Octopus: Efficient Hypergraph Pattern Mining with Practical Processing-in-Memory Architecture

YI ZHANG[*†], Huazhong University of Science and Technology, China
DETING CHEN[*‡], Huazhong University of Science and Technology, China
YU HUANG[*‡§], Huazhong University of Science and Technology, China
CHAOQIANG LIU[*†], Huazhong University of Science and Technology, China
HAIFENG LIU[*†], Huazhong University of Science and Technology, China
JIANHUI YUE, Michigan Technological University, USA
XIAOFEI LIAO[*†], Huazhong University of Science and Technology, China
HAI JIN[*†], Huazhong University of Science and Technology, China

Hypergraph pattern mining (HPM) is a key analytical primitive for discovering higher-order relationships in complex data. Despite decades of algorithmic progress, existing systems remain far from saturating available compute resources, as the memory-bound and irregular nature of hypergraph workloads severely constrains sustained throughput. Our empirical characterization shows that even *state-of-the-art* (SOTA) HPM systems achieve only a small fraction of their theoretical peak performance on real workloads.

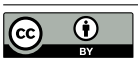
In this paper, we propose Octopus, the first full-stack hardware-software co-designed system for accelerating HPM on practical *processing-in-memory* (PIM) hardware. Octopus targets UPMEM, an emerging commercially available PIM platform that integrates thousands of lightweight in-memory compute units within standard DRAM modules. To fully exploit UPMEM's massive parallelism and bandwidth potential while addressing its stringent architectural constraints, Octopus introduces two tightly integrated components: (i) an inter-DPU coordination framework that orchestrates compact data partitioning and balanced workload distribution across thousands of DPUs, and (ii) an intra-DPU mining engine that enables efficient hyperedge-level candidate generation and asynchronous multithreaded execution. We evaluate Octopus on a real UPMEM platform using diverse real-world hypergraph workloads. Experimental results demonstrate up to 55.37×, 19.80×, 1033.89×, and 795.08× speedups over SOTA solutions HGMatch, OHMiner, Pangolin, and PimPam, respectively.

CCS Concepts: • **Hardware → Emerging architectures**; • **Mathematics of computing → Hypergraphs**; • **Computing methodologies** → *Parallel algorithms*.

Additional Key Words and Phrases: Hypergraph Pattern Mining, Processing in Memory

[*]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology (HUST), Wuhan, 430074, China
[†]Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China
[‡]School of Software Engineering, HUST, Wuhan, 430074, China
[§]Yu Huang is the corresponding author.

Authors' Contact Information: Yi Zhang, Huazhong University of Science and Technology, Wuhan, Hubei, China, yizh@hust.edu.cn; Deting Chen, deting@hust.edu.cn, Huazhong University of Science and Technology, Wuhan, Hubei, China; Yu Huang, yuh@hust.edu.cn, Huazhong University of Science and Technology, Wuhan, Hubei, China; Chaoqiang Liu, chqliu@hust.edu.cn, Huazhong University of Science and Technology, Wuhan, Hubei, China; Haifeng Liu, hfliu@hust.edu.cn, Huazhong University of Science and Technology, Wuhan, Hubei, China; Jianhui Yue, Michigan Technological University, Houghton, USA, jyue@mtu.edu; Xiaofei Liao, xfliao@hust.edu.cn, Huazhong University of Science and Technology, Wuhan, Hubei, China; Hai Jin, hjin@hust.edu.cn, Huazhong University of Science and Technology, Wuhan, Hubei, China.



ACM 2836-6573/2026/6-ART217
https://doi.org/10.1145/3802094

ACM Reference Format:
Yi Zhang, Deting Chen, Yu Huang, Chaoqiang Liu, Haifeng Liu, Jianhui Yue, Xiaofei Liao, and Hai Jin. 2026. Octopus: Efficient Hypergraph Pattern Mining with Practical Processing-in-Memory Architecture. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 217 (June 2026), 27 pages. https://doi.org/10.1145/3802094

1 Introduction

Hypergraphs generalize conventional graphs by allowing a hyperedge to simultaneously connect multiple vertices [1–5]. Unlike ordinary graphs, where edges capture only pairwise relationships, hypergraphs provide a natural representation for higher-order interactions among entities. *Hypergraph pattern mining* (HPM) aims to identify all subhypergraphs, or *embeddings*, within a data hypergraph that are isomorphic to a given pattern hypergraph. HPM has emerged as an essential computational primitive in a broad range of data-intensive domains, including collaborative and biological network analysis [4–8], object detection [9–11], and knowledge discovery in complex relational data [12–14]. For example, protein-protein interaction networks can be modeled as hypergraphs, where vertices represent proteins and hyperedges denote protein complexes [5, 6]. In such settings, biologists query hypergraphs that describe specific molecular complexes and search for matching embeddings within large biological networks, thereby uncovering latent interaction patterns and biological functions.

Since hypergraphs generalize ordinary graphs by enabling edges to connect multiple vertices, a straightforward way to perform HPM is to reduce a hypergraph into an ordinary graph, transform the problem into a conventional *graph pattern mining* (GPM) task, and then apply existing GPM systems [15–23]. However, this transformation causes an exponential blow-up in graph size, leading to excessive storage overhead and significantly increased computational complexity. For instance, converting a hypergraph with 2 million vertices and 15 million hyperedges results in an ordinary graph containing 17 million vertices and nearly 1 billion edges [24].

Consequently, recent studies [8, 25–30] perform HPM directly on hypergraphs, where the search process iteratively extends and validates partial embeddings to progressively enumerate all matches. Depending on the expansion granularity, existing techniques fall into *match-by-vertex* [8, 25–28] and *match-by-hyperedge* [29, 30]. The former expands one vertex per expansion step, while the latter expands one hyperedge at a time. By reducing redundant expansions and backtracking, match-by-hyperedge approaches significantly shrink the search space and enumeration overhead, achieving orders-of-magnitude speedups over vertex-based methods.

Despite algorithmic advances, executing HPM efficiently on conventional architectures remains highly inefficient and difficult to scale. Under von Neumann systems that physically decouple processors from memory modules, performance is inherently constrained by data movement. Profiling the representative match-by-hyperedge system HGMatch [29] on an Intel Xeon Silver 4216 processor reveals a computational intensity of only 0.55 GOPS/Byte and a sustained throughput of 42 GOPS, less than 4% of the processor's theoretical peak. This substantial inefficiency arises from the irregular, data-dependent access patterns of HPM, which result in frequent cache misses, low instruction-level parallelism, and limited effective bandwidth utilization.

Processing-in-memory (PIM) architectures [22, 31–33] offer a promising alternative by physically integrating computation into memory arrays, thereby eliminating the data movement bottleneck and exposing massive internal bandwidth. With the emergence of commercial-grade hardware platforms [34, 35], large-scale deployment of near-memory acceleration has become practically feasible. Among these, UPMEM [34] stands out as the first production-ready PIM system, incorporating thousands of *DRAM Processing Units* (DPUs) within standard DDR4 memory modules. Each DPU executes computation directly within memory, enabling fine-grained parallelism and high bandwidth utilization. In this work, we target UPMEM as our experimental platform and explore

the potential of PIM for accelerating HPM. To the best of our knowledge, this study presents the *first practical implementation of an HPM system on a real, production-grade PIM architecture.*

While promising in principle, deploying hypergraph pattern mining on real PIM hardware remains non-trivial. Recent efforts [22, 31–33, 36] have leveraged the UPMEM platform to accelerate a variety of data-intensive workloads. Among them, PimPam [22] is the most relevant prior work, targeting conventional graph mining on UPMEM. However, directly applying graph-oriented PIM designs to hypergraphs is inherently suboptimal. Converting a hypergraph into an ordinary graph introduces substantial structural expansion and duplication, which significantly increase memory footprint and computational overhead, leading to degraded performance. This inefficiency stems from the fact that the intrinsic characteristics of HPM are fundamentally misaligned with the architectural organization and resource constraints of PIM systems, making it difficult to fully harness the computational potential of UPMEM for hypergraph workloads.

This misalignment manifests across both the inter-DPU and intra-DPU levels. At the inter-DPU level, UPMEM provides thousands of autonomous DPUs, each constrained to 64 MB of local MRAM and lacking hardware support for inter-DPU communication. Consequently, hypergraph data must be decomposed into compact, memory-resident fragments that fit within each DPU's limited MRAM while preserving structural completeness for embedding expansion and achieving balanced distribution across massive DPUs. Moreover, due to the combinatorial and data-dependent nature of HPM, task cost cannot be inferred from input size alone, making accurate load estimation a key challenge for effective inter-DPU coordination. These constraints elevate data management to a first-class design challenge for scaling HPM efficiently on the UPMEM platform. At the intra-DPU level, computation is executed on simple in-order pipelines without cache hierarchy or branch prediction. HPM workloads, dominated by pointer chasing and irregular set intersections, exhibit low arithmetic intensity and frequent control divergence. Consequently, DPUs experience severe pipeline stalls and limited utilization, even with hardware multithreading. Achieving high throughput under such architectural constraints requires rethinking candidate generation and scheduling to expose parallelism and effectively overlap computation with memory access.

To address these challenges, we introduce Octopus, a full-stack hardware-software co-designed system. Octopus aligns the execution characteristics of HPM with the microarchitectural constraints of PIM through two tightly integrated components. The inter-DPU coordination framework orchestrates global data organization and workload distribution across thousands of DPUs, leveraging a load estimation model to perform compact data partitioning and load-balanced task assignment for scalable parallel execution. Complementarily, the intra-DPU mining engine optimizes fine-grained execution within each DPU through hyperedge-level candidate generation and asynchronous dataflow-based multithreaded processing, effectively mitigating control and memory stalls. Together, these components transform HPM from a bandwidth-bound, irregular workload into a structured and scalable computation that fully exploits the potential of UPMEM's PIM architecture.

In summary, this paper makes the following contributions:

- We conduct a comprehensive workload characterization of HPM, identifying the key performance bottlenecks and motivating the adoption of the UPMEM PIM architecture for acceleration.
- We propose Octopus, the first hardware–software co-designed system for accelerating HPM on a real PIM platform. Octopus introduces (i) an inter-DPU coordination framework that performs compact data partitioning and balanced task assignment, for scalable execution across thousands of DPUs, and (ii) an intra-DPU mining engine that enables efficient candidate generation and asynchronous multithreaded execution within each DPU.
- We implement and evaluate Octopus on a real UPMEM system using diverse real-world workloads. Results show that Octopus achieves up to 55.37× and 19.80× speedups over state-of-the-art HPM

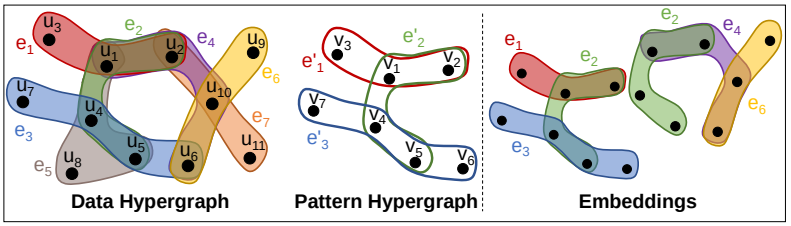


Fig. 1. An example of hypergraph pattern mining problem

systems HGMatch [29] and OHMiner [30], respectively, and up to 1033.89× and 795.08× over leading GPM systems on GPU (Pangolin [18]) and PIM (PimPam [22]) platforms.

2 Background and Motivation

We present the basics of hypergraph pattern mining and the UPMEM architecture, analyze performance bottlenecks, and motivate accelerating hypergraph pattern mining on the UPMEM platform.

2.1 Hypergraph Pattern Mining Preliminaries

Definitions. Unlike ordinary graphs, where edges represent binary relationships between pairs of vertices, hypergraphs generalize this notion by allowing edges, referred to as *hyperedges*, to connect arbitrary subsets of vertices. A hypergraph H is formally defined as a tuple $H = (V, E, L)$, where V is the set of vertices, E is the set of hyperedges, and L is a labeling function that assigns labels to the vertices. A vertex $v \in V$ and a hyperedge $e \in E$ are said to be *incident* if v is contained in e. The degree of a hyperedge e, denoted $D(e)$, is the number of vertices it connects; similarly, the degree of a vertex v, denoted $D(v)$, is the number of hyperedges to which it belongs. Two vertices are defined as *adjacent* if they share at least one hyperedge, and two hyperedges are *adjacent* if they share at least one common vertex, i.e., $e_1 \cap e_2 \neq \emptyset$, as illustrated in Figure 1.

