GEN-Graph: Heterogeneous PIM Accelerator for General Computational Patterns in Graph-based Dynamic Programming

Yanru Chen[1], Runyang Tian[1], Zheyu Li[1], Mahbod Afarin[1], Weihong Xu[2], Tajana Šimunić Rosing[1] *Fellow, IEEE*
[1]University of California San Diego, La Jolla, CA, USA
[2]Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
{yac054, r3tian, zhl178, mafarin, tajana}@ucsd.edu; weihong.xu@epfl.ch

***Abstract*—While graph-based dynamic programming (DP) is a cornerstone of genomics and network analytics, its efficiency is hampered by fundamentally conflicting computational patterns. Matrix-centric DP drives regular, compute-bound network analytics, while topology-centric DP handles irregular, memory-bound genomic traversals. These two categories of DP have substantially different computation patterns and dataflows, which makes it difficult for a single homogeneous processing-in-memory (PIM) architecture to efficiently support both. This work presents GEN-Graph, a novel heterogeneous PIM chiplet that integrates two types of specialized compute tiles within a 2.5D package: Matrix-tile, a processing-using-memory (PUM) tile optimized for matrix-centric workloads, such as all-pairs shortest path (APSP); and traversal-tile, a processing-near-memory (PNM) tile optimized for traversal-centric DP workloads, such as DNA sequence alignment. Our hardware-software co-design employs recursive partitioning and reconfigurable windowed bit-parallel logic to ensure exact computation. Results show the matrix tile achieves $42.8\times$ speedup and $392\times$ energy efficiency over the NVIDIA H100 GPU for APSP. For sequence-to-graph alignment, the traversal tile sustains 2.56 million reads/s (short-reads) and 39.3 thousand reads/s (long-reads), outperforming state-of-the-art accelerators by up to $2.56\times$ in throughput. GEN-Graph provides the first scalable, exact solution for general DP dataflows by matching hardware specialization to algorithmic structure.**

***Index Terms*—Processing-in-memory, heterogeneous chiplet accelerator, graph-based dynamic programming, hardware-software co-design**

I. Introduction

Graph analytics provides the computational foundation for a vast range of applications spanning social network analysis [1]–[3], bioinformatics [4]–[7], and logistics [8], [9]. As social networks [10] and genomics graphs [11], [12] scale beyond billions of nodes, they demand different algorithmic strategies. These range from fundamental traversals like Breadth-First Search (BFS) [13] for connectivity, to iterative methods like PageRank [14] focused on node importance. Dynamic programming (DP) [15], [16] further optimizes complex path-finding by memoizing overlapping subproblems. By avoiding full-graph recomputation, DP enables efficient, real-time updates in large-scale evolving networks—a critical capability across numerous domains. In bioinformatics, key applications include sequence-to-graph (S2G) alignment [17], which is crucial for modern pangenome analysis [18]. In broader domains such as network science and transportation, the all-pairs shortest path (APSP) [19] problem remains a classic and vital computation.

Although these algorithms share mathematical foundations, they employ diverse computational structures with unique recurrence relations, data dependencies, and update mechanisms [20]–[22]. We classify graph DP into two distinct computational archetypes: matrix-centric DP and topology-centric DP. Matrix-centric DP problems operate on static graph structures where connectivity is dense or can be regularized. The dataflow resembles blocked matrix multiplication, characterized by high spatial locality and predictable reuse [23], [24]. Here, the bottleneck is peak arithmetic throughput. Conversely, topology-centric DP kernels exhibit large-scale graphs where edges represent dynamic transitions [22]. The dataflow is dominated by pointer chasing and indirect memory addressing. This poor data locality leads to frequent memory stalls and low effective bandwidth from random memory access. These distinct dataflows create a significant gap in general graph-based DP hardware acceleration. To understand how to bridge this gap, we analyze APSP and S2G alignment as representative workloads for these two archetypes in Section IV-A.

Recent GPU-based accelerators launch massive parallelism, but graph DP still faces scaling and efficiency limits [25], [26]. For instance, the performance of dense APSP is dictated by the efficiency of executing $O(n^3)$ min-plus (MP) matrix multiplication operations [27]. State-of-the-art (SOTA) designs focus on offloading this compute-intensive part to massively parallel GPU architectures to approach theoretical peak performance: Partitioned-APSP computes APSP for a 2M-vertex graph in approximately 30 minutes but requires 128 GPUs with extensive DRAM reliance [27]; Co-ParallelFW achieves 8.1 PFLOP/s but requires complex coordination among 4 608 GPUs [28], showing the need for large hardware footprints, heavy interconnect communication, and high energy [25], [26]. In SOTA traversal sequence alignment accelerator designs, heterogeneous graph aligner (HGA) [29] achieves a $15.8\times$ speedup via CPU-GPU co-processing, yet it remains constrained by long reads where larger alignment scores necessitate more bits, thereby reducing SIMD lanes and overall parallel efficiency. Overall, data movement remains a dominant bottleneck for graph DP at scale [30].

Processing-in-memory (PIM) mitigates this bottleneck by moving computation closer to where data resides [31], [32]. PIM can be categorized into two architectural forms: processing-using-memory (PUM) and processing-near-memory (PNM). PUM exploits the physical properties of non-volatile devices to execute in-situ bit-serial operations, providing high throughput for regular workloads [33]–[35]. Conversely, PNM integrates CMOS logic near memory banks, utilizing technologies like HBM to provide the bandwidth

and flexibility required for irregular addressing logic [36]–[38]. However, most PIM accelerators are domain-specific and optimized for a single kernel, which conflicts with the compute–memory divergence in graph DP. On one end, S2G alignment is dominated by latency-sensitive, irregular memory behaviors: profiling of leading S2G software shows that both seeding and alignment suffer from high DRAM latency, driven by unpredictable pointer chasing, large intermediate tables, and high cache miss rates [39], [40]. On the other end, APSP workloads exhibit compute-dense min-plus updates that map well to PIM. RAPID-Graph [41] shows strong speed and energy gains for APSP, but its datapath is too rigid for traversal-centric computation. As a result, a one-size-fits-all PIM design is underutilized across these regimes.

This compute-memory divergence defines the design of existing PIM-based graph accelerators. One class integrates fixed logic for high-performance sequence alignment, as seen in GenASM [40] and SeGraM [39], but these rigid structures cannot adapt to diverse graph DP patterns. Conversely, programmable frameworks like GenDP [42] offer generality at the cost of significant performance penalties compared to dedicated hardware. This inefficiency stems from the mismatch between varied dataflows and a single homogeneous architecture. These approaches face a fundamental conflict between architectural rigidity and execution efficiency. These limitations necessitate a heterogeneous solution that balances hardware specialization with general applicability.

To address this challenge, we propose GEN-Graph, a software-hardware co-designed chiplet-based heterogeneous PIM architecture for general graph-based DP. A matrix tile is designed to efficiently execute dense, matrix-like operations characteristic of algorithms such as APSP. A traversal tile is optimized for the sparse, pointer-chasing workloads found in S2G alignment. This heterogeneity optimizes resource to workload matching. Our core contributions are as follows:

- We perform a detailed analysis of graph-based DP algorithms and identify two fundamental and distinct computational patterns that motivate the need for a heterogeneous architecture.
- We propose GEN-Graph, the first chiplet-based heterogeneous PIM architecture designed to efficiently accelerate both dense and sparse computational patterns.
- We develop a compiler that implements tailored data mapping and scheduling schemes for each tile to maximize hardware utilization. This automated flow enables high efficiency across diverse graph applications.
- We conduct a comprehensive evaluation of GEN-Graph, demonstrating a $42.8\times$ speedup over NVIDIA H100 for APSP by exploiting in-situ bit-parallelism, and a $2.56\times$ throughput gain over SOTA accelerators for S2G alignment by localizing irregular dependencies within shared banked SRAM.

II. Background on Graph Dynamic Programming

A graph $G = (V, E, w)$ is defined by its vertices V, edges E, and weights w. Fig. 1 illustrates an 8-vertex graph through three standard lenses: (a) the original topology, (b) a dense matrix with finite weights, and (c) the CSR format with $\{\texttt{rowptr}, \texttt{col}, \texttt{val}\}$, which optimizes storage when $|E| \ll n^2$. Beyond standard representations, genome graphs (Fig. 1(d)) incorporate features like bubbles and joins to encode genetic diversity. These divergences define the DP archetypes described below: regular matrices enable high compute parallelism for APSP, while irregular topologies induce the sparse, non-local dependencies of sequence alignment.

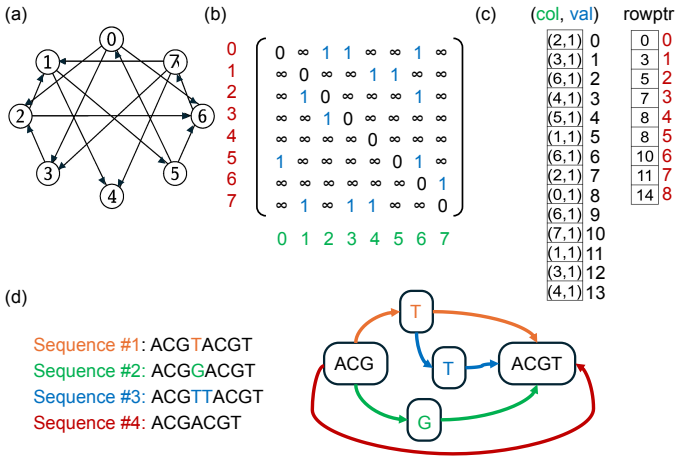

Fig. 1. Graph representations (a) Topology (b) Adjacent matrix (c) Compressed sparse row (CSR) (d) Genome graph

A. Graph-based DP for APSP

The classic FW algorithm [19] solves the APSP problem on a weighted graph $G = (V, E, w)$ via an in-place DP 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 statically updates all entries by checking if paths through vertex k yield shorter distances by:

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

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. This regularity allows the workload to map effectively to systolic arrays or bit-serial compute-in-memory arrays, where performance scales linearly with logic density.

