RAPID-Graph: Recursive All-Pairs Shortest Paths Using Processing-in-Memory for Dynamic Programming on Graphs

Yanru Chen[1*], Zheyu Li[1*], Keming Fan[1], Runyang Tian[1], John Hsu[1],
Weihong Xu[2#], Minxuan Zhou[3], Tajana Rosing[1]
[1]University of California San Diego, La Jolla, CA, USA
[2]Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
[3]Illinois Institute of Technology, Chicago, IL, USA
{yac054, zhl178, k4fan, r3tian, joh048, tajana}@ucsd.edu; weihong.xu@epfl.ch; mzhou26@iit.edu
* Equal contribution. # Corresponding author.

***Abstract*—All-pairs shortest paths (APSP) remains a major bottleneck for large-scale graph analytics, as data movement with cubic complexity overwhelms the bandwidth of conventional memory hierarchies. We propose RAPID-Graph, a processing-in-memory (PIM) system co-designed across algorithm, architecture, and device levels to address this challenge. At the algorithm level, we introduce a recursion-aware partitioner that enables an exact APSP computation by decomposing graphs into vertex tiles to reduce data dependency, such that both Floyd-Warshall and Min-Plus kernels execute fully in-place within digital PIM arrays. At the architecture and device levels, we design a 2.5D PIM stack integrating two phase-change memory compute dies, a logic die, and high-bandwidth scratchpad memory within a unified advanced package. An external non-volatile storage stack stores large APSP results persistently. The design achieves both tile-level and unit-level parallel processing to sustain high throughput. On the 2.45M-node OGBN-Products dataset, RAPID-Graph is $5.8\times$ faster and $1\,186\times$ more energy efficient than state-of-the-art GPU clusters, while exceeding prior PIM accelerators by $8.3\times$ in speed and $104\times$ in efficiency. It further delivers up to $42.8\times$ speedup and $392\times$ energy savings over an NVIDIA H100 GPU.**

***Index Terms*—Processing-in-memory, phase-change memory, all-pairs shortest paths, Floyd–Warshall, dynamic programming**

I. Introduction

Graphs serve as the backbone of workflows spanning urban planning and transportation [1], [2], social and commercial analytics [3], [4], and autonomous LiDAR navigation [5]. Central to these applications, all-pairs shortest paths (APSP) provides a fundamental computational core, transforming graph connectivity into actionable insights. However, exact APSP computation remains computationally prohibitive due to inherent algorithmic complexity. The classic Floyd-Warshall (FW) algorithm requires $O(n^3)$ time and $O(n^2)$ space [6], while the repeated Dijkstra algorithm exhibits super-quadratic complexity with poor memory locality [7].

Recent GPU-based APSP accelerators launch massive parallelism, but still require large hardware footprints, heavy interconnect communication, and high energy [8], [9]. Partitioned-APSP computes APSP for a 2M-vertex graph in approximately 30 minutes but requires 128 GPUs with extensive DRAM reliance [10]; Co-ParallelFW achieves 8.1 PFLOP/s but requires complex coordination among 4,608 GPUs [11]; Fast-APSP scales up to 11.5M vertices utilizing 2 048 GPUs with compressed sparse row (CSR)-tile layouts and complex min-plus (MP) reductions, yet remains bottlenecked by inter-GPU communication overhead [12].

Processing-in-memory (PIM) reduces data movement by executing logic inside memory arrays. Prior PIM architectures validated through graph traversals: GraphR accelerates BFS through in-situ SpMV on ReRAM crossbars [13]; GraphPIM enables atomic rank-level SSSP updates using DRAM bit-line logic [14]; and Tesseract manages vertex-centric execution via remote procedure calls across HMC stacks [15]. Similarly, Temporal State Machines [16] accelerate Dijkstra SSSP by executing tropical algebra operations on time-coded wavefronts, achieving an edge traversal rate of 10 giga-edge traversals per second with a memristive temporal processor. Later systems such as GraphP [17], GraphH [18], and GraphQ [19] improve inter-cube communication and scheduling, but remain centered on sparse traversals with atomic updates over lightweight data. Exact APSP instead requires dense $O(n^2)$ matrices and heavy MP merges that quickly overwhelm on-chip SRAM and local memory bandwidth. No existing PIM design addresses this quadratic storage requirement or provides efficient in-memory MP reductions needed for APSP dynamic programming (DP).

To support dense APSP, memory technology must sustain cubic-time and quadratic-space DP with high-capacity, high-speed, and energy-efficient in-place updates. Filamentary RRAM enables sub-0.1 pJ writes, but 4-bit HfO_2 stacks [20] need $\sim$ $50\,\mu$s pulses and retain data for only $\sim 10^5$ s at 85°C, limiting throughput and longevity. Spin-orbit-torque MRAM [21] writes in 0.35 ns at 156 fJ/bit, but its $6-8\,F^2$ MTJ-transistor stack has lowers density. By contrast, $SiTe_x$ filamentary PCM [22] switches in 150 ns and 20 ns at ~ 10 pJ. We therefore adopt $Sb_2Te_3/Ge_4Sb_6Te_7$ 40 nm PCM [23], which provides ~ 20 ns switching, 1.5 pJ reset, 10^8 endurance, 10^5 h retention at 83°C, and a $150\times$ $R_{\text{on}}/R_{\text{off}}$ ratio.

