2x85 W
NVIDIA Tesla V100 [118]	12 nm	5120 CUDA cores	1.25 GHz	14.13 TFLOPS	32 GB	897 GB/s	300 W
PIM System	2x nm	2528 DPUs	350 MHz	4.66 GFLOPS	159 GB	1.77 TB/s	379 W

Table 5. Evaluated CPU, GPU, and UPMEM PIM Systems.

7.1 Performance Comparison

For the CPU system, we use the optimized CSR kernel from the TACO library [77]. For the GPU system, we use the CSR5 CUDA [25, 96] for the int32 data type and cuSparse [26] for the other data types. For the UPMEM PIM system, we use the lock-free COO 1D-partitioned kernel (**`COO`**`.nnz-lf`) and the *equally-sized* COO 2D-partitioned kernel (**`DCOO`**). In the former, we run experiments from 64 to 2528 DPUs, and in the latter, we use 2528 DPUs, and vary the number of vertical partitions from 2 to 32, in steps of multiple of 2. In both schemes, we select the best-performing end-to-end execution throughput. We also include the lock-free COO 1D-partitioned kernel using 2528 DPUs, named **`COO.kl`**, to evaluate SpMV execution using *all* available DPUs of the system.

Figure 28 shows the throughput of SpMV (in GOperations per second) in all systems, comparing both the end-to-end execution throughput (i.e., including the `load` and `retrieve` data

transfer costs for the input and output vectors in case of the UPMEM PIM and GPU systems), and only the actual kernel throughput (i.e., including the `kernel` time in DPUs and the `merge` time in host CPU for the UPMEM PIM system).

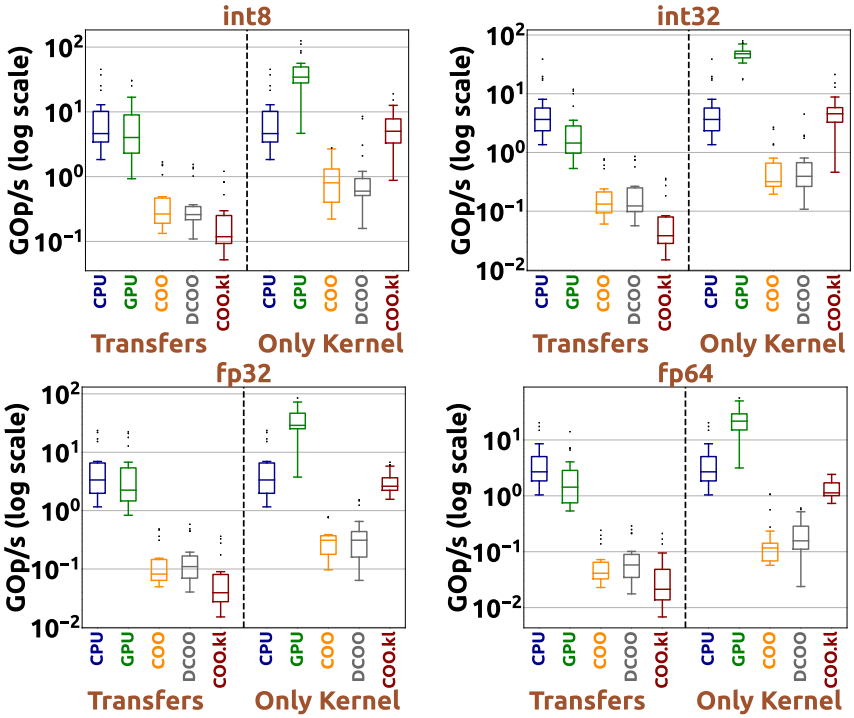


Fig. 28. Performance comparison between the UPMEM PIM sytem, Intel Xeon CPU and Tesla V100 GPU.

We draw three conclusions. First, when data transfer costs to/from host CPU are included, CPU outperforms both GPU and UPMEM PIM systems, since data transfers impose high overhead. When only the actual kernel time is considered, GPU performs best, since it is the system that provides the highest computation throughput, e.g., 14.13 TFlops for the fp32 data type. Second, we evaluate the portion of the machine's peak performance achieved on SpMV in all systems, and observe that SpMV execution on the UPMEM PIM system achieves a much higher fraction of the peak performance compared to CPU and GPU systems. For the fp32 data type, SpMV achieves on average 0.51% and 0.21% of the peak performance in CPU and GPU, respectively, while it achieves 51.7% of the peak performance in the UPMEM PIM system using the `COO.kl` scheme. Achieving a high portion of machine's peak performance is highly desirable, since the software highly exploits the computation capabilities of the underlying hardware. This way, it improves the processor/resource utilization, and the cost of ownership of the underlying hardware. Third, we observe that when all DPUs are used, as in `COO.kl`, SpMV execution on the UPMEM PIM outperforms SpMV execution on the CPU by 1.09× and 1.25× for the int8 and int32 data types, respectively, the multiplication of which is supported by hardware. In contrast, SpMV execution on the UPMEM PIM performs 1.27× and 2.39× worse

than SpMV execution on the CPU for the fp32 and fp64 data types, the multiplication of which is software emulated in DPUs.

OBSERVATION 19:
SpMV execution can achieve a *significantly higher* fraction of the peak performance on real memory-centric PIM architectures compared to that on processor-centric CPU and GPU systems, since PIM architectures greatly mitigate data movement costs.

7.2 Energy Comparison

For energy measurements, we consider only the actual kernel time in all systems (in the UPMEM PIM we consider the `kernel` and `merge` steps of SpMV execution). We use Intel RAPL [73] on the CPU, and NVIDIA SMI [117] on the GPU. For the UPMEM PIM system, we measure the number of cycles, instructions, WRAM accesses and MRAM accesses of each DPU, and estimate energy with energy weights provided by the UPMEM company [144].

Figure 29 shows the energy consumption (in Joules) and performance per energy (in GOps/W) for all systems. We draw three findings. First, GPU provides the lowest energy on SpMV over the other two systems, since the energy results typically follow the performance results. Second, we find that the 2D-partitioned kernel, i.e., `DCOO`, consumes more energy than the 1D-partitioned kernels, i.e., `COO` and `COO.kl`, due to the energy consumed in host CPU cores. CPU cores merge a large number of partial results in the 2D-partitioned kernels to assemble the final output vector, thereby increasing the energy consumption. Finally, we find that the 1D-partitioned kernels provide better energy efficiency on SpMV over the CPU system, when the multiplication operation is supported by hardware. Specifically, 1D-partitioned kernels provide 3.16× and 4.52× less energy consumption, and 1.74× and 1.14× better performance per energy over the CPU system for the int8 and int32 data types, respectively.

OBSERVATION 20:
Real PIM architectures can provide high energy-efficiency on SpMV execution.