Given a pattern hypergraph $P = (V', E', L')$ and a data hypergraph $H = (V, E, L)$, *hypergraph pattern mining* (HPM) aims to identify all subhypergraphs of H, referred to as embeddings, that are *isomorphic* to P. As shown in Figure 1, the subhypergraphs $\{e_1, e_2, e_3\}$ and $\{e_6, e_4, e_2\}$ constitute two distinct embeddings that match the pattern within the data hypergraph. An embedding that comprises fewer hyperedges than the full pattern but remains isomorphic to one of its substructures is referred to as a partial embedding. For clarity of exposition, all examples in this paper use unlabeled hypergraphs, although our framework supports both labeled and unlabeled workloads.

HPM Approaches. To address the HPM problem, existing approaches can be broadly categorized into three classes. The first and most straightforward approach transforms a hypergraph into a bipartite graph and then applies conventional graph pattern mining techniques for matching [15, 18–22]. However, this transformation typically leads to a significant expansion in graph size, resulting in considerable storage overhead and increased computational complexity during matching [37].

The second approach, referred to as *match-by-vertex*, directly extends *graph pattern mining* (GPM) techniques to hypergraphs [25, 26]. It recursively expands partial embeddings vertex by vertex, mapping each query vertex to a data vertex at every step, enumerating all possible mappings in a predefined order, and performing backtracking when necessary [29]. However, this method treats hyperedges merely as verification constraints, failing to exploit the higher-order structural information of hypergraphs and consequently incurring a large search space [25, 26] and high enumeration overhead [27, 28].

In comparison, the third approach, termed *match-by-hyperedge*, performs embedding expansion and verification at the hyperedge level [29, 30]. This strategy eliminates redundant vertex-level operations and substantially reduces the search space. Due to its superior efficiency, which achieves

Algorithm 1: Hypergraph Pattern Mining Procedure

Input: Data Hypergraph - H, Pattern Hypergraph - P
Output: Embeddings (M)

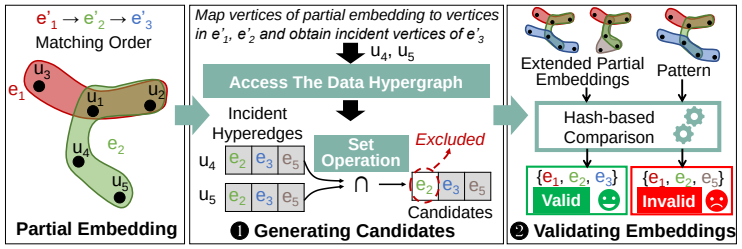
```
1  Function HPM_Procedure(P, H):
2      mo = GenSchedule(P, H) // Generate a schedule for P
3      PE = MatRoot(mo, H), dep = 1
4      M = Mining(mo, PE, P, H, dep)
5  Function Mining(mo, PE, P, H, dep):
6      M' = ∅ // Initial new partial embeddings
7      foreach partial embedding pe ∈ PE do
8          Candidates ← GenCandidates(mo, pe, P, H)
9          foreach hyperedge e ∈ Candidates do
10             pe' = pe ∪ {e}
11             if ValEmbeddings(pe', P) then
12                 M' = M' ∪ {pe'}
13     dep = dep + 1 // Current expansion is completed
14     if depth! = P.size() then // Enter next depth
15         Mining(mo, M', P, H, dep)
16     else // Finish mining and output embeddings
17         return M'
```

orders-of-magnitude improvements over the other two approaches, it represents the current state of the art and forms the basis of this work.

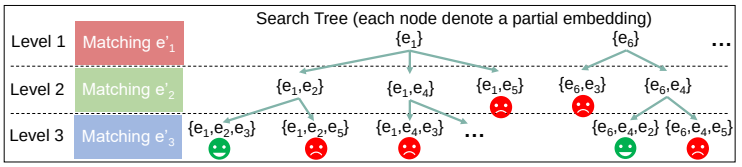
HPM Procedure. Algorithm 1 outlines the procedure of the *match-by-hyperedge* approach in detail. Given a pattern hypergraph and a data hypergraph, GenSchedule first determines a matching order (i.e., schedule) for the hyperedges in the pattern (line 2). Then, MatRoot identifies all hyperedges in the data hypergraph that correspond to the first hyperedge in this order (line 3). Finally, Mining recursively expands hyperedges following the schedule to enumerate all embeddings (line 4).

The core of the procedure lies in the Mining function, which fundamentally differs from conventional graph pattern mining. In traditional GPM, embeddings are expanded at the vertex level, whereas in HPM, Mining performs expansion and validation at the hyperedge level to capture higher-order vertex relationships. Specifically, the Mining function comprises two steps: GenCandidates and ValEmbeddings. Given a partial embedding pe, GenCandidates (line 8) identifies the vertices required in the candidate hyperedges, retrieves their incident hyperedges filtered by the degree of the corresponding pattern hyperedge, and intersects these sets to generate candidate hyperedges. Unlike GPM, where candidate generation typically involves pairwise edge intersections, this process operates over multi-vertex hyperedges, leading to more complex but semantically richer candidate construction. Each candidate hyperedge is then appended to pe to construct extended partial embeddings. Subsequently, ValEmbeddings (line 11) performs a hash-based structural validation between the extended embeddings and the pattern to verify isomorphism.

Figure 2(a) illustrates the two steps using the partial embedding $\{e_1, e_2\}$ within a mining iteration. ❶ To generate candidates for matching hyperedge e'_3, the vertices of $\{e_1, e_2\}$ are mapped to their corresponding pattern vertices, resulting in u_4 and u_5 as the incident vertices of e'_3. The data hypergraph is then accessed to retrieve all incident hyperedges of degree four for u_4 and u_5 (since e'_3 has degree four), and their intersection, $\{e_2, e_3, e_5\} \cap \{e_2, e_3, e_5\}$, produces the candidate set after excluding e_2, which already belongs to $\{e_1, e_2\}$. ❷ Among the extended embeddings $\{e_1, e_2, e_3\}$ and



(a) The Two Core Steps of Hypergraph Pattern Mining



(b) A Search-Tree View of Hypergraph Pattern Mining

Fig. 2. (a) An example illustrating the two steps (❶ GenCandidates and ❷ ValEmbeddings) in a mining loop; (b) The search tree of the mining process for the example in Figure 1

$\{e_1, e_2, e_5\}$, only $\{e_1, e_2, e_3\}$ satisfies the hash-based structural validation in ValEmbeddings, confirming it as a valid embedding. Figure 2(b) provides a global view of the search tree corresponding to the example in Figure 1. The tree is constructed by recursively expanding partial embeddings that pass ValEmbeddings, while invalid ones are pruned. Only validated embeddings generate new branches in the search tree, leading to a systematic and efficient exploration of the solution space.

2.2 UPMEM PIM Architecture

Processing-in-Memory (PIM) is an architectural paradigm that aims to overcome the data movement bottleneck between compute and memory by colocating lightweight processing units within memory modules. By minimizing data transfers across the memory hierarchy, PIM improves effective bandwidth utilization and offers the potential for enhanced performance and energy efficiency, particularly in memory-bound workloads [22, 31, 33, 36]. Among various PIM designs, UPMEM represents the first commercially available implementation [34, 35], integrating thousands of *DRAM Processing Units* (DPUs) into standard DDR4 DIMMs. This architecture enables massive in-memory parallelism at the system level while maintaining compatibility with conventional server infrastructures.

System Organization. Figure 3(a) depicts the architecture of a UPMEM-based PIM system. Each UPMEM module adheres to the standard DDR4 DIMM interface and integrates 16 PIM chips, with each chip hosting 8 lightweight processing cores known as DPUs. A full UPMEM system supports up to 20 DIMMs, enabling a total of 2,560 DPUs (computed as 20 modules × 16 chips × 8 DPUs) to operate concurrently. This configuration delivers a peak aggregate memory bandwidth of up to 7.2 TB/s, providing substantial parallelism and throughput for memory-intensive workloads. Unlike conventional CPU and GPU architectures, where memory bandwidth is fundamentally constrained by the limited number of I/O pins [33], the UPMEM architecture allows near-linear bandwidth scaling with the number of DIMMs, thereby fully utilizing both memory capacity and parallel compute resources.

DPU Microarchitecture. Each DPU is a 14-stage in-order pipelined processor that adheres to a RISC-style *instruction set architecture* (ISA), operates at 350 MHz, and supports up to 24 interleaved hardware tasklets (i.e., threads). To fully utilize the pipeline, at least 11 threads must remain

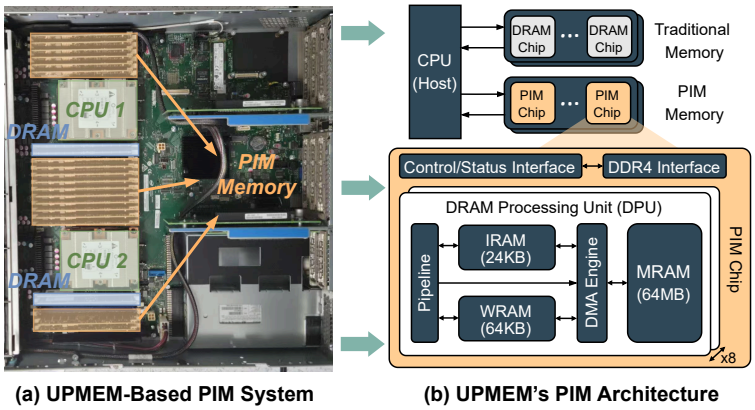


Fig. 3. UPMEM PIM architecture

active [35]; otherwise, computational units are underutilized, making efficient multithreading essential for maximizing performance. As shown in Figure 3(b), each DPU contains 64 MB of *main DRAM* (MRAM), 24 KB of *instruction memory* (IRAM), and 64 KB of *working memory* (WRAM). Data transfers between MRAM and WRAM are explicitly managed in software via a DMA engine, enabling fine-grained control over memory locality. The lack of hardware caches and inter-DPU communication simplifies the microarchitecture but imposes a constrained programming model, which requires explicit coordination of data movement and careful scheduling of threads at the software level. All inter-DPU data transfers must be coordinated by the host CPU.

2.3 Characterizing Hypergraph Pattern Mining

Roofline Analysis. To investigate the computational characteristics of HPM, we conduct a roofline analysis of representative HPM workloads using Intel Advisor [38] on an Intel Xeon Silver 4216 processor. Figure 4 reports the measured operational intensity and attained performance based on profiling results from HGMatch [29], a representative match-by-hyperedge system. The analysis is performed on three real-world datasets, *contact-primary-school* (CP), *senate-bills* (SB), and *contact-high-school* (CH) under three different pattern configurations (P_4, P_5, and P_6). Detailed descriptions of the datasets and experimental settings are provided in § 7.1. Figure 4 shows that all analyzed HPM workloads fall deep within the memory-bound region of the roofline model. The measured operational intensity reaches at most 0.55 GOPS/Byte, while the peak sustained throughput is only 42 GOPS, accounting for less than 4% of the processor's theoretical peak. This disparity originates not from insufficient compute capability but rather from the inherently irregular memory access behavior of HPM workloads.

During execution, the Mining phase continuously issues fine-grained, data-dependent memory accesses whose address streams are determined by the evolving partial embeddings. Each invocation of GenCandidates retrieves multiple adjacency-like incidence lists corresponding to the incident vertices of the next pattern hyperedge, filters them by degree constraints, and performs multiway intersections to generate candidate hyperedges. The retrieval process entails irregular pointer chasing and random index lookups over variable-length incidence lists. Moreover, each partial embedding accesses a distinct region of the incidence structure, resulting in minimal temporal reuse across iterations. Consequently, hardware prefetchers are largely ineffective, cache lines are underutilized, and memory-level parallelism is insufficient to hide DRAM latency. In contrast, the amount of computation performed per byte of data is minimal. The dominant operations involve

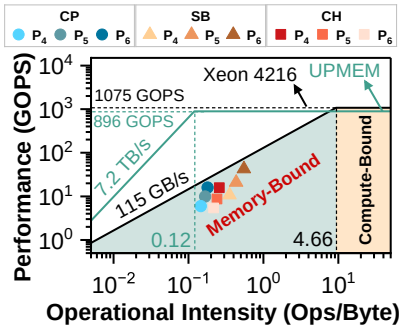


Fig. 4. Roofline analysis of HPM

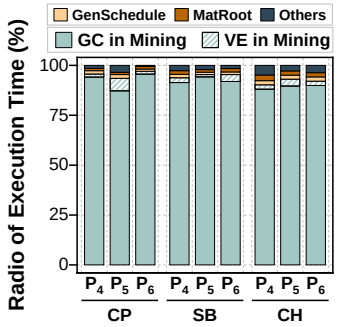


Fig. 5. Execution time breakdown of HPM

set intersections, membership checks, and lightweight hash probes for structural validation, with no expensive floating-point or vector arithmetic. As a result, the work per byte remains extremely low, keeping the operational intensity depressed and confining performance to the limits imposed by memory bandwidth.