To break through the fundamental memory and power walls in exact APSP computation, we introduce RAPID-Graph, a novel software-hardware co-designed PIM system. Our approach overcomes the communication limits of multi-GPU clusters and the dense-workload constraints of prior PIM architectures. Our core contributions are as follows:

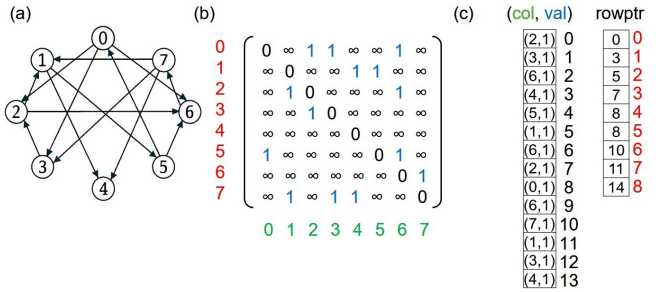

Fig. 1. Graph representations (a) Graph topology (b) Adjacent matrix (c) CSR

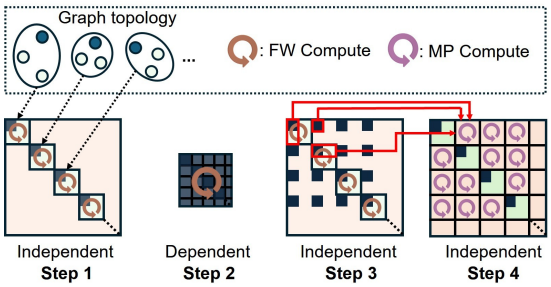

Fig. 2. Illustration of partition APSP

- Recursive in-memory APSP dataflow. We design an algorithm that recursively decomposes the graph into subproblems sized for a single PIM tile, enabling massively parallel FW execution entirely in PCM arrays and eliminating off-chip data movement.
- Heterogeneous 2.5D PIM architecture. We build a tailored architecture for this dataflow, integrating PCM compute dies with a logic die via UCIe, supported by an on-package HBM3 for hot submatrices and external FeNAND for bulk storage, forming a balanced memory hierarchy.
- SOTA performance and efficiency. RAPID-Graph delivers up to 1 061× speedup and 7 208× energy efficiency over CPU. On the OGBN-Products dataset (2.5M nodes), it outperforms GPU clusters by 5.8× in runtime and 1 186× in energy efficiency.

II. Background

A. Graphs

A graph $G = (V, E, w)$ can be stored as an adjacency matrix or CSR. Computation is most convenient on its dense adjacency matrix A, defined by $A_{ij} = w(i, j)$ for $(i, j) \in E$ and $+\infty$ otherwise, enabling regular dataflow for FW and MP kernels. For storage, we adopt CSR, which records only the nonzero weights together with row boundaries, reducing space from n^2 to $|E|$. Fig. 1 illustrates an 8-vertex toy graph in three views: (a) the original topology, (b) the dense matrix with finite weights marked, and (c) the compressed CSR `rowptr, col, val`, demonstrating space savings when $|E| \ll n^2$.

B. APSP Algorithms

1) Classic FW DP: The FW algorithm [6] solves the APSP problem on a weighted graph $G = (V, E, w)$ via an in-place dynamic program over an $n \times n$ distance matrix D. $D[i][j]$ is set to the weight of edge (i, j), or ∞ if no edge exists. The algorithm updates all entries by checking whether paths through vertex k yield shorter distances by:

$$D[i][j] = \min\left(D[i][j],\ D[i][k] + D[k][j]\right), k \in [1, n].$$

After n iterations, D contains the exact shortest path lengths between all vertex pairs. The algorithm runs in $O(n^3)$ time and $O(n^2)$ space, following a computation pattern equal to a dense outer product of row i and column k against column j.

2) Partitioned APSP: Our recursive partition APSP extends the four-stage scheme by [10], which reshapes FW into tile-local semiring operations, aligning with PIM's in-place parallelism. Here, the semiring concretely refers to the min-plus algebra used by APSP ($\oplus = \min$, $\otimes = +$). The algorithm is illustrated in Fig. 2 and summarized in Algorithm 1. Graph preprocessing runs on host CPU, where a weighted graph G is partitioned into components $C_1, \ldots, C_k$ and their boundary set B via a k-way METIS [24]. Within each component, a boundary vertex has an edge connecting to another component, while an internal vertex only has edges to vertices within its own component. For computational efficiency, boundary vertices are reordered before internal vertices.

Algorithm 1 Partition APSP Pseudocode

```
Partition G into k components C_1, ..., C_k via METIS [24]
for i = 1 to k do                                   ▷ Step 1
    FW(C_i)
G_B ← EXTRACTBOUNDARYGRAPH(G)                       ▷ Step 2
FW(G_B)
for i = 1 to k do                                   ▷ Step 3 with injected d_B
    FW(C_i)
for i = 1 to k do                                   ▷ Step 4
    for j = 1 to k do
        MINPLUSMERGE(C_i, C_j)
return global distance matrix
```

Step 1: Local APSP. Each component independently runs FW to fill its intra-component distance matrix d_{intra}; all passes execute in parallel and scale linearly.

Step 2: Boundary-graph APSP. All boundary vertices form a reduced graph G_B, with edges comprising: (i) cross-component edges from G, and (ii) virtual edges within components weighted by d_{intra}. A single FW run computes the boundary distance matrix d_B. As this step involves dense $O(|B|^3)$ work, it becomes the primary bottleneck when $|B|$ nears hardware limits. Section III-A addresses this bottleneck through recursive partitioning.

Step 3: Boundary injection. Each component copies the relevant rows and columns of d_B into its local matrix and reruns FW once, propagating inter-component shortcuts.

Step 4: Cross-component merge. An MP merge combines (i) source to boundary, (ii) boundary to boundary, and (iii) boundary to destination paths, producing the final cross-component distances d_{cross} and thus completing global APSP.

C. Phase Change Memory

PCM [25] encodes data by switching a chalcogenide cell between high-resistance amorphous state (reset, 0) and low-resistance crystalline state (set, 1) via Joule heating. Cells form 1T1R cross-point arrays whose wordlines and bitlines are managed by selection transistors. PCM offers fast random reads, low leakage, and non-volatility, enabling in-place parallel updates and persistent storage in our PIM system.