7.3 Discussion

These evaluations are useful for programmers to anticipate how much performance and energy savings memory-centric PIM systems can provide on SpMV over commodity processor-centric CPU and GPU systems. However, our evaluated SpMV kernels do not constitute the best-performing approaches for *all* matrices. Designing methods to select the best-performing SpMV parallelization scheme depending on the particular characteristics of the input matrix would further improve performance and energy savings of SpMV execution on memory-centric PIM systems. Moreover, the UPMEM PIM hardware is still maturing and is expected to run at a higher frequency in the near future (500 MHz instead of 350 MHz) [45, 144]. Hence, SpMV execution on the UPMEM PIM architecture might achieve even higher performance and energy benefits over the results we report in this comparison. Finally, note that our proposed *SparseP* kernels can be adapted and evaluated on other current and future real PIM systems with potentially higher computation capabilities and energy-efficiency than the UPMEM PIM system.

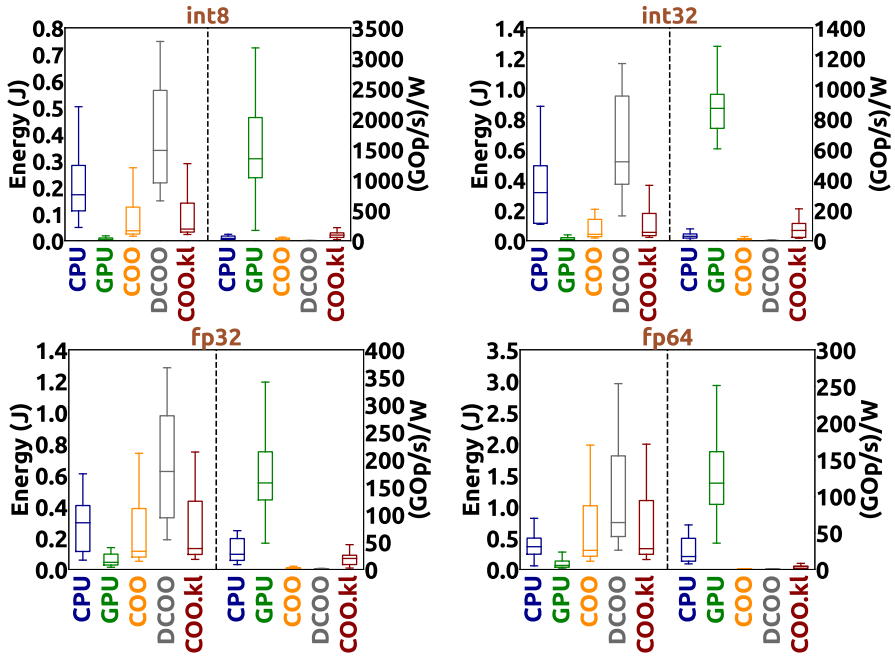


Fig. 29. Energy comparison between the UPMEM PIM system, Intel Xeon CPU and Tesla V100 GPU.

8 Key Takeaways and Recommendations

This section summarizes our key takeaways in the form of recommendations to improve multiple aspects of PIM hardware and software.

Recommendation #1. *Design algorithms that provide high load balance across threads of a PIM core in terms of computations, loop control iterations, synchronization points and memory accesses.* Section 5 shows that in matrices and formats where the parallelization scheme used causes *high disparity* in the number non-zero elements/blocks/rows processed across threads of a PIM core, or the number of lock acquisitions/lock releases/DRAM memory accesses performed across threads, SpMV performance severely degrades in compute-bound DPUs [45, 53]. Therefore, from a programmer's perspective providing high operation balance across parallel threads is of vital importance in low-area and low-power PIM cores with relatively low computation capabilities [45, 53].

Recommendation #2. *Design compressed data structures that can be effectively partitioned across DRAM banks, with the goal of providing high computation balance across PIM cores.* Sections 6.1.1 and 6.2.2 demonstrate that (i) the compressed matrix format used to store the input matrix determines the data partitioning across DRAM banks of PIM-enabled memory, and (ii) SpMV execution using CSR and BCSR formats performs significantly worse than SpMV execution using COO and BCOO formats. This is because the matrix is stored in row- or block-row order for CSR and BCSR formats, respectively, and thus data partitioning across DRAM banks is performed at row- or block-row granularity, respectively, leading to high non-zero element imbalance across PIM cores. Therefore, we recommend that programmers design compressed data structures that can provide effective data partitioning schemes with high computation balance across thousands of PIM cores.

Recommendation #3. *Design adaptive algorithms that (i) trade off computation balance across PIM cores for lower data transfer costs to PIM-enabled memory, and (ii) adapt their configuration to the particular patterns of each input given, as well as the characteristics of the PIM hardware.* Our analysis in Sections 6.1.1, 6.2.1 and 6.2.3 demonstrates that the best-performing SpMV execution on the UPMEM PIM system can be achieved using algorithms that (i) trade off computation for lower data transfer costs, and (ii) select the load balancing strategy and data partitioning policy based on the particular sparsity pattern of the input matrix. In addition, the performance of each balancing scheme and data partitioning technique for SpMV execution highly depends on the characteristics of the underlying PIM hardware, as we explain in Section 6.2.1. To this end, we recommend that software designers implement heuristics and selection methods for their algorithms to adapt their configuration to the underlying hardware characteristics of the PIM system and the input matrix.

Recommendation #4. *Provide low-cost synchronization support and hardware support to enable concurrent memory accesses by multiple threads to the local DRAM memory bank to increase parallelism in a multithreaded PIM core.* Section 5 shows that (i) lock acquisitions/releases can cause high overheads in the DPU pipeline, and (ii) fine-grained locking approaches to increase parallelism in critical sections do not improve performance over coarse-grained approaches in the UPMEM PIM hardware. This is because the DMA engine of the DPU serializes DRAM memory accesses included in the critical sections. Based on these key takeaways, we recommend that hardware designers provide lightweight synchronization mechanisms for multithreaded PIM cores [44], and enable concurrent access to local DRAM memory arrays to increase execution parallelism. For example, sub-array level parallelism [20, 75] or multiple DRAM banks per PIM core could be supported in the PIM hardware to improve parallelism.

Recommendation #5. *Optimize the broadcast collective operation in data transfers from main memory to PIM-enabled memory to minimize overheads of copying the input data into all DRAM banks in the system.* Figures 14 and 15 show that SpMV execution using the 1D partitioning technique cannot scale up to a high number of PIM cores. This is because it is severely limited by data transfer costs to broadcast the input vector into *each* DRAM bank of PIM-enabled DIMMs via the narrow off-chip memory bus. To this end, we suggest that hardware and system designers provide a fast broadcast collective primitive to DRAM banks of PIM-enabled memory DIMMs.

Recommendation #6. *Optimize the gather collective operation at DRAM bank granularity for data transfers from PIM-enabled memory to host CPU to minimize overheads of retrieving the output results.* Figures 18, 19 and 20 demonstrate that SpMV execution using the *equally-wide* and *variable-sized* 2D partitioning schemes is severely limited by data transfers to retrieve results for the output vector from DRAM banks of PIM-enabled DIMMs. This is due to two reasons: (i) 2D-partitioned kernels create a large number of partial results that need to be transferred from PIM-enabled memory to the host CPU via the narrow memory bus in order to assemble the final output vector, and (ii) the UPMEM PIM system has the limitation that the transfer sizes from/to all DRAM banks involved in the same parallel transfer need to be the same, and therefore a large amount of padding with empty bytes is performed in the *equally-wide* and *variable-sized* schemes. Therefore, we suggest that hardware and system designers provide an optimized *gather* primitive to efficiently collect results from multiple DRAM banks to host CPU, and support parallel fine-grained data transfers from PIM-enabled memory to host CPU *at DRAM bank granularity* to avoid padding with empty bytes.