The SOTA GPU algorithm partitioned APSP [27] is summarized in Algorithm 1 and illustrated in Fig.2. 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 [43]. 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. The algorithm then executes in four stages:

- 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

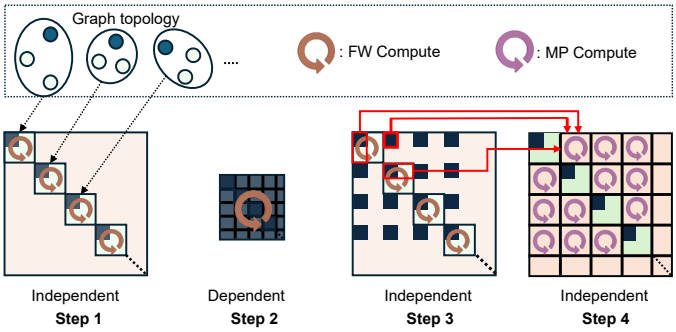

Fig. 2. Illustration of partition APSP [27]

Algorithm 1 Partition APSP Pseudocode

```
Partition G into k components C_1, ..., C_k via METIS [43]
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
```

computes the boundary distance matrix d_B. The resulting boundary distance matrix (d_B) involves dense $O(|B|^3)$ operations, creating the primary compute bottleneck addressed via recursive partitioning in Section IV-D1.

- Step 3: Boundary injection. Each component copies the relevant rows and columns of d_B into its local matrix and re-runs 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}, thus completing global APSP.

B. Graph-based DP for Genome Sequence Alignment

Genome graphs provide a powerful representation of genetic diversity by integrating a linear reference with known variations. Unlike a conventional linear genome that introduces reference bias toward a single individual, genome graphs represent diverse populations as a directed acyclic graph. In this model, nodes represent sequences of DNA base pairs (bps) and edges connect nodes adjacent in the genome of one or more individuals. Structural irregularities such as bubbles and joins encode specific variations, including single-nucleotide polymorphisms (SNPs) and insertions/deletions (indels) [4].

In traditional sequence-to-sequence (S2S) alignment, the use of a linear reference restricts data dependencies to immediate spatial neighbors, maximizing cache locality as depicted in Fig. 3(a). In contrast, sequence-to-graph (S2G) alignment which incorporates genetic variations through complex graph topology as shown in Fig. 3(b). The S2G recurrence relation introduces topological irregularity defined by

$$S[v] \leftarrow \max_{u \in \text{Predecessors}(v)} (S[u] + \text{score}(u, v)) \tag{2}$$

where v denotes the current graph vertex and u represents a predecessor node. This variable set Predecessors(v) creates arbitrary long-range hops. These sparse dependencies scatter

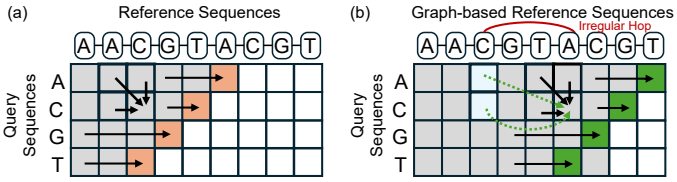

Fig. 3. Genome sequencing (a) Sequence-to-sequence (b) Sequence-to-graph

memory accesses across the address space, disrupting the spatial locality and causing frequent cache misses.

Computational characteristics further bifurcate according to read length, imposing distinct processing requirements on the underlying hardware. Short-read platforms like Illumina [44] generate massive volumes of 100–300 bp sequences with low error rates below 1%. The primary bottleneck here is maximizing throughput by amortizing the control overhead of millions of fine-grained, independent queries. In contrast, long-read technologies such as PacBio [45] and Oxford Nanopore [46] produce sequences ranging from 10 kbp to more than 100 kbp with higher error profiles of 5–15%. These extensive reads create deep dependency chains within the alignment matrix, requiring sustained pipeline utilization to manage the latency of traversing large, complex graphs.

III. Related Work

Prior work on accelerators for graph-based DP has largely converged on domain-specific designs. Existing approaches typically optimize for either sparse traversal behavior or dense matrix-style computation, reflecting recurring computational archetypes rather than application-specific choices. Each archetype necessitates a tailored hardware strategy, a principle that has become increasingly central with the shift toward domain-specific architectures [47]. Accordingly, we classify existing work into three categories: topology-centric, matrix-centric, and generalized frameworks.

SeGraM [39] integrates seeding and bit-parallel alignment into a topology-centric design that supports both S2S and S2G mapping. It achieves up to 742× speedup over software baselines and 4.8× improvement over prior accelerators. GenASM [40] targets approximate alignment using register-level bit vectors with minimal area and energy cost but lacks support for graph-shaped references or traceback. ASGDP [48] implements the full DP in FPGA with prediction-based pruning and MaxHop control, delivering a 17.8× throughput gain while preserving over 99.8% alignment accuracy.

Dense workloads such as APSP benefit from regular, compute-bound execution patterns. RAPID-Graph [41] accelerates APSP into recursively tiled MP operations on a domain-specific PCM-based accelerator and outperforms SOTA in both speed and energy. In contrast, RACE logic [49] encodes values as propagation delays in logic and remains effective only for regular graphs with limited control-flow irregularity.

Recent work has attempted to bridge this gap through a programmable framework. GenDP [42] proposes a programmable systolic array capable of supporting multiple genomics kernels, delivering 132× area-normalized speedup over CPUs. Despite its flexibility, GenDP incurs a 2.8× performance penalty compared to task-specific ASICs GenAx [50] due to the

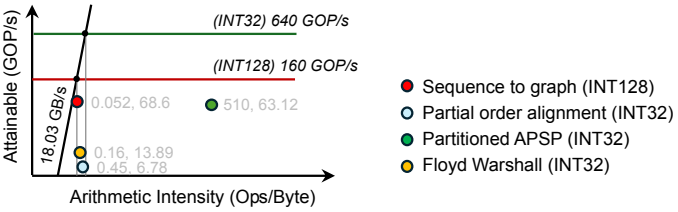

Fig. 4. Roofline model of four algorithms on CPU AVX-512

inefficiency of mapping diverse computational patterns to a homogeneous substrate. This trade-off underscores the necessity of GEN-Graph's heterogeneous design, which physically decouples compute-bound and memory-bound resources.

IV. GEN-GRAPH: HETEROGENEOUS PIM DESIGN

GEN-Graph resolves the compute-memory divergence in graph DP by mapping specialized hardware tiles to specific algorithmic dataflows. Profiling reveals orders-of-magnitude gap in arithmetic intensity that necessitates the physical decoupling of matrix-centric and topology-centric architectures (Section IV-A). We establish a memory hierarchy using HBM3, PCM, and FeNAND to align physical device strengths with varied access frequencies (Section IV-B).

The accelerator integrates high-density matrix tiles for cubic complexity updates and reconfigurable traversal tiles that adapt to varying sequence lengths (Sections IV-D and IV-E). For APSP workloads,the matrix tile executes a co-designed recursive partitioned algorithm via in-situ bit-parallelism to saturate throughput during dense updates. For genomic workloads, the traversal tile implements a windowed S2G algorithm using a tiered storage hierarchy to sustain exact alignment accuracy without excessive area overhead. This co-designed accelerator enables high hardware utilization by overlapping in-situ computation with hierarchical data streaming.

A. GEN-Graph Motivation

Fig. 4 provides a roofline analysis that we performed on an Intel i7-11700K (AVX-512) to quantify the compute–memory divergence between APSP and S2G. Profiling reveals orders-of-magnitude gap in arithmetic intensity between graph DP archetypes, necessitating architectural heterogeneity.

Matrix-centric DP. In Fig. 4, partitioned APSP is compute bound with an intensity of 510 Ops/Byte. It represents an optimized version of the FW algorithm (0.16 Ops/Byte). While classic FW remains memory-limited, the partitioning strategy maximizes cache residency to achieve 63.12 GOP/s. This compute-limited state motivates a PUM architecture that uses bit-serial parallelism to maximize arithmetic throughput.

Topology-centric DP. In Fig. 4, S2G and POA are memory bound, with intensities of 0.052 and 0.45 Ops/Byte respectively. Their irregular graph traversals cause frequent cache misses and force compute units to wait for data. These results confirm that data movement is the primary bottleneck for topology-based workloads. This motivates a PNM approach that integrates logic near HBM to reduce latency and improve effective bandwidth.

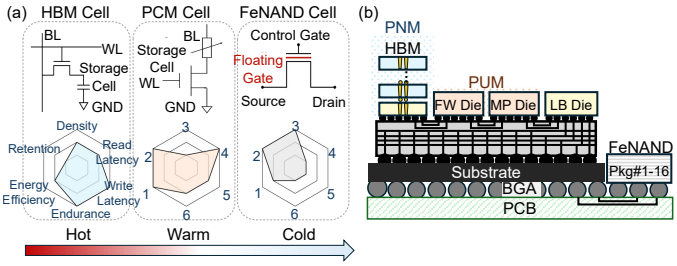

Fig. 5. Heterogeneous PIM memory overview (a) Device-level tradeoffs (b) 2.5D advanced packaging cross-section

B. Heterogeneous PIM Hierarchy

The computational patterns described in motivation above highlight clear workload differences between APSP and S2G graph problems. GEN-Graph addresses this by integrating heterogeneous HBM3, PCM, and FeNAND into a complementary memory hierarchy. Fig. 5 characterizes the heterogeneous memory organization of GEN-Graph. The spider charts in Fig. 5(a) profile the multi-dimensional trade-offs across six hardware features: density, read latency, write latency, energy efficiency, endurance, and retention. These quantitative profiles dictate a tiered memory hierarchy that aligns physical device strengths with the divergent requirements of graph DP.

① **Hot Tier (HBM3):** S2G alignment exhibits latency-critical irregular memory access. HBM3 dominates in write latency and write/read endurance, making it the optimal substrate for the high-frequency updates required by dynamic traversals. Programmable PNM logic on the HBM logic base die exposes this bandwidth directly to the traversal tile to minimize pipeline stalls.

② **Warm Tier (PCM):** APSP features compute-dense updates where energy and data persistence are critical. PCM offers superior energy efficiency and non-volatile retention. These properties enable bit-serial PUM operations without SRAM area overhead or refresh costs.

③ **Cold Tier (FeNAND):** Pangenome graphs and large distance matrices necessitate extreme storage capacity. FeNAND excels in density and non-volatile retention, functioning as the cold tier for persistent storage. It streams static graph structures to upper tiers only when needed.

