ALPHA-PIM: Analysis of Linear Algebraic Processing for High-Performance Graph Applications on a Real Processing-In-Memory System

Marzieh Barkhordar
Simon Fraser University
Burnaby, BC, Canada
mba186@sfu.ca

Alireza Tabatabaeian
Simon Fraser University
Burnaby, BC, Canada
ata175@sfu.ca

Mohammad Sadrosadati
ETH Zürich
Zürich, Switzerland
m.sadr89@gmail.com

Christina Giannoula
University of Toronto & Vector Institute
Toronto, ON, Canada
christina.giann@gmail.com

Juan Gomez Luna
NVIDIA
Zürich, Switzerland
jgomezluna@nvidia.com

Izzat El Hajj
American University of Beirut
Beirut, Lebanon
izzat.elhajj@aub.edu.lb

Onur Mutlu
ETH Zürich
Zürich, Switzerland
omutlu@gmail.com

Alaa R. Alameldeen
Simon Fraser University
Burnaby, BC, Canada
alaa@cs.sfu.ca

***Abstract*—Processing large-scale graph datasets is computationally intensive and time-consuming. Processor-centric CPU and GPU architectures, commonly used for graph applications, often face bottlenecks caused by extensive data movement between the processor and memory units due to low data reuse. As a result, these applications are often memory-bound, limiting both performance and energy efficiency due to excessive data transfers. Processing-In-Memory (PIM) offers a promising approach to mitigate data movement bottlenecks by integrating computation directly within or near memory. Although several previous studies have introduced custom PIM proposals for graph processing, they do not leverage real-world PIM systems.**

This work aims to explore the capabilities and characteristics of common graph algorithms on a real-world PIM system to accelerate data-intensive graph workloads. To this end, we (1) implement representative graph algorithms on UPMEM's general-purpose PIM architecture; (2) characterize their performance and identify key bottlenecks; (3) compare results against CPU and GPU baselines; and (4) derive insights to guide future PIM hardware design.

Our study underscores the importance of selecting optimal data partitioning strategies across PIM cores to maximize performance. Additionally, we identify critical hardware limitations in current PIM architectures and emphasize the need for future enhancements across computation, memory, and communication subsystems. Key opportunities for improvement include increasing instruction-level parallelism, developing improved DMA engines with non-blocking capabilities, and enabling direct interconnection networks among PIM cores to reduce data transfer overheads.

***Index Terms*—Graph Processing, Processing-In-Memory, Near-data Processing, Workload Characterization, Linear Algebra**

I. Introduction

Graph data structures are widely used to model complex problems across domains due to their simplicity and versatility [1]–[6]. As real-world graphs scale to billions of nodes and edges, parallel computing systems and algorithms have been developed to improve performance. However, efficient parallel graph processing remains challenging due to irregular memory access, load imbalance, and low arithmetic intensity [7], [8].

Linear algebra is a promising paradigm for modeling graph applications using fundamental matrix operations [7]. It offers a concise set of primitives that, combined with algebraic semirings, efficiently represent diverse graph algorithms [7], [9]–[14]. Graph algorithms are often more compact and easier to express using sparse-matrix linear algebra. Describing an algorithm in this format eliminates the need for additional data structures and can be directly implemented in array-based programming environments, making it easier to optimize.

Given the sparsity of adjacency matrices in graph datasets, matrices are stored in a compressed format. Consequently, sparse matrix operations, such as sparse matrix-vector multiplication (SpMV) are used to implement graph applications. However, these operations involve indirect memory references due to the compressed storage format and irregular memory accesses to the input vector caused by sparsity, which hinders their ability to achieve peak performance in processor-centric CPU and GPU architectures. Many real-world graph datasets across various application domains are highly sparse, making

We acknowledge the generous gifts from our industrial partners, including Intel, Ampere Computing, Google, Huawei, and Microsoft. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) under RGPIN-2021-03729 and RGPIN-2018-06795; an SFU Faculty Recruitment Grant; the Semiconductor Research Corporation (SRC), the ETH Future Computing Laboratory (EFCL), AI Chip Center for Emerging Smart Systems (ACCESS), sponsored by InnoHK funding, Hong Kong SAR, and European Union's Horizon programme for research and innovation [101047160 - BioPIM].

SpMV a memory-bound kernel in processor-centric systems that is limited by data movement between memory and processors [15]–[23].

Processing-In-Memory (PIM) offers a promising solution to the data movement bottleneck by moving computation closer to the data, and integrating processing capabilities within memory chips [24]–[32]. Several prior PIM proposals aim to accelerate graph processing [21], [31], [33]–[40]. However, none provides a comprehensive evaluation of general-purpose real-world PIM systems. To our knowledge, this is the first paper to comprehensively study the opportunities and limitations of linear algebraic graph processing using a real PIM system.

We analyze the execution time breakdown of SpMV implementations from SparseP [41], the first open-source SpMV library specifically designed for real PIM systems. Prior work indicates that traversal-based graph applications often require multiple iterations of matrix-vector multiplication, with highly-sparse input vectors in several iterations [42]–[45]. We observe that SpMV kernels suffer from high data transfer costs due to using a dense input vector format. To address this issue, we further minimize these data transfer costs through sparse-matrix sparse-vector multiplication (SpMSpV) [46], [47]. Using a compressed format for the input vector in SpMSpV reduces the significant overhead associated with loading input vectors into DRAM banks.

Our goal is to understand the capabilities and characteristics of a real-world PIM system for accelerating graph applications. To do so, we introduce *ALPHA-PIM*, the first linear algebraic graph application framework designed for a real-world PIM architecture. We choose UPMEM's PIM system [48] for our study because it is the first commercially available general-purpose PIM architecture, combining conventional 2D DRAM arrays with general-purpose processing cores, known as DRAM Processing Units (DPUs), on the same chip. We design and evaluate *ALPHA-PIM* in five key steps. First, we implement and evaluate different versions of the SpMSpV kernel using various compressed matrix formats (CSR, CSC, COO) and partitioning strategies (row-wise, column-wise, 2D) to identify the most efficient approach for graph applications. Second, we implement three well-known linear algebraic graph applications: Breadth-First Search (BFS) [49], [50], Single Source Shortest Path (SSSP) [51]–[53], and Personalized PageRank (PPR) [54], using both SpMV and SpMSpV kernels. Third, we characterize performance of these graph applications and identify key bottlenecks. Fourth, we compare our graph implementation against state-of-the-art graph processing frameworks on both CPU and GPU platforms. Finally, we explore implications for future PIM hardware enhancements, emphasizing the importance of algorithm–hardware co-design to reduce communication costs and improve the performance of linear-algebraic graph processing on a real-world PIM system.

Our results highlight two major observations. First, it is critically important to select appropriate partitioning strategies and compressed matrix formats to maximize performance on PIM systems. In our experiments, we observed up to a 25× difference in SpMSpV's execution time between the best and worst-performing strategies, underscoring the importance of careful strategy and format selection for maximizing performance. Second, the performance of these applications is mainly limited due to (i) computation-side bottlenecks, e.g., idle DPU cycles caused by structural hazards in the revolver pipeline; (ii) memory-side bottlenecks, e.g., idle DPU cycles due to waiting on memory operations; and (iii) communication-side bottlenecks from loading and retrieving vectors between iterations, compounded by the lack of inter-DPU communication. Based on these observations, we suggest several hardware optimizations to enhance the performance of graph processing on a real PIM architecture.

In this paper, we make the following contributions:

- We present the first linear-algebraic implementation of several graph applications on a real-world PIM system (UPMEM), and conduct a thorough design-space exploration of data structures used to represent graph datasets and input vectors.
- We implement and conduct the first comprehensive study of the widely used SpMSpV kernel for graph applications on a state-of-the-art real-world PIM system.
- We compare the performance of graph implementations on the state-of-the-art UPMEM PIM system vs. conventional CPU and GPU systems. We show that *ALPHA-PIM* achieves kernel speedups of 10.2× / 48.8× / 3.6× and total execution speedups of 2.6× / 10.4× / 1.7× for BFS / SSSP / PPR, respectively, over a conventional CPU baseline, while also delivering better compute utilization than both CPU and GPU systems.

II. Background and Motivation

A. Graph Applications Using Linear Algebra

Graph algorithms can be expressed as linear algebra operations due to the duality between a graph and its adjacency matrix [7]. For a graph $G = (V, E)$ with N vertices, its $N \times N$ adjacency matrix A has $A_{ij} = 1$ if v_i is adjacent to v_j, and 0 otherwise. This enables algorithms like BFS to be implemented as iterative matrix-vector multiplications, e.g., $v = A^T v$, where non-zero elements in v indicate visited nodes. To support a broader range of algorithms, this framework can be extended using a *semiring* [7], [9]–[14], which generalizes addition and multiplication to $\oplus$ and $\otimes$ with algebraic properties. For instance, SSSP can be expressed using the $(\min, +)$ semiring over extended real numbers.

Real-world graphs are typically sparse, making efficient matrix storage essential. Three widely used compressed formats are Coordinate List (COO), Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC) [7], [55]–[57]. COO stores non-zero elements as (i, j, value) tuples; while simple and parallel-friendly, it lacks row-wise grouping, leading to scattered updates that require atomics and reduce efficiency for large matrices. CSR compresses row indices using three

arrays: *values*, *col_indices*, and *row_ptr*, supporting efficient row-wise operations. Conversely, CSC compresses columns via *values*, *row_indices*, and *col_ptr*, making it more suitable for column-wise operations like vector-matrix multiplications.