While algorithmic optimizations such as schedule-based pruning [27, 28] can reduce the number of candidate extensions, each valid extension in HPM still requires accessing adjacency structures. Consequently, these optimizations primarily reduce constant factors and do not asymptotically decrease the amount of data that must be read per valid match, nor do they increase the compute-to-data ratio sufficiently to escape the bandwidth-bound regime. In this work, we focus on addressing this architectural bottleneck, while algorithmic optimizations remain orthogonal to our approach.

Performance Breakdown. We further decompose the end-to-end execution time across different stages and workloads. As shown in Figure 5, the *GenCandidates* (GC) phase overwhelmingly dominates the runtime, accounting for approximately 87% $\sim$ 96% of the total execution time across CP, SB, and CH under the P_4, P_5, and P_6 configurations. This dominance arises from the data-dependent expansion behavior of GC: its cost scales with (i) the degrees of the participating vertices in the next pattern hyperedge and (ii) the breadth of the search frontier, that is, the number of concurrent partial embeddings to be expanded. Collectively, these factors amplify irregular memory access patterns and impose significant pressure on the CPU memory hierarchy.

By contrast, the *ValEmbeddings* (VE) phase performs lightweight hash-based structural validations on compact working sets, contributing less than 5% of the total runtime. Both GenSchedule and MatRoot execute only once during initialization, where GenSchedule determines the matching schedule for pattern hyperedges and MatRoot identifies the initial set of candidate hyperedges. Their combined cost is negligible compared to the overall runtime. Overall, the profiling results reveal that execution time is overwhelmingly dominated by memory-bound candidate generation.

2.4 Opportunities and Challenges

The preceding analysis exposes a fundamental architectural mismatch between conventional compute-centric architectures and the access patterns of HPM. To overcome this inefficiency, PIM architectures such as UPMEM provide an attractive alternative by integrating thousands of lightweight processing units directly within memory arrays, thereby minimizing data movement. Recent studies have demonstrated the potential of UPMEM across a broad spectrum of data-intensive workloads [22, 31, 32]. Among them, PimPam [22] is most relevant, as it accelerates subgraph matching. However, directly applying graph-oriented PIM designs to HPM is inherently suboptimal. As discussed in § 2.1, converting a hypergraph into an ordinary graph introduces structural expansion and duplication, which significantly increase memory footprint and computational

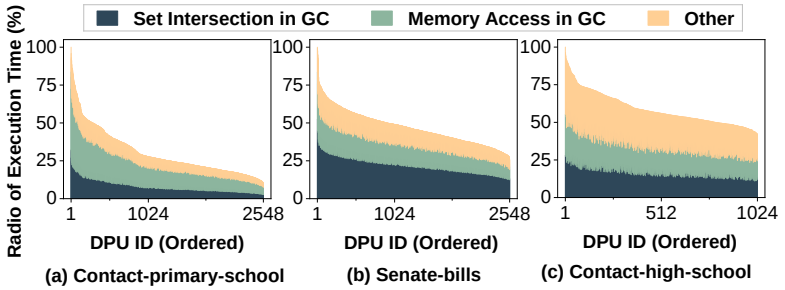


Fig. 6. Execution time breakdown of the UPMEM-based HPM prototype across DPUs under the P_3 pattern

overhead. Consequently, existing PIM-based graph frameworks cannot fully exploit the architectural advantages of UPMEM for hypergraph workloads.

To systematically investigate these limitations, we implement an HPM prototype on the UPMEM platform by extending HGMatch [29] and conduct controlled experiments using three smaller datasets under the P_3 pattern configuration. This setting ensures that the entire data hypergraph fits within MRAM without partitioning, allowing us to isolate system behavior from partitioning effects. The detailed experimental configuration is provided in § 7.1. Although UPMEM provides substantial computational parallelism, with up to 2,560 DPUs and 7.2 TB/s high internal bandwidth, fully harnessing these resources for HPM requires addressing key challenges at both the inter-DPU and intra-DPU levels.

Inter-DPU Data Management. From a data management perspective, the central challenge lies in mapping a massive, highly irregular hypergraph onto thousands of DPUs with severely constrained local memory. Each DPU can access only its 64 MB MRAM, whereas real-world hypergraphs typically contain millions of vertices and hyperedges that far exceed this capacity. To make effective use of the limited on-device memory, the hypergraph must first be compacted into fine-grained, self-contained fragments that encapsulate tightly connected substructures. These fragments are then orchestrated across DPUs to jointly satisfy capacity and load-balance constraints while maintaining full DPU autonomy, thereby avoiding any inter-DPU dependencies that would otherwise require host-mediated coordination.

Achieving this placement requires reconciling multiple and often conflicting objectives. The partitioning must preserve intra-fragment locality to enable efficient candidate generation, eliminate cross-fragment dependencies that trigger remote data exchanges, and balance computation across DPUs despite the highly skewed degree distributions observed in real hypergraphs. However, as shown in Figure 6(a), our prototype exhibits severe load imbalance across DPUs, with utilization differences reaching up to 89.88%. The lack of inter-DPU communication mechanisms makes it impossible to apply runtime balancing techniques such as work stealing, leaving some DPUs idle while others remain heavily loaded. In essence, data organization emerges as a first-class design dimension that determines how the UPMEM platform achieves efficient coordination among thousands of autonomous DPUs.

Intra-DPU Execution Efficiency. From an execution perspective, the challenge lies in maximizing computation efficiency within each DPU under stringent architectural constraints. Each DPU is a lightweight, in-order processor operating at 350 MHz, without caches, branch prediction, or hardware prefetching. Consequently, its performance critically depends on sustaining instruction throughput despite frequent control and memory stalls. In HPM, the set-intersection operation used during candidate generation dominates the overall computation, accounting for up to 57.81% of the total runtime as shown in Figure 6(b). These intersections involve extensive branching and

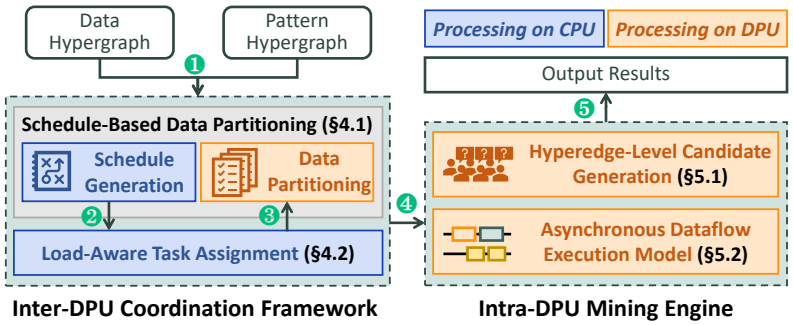


Fig. 7. System-level overview of Octopus

pointer-chasing over irregular memory regions. On UPMEM's strictly in-order pipeline, which lacks branch prediction, such irregular control flow triggers frequent control hazards and severely limits instruction-level parallelism.

Furthermore, as shown in Figure 6, memory-intensive operations such as hypergraph traversals and set intersections proceed sequentially, since the absence of cache hierarchy and hardware prefetching prevents latency hiding. Each MRAM access must be serviced through the memory interface, making the pipeline highly sensitive to irregular memory access patterns and resulting in frequent stalls that dominate DPU execution time. Overall, intra-DPU efficiency is fundamentally constrained by the tight coupling between lightweight computation and latency-bound memory access, revealing the inherent limits of fine-grained parallelism on UPMEM.

3 Overview of Octopus

In this paper, we present Octopus, a system designed to address the aforementioned challenges and accelerate HPM on the UPMEM platform. Figure 7 provides an overview of Octopus, which comprises two tightly integrated components: the inter-DPU coordination framework and the intra-DPU mining engine.

Inter-DPU Coordination Framework (§4). This framework is responsible for global data management and workload organization across thousands of DPUs. Octopus first constructs a memory-efficient scheduling plan tailored to the DPU architecture based on the structural characteristics of the data hypergraph and the query pattern (**Step** ❶), enabling compact data representation and minimal memory fragmentation. It then performs load-aware balanced task assignment by estimating per-DPU workloads and applying skew-aware splitting to achieve balanced distribution across DPUs (**Step** ❷). Finally, the global hypergraph is streamed from the host to the DPUs, where each DPU concurrently extracts the portions required by its assigned tasks and performs data partitioning entirely *in situ* (**Step** ❸). This design avoids centralized preprocessing on the host and minimizes redundant data movement, thereby enabling scalable parallel execution.

Intra-DPU Mining Engine (§5). This engine maximizes the utilization of UPMEM's computing resources to enhance overall execution efficiency (**Step** ❹). Octopus adopts a hyperedge-level candidate generation strategy built upon a LUT-based connectivity graph, which maintains consistency between candidate generation and embedding extension, thereby simplifying computation and reducing on-chip memory overhead. To further mitigate pipeline stalls, Octopus leverages dedicated threads for asynchronous execution, overlapping computation with data movement to improve instruction throughput and resource utilization. Finally, the engine aggregates the final results, outputting either the explicit embeddings or their counts as required (**Step** ❺).

4 Inter-DPU Coordination Framework

Octopus coordinates inter-DPU execution through schedule-based data partitioning and load-aware task assignment, which collectively facilitate compact data organization and balanced workload distribution across DPUs, thereby enabling efficient and scalable in-memory parallel processing.

4.1 Schedule-Based Compact Data Partitioning

Given that the 64 MB MRAM on each DPU cannot accommodate the full data hypergraph, Octopus adopts a schedule-based compact data partitioning strategy that couples structural awareness with memory efficiency. Instead of performing static graph decomposition, Octopus redefines partitioning as a scheduling problem driven by the exploration pattern of mining tasks. By repositioning the first matching hyperedge through a chain unfolding process, Octopus constructs a memory-friendly schedule that minimizes the working set of each task while maintaining complete structural context for pattern matching.

To formalize the principle underlying this design, we extend the notion of *distance* from classical graph theory to hypergraphs. The distance between two hyperedges is defined as the minimum number of hyperedges that must be traversed to connect them; if two hyperedges share at least one vertex, their distance is 1. Each mining task can thus be modeled as a localized exploration rooted at a specific hyperedge, with data access restricted to hyperedges within distance $\mathbb{R}$, referred to as the exploration radius. This radius quantifies the spatial scope of a task's data dependency and directly determines the memory footprint that must reside within one DPU's MRAM. Formally, for a pattern hypergraph with a schedule $[e_1 \rightarrow e_2 \rightarrow ... \rightarrow e_n]$, the exploration radius $\mathbb{R} = \max\{d_2, d_3, ..., d_n\}$, where d_i is the distance between e_i and the starting hyperedge e_1.

Memory-Friendly Schedule. Different schedules yield distinct exploration radii, which directly translate into different task memory requirements. The schedule with the smallest $\mathbb{R}$, termed the *memory-friendly schedule*, minimizes per-task data volume and thereby maximizes DPU memory utilization. Traditional cardinality-based schedules [29], which order hyperedges by ascending cardinality to minimize task count, are suboptimal on UPMEM because they ignore memory constraints. As illustrated in Figure 8(a), a cardinality-based schedule $[e'_1 \rightarrow e'_2 \rightarrow e'_3]$ yields $n = C(e'_1)$ tasks, each requiring data within two hops, whereas a memory-friendly schedule $[e'_2 \rightarrow e'_1 \rightarrow e'_3]$ generates $m = C(e'_2)$ tasks but restricts access to only one-hop neighborhoods. Although the latter produces more tasks, the massive parallelism provided by UPMEM amortizes this increase through concurrent execution, while tighter data locality reduces both MRAM usage and data-transfer overhead, resulting in higher overall throughput.