TABLE I
KEY VARIABLES IN RECURSIVE PARTITIONED APSP

Variable	Description
$G = (V, E, w)$	Graph with vertices V, edges E, edge weights w
$C_i^{(\ell)}$	Component at level ℓ (ℓ=1,...,n and i=1,...,k)
$B_i^{(\ell)}$	Boundary vertex set of component $C_i^{(\ell)}$
$G_B^{(\ell)}$	Level-ℓ boundary graphs
$\mathrm{DB}^{(\ell)}$	Boundary-to-boundary distance matrix at level ℓ
D_C	Intra-component APSP distance of component C
$D_{C_1}[m, n]$	Cross-component distance from $m \in C_1$ to $n \in C_2$

Our PCM array supports bit-serial logic primitives inspired by FELIX [26]. Single-cycle NOR, NOT, NAND, minority, OR and 2-cycle XOR operations execute natively in crossbar memory via voltage-controlled switching. These primitives enable efficient composition of more complex functions: 1-bit addition and min-comparison are implemented via FELIX [26] primitives. Addition computes the sum bit $S = A \oplus B \oplus C_{in}$, and the carry-out $C_{out} = Maj(A, B, C_{in})$. Min-comparison uses bit-serial subtraction $S = A \oplus (\neg B) \oplus 1$, where most significant bit of S is the sign bit gating selective updates.

III. RAPID-GRAPH SOFTWARE-HARDWARE CO-DESIGN

A. Recursive Partitioned APSP

To efficiently scale APSP computation on large graphs, we design a recursive partitioning strategy that enables fully independent subgraphs sized to fit within PIM tile limits. Accordingly, we partition each component at $|V| \leq 1024$, matching practical array dimensions per PCM tile and the maximum parallelism achievable with dense, high-yield fabrication. The input graph $G = (V, E, w)$, with vertex set V, edge set E, and non-negative weights w, is first partitioned by METIS [24] into base-level components $C_1^{(0)}, \ldots, C_k^{(0)}$. Each component $C_i^{(0)}$ contains internal vertices and boundary vertices, where boundary vertices connect to other components. We extract boundary vertices from $C_i^{(0)}$ to construct the level-0 boundary graph $G_B^{(0)}$. If boundary graph $G_B^{(\ell)}$ at level ℓ exceeds the 1024-vertex tile limit, we recursively partition it, creating a coarser graph $G_B^{(\ell+1)}$. This continues until $|V(G_B^{(n)})| \leq 1024$, ensuring all graphs fit entirely within tiles. Table I summarizes key variables. At each recursion level, APSP is computed locally within components and boundary graphs, propagating distances back into components and performing cross-component updates via MP products. Fig. 3 illustrates this process. After recursive partitioning each memory array holds one dense distance block with $N \leq 1024$ vertices. To maximize parallelism during the FW updates, we adopt a specialized data remapping strategy. This strategy logically separates the current pivot row and column (the `Panel_Row` and `Panel_Col`) from the rest of the distance matrix (the `Main_Block`). This layout allows the FW die to perform massively parallel updates on the `Main_Block`. The detailed mapping and scheduling schemes for this process are described in Section III-D.

Algorithm 2 summarizes the recursive APSP procedure across hierarchy levels. It follows the same four steps as Algorithm 1, but operates bottom-up: starting from base-level partitions, each level computes local APSP and propagates boundary summaries upward. This structure maximizes tile-level parallelism and fits within constrained PIM resources without requiring global synchronization.

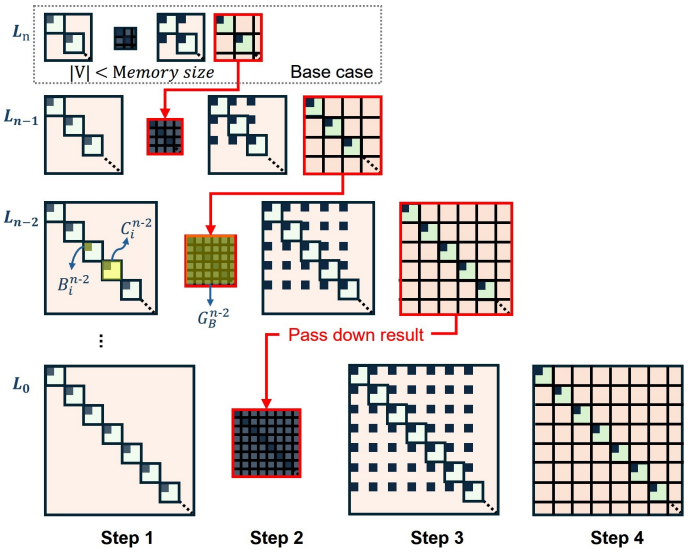

Fig. 3. Illustration of recursive partitioned APSP

Algorithm 2 Recursive Partition APSP Pseudocode

for $\ell = n$ down to 0 **do**
  parallel for C in levels[ℓ] **do** ▷ Step 1
    $D_C \leftarrow$ FloydWarshall(C)
    $B_C \leftarrow$ find_boundary(C)
    if DB_prev == $\varnothing$: $DB_C \leftarrow$ restrict(D_C, B_C)
  if DB_prev == $\varnothing$: ▷ Step 2
    $G_B \leftarrow$ build_boundary_graph($\{DB_C$ for all $C\}$)
    DB_prev $\leftarrow$ FloydWarshall(G_B)
  parallel for C in levels[ℓ] **do** ▷ Step 3
    $D_C \leftarrow$ inject(DB_prev, B_C)
  parallel for (C_1, C_2) in levels[ℓ] **do** ▷ Step 4
    for m in B_{C_1}, n in B_{C_2} **do**
      $D_{C_1}[m, n] \leftarrow \min_{\substack{i \in B_{C_1} \\ j \in B_{C_2}}} (D_{C_1}[m, i] + \text{DB\_prev}[i, j] + D_{C_2}[j, n])$
  DB_prev $\leftarrow$ merge($\{$restrict(D_C, B_C)$\}$)
return DB_prev