B. Linear Algebraic Graph Applications in Processor-Centric Systems

Linear algebra provides an effective framework for modeling graph applications via matrix operations, simplifying parallel algorithm implementation [7]. By combining linear-algebraic primitives with algebraic semirings, diverse graph algorithms can be efficiently expressed. For example, in BFS, replacing addition/multiplication with OR/AND reduces computation and allows early termination, improving runtime. These primitives also enable 2D matrix partitioning—a critical capability for scaling to large graphs that many other frameworks lack [42]. As a result, several linear-algebraic graph frameworks have been developed for CPUs and GPUs to leverage these advantages [42], [58]–[60].

Despite their advantages, linear-algebraic graph frameworks on CPUs and GPUs face performance challenges due to the sparsity of real-world graphs, which necessitates compressed formats and sparse matrix operations like SpMV and SpMM. Studies show these operations often fall short of peak performance [15]–[23], primarily due to algorithmic characteristics, storage formats, and sparsity patterns. SpMV is especially memory-bound: its irregular access pattern hinders cache locality; each matrix element is accessed only once, i.e., no temporal locality; and auxiliary structures for non-zero indexing add bandwidth pressure and increase contention.

In contrast, PIM systems are well suited to address the challenges of linear-algebraic graph processing. By moving computation closer to memory and reducing data movement, PIM systems can exploit high levels of parallelism and provide substantial memory bandwidth [24]–[30], which significantly improve performance of memory-bound operations like SpMV. Several prior PIM proposals have aimed to accelerate graph processing [21], [33]–[40]. However, none have comprehensively evaluated real general-purpose PIM architectures. This motivates our exploration of the capabilities and characteristics of popular graph algorithms on UPMEM's PIM architecture.

C. The UPMEM PIM Architecture

1) ***System Organization:*** Figure 1 illustrates the UPMEM architecture [48], [61]. A UPMEM system consists of a host CPU (e.g., x86, ARM64, or 64-bit RISC-V), standard DRAM modules, and PIM-enabled memory modules designed in the DDR4-2400 DIMM form factor. These modules house multiple PIM cores, called DRAM Processing Units (DPUs), with each module comprising multiple ranks, each containing 8 DPUs. A typical configuration supports up to 2,560 DPUs across 20 double-ranked DIMMs. UPMEM DIMMs connect to the host CPU via memory channels and integrate PIM chips, each pairing a DPU core with a DRAM bank. The host CPU manages data transfers between its memory and the DPUs' Main RAM (MRAM) using a transposition library provided by the UPMEM Software Development Kit (SDK). This library ensures that data layout transformations are handled transparently, facilitating efficient data movement between main memory and PIM-enabled memory. The UPMEM SDK supports both serial and parallel data transfers. Parallel transfers can be performed across multiple MRAM banks which allows for concurrent CPU-DPU and DPU-CPU data transfers.

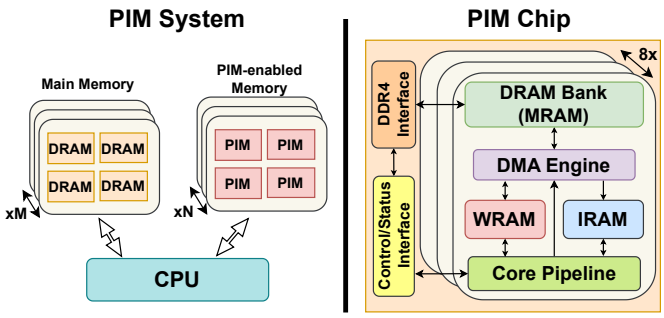

Fig. 1. High-level organization of the UPMEM PIM system with a host CPU, DRAM main memory, and PIM-enabled memory (left), and a more detailed view of the UPMEM PIM chip (right) [48].

2) ***DPU Architecture:*** Each DPU is a multithreaded, in-order 32-bit RISC core designed with a 14-stage pipeline. It features a 64 MB DRAM bank (MRAM), a 64 KB scratchpad memory (WRAM), and a 24 KB instruction memory (IRAM). The MRAM serves as the main memory, the WRAM as a high-speed temporary storage, and the IRAM stores the instructions to be executed. The DPU can concurrently run up to 24 hardware PIM threads, i.e., tasklets, which share access to the WRAM, IRAM, and MRAM. The DPU's unique "revolver pipeline" design enforces a scheduling constraint where consecutive instructions within the same thread must be dispatched 11 cycles apart. This constraint simplifies the microarchitecture by eliminating the need for complex data forwarding and pipeline interlocks. The register file within the DPU is split into even and odd sections, preventing simultaneous access to multiple registers within the same group due to structural hazards.

3) ***DPU Programming:*** Programming the UPMEM PIM system follows the single-program multiple-data (SPMD) model, where all tasklets execute the same program across multiple DPUs on distinct data partitions. Efficient execution requires careful input data partitioning to ensure workload balance. On the host side, the CPU manages DPU allocation, loads program binaries and data, and controls execution. Programs are written in UPMEM's C-like language and compiled using an LLVM-based toolchain to produce binaries for both host and DPUs. On the DPU side, tasklets access data via WRAM, requiring explicit transfers from MRAM using DMA instructions. Since computation is limited to WRAM-resident data, careful data movement is critical. Intra-DPU synchronization uses WRAM or MRAM primitives like mutexes and barriers, while inter-DPU coordination must go

through the host CPU due to the absence of direct DPU-to-DPU communication.

III. Limitations of SpMV Implementation for Graph Applications on UPMEM

Linear-algebraic graph applications often use SpMV to process sparse matrices efficiently by avoiding redundant computations on zero elements. SpMV multiplies a sparse matrix of size $M \times N$ by a dense vector of size $N \times 1$, producing an output vector of size $M \times 1$. SparseP [41], the first SpMV library for real PIM systems, evaluates SpMV using 1D and 2D partitioning. In 1D, the matrix is row-partitioned and each DPU receives the full input vector, enabling local computation with minimal merging. In 2D, matrices are tiled, transferring only relevant input vector segments, but overlapping tiles increase CPU merge overhead. SparseP's top-performing variants are COO.nnz (1D) and DCOO (2D), both using static, equal-sized COO-format tiles.

In Figure 2, we report the execution time breakdown of two top 1D/2D partitioning methods for SpMV in SparseP using 2048 DPUs and INT32 data (normalized to 1D partitioning). The total time is divided into four parts: (i) loading the input vector from the CPU to DPU DRAM (Load); (ii) kernel execution on DPUs (Kernel); (iii) retrieving results from DPUs to the CPU (Retrieve); and (iv) merging partial results on the CPU (Merge). Section V presents methodology details. We note that 1D partitioning incurs a high cost for transferring and broadcasting the input vector to each DRAM bank in the PIM core. While 2D partitioning reduces the transfer cost, it adds extra data transfer overhead for gathering and merging results from PIM memory to the host CPU.

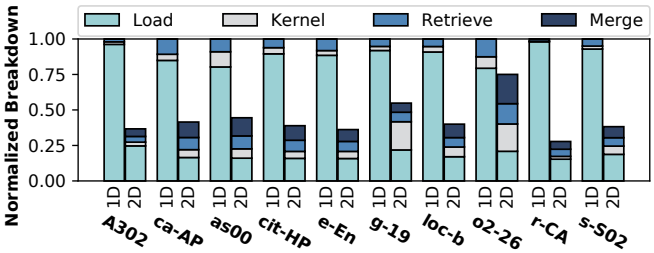

Fig. 2. Execution time breakdown using 2048 DPUs and int32 data type for 1D/2D SpMV partitioning (normalized to 1D).

SparseP addresses the high cost of input vector loads in SpMV using 2D partitioning, but its reliance on the dense format limits its effectiveness in graph applications. Previous works [42]–[45] show that in many iterations of linear-algebraic graph applications, input vectors are highly sparse—e.g., in BFS, only a small subset of vertices (the frontier) is active at each level, leaving most entries as zero. To quantify the input vector sparsity in BFS, we measured its *density* (the ratio of non-zero elements to the total number of nodes, expressed as a percentage) across several datasets. Our results indicate that, for most cases, the input vector's density remains below 50% during the first half of the iterations. Therefore, there is still potential to reduce the cost of input vector loads in linear-algebraic graph applications by exploring the potential of Sparse Matrix-Sparse Vector (SpMSpV) multiplication [46], [47].

IV. *ALPHA-PIM* Design and Optimizations

In this paper, we target graph applications with low arithmetic intensity and limited data reuse, such as BFS, SSSP, and PPR. These traversal-based algorithms rely on multiple iterations of matrix-vector multiplication, making them memory-bound and ideal candidates for PIM systems. These algorithms require the use of both SpMV and SpMSpV due to the varying sparsity of the input vector across iterations [42], [62]. While SparseP [41] provides a comprehensive analysis of SpMV, we focused on developing an efficient SpMSpV implementation for UPMEM.

Our approach involves three key steps: ① We create and evaluate various SpMSpV implementations using different compressed matrix formats and partitioning strategies to identify the most efficient option for graph applications; ② we employ an empirical cost model to determine the optimal kernel (SpMV or SpMSpV) based on the input vector's sparsity and graph dataset characteristics; and ③ we extend these implementations to support a range of traversal-based graph applications by incorporating different semirings into both SpMV and SpMSpV kernels.