Fig. 5(b) illustrates the physical integration via 2.5D advanced packaging. A silicon interposer connects the HBM stack and PCM dies into a unified compute package to reduce communication cost. FeNAND attaches via high-speed ONFI channels to maintain a scalable storage footprint. This organization keeps high-reuse data on-package in its most suitable compute medium.

C. System Overview

GEN-Graph resolves the compute–memory divergence in graph DP by matching specialized compute tiles to a shared HBM–PCM–FeNAND hierarchy on a silicon interposer substrate. This co-designed accelerator enables hardware utilization by pinning data-intensive kernels to their optimal memory technology while offloading infrequent tasks to the host CPU. Fig. 6(a) illustrates the 2.5D physical architecture. An HBM3 stack, PCM dies, and a logic base die integrate

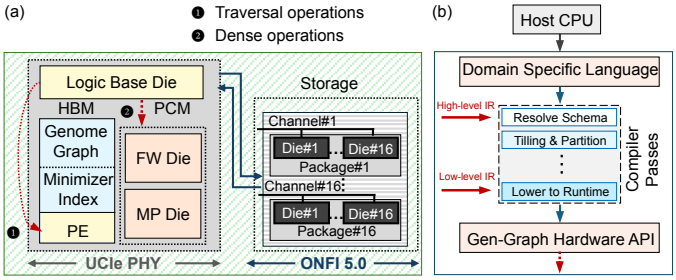

Fig. 6. GEN-Graph heterogeneous PIM accelerator (a) 2.5D physical architecture on the silicon interposer (b) Illustration of compilation flow

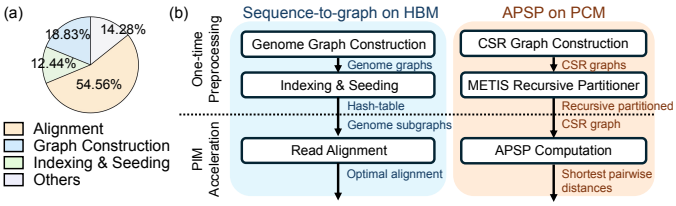

Fig. 7. General graph-based DP (a) S2G runtime breakdown (b) Mapping pipeline: S2G (Left) and APSP (Right)

via a UCIe physical interface. The matrix tile serves as the PCM-based PUM engine for dense APSP updates, while the traversal tile operates as the HBM-based PNM engine for sparse S2G alignment. The logic base die integrates stream engines to orchestrate data movement across the hierarchy and off-package FeNAND via ONFI 5.1.

The system partitions tasks based on computational profiling to maximize execution efficiency. Based on the S2G runtime analysis in Fig. 7(a), it shows that alignment tasks consume over 70% of total runtime, whereas host-side preprocessing overhead remains trivial. As illustrated in the mapping pipeline (Fig. 7(b)), the host CPU handles lightweight one-time preprocessing: genome graph construction, indexing, seeding and METIS [43] partitioning. The steady state DP kernels below the dashed line instead execute on specialized PIM tiles, with S2G mapped to the traversal tile and APSP to the matrix tile. GEN-Graph therefore keeps the dominant work close to the most suitable memory technology while using the host only for light, infrequent tasks.

Fig. 8 details workload-specific dataflows. In Fig. 8(a) the matrix tile processes APSP. Step ① expands CSR encoded components from HBM into dense distance tiles inside the PCM-FW die and runs bit-serial FW entirely in place. Step ② writes updated tiles back to HBM. Step ③ extracts boundary rows and columns on the logic base die and assembles the boundary graph. Step ④ streams boundary tiles into the PCM-MP die, which performs MP reductions on boundary-to-boundary paths. Step ⑤ returns the reduced boundary tiles to HBM and injects them back into component tiles for the next FW phase. After a recursion level completes, Step ⑥ compresses dense tiles back to CSR and Step ⑦ streams them to FeNAND. This streaming schedule overlaps PCM computation with HBM-side boundary construction and FeNAND I/O, enabling APSP to operate at high sustained throughput on the matrix tile. In Fig. 8(b) the traversal tile executes S2G alignment. The full pangenome graph and minimizer index reside in FeNAND. During initialization, Step ① streams

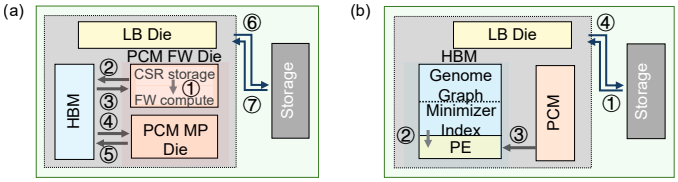

Fig. 8. GEN-Graph workload dataflow (a) Matrix tile (b) Traversal tile

these structures through the logic base die, builds a working copy in HBM and pins highly reused graph fragments and index buckets in PCM as a non-volatile cache. For each batch of reads, Step ② runs the bit vector alignment kernel on PEs attached to HBM channels, using subgraphs stored in HBM and the precomputed seeds. When a batch needs cached subgraphs, Step ③ fetches the corresponding subgraph from PCM into HBM and forwards it to the PEs so hotspots bypass FeNAND. After a batch completes, Step ④ writes compact alignment results or intermediate states from HBM back to FeNAND when long term storage is required.

Finally, a domain-specific compiler drives the hardware (Fig. 6(b)) by translating domain-specific language (DSL) on the host into accelerator commands. The compiler first translates the program into a high-level intermediate representation (IR), then resolves schemas and applies key transformations such as tiling and recursive partitioning to match the hardware tiles. Next, it lowers the optimized IR into a runtime-level representation and emits GEN-Graph hardware API calls that drive execution on the heterogeneous PIM accelerator. The generated runtime calls select the appropriate tile, allocate buffers in HBM or PCM, and orchestrate data transfers between FeNAND and compute tiles to optimize data locality.

D. Matrix Tile: APSP Optimized Compute Tile

The matrix tile scales APSP by partitioning large graphs into 1024-vertex clusters that match physical PIM capacities (Section IV-D1). In Section IV-D2, we describe the hardware architecture and specialized logic units where PCM-FW dies execute data permutations and PCM-MP dies accelerate min-reductions. Finally, tailored mapping and scheduling schemes drive bit-serial execution to saturate near-memory bandwidth and maximize parallel throughput (Section IV-D3).

1) Co-Designed 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. 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. 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 [43] 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 $C_i^{(0)}$

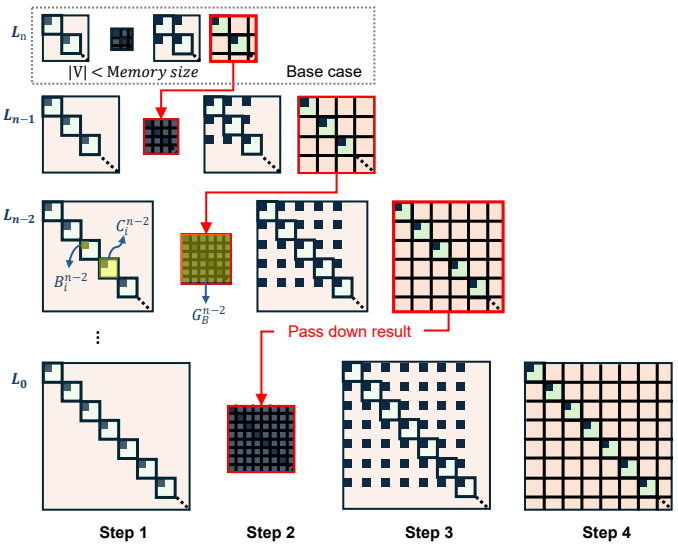

Fig. 9. Illustration of Recursive Partition APSP

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 $n \in C_2$

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. 9 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 structure maximizes tile-level parallelism on the `Main_Block` and fits within constrained PIM resources without requiring global synchronization. The detailed mapping and scheduling schemes for this process are described in Section IV-D3.

2) Matrix Tile Hardware Design and Implementation: Both PCM dies use the same 1T1R SLC PCM cell technology [51], organized into 1024×1024 units. Each unit, together with its peripheral circuits, forms a macro, as shown in Fig.10(a). Each tile contains 130 parallel units connected via an H-tree interconnect [52] for efficient data exchange (Fig. 10(b)). The design ensures full crossbar activity without idle cycles. Matching arrow colors denote concurrent unit-to-unit transfers. Tiles operate concurrently, each with a local controller linked to a global main controller for coordinated execution (Fig. 10(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 integrates a 32-bit 6-level min-comparator tree for efficient MP reductions.

Algorithm 2 Recursive Partition APSP Pseudocode

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

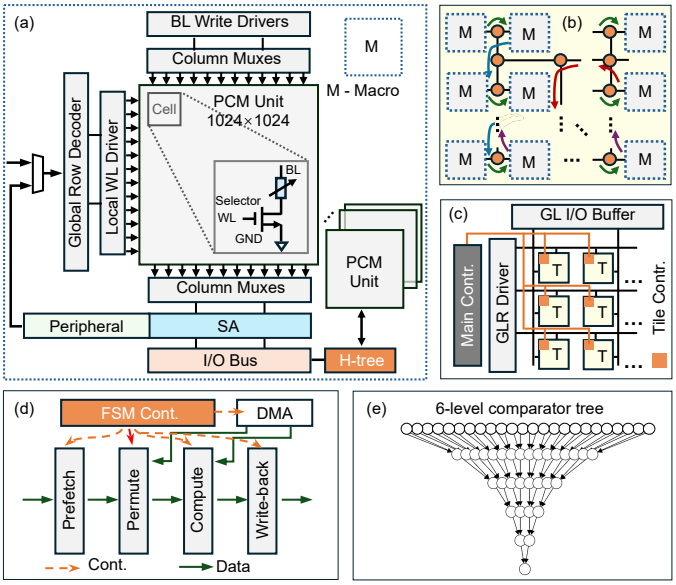

Fig. 10. Matrix tile detail architecture (a) Macro (M) (b) Tile (T) (c) Die (d) Permutation unit (e) 6-level min comparator tree

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. 10(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 → Permute → Compute → 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 skips futile writes, sustaining near-peak occupancy and reducing wear.