B. Heterogeneous Architecture Overview

To scale APSP, we build a heterogeneous multi-die architecture that integrates high-bandwidth in-memory compute with large nonvolatile storage, as shown in Fig. 4(a). We co-package the PCM-FW die, PCM-MP die, logic base die, and HBM3 using a 2.5D silicon interposer with a UCIe v1.0 raw interface [27], delivering 64 full-duplex lanes at 32 Gb/s each, as illustrated in Fig. 4(b). Off-package FeNAND is an external storage array mounted on the PCB and connected via ONFI 5.1 ×16 channels. It delivers several key advantages over conventional charge-trap NAND, including higher capacity, lower program voltage, faster access, and a shorter string length [28]. This multi-die design maps the recursive APSP algorithm's FW and MP kernels onto dedicated PIM compute dies. Fine-grained partitioning, coupled through the high-bandwidth UCIe link, creates a tailored dataflow that eases the memory and communication bottlenecks of large-scale APSP.

Inside the package, the system components are organized by function. The logic base die serves as the central controller, managing system-wide dataflow and leveraging its dual stream

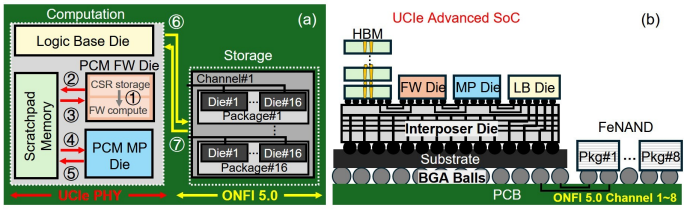

Fig. 4. RAPID-Graph heterogeneous PIM system (a) System-level dataflow of UCIe-based multi-die SoC and off-package FeNAND storage, annotated with computation-storage dataflow (b) 2.5D advanced packaging cross-section

engines for high-throughput CSR-to-dense format conversion. In-memory computation is executed on two specialized 2 GB PCM compute dies: the FW die is tailored for intra-component updates, while the MP die accelerates cross-partition merges. This processing core is supported by a tiered memory hierarchy, where a 16 GB HBM3 stack provides high-bandwidth scratchpad buffering, and a 16 TB FeNAND [29] array offers dense persistent storage for boundary data and final results in compressed CSR format. The logic die orchestrates a multi-stage recursive APSP across this hierarchy (Fig. 4(a)):

① CSR-partitioned components are streamed from the cold-storage region of the PCM-FW die into its compute region and expanded into dense matrices. ② The PCM-FW die performs local FW updates, and results are streamed back to HBM3 in row-wise segments. ③ HBM3 (i) extracts boundary nodes and edges to construct a boundary graph for the next FW iteration, and (ii) prefetches next intra-component FW blocks for pipelined execution. ④ The PCM-MP die fetches sub-matrices from HBM3 to perform cross-partition MP merges. ⑤ HBM3 synchronizes boundary data across partitions to prepare for the next boundary-aware FW round. ⑥ (i) Updated boundary matrices are stored in dense format in FeNAND, whereas final cross-partition APSP results from PCM-MP and (ii) intra-component APSP from PCM-FW are compressed to CSR format before writing to FeNAND. ⑦ The PCM-MP die fetches interleaved boundary matrices from FeNAND to complete final cross-partition merges.

C. Hardware Design and Implementation

Both PCM-FW and PCM-MP dies use the same 1T1R SLC PCM cell technology [23], organized into 1024×1024 units. The parameters for the SLC cells are listed in Table II. Each unit, together with its peripheral circuits, forms a macro, as shown in Fig. 5(a). Each tile contains 130 parallel units connected via an H-tree interconnect [30] for efficient data exchange (Fig. 5(b)). A unit is a 1024×1024 PCM macro including array and periphery. The design ensures full crossbar activity without idle cycles. Same-colored arrows represent concurrent unit-to-unit transfers. Tiles operate concurrently, each with a local controller linked to a global main controller for coordinated execution (Fig. 5(c)). In-memory operations are triggered by row-segment broadcasts, with each tile controller managing parallel processing across its units. In addition to shared peripheral logic, each PCM die integrates specialized circuits for its role. The PCM-FW tile includes a permutation unit for rearranging data blocks, while the PCM-MP tile

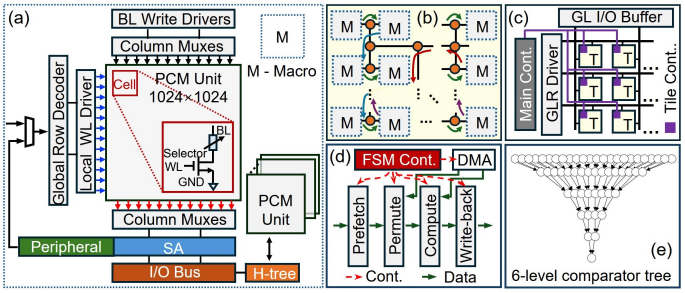

Fig. 5. PCM die detail architecture (a) Macro - M (b) Tile - T (c) Die (d) Permutation unit for PCM-FW (e) 6-level min comparator tree for PCM-MP

TABLE II
KEY PARAMETERS OF $Sb_2Te_3/Ge_4Sb_6Te_7$ SLC PCM

Parameter	Value
Reset pulse	40 μA @ 0.70 V
Programming energy	$\approx$ 0.56 pJ
Resistance (LRS)	$\approx$ 30 kΩ, 150$\times$ on/off ratio
Set/reset time	20 ns / 20 ns
Clock cycle	2 ns (500 MHz)

integrates a 32-bit 6-level min-comparator tree for efficient MP reductions.