Schedule Generation via Chain Unfolding. Enumerating all $n!$ possible schedules to identify the minimal $\mathbb{R}$ is computationally infeasible. Octopus therefore introduces a lightweight chain unfolding algorithm that approximates the optimal schedule through structural analysis of the pattern hypergraph. The algorithm first constructs an auxiliary connectivity graph that encodes adjacency relationships among pattern hyperedges, and then unfolds this graph into a linear chain. The hyperedge located at the center of the chain is selected as the starting point, ensuring balanced expansion in both directions of the search space. The remaining hyperedges are ordered according to their connectivity density and cardinality to produce the final schedule.

This unfolding strategy reshapes the search space to achieve balanced expansion and effectively reduces the exploration radius from the worst case $n - 1$ in cardinality-based scheduling to approximately $\lceil n/2 \rceil$, as demonstrated in Figure 8(b). For example, in pattern A, the auxiliary graph positions e'_2 at the center as the starting hyperedge, allowing both e'_1 and e'_3 to be explored within a single-hop neighborhood. In pattern B, two potential roots, e'_2 and e'_3, appear; choosing e'_2 yields a smaller radius $\mathbb{R} = 1$ and therefore becomes the preferred root. Before partitioning, a quick check is

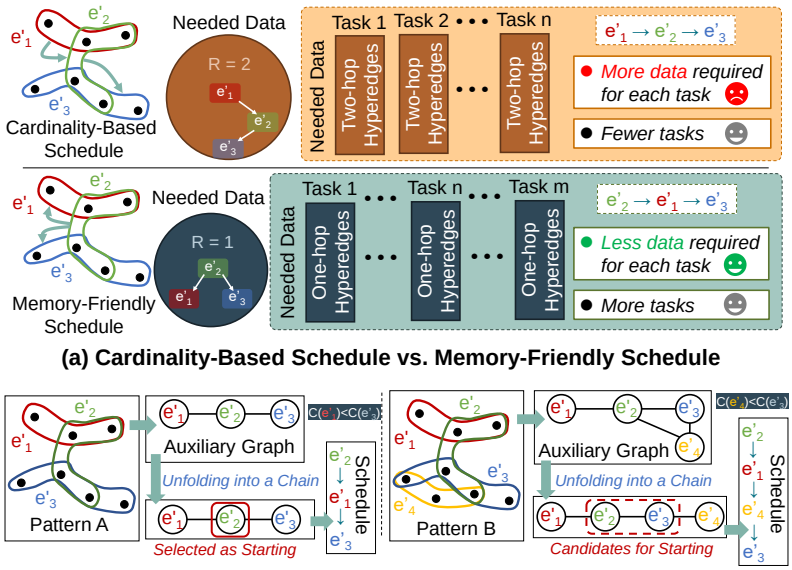


Fig. 8. (a) Traditional cardinality-based schedule vs. memory-friendly schedule; (b) Generating memory-friendly schedules via chain unfolding on patterns A and B

performed: Octopus generates a memory-friendly schedule only if the data of a task partitioned by the cardinality-based schedule exceeds the MRAM capacity, thereby balancing execution efficiency and memory utilization. Furthermore, Octopus adopts a conservative MRAM-aware budgeting strategy during partitioning by decomposing large hypergraphs that cannot fit entirely in MRAM into multiple subhypergraphs. These subhypergraphs are streamed to DPUs and processed in batches. After each batch completes, the occupied MRAM space is released to accommodate subsequent subhypergraphs, thereby ensuring sufficient MRAM capacity throughout execution. In addition, to account for the memory footprint of runtime data structures constructed later in execution, Octopus conservatively bounds the number of hyperedges assigned to each DPU. The detailed design and analysis are discussed in §5.1.

4.2 Load-Aware Balanced Task Assignment

Since UPMEM lacks inter-DPU communication, conventional dynamic load balancing techniques such as task stealing are infeasible, making static load estimation and balanced pre-assignment essential for efficient execution. In hypergraph pattern mining, the computational cost of a task can be approximated by the size of its search tree, which is determined by the number of candidate hyperedges and the cost of embedding validation. However, the search tree size depends on highly dynamic factors, including the data hypergraph structure, the specific query pattern, and the chosen starting hyperedge. Accurately estimating task load without full execution is therefore non-trivial. To address this, Octopus introduces a lightweight analytical model that predicts task load based on structural indicators derived from the pattern schedule and the data hypergraph, enabling balanced task assignment across DPUs.

Load Estimation Model. The estimation model leverages three key hyperedge indicators, namely cardinality, degree, and connection probability, to approximate the expansion complexity of each task's search tree. Given a hyperedge matching order $[e'_1, e'_2, ..., e'_k]$ and a task T_n rooted at hyperedge e_n, the expected number of second-level expansions is represented by the root cardinality $\mathcal{R}_n$, defined

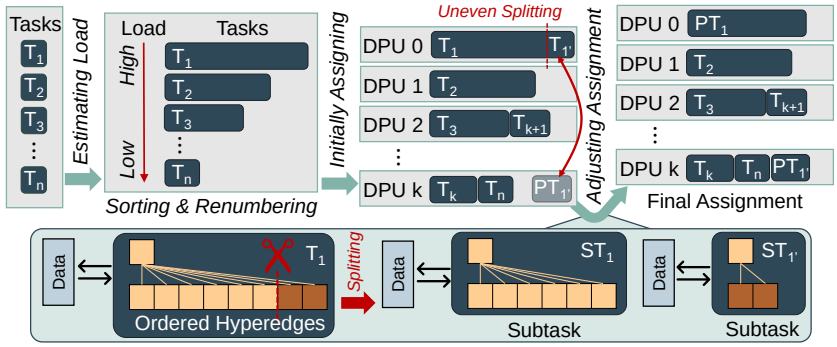


Fig. 9. Workflow of load-aware balanced task assignment

as the number of hyperedges in the data hypergraph that are connected to e_n and share the same degree as e'_2. For the i-th level ($2 \le i \le k$), the expected search tree size can be estimated as $\mathcal{R}_n \cdot \prod_{j=2}^{i}(\mathcal{T}_{D(j-1)D(j)} \cdot \hat{d}_e)$, where $D(j)$ denotes the degree of the hyperedge matched at level j, $\mathcal{T}_{ab}$ is the probability that a hyperedge of degree a connects to one of degree b, and $\hat{d}_e$ is the degree of the hyperedge matched at level i. Accordingly, the complete load estimation function for task T_n is defined as:

$$Load(T_n) = \sum_{i=1}^{k}[\mathcal{R}_n \cdot \prod_{j=2}^{i}(\mathcal{T}_{D(j-1)D(j)} \cdot \hat{d}_e)], \ (\prod_{j=2}^{1}(\cdot) = 1) \tag{1}$$

In practice, $\hat{d}_e$ is replaced by the average hyperedge degree $\mathcal{A}$ to capture the global structural characteristics of the data hypergraph. All indicators can be computed in a preprocessing phase through a single hypergraph traversal, introducing negligible overhead. This model enables pre-runtime estimation of task workloads, allowing Octopus to perform static yet accurate load balancing across thousands of DPUs. In traditional graph pattern mining, several methods [20, 21] have been proposed to estimate task loads, mainly to guide the choice of optimal matching orders. However, these methods apply only to ordinary graphs with binary relations. In contrast, hypergraphs capture higher-order relations among multiple vertices and exhibit much greater structural complexity. Load estimation methods for binary relations cannot capture such dependencies or leverage the structural information in hyperedges, and thus are not applicable to hypergraph pattern mining.

After estimating task loads, we initially assign tasks to DPUs by leveraging principles from the Minimum Makespan Scheduling problem [39, 40], followed by splitting and adjustment to obtain the final assignment.

Minimum Makespan Scheduling. As shown in Figure 9, after estimating task loads, we sort the n tasks in descending order and assign them sequentially. Let $Sum\_Task(i)$ denote the total load of DPU i (initialized to 0), and let V_i denote the union of vertices in the hyperedges of tasks assigned to DPU i. The k-th task T_k is assigned to the DPU $i^* = \arg \max_{i \in \arg\min_j Sum\_Task(j)} \left|V(T_k) \cap V_j\right|$. That is, task T_k is assigned by first minimizing load imbalance across DPUs and then maximizing vertex overlap with previously assigned tasks. After assignment, $Sum\_Task(i^*)$ is updated by adding $Load(T_k)$, and V_{i^*} is updated as $V_{i^*} \leftarrow V_{i^*} \cup V(T_k)$.

Under this scheduling strategy, each DPU materializes a single compact fragment representing the union of the $\mathbb{R}$-hop neighborhoods of all assigned tasks. Within a DPU, each hyperedge and its adjacency information are stored at most once, with overlapping regions shared across tasks. Replication therefore occurs only across DPUs, i.e., when a hyperedge lies on the boundary of fragments assigned to different DPUs. In this way, the worst-case replication factor is structurally

bounded by the number of DPUs participating in execution. Furthermore, by co-locating structurally similar tasks on the same DPU, tasks with highly overlapping $\mathbb{R}$-hop neighborhoods naturally share data in MRAM, effectively reducing redundant data placement and mitigating replication overhead. A quantitative analysis of replication overhead is provided in §7.5.

Adjusting Task Assignment via Splitting. After the initial assignment, we introduce a task splitting mechanism to ensure tasks fit within each DPU's MRAM capacity and mitigate load imbalance. Specifically, any root hyperedge e with degree $d_e > \theta$ (where θ is a tunable threshold, default 1024) is partitioned into $\epsilon = \lceil d_e/\theta \rceil$ disjoint segments. Each segment e_i is treated as an independent partial task with cost $Load(T_{e_i}) = Load(T_e)/\epsilon$, and the split tasks are scheduled independently. This process guarantees that the data required by each DPU remains within MRAM capacity. To further alleviate imbalance, we identify the most loaded DPU a and the least loaded DPU b. The heaviest task T_n on a is unevenly split into two subtasks, ST_n and $ST_{n'}$, assigned to a and b, respectively. This uneven split is realized by restricting the expansion at the second level of the search tree: PT_n expands only the top $\mathcal{P}\%$ of ordered hyperedges, while $PT_{n'}$ handles the remainder. The ratio $\mathcal{P}$ is defined as $\mathcal{P} = 1 - \frac{(MAX-MIN)/2}{Load(T_n)}$, where MAX and MIN denote the maximum and minimum DPU loads, respectively. We iteratively split the most imbalanced DPU pair until the load gap falls below 5% of the maximum to balance load and memory efficiency.

5 Intra-DPU Mining Engine

To achieve efficient runtime execution under the architectural constraints of UPMEM DPUs, Octopus introduces a hyperedge-level candidate generation mechanism and an asynchronous dataflow execution model with dedicated threads to maximize parallelism and resource utilization.

5.1 Hyperedge-Level Candidate Generation

Each DPU is a lightweight in-order unit without branch prediction, and candidate generation, which dominates HPM runtime (as analyzed in § 2.3), requires frequent set intersections poorly supported by this architecture. To avoid this becoming a bottleneck, Octopus introduces hyperedge-level candidate generation. By reducing the data hypergraph into a hyperedge connectivity graph, Octopus converts fine-grained, vertex-level intersection operations into a small number of coarse-grained, hyperedge-level intersections, thereby lowering the cost of candidate hyperedge generation.

Existing approaches extend partial embeddings at the hyperedge level but still generate candidates at the vertex level. This inconsistency results in redundant intersection computations. As shown in Figure 10(a), when extending local embedding $\{e_1, e_2\}$, the vertex-level approach accesses the incident hyperedge sets of vertices u_4, u_5, and u_6, and intersects them to obtain candidates for matching e_3' in the pattern. However, the incident hyperedge sets of u_4 and u_5 are identical, making such computations unnecessary.

To eliminate redundant fine-grained operations, Octopus elevates candidate generation to the hyperedge level, ensuring consistent granularity between candidate generation and embedding extension. The core idea is a space-for-time trade-off: the hypergraph is reduced to a two-dimensional connectivity graph that captures adjacency among hyperedges, and candidates are generated from this connectivity graph rather than the original hypergraph. At the vertex level, the complexity of candidate generation depends on both hyperedge order and vertex degree, whereas at the hyperedge level, dimensionality reduction simplifies the complexity to depend only on vertex degree. As shown in Figure 10(b), once the hypergraph is reduced to a connectivity graph, extending a partial embedding $\{e_1, e_2\}$ only requires accessing e_2's adjacent hyperedges and filtering to obtain candidates $\{e_3, e_4, e_5\}$, avoiding repeated fine-grained intersections. Octopus realizes hyperedge-level candidate generation via two components: the *Dimensionality Reducer* and the *Candidates Filter*.