A. SpMSpV Implementation on UPMEM

The SpMSpV operation, expressed as $y \leftarrow Ax$, multiplies a sparse matrix A by a sparse vector x to produce a vector y [46], [47]. Both the input matrix A and the input vector x are stored in compressed formats to efficiently manage their sparsity. Focusing on the non-zero elements of x reduces the number of computations and memory accesses, making SpMSpV suitable for large-scale, sparse datasets, unlike SpMV which assumes a dense input vector.

In this work, we implement SpMSpV using three compressed matrix formats: COO, CSR, and CSC. In SpMSpV using CSR or COO formats, the computation involves iterating through the rows of the input matrix, with the input vector in a compressed format, ensuring that only non-zero elements are processed. This requires considering the entire adjacency matrix, and matching each row element with the corresponding non-zero vector elements based on their indices. In contrast, SpMSpV using the CSC format focuses on active columns of the matrix, meaning it processes only those columns whose indices match the indices of non-zero elements in the input vector. This approach reduces the overall number of operations by excluding columns that do not align with non-zero entries in the input vector, thereby improving efficiency.

To execute SpMSpV on UPMEM, we perform four steps: ① Load the input vector into the UPMEM's DRAM banks (Load); ② run the SpMSpV kernel on DPUs (Kernel); ③ transfer results from DRAM banks to the host CPU (Retrieve); and ④ merge partial results to form the final output vector on the

host CPU (Merge). We exclude the time needed for loading the matrix into UPMEM memory from our analysis, as this step is typically amortized over multiple kernel iterations or overlapped with other computations, making it negligible in real-world graph applications. We next describe SpMSpV implementation details including partitioning, parallelism, and memory access patterns.

1) ***Partitioning Across DPUs****:* Partitioning strategies in UPMEM face different challenges than CPU/GPU due to architectural constraints. Unlike CPU/GPU, which optimize memory access via caching and coalescing, DPUs have limited MRAM and WRAM, restricting the amount of data each DPU can store and process efficiently. UPMEM lacks direct inter-DPU communication, which increases communication overhead due to relying on the host CPU for communication across DPUs. Efficient partitioning must minimize CPU-DPU transfers to reduce data movement and performance loss. According to Figure 3, we partition the matrix in three ways: Row-wise, Column-wise, and 2D. Each method is tailored to different aspects of parallel computation and data access patterns. We next discuss each of these partitioning strategies in more detail.

Row-wise Partitioning. This involves dividing the adjacency matrix into D partitions, each corresponding to a set of rows, where D is the number of DPUs utilized in the computation. Each partition is then converted into the desired graph representation format (COO, CSR, or CSC) and loaded into the MRAM of a DPU. This method requires copying the entire compressed input vector into the DRAM bank of each PIM core. Each DPU computes a portion of the output vector result, but the computation is carried out exclusively by the PIM cores, eliminating the need for a merging step. We implemented three row-wise partitioning versions for SpMSpV, utilizing three different compressed formats: CSR, COO, and CSC (referred to as CSC-R).

Column-wise Partitioning. This involves dividing the adjacency matrix into D partitions by column, where D is the number of DPUs utilized in the computation. Each partition is converted into the desired graph representation. Only a subset of compressed input vector elements is loaded into each PIM core's DRAM bank. Each PIM core generates a complete output vector of size equal to the number of nodes, representing a partial result of the final output vector, which then requires merging on the host CPU. For column-wise partitioning, we only utilize the CSC compressed format, as the other formats (CSR and COO) are inefficient for this partitioning. We refer to this version as CSC-C for brevity.

2D Partitioning. We divide the adjacency matrix into a grid of tiles matching the number of available DPUs. This method seeks to balance the trade-off between input vector transfer and output vector transfer as each DPU handles only a portion of the input and output vector stored in its DRAM bank. However, when DPUs are assigned overlapping tiles—where tiles share rows of the matrix that are divided among multiple tiles—each DPU generates numerous partial results for the output vector. These results are then transmitted to the host CPU, where they are merged to form the final output vector. In our work, the CPU cores execute the merge step in parallel using the OpenMP API. Similar to column-wise partitioning, we exclusively use the CSC compressed format for 2D partitioning (denoted as CSC-2D) due to the inefficiency of the other formats.

2) ***Thread-Level Parallelism****:* To leverage the high memory bandwidth of PIM banked DRAM, each PIM core executes multiple hardware threads. We assign one partition to each core and ensure balanced computation across its threads by evenly distributing the workload, either by row or by column depending on the matrix compression format. This thread-level workload balancing helps minimize kernel execution time.

3) ***Memory Access Patterns and Kernel Logic****:* We briefly describe how threads access data from and to their local DRAM bank during kernel execution.

First, SpMSpV uses streaming accesses for reading non-zero values and their indices. To exploit spatial locality and high bandwidth, each thread fetches large chunks of bytes in a coarse-grained fashion from DRAM into WRAM (UPMEM's scratchpad), then accesses them in a fine-grained fashion. This also applies to loading the compressed input vector in SpMSpV, where memory access patterns are more localized than in SpMV. In SpMV, input vector accesses are input-driven, determined by the column indices of the non-zero matrix elements, leading to irregular memory accesses.

Second, threads buffer partial results for the same output row in WRAM, exploiting temporal locality before writing back to DRAM. Since multiple threads may update the same output vector elements concurrently, synchronization primitives (mutexes and barriers) are used to ensure correctness and avoid race conditions.

B. Adaptive SpMSpV–SpMV Switching

Traversal-based graph algorithms such as BFS, SSSP, and PPR rely on iterative matrix-vector multiplications. A critical characteristic of these algorithms is that the sparsity of the input vector changes across iterations—typically starting sparse and becoming denser as the algorithm progresses. Since SpMSpV is more efficient when the input vector is sparse while SpMV performs better when it is dense, switching between them at the right point can significantly reduce total execution time.

We analyzed various SpMV and SpMSpV implementations to identify the most efficient options for graph applications, selecting CSC-2D for SpMSpV and DCOO (the best 2D-partitioned SpMV from SparseP) for SpMV (see Sections VI-A and III for details). Figure 4 compares per-iteration total execution time for two algorithms and datasets using two strategies: ① SpMV-only for all iterations and ② SpMSpV-only for all iterations. The x-axis shows iteration time (ms), and the y-axis shows input vector density. We observe that SpMSpV performance scales with input sparsity, while SpMV

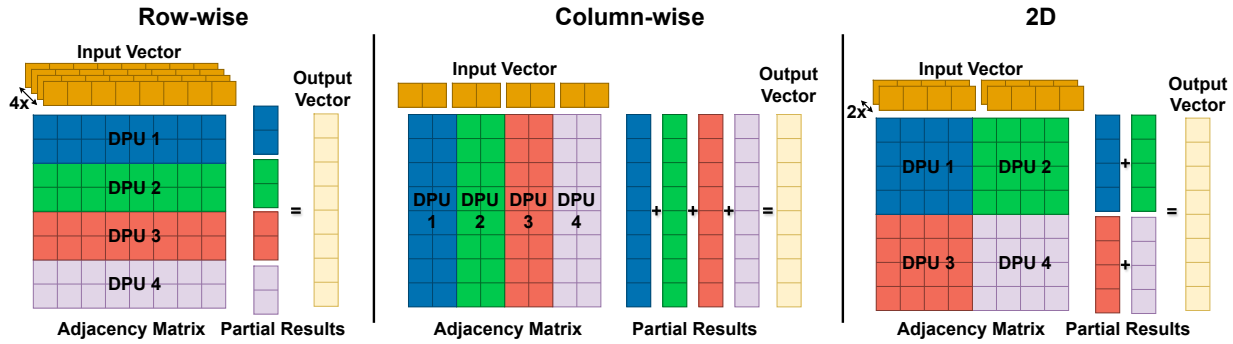

Fig. 3. Partitioning Strategies of Adjacency Matrix.

remains steady across iterations. This experiment shows how performance shifts between these kernels at different densities.

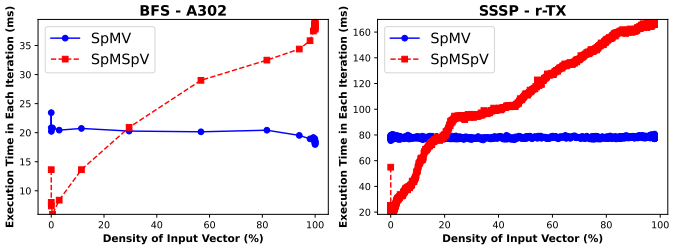

Fig. 4. Execution time per iteration on UPMEM for BFS and SSSP of two datasets using SpMV and SpMSpV.

1) ***Empirical Cost Model and Switch Point****:* We define the optimal switching point as the input vector density—the ratio of non-zero entries to the total number of nodes, expressed as a percentage—at which SpMV begins to outperform SpMSpV. This threshold varies based on the structural characteristics of the graph dataset.

To identify these switching behaviors, we analyzed a diverse set of real-world graphs and found a consistent trend: ① Regular graphs (e.g., road networks), which have low average degrees and uniform degree distributions, exhibit an optimal switching point around 20% density, and ② Scale-free graphs (e.g., web and social networks), which have skewed degree distributions and higher average degrees, exhibit a switching point around 50% density. Based on this observation, we categorize datasets into two classes—regular and scale-free—and use this classification to guide the kernel selection process.