PCM-FW Permutation Unit. The PCM-FW die includes a dedicated permutation macro that locally rearranges data blocks, avoiding off-die data movement. As shown in Fig. 5(d), the permutation macro comprises:

1) a 32-row burst row-buffer controller,
2) a reorder buffer for panel masking and block pruning,
3) a lightweight on-tile DMA engine (1-cycle read, 10-cycle write) with address remapper, and
4) a four-stage FSM pipeline (Prefetch $\rightarrow$ Permute $\rightarrow$ Compute $\rightarrow$ Write-back),

so that data movement overlaps computation. The permutation unit packs `Panel_Row` and mirrored `Panel_Col` into 32-row windows for coalesced bursts without H-tree stalls, while a `prefetch` buffer hides DMA latency and skip futile writes, sustaining near-peak occupancy and reducing wear.

PCM-MP Min-Compare & Update. Each unit integrates a pipelined comparator tree (Fig. 5(e)) that reduces 1024 32-bit inputs to one 32-bit minimum in following periods: A 1024 $\times$ 32-bit row is streamed into the buffer in 1 cycle. Thirty-two parallel five-level carry look-ahead (CLA) trees extract sign bits and 5-bit indices for block minima in 6 cycles. A second five-level tree reduces these to a global minimum in another 6 cycles, ... totaling 13 cycles per row. A sign-bit mask gates PCM writes, updating only improved entries. Two staging buffers hold operands across adds while $DB[i, j]$ streams, sustaining one 1024-wide vector per cycle. The reduction tree outputs an update mask that enables compare-and-swap selective writes, avoiding read–modify–write and lowering energy and wear.

D. Mapping and Scheduling Schemes

To execute FW and MP efficiently in PCM tiles, we co-optimize software layout and hardware design through tailored intra-tile mapping and scheduling. Fig. 6(a) illustrates the FW DP update flow. Since diagonal pivot elements p_k always have zero distance, their propagation along pivot row (r_1,...r_n) and column (c_1,...c_n) is omitted. Instead, row and column

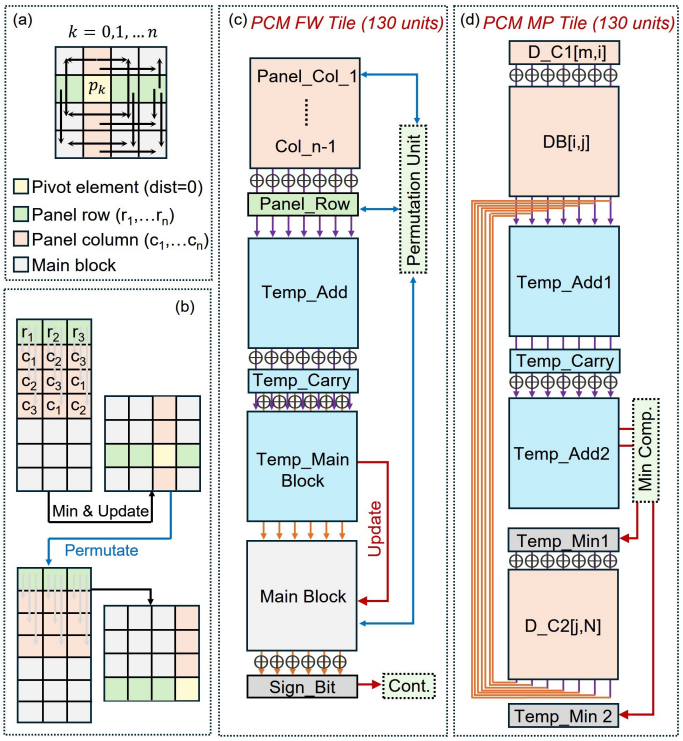

Fig. 6. SW/HW mapping for recursive partitioned APSP (a) FW illustration (b) FW remapping, every step includes add, min update and permutation (c) PCM-FW tile performing FW (d) PCM-MP tile performing two-stage MP

elements propagate into the main block to perform add and min operations, updating main block values with MP products. Each pivot element triggers one such update; the process repeats until all diagonal pivots p_k (k=1,..,n) are processed. Fig. 6(b) shows our remapping strategy that maximizes array parallelism. We extract the pivot row, copy its column beneath it, and form an $(n-1) \times (n-1)$ block; this lets all updates finish with one add and one min. The next pivot element is then permutated, and the process repeats.

Both PCM FW and PCM MP tiles contain 130 units and are partitioned into regular functional regions. In the PCM FW tile, the pivot column is stored in `Panel_Col`$[1: n-1]$, the pivot row is stored in `Panel_Row`, and the remaining entries reside in `Main_Block`. Intermediate values are buffered in `Temp_Add`, `Temp_Carry`, and `Temp_Main_Block`, while the final sign bit is recorded in `Sign_Bit`. Fig. 6(c) shows the PCM FW datapath. Add is implemented with FELIX [26] bit-serial adders using `Temp_Add` and `Temp_Carry`. A FELIX [26] bit-serial subtraction compares `Temp_Main_Block` against `Main_Block`, and the resulting sign bit gates selective writes back to `Main_Block`. The permutation unit then reorders `Panel_Row`, `Panel_Col`, and `Main_Block` for the next pivot, as detailed in Section III-C. Fig. 6(d) shows the PCM MP tile, which also comprises 130 units and performs a two-stage MP merge. In the first stage, a logical row (1×1024) from C_1 is read into $D_{C_1}[m,i]$, while $DB[i,j]$ and the corresponding row $D_{C_2}[j,n]$ are presented in parallel. The merge applies two successive MP steps, $D_{C_1}[m,i]$ + $DB[i,j]$ followed by $(\cdot)$ + $D_{C_2}[j,n]$, each followed by a 1024-way min reduction. Computation uses FELIX bit-serial adders in `Temp_Add1`, `Temp_Carry`, and `Temp_Add2`, and a pipelined comparator tree reduces all candidates to a single minimum stored in `Temp_Min1` and `Temp_Min2`.