Recommendation #7. *Design high-speed communication channels and optimized libraries for data transfers to/from thousands of DRAM banks of PIM-enabled memory.* Section 7 demonstrates that SpMV execution on the memory-centric UPMEM PIM system achieves a much higher fraction of the machine's peak performance (on average 51.7% for the 32-bit float data type), compared to processor-centric CPU and GPU systems. However, the end-to-end performance of both 1D- and 2D-partitioned kernels is significantly limited by data transfer overheads on the narrow memory bus. To this end, we recommend that the hardware architecture and the software stack of real PIM systems be enhanced with low-cost and fast data transfers to/from PIM-enabled memory modules.

9 Related Work

To our knowledge, this is the first work that (i) extensively characterizes the Sparse Matrix Vector Multiplication (SpMV) kernel in a real PIM system, and (ii) presents an open-source SpMV library for real-world PIM systems. We briefly discuss closely related prior work.

Sparse Matrix Kernels in PIM Systems. Xie et al. [153] design heterogenous PIM units to accelerate SpMV in HMC-based PIM systems. Fujiki et al. [40] enhance the memory controllers of GPUs with PIM cores to transform the matrix from CSR to DCSR format [59] on the fly to minimize memory traffic. Zhu et al. [164] propose a PIM accelerator for Sparse Matrix Matrix Multiplication. These works propose hardware designs for sparse matrix kernels, while our work studies software optimizations to accelerate the SpMV execution on real PIM systems.

SpMV in Commodity Systems. Numerous prior works propose optimized SpMV algorithms for CPUs [18, 33–35, 77, 103, 112, 141, 150, 151], GPUs [13, 24, 51, 60, 94, 137, 139, 152, 154, 154, 155], heterogeneous CPU-GPU systems [6, 65, 156, 158], and distributed CPU systems [10, 19, 71, 85, 93, 120]. Optimized SpMV kernels for processor-centric CPU and GPU systems exploit the shared memory model of these systems and data locality in deep cache hierarchies. However, these kernels cannot be directly mapped to most near-bank PIM systems, which have a distributed memory model and a shallow cache hierarchy. Most well-tuned SpMV kernels for distributed CPU and CPU-GPU systems improve performance by overlapping computation with communication among processing units, and exploiting data locality in large cache memories. In contrast, real near-bank PIM architectures are fundamentally different from CPU-GPU systems, since they are *highly distributed*, i.e., there is no direct communication among PIM cores, and include a shallow memory hierarchy. Thus, SpMV kernels designed for common processor-centric systems cannot be directly used in near-bank PIM systems.

Compressed Matrix Storage Formats. Prior works propose a range of compressed matrix storage formats [5, 11, 17, 59, 63, 76, 78, 78, 79, 83, 95, 97, 101, 102, 105, 124, 125, 127, 129, 131, 148, 154, 157] and selection methods to find the most efficient compressed format [2, 7, 8, 88, 89, 100, 114, 130, 139, 140, 161, 162]. In this work, we extensively explore the four most widely used compressed formats, and observe that the compressed format (i) needs to provide good balance between computation and memory accesses inside the core pipeline, and (ii) affects load balancing across PIM cores, with corresponding performance implications. Thus, some compressed formats designed for commodity processor-centric systems might not be suitable or efficient for real PIM systems. We leave the exploration of other PIM-suitable formats for future work.

Hardware Accelerators for SpMV. Recent works design accelerators for SpMV [39, 48, 69, 90, 106, 115, 128, 142] or other sparse kernels [3, 56, 104, 116, 121, 126, 159, 163]. In contrast,

our work proposes software optimizations and provides the first characterization study of SpMV on a real PIM system.

10 Conclusion

We present *SparseP*, the first open-source SpMV library for real Processing-In-Memory (PIM) systems, and conduct the first comprehensive characterization analysis of the widely-used SpMV kernel on a real-world PIM architecture.

First, we design efficient SpMV kernels for real PIM systems. Our proposed *SparseP* software package supports (1) a wide range of data types, (2) two types of well-crafted data partitioning techniques of the sparse matrix to DRAM banks of PIM-enabled memory, (3) the most popular compressed matrix formats, (4) a wide variety of load balancing schemes across PIM cores, (5) several load balancing schemes across threads of a multithreaded PIM core, and (6) three synchronization approaches among threads within PIM core.

Second, we conduct an extensive characterization study of *SparseP* kernels on the state-of-the-art UPMEM PIM system. We analyze SpMV execution on one single multithreaded PIM core and thousands of PIM cores using 26 sparse matrices with diverse sparsity patterns. We also compare the performance and energy consumption of SpMV on the UPMEM PIM system with those of state-of-the-art CPU and GPU systems to quantify the potential of a real memory-centric PIM architecture on the widely-used SpMV kernel over conventional processor-centric architectures. Our analysis of *SparseP* kernels provides programming recommendations for software designers, as well as suggestions and hints for hardware and system designers of future PIM systems.

We believe and hope that our work will provide valuable insights to programmers in the development of efficient sparse linear algebra kernels and other irregular kernels from different application domains tailored for real PIM systems, as well as to architects and system designers in the development of future memory-centric computing systems.

Acknowledgments

We thank the UPMEM company for valuable support. We thank the anonymous reviewers from SIGMETRICS 2022, and our shepherd, Bhuvan Urgaonkar, for their comments and suggestions. We acknowledge the support of SAFARI Research Group's industrial partners, especially ASML, Facebook, Google, Huawei, Intel, Microsoft, VMware, the Semiconductor Research Corporation and the ETH Future Computing Laboratory. Christina Giannoula is funded for her postgraduate studies from Foundation for Education and European Culture. *SparseP* software package is publicly available at [49].

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 *ISCA*.

[2] Bahar Asgari, Ramyad Hadidi, Joshua Dierberger, Charlotte Steinichen, and Hyesoon Kim. 2020. Copernicus: Characterizing the Performance Implications of Compression Formats Used in Sparse Workloads. In *CoRR*. https://arxiv.org/abs/2011.10932

[3] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. 2020. ALRESCHA: A Lightweight Reconfigurable Sparse-Computation Accelerator. In *HPCA*.

[4] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. 2016. Chameleon: Versatile and PracticalNear-DRAM Acceleration Architecture for Large Memory Systems. In *MICRO*.