To automate this selection at runtime, we design a lightweight decision tree model trained on a diverse set of real-world graphs. Our decision tree distinguishes between two dominant graph types—regular and scale-free—which cover most real-world graphs, enabling a practical and generalizable kernel switching mechanism. The model takes as input two graph features: the average node degree and the standard deviation of degrees. It classifies the graph into one of the two categories and selects the appropriate switching threshold. At runtime, we monitor the input vector density in each iteration. Once the density exceeds the threshold predicted by the decision tree, we transition from the SpMSpV kernel to the SpMV kernel.

We conducted a sensitivity analysis on the switching threshold and found that a 10% deviation results in less than 5% increase in total runtime on average. For example, as shown in Figure 4, using a 60% switching threshold instead of 50% for the 'A302' dataset increases the total runtime by only 2.5%.[1] This demonstrates that our adaptive model is robust to modest threshold errors.

2) ***Runtime Overhead and Practicality****:* Both the required graph statistics and the classification are computed once during pre-processing on the CPU. The model is computationally lightweight, incurs negligible runtime overhead, and runs efficiently on the CPU. This strategy enables adaptive and dataset-aware kernel selection, ensuring that the most efficient matrix-vector multiplication kernel is used throughout execution.

V. Evaluation Methodology

A. Graph Applications

We implement three widely-used graph algorithms—BFS, SSSP, and PPR—each with real-world applications [10], [63], [64]. BFS finds neighboring nodes [49], [50]; SSSP enables routing in road networks [51]–[53]; PageRank ranks web pages [65], and its personalized variant, PPR, emphasizes node importance from a specific source for recommendations and local search [54]. All rely on matrix-vector multiplication with different semirings (see Table I). A broader set is listed in [7]. These algorithms are representative and our findings generalize to similar workloads.

TABLE I
Algorithms with their semirings.

Algorithm	Semiring	Operations ($\oplus$ and $\otimes$)
BFS	$0, 1$	$\mid$ &
SSSP	$R \cup \infty$	min +
PPR	R	+ ×

[1]We note that the 'A302' has an iteration at around 30% input vector density followed by an iteration with around 60% density, so no single iteration has a 50% density. Our switching point is the first iteration after exceeding 50% density.

B. Tools

For the real machine experiments, we conduct our evaluation on an UPMEM PIM system featuring a 2-socket Intel Xeon Silver 4110 CPU [66] running at 2.10 GHz (host CPU), 128 GB of standard DDR4-2400 main memory [67], and 20 UPMEM PIM DIMMs offering 160 GB of PIM-capable memory and 2560 DPUs. We also use PIMulator, an advanced UPMEM simulator [68] to gain deeper insights and gather additional performance metrics.

C. Data Sets

We selected 65 graph datasets from GraphChallenge [69], spanning diverse domains. Table II summarizes 13 representative datasets used in our experiments, showing their names, abbreviations, edge/node counts, average degrees, degree standard deviations, and sparsity. Sparsity is defined as NNZ/N^2, where NNZ is the number of edges and N is the number of graph nodes. These cover several of our evaluations. Details on the full set are available in [70].

TABLE II
THE CHARACTERISTICS OF THE EXAMPLE 13 REAL-WORLD DATASETS

Datasets	Abbreviation	Edge	Node	AVG-Deg	Deg-std	Sparsity
amazon0302	A302	899792	262111	6.86	5.41	1.31E-05
as20000102	as00	12572	6474	3.88	24.99	3.00E-04
ca-GrQc	ca-Q	14484	5242	5.52	7.91	5.27E-04
cit-HepPh	cit-HP	420877	34546	24.36	30.87	3.53E-04
email-Enron	e-En	183831	36692	10.02	36.1	1.37E-04
facebook_combined	face	88234	4039	43.69	52.41	5.41E-03
graph500-scale18	g-18	3800348	174147	43.64	229.92	1.25E-04
loc-brightkite_edges	loc-b	214078	58228	7.35	20.35	6.31E-05
p2p-Gnutella24	p2p-24	65369	26518	4.93	5.91	9.30E-05
roadNet-TX	r-TX	1541898	1088092	2.78	1.0	1.01E-06
soc-Slashdot0902	s-S02	504230	82168	12.27	41.07	7.47E-05
soc-Slashdot0811	s-S11	469180	77360	12.12	40.45	7.84E-05
flickrEdges	flk-E	2316948	105938	43.74	115.58	2.06E-04

VI. Performance Evaluation

A. Trade-Offs in SpMSpV Strategies

To understand the advantages and disadvantages of SpMSpV implementations based on input vector density, we compare various versions across several real-world datasets from different domains. Figure 5 compares four SpMSpV implementations: COO, CSC-R, CSC-C, and CSC-2D. The execution times are broken down by Load, Kernel, Retrieve, and Merge phases for input vector densities of 1%, 10%, and 50%. The x-axis represents the selected datasets and different implementation types, while the y-axis shows the normalized execution times relative to the COO version for each dataset. For clarity, we display only a few representative datasets that show similar trends, as well as the normalized geometric mean of all implementations across all datasets. We omit density results between 1% and 10% and results between 10% and 50% as we observed very similar patterns. Additionally, we observed that CSR consistently performs worse than other SpMSpVs across various datasets and input vector densities. So, we exclude CSR as its performance was, on average, 2.8×, 12.68×, and 25.23× slower than the other versions at input vector densities of 1%, 10%, and 50%, respectively. SpMSpV performance varies depending on dataset features and input vector density. In particular, we make four key observations.

► First, ***CSC-2D*** *outperforms other SpMSpV versions across various datasets at higher input vector densities* due to lower DPU kernel times and partitioning both input and output vectors. The former will be further analyzed in Section VI-D. The latter reduces input vector load times compared to CSC-R and accelerates output vector retrieval compared to CSC-C. For example, in the 'face' dataset at 50% density, CSC-2D reduces input load by 7.3× vs. CSC-R and output storage by 17.5× vs. CSC-C.

► Second, ***CSC-C*** *outperforms other versions for datasets like 'r-PA' across input vector densities*, especially due to lower retrieve times. The overall execution time in CSC-C is mainly determined by the time taken to retrieve the output vector, which is larger than in other CSC-based versions where each DPU transfers only a small portion of the output to the CPU. The 'r-PA' has small, uniform node degrees, so each DPU produces a small amount of output, which is efficiently compressed within the DPU, resulting in lower overall execution time for CSC-C compared to other formats.

► Third, *at densities <10%, CSC-2D is not always optimal.* For instance, **CSC-R** outperforms other versions for the 'g-18' dataset when the input vector density is below 10%. COO and CSC-R, which avoid output merging, can be more efficient than CSC-2D in such cases.

► Fourth, ***COO*** *generally underperforms* due to processing the full adjacency matrix and transferring the entire input vector, but remains competitive in select cases.

Summary: CSC format outperforms COO and CSR, but the optimal partitioning strategy depends on the input vector density and dataset characteristics.

B. Effectiveness of Kernel Design

1) ***SpMSpV vs. SpMV****:* We evaluate how effectively we reduce the time associated with input vector loading using SpMSpV. Figure 6 compares the best-performing SpMV from SparseP with the most efficient SpMSpV version (CSC-2D) at input vector densities of 1%, 10%, 30%, and 50%. The execution times are categorized into Load, Kernel, Retrieve, and Merge phases. The y-axis displays normalized execution times relative to SpMV for each dataset. We report the normalized geometric mean of different implementations across all datasets in the last grouped bar. We make two main observations: ① *SpMSpV significantly lowers input vector load time across all densities compared to SpMV*, with the largest gains below 30%. Even at 50%, it remains more efficient, and ② *SpMSpV consistently outperforms SpMV version across most densities, and matches SpMV's performance at 50% density*, showing robust effectiveness.

2) ***End-to-End Comparison:*** **ALPHA-PIM** ***vs. SparseP****:* We compare the end-to-end performance of *ALPHA-PIM* with the best-performing SpMV from SparseP, covering the full

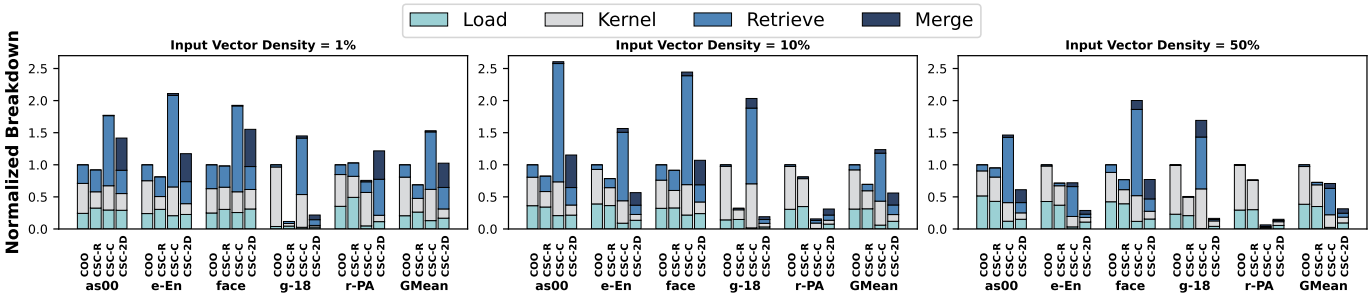

Fig. 5. Execution time breakdown for SpMSpV variations using 2048 DPUs at input vector densities of 1%, 10%, and 50% normalized to COO.