IV. Experimental Results and Analysis

A. Experimental Setup

We benchmark RAPID-Graph on the real-world OGBN-Products graph [31] for fair SOTA comparison, and on synthetic Newman–Watts–Strogatz (NWS) [32] and Erdős–Rényi (ER) [33] graphs generated via NiemaGraphGen [34] for controlled scalability and topology analysis. NWS preserves dense intra-community but sparse inter-community links, while ER has uniformly random edges. These datasets reveal how graph topology impacts RAPID-Graph's scalability.

We compare against CPU, GPU, SOTA PIM method Temporal PIM SSSP [16], and SOTA GPU distributed methods including Partitioned APSP [10] and Co-Parallel APSP [11]. Since no SOTA PIM methods directly implement APSP, we estimate the performance of the Temporal PIM SSSP [16] to establish a comparable APSP PIM baseline, hereafter referred as PIM-APSP. Graph partitioning is performed using METIS 5.1.0 [24]. Consistent with prior work [10], [11], the METIS partitioning overhead is not included in our results as it is a preprocessing step. The baseline configurations are:

1) **CPU**: Intel i7-11700K (64 GB)
2) **GPU A100**: NVIDIA A100-SXM4 (80 GB)
3) **Estimated GPU H100 [35]**: NVIDIA H100 (80 GB)

We develop an in-house cycle-accurate simulator to model RAPID-Graph's functionality. PCM arrays and peripheral circuits are modeled with NeuroSim [36] for device-level accuracy. We synthesized the RTL in System Verilog using Synopsys Design Compiler with a 40 nm CMOS PDK at 500 MHz, including custom permutation unit, min-comparator tree, and controller; all results are scaled to 14 nm using models [37].

B. Hardware Configurations

Each PCM-FW tile integrates a permutation unit for FW, while each PCM-MP tile includes a min-comparator tree for MP. As shown in Table III, 82% of unit area stems from peripheral circuits (sense amplifiers, 1-bit comparators, WL/SL drivers, decoders, D latch buffers), while compute units contribute negligible overhead, supporting dense integration and efficient APSP execution. PIM logic integration lower PCM density but removes costly data movement, delivering higher performance and efficiency.

System-level supporting components contribute moderate power and area: HBM3 (16 GB) adds 8.6 W and 121 mm^2 [38]; FeNAND (16 TB) adds 6.4 W across 3000 mm^2; the SM2508 controller adds 3.5 W within a 225 mm^2 BGA package. The total power of $\sim$ 18.5 W remains significantly lower than high-end GPUs such as the NVIDIA H100, which consumes up to 700 W under peak workloads [35] with higher latency.

TABLE III
AREA AND POWER BREAKDOWN PER PCM UNIT

Component	PCM-FW		PCM-MP	
	Area (μm^2)	Power (mW)	Area (μm^2)	Power (mW)
PCM Subarray	3288 (13.80%)	557 (80.64%)	3288 (13.61%)	557 (80.61%)
Permutation Unit	917.3 (3.85%)	0.586 (0.08%)	-	-
Min Comparator	-	-	1268 (5.25%)	0.684 (0.10%)
Controller	5.94 (0.02%)	0.00126 (<0.01%)	5.94 (0.02%)	0.00126 (<0.01%)
Others	19610 (82.33%)	133.29 (19.28%)	19610 (81.12%)	133.29 (19.29%)
Total	23821.24 (100%)	690.88 (100%)	24171.94 (100%)	690.98 (100%)

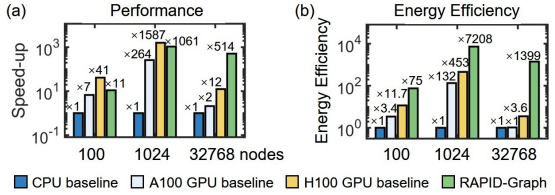

Fig. 7. RAPID-Graph vs. CPU, A100 GPU and H100 GPU baselines across graph sizes (a) Speedup (b) Energy efficiency

C. RAPID-Graph Performance and Efficiency

1) Speedup and Energy Efficiency: We compare RAPID-Graph against CPU and GPU (A100 and estimated H100) baselines on graphs with 100, 1 024, and 32 768 nodes synthesized using NiemaGraphGen [34]. These sizes avoid memory bottlenecks on single-node hardware. Fig. 7 shows large gains in speed and energy. At 1 024 nodes, RAPID-Graph delivers $1\,061\times$ speedup and $7\,208\times$ energy efficiency over CPU. At 32 768 nodes, it outperforms H100 by $42.8\times$ in speed and $392\times$ in energy. RAPID-Graph's performance gains grow with graph size due to enhanced parallelism and in-memory execution. This performance gap widens dramatically with graph size because conventional systems are overwhelmed by $O(n^3)$ data movement. This massive data transfer saturates memory bandwidth that RAPID-Graph inherently avoids.

We also compare RAPID-Graph with SOTA PIM method PIM-APSP [16] and SOTA GPU clusters methods Partitioned APSP [10] and Co-Parallel APSP [11]. On OGBN-Products (2.5M nodes) dataset [31], we estimate their performance from reported scaling trends. As shown in Fig. 8, RAPID-Graph outperforms both, achieving $5.8\times$ speedup over Co-Parallel APSP and $1\,186\times$ energy savings over Partitioned APSP. While PIM-APSP improves energy efficiency by $11.4\times$, it slows down performance to $0.7\times$ of the baseline. The advantage of RAPID-Graph comes from removing inter-GPU communication, the dominant limiter at large-scale, multi-node APSP solutions.