[5] Mehmet Belgin, Godmar Back, and Calvin J. Ribbens. 2009. Pattern-Based Sparse Matrix Representation for Memory-Efficient SMVM Kernels. In *ICS*.
[6] Akrem Benatia, Weixing Ji, and Yizhuo Wang. 2019. Sparse Matrix Partitioning for Optimizing SpMV on CPU-GPU Heterogeneous Platforms. In *IJHPCA*.
[7] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2016. Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU. In *ICPP*.
[8] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2018. BestSF: A Sparse Meta-Format for Optimizing SpMV on GPU. In *TACO*.
[9] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *IPDPS*.
[10] Rob H. Bisseling and Wouter Meesen. 2005. Communication Balancing in Parallel Sparse Matrix-Vector Multiplication. In *ETNA. Electronic Transactions on Numerical Analysis*.
[11] Åke Björck. 1996. Numerical Methods for Least Squares Problems. In *SIAM*.
[12] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. 2003. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In *SIGGRAPH*.
[13] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. 2003. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In *ACM Transactions on Graphics*.
[14] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. 2019. CoNDA: Efficient Cache Coherence Support for Near-data Accelerators. In *ISCA*.
[15] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. 2017. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. In *CAL*.
[16] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *WWW*.
[17] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In *SPAA*.
[18] Aydin Buluç, Samuel Williams, Leonid Oliker, and James Demmel. 2011. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In *IPDPS*.
[19] Beata Bylina, Jarosław Bylina, Przemysław Stpiczyński, and Dominik Szałkowski. 2014. Performance Analysis of Multicore and Multinodal Implementation of SpMV Operation. In *FedCSIS*.
[20] Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti, Alaa R. Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. 2014. Improving DRAM performance by parallelizing refreshes with accesses. In *HPCA*.
[21] Benjamin Y. Cho, Jeageun Jung, and Mattan Erez. 2021. Accelerating Bandwidth-Bound Deep Learning Inference with Main-Memory Accelerators. In *SC*.
[22] Benjamin Y. Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. 2020. Near Data Acceleration with Concurrent Host Access. In *ISCA*.
[23] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. 2019. Concurrent Data Structures with Near-Data-Processing: An Architecture-Aware Implementation. In *SPAA*.
[24] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. In *PpopP*.
[25] CSR5. 2015. CSR5 Cuda. https://github.com/weifengliu-ssslab/Benchmark_SpMV_using_CSR5
[26] cuSparse. 2021. cuSparse. https://docs.nvidia.com/cuda/cusparse/index.html
[27] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. In *IEEE Comput. Sci. Eng.*
[28] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2018. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. In *IEEE TCAD*.
[29] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. In *TOMS*.
[30] F. Devaux. 2019. The True Processing In Memory Accelerator. In *Hot Chips*.
[31] Jack Dongarra, Andrew Lumsdaine, Xinhui Niu, Roldan Pozoz, and Karin Remington. 1994. Sparse Matrix Libraries in C++ for High Performance Architectures. In *Mathematics*.
[32] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. 2017. The Mondrian Data Engine. In *ISCA*.

[33] Athena Elafrou, G. Goumas, and N. Koziris. 2017. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors. In *ICPP*.
[34] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. 2019. Conflict-Free Symmetric Sparse Matrix-Vector Multiplication on Multicore Architectures. In *SC*.
[35] Athena Elafrou, Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. 2018. SparseX: A Library for High-Performance Sparse Matrix-Vector Multiplication on Multicore Platforms. In *ACM TOMS*.
[36] R. D. Falgout. 2006. An Introduction to Algebraic Multigrid. In *Computing in Science Engineering*.
[37] Robert D Falgout and Ulrike Meier Yang. 2002. hypre: A Library of High Performance Preconditioners. In *ICCS*.
[38] Ivan Fernandez, Ricardo Quislant, Christina Giannoula, Mohammed Alser, Juan Gómez-Luna, Eladio Gutiérrez, Oscar Plata, and Onur Mutlu. 2020. NATSA: A Near-Data Processing Accelerator for Time Series Analysis. In *ICCD*.
[39] 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 *FCCM*.
[40] Daichi Fujiki, Niladrish Chatterjee, Donghyuk Lee, and Mike O'Connor. 2019. Near-Memory Data Transformation for Efficient Sparse Matrix Multi-Vector Multiplication. In *SC*.
[41] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *PACT*.
[42] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *ASPLOS*.
[43] Saugata Ghose, Amirali Boroumand, Jeremie Kim, Juan Gómez-Luna, and Onur Mutlu. 2019. Processing-in-Memory: A Workload-Driven Perspective. In *IBM JRD*.
[44] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios I. Goumas, and Onur Mutlu. 2021. SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures. In *HPCA*.
[45] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2021. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. In *CoRR*. https://arxiv.org/abs/2105.03814
[46] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. 2008. Understanding the Performance of Sparse Matrix-Vector Multiplication. In *PDP*.
[47] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. 2009. Performance Evaluation of the Sparse Matrix-Vector Multiplication on Modern Architectures. In *J. Supercomput.*
[48] Paul Grigoras, Pavel Burovskiy, Eddie Hung, and Wayne Luk. 2015. Accelerating SpMV on FPGAs by Compressing Nonzero Values. In *FCCM*.
[49] SAFARI Research Group. 2022. *SparseP* Software Package. https://github.com/CMU-SAFARI/SparseP
[50] Peng Gu, Xinfeng Xie, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. 2020. iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture. In *ISCA*.
[51] Ping Guo, Liqiang Wang, and Po Chen. 2014. A Performance Modeling and Optimization Analysis Tool for Sparse Matrix-Vector Multiplication on GPUs. In *IEEE TPDS*.
[52] Udit Gupta, Xiaodong Wang, Maxim Naumov, Carole-Jean Wu, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2019. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. In *CoRR*.
[53] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2021. Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-In-Memory Hardware. In *IGSC*.
[54] Ramyad Hadidi, Bahar Asgari, Burhan Ahmad Mudassar, Saibal Mukhopadhyay, Sudhakar Yalamanchili, and Hyesoon Kim. 2017. Demystifying the Characteristics of 3D-stacked Memories: A case Study for Hybrid Memory Cube. In *IISWC*.
[55] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. 2021. SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM. In *ASPLOS*.
[56] 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 *MICRO*.