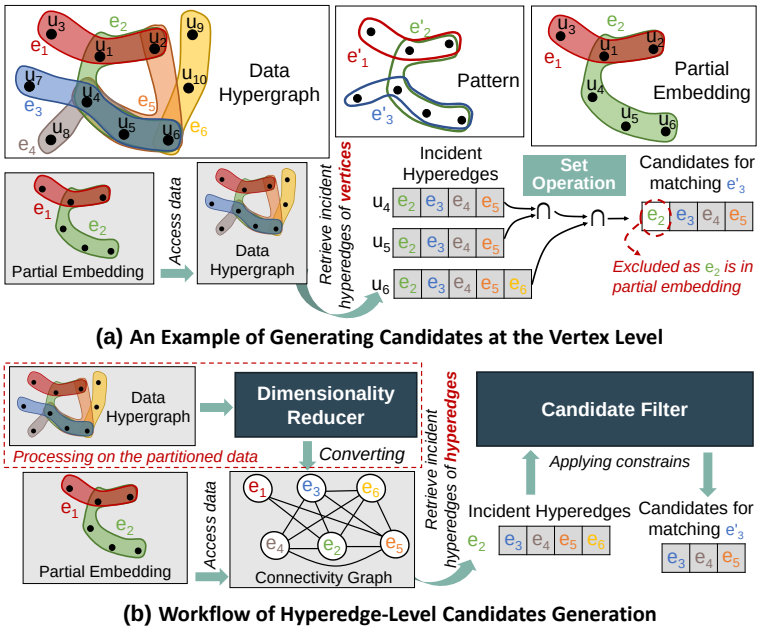
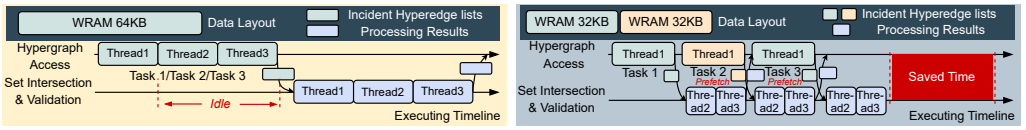


Fig. 10. (a) An example of vertex-level candidate generation when extending partial embedding $\{e_1, e_2\}$; (b) The workflow of hyperedge-level candidate generation

Dimensionality Reducer. It is responsible for efficiently constructing the connectivity graph from the original data hypergraph. In the connectivity graph, each vertex corresponds to a hyperedge in the data hypergraph, and if two hyperedges share a common vertex, an edge is added between their corresponding vertices. A naive construction approach is to traverse each hyperedge's vertex set and check for intersections with all other hyperedges, which incurs frequent set lookups and high overhead. To address this, the Dimensionality Reducer leverages a *lookup table* (LUT) with row inputs and column scans. In the LUT, rows represent hyperedges and columns represent vertices; if hyperedge e_i contains vertex u_j, the entry (i, j) is set to 1. The LUT is constructed by iterating through all hyperedges and filling in their corresponding rows. Once constructed, the LUT is scanned column by column: hyperedges with a value of 1 in the same column share a same vertex and are connected in the connectivity graph. Assume the original hypergraph data contains $|E|$ hyperedges and $|V|$ vertices, with each hyperedge containing an average of D vertices. Directly intersection checking on the data has complexity $O(|E|^2D)$, while the LUT-based approach has $O(|E||V|)$. Since $|E| \gg |V|$ in most hypergraphs, we have $O(|E||V|) < O(|E|^2D)$, making the LUT-based construction more efficient.

Candidate Filter. It is responsible for applying constraints such as labels and degrees from the original data during candidate hyperedge generation, thereby pruning the search tree effectively. Specifically, the labels and degrees of hyperedges in the original data are stored as vertex attributes in the connectivity graph. The Candidate Filter applies these constraints to the candidate set derived solely from connectivity, discarding hyperedges that clearly do not satisfy the pattern requirements. This reduces unnecessary data access and computation. As shown in Figure 10(b), the connectivity graph alone produces $\{e_3, e_4, e_5, e_6\}$ as candidates, but e_6, with degree 3 instead of the required 4, is discarded, leaving $\{e_3, e_4, e_5\}$ as the final candidates for e'_3.

Discussion. In theory, a partition containing E hyperedges may induce a worst-case connectivity graph size of $O(E^2)$ when every pair of hyperedges shares at least one vertex, potentially exceeding



(a) Synchronous Execution on Homogeneous Threads (b) Asynchronous Execution via Dedicated Threads

Fig. 11. Execution flow: synchronous with homogeneous threads vs. asynchronous via dedicated threads

MRAM capacity. In practice, the connectivity graph is much sparser and its size is governed by hyperedge degree: each hyperedge connects only to other hyperedges with which it shares at least one vertex, and thus has a bounded number of neighbors. As described in § 4.2, Octopus splits hyperedges with degree greater than 1,024 during partitioning, effectively capping the degree of each hyperedge and bounding the connectivity graph footprint by $O(E \cdot 1{,}024)$. Given a 64 MB MRAM budget and 4 B per connectivity graph edge, this bound allows fewer than $(64\,\text{MB}/4\,\text{B})/1{,}024 \approx 1.6 \times 10^4$ hyperedges to be assigned to a single DPU, thereby guaranteeing MRAM fitting. This conservative upper-bound estimation also preserves sufficient MRAM capacity to accommodate other runtime data structures, including the local subhypergraph itself.

Regarding the memory footprint of intermediate candidates, Octopus adopts a depth-first expansion strategy, under which each DPU maintains only three adjacency lists at any time: the adjacency of the current hyperedge, that of a single candidate hyperedge, and the adjacency of the resulting extension. Since hyperedges with degree greater than 1,024 are split during partitioning, each adjacency list contains at most 1,024 entries. With 4 B per vertex ID, the worst-case intermediate state is bounded by $3 \times 1{,}024 \times 4\,\text{B} = 12\,\text{KB}$, which comfortably fits within a fixed-size buffer.

5.2 Asynchronous Dataflow Execution Model

UPMEM DPUs lack hardware-supported caches, and MRAM accesses are explicitly managed by the DMA engine. This constraint introduces substantial memory access latency, and the synchronous execution on conventional threads further causes pipeline stalls. As shown in Figure 11(a), when Threads 1~3 perform hypergraph access, set intersection, and validation sequentially, the pipeline stalls and compute resources are underutilized. To mitigate this bottleneck, Octopus exploits UPMEM's *single program multiple data* (SPMD) execution model to construct a two-stage software pipeline within each DPU. Threads are functionally specialized into *loader* and *worker* roles, decoupling data movement from computation to form an asynchronous dataflow execution model.

Loader Threads. Loader threads asynchronously issue the DMA-managed `mram_read()` to fetch hyperedge data from MRAM, package it into data tuples, and stage it into the WRAM FIFO queue. Each loader thread uses double buffering to sustain a continuous inflow: while the current buffer is consumed, it prefetches the data for the next task into the alternate buffer. As shown in Figure 11(b), while Threads 2 and 3 process the data for finishing Task 1, Thread 1 has already prefetched the data for Task 2 into the buffer. This software pipeline overlaps data movement with computation and keeps the worker threads supplied with ready data.

Worker Threads. Worker threads continuously dequeue ready-to-process data tuples from the WRAM-resident FIFO. They require no additional MRAM access: each worker performs set intersection and subsequent validation directly on the data tuples in WRAM. Octopus implements the FIFO as a lock-free buffer using WRAM-resident tuples and per-thread registers, enabling high-throughput coordination between loader and worker threads.

Threads Coordination. Octopus employs a lock-free circular FIFO queue built on WRAM-resident control blocks and thread-local registers to coordinate data exchange between loader and worker

threads. This queue implements decoupled producer-consumer progress, where the loader asynchronously streams prefetched data into WRAM, and the worker consumes it upon readiness, forming a fine-grained dataflow within each DPU. Synchronization relies on lightweight head-tail index polling without locks or global barriers.

To maximize pipeline utilization, Octopus defaults to 4 loader threads and 12 worker threads. This configuration is sufficient to cover the DPU's 11-cycle dispatch gap in its deep in-order pipeline, where only the last 3 of 14 stages support intra-thread overlap [41]. The configuration ensures a steady data supply and maximizes pipeline occupancy, sustaining memory-compute concurrency.

6 Implementation

Built on the UPMEM SDK [35], Octopus[1] comprises approximately 3,000 lines of C code spanning both host-side and DPU kernels. The host CPU handles schedule generation, load estimation, and task assignment. Once the data is ready, each DPU converts it into a connectivity graph using the method in § 5.1, and iteratively performs embedding extension and validation. Data transfer between the CPU and DPUs requires explicit movement and management. During task distribution, the host employs `dpu_prepare_xfer` and `dpu_push_xfer` to scatter tasks, assigning those associated with different root hyperedges to individual DPUs. To reduce partitioning overhead, data partitioning is offloaded to the DPUs: hypergraph data is divided into blocks aligned with the DPU block size and streamed to all DPUs via `dpu_broadcast_to`, ensuring each DPU receives the necessary data until all blocks are processed. This implementation fully exploits UPMEM's parallelism and minimizes idle time caused by host delays. Finally, each DPU returns its local execution results to the host via `dpu_copy_from`, where results are aggregated into the final output.

7 Evaluation

This section outlines the configuration of Octopus and presents an evaluation that substantiates its superior performance.

7.1 Experimental Setup

Baselines. We evaluate Octopus's performance with HGMatch [29], OHMiner [30], and UPHPM. HGMatch and OHMiner are state-of-the-art hypergraph pattern mining systems employing the match-by-hyperedge approach, where HGMatch first introduced this method and OHMiner built on HGMatch by incorporating hyperedge intersection operations. We implement a baseline version of hypergraph pattern mining on UPMEM, referred to as UPHPM, which follows the match-by-hyperedge paradigm. Unlike Octopus, UPHPM does not integrate any UPMEM-oriented optimization techniques. Additionally, we compare Octopus with two state-of-the-art GPM systems, Pangolin [18] (GPU) and PimPam [22] (UPMEM).

Hypergraph Datasets and Patterns. We use eight real-world hypergraph datasets [42] for evaluation, as shown in Table 1. We characterize each dataset using $|V|$, $|E|$, D_{max}, $\bar{D}$, $Size$, and T_Size, which denote the number of vertices, the number of hyperedges, the maximum and average hyperedge degrees, the hypergraph size, and the size of the corresponding ordinary graph after bipartite transformation, respectively. All repeated hyperedges and all repeated vertices in one hyperedge are removed in preprocessing. For fair and standardized comparison, we follow the established evaluation methodology adopted by prior HPM systems [29, 30], where pattern hypergraphs are generated via random walks under controlled size and degree constraints, and results are averaged over multiple patterns per configuration. The detailed pattern settings are presented in Table 2. For each pattern setting, five patterns are generated, and the average result is reported. We preset

[1]Octopus is open-sourced at https://github.com/CGCL-codes/Octopus

Table 1. Real-world hypergraph datasets

Dataset	$\|V\|$	$\|E\|$	D_{max}	$\bar{D}$	*Size*	*T_Size*
Contact-high-school (CH) [43]	327	7,818	5	2.3	0.06MB	0.2MB
Contact-primary-school (CP) [44]	242	12,704	5	2.4	0.1MB	0.4MB
Senate-bills (SB) [45]	294	20,584	99	8.0	0.8MB	3.8MB
House-bills (HB) [46]	1,494	60,987	399	20.5	4.9MB	33.8MB
Walmart-trips (WT) [47]	88,860	69,906	25	6.6	2.4MB	12.5MB
Trivago-clicks (TC) [48]	172,738	233,202	86	4.1	4.4MB	22.4MB
Stackoverflow-answers (SA) [49]	15,211,989	1,103,243	61,315	23.7	203.7MB	2.4GB
Amazon-reviews (AR) [50]	2,268,231	4,285,363	9,350	17.1	523.6MB	5.6GB

Table 2. Pattern settings

Pattern	P_2	P_3	P_4	P_5	P_6
$\|E\|$	2	3	4	5	6
$\|V\|_{min}$	5	10	10	15	15
$\|V\|_{max}$	15	20	30	35	40

Table 3. Hardware specifications

Hardware	Approx. Price	Peak Power
2×Intel Xeon Silver 4216	2,800 USD	220 W
NVIDIA H100 Tensor Core GPU	20,000 USD	350 W
20×UPMEM PIM (2,560 DPUs)	5,800 USD	462 W

five settings, denoted as P_i, where i indicates the number of hyperedges in a pattern, as shown in column $|E|$. The number of vertices is constrained to the range $[|V|_{min}, |V|_{max}]$.