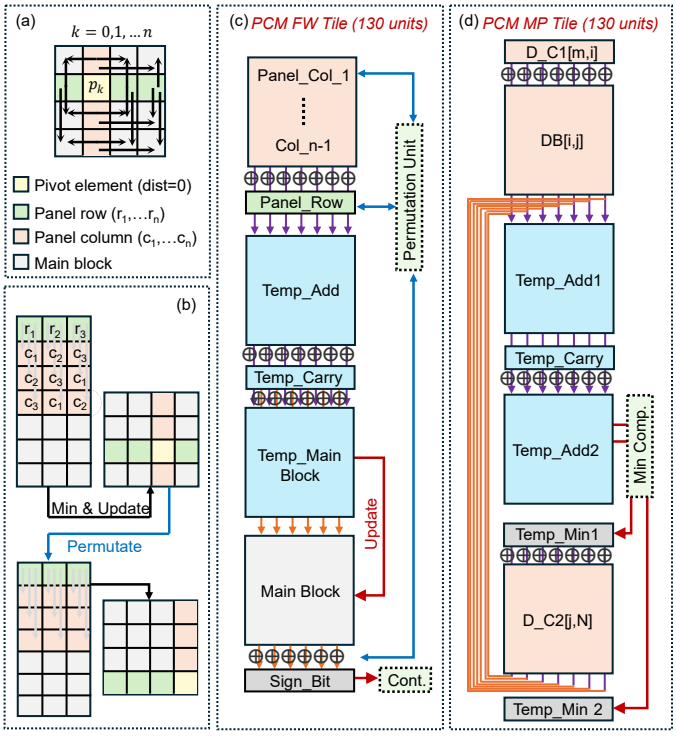

Fig. 11. SW/HW mapping for recursive partitioned APSP (a) FW illustration (b) FW remapping, every step includes add, min update and permutation (c) PCM-FW tile performing FW (d) PCM-MP tile performing two-stage MP

PCM-MP Min-Compare & Update. Each unit integrates a pipelined comparator tree (Fig. 10(e)) that reduces 1024 32-bit inputs to one 32-bit minimum in following steps: A 1024 × 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. The final sign-bit mask gates PCM writes, updating only entries with smaller values. Two staging buffers hold operands across adds while $DB[i,j]$ streams, enabling 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.

3) APSP Mapping and Scheduling Scheme: 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. 11(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 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. 11(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 permuted, 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. 11(c) shows the PCM-FW datapath. Add is implemented with FELIX [34] bit-serial adders using `Temp_Add` and `Temp_Carry`. A FELIX [34] 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 IV-D2. Fig. 11(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`.

E. Traversal Tile: S2G Optimized Compute Tile

The traversal tile co-designs the windowed bit-parallel algorithm to resolve the complex topological dependencies (Section IV-E1). To mitigate memory bottlenecks, the hardware integrates a tiered storage hierarchy on HBM3 logic dies to stage alignment states and pattern masks (Section IV-E2). This organization sustains high utilization via an adaptive mapping scheme that reconfigures processing elements (PEs) for high-throughput parallel execution of short reads or deep pipelining for long sequences (Section IV-E3).

1) Sequence-to-Graph Alignment: The traversal tile implements S2G alignment using a windowed bit-parallel DP formulation. It represents the DP wavefront using dense bit vectors and expresses state updates as simple bitwise operations such as AND, OR, and shifts.

As detailed in Algorithm 3, the process begins by partitioning the query Q into k segments Q_i to match the hardware vector width W. For each segment, precomputed match masks $\mathcal{M}$ are generated for the current window. The inner loop processes graph nodes v in topological order to resolve two-dimensional dependencies. First, spatial dependencies are resolved by aggregating predecessor state vectors $\mathcal{S}[u]$ into $\vec{D}_{in}$ via bitwise OR ($\bigvee$) operations. Second, temporal dependencies across windows are resolved by computing the updated state $\vec{D}_v$ through a bitwise shift of $\vec{D}_{in}$ and the injection of a carry-in bit c_{in}. This c_{in} is initialized to 1 for the first segment and retrieves the carry bit $C[v]$ from the previous segment otherwise. Finally, the most significant bit (MSB) of $\vec{D}_v$ is stored in $C[v]$ for the next segment iteration, while $\mathcal{S}[v]$ is updated to record the alignment state and track the global $Score_{max}$. Windowed bit-parallelism reduces the computational complexity from $O(N \cdot M)$ to $O(N \cdot M/W)$,

TABLE II
KEY VARIABLES IN WINDOWED S2G ALIGNMENT

Variable	Description
Q, W	Query sequence and vector width ($W = 128$ bits)
k	Number of window segments ($k = \lceil \|Q\|/W \rceil$)
Q_i	The i-th query segment ($1 \le i \le k$)
$\mathcal{M}$	Precomputed match bit-masks for segment Q_i
$\mathcal{S}[v]$	Alignment state bit-vector for graph node v
$\vec{D}_{in}$	Aggregated predecessor state (Spatial Dependency)
$C[v]$	Carry bit for node v (Temporal Dependency)
$Score_{max}$	The global maximum alignment score recorded

Algorithm 3 Windowed Bit-Parallel S2G Alignment

Input: Graph $G(V, E)$, Query Q, Window Width W
Output: $Score_{max}$

$k \leftarrow \lceil |Q|/W \rceil$; Initialize $\mathcal{S}[v] \leftarrow \vec{0}, C[v] \leftarrow 1 \ \forall v \in V$
for $i \leftarrow 1$ **to** k **do** ▷ Outer loop: Query segments
  $\mathcal{M} \leftarrow \text{PrecomputeMasks}(Q_i)$
  for each $v \in V$ **in Topological Order do**
    Step 1: Spatial Dependency Aggregation
    $\vec{D}_{in} \leftarrow \bigvee_{u \in \text{Pred}(v)} \mathcal{S}[u]$ ▷ Resolve graph topology
    Step 2: Bit-Parallel State Update
    $c_{in} \leftarrow (i == 1)?1 : C[v]$ ▷ Inject inter-window carry
    $\vec{S}_{new} \leftarrow ((\vec{D}_{in} \ll 1) \vee c_{in}) \wedge \mathcal{M}[\text{char}(v)]$
    Step 3: State and Carry Preservation
    $\mathcal{S}[v] \leftarrow \vec{S}_{new}$
    $C[v] \leftarrow \text{GetMSB}(\vec{S}_{new})$ ▷ Store for next window
    Step 4: Score Logging
    if $\vec{S}_{new} \neq \vec{0}$ **then** Update $Score_{max}$
return $Score_{max}$

providing the throughput necessary for pangenome-scale alignment. Table II summarizes the key variables. By decoupling control flow from data movement, this organization enables parallel window execution.

2) Traversal Tile Hardware Design and Implementation: The traversal tile resides on the logic base die of the HBM3 stack and couples bit-parallel computation with bank-level parallel data management. As shown in Fig. 12(a), we attach one PU to each of the 16 independent HBM channels. Each channel provides 8 bank groups with 32 banks and a 64-bit interface. This static one-to-one mapping maximizes local bandwidth utilization and eliminates contention, with cross-channel communication occurring only for subgraphs spanning multiple channels via a lightweight ring router.

Fig. 12(b) details the PU architecture, which contains 64 PEs, a 32 KB input scratchpad, and a 256 KB shared banked SRAM. The input scratchpad stages the linearized reference graph, per-node hop flags, and the pattern bitmasks for the query read, enabling single-cycle streaming into the PE group. To maximize burst locality in HBM, we batch reads by query position. We store the i-th query base and its masks contiguously across the batch, so each PU issues unit-stride HBM bursts. We size the shared banked SRAM at 256 KB to accommodate alignment states for 16 384 nodes per PU. We stripe node states across 32 banks so predecessor gathers spread across banks. Most human variants form short local bubbles, which bounds the active hop working set and increases SRAM hit rate [53]. To minimize conflicts, we partition this SRAM into 32 banks using hashed bank mapping. Section V-C shows 128 KB eliminates most spills, and we use 256 KB to provide headroom for long reads. A 1 KB instruction buffer stores the compact S2G microcode.

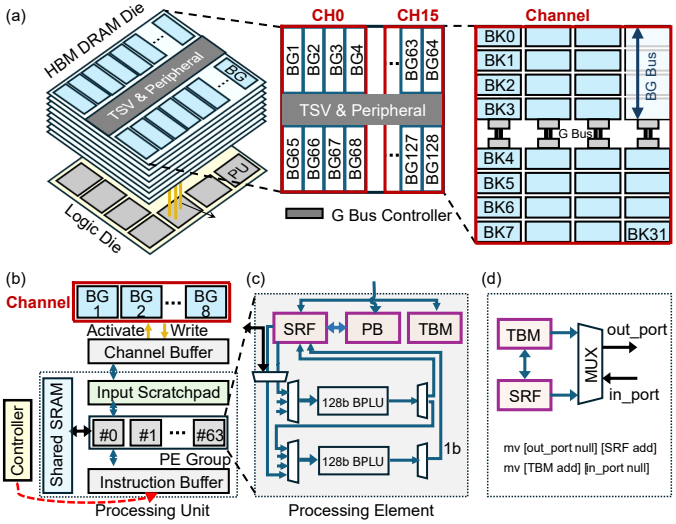

Fig. 12. Traversal tile detail architecture (a) HBM3 organization (b) Processing unit (c) Processing element (d) PE data movement

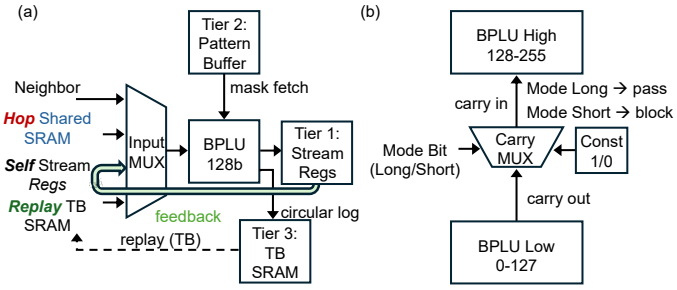

Fig. 13. Refined HBM3 PNM micro architecture design (a) Three-tier storage hierarchy (b) Configurable carry chain

The PE micro-architecture illustrated in Fig. 12(c) centers on dual 128-bit BPLUs supported by a tiered storage hierarchy comprising a stream register file (SRF), a pattern buffer (PB), and a traceback memory (TBM). Fig. 12(d) illustrates the explicit data movement control: micro-instructions `mv` direct data between local storage and external ports.