[57] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. 2021. ALTO: Adaptive Linearized Storage of Sparse Tensors. In *ICS*.
[58] Pascal Hénon, Pierre Ramet, and Jean Roman. 2002. PASTIX: A High-Performance Parallel Direct Solver for Sparse Symmetric Positive Definite Systems. In *PMAA*.
[59] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Çatalyürek, Srinivasan Parthasarathy, and P. Sadayappan. 2018. Efficient Sparse-Matrix Multi-Vector Product on GPUs. In *HPDC*.
[60] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Çatalyürek, Srinivasan Parthasarathy, and P. Sadayappan. 2018. Efficient Sparse-Matrix Multi-Vector Product on GPUs. In *HPDC*.
[61] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation. In *ICCD*.
[62] Jiayi Huang, Ramprakash Reddy Puli, Pritam Majumder, Sungkeun Kim, Rahul Boyapati, Ki Hwan Yum, and Eun Jung Kim. 2019. Active-Routing: Compute on the Way for Near-Data Processing. In *HPCA*.
[63] Eun-Jin Im and Katherine A. Yelick. 1999. Optimizing Sparse Matrix Vector Multiplication on SMP. In *PPSC*.
[64] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization Framework for Sparse Matrix Kernels. In *The International Journal of High Performance Computing Applications*.
[65] Sivaramakrishna Bharadwaj Indarapu, Manoj Maramreddy, and Kishore Kothapalli. 2014. Architecture- and Workload- Aware Heterogeneous Algorithms for Sparse Matrix Vector Multiplication. In *COMPUTE*.
[66] Intel. 2017. Intel Xeon Silver 4110 Processor. https://ark.intel.com/content/www/us/en/ark/products/123547/intel-xeon-silver-4110-processor-11m-cache-2-10-ghz.html
[67] JEDEC. 2012. JESD79-4 DDR4 SDRAM standard.
[68] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. HBM DRAM Technology and Architecture. In *IMW*.
[69] 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 *MICRO*.
[70] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2009. Perfomance Models for Blocked Sparse Matrix-Vector Multiplication Kernels. In *ICPP*.
[71] Enver Kayaaslan, Bora Uçar, and Cevdet Aykanat. 2015. Semi-Two-Dimensional Partitioning for Parallel Sparse Matrix-Vector Multiplication. In *IPDPS Workshop*.
[72] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David Brooks, Vikas Chandra, Utku Diril, et al. 2020. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *ISCA*.
[73] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. 2018. Rapl in Action: Experiences in Using RAPL for Power Measurements. In *TOMPECS*.
[74] Min-Jae Kim, Jeong-Geun Kim, Su-Kyung Yoon, and Shin-Dug Kim. 2021. Functionality-Based Processing-in-Memory Accelerator for Deep Convolutional Neural Networks. In *IEEE Access*.
[75] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. 2012. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*.
[76] David R Kincaid, Thomas C Oppe, and David M Young. 1989. Itpackv 2D User's Guide.
[77] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. TACO: A Tool to Generate Tensor Algebra Kernels. In *ASE*.
[78] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. 2008. Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression. In *CF*.
[79] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2011. CSX: An Extended Compression Format for Spmv on Shared Memory Systems. In *PPoPP*.
[80] Pranith Kumar and Hyesoon Kim. 2020. Parallel Hash Table Design for NDP Systems. In *MEMSYS*.
[81] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *MICRO*.
[82] 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 *ISSCC*.

[83] Daniel Langr and Pavel Tvrdík. 2016. Evaluation Criteria for Sparse Matrix Storage Formats. In *TPDS*.

[84] Dominique Lavenier, Remy Cimadomo, and Romaric Jodin. 2020. Variant Calling Parallelization on Processor-in-Memory Architecture. In *BIBM*.

[85] Seyong Lee and Rudolf Eigenmann. 2008. Adaptive Runtime Tuning of Parallel Sparse Matrix-Vector Multiplication on Distributed Memory Systems. In *ICS*.

[86] Sukhan Lee, Shin-Haeng Kang, Jaehoon Lee, H. Kim, Eojin Lee, Seung young Seo, H. Yoon, Seungwon Lee, K. Lim, Hyunsung Shin, Jinhyun Kim, O. Seongil, Anand Iyer, David Wang, K. Sohn, and N. Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product. In *ISCA*.

[87] J. Leskovec and R. Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. In *TIST*.

[88] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: An Input Adaptive Auto-Tuner for Sparse Matrix-Vector Multiplication. In *PLDI*.

[89] Kenli Li, Wangdong Yang, and Keqin Li. 2015. Performance Analysis and Optimization for SpMV on GPU Using Probabilistic Modeling. In *IEEE TPDS*.

[90] Colin Yu Lin, Zheng Zhang, Ngai Wong, and Hayden Kwok-Hay So. 2010. Design Space Exploration for Sparse Matrix-Matrix Multiplication on FPGAs. In *FPT*.

[91] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. In *IC*.

[92] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. 2015. Sparse Convolutional Neural Networks. In *CVPR*.

[93] Changxi Liu, Biwei Xie, Xin Liu, Wei Xue, Hailong Yang, and Xu Liu. 2018. Towards Efficient SpMV on Sunway Manycore Architectures. In *ICS*.

[94] Weifeng Liu and Brian Vinter. 2014. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *IPDPS*.

[95] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *ICS*.

[96] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *ICS*.

[97] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient Sparse Matrix-Vector Multiplication on X86-Based Many-Core Processors. In *ICS*.

[98] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-Memory Computing. In *SPAA*.

[99] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. 2020. Livia: Data-Centric Computing Throughout the Memory Hierarchy. In *ASPLOS*.

[100] Marco Maggioni and Tanya Berger-Wolf. 2013. AdELL: An Adaptive Warp-Balancing ELL Format for Efficient Sparse Matrix-Vector Multiplication on GPUs. In *ICPP*.

[101] Michele Martone. 2014. Efficient Multithreaded Untransposed, Transposed or Symmetric Sparse Matrix–Vector Multiplication with the Recursive Sparse Blocks Format. In *Parallel Computing*.

[102] Michele Martone, Salvatore Filippone, Marcin Paprzycki, and Salvatore Tucci. 2010. On BLAS Operations with Recursively Stored Sparse Matrices. In *SYNASC*.

[103] Duane Merrill and Michael Garland. 2016. Merge-Based Parallel Sparse Matrix-Vector Multiplication. In *SC*.

[104] Asit K. Mishra, Eriko Nurvitadhi, Ganesh Venkatesh, Jonathan Pearce, and Debbie Marr. 2017. Fine-grained Accelerators for Sparse Machine Learning Workloads. In *ASP-DAC*.

[105] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In *HiPEAC*, Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell (Eds.).

[106] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *MICRO*.

[107] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2019. Enabling Practical Processing In and Near Memory for Data-Intensive Computing. In *DAC*.

[108] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2019. Processing Data Where It Makes Sense: Enabling In-Memory Computation. In *MICPRO*.

[109] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2021. A Modern Primer on Processing in Memory. In *Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann*. https://arxiv.org/pdf/2012.03112.pdf

[110] Anirban Nag and Rajeev Balasubramonian. 2021. OrderLight: Lightweight Memory-Ordering Primitive for Efficient Fine-Grained PIM Computations. In *MICRO*.

[111] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *HPCA*.

[112] Naveen Namashivayam, Sanyam Mehta, and Pen-Chung Yew. 2021. Variable-Sized Blocks for Locality-Aware SpMV. In *CGO*.

[113] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. In *CoRR*.

[114] 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 *IPDPS*.

[115] Eriko Nurvitadhi, Asit Mishra, and Debbie Marr. 2015. A Sparse Matrix Vector Multiply Accelerator for Support Vector Machine. In *CASES*.

[116] Eriko Nurvitadhi, Asit Mishra, Yu Wang, Ganesh Venkatesh, and Debbie Marr. 2016. Hardware Accelerator for Analytics of Sparse Data. In *DAC*.

[117] NVIDIA. 2016. NVIDIA System Management Interface Program. http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf.

[118] NVIDIA. 2017. NVIDIA Tesla V100 GPU Architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[119] Geraldo F. Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. 2021. DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks. In *IEEE Access*.