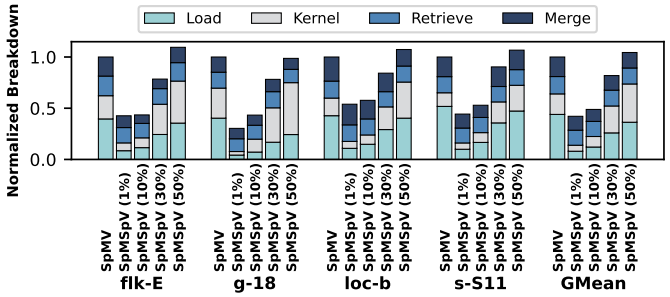

Fig. 6. Comparison of execution time breakdown between the best-performing SpMV and SpMSpV at input vector densities of 1%, 10%, 30%, and 50% using 2048 DPUs normalized to SpMV.

execution time per iteration, including data transfers, kernel computation, and result merging. While our earlier kernel-level comparison (Section VI-B1) focuses on single-iteration execution, this evaluation captures the complete multi-iteration behavior of graph applications.

Figure 7 reports the normalized execution time of *ALPHA-PIM* (with adaptive kernel switching) against SparseP's SpMV baseline across three algorithms. *ALPHA-PIM* achieves an average speedup of 1.72× for BFS, 1.34× for SSSP, and 1.22× for PPR, highlighting the practical benefit of adaptive switching over using SpMV exclusively.

C. System-Level Comparison

1) ***Impact of DPU Scaling on*** **ALPHA-PIM** ***Performance:*** Figure 8 shows the execution time breakdown of three graph algorithms—BFS, SSSP, and PPR—when varying the number of DPUs (512, 1024, 2048), normalized to 512. The execution times are broken down by Load, Kernel, Retrieve, and Merge phases. We note that for these algorithms, each iteration includes a convergence check to determine if the algorithm should terminate. Therefore, we include the time spent on convergence checks in the merge time. The y-axis displays normalized execution times relative to 512 DPUs for each dataset. We also report the normalized geometric mean of different implementations across all datasets in the last grouped bar. We make three main observations.

► First, *in BFS and SSSP, much of the execution time goes to loading the input vector and retrieving the output,* due to their iterative nature. Each iteration involves a matrix-vector multiplication, where the input vector is the output vector from the previous iteration. Without inter-DPU communication, each DPU computes only a portion of the output, which must be sent back to the CPU, updated, and reloaded into the DPUs' MRAM for the next iteration.

► Second, in contrast, *PPR's execution is kernel-dominated.* PPR requires numerous floating-point operations, particularly multiplications, which are slow on UPMEM DPUs as they rely on software-emulated multiplication instead of faster hardware-based implementations.

► Third, *using 2048 DPUs increases input vector load time, limiting speedup over 1024 DPUs* due to higher data transfer overhead. However, PPR benefits from more DPUs as its heavy kernel workload parallelizes well.

Recommendations: The lack of inter-DPU communication leads to substantial vector transfer overhead between iterations, which could be mitigated by enabling direct interconnections. In algorithms like PPR, where kernel time dominates due to extensive floating-point operations, improved hardware/software support for floating-point is essential for better performance.

2) ***UPMEM vs. CPU and GPU****:* We evaluate the UPMEM PIM system (Section V-B) against an Intel CPU [71] and an NVIDIA RTX 3050 GPU [72], focusing on latency, energy consumption, and compute utilization—defined as the achieved operations per second as a percentage of peak theoretical throughput. We report the compute utilization metric to evaluate the portion of the machine's peak performance achieved on these applications in all systems.

This evaluation aims to assess the potential of UPMEM PIM as a general-purpose accelerator. Detailed micro-architectural specifications of the CPU and GPU are provided in Table III. Peak performance for the CPU and GPU systems was calculated using peakperf [73], while the peak performance of the UPMEM PIM was determined using the method proposed in [41]. The peak performance for the GPU, CPU, and UPMEM systems are 9.1 TFLOPS, 647.25 GFLOPS, and 4.66 GFLOPS, respectively.

For both CPU and GPU, we report results related to kernel execution without including data transfers between

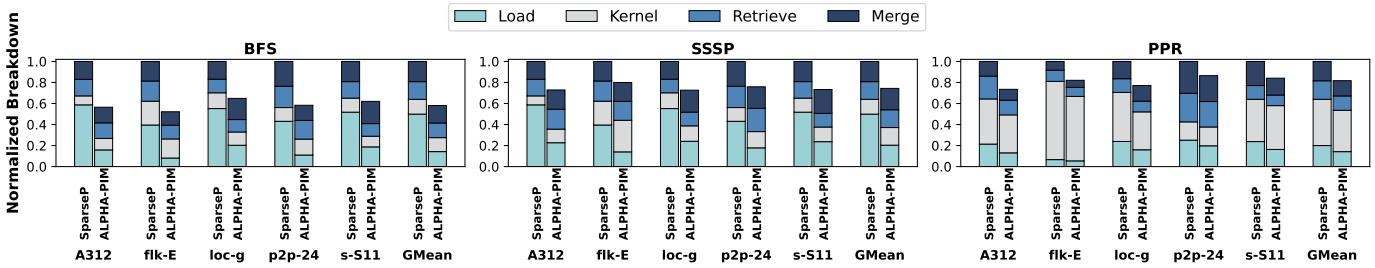

Fig. 7. Performance Comparison: ALPHA-PIM vs. SparseP SpMV

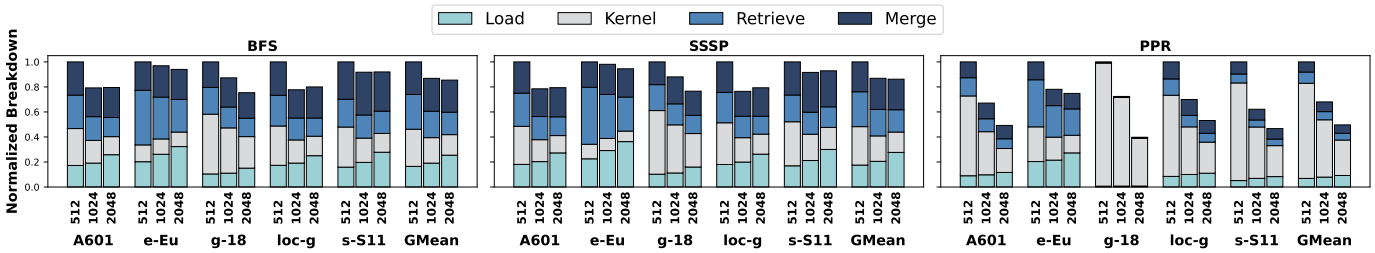

Fig. 8. Execution time breakdown for BFS, SSSP, and PPR algorithms across varying DPU counts (512, 1024, 2048), normalized to 512 DPUs.

the host and GPU. In the case of UPMEM-based PIM, results are presented in two ways: first, reflecting kernel execution alone, and second, the total execution, which includes both DPU execution and inter-DPU synchronization. Energy consumption is measured using Intel RAPL [74] for the CPU, and NVIDIA SMI [75] for the GPU. For the UPMEM PIM system, energy is obtained from the DIMMs connected to the memory controllers via x86 sockets [76].

TABLE III
CPU and GPU Micro-architectural Specs.

System	Total Cores	Frequency	Capacity	Bandwidth
Intel i7-1265U	10 (12 threads)	1.8 GHz	64GB	83.2 GB/s
NVIDIA RTX 3050	2560 CUDA cores	1.55 GHz	8GB	224 GB/s

We utilize the GridGraph library [77] for CPU-based processing and employ cuGraph [78] from the RAPIDS suite [79] on the GPU, which incorporates algorithms from NVIDIA's widely-used nvGRAPH library [60] for efficient graph analytics. Table IV presents the execution time (in milliseconds), energy consumption (in joules), and compute utilization (as a percentage) for three algorithms across all iterations on selected datasets for each system. We derive three key observations.

▶ First, *ALPHA-PIM achieves an average speedup of 10.2× (kernel execution time) and 2.6× (total time) for BFS, 48.8× and 10.4× for SSSP, and 3.6× and 1.7× for PPR over a conventional CPU baseline.*

▶ Second, *ALPHA-PIM shows much higher compute utilization with UPMEM averaging: 18.5% (kernel) / 5% (total) for BFS, 31.8% / 7.1% for SSSP, and 6.4% / 2.8% for PPR*, compared to only 0.01–0.05% on CPU/GPU. This suggests that the PIM system is better balanced between memory and compute resources for SpMV/SpMSpV operations. While CPUs and GPUs often face limitations due to memory bandwidth, PIM systems are more constrained by their computing capabilities, highlighting an opportunity for further improvements.

▶ Third, *the GPU outperforms CPU and UPMEM in execution time and energy efficiency*, owing to its mature, highly optimized parallel architecture. UPMEM, by contrast, faces two major architectural challenges: ① the lack of direct inter-DPU communication, which requires repeated CPU-mediated data transfers—especially costly in iterative graph workloads where input and output vectors are exchanged in every iteration (see Section VI-C1); and ② a distributed DRAM architecture without global memory sharing, which prevents data reuse across DPUs and increases memory movement overhead. Additionally, as we will analyze in Section VI-D, UPMEM suffers from underutilized DPU kernel execution, with low IPC and idle cycles caused by structural hazards and synchronization overheads. While PIM shows great promise for near-data processing, these bottlenecks currently limit its competitiveness. Our goal is to deeply characterize these issues and inform future PIM hardware and software optimizations.