Experimental Setup. In our experiments, we use a commercial UPMEM server equipped with two Intel Xeon Silver 4216 CPUs and 512 GB of DDR4 memory. All reported results are obtained from executions on real hardware. Besides, the server contains 20 PIM-enabled DIMMs, comprising a total of 2,560 DPUs operating at 350 MHz. Since 12 DPUs are faulty—which is a common occurrence also reported in prior works [22, 41]—unless otherwise specified, both UPHPM and Octopus effectively utilize 2,548 DPUs, each supporting 16 hardware threads. The host-side programs are compiled using GCC 8.3.0, while the DPU-side programs are developed using UPMEM SDK 2025.1.0 together with the DPU runtime libraries, and compiled with `dpu-upmem-dpurte-clang` under the LLVM backend with `-O3` optimizations. HGMatch and OHMiner are executed on the same CPUs but utilize only the traditional DDR4 memory. PimPam is evaluated on the same UPMEM server, while Pangolin is evaluated on an NVIDIA H100 GPU, compiled with nvcc v11.7 and optimized using -O3 for a fair comparison. Table 3 summarizes the hardware specifications used in our evaluation.

7.2 Overall Performance

Unlabeled Workloads. We evaluate Octopus against HGMatch, OHMiner, and UPHPM for unlabeled HPM on six real-world hypergraph datasets under five pattern settings. As shown in Figure 12, UPHPM achieves average speedups of 7.84×, 6.27×, 6.07×, 5.88×, 4.88×, and 3.44× across the six datasets compared with HGMatch, highlighting the performance gains obtained by leveraging the UPMEM architecture for memory-bound applications. When compared with OHMiner, we observe that UPHPM sometimes outperforms OHMiner, while at other times it falls behind. As discussed in § 2.4, directly accelerating HPM on the UPMEM architecture still faces challenges such as load imbalance and limited computational resources. To address these issues, Octopus introduces a series of system-level design optimizations, which fully exploit the potential of the UPMEM architecture and achieve a 4.37× ∼ 9.32× performance improvement over UPHPM. Overall, Octopus delivers speedups from 18.19× to 55.37× compared with HGMatch, and achieves speedups from 2.16× to 19.80× compared with OHMiner. Octopus's optimizations are analyzed in detail in § 7.3.

Large Hypergraphs. We evaluate the HPM performance of Octopus on large hypergraphs, specifically the *stackoverflow-answers* (SA) and *amazon-reviews* (AR) datasets listed in Table 1. The SA

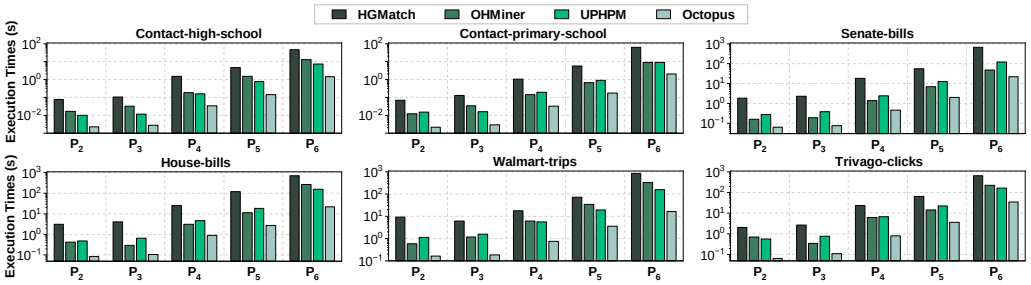


Fig. 12. Overall performance of HGMatch, OHMiner, UPHPM, and Octopus on unlabeled workloads

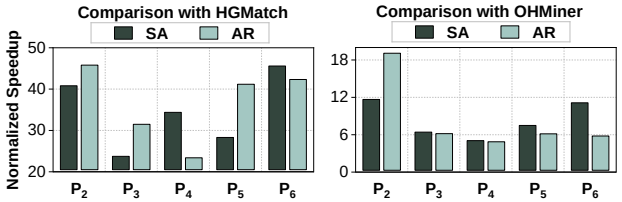


Fig. 13. Performance comparison on large hypergraphs

dataset contains 1.10 million hyperedges and the AR dataset contains 4.28 million, making it infeasible to fit such large-scale data directly into UPMEM's limited MRAM. UPHPM is unable to process HPM on these two large hypergraphs and is therefore excluded from this experiment. Consequently, we compare Octopus only against HGMatch and OHMiner. As shown in Figure 13, Octopus achieves 23.38× ∼ 45.81× speedups over HGMatch and 5.05× ∼ 19.08× speedups over OHMiner. These results confirm that Octopus consistently delivers substantial performance advantages on large-scale hypergraph workloads. Even as dataset sizes increase, Octopus leverages schedule-based compact data partitioning on DPUs to precisely determine the data required by each DPU, overcoming MRAM capacity limits and ensuring scalability to large workloads.

Labeled Workloads. We evaluate the performance of HGMatch, OHMiner, UPHPM, and Octopus under labeled workloads, using HGMatch as the baseline for speedup calculation. By introducing label-based pruning during candidate generation to eliminate mismatched hyperedges and incorporating label mapping checks during validation, these systems can naturally be extended to handle labeled HPM workloads. As shown in Figure 14, Octopus achieves average speedups of 20.30×, 3.90×, and 6.59× over HGMatch, OHMiner, and UPHPM, respectively, demonstrating that it consistently delivers significant performance gains even under labeled workloads. Although labeled workloads introduce additional conditional checks and increase runtime complexity, the label accesses in the extension phase further amplify bandwidth demands. Octopus exploits UPMEM's high bandwidth through system-level optimizations, sustaining substantial performance gains.

Comparison with GPU-Based and UPMEM-Based GPM Systems. We compare Octopus with two state-of-the-art GPM systems: Pangolin (based on the GPU architecture) and PimPam (based on the UPMEM architecture) on the HB, WT, and TC datasets. To enable HPM on Pangolin and PimPam, we convert both the data and pattern hypergraphs into bipartite graphs, thus transforming HPM tasks into GPM tasks. This transformation introduces additional overhead and increases graph storage costs (e.g., the transformed TC dataset is 5.11× larger than the original). Ignoring the transformation overhead, Figure 15 shows the execution time of the matching process across Pangolin, PimPam, and Octopus. The results show that Octopus achieves speedups of up to 1036.10× and 795.05×, with averages of 906.56× and 642.43×, over Pangolin and PimPam, respectively. This

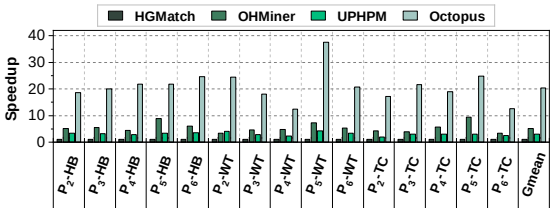


Fig. 14. Performance comparison on labeled workloads

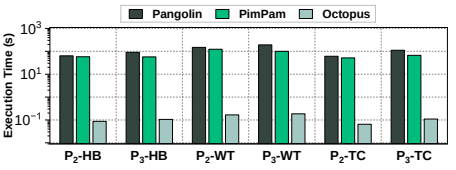


Fig. 15. Performance comparison of Octopus with the GPM systems: Pangolin (GPU) and PimPam (PIM)

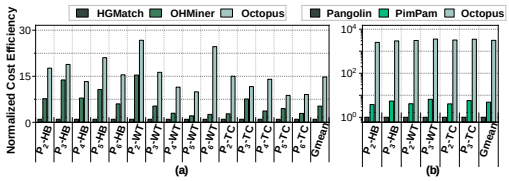


Fig. 16. Comparison of cost-efficiency, with results normalized to (a) HGMatch and (b) Pangolin

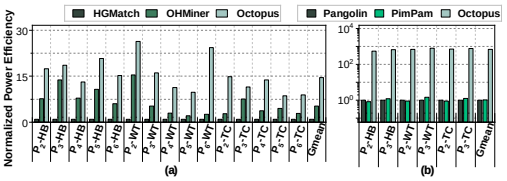


Fig. 17. Comparison of power-efficiency, with results normalized to (a) HGMatch and (b) Pangolin

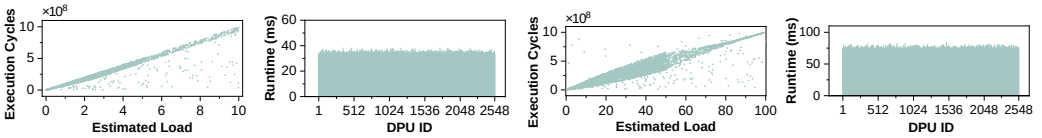
is because converting HPM tasks into GPM tasks significantly increases the complexity of task matching. Specifically, matching a pattern hypergraph with three hyperedges is transformed into matching a bipartite pattern graph with 23 vertices and 22 edges, which substantially increases the search space and computational overhead. By performing matching at the hyperedge level, Octopus achieves higher mining efficiency.

Cost-Efficiency. Cost-efficiency is commonly measured as average performance normalized by capital expenditure [51]. As shown in Figure 16, Octopus achieves average cost-efficiency improvements of 14.76×, 2.79×, 3126.06×, and 642.43× over HGMatch, OHMiner, Pangolin, and PimPam, respectively. Pangolin and PimPam are designed for graph mining and do not scale to complex hypergraph pattern matching; they are unable to complete such workloads within 12 hours and are therefore evaluated only on simpler patterns. These gains arise from a fundamental mismatch between conventional CPU/GPU architectures and HPM workloads. While CPUs and GPUs offer high compute throughput, HPM is predominantly bandwidth-bound, limiting their ability to translate raw compute capacity into proportional performance gains. In contrast, UPMEM provides substantially higher effective memory bandwidth per dollar, enabling Octopus to more directly convert hardware cost into performance and thereby achieve superior cost-efficiency.

Power-Efficiency. As shown in Figure 17, Octopus reduces average energy consumption by 14.55× and 2.75× compared to the CPU-based systems HGMatch and OHMiner, respectively, and by 686.79× compared to the GPU-based system Pangolin. Although the UPMEM platform exhibits higher peak power than the CPU and GPU baselines, the substantial speedups achieved by Octopus result in significantly lower overall energy consumption. This improvement stems from in-MRAM data processing, which eliminates costly DRAM-CPU/GPU data movement, and from lightweight, ultra-low-power DPU cores that maintain high utilization for HPM workloads. Consequently, Octopus delivers significantly better power-efficiency in practice.

7.3 Effectiveness of Optimizations in Octopus

Load-Aware Balanced Task Assignment. We conduct experiments on two workloads (matching P_3 on the CP and SB datasets) that previously exhibit load imbalance in UPHPM, as shown in



(a) Load estimation and DPU execution time on CP (b) Load estimation and DPU execution time on SB

Fig. 18. Evaluation of load-aware balanced task assignment on CP and SB

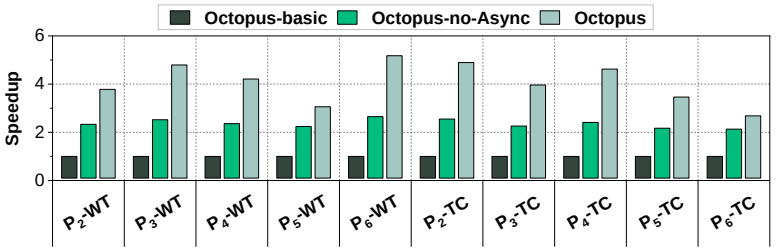


Fig. 19. Performance gains of Octopus's intra-DPU mining engine

Figure 6, to evaluate the effectiveness of Octopus's load-aware balanced task assignment mechanism. Figure 18 presents the results, with the left subfigures showing the predicted task loads and the right subfigures showing the actual DPU execution after task assignment. As illustrated in Figure 18(a) (left), the predicted results exhibit a strong positive correlation with the actual number of execution cycles, demonstrating that our load estimation model effectively captures task loads. Even in more complex hypergraphs where estimation is more challenging, such as in Figure 18(b) (left), the model still provides sufficiently accurate estimates. Moreover, comparing the execution results on the right side of Figure 18 with those in Figure 6 clearly shows that Octopus significantly improves load balance across DPUs. Overall, these results validate the effectiveness of the load-aware balanced task assignment in Octopus.