[120] Brian A. Page and Peter M. Kogge. 2018. Scalability of Hybrid Sparse Matrix Dense Vector (SpMV) Multiplication. In *HPCS*.

[121] 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 *HPCA*.

[122] Jaehyun Park, Byeongho Kim, Sungmin Yun, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. 2021. TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory. In *MICRO*.

[123] peakperf. 2021. peakperf. https://github.com/Dr-Noob/peakperf.git

[124] Ali Pinar and Michael T. Heath. 1999. Improving Performance of Sparse Matrix-Vector Multiplication. In *SC*.

[125] Udo W. Pooch and Al Nieder. 1973. A Survey of Indexing Techniques for Sparse Matrices. In *ACM Comput. Surv.*

[126] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *HPCA*.

[127] Youcef Saad. 1989. Krylov Subspace Methods on Supercomputers. In *SIAM J. Sci. Stat. Comput.*

[128] 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 *MICRO*.

[129] SciPy. 2021. List-of-list Sparse Matrix.

[130] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. 2015. Automatic Selection of Sparse Matrix Representation on GPUs. In *ICS*.

[131] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. 2007. Scan Primitives for GPU Computing. In *GH*.

[132] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *MICRO*.

[133] 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 *MICRO*.

[134] Vivek Seshadri and Onur Mutlu. 2017. Simple Operations in Memory to Reduce Data Movement. In *Advances in Computers*.
[135] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. 2019. Near-Memory Computing: Past, Present, and Future. In *MICPRO*.
[136] A. Smith. 2019. 6 New Facts About Facebook. http://mediashift.org
[137] Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. 2017. Globally Homogeneous, Locally Adaptive Sparse Matrix-Vector Multiplication on the GPU. In *ICS*.
[138] stream. 2021. stream. https://github.com/jeffhammond/STREAM.git
[139] Bor-Yiing Su and Kurt Keutzer. 2012. ClSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *ICS*.
[140] Guangming Tan, Junhong Liu, and Jiajia Li. 2018. Design and Implementation of Adaptive SpMV Library for Multicore and Many-Core Architecture. In *ACM Trans. Math. Softw.*
[141] Wai Teng Tang, Ruizhe Zhao, Mian Lu, Yun Liang, Huynh Phung Huyng, Xibai Li, and Rick Siow Mong Goh. 2015. Optimizing and Auto-Tuning Scale-Free Sparse Matrix-Vector Multiplication on Intel Xeon Phi. In *CGO*.
[142] Yaman Umuroglu and Magnus Jahre. 2014. An Energy Efficient Column-Major Backend for FPGA SpMV Accelerators. In *ICCD*.
[143] UPMEM. 2018. Introduction to UPMEM PIM. Processing-in-memory (PIM) on DRAM Accelerator (White Paper).
[144] UPMEM. 2020. UPMEM Website. https://www.upmem.com
[145] UPMEM. 2021. UPMEM User Manual. Version 2021.3.
[146] R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, and B. Lee. 2002. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *SC*.
[147] Richard Wilson Vuduc and James W. Demmel. 2003. Automatic Performance Tuning of Sparse Matrix Kernels. In *PhD Thesis*.
[148] Richard W. Vuduc and Hyun-Jin Moon. 2005. Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure. In *HPCC*.
[149] J.B. White and P. Sadayappan. 1997. On Improving the Performance of Sparse Matrix-Vector Multiplication. In *HIPC*.
[150] Jeremiah Willcock and Andrew Lumsdaine. 2006. Accelerating Sparse Matrix Computations via Data Compression. In *ICS*.
[151] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *SC*.
[152] Tianji Wu, Bo Wang, Yi Shan, Feng Yan, Yu Wang, and Ningyi Xu. 2010. Efficient PageRank and SpMV Computation on AMD GPUs. In *ICPP*.
[153] 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 *HPCA*.
[154] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. YaSpMV: Yet Another SpMV Framework on GPUs. In *PPoPP*.
[155] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. YaSpMV: Yet Another SpMV Framework on GPUs. In *PPoPP*.
[156] Wangdong Yang, Kenli Li, and Keqin Li. 2017. A Hybrid Computing Method of SpMV on CPU–GPU Heterogeneous Computing Systems. In *JPDC*.
[157] Wangdong Yang, Kenli Li, Yan Liu, Lin Shi, and Lanjun Wan. 2014. Optimization of Quasi-Diagonal Matrix-Vector Multiplication on GPU. In *Int. J. High Perform. Comput. Appl.*
[158] Wangdong Yang, Kenli Li, Zeyao Mo, and Keqin Li. 2015. Performance Optimization Using Partitioned SpMV on GPUs and Multicore CPUs. In *IEEE Transactions on Computers*.
[159] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An Accelerator for Sparse Neural Networks. In *MICRO*.
[160] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the Gap between Deep Learning and Sparse Matrix Format Selection. In *PPoPP*.
[161] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the Gap between Deep Learning and Sparse Matrix Format Selection. In *PPoPP*.
[162] Yue Zhao, Weijie Zhou, Xipeng Shen, and Graham Yiu. 2018. Overhead-Conscious Format Selection for SpMV-Based Applications. In *IPDPS*.
[163] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *MICRO*.

[164] 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 *HPEC*.
[165] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *MICRO*.

APPENDIX

A Extended Results

A.1 Synchronization Approaches in Block-Based Formats

We compare the coarse-grained locking (*lb-cg*) and the fine-grained locking (*lb-fg*) approaches in BCOO format. Figure 30 shows the performance achieved by BCOO format for all the data types when balancing the blocks or the non-zero elements across 16 tasklets of one DPU. We evaluate all small matrices of Table 3, i.e., delaunay_n13 (**D**), wing_nodal (**W**), raefsky4 (**R**) and pkustk08 (**P**) matrices.

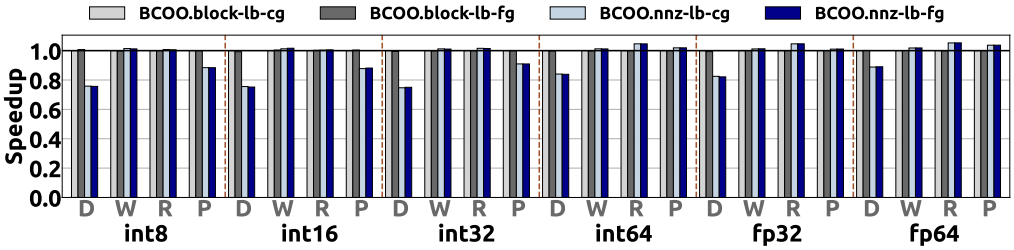


Fig. 30. Performance of BCOO format for all the data types and small matrices using 16 tasklets of one DPU.

Our key finding is that the fine-grained locking approach performs similarly with the coarse-grained locking approach. The fine-grained locking approach does not increase parallelism in the UPMEM PIM architecture, since memory accesses to the local DRAM bank are serialized in the DMA engine of the DPU. The same key finding holds independently of the compressed matrix format used.