Inside the PE micro-architecture, a four-way input MUX acts as the centralized data entry point to avoid the overhead of general-purpose register files. As illustrated in Fig. 13(a), this selector retrieves operands from the local feedback path (***Self***), the traceback buffer (***Replay***), the shared banked SRAM (***Hop***), or adjacent PEs (Neighbor). This design allows the PE to switch between regular systolic streaming and irregular graph-driven memory accesses without architectural stalls. The MUX orchestrates data from the following:

- Tier 1 Stream Registers (***Self***). A 384-bit register file holds the working set of current, previous, and dependency vectors. During linear alignment, the MUX selects the ***Self*** port to create a local feedback loop from the BPLU output, sustaining high throughput for regular updates.
- Tier 2 Pattern Buffer. A 256-Byte SRAM provides the pattern masks directly to the BPLU. This capacity allows the hardware to prefetch masks for the next window during current computation to effectively hide load latency.
- Tier 3 Traceback Memory (***Replay***). A 4 KB circular

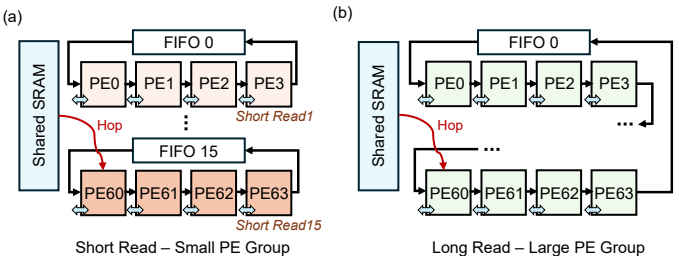

Fig. 14. Workload adaptive mapping scheme (a) Short read parallel mapping (b) Long read pipeline mapping

buffer logs historical direction bits in the background. This capacity covers approximately the same 16 000 steps to accommodate typical segment lengths. When traceback is enabled, the MUX switches to the ***Replay*** port to reconstruct paths without global memory access.

- Shared Banked SRAM (***Hop***). When encountering graph bifurcations, the MUX switches to the ***Hop*** port. The state pauses local feedback to fetch non-adjacent predecessor states from the PU-level SRAM and resolves the irregular dependencies inherent to complex topologies.

Finally, Fig. 13(b) depicts the reconfigurable link between the two 128-bit BPLU cores. A mode-controlled carry MUX configures the blocks as a 256-bit systolic unit for high-precision or dual 128-bit execution units. This reconfigurability maximizes utilization across diverse sequence lengths.

3) Sequence-to-Graph Mapping and Scheduling Scheme: The mapping challenge arises from the structural divergence between linear sequence alignment and graph alignment as shown in Fig. 3. Rigid datapaths with fixed buffering cannot fetch non-local predecessors efficiently in pangenomes. Moreover, hardware must adapt to different read lengths: short reads (50 to 300 bp) require high parallelism to manage repeat regions, while long reads (above 10 kb) demand deep computation and high bandwidth to resolve structural variants. Our strategy resolves this by designing a flexible data flow that adapts the hardware configuration to both the read length and the dependency structure.

Short-Read Parallel Mapping. Short reads on wide pipelines incur high initialization overhead and low utilization. To maximize accelerator throughput, we partition the 64-PE array into 16 independent groups of four PEs as illustrated in Fig. 14(a). Independent groups process unique read streams in parallel while the PU controller distributes subgraphs and query data to their input scratchpads. This turns the traversal tile into a high-throughput parallel engine that effectively amortizes the control cost of fine-grained tasks. In this parallelized mode, the PEs retain access to the shared banked SRAM to resolve local branches via the *Hop* state.

Long-Read Pipeline Mapping. Long reads involve deep dependency chains. In this mode, we configure entire 64-PE group as a single deep pipeline, as depicted in Fig. 14(b). The execution logic remains the same: PEs seamlessly alternate between the *Self* state for linear extensions and the *Hop* state for SRAM-assisted topological traversals. To handle sequences exceeding 10 kb without a linear increase in PE count, the architecture employs a windowed bit-parallel approach with a fixed vector width ($W = 128$). This iterative processing provides the necessary computational depth to navigate complex branches and long-range dependencies without stalling, effectively decoupling the physical hardware scale from the total sequence length.

TABLE III
HARDWARE PLATFORM SPECIFICATIONS

Feature	**Intel i7-11700K** [54]	**NVIDIA A100** [55]	**NVIDIA H100** [56]
Architecture	Rocket Lake	Ampere	Hopper
Max Freq.	5.0 GHz (Turbo)	1.41 GHz (Boost)	1.98 GHz (Boost)
Memory	64 GB DDR4-3200	80 GB HBM2e	80 GB HBM3
Memory BW	50 GB/s	2 039 GB/s	3.35 TB/s
TDP	125 W	400 W	700 W

V. EVALUATION

A. Experimental Setup

1) Baselines: We evaluate GEN-Graph against three classes of baselines: (i) general-purpose CPU and GPU platforms, (ii) prior S2G accelerators, and (iii) prior large-scale APSP accelerators. We use an Intel i7-11700K [54], an NVIDIA A100-SXM4 [55], and an NVIDIA H100-SXM5 [56]. Table III summarizes key parameters.

For the APSP task, we compare against CPU, GPUs, SOTA PIM method temporal PIM SSSP [57], and SOTA GPU distributed methods including Partitioned APSP [27] and Co-Parallel APSP [28]. Since no SOTA PIM methods directly implement APSP, we estimate the performance of the temporal PIM SSSP [57] to establish a comparable APSP PIM baseline, hereafter referred to as PIM-APSP. For the S2G alignment task, we evaluate the traversal tile against representative implementations across SOTA CPU, GPU, and SOTA accelerator platforms. On CPUs and GPUs, we use SOTA Smith–Waterman based engines, including PASGAL [58] and HGA [29], as algorithmic baselines executed on general-purpose hardware. We further compare against two specialized accelerators with distinct design paradigms, namely SeGraM [39], which follows a PIM approach, and ASGDP [48], which adopts a task-specific ASIC design.

2) Datasets: To evaluate GEN-Graph across diverse compute domains, we employ SOTA large-scale datasets for the different compute domains of GEN-Graph: For network analysis workload, we profile the real-world performance using the OGBN-Products graph (2.45M nodes) [59], while architectural sensitivity to topology is assessed via synthetic Newman–Watts–Strogatz (NWS) [60] and Erdős–Rényi (ER) [61] graphs generated by NiemaGraphGen [62]. For genomic workloads, we further utilize the GRCh38 [63] human genome augmented with Genome in a Bottle (GIAB) variations. We then generate a comprehensive read suite using PBSIM2 [64] for long reads (PacBio [45], ONT [46]) and Mason [65] for short reads (Illumina [44]), covering a wide range of error profiles. Benchmarks include 100 bp short-reads (LRC-L1: 317.6k reads, MHC1-M1: 497.0k reads) and 10 kbp long-reads (LRC-L2: 3.2k reads, MHC1-M2: 4.9k reads).

3) Evaluation Platform: We developed an in-house cycle-accurate simulator to model the GEN-Graph, validated against RTL synthesis results. With the specialized compiler that maps high-level graph algorithms to tile-specific instructions, the

TABLE IV
GEN-GRAPH HARDWARE COMPONENT SUMMARY

Component	Key Feature	Function
On-Interposer (2.5D Package)		
HBM3	16 GB capacity; 8-Hi DRAM stack	Traversal computation
PCM-FW Die	2 GB capacity; on-tile permutation	PIM-based FW computation
PCM-MP Die	2 GB capacity; on-tile min-tree	PIM-based MP merge
Logic Base Die	Dual 64 KB stream engines	Control, dataflow, format conversion
On-PCB		
FeNAND	16 TB capacity; ONFI 5.1	Dense persistent storage

TABLE V
HARDWARE CONFIGURATIONS OF SLC PCM [51]

SLC PCM Die Parameters			
Read Energy/bit	0.05 pJ	Write Energy/bit	0.56 pJ
Read/Write Latency	2 ns/20 ns	Die Capacity	2 GB
Clock Cycle	2 ns (500 MHz)	Reset Pulse	40 µA @ 0.70 V
Resistance (LRS)	≈ 30 kΩ	On/Off Ratio	150×
PCM Organization			
Organization	1024 by 1024 cells per unit; 130 units per tile; 128 tiles per die.		

simulator captures the power and performance from the fully end-to-end flow. For the PCM-based matrix tile, we integrate device-level parameters from NeuroSim [66] to model analog array operations. For the HBM-based traversal tile, we synthesize the logic die processor using Synopsys Design Compiler with a 28 nm technology library. To enable a fair comparison with the 7 nm GPU baselines, we scale all area and power results to the 7 nm node using established scaling models [67].

4) GEN-Graph configuration: Table IV lists the GEN-Graph configuration, which integrates HBM3-backed traversal tile and PCM-based matrix tile on a 2.5D interposer, with FeNAND providing off-package capacity for large graphs and persistent data. Table V summarizes the PCM in-memory processor used by the matrix tile. Each tile operates on 1024×1024 arrays, enabling fully in-place execution. Table VI summarizes the HBM3 near-memory processor used by the traversal tile: 16 HBM channels, each with 32 banks, provide high bandwidth for irregular accesses. We assess the impact of these architectural parameters in the scalability tests for both the GEN-Graph matrix tile (Section V-B) and traversal tile (Section V-C).