2) Scalability: Fig. 9 compares the scalability of RAPID-Graph and the H100 GPU baseline across graph degree, size, and topology. In Fig. 9(a,d), both systems maintain stable performance across degree, suggesting edge count has limited effect on exact APSP when memory is sufficient. In Fig. 9(b,e), RAPID-Graph scales linearly to 2.45M nodes, while H100 exhibits rising latency and superlinear energy growth beyond 10^3 nodes due to communication overhead. H100 is limited by the $O(n^2)$ distance matrix overwhelms its caches, whereas RAPID-Graph's in-situ computation preserves locality. In Fig. 9(c,f), RAPID-Graph achieves better efficiency on clustered and real-world graphs than on random ones, benefiting from structural locality and fewer partitioning boundaries, while H100 remains largely topology-insensitive. This topology-awareness is a direct result of our partitioning algorithm, as clustered graphs like NWS produce smaller boundary sets, reducing the workload of the computationally dominant boundary-graph APSP step. RAPID-Graph provides competitive scalability without the immense hardware cost and overhead of GPU clusters. Partitioned-FW [10] uses 2,560 GPUs for a 1.9M-node graph but hits synchronization and memory walls. Co-ParallelFW [11] achieves only 45% weak-scaling efficiency on a 300K-node graph. Overall, RAPID-Graph avoids the complexity, synchronization, and energy overhead of distributed GPU clusters.

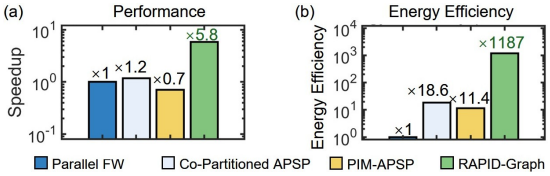

Fig. 8. RAPID-Graph vs. PIM-APSP [16], Partitioned APSP [10], and Co-Parallel APSP [11] running APSP on OGBN-Products dataset [31] (a) Speedup (b) Energy efficiency

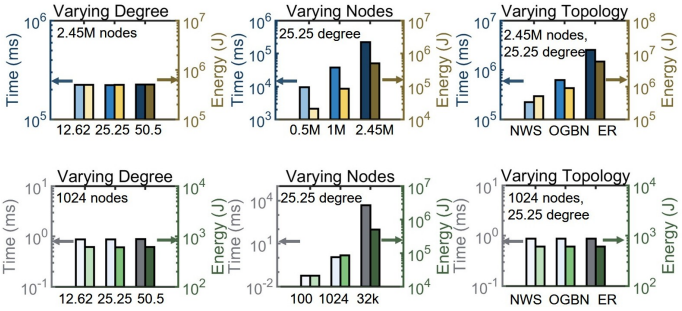

Fig. 9. Scalability of RAPID-Graph (top row) and H100 GPU (bottom row) (a,d) Degree sweep at fixed size (b,e) Size sweep at degree 25.25 (c,f) Topology clustered (NWS), real (OGBN), and random (ER) at fixed size and degree

V. CONCLUSION

We present RAPID-Graph, the first software–hardware co-designed PIM system for exact APSP. A recursive partitioner maps large graphs into tile-sized subproblems that fit within PCM arrays, enabling fully in-place and massively parallel Floyd–Warshall and Min-Plus execution. A heterogeneous 2.5D stack integrates two PCM compute dies with on-package HBM3 and off-package FeNAND through a high-bandwidth UCIe interposer, sustaining the end-to-end dataflow required by dense APSP. Across real and synthetic graphs, RAPID-Graph consistently outperforms CPU, GPU, and prior PIM-based solutions in both runtime and energy, and scales linearly to million-node graphs while remaining robust to degree and topology variations. Beyond APSP, the proposed recursive partitioning and in-memory min-plus execution naturally extend to dense DP kernels with associative combine-and-reduce patterns.

ACKNOWLEDGMENT

This work was supported in part by PRISM and CoCoSys—centers in JUMP 2.0, an SRC program sponsored by DARPA (SRC grant number - 2023-JU-3135). This work was also supported by NSF grants #2003279, #1911095, #2112167, #2052809, #2112665, #2120019, #2211386.

REFERENCES

[1] A. D. Putriani, S. L. Chasanah *et al.*, "Comparison of the dijkstra's algorithm and the floyd-warshall's algorithm to determine the shortest path between hospitals in several cities in lampung province," *Journal of Mathematical Sciences and Optimization*, vol. 1, no. 2, pp. 44–52, 2024.

[2] D. Aktaş, E. Vermeir, and P. Vansteenwegen, "Speeding up the passenger assignment problem in transit planning by the direct link network representation," *Computers & Operations Research*, vol. 167, p. 106647, 2024.

[3] M. N. P. Ma'ady, T. S. N. Syahda, A. F. Rizqi, and M. C. A. Ratna, "On using floyd-warshall under uncertainty for influence maximization in instagram social network: A case study of indonesian fnb unicorn company," *Procedia Computer Science*, vol. 234, pp. 164–171, 2024.

[4] T. Simas, R. B. Correia, and L. M. Rocha, "The distance backbone of complex networks," *Journal of Complex Networks*, vol. 9, no. 6, p. cnab021, 2021.

[5] Z. Wan, B. Yu, T. Y. Li, J. Tang, Y. Zhu, Y. Wang, A. Raychowdhury, and S. Liu, "A survey of fpga-based robotic computing," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 48–74, 2021.

[6] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, pp. 345–345, 1962.

[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Edsger Wybe Dijkstra: his life, work, and legacy*, 2022, pp. 287–290.

[8] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on gpus: A survey," *CSUR*, vol. 50, no. 6, pp. 1–35, 2018.