D. *In-Depth Kernel Bottleneck Analysis in* ALPHA-PIM

We previously observed that GPUs outperform UPMEM due to their superior computational throughput, while UPMEM, as a newer technology, requires further optimization to achieve competitive performance. This section examines architectural bottlenecks in SpMSpV and SpMV kernels that hinder the performance of graph applications on UPMEM. We analyzed detailed profiling metrics obtained from PIMulator [68]. Using PIMulator, we identify key microarchitectural bottlenecks in UPMEM-PIM that contribute to performance loss, providing

TABLE IV
EXECUTION TIME (MS), COMPUTE UTILIZATION (%), AND ENERGY CONSUMPTION (J) IN CPU, GPU, AND PIM FOR DIFFERENT DATASETS

Algo.	System	Execution Time (ms)						Compute Utilization (%)						Energy Consumption (J)					
		A302	as00	s-S11	p2p-24	e-En	face	A302	as00	s-S11	p2p-24	e-En	face	A302	as00	s-S11	p2p-24	e-En	face
BFS	CPU	541.1	38.5	44.5	117.1	44.5	27.1	0.09	0.002	0.07	0.001	0.04	0.02	17.30	0.90	1.05	3.18	1.10	0.64
	GPU	7.08	0.89	2.2	1.23	1.22	0.96	0.50	0.01	0.11	0.01	0.09	0.04	0.14	0.02	0.05	0.03	0.02	0.02
	UPMEM-Kernel	76.6	2.62	8.2	5.67	8.24	3.53	90.1	3.58	55.7	3.81	26.8	22.2	35.6	1.22	3.80	2.63	3.82	1.64
	UPMEM-Total	241.1	13.3	33.4	23.0	31.5	9.55	28.6	0.71	13.7	0.94	7.01	8.20	111.9	6.15	15.5	10.7	14.6	4.43
SSSP	CPU	1900	61	1056	166.5	656.1	232	0.04	0.002	0.01	0.002	0.01	0.01	57.2	1.52	24.5	4.74	13.6	6.44
	GPU	12.7	13.0	12.9	12.8	12.5	13.1	0.42	0.0007	0.04	0.002	0.01	0.02	0.25	0.26	0.26	0.26	0.25	0.28
	UPMEM-Kernel	62.7	4.3	8.3	7.9	11.8	5.3	165	4.36	131	6.98	39.5	40.1	29.1	0.20	3.84	3.67	5.46	2.46
	UPMEM-Total	340	19.9	49.3	29.9	43.3	20.2	30.5	0.94	21.9	1.85	10.7	10.5	158	9.23	22.9	13.9	20.1	9.36
PPR	CPU	216	126	177	88.5	197	84.0	0.13	0.002	0.02	0.004	0.02	0.01	7.25	3.46	4.48	2.22	5.63	2.40
	GPU	18.2	14.3	18.6	13.0	18.0	12.7	0.11	0.002	0.02	0.003	0.01	0.01	0.35	0.28	0.33	0.27	0.31	0.26
	UPMEM-Kernel	78.5	37.2	76.5	17.7	58.7	22.4	51.4	0.90	10.6	2.71	8.66	6.49	36.6	13.8	17.3	26.3	27.4	10.1
	UPMEM-Total	196.2	45.9	144	46.9	84.4	104	20.5	0.73	3.97	1.22	6.09	1.40	91.0	21.3	66.7	21.8	38.7	48.6

insights into kernel behavior that are difficult to obtain from real-system experiments.

*1) **DPU Utilization and Thread-Level Activity**:* Figure 9 breaks down DPU execution into: ① periods when the thread scheduler actively issues threads into the pipeline (green bar); and ② periods when the scheduler is idle (non-green bars), caused by threads waiting on memory operations, pipeline scheduling constraints, or structural hazards in the register file. These factors limit DPU utilization, leading to low IPC. We make four key observations.

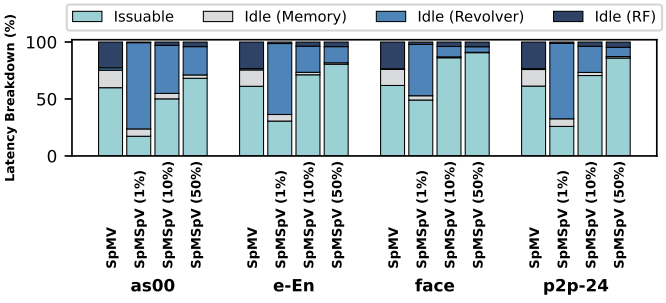

Fig. 9. DPU runtime breakdown into active (green) and idle (non-green bars) cycles for SpMV and SpMSpV at input vector densities of 1%, 10%, and 50%. Idle cycles are categorized by stall reasons: memory (gray), revolver pipeline (light blue), and RF structural hazard (dark blue).

► First, *SpMSpV with input densities $>$ 10% shows a higher percentage of issuable cycles than SpMV.* This is because SpMSpV focuses only on the non-zero entries of the input vector, reducing unnecessary computation and improving memory access locality. This results in fewer stalls and better DPU pipeline utilization, allowing more instructions to be issued per cycle.

► Second, *as input density increases in SpMSpV (from 1% to 50%), revolver pipeline stalls decrease.* Higher input density means more non-zero elements are available per active column, enabling better instruction-level parallelism. This reduces scheduling gaps in UPMEM's revolver pipeline, which requires 11 cycles between dispatching consecutive instructions from the same thread.

► Third, *SpMV suffers from more memory and register file (RF) stalls compared to SpMSpV.* SpMV accesses the entire input vector—regardless of actual sparsity—leading to irregular and less predictable memory access patterns, which increase memory stalls. Additionally, since every matrix element contributes to output updates, SpMV generates more RF contention, whereas SpMSpV avoids many such operations by skipping zero elements.

► Fourth, *at 1% density, SpMSpV exhibits higher revolver pipeline stalls compared to higher densities.* At low density, fewer active columns lead to reduced useful computation per thread and more frequent synchronization on shared output vector entries. In CSC-based SpMSpV, mutexes serialize these updates, preventing multiple tasklets from progressing simultaneously, which underutilizes the pipeline despite having sufficient tasklets. This combination of low computational load and contention leads to elevated revolver stalls.

Figure 10 presents the average number of active threads per cycle for SpMV and SpMSpV kernels across various datasets and input vector densities. *SpMSpV achieves higher thread activity as input density increases, benefiting from more parallel work per DPU.* At 1% density, limited active columns reduce thread engagement, but this improves at 10% and 50%. In contrast, SpMV shows lower thread activity due to irregular input access, reinforcing its lower DPU utilization observed earlier.

Recommendations: Revolver pipeline efficiency could be improved by adding intra-thread data forwarding for independent instructions, as proposed in [68]. Further, enabling non-blocking DMA would let tasklets continue computation while waiting for memory, requiring changes to the pipeline's thread dispatch logic to improve instruction-level parallelism.

*2) **Instruction Mix and Synchronization Overheads**:* Figure 11 presents the instruction mix across SpMV and SpMSpV at 1%, 10%, and 50% input vector densities. Three key insights follow:

► First, *synchronization instructions (e.g., barrier, mutex_lock) account for a large share in SpMSpV, especially at low densities.* Sparse input leads to fewer active updates and higher contention over shared output entries, increasing synchronization. Higher densities distribute updates more evenly, reducing contention and synchronization overhead.

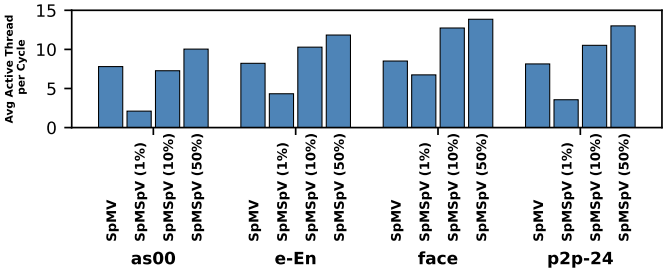

Fig. 10. Average Active Thread Per Cycle

▶ Second, S*pMV exhibits more arithmetic operations than SpMSpV*, as it processes all rows regardless of input vector sparsity, while SpMSpV skips inactive columns.

▶ Third, s*cratchpad instructions account for a non-trivial share across all implementations due to UPMEM's scratchpad-centric model*, which requires explicit DRAM-to-WRAM transfers as all register operands must be loaded from WRAM. To mitigate this overhead, UPMEM executes Load/Store instructions with single-cycle latency [68], preserving efficiency despite the lack of automated caching.

Recommendations: Implementing more efficient synchronization techniques could reduce idle time and improve system efficiency.

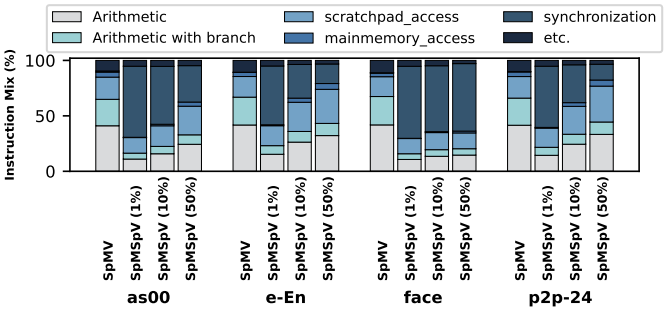

Fig. 11. Instruction mix for SpMV and SpMSpV at input vector densities of 1%, 10%, and 50%.

VII. Related Work

To our knowledge, this is the first study to extensively characterize linear-algebraic graph processing on a real PIM system. We provide a brief overview of related prior work.