B. GEN-Graph Performance on APSP Workloads

Fig. 15 shows a comparison of GEN-Graph matrix tile running APSP workloads to CPU and GPU (A100 and H100) baselines using graphs with 100, 1 024, and 32 768 nodes synthesized using NiemaGraphGen [62]. These sizes avoid memory bottlenecks on single-node hardware. At 1 024 nodes, matrix tile 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. Matrix tile's performance gains grow with graph size due to enhanced parallelism and in-memory execution. This performance gap widens dramatically with larger graph sizes because conventional systems are overwhelmed by $O(n^3)$ data movement. This massive data transfer saturates memory bandwidth that matrix tile inherently avoids. Fig. 16 then compares GEN-Graph matrix tile, running APSP workloads, with SOTA accelerator PIM-APSP [57], as

TABLE VI
HARDWARE CONFIGURATIONS OF HBM3 DRAM

HBM3 DRAM Device Parameters			
#layers	8-Hi	Technology Node	10nm (1Ynm or 1Znm)
Bank Capacity	256 Mb	Bank Area	0.219 mm^2
Channel Row Buffer	16 Kb	Read/Write Energy/bit	0.4 pJ / 0.45 pJ
Chip Area	121 mm^2	Read/Write Latency/ns	10-20 ns
HBM3 System Parameters			
Organization	16 channels per chip (64-bit I/O per channel); 32 banks per channel.		
Processing Element (PE)			
Compute Unit Array	Dual-BPLU (2×128b)	Stream Register File	384b
Pattern Buffer	256 B	Traceback Memory	4 KB
Processing Unit (PU)			
#PEs	64	Shared Banked SRAM	256 KB
Instruction Buffer	1KB	Ring Router	128 GB/s/link
HBM3 DRAM NMP Processor			
Basic	28 nm process; 0.7 V supply; 113 mm^2 die area; INT8/INT32 format		
#PUs	16	SRAM Capacity	4 MB (Distributed)
Peak Bandwidth	819.2 GB/S	Peak Power	8.52 W

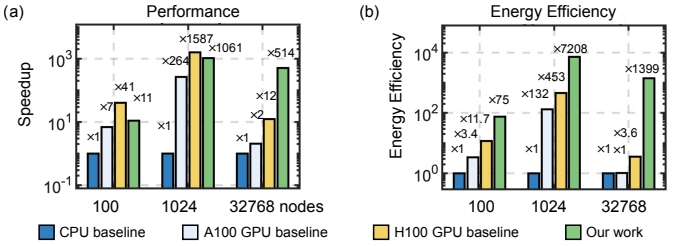

Fig. 15. GEN-Graph matrix tile vs. CPU, A100 GPU and H100 GPU baselines across graph sizes (a) Speedup (b) Energy efficiency

well as with GPU cluster methods – Partitioned APSP [27] and Co-Parallel APSP [28]. On OGBN-Products (2.45M nodes) dataset [59], we estimate their performance from reported scaling trends. Matrix tile outperforms both, achieving $5.8\times$ speedup over Co-Parallel APSP [28] and $1\,186\times$ energy savings over Partitioned APSP [27]. While PIM-APSP improves energy efficiency by $11.4\times$, it slows down performance to 70% of the baseline. The advantage of matrix tile comes from removing inter-GPU communication at large-scale, multi-node APSP solutions.

Scalability Analysis of Matrix Tile. We compare the scalability of GEN-Graph matrix tile and the H100 GPU baseline by varying graph degrees, size, and topology. In Fig. 17(a) and (d), both systems maintain stable performance as a function of degree, suggesting edge count has a limited effect on exact APSP when there is enough memory. In Fig. 17(b) and (e), GEN-Graph matrix tile 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 matrix tile's in-situ computation preserves locality. In Fig. 17(c) and (f), GEN-Graph matrix tile 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. GEN-Graph matrix tile provides competitive scalability without the immense hardware cost and overhead of GPU clusters. Partitioned-FW [27]

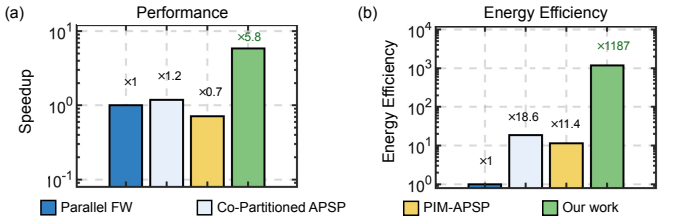

Fig. 16. GEN-Graph matrix tile vs. PIM-APSP [57], Partitioned APSP [27], and Co-Parallel APSP [28] running APSP on OGBN-Products dataset [59] (a) Speedup (b) Energy efficiency

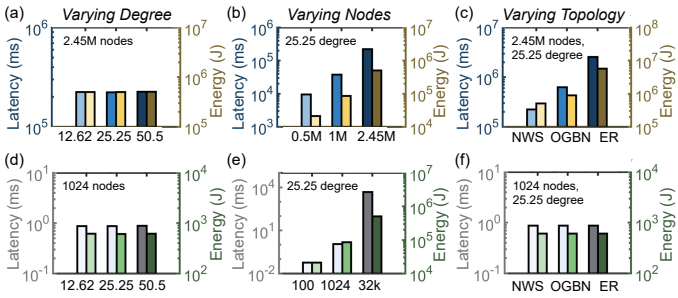

Fig. 17. Scalability of matrix tile across (a) Degree (b) Size and (c) Graph topologies; and H100 across (d) Degree (e) Size and (f) Graph topologies. Topologies include clustered (NWS), real (OGBN), and random (ER)

uses 2 560 GPUs for a 1.9M-node graph but hits synchronization and memory walls. Co-ParallelFW [28] achieves only 45% weak-scaling efficiency on a 300K-node graph. Overall, GEN-Graph matrix tile avoids the complexity, synchronization, and energy overheads of distributed GPU clusters.

We further compared to prior GPU-based APSP systems, GEN-Graph matrix tile offers competitive scalability with far lower hardware cost. Partitioned-FW [27] scales to 2 560 GPUs and solves 1.9M-node graphs in six minutes, but faces synchronization and memory bottlenecks at scale. Co-ParallelFW [28] achieves $4.6\times$ speedup from 16 to 256 GPUs per node with 45% efficiency on 300K nodes via weak scaling. In summary, GEN-Graph matrix tile leverages topology awareness to scale efficiently, avoiding the synchronization and energy penalties that restrict distributed GPU clusters.

Sensitivity Analysis of Matrix Tile. We examine how the effective GEN-Graph matrix tile capacity N affects end-to-end APSP performance on OGBN-Products. In Fig. 18, we vary $N \in \{256, 512, 1024, 2048\}$ and normalize latency and energy to the $N = 1024$ design point. The sensitivity curve reflects three competing effects: smaller tiles increase recursive partitioning and cross-component merge overhead, while larger tiles reduce on-die parallelism (scaling approximately as $1/N^2$) and incur RC-limited frequency degradation modeled as $(N/1024)^{\alpha}$ with $\alpha = 1.3$. As a result, latency follows a convex trend and is minimized at $N = 1024$ with normalized latency $1.0\times$. In comparison, $N = 256$, 512, and 2048 incur respectively $2.41\times$, $1.28\times$, and $2.29\times$ higher latency. We therefore adopt 1024×1024 PCM arrays as the default matrix tile configuration for the remainder of this work.

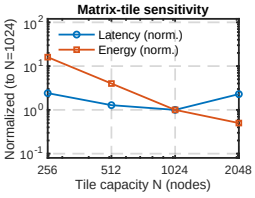

Fig. 18. Sensitivity to matrix tile capacity N.

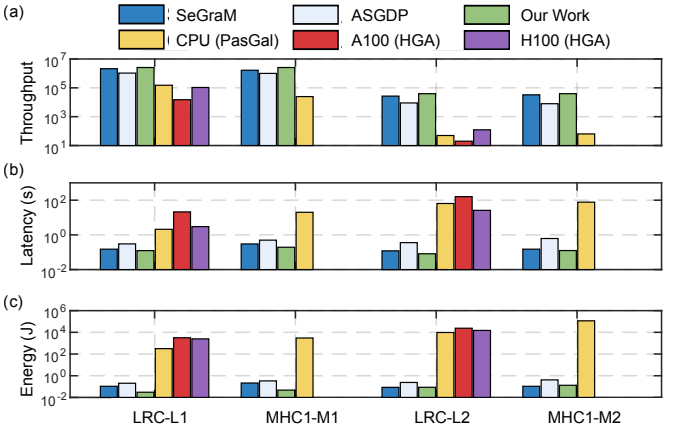

Fig. 19. GEN-Graph traversal tile vs. SOTA CPU, GPU and HW accelerators on S2G including SeGraM [39], ASGDP [48], PasGal [58] and HGA [29] (a) Throughput (b) Latency (c) Energy

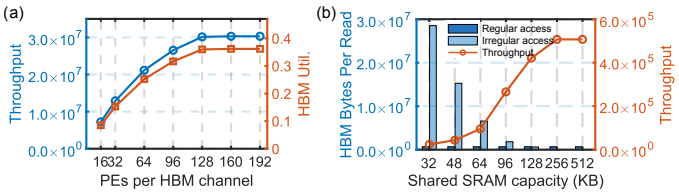

Fig. 20. Sensitivity to (a) PE scalability and (b) Shared SRAM capacity

C. GEN-Graph Performance on S2G Workload

We evaluate the end-to-end throughput of GEN-Graph traversal tile using four representative workloads SeGraM [39], ASGDP [48], PasGal [58] and HGA [29], targeting the short and long read workloads. In Fig. 19(a), the GEN-Graph traversal tile achieves the best throughput across all benchmarks. On the two short read workloads (LRC-L1 and MHC1-M1), it reaches about 2.56 million reads/s and improves throughput over SeGraM [39] by 21% and 55%. It also outperforms ASGDP [48] by $2.44\times$ and $2.56\times$. Here, the gain comes from sustaining higher PU utilization under irregular dependencies by the tiered storage hierarchy. This co-designed approach enables GEN-Graph to sustain high hardware utilization by overlapping in-place computation with hierarchical data streaming.

In Fig. 19(b), the GEN-Graph traversal tile is $171\times$ faster on LRC-L1 and $1963\times$ faster on LRC-L2, while reducing energy by 1.05×10^5 and 2.79×10^5 compared to same workloads running on A100 GPU [55]. The GPU runtime grows sharply on long reads because DP tables expand and exceed on-chip cache capacity. This forces frequent global memory accesses and exposes DRAM latency. GEN-Graph avoids this mode by storing the live dependency state in SRAM and executing the inner DP loop on the logic die. Fig. 19(c) shows that GEN-Graph also improves energy efficiency. It reduces energy versus SeGraM [39] by $3.53\times$ (LRC-L1) and $4.51\times$ (MHC1-M1), and versus ASGDP [48] by $6.72\times$ and $7.06\times$. This comes from data movement. GPU executions repeatedly move DP tables through high-power memory and interconnects.