A.2 Fine-Grained Data Transfers in 2D Partitioning Techniques

Figures 31 and 32 compare coarse-grained data transfers (i.e., performing parallel data transfers to all 2048 DPUs at once, padding with empty bytes at the granularity of 2048 DPUs) with fine-grained data transfers (i.e., iterating over the ranks and for each rank performing parallel data transfers to the 64 DPUs of the same rank, padding with empty bytes at the granularity of 64 DPUs) for all matrices of our large matrix dataset in *equally-wide* and *variable-sized* schemes, respectively. The reported key findings of Figure 16 (Section 6.2.1) apply to all matrices with diverse sparsity patterns.

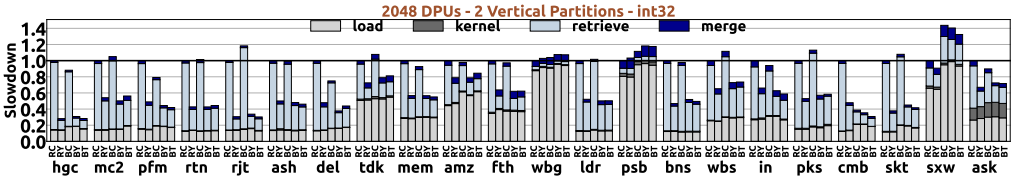


Fig. 31. Performance comparison of RC: RBDC00 with coarse-grained transfers, RY: RBDC00 with fine-grained transfers in the output vector, BC: BDC00 with coarse-grained transfers, BY: BDC00 with fine-grained transfers only in the output vector, and BT: BDC00 with fine-grained transfers in both the input and the output vector using the int32 data type, 2048 DPUs and having 2 vertical partitions. Performance is normalized to that of the RC scheme.

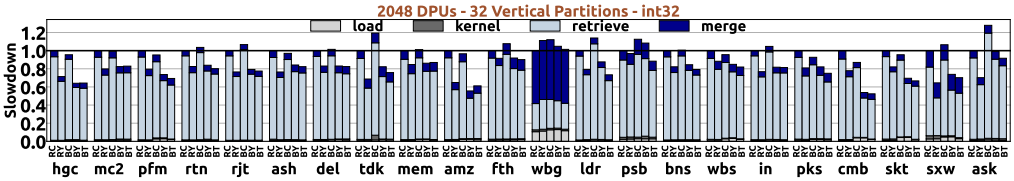


Fig. 32. Performance comparison of RC: RBDC00 with coarse-grained transfers, RY: RBDC00 with fine-grained transfers in the output vector, BC: BDC00 with coarse-grained transfers, BY: BDC00 with fine-grained transfers only in the output vector, and BT: BDC00 with fine-grained transfers in both the input and the output vector using the int32 data type, 2048 DPUs and having 32 vertical partitions. Performance is normalized to that of the RC scheme.

A.3 Performance of Compressed Formats Using 2D Partitioning Techniques

Figures 33, 34, 35 compare the performance achieved by various compressed matrix formats for each of the three types of the 2D partitioning technique for all matrices of our large matrix dataset. The reported key findings explained in Section 6.2.2 apply to all matrices with diverse sparsity patterns.

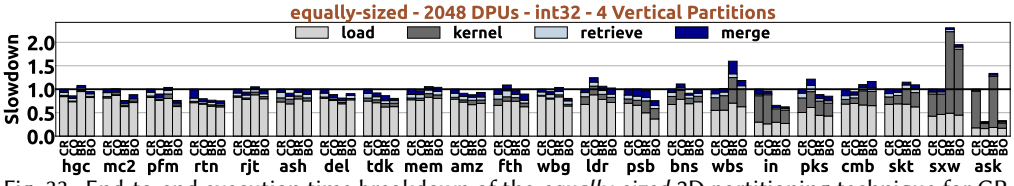


Fig. 33. End-to-end execution time breakdown of the *equally-sized* 2D partitioning technique for CR: DCSR, CO: DC00, BR: DBCSR and BO: DBC00 schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of DCSR.

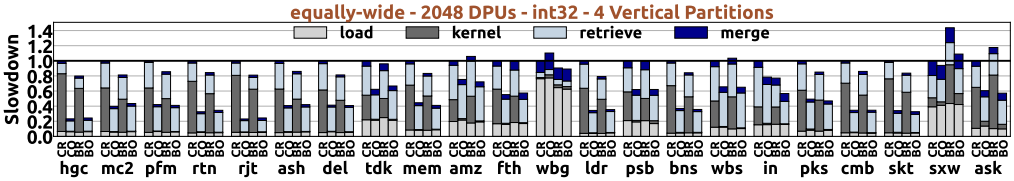


Fig. 34. End-to-end execution time breakdown of the *equally-wide* 2D partitioning technique for CR: RBDCSR, CO: RBDC00, BR: RBDBCSR and BO: RBDBC00 schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of RBDCSR.

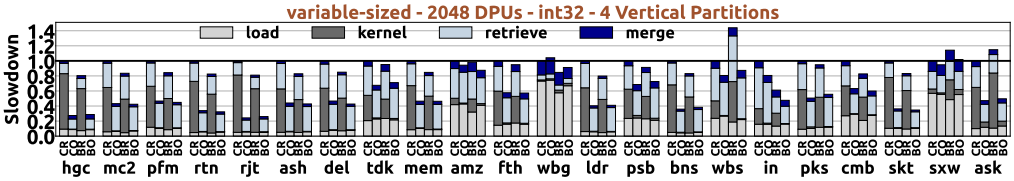


Fig. 35. End-to-end execution time breakdown of the *variable-sized* 2D partitioning technique for CR: BDC00, CO: BDC00, BR: BDBCSR and BO: BDBC00 schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of BDCSR.

B Arithmetic Throughput of One DPU

We evaluate the arithmetic throughput of the DPU for the multiplication (MUL) operation. We use the arithmetic throughput microbenchmark of PrIM [45, 53] and configure it for the all data types. Figure 36 shows the measured arithmetic throughput (in MOperations per second) for the MUL operation varying the number of tasklets of one DPU for all the data types. The

arithmetic throughput for MUL operation is 12.94 MOps, 10.52 MOps, 8.86 MOps, 2.38 MOps, 1.84 MOps, and 0.51 MOps for the int8, int16, int32, int64, fp32 and fp64 data types, respectively.

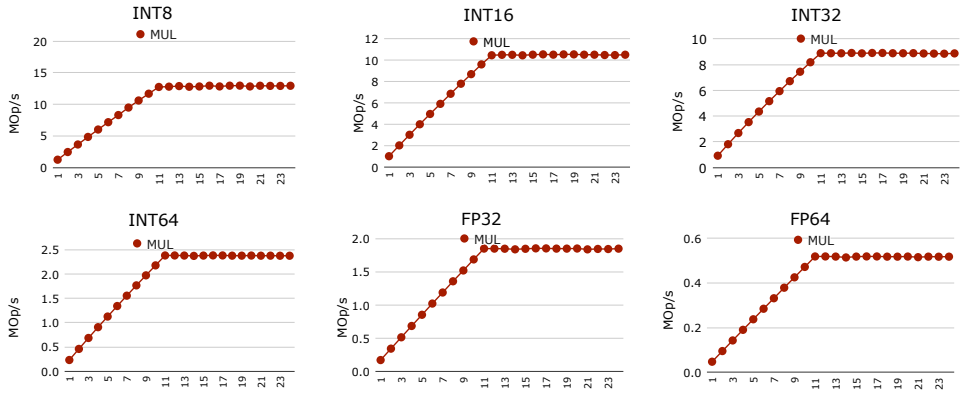


Fig. 36. Throughput of the MUL operation on one DPU for all the data types.