Intra-DPU Mining Engine. To evaluate the effectiveness of the intra-DPU mining engine in Octopus, particularly the hyperedge-level candidate generation and asynchronous execution strategies, we construct two variants of Octopus for comparison: Octopus-basic, which disables both strategies, and Octopus-no-Async, which disables only asynchronous execution. Figure 19 presents the normalized speedups of Octopus and Octopus-no-Async over Octopus-basic on the WT and TC datasets. The results show that, with the hyperedge-level candidate generation strategy enabled, Octopus-no-Async achieves 2.13× ∼ 2.65× performance improvements over Octopus-basic. On top of this, Octopus further outperforms Octopus-no-Async by 1.26× ∼ 1.95×, demonstrating the additional benefits brought by adopting the asynchronous execution strategy.

7.4 End-to-End Execution Time Breakdown

Table 4 presents an end-to-end execution-time breakdown of Octopus for matching P_6 across eight datasets of varying scales, decomposing the total runtime into CPU-side preprocessing, host-PIM data transfer, and DPU-side computation (*convert* and *match*). Overall, pattern matching dominates end-to-end execution, accounting for 79.19% to 90.26% of the total runtime. Benefiting from our lightweight estimation model, CPU-side preprocessing contributes only 7.93% to 18.27% of the end-to-end time. While its absolute cost grows approximately linearly with graph scale, the DPU-side HPM computation increases much more rapidly with input size, leading to a decreasing relative preprocessing overhead as dataset size grows. Moreover, preprocessing is typically a one-time cost and can be amortized across repeated executions.

Table 4. End-to-end execution time breakdown for matching P_6 across datasets. Absolute time and percentage are reported for each phase (darker colors indicate higher proportions).

Phase		Metric	CH	CP	SB	HB	WT	TC	SA	AR
Preprocess (CPU-side)		Time	38.36 ms	51.78 ms	0.88 s	3.18 s	1.34 s	2.53 s	116.13 s	454.18 s
		%	18.27%	15.23%	17.91%	10.92%	11.37%	7.93%	8.37%	9.67%
Transfer (CPU→PIM)		Time	1.98 ms	2.94 ms	23.17 ms	81.78 ms	39.94 ms	67.19 ms	2.50 s	9.39 s
		%	0.94%	0.86%	0.47%	0.28%	0.33%	0.21%	0.18%	0.20%
Computation (DPU-side)	**Convert**	Time	3.66 ms	6.83 ms	0.14 s	1.43 s	0.41 s	0.57 s	34.79 s	102.37 s
		%	1.74%	2.01%	2.90%	4.93%	3.52%	1.81%	2.51%	2.18%
	Match	Time	0.16 s	0.28 s	3.89 s	24.57 s	10.05 s	28.87 s	1236.57 s	4140.30 s
		%	79.99%	82.76%	79.19%	84.15%	85.11%	90.26%	89.12%	88.15%

Low High

Table 5. Overall memory usage and data amplification

Dataset	TC			SA			AR		
Pattern	P_2	P_4	P_6	P_2	P_4	P_6	P_2	P_4	P_6
Total MRAM (MB)	5.87	41.54	338.93	423.78	3763.11	31687.21	988.47	6479.97	69407.33
Data Amp. (×)	1.33	9.44	77.03	2.08	18.47	155.56	1.89	12.39	132.71

Host-PIM data transfer accounts for only 0.18% to 0.94% of total execution time and remains negligible even for large datasets such as SA and AR. This behavior fundamentally differs from that of communication-intensive PIM workloads [52–55], where computational complexity typically scales linearly with input size. In such settings, host-to-PIM transfer time grows proportionally and often becomes a performance bottleneck once the available bandwidth is saturated. In contrast, HPM is NP-complete, with computational cost typically growing exponentially with input size. Since host-PIM transfer latency increases only linearly and is decoupled from the combinatorial search performed within DPUs, the much faster growth of DPU-side computation causes data transfer to constitute only a small fraction of the end-to-end execution time.

7.5 Memory Overhead Analysis

Overall Memory Overhead. Table 5 reports the total MRAM usage across all DPUs and the corresponding data amplification factors when Octopus matches different patterns on large hypergraphs. Even on the densest datasets and with the most complex patterns, the replication factor remains modest. For example, matching P_6 on SA results in a 155.56× amplification, accounting for only 19.34% of MRAM capacity on average. Moreover, replication incurs minimal impact on execution time, as the hypergraph is streamed from the host to DPUs, where each DPU extracts and partitions the data required for its assigned tasks in situ. Consequently, CPU–DPU transfer cost remains invariant across configurations.

Per-DPU Memory Overhead. Figure 20 shows the additional MRAM overhead incurred by storing connectivity graphs in Octopus on the WT and TC datasets under different pattern settings. We report the maximum MRAM utilization across all DPUs. The results indicate that even after storing connectivity graphs, MRAM usage remains below 50% in all cases (as marked by the red dashed line in Figure 20), demonstrating that ample MRAM capacity remains available. Specifically, in the most demanding scenario—matching P_6 on the WT dataset—the total MRAM utilization after storing the connectivity graph is only 45.58%. This efficiency arises from the schedule-based compact data

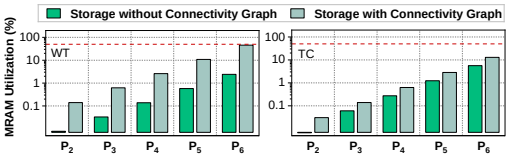


Fig. 20. MRAM utilization for connectivity graphs

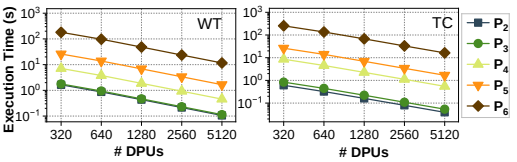


Fig. 21. Scalability of Octopus on WT and TC

partitioning strategy, which minimizes the size of each hypergraph partition placed on a single DPU, ensuring that the converted connectivity graphs fit within MRAM.

7.6 Scalability

We evaluate the scalability of Octopus by varying the number of active DPUs from 320 to 5,120. Since a single server provides only 2,548 DPUs, we emulate larger configurations (up to 5,120 DPUs) by partitioning the workload and executing it sequentially in batches on the same server. For each configuration, we record the tail latency of DPU execution to capture the worst-case runtime. As shown in Figure 21, Octopus demonstrates near-linear scalability when both the number of DPUs and runtime are plotted on a logarithmic scale and increasing the number of DPUs from 320 to 5,120 yields an average speedup of about 15.55×. For example, when matching P_2 on the TC dataset with the number of DPUs increasing from 320 to 5,120, the execution time decreases from 611.30 ms to 39.23 ms, corresponding to a 15.58× speedup.

8 Related Work

Graph Pattern Mining. Previous research on GPM are generally classified into embedding-centric systems [15–19] and pattern-centric systems [20–23]. The former discovers embeddings via vertex extensions and isomorphism tests, while the latter reduces the search space through nested-loop set operations to avoid costly isomorphism checks. However, these GPM-oriented designs cannot directly support HPM, often requiring additional hypergraph-to-bipartite transformations that undermine hypergraphs' ability to represent multi-entity relationships and substantially increase computational complexity [24].

Hypergraph Pattern Mining. Existing HPM approaches fall into two categories: *match-by-vertex* and *match-by-hypergraph*. The *match-by-vertex* approach [25, 26] enumerates embeddings by recursively extending vertices with backtracking, while efficiency has been improved through structural indexing [8] and hyperedge-based pruning [27, 28]. Nonetheless, these systems still incur high enumeration overhead and face a vast search space. To overcome this, HGMatch [29] adopts a *match-by-hypergraph* strategy, extending and validating embeddings hyperedge by hyperedge, yielding orders-of-magnitude speedups. Building on HGMatch, OHMiner [30] applies a Venn diagram-based scheme to further accelerate hyperedge-level matching. Octopus advances this paradigm by systematically analyzing HPM and introducing system-level optimizations tailored to DPU resource constraints, thereby overcoming the memory wall with the commercial UPMEM architecture. To the best of our knowledge, Octopus is the first HPM system implemented on commercial PIM hardware.

Other Applications using UPMEM. As the first commercially available PIM architecture, UPMEM has attracted extensive research [31–33, 36, 56] exploring its potential for accelerating diverse applications. UpDLRM [33] leverages UPMEM to increase memory bandwidth and reduce latency in accelerating *Deep Learning Recommendation Models* (DLRM). PIMLex [56] employs a decoupled two-layer design and a PIM-friendly model structure to address the memory-bound nature of

learned indexes. Since the execution flow of HPM fundamentally differs from these applications, our work is orthogonal to theirs.

Data-Space PIM Systems. Existing data-space PIM approaches, such as eager filtering for OLAP scans on SSDs [57–59], primarily reduce host-storage I/O by offloading simple selection predicates to the storage layer and applying them early in a linear scan pipeline. In contrast, HPM is characterized by highly irregular, multi-hop adjacency expansions, whose dataflow and intermediate state complexity substantially exceed what can be captured by per-tuple filtering. Consequently, existing data-space PIM techniques do not naturally align with the characteristics of HPM workloads, a gap that Octopus is designed to address.

PIM Technology. PIM technology integrates computation within or near memory to reduce data movement and deliver low-latency, high-bandwidth execution. Recently, several PIM architectures have been proposed [60–65]. FIMDRAM [60] embeds processing units into DRAM banks based on *High Bandwidth Memory* (HBM) to accelerate machine learning. SK Hynix AiM [61] integrates vector units with GDDR6 for deep learning acceleration. Unlike these domain-specific designs, UPMEM provides a general-purpose and publicly available PIM architecture.

9 Conclusion

This paper presents Octopus, a full-stack hypergraph pattern mining system co-designed with commercial UPMEM PIM hardware. Departing from conventional CPU- and GPU-centric paradigms, Octopus fully exploits the massive internal bandwidth and fine-grained parallelism of the PIM architecture to efficiently execute irregular, data-intensive workloads. Through its inter-DPU coordination framework and intra-DPU mining engine, Octopus orchestrates thousands of lightweight in-order DPUs into a cohesive and scalable computing substrate that bridges algorithmic complexity and architectural constraints. Comprehensive evaluation on real UPMEM hardware demonstrates that Octopus substantially outperforms state-of-the-art HPM and GPM systems, achieving significant performance gains and scalable efficiency across diverse workloads.

Extending hypergraph pattern mining to dynamic hypergraphs and exploring distributed execution are promising directions for future work, given the evolving nature of real-world datasets and the inherent task-level parallelism of HPM workloads.

Acknowledgments

We sincerely thank the reviewers for their helpful comments and feedback. This work is supported by the National Key Research and Development Program of China (No.2023YFB4502300), the National Natural Science Foundation of China (No. 62502172, 62322205, and U25B2023), and Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM103).

References

[1] Lei Li and Tao Li. 2013. News Recommendation via Hypergraph Learning: Encapsulation of User Behavior and News Content. In *Proceedings of the ACM International Conference on Web Search and Data Mining*. 305–314.

[2] Emad Y. Ramadan, Arijit Tarafdar, and Alex Pothen. 2004. A Hypergraph Model for the Yeast Protein Complex Network. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. 189–196.

[3] Dingqi Yang, Bingqing Qu, Jie Yang, and Philippe Cudré-Mauroux. 2019. Revisiting User Mobility and Social Relationships in LBSNs: A Hypergraph Embedding Approach. In *Proceedings of the World Wide Web Conference*. 2147–2157.

[4] Song Feng, Emily Heath, Brett A. Jefferson, Cliff A. Joslyn, Henry Kvinge, Hugh D. Mitchell, Brenda Praggastis, Amie J. Eisfeld, Amy C. Sims, Larissa B. Thackray, Shufang Fan, Kevin B. Walters, Peter J. Halfmann, Danielle Westhoff-Smith, Qing Tan, Vineet D. Menachery, Timothy P. Sheahan, Adam S. Cockrell, Jacob F. Kocher, Kelly G. Stratton, Natalie C. Heller, Lisa M. Bramer, Michael S. Diamond, Ralph S. Baric, Katrina M. Waters, Yoshihiro Kawaoka, Jason E. McDermott,

and Emilie Purvine. 2021. Hypergraph Models of Biological Networks to Identify Genes Critical to Pathogenic Viral Response. *BMC Bioinformatics* 22, 1, Article 287 (2021), 21 pages.

[5] TaeHyun Hwang, Ze Tian, Rui Kuang, and Jean-Pierre A. Kocher. 2008. Learning on Weighted Hypergraphs to Integrate Protein Interactions and Gene Expressions for Cancer Outcome Prediction. In *Proceedings of the IEEE International Conference on Data Mining*. 293–302.