[9] P. Xie, Z. Zheng, Y. Zhou, Y. Xiu, H. Liu, Z. Yang, Y. Zhang, and B. Du, "Gpu architectures in graph analytics: A comparative experimental study," 2025.

[10] H. Djidjev, G. Chapuis, R. Andonov, S. Thulasidasan, and D. Lavenier, "All-pairs shortest path algorithms for planar graph for gpu-accelerated clusters," *JPDC*, vol. 85, pp. 91–103, 2015.

[11] P. Sao, H. Lu, R. Kannan, V. Thakkar, R. Vuduc, and T. Potok, "Scalable all-pairs shortest paths for huge graphs on multi-gpu clusters," in *HPCA*, 2021, pp. 121–131.

[12] S. Yang, X. Liu, Y. Wang, X. He, and G. Tan, "Fast all-pairs shortest paths algorithm in large sparse graph," in *ICS*, 2023, pp. 277–288.

[13] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 HPCA*, 2018, pp. 531–543.

[14] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *HPCA*, 2017, pp. 457–468.

[15] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, 2015, pp. 105–117.

[16] A. Madhavan, M. W. Daniels, and M. D. Stiles, "Temporal state machines: Using temporal memory to stitch time-based graph computations," *JETC*, vol. 17, no. 3, pp. 1–27, 2021.

[17] 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 *HPCA*. IEEE, 2018, pp. 544–557.

[18] P. Sun, Y. Wen, T. N. B. Duong, and X. Xiao, "Graphh: High performance big graph analytics in small clusters," in *2017 CLUSTER*. IEEE, 2017, pp. 256–266.

[19] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *MICRO*, 2019, pp. 712–725.

[20] H. He, Y. Tan, C. Lee, and Y. Zhao, "Ti/hfo2-based rram with superior thermal stability based on self-limited tiox," *Electronics*, vol. 12, no. 11, p. 2426, 2023.

[21] T. Nguyen, H. Naganuma, H. Honjo, Y. Sato, T. Tanigawa, S. Ikeda, and T. Endoh, "Low write power and field-free sub-ns write speed sot-mram cell with design technology of canted sot structure and magnetic anisotropy for nvm," in *IMW*. IEEE, 2025, pp. 1–4.

[22] S.-O. Park, S. Hong, S.-J. Sung, D. Kim, S. Seo, H. Jeong, T. Park, W. J. Cho, J. Kim, and S. Choi, "Phase-change memory via a phase-changeable self-confined nano-filament," *Nature*, vol. 628, no. 8007, pp. 293–298, 2024.

[23] X. Wu, A. I. Khan, H. Lee, C.-F. Hsu, H. Zhang, H. Yu, N. Roy, A. V. Davydov, I. Takeuchi, X. Bao *et al.*, "Novel nanocomposite-superlattices for low energy and high stability nanoscale phase-change memory," *Nature Communications*, vol. 15, no. 1, p. 13, 2024.

[24] G. Karypis and V. Kumar, "Multilevelk-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed computing*, vol. 48, no. 1, pp. 96–129, 1998.

[25] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[26] S. Gupta, M. Imani, and T. Rosing, "Felix: Fast and energy-efficient logic in memory," in *ICCAD*. IEEE, 2018, pp. 1–7.

[27] M.-S. Lin, C.-C. Tsai, S. Li, W.-C. Chen, W.-H. Huang, Y.-C. Chen, Y.-J. Huang, A. Drake, C.-H. Wen, P. Ranucci *et al.*, "36.1 a 32gb/s 10.5 tb/s/mm 0.6 pj/b ucie-compliant low-latency interface in 3nm featuring matched-delay for dynamic clock gating," in *ISSCC*, vol. 68. IEEE, 2025, pp. 586–588.

[28] M. Yun, G. Lee, G. Ryu, H. Kim, and M. Kang, "The optimization of program operation for low power consumption in 3d ferroelectric (fe)-nand flash memory," *Electronics*, vol. 13, no. 2, p. 316, 2024.

[29] J.-G. Lee, W.-T. Koo, G. Lee, J. Kim, W. Lee, H. D. Lee, S. Yoon, S.-I. Hong, I.-K. Kang, J. Kim, H. Choi, S. G. Kim, S. Lee, J. Yi, and S. Y. Cha, "Analog computation in ultra-high density 3d fenand for tb-level hyperscale ai models," in *IEDM*, 2024, pp. 1–4.

[30] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *ISCA*, 2016, pp. 14–26.

[31] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks," in *KDD*, 2019, pp. 257–266.

[32] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[33] P. ERDdS and A. R&wi, "On random graphs i," *Publ. math. debrecen*, vol. 6, no. 290-297, p. 18, 1959.

[34] N. Moshiri, "Niemagraphgen: A memory-efficient global-scale contact network simulation toolkit," *GIGAbyte*, vol. 2022, p. gigabyte37, 2022.

[35] W. Luo, R. Fan, Z. Li, D. Du, Q. Wang, and X. Chu, "Benchmarking and dissecting the nvidia hopper gpu architecture," in *2024 IPDPS*. IEEE, 2024, pp. 656–667.

[36] P. Xiaochen, S. Huang, H. Jiang, A. Lu, and S. Yu, "Dnn+ neurosim v2. 0: An end-to-end benchmarking framework for compute-in-memory accelerators for on-chip training," *TCAD*, no. 11, pp. 2306–2319, 2020.

[37] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm," *Integration*, vol. 58, pp. 74–81, 2017.

[38] J. Park, J. Choi, K. Kyung, M. J. Kim, Y. Kwon, N. S. Kim, and J. H. Ahn, "Attacc! unleashing the power of pim for batched transformer-based generative model inference," in *ASPLOS, Volume 2*, 2024, pp. 103–119.