Graph Processing in PIM Systems. PIM-based accelerators boost graph processing by improving memory bandwidth efficiency, integrating computation directly within memory to reduce data transfers and mitigate bandwidth bottlenecks [21], [33]–[40], [80]–[82].

Sparse Computation in PIM Systems. Giannoula et al. [41] present the first SpMV library for real PIM systems. Sun et al. [83] design buffer-optimized DIMMs enabling inter-DIMM communication for irregular tasks like SpMV and graph processing. Fujiki et al. [84] enhance GPU memory controllers by integrating PIM cores to convert matrices into DCSR format. Xie et al. [85] propose a 3D PNM design for SpMV using heterogeneous PIM in HMC. Zhu et al. [86] develop a PIM accelerator for SpMM in 3D-stacked memory systems.

Graph Processing in Commodity Systems. Several CPU-based frameworks support either sequential or coarse-grained parallel programming models. To address scalability challenges, distributed CPU-based frameworks have been developed to tackle issues such as synchronization overhead and load imbalance [87]–[94]. Several GPU-based frameworks implement graph primitives, with specialized parallel graph algorithms [62], [95]–[117]. However, these frameworks do not generalize well across different types of graph applications but show significant performance gains. Generalized frameworks [59], [118]–[124], [124]–[143] offer greater programming flexibility. Studies on GPU-optimized SpMV [44], [58], [144]–[150] focus on efficient matrix formats (e.g., CSR, COO). Tools like ClSpMV [151] and ML methods [57] help select formats, while libraries such as MKL [152] and cuSPARSE [153] are widely used, with some research exploring tensor cores [154].

VIII. Conclusion

We introduce *ALPHA-PIM*, the first linear-algebraic graph application framework designed for a real PIM system, and conduct a comprehensive analysis of widely-used graph algorithms on UPMEM. Our study underscores the crucial role of selecting effective partitioning strategies and compressed matrix formats to enhance performance on PIM systems. We demonstrate the critical need for PIM architectures incorporating direct inter-DPU communication capabilities to address data transfer challenges in graph applications, paving the way for future advancements in PIM technology for large-scale graph processing tasks.

Acknowledgment

We are grateful to UPMEM for granting us access to their systems. We thank the anonymous reviewers for their constructive feedback.

References

[1] G. L. Polites and R. T. Watson, "The centrality and prestige of cacm," *Communications of the ACM*, vol. 51, no. 1, pp. 95–100, 2008.
[2] A. Azad, A. Buluç, and A. Pothen, "A parallel tree grafting algorithm for maximum cardinality matching in bipartite graphs," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 1075–1084.
[3] O. Ivanciuc and A. Balaban, "The graph description of chemical structures," *Topological indices and related descriptors in QSAR and QSPR*, pp. 59–167, 1999.
[4] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.
[5] U. Hahn and U. Reimer, "Computing text constituency: An algorithmic approach to the generation of text graphs," in *Proceedings of the 7th annual international ACM SIGIR conference on Research and development in information retrieval*, 1984, pp. 343–368.
[6] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

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

[8] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on gpus: Where are the bottlenecks?" in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 140–149.

[9] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press Cambridge, 2001, vol. 5.

[11] M. Gondran and M. Minoux, *Graphs, dioids and semirings: new models and algorithms*. Springer Science & Business Media, 2008, vol. 41.

[12] ——, "Dioïds and semirings: Links to fuzzy sets and other applications," *Fuzzy Sets and Systems*, vol. 158, no. 12, pp. 1273–1294, 2007.

[13] S. Dolan, "Fun with semirings: a functional pearl on the abuse of linear algebra," in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, 2013, pp. 101–110.

[14] D. Dolžan and P. Oblak, "Commuting graphs of matrices over semirings," *Linear algebra and its applications*, vol. 435, no. 7, pp. 1657–1665, 2011.

[15] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system," *IEEE Access*, vol. 10, pp. 52 565–52 608, 2022.

[16] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking memory-centric computing systems: Analysis of real processing-in-memory hardware," in *2021 12th International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2021, pp. 1–7.

[17] A. Elafrou, G. Goumas, and N. Koziris, "Conflict-free symmetric sparse matrix-vector multiplication on multicore architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–15.

[18] ——, "Performance analysis and optimization of sparse matrix-vector multiplication on modern multi-and many-core processors," in *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 2017, pp. 292–301.

[19] J. Dongarraxz, A. Lumsdaine, X. Niu, R. Pozoz, and K. Remingtonx, "Sparse matrix libraries in c++ for high performance architectures," 1994.

[20] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *The International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004.

[21] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.

[22] I. Afanasyev and V. Voevodin, "The comparison of large-scale graph processing algorithms implementation methods for intel knl and nvidia gpu," in *Russian Supercomputing Days*. Springer, 2017, pp. 80–94.

[23] A. Gharaibeh, E. Santos-Neto, L. B. Costa, and M. Ripeanu, "The energy case for graph processing on hybrid cpu and gpu systems," in *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, 2013, pp. 1–8.

[24] S. Aga, N. Jayasena, and M. Ignatowski, "Co-ml: a case for co llaborative ml acceleration using near-data processing," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 506–517.

[25] J. Choe, A. Huang, T. Moreshet, M. Herlihy, and R. I. Bahar, "Concurrent data structures with near-data-processing: An architecture-aware implementation," in *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, 2019, pp. 297–308.

[26] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 283–295.

[27] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[28] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 316–331.

[29] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45Th annual international symposium on computer architecture (ISCA)*. IEEE, 2018, pp. 383–396.

[30] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu, "Processing-in-memory: A workload-driven perspective," *IBM Journal of Research and Development*, vol. 63, no. 6, pp. 3–1, 2019.

[31] C. Giannoula, P. Yang, I. Fernandez, J. Yang, S. Durvasula, Y. X. Li, M. Sadrosadati, J. G. Luna, O. Mutlu, and G. Pekhimenko, "Pygim: An efficient graph neural network library for real processing-in-memory architectures," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 8, no. 3, Dec. 2024.

[32] C. Giannoula, N. Vijaykumar, N. Papadopoulou, V. Karakostas, I. Fernandez, J. Gómez-Luna, L. Orosa, N. Koziris, G. Goumas, and O. Mutlu, "Syncron: Efficient synchronization support for near-data-processing architectures," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

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

[34] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "Graphh: A processing-in-memory architecture for large-scale graph processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 640–653, 2018.

[35] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *2017 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 2017, pp. 457–468.

[36] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan *et al.*, "Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 282–297.

[37] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 336–348, 2015.

[38] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng *et al.*, "Conda: Efficient cache coherence support for near-data accelerators," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 629–642.

[39] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "Lazypim: An efficient cache coherence mechanism for processing-in-memory," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 46–50, 2016.

[40] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 712–725.

[41] C. Giannoula, I. Fernandez, J. G. Luna, N. Koziris, G. Goumas, and O. Mutlu, "Sparsep: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–49, 2022.

[42] C. Yang, A. Buluc, and J. D. Owens, "Graphblast: A high-performance linear algebra-based graph framework on the gpu," *arXiv preprint arXiv:1908.01407*, 2019.

[43] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[44] C. Yang, Y. Wang, and J. D. Owens, "Fast sparse matrix and sparse vector multiplication algorithm on the gpu," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 841–847.

[45] C. Yang, A. Buluç, and J. D. Owens, "Implementing push-pull efficiently in graphblas," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–11.

[46] A. Azad and A. Buluç, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 688–697.

[47] H. Ji, H. Song, S. Lu, Z. Jin, G. Tan, and W. Liu, "Tilespmspv: A tiled algorithm for sparse matrix-sparse vector multiplication on gpus," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–11.

[48] UPMEM, "Introduction to upmem pim. processing-in-memory (pim) on dram accelerator (white paper)," 2018.

[49] B. Yang and H. Garcia-Molina, "Improving search in peer-to-peer networks," in *Proceedings 22nd International Conference on Distributed Computing Systems*. IEEE, 2002, pp. 5–14.

[50] V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti, "A local search mechanism for peer-to-peer networks," in *Proceedings of the eleventh international conference on Information and knowledge management*, 2002, pp. 300–307.

[51] G. T. Talk, "Fast route planning," 2009.

[52] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck, "Highway dimension, shortest paths, and provably efficient algorithms," in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 2010, pp. 782–793.

[53] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "A hub-based labeling algorithm for shortest paths in road networks," in *International Symposium on Experimental Algorithms*. Springer, 2011, pp. 230–241.

[54] M. Yang, H. Wang, Z. Wei, S. Wang, and J.-R. Wen, "Efficient algorithms for personalized pagerank computation: A survey," *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[55] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," 2007.

[56] U. W. Pooch and A. Nieder, "A survey of indexing techniques for sparse matrices," *ACM Computing Surveys (CSUR)*, vol. 5, no. 2, pp. 109–133, 1973.

[57] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse matrix format selection with multiclass svm for spmv on gpu," in *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016, pp. 496–505.

[58] M. Besta, F. Marending, E. Solomonik, and T. Hoefler, "Slimsell: A vectorizable graph representation for breadth-first search," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 32–41.

[59] P. Zhang, M. Zalewski, A. Lumsdaine, S. Misurda, and S. McMillan, "Gbtl-cuda: Graph algorithms and primitives for gpus," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 912–920.

[60] NVIDIA, "nvgraph library," https://developer.nvidia.com/nvgraph, 2019.

[61] UPMEM, "Upmem website," https://www.upmem.com, 2020.