[6] Loc Tran. 2012. Hypergraph and Protein Function Prediction with Gene Expression Data. *CoRR* abs/1212.0388 (2012).

[7] Yi Han, Bin Zhou, Jian Pei, and Yan Jia. 2009. Understanding Importance of Collaborations in Co-Authorship Networks: A Supportiveness Analysis Approach. In *Proceedings of the SIAM International Conference on Data Mining*. 1112–1123.

[8] Xinran Yu and Turgay Korkmaz. 2016. Hypergraph Querying Using Structural Indexing and Layer-Related-Closure Verification. *Knowledge and Information Systems* 46, 3 (2016), 537–565.

[9] Alain Bretto, Hocine Cherifi, and Driss Aboutajdine. 2002. Hypergraph Imaging: An Overview. *Pattern Recognition* 35, 3 (2002), 651–658.

[10] Ping Jian, Keming Chen, and Chenwei Zhang. 2016. A Hypergraph-Based Context-Sensitive Representation Technique for VHR Remote-Sensing Image Change Detection. *International Journal of Remote Sensing* 37, 8 (2016), 1814–1825.

[11] Xi Li, Yao Li, Chunhua Shen, Anthony R. Dick, and Anton van den Hengel. 2013. Contextual Hypergraph Modeling for Salient Object Detection. In *Proceedings of the IEEE International Conference on Computer Vision*. 3328–3335.

[12] Sameer Agarwal, Kristin Branson, and Serge J. Belongie. 2006. Higher Order Learning with Graphs. In *Proceedings of the International Conference on Machine Learning*, Vol. 148. 17–24.

[13] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. 2006. Learning with Hypergraphs: Clustering, Classification, and Embedding. In *Proceedings of the International Conference on Neural Information Processing Systems*. 1601–1608.

[14] Tianming Hu, Hui Xiong, Wenjun Zhou, Sam Yuan Sung, and Hangzai Luo. 2008. Hypergraph Partitioning for Document Clustering: A Unified Clique Perspective. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*. 871–872.

[15] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 425–440.

[16] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: An Efficient Task-Oriented Graph Mining System. In *Proceedings of the European Conference on Computer Systems*. 32:1–32:12.

[17] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 763–782.

[18] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1190–1205.

[19] Yi Zhang, Xiaomeng Yi, Yu Huang, Jingrui Yuan, Chuangyi Gui, Dan Chen, Long Zheng, Jianhui Yue, Xiaofei Liao, Hai Jin, and Jingling Xue. 2025. Cheetah: Accelerating Dynamic Graph Mining with Grouping Updates. *ACM Transactions on Architecture and Code Optimization* 22, 2, Article 79 (2025), 26 pages.

[20] Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 509–523.

[21] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: High Performance Graph Pattern Matching Through Effective Redundancy Elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 100:1–100:14.

[22] Shuangyu Cai, Boyu Tian, Huanchen Zhang, and Mingyu Gao. 2024. PimPam: Efficient Graph Pattern Matching on Real Processing-in-Memory Hardware. *Proceedings of the ACM on Management of Data* 2, 3, Article 161 (2024), 25 pages.

[23] Yi Zhang, Yu Huang, Chaoqiang Liu, Haifeng Liu, Juntao Chen, Jingrui Yuan, Jianhui Yue, Xiaofei Liao, Hai Jin, and Jingling Xue. 2026. Gopher: Efficient Dynamic Graph Pattern Mining via DAG-Driven Execution. In *Proceedings of the European Conference on Computer Systems*.

[24] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *Proceedings of the IEEE International Conference on Data Mining*. 745–754.

[25] Horst Bunke, Peter J. Dickinson, and Miro Kraetzl. 2005. Theoretical and Algorithmic Framework for Hypergraph Matching. In *Proceedings of the International Conference on Image Analysis and Processing*, Vol. 3617. 463–470.

[26] Horst Bunke, Peter J. Dickinson, Miro Kraetzl, Michel Neuhaus, and Marc Stettler. 2008. Matching of Hypergraphs - Algorithms, Applications, and Experiments. In *Applied Pattern Recognition*. Studies in Computational Intelligence, Vol. 91. 131–154.

[27] Tae Wook Ha, Jung Hyuk Seo, and Myoung Ho Kim. 2018. Efficient Searching of Subhypergraph Isomorphism in Hypergraph Databases. In *Proceedings of the IEEE International Conference on Big Data and Smart Computing*. 739–742.

[28] Yuhang Su, Yu Gu, Zhigang Wang, Ying Zhang, Jianbin Qin, and Ge Yu. 2023. Efficient Subhypergraph Matching Based on Hyperedge Features. *IEEE Transactions on Knowledge and Data Engineering* 35, 6 (2023), 5808–5822.
[29] Zhengyi Yang, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Shunyang Li. 2023. HGMatch: A Match-by-Hyperedge Approach for Subgraph Matching on Hypergraphs. In *Proceedings of the IEEE International Conference on Data Engineering*. 2063–2076.
[30] Hao Qi, Kang Luo, Ligang He, Yu Zhang, Minzhi Cai, Jingxin Dai, Bingsheng He, Hai Jin, Zhan Zhang, Jin Zhao, Hengshan Yue, Hui Yu, and Xiaofei Liao. 2025. OHMiner: An Overlap-centric System for Efficient Hypergraph Pattern Mining. In *Proceedings of the European Conference on Computer Systems*. 621–636.
[31] Sitian Chen, Amelie Chi Zhou, Yucheng Shi, Yusen Li, and Xin Yao. 2025. UpANNS: Enhancing Billion-Scale ANNS Efficiency with Real-World PIM Architecture. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 789–804.
[32] Liang-Chi Chen, Chien-Chung Ho, and Yuan-Hao Chang. 2025. Accelerating RNA-Seq Quantification on a Real Processing-in-Memory System. *IEEE Transactions on Computers* 74, 7 (2025), 2334–2347.
[33] Sitian Chen, Haobin Tan, Amelie Chi Zhou, Yusen Li, and Pavan Balaji. 2024. UpDLRM: Accelerating Personalized Recommendation using Real-World PIM Architecture. In *Proceedings of the ACM/IEEE Design Automation Conference*. 211:1–211:6.
[34] UPMEM. 2025. Introduction to UPMEM DIMM. https://www.upmem.com/technology/
[35] UPMEM. 2025. UPMEM SDK User Manual. https://sdk.upmem.com/2025.1.0/index
[36] Shunchen Shi, Fan Yang, Zhichun Li, Xueqi Li, and Ninghui Sun. 2025. Exploring the DIMM PIM Architecture for Accelerating Time Series Analysis. *IEEE Computer Architecture Letters* 24, 1 (2025), 169–172.
[37] Jin Huang, Rui Zhang, and Jeffrey Xu Yu. 2015. Scalable Hypergraph Learning and Processing. In *Proceedings of the IEEE International Conference on Data Mining*. 775–780.
[38] 2025. Intel Advisor. https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html
[39] Aditya Bhaskara, Ravishankar Krishnaswamy, Kunal Talwar, and Udi Wieder. 2013. Minimum Makespan Scheduling with Low Rank Processing Times. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. 937–947.
[40] Yan Lan, Xin Chen, Ning Ding, György Dósa, and Xin Han. 2012. Online Minimum Makespan Scheduling with a Buffer. In *Proceedings of the International Frontiers in Algorithmics, and Proceedings of the International Conference on Algorithmic Aspects in Information and Management*, Vol. 7285. 161–171.
[41] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* 10 (2022), 52565–52608.
[42] Austin R. Benson. 2022. Hypergraph Datasets. https://www.cs.cornell.edu/~arb/data/
[43] 2020. Contact High School Dataset. https://www.cs.cornell.edu/~arb/data/contact-high-school-labeled/
[44] 2020. Contact Primary School Dataset. https://www.cs.cornell.edu/~arb/data/contact-primary-school-labeled/
[45] 2020. Senate Bills Dataset. https://www.cs.cornell.edu/~arb/data/senate-bills/
[46] 2020. House Bills Dataset. https://www.cs.cornell.edu/~arb/data/house-bills/
[47] 2020. Walmart Trips Dataset. https://www.cs.cornell.edu/~arb/data/walmart-trips/
[48] 2020. Trivago Clicks Dataset. https://www.cs.cornell.edu/~arb/data/trivago-clicks/
[49] 2020. Stack Overflow Answers Dataset. https://www.cs.cornell.edu/~arb/data/stackoverflow-answers/
[50] 2020. Amazon Reviews Dataset. https://www.cs.cornell.edu/~arb/data/amazon-reviews/
[51] Yufeng Gu, Alireza Khadem, Sumanth Umesh, Ning Liang, Xavier Servot, Onur Mutlu, Ravi Iyer, and Reetuparna Das. 2025. PIM Is All You Need: A CXL-Enabled GPU-Free System for Large Language Model Inference. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 862–881.
[52] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim. 2023. Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs. *Proceedings of the ACM on Management of Data*, Article 113 (2023), 27 pages.
[53] Gilbert Jonatan, Haeyoon Cho, Hyojun Son, Xiangyu Wu, Neal Livesay, Evelio Mora, Kaustubh Shivdikar, José L. Abellán, Ajay Joshi, David Kaeli, and John Kim. 2024. Scalability Limitations of Processing-in-Memory using Real System Evaluations. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, Vol. 8. Article 5, 28 pages.
[54] Dongjae Lee, Bongjoon Hyun, Taehun Kim, and Minsoo Rhu. 2024. Analysis of Data Transfer Bottlenecks in Commercial PIM Systems: A Study With UPMEM-PIM. *IEEE Computer Architecture Letters* 23, 2 (2024), 179–182.
[55] Si Ung Noh, Junguk Hong, Chaemin Lim, Seongyeon Park, Jeehyun Kim, Hanjun Kim, Youngsok Kim, and Jinho Lee. 2025. PID-Comm: A Fast and Flexible Collective Communication Framework for Commodity Processing-in-DIMM Devices. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*. 245–260.
[56] Lixiao Cui, Kedi Yang, Yusen Li, Gang Wang, and Xiaoguang Liu. 2025. PIMLex: A High-Performance Learned Index with Processing-in-Memory. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 287–303.

[57] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*. 1–13.

[58] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*. 153–165.

[59] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. 2022. Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads. *Proceedings of the VLDB Endowment* 15, 10 (2022), 1991–2004.

[60] Young-Cheon Kwon, Sukhan Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je-Min Ryu, Jong-Pil Son, Seongil O, Hak-soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyunsung Shin, Jin Kim, BengSeng Phuah, Hyoungmin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, Sooyoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, Joon-Ho Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 2021. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In *Proceedings of the IEEE International Solid-State Circuits Conference*. 350–352.

[61] Seong Ju Lee, Kyu-Young Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dong Yoon Ka, Kyu-Dong Hwang, Jeongje Park, Kyeong Pil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Kornijcuk Vladimir, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Jaewook Lee, Donguc Ko, Younggun Jun, Keewon Cho, Ilwoong Kim, Choungki Song, Chunseok Jeong, Dae-Han Kwon, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. 2022. A 1Ynm 1.25V 8Gb, 16Gb/s/Pin GDDR6-Based Accelerator-in-Memory Supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications. In *Proceedings of the IEEE International Solid-State Circuits Conference*. 1–3.

[62] Haifeng Liu, Long Zheng, Yu Huang, Chaoqiang Liu, Xiangyu Ye, Jingrui Yuan, Xiaofei Liao, Hai Jin, and Jingling Xue. 2023. Accelerating Personalized Recommendation with Cross-level Near-Memory Processing. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*. 66:1–66:13.

[63] Chaoqiang Liu, Haifeng Liu, Dan Chen, Yu Huang, Yi Zhang, Wenjing Xiao, Xiaofei Liao, and Hai Jin. 2025. Heter-RAG: Heterogeneous Processing-in-Memory Acceleration for Retrieval-augmented Generation. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*. 884–898.

[64] Yu Huang, Long Zheng, Pengcheng Yao, Qinggang Wang, Xiaofei Liao, Hai Jin, and Jingling Xue. 2022. Accelerating Graph Convolutional Networks Using Crossbar-based Processing-In-Memory Architectures. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 1029–1042.

[65] Yu Huang, Long Zheng, Pengcheng Yao, Jieshan Zhao, Xiaofei Liao, Hai Jin, and Jingling Xue. 2020. A Heterogeneous PIM Hardware-Software Co-Design for Energy-Efficient Graph Processing. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. 684–695.

Received October 2025; revised January 2026; accepted February 2026