GEN-Graph keeps the DP working set on-package and streams inputs through a narrow, predictable path.

Scalability Analysis of Traversal Tile. Fig. 19(b) exposes the performance comparison across different read length workloads include 100 bp short-reads (LRC-L1: 317.6k reads, MHC1-M1: 497.0k reads) and 10 kbp long-reads (LRC-L2: 3.2k reads, MHC1-M2: 4.9k reads). For 100 bp reads, the compiler partitions PUs into independent parallel groups and sustains 2.56 million reads/s on LRC-L1. When scaling to long reads in LRC-L2, it switches to a deep pipeline and sustains 39.3 thousand reads/s on LRC-L2. In contrast, the GPU baselines show exponential latency growth on long reads, suffering from limited on-chip cache capacity. This shows GEN-Graph's configurable mapping strategy maintains high utilization across varied read lengths.

Sensitivity Analysis of Traversal Tile. We evaluate the sensitivity of the traversal tile to PE density and local memory capacity to validate our allocation strategy on the logic die.

PE Density Effects. Fig. 20(a) sweeps the number of PEs per HBM channel from 16 to 192 with a mixed workload of short and long reads. The left axis in blue shows aggregate throughput, while the right axis in red shows HBM bandwidth utilization. Aggregate throughput increases from 7.2M reads/s at 16 PEs to 21.2M reads/s at 64 PEs, corresponding to a $2.9\times$ improvement. Increasing the density to 96 PEs further raises throughput to 26.5M reads/s. Between 128 and 192 PEs, throughput saturates near 30.2M reads/s, improving by less than 1.1% despite a $1.5\times$ increase in PE count. This saturation coincides with HBM bandwidth utilization increasing from 0.09 at 16 PEs to approximately 0.36 at 128 PEs, and remaining nearly constant thereafter. The selected PE density 64 therefore lies near the knee of the throughput saturation curve, capturing most attainable performance without over-provisioning compute resources.

Shared SRAM Capacity Effects. Fig. 20(b) sweeps the shared SRAM from 32 KB to 512 KB using long reads. The bars on the left axis in blue report HBM bytes per read split into topology streaming (regular) and traversal accesses (irregular), while the right axis in red plots kernel-only throughput versus shared SRAM capacity. We use long reads to expose memory-bound scaling, since short-read dependencies usually fit within minimal local storage. At 32 KB, limited capacity causes dependency state spills that generate 29.2 MB of HBM traffic and restrict throughput to 23.9K reads/s. Expanding the capacity to 128 KB reduces total traffic to 1.3 MB by dropping state spills from 28.5 MB to 0.62 MB. This results in a $17.6\times$ throughput gain to 421K reads/s. Topology streaming stays constant at 0.68 MB. Once state spills remain trivial, throughput saturates at 507K reads/s for both 256 KB and 512 KB. This trend indicates that 128 KB shared banked SRAM is sufficient to store the active working set; beyond that, performance is bounded by topology streaming, not by capacity misses.

D. Area and Power Analysis

GEN-Graph achieves these gains while preserving exact DP updates, with no pruning or approximation. Across our workloads, it exceeds A100 [55] by $42.8\times$ in APSP speed and exceeds H100 [55] by up to $1963\times$ in S2G latency, while reducing energy by up to 2.79×10^5.

TABLE VII
AREA AND POWER BREAKDOWN BY TILE COMPONENT

Matrix Tile (PCM Unit Level)				
Component	**PCM-FW Unit**		**PCM-MP Unit**	
	Area (μm^2)	**Power (mW)**	**Area (μm^2)**	**Power (mW)**
PCM Subarray	3 288 (13.8%)	557 (80.6%)	3 288 (13.6%)	557 (80.6%)
Permutation Unit	917.3 (3.9%)	0.59 (<0.1%)	—	—
Min Comparator	—	—	1 268 (5.3%)	0.68 (0.1%)
Controller	5.9 (<0.1%)	<0.01 (<0.1%)	5.9 (<0.1%)	<0.01 (<0.1%)
Peripherals*	19 610 (82.3%)	133.3 (19.3%)	19 610 (81.1%)	133.3 (19.3%)
PCM Total	**23 821**	**690.9**	**24 172**	**691.0**

Traversal Tile (HBM Logic Die - Per PU)		
Component	**Area (μm^2)**	**Power (mW)**
PE Array (64 PEs)	24 400 (49.08%)	18.4
Shared SRAM	22 400 (44.99%)	19.4
Instr. Buffer	91 (0.18%)	1.36
Input Scratchpad	2 860 (5.75%)	2.91
PU Total	**50 000**	**41**

*Peripherals include row drivers, sense amplifiers, and global wires.

Area and Power Breakdown. Table VII details the physical parameters of the compute tiles based on synthesis and simulation results. In the matrix tile, peripherals, including row drivers, sense amplifiers, and global wires, occupy over 81% of area to meet PCM operation constraints. During updates, PCM subarrays dominate power and account for 80.6% of consumption. In the traversal tile, the PE array and shared SRAM account for 94% of area. Power splits between compute (18.4 mW) and local memory access (19.4 mW), consistent with a state-heavy inner loop. The full system integrates these tiles with HBM3 (16 GB) adds 8.6 W and $121\,mm^2$ [68]; 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 ~ 18.5 W remains significantly lower than high-end GPUs such as the NVIDIA H100 [56], which consume up to 700 W under peak workloads [69] with higher latency.

Overhead Analysis. We evaluate the hardware cost of specialized computation by isolating logic area and power against the baseline memory arrays. Integrating the permutation unit and min-comparator adds only 3.9% and 5.3% to the unit area, respectively. The power overhead for this logic remains below 0.1% compared to the energy required for PCM cell operations. On the traversal tile, the 16 PUs utilize available silicon space on the HBM logic die without increasing the package footprint. By isolating state feedback, mask prefetching, and traceback into specialized buffers, the architecture reduces the area and power overhead associated with complex address generation. Overall, GEN-Graph provides accelerator-level throughput with array-dominated area and power.

VI. CONCLUSION

GEN-Graph resolves the fundamental compute-memory divergence in graph-based DP through a heterogeneous 2.5D PIM architecture. By matching hardware specialization to algorithmic structure, GEN-Graph integrates high-density PCM-based matrix tiles for compute-bound patterns and low-latency HBM-based traversal tiles for memory-bound genomic traver-

sals. Our hardware-software co-design utilizes recursive partitioning and reconfigurable bit-parallel logic to ensure exact computation while maximizing resource utilization across billion-node graphs. Our compiler-driven mapping further maximizes hardware occupancy by overlapping bit-serial PCM updates with hierarchical data streaming. Evaluation results demonstrate that GEN-Graph achieves a $42.8\times$ speedup and $392\times$ energy efficiency improvement over the NVIDIA H100 GPU for APSP, while exceeding prior PIM accelerators Temporal PIM SSSP [57] by $8.3\times$ and $104\times$. For S2G alignment, the architecture sustains a throughput of up to 2.56 million reads/s, outperforming SeGraM by up to 55% and SOTA ASIC ASGDP [48] by up to $2.56\times$. GEN-Graph establishes a scalable foundation for exact DP acceleration in large-scale genomic and network applications.

ACKNOWLEDGMENT

This work was supported by PRISM and CoCoSys—centers in JUMP 2.0, an SRC program sponsored by DARPA (SRC grant number - 2023-JU-3135); the U.S. DOE DeCoDe project #84245 at PNNL; and NSF grants #2003279, #1911095, #2112167, #2052809, #2112665, #2120019, #2211386.

REFERENCES

[1] A. Rajaraman and J. D. Ullman, *Mining of massive datasets*. Autoedicion, 2011.

[2] 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.

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

[4] B. Paten, A. M. Novak, J. M. Eizenga, and E. Garrison, "Genome graphs and the evolution of genome inference," *Genome research*, vol. 27, no. 5, pp. 665–676, 2017.

[5] A. Ameur, "Goodbye reference, hello genome graphs," *Nature biotechnology*, vol. 37, no. 8, pp. 866–868, 2019.

[6] G. Rakocevic, V. Semenyuk, W.-P. Lee, J. Spencer, J. Browning, I. J. Johnson, V. Arsenijevic, J. Nadj, K. Ghose, M. C. Suciu *et al.*, "Fast and accurate genomic analyses using genome graphs," *Nature genetics*, vol. 51, no. 2, pp. 354–362, 2019.

[7] J. M. Eizenga, A. M. Novak, J. A. Sibbesen, S. Heumos, A. Ghaffaari, G. Hickey, X. Chang, J. D. Seaman, R. Rounthwaite, J. Ebler *et al.*, "Pangenome graphs," *Annual review of genomics and human genetics*, vol. 21, no. 1, pp. 139–162, 2020.

[8] D. Cattaruzza, N. Absi, D. Feillet, and J. González-Feliu, "Vehicle routing problems for city logistics," *EURO Journal on Transportation and Logistics*, vol. 6, no. 1, pp. 51–79, 2017.

[9] 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 Syst. Mag.*, vol. 21, no. 2, pp. 48–74, 2021.

[10] Stanford Network Analysis Project (SNAP), "Stanford large network dataset collection," https://snap.stanford.edu/data/, 2008–present, accessed: 2026-02-19.

[11] M. Jain, S. Koren, K. H. Miga, J. Quick, A. C. Rand, T. A. Sasani, J. R. Tyson, A. D. Beggs, A. T. Dilthey, I. T. Fiddes *et al.*, "Nanopore sequencing and assembly of a human genome with ultra-long reads," *Nature biotechnology*, vol. 36, no. 4, pp. 338–345, 2018.

[12] J. M. Zook, D. Catoe, J. McDaniel, L. Vang, N. Spies, A. Sidow, Z. Weng, Y. Liu, C. E. Mason, N. Alexander *et al.*, "Extensive sequencing of seven human genomes to characterize benchmark reference materials," *Scientific data*, vol. 3, no. 1, p. 160025, 2016.