[62] S. Beamer, K. Asanovic, and D. Patterson, "Ligra breadth-first search," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–10.

[63] J. van Leeuwen, "Graph algorithms," in *Algorithms and complexity*. Elsevier, 1990, pp. 525–631.

[64] S. Khuller and B. Raghavachari, "Graph and network algorithms," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 43–45, 1996.

[65] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[66] Intel, "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, 2017.

[67] JEDEC, "Jesd79-4 ddr4 sdram standard)," in *JEDEC*, 2012.

[68] B. Hyun, T. Kim, D. Lee, and M. Rhu, "Pathfinding future pim architectures by demystifying a commercial pim technology," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 263–279.

[69] GraphChallenge, "Graphchallenge website," https://graphchallenge.mit.edu/data-sets, 2003.

[70] S. L. N. D. Collection, "Stanford large network dataset collection website," https://snap.stanford.edu/data/, 2009.

[71] Intel, "Intel® core™ i7-1265u," https://www.intel.com/content/www/us/en/products/sku/226258/intel-core-i71265u-processor-12m-cache-up-to-4-80-ghz/specifications.html, 2022.

[72] NVIDIA, "Nvidia rtx 3050," https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3050/, 2022.

[73] PeakPerf, "Peakperf," https://github.com/Dr-Noob/peakperf, 2021.

[74] I. O. Source, "Rapl power meter," https://www.intel.com/content/www/us/en/developer/topic-technology/open/overview.html, 2014.

[75] ——, "Nvidia system management interface program," https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf, 2016.

[76] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, no. 11, pp. 0–40, 2011.

[77] X. Zhu, W. Han, and W. Chen, "{GridGraph}:{Large-Scale} graph processing on a single machine using 2-level hierarchical partitioning," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 375–386.

[78] RAPIDS, "cugraph library," https://github.com/rapidsai/nvgraph, 2021.

[79] DataScienceFramework, "Rapids," https://rapids.ai/start.html, 2021.

[80] S. Cai, B. Tian, H. Zhang, and M. Gao, "Pimpam: Efficient graph pattern matching on real processing-in-memory hardware," *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–25, 2024.

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

[82] D. Baek, S. Hwang, and J. Huh, "psyncpim: Partially synchronous execution of sparse matrix operations for all-bank pim architectures," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 354–367.

[83] W. Sun, Z. Li, S. Yin, S. Wei, and L. Liu, "Abc-dimm: Alleviating the bottleneck of communication in dimm-based near-memory processing with inter-dimm broadcast," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 237–250.

[84] C. Hong, A. Sukumaran-Rajam, B. Bandyopadhyay, J. Kim, S. E. Kurt, I. Nisa, S. Sabhlok, Ü. V. Çatalyürek, S. Parthasarathy, and P. Sadayappan, "Efficient sparse-matrix multi-vector product on gpus," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 66–79.

[85] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, and Y. Xie, "Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 570–583.

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

[87] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[88] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a {PC}," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 31–46.

[89] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 599–613.

[90] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: a dsl for easy and efficient graph analysis," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 349–362.

[91] S. Salihoglu and J. Widom, "Help: High-level primitives for large-scale graph processing," in *Proceedings of Workshop on GRAph Data management Experiences and Systems*, 2014, pp. 1–6.

[92] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 456–471.

[93] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 17–30.

[94] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[95] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *International conference on high-performance computing*. Springer, 2007, pp. 197–208.

[96] A. McLaughlin and D. A. Bader, "Scalable and high performance betweenness centrality on the gpu," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 572–583.

[97] A. E. Sariyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek, "Betweenness centrality on gpus and heterogeneous architectures," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, 2013, pp. 76–85.

[98] F. Busato and N. Bombieri, "Bfs-4k: an efficient implementation of bfs for kepler gpu architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 1826–1838, 2014.

[99] J. Greiner, "A comparison of parallel algorithms for connected components," in *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, 1994, pp. 16–25.

[100] P. Pande and D. A. Bader, "Computing betweenness centrality for small world networks on a gpu," in *15th Annual High performance embedded computing workshop (HPEC)*, 2011.

[101] J. Soman, K. Kishore, and P. Narayanan, "A fast gpu algorithm for graph connectivity," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, pp. 1–8.

[102] U. Meyer and P. Sanders, "δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[103] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[104] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[105] O. Green, P. Yalamanchili, and L.-M. Munguía, "Fast triangle counting on the gpu," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, 2014, pp. 1–8.

[106] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," *Acm Sigplan Notices*, vol. 46, no. 8, pp. 267–276, 2011.

[107] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," *ACM Sigplan Notices*, vol. 47, no. 8, pp. 117–128, 2012.

[108] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 349–359.

[109] J. Barnat, P. Bauch, L. Brim, and M. Ceška, "Computing strongly connected components in parallel on cuda," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 544–555.

[110] H. Liu, H. H. Huang, and Y. Hu, "ibfs: Concurrent breadth-first search on gpus," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 403–416.

[111] A. Polak, "Counting triangles in large graphs on gpu," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 740–746.

[112] A. Rungsawang and B. Manaskasemsak, "Fast pagerank computation on a gpu cluster," in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2012, pp. 450–456.

[113] M. Almasri, Y.-H. Chang, I. El Hajj, R. Nagi, J. Xiong, and W.-m. Hwu, "Parallelizing maximal clique enumeration on gpus," in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2023, pp. 162–175.

[114] M. Almasri, I. E. Hajj, R. Nagi, J. Xiong, and W.-m. Hwu, "Parallel k-clique counting on gpus," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–14.

[115] P. Yamout, K. Barada, A. Jaljuli, A. E. Mouawad, and I. El Hajj, "Parallel vertex cover algorithms on gpus," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 201–211.

[116] S. Diab, M. G. Olabi, and I. El Hajj, "Ktrussexplorer: Exploring the design space of k-truss decomposition optimizations on gpus," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–8.

[117] M. G. Olabi, J. G. Luna, O. Mutlu, W.-m. Hwu, and I. El Hajj, "A compiler framework for optimizing dynamic parallelism on gpus," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 1–13.

[118] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: vertex-centric graph processing on gpus," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, 2014, pp. 239–252.

[119] A. Gharaibeh, T. Reza, E. Santos-Neto, L. B. Costa, S. Sallinen, and M. Ripeanu, "Efficient large-scale graph processing on hybrid cpu and gpu systems," *arXiv preprint arXiv:1312.3018*, 2013.

[120] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 195–207.

[121] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a gpu," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 233–245.

[122] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel *et al.*, "Gunrock: Gpu graph analytics," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 1, pp. 1–49, 2017.

[123] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," in *Proceedings of Workshop on GRAph Data management Experiences and Systems*, 2014, pp. 1–6.

[124] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2013.

[125] X. Shi, J. Liang, S. Di, B. He, H. Jin, L. Lu, Z. Wang, X. Luo, and J. Zhong, "Optimization of asynchronous graph processing on gpu with hybrid coloring model," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 271–272, 2015.

[126] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang, "Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on gpu," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 38–52.

[127] H. Liu and H. H. Huang, "Simd-x: Programming and processing of graph algorithms on gpus," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 411–428.

[128] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: minimizing data transfer during out-of-gpu-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[129] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for gpu-friendly graph processing," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 622–636, 2018.

[130] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, vol. 2, pp. 1–18, 2005.

[131] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on gpus," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 1–19.

[132] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken, "A distributed multi-gpu system for fast graph processing," *Proceedings of the VLDB Endowment*, vol. 11, no. 3, pp. 297–310, 2017.

[133] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim, "Gts: A fast and scalable graph processing method based on streaming topology to gpus," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 447–461.

[134] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-gpu programming model for irregular computations," *ACM SIGPLAN Notices*, vol. 52, no. 8, pp. 235–248, 2017.

[135] Y. Zhang, X. Liao, H. Jin, B. He, H. Liu, and L. Gu, "Digraph: An efficient path-based iterative directed graph processing system on multiple gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 601–614.

[136] C. R. Aberger, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: boolean algebra based graph processing," *ArXiv e-prints*, 2015.

[137] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan, "Multi-graph: Efficient graph processing on gpus," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 27–40.

[138] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu, "A yoke of oxen and a thousand chickens for heavy lifting graph processing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 345–354.

[139] H. Seo, J. Kim, and M.-S. Kim, "Gstream: A graph streaming processing method for large-scale graphs on gpus," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 253–254, 2015.

[140] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[141] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 78–88.

[142] J. Zhong and B. He, "Towards gpu-accelerated large-scale graph processing in the cloud," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 1. IEEE, 2013, pp. 9–16.

[143] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 39–50.

[144] C. Zheng, S. Gu, T.-X. Gu, B. Yang, and X.-P. Liu, "Biell: A bisection ellpack-based storage format for optimizing spmv on gpus," *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2639–2647, 2014.

[145] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 781–792.

[146] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–11.

[147] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for spmv on gpu using probabilistic modeling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 196–205, 2014.

[148] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on gpgpus," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 4, pp. 1–49, 2017.

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

[150] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu, "Efficient pagerank and spmv computation on amd gpus," in *2010 39th International Conference on Parallel Processing*. IEEE, 2010, pp. 81–89.

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

[152] Intel, "Mkl sparse-blas library," http://software.intel.com/en-us/intel-mkl, 2020.

[153] NVIDIA, "Nvidia's cusparse," https://docs.nvidia.com/cuda/cusparse/index.html, 2020.

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