C *SparseP* Software Package

Table 6 summarizes the SpMV PIM kernels provided by *SparseP* library. All kernels support a wide range of data types, i.e., 8-bit integer, 16-bit integer, 32-bit integer, 64-bit integer, 32-bit float, and 64-bit float data types.

Partitioning Technique	Compressed Format	Balancing Among PIM Cores	Balancing Among Threads	Synchronization Approach
1D	CSR	rows	rows, nnz*	-
		nnz*	rows, nnz*	-
	COO	rows	rows, nnz*	-
		nnz*	rows, nnz*	-
		nnz	nnz	lb-cg / lb-fg / lf
	BCSR	blocks†	blocks†, nnz†	lb-cg‡ / lb-fg‡
		nnz†	blocks†, nnz†	lb-cg‡ / lb-fg‡
	BCOO	blocks	blocks, nnz	lb-cg / lb-fg / lf
		nnz	blocks, nnz	lb-cg / lb-fg / lf
2D *equally-sized*	CSR	-	rows, nnz*	-
	COO	-	nnz	lb-cg / lb-fg / lf
	BCSR	-	blocks†, nnz†	lb-cg‡ / lb-fg‡
	BCOO	-	blocks, nnz	lb-cg / lb-fg
2D *equally-wide*	CSR	nnz*	rows, nnz*	-
	COO	nnz	nnz	lb-cg / lb-fg / lf
	BCSR	blocks†	blocks†, nnz†	lb-cg‡ / lb-fg‡
		nnz†	blocks†, nnz†	lb-cg‡ / lb-fg‡
	BCOO	blocks	blocks, nnz	lb-cg / lb-fg
		nnz	blocks, nnz	lb-cg / lb-fg
2D *variable-sized*	CSR	nnz*	rows, nnz*	-
	COO	nnz	nnz	lb-cg / lb-fg / lf
	BCSR	blocks†	blocks†, nnz†	lb-cg‡ / lb-fg‡
		nnz†	blocks†, nnz†	lb-cg‡ / lb-fg‡
	BCOO	blocks	blocks, nnz	lb-cg / lb-fg
		nnz	blocks, nnz	lb-cg / lb-fg

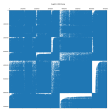
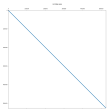
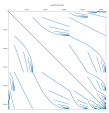
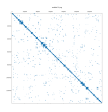
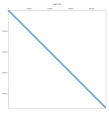
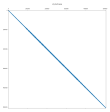
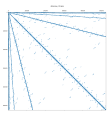
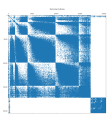
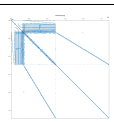
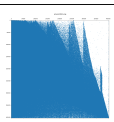
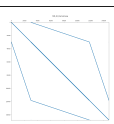
Table 6. *SparseP* library. *: row-granularity, †: block-row-granularity, ‡: (only for 8-bit integer and small block sizes)

D Large Matrix Dataset

We present the characteristics of the sparse matrices of our large matrix data set. Table 7 presents the sparsity of the matrix (i.e., NNZ / (rows x columns)), the standard deviation of non-zero elements among rows (NNZ-r-std) and columns (NNZ-c-std). Table 8 visualizes the sparsity patterns of each sparse matrix of our large matrix data set.

Matrix Name	Rows x Columns	NNZs	Sparsity	NNZ-r-std	NNZ-c-std
hugetric-00020	7122792 x 7122792	21361554	4.21e-07	0.031	0.031
mc2depi	525825 x 525825	2100225	7.59e-06	0.076	0.076
parabolic_fem	525825 x 525825	3674625	1.33e-05	0.153	0.153
roadNet-TX	1393383 x 1393383	3843320	1.98e-06	1.037	1.037
rajat31	4690002 x 4690002	20316253	9.24e-07	1.106	1.106
af_shell1	504855 x 504855	17588875	6.90e-05	1.275	1.275
delaunay_n19	524288 x 524288	3145646	1.14e-05	1.338	1.338
thermomech_dK	204316 x 204316	2846228	6.81e-05	1.431	1.431
memchip	2707524 x 2707524	14810202	2.02e-06	2.062	1.173
amazon0601	403394 x 403394	3387388	2.08e-05	2.79	15.29
FEM_3D_thermal2	147900 x 147900	3489300	1.59e-04	4.481	4.481
web-Google	916428 x 916428	5105039	6.08e-06	6.557	38.366
ldoor	952203 x 952203	46522475	5.13e-05	11.951	11.951
poisson3Db	85623 x 85623	2374949	3.24e-04	14.712	14.712
boneS10	914898 x 914898	55468422	6.63e-05	20.374	20.374
webbase-1M	1000005 x 1000005	3105536	3.106e-06	25.345	36.890
in-2004	1382908 x 1382908	16917053	8.846e-06	37.230	144.062
pkustk14	151926 x 151926	14836504	6.428e-04	46.508	46.508
com-Youtube	1134890 x 1134890	5975248	4.639e-06	50.754	50.754
as-Skitter	1696415 x 1696415	22190596	7.71e-06	136.861	136.861
sx-stackoverflow	2601977 x 2601977	36233450	5.352e-06	137.849	65.367
ASIC_680	682862 x 682862	3871773	8.303e-06	659.807	659.807

Table 7. Large Matrix Dataset. Matrices are sorted by NNZ-r-std, i.e., based on their irregular pattern.

Matrix Name	Plot
hugetric-00020	
mc2depi	
parabolic_fem	
roadNet-TX	
rajat31	
af_shell1	
delaunay_n19	
thermomech_dK	
memchip	
amazon0601	
FEM_3D_thermal2	

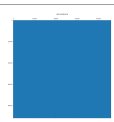
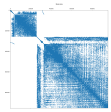
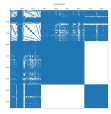
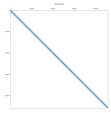
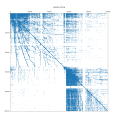
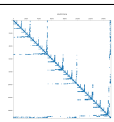
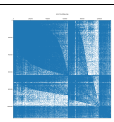
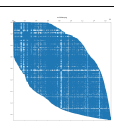
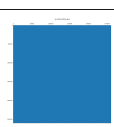
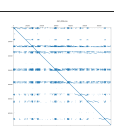
Matrix Name	Plot
web-Google	
ldoor	
poisson3Db	
boneS10	
webbase-1M	
in-2004	
pkustk14	
com-Youtube	
as-Skitter	
sx-stackoverflow	
ASIC_680	

Table 8. Sparsity patterns of the sparse matrices of our large matrix data set.