[13] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *CUG*, vol. 19, no. 45-74, p. 22, 2010.

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

[15] R. Bellman, "Dynamic programming," *science*, vol. 153, no. 3731, pp. 34–37, 1966.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[17] E. Garrison, J. Sirén, A. M. Novak, G. Hickey, J. M. Eizenga, E. T. Dawson, W. Jones, S. Garg, C. Markello, M. F. Lin *et al.*, "Variation graph toolkit improves read mapping by representing genetic variation in the reference," *Nature biotechnology*, vol. 36, no. 9, pp. 875–879, 2018.

[18] W.-W. Liao, M. Asri, J. Ebler, D. Doerr, M. Haukness, G. Hickey, S. Lu, J. K. Lucas, J. Monlong, H. J. Abel *et al.*, "A draft human pangenome reference," *Nature*, vol. 617, no. 7960, pp. 312–324, 2023.

[19] R. W. Floyd, "Algorithm 97: shortest path," *CACM*, vol. 5, no. 6, pp. 345–345, 1962.

[20] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what COST?" in *HotOS XV*, 2015.

[21] J. Kepner and J. Gilbert, Eds., *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2011.

[22] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *PPoPP*. ACM, February 2013, pp. 135–146.

[23] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software*, vol. 34, no. 3, pp. 1–25, 2008.

[24] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *CACM*, vol. 52, no. 4, pp. 65–76, 2009.

[25] 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.

[26] 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.

[27] 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.

[28] 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 *HPDC*, 2021, pp. 121–131.

[29] Z. Feng and Q. Luo, "Accelerating sequence-to-graph alignment on heterogeneous processors," in *ICPP*, 2021, pp. 1–10.

[30] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH*, vol. 23, no. 1, pp. 20–24, 1995.

[31] 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.

[32] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A modern primer on processing in memory," in *Emerging computing: from devices to systems: looking beyond Moore and Von Neumann*. Springer, 2022, pp. 171–243.

[33] 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.

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

[35] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*, 2016, pp. 1–6.

[36] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin *et al.*, "25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *ISSCC*. IEEE, 2014, pp. 432–433.

[37] G. H. Loh, "3d-stacked memory architectures for multi-core processors," *SIGARCH*, vol. 36, no. 3, pp. 453–464, 2008.

[38] S. H. Seo, J. Kim, D. Lee, S. Yoo, S. Moon, Y. Park, and J. W. Lee, "Facil: Flexible dram address mapping for soc-pim cooperative on-device llm inference," in *HPCA*. IEEE, 2025, pp. 1720–1733.

[39] D. S. Cali, K. Kanellopoulos, J. Lindegger, Z. Bingöl, G. S. Kalsi, Z. Zuo, C. Firtina, M. B. Cavlak, J. Kim, N. M. Ghiasi *et al.*, "Segram: A universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping," in *ISCA*, 2022, pp. 638–655.

[40] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand *et al.*, "Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis," in *MICRO*. IEEE, 2020, pp. 951–966.

[41] Y. Chen, Z. Li, K. Fan, R. Tian, J. Hsu, W. Xu, M. Zhou, and T. Rosing, "Rapid-graph: Recursive all-pairs shortest paths using processing-in-memory for dynamic programming on graphs," *arXiv preprint arXiv:2601.19907*, 2025.
[42] Y. Gu, A. Subramaniyan, T. Dunn, A. Khadem, K.-Y. Chen, S. Paul, M. Vasimuddin, S. Misra, D. Blaauw, S. Narayanasamy *et al.*, "Gendp: A framework of dynamic programming acceleration for genome sequencing analysis," *CACM*, vol. 68, no. 5, pp. 81–90, 2025.
[43] G. Karypis and V. Kumar, "Multilevelk-way partitioning scheme for irregular graphs," *JPDC*, vol. 48, no. 1, pp. 96–129, 1998.
[44] Illumina, Inc., "Illumina: Advancing genomics," https://www.illumina.com/, 2026, date: 2026-01-26.
[45] Pacific Biosciences of California, Inc., "Pacific Biosciences," https://www.pacb.com, 2025, accessed Jan. 2026.
[46] Oxford Nanopore Technologies, "Oxford Nanopore Technologies," https://nanoporetech.com, 2025, accessed Jan. 2026.
[47] J. Hennessy and D. Patterson, "A new golden age for computer architecture: domain-specific hardware/software co-design, enhanced," in *ISCA*, 2018.
[48] G. Zeng, J. Zhu, Y. Zhang, G. Chen, Z. Yuan, S. Wei, and L. Liu, "A high-performance genomic accelerator for accurate sequence-to-graph alignment using dynamic programming algorithm," *TPDS*, vol. 35, no. 2, pp. 237–249, 2023.
[49] A. Madhavan, T. Sherwood, and D. Strukov, "Race logic: A hardware acceleration for dynamic programming algorithms," *SIGARCH*, vol. 42, no. 3, pp. 517–528, 2014.
[50] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "Genax: A genome sequencing accelerator," in *ISCA*. IEEE, 2018, pp. 69–82.
[51] 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.
[52] 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.
[53] . G. P. Consortium *et al.*, "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, p. 68, 2015.
[54] Intel, "Intel Core i7-11700K Processor Specifications," https://www.intel.com/content/www/us/en/products/sku/212047/intel-core-i711700k-processor-16m-cache-up-to-5-00-ghz/specifications.html, 2021, accessed: Oct 1, 2025.
[55] "NVIDIA A100 Tensor Core GPU Datasheet," https://www.nvidia.com/en-us/data-center/a100/, NVIDIA, 2021, accessed: Oct 1, 2025.
[56] "NVIDIA H100 Tensor Core GPU Datasheet," https://www.nvidia.com/en-us/data-center/h100/, NVIDIA, 2022, accessed: Oct 1, 2025.
[57] 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.
[58] C. Jain, S. Misra, H. Zhang, A. Dilthey, and S. Aluru, "Accelerating sequence alignment to graphs," in *2019 IPDPS*. IEEE, 2019, pp. 451–461.
[59] 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 *SIGKDD*, 2019, pp. 257–266.
[60] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.
[61] P. ERDdS and A. R&wi, "On random graphs i," *Publ. math. debrecen*, vol. 6, no. 290-297, p. 18, 1959.
[62] N. Moshiri, "Niemagraphgen: A memory-efficient global-scale contact network simulation toolkit," *GIGAbyte*, vol. 2022, p. gigabyte37, 2022.
[63] Genome Reference Consortium, "Human genome assembly GRCh38.p13," 2019, refSeq/GenBank assembly, accessed Jan. 2026. [Online]. Available: https://www.ncbi.nlm.nih.gov/assembly/GCA_000001405.28
[64] Y. Ono, K. Asai, and M. Hamada, "Pbsim2: a simulator for long-read sequencers with a novel generative model of quality scores," *Bioinformatics*, vol. 37, no. 5, pp. 589–595, 2021.
[65] M. Holtgrewe, "Mason–a read simulator for second generation sequencing data," *Technical Report FU Berlin*, 2010.
[66] X. Peng, 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*, vol. 40, pp. 2306–2319, 2020.
[67] 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.
[68] 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.
[69] W. Luo, R. Fan, Z. Li, D. Du, Q. Wang, and X. Chu, "Benchmarking and dissecting the nvidia hopper gpu architecture," in *IPDPS*. IEEE, 2024, pp. 656–667.

Yanru Chen is currently a Ph.D. student in Electrical and Computer Engineering at the University of California San Diego, La Jolla, CA, USA. She received the M.S. degree in Electronic Information (Intelligent Manufacturing) from Tsinghua University, Beijing, China, in 2024, and the B.E. degree in Electronic Science and Technology from Jilin University, Changchun, China, in 2021. Her research interests include SW/HW co-design, processing-in-memory, in-storage processing, graph analytics, hyperdimensional computing, and database search.

Runyang Tian is currently an M.S. student in Electrical and Computer Engineering at the University of California San Diego, La Jolla, CA, USA. He received the B.E. degree in Microelectronic Science and Engineering from Xi'an Jiaotong University, Xi'an, China, in 2024. His research interests include memory-centric computation and domain-specific accelerators.

Zheyu Li received a B.Sc degree in Electrical Engineering and a Ph.D. in Computer Science and Engineering from Pennsylvania State University, University Park, PA, USA, in 2017 and 2023 respectively. He currently is a postdoc in the Energy Efficiency Lab at the University of California, San Diego (UCSD). His research interests lie in hardware and software acceleration of data-intensive emerging workloads, including bioinformatics applications, brain-inspired computing, near-memory computing, and FPGA-based architectures.

Mahbod Afarin is currently a postdoctoral researcher in the Department of Computer Science at the University of California, San Diego, La Jolla, CA, USA. He received his Ph.D. in Computer Science from the University of California, Riverside, CA, USA, in 2024, and his M.Sc. in Computer Engineering from Sharif University of Technology, Tehran, Iran, in 2018. His research interests include hardware accelerators, compiler optimization, and graph analytics.

Weihong Xu is currently a Postdoc in École polytechnique fédérale de Lausanne. He received PhD degree in Computer Engineering at the University of California San Diego, La Jolla, CA, USA. Before joining UCSD, he received B.E. and M.E. degrees from Southeast University. His research focuses on next-generation computing architectures for efficient and reliable AI systems, including: a) RISC-V-based AI chip, b) computer architecture and EDA co-design, c) near-data computing, d) LLM and AI acceleration.

Tajana Rosing received her Ph.D. degree from Stanford University, Stanford, CA, USA, in 2001. She is a Professor, a Holder of the Fratamico Endowed Chair, and the Director of System Energy Efficiency Laboratory, University of California at San Diego, La Jolla, CA, USA. From 1998 to 2005, she was a full-time Research Scientist with HP Labs, Palo Alto, CA, USA, while also leading research efforts with Stanford University, Stanford, CA, USA. She was a Senior Design Engineer with Altera Corporation, San Jose, CA, USA. She is leading a number of projects, including efforts funded by DARPA/SRC JUMP 2.0 PRISM program with focus on design of accelerators for analysis of big data, DARPA and NSF funded projects on hyperdimensional computing, and SRC funded project on IoT system reliability and maintainability. Her current research interests include energy-efficient computing, cyber–physical, and distributed systems.