PimPam: Efficient Graph Pattern Matching on Real Processing-in-Memory Hardware

SHUANGYU CAI, Tsinghua University, China
BOYU TIAN, Tsinghua University, China
HUANCHEN ZHANG, Tsinghua University, China and Shanghai Qi Zhi Institute, China
MINGYU GAO, Tsinghua University, China and Shanghai Qi Zhi Institute, China

Graph pattern matching is powerful and widely applicable to many application domains. Despite the recent algorithm advances, matching patterns in large-scale real-world graphs still faces the memory access bottleneck on conventional computing systems. Processing-in-memory (PIM) is an emerging hardware architecture paradigm that puts computing cores into memory devices to alleviate the memory wall issues. Real PIM hardware has recently become commercially accessible to the public. In this work, we leverage the real PIM hardware platform to build a graph pattern matching framework, PimPam, to benefit from its abundant computation and memory bandwidth resources. We propose four key optimizations in PimPam to improve its efficiency, including (1) load-aware task assignment to ensure load balance, (2) space-efficient and parallel data partitioning to prepare input data for PIM cores, (3) adaptive multi-threading collaboration to automatically select the best parallelization strategy during processing, and (4) dynamic bitmap structures that accelerate the key operations of set intersection. When evaluated on five patterns and six real-world graphs, PimPam outperforms the state-of-the-art CPU baseline system by 22.5× on average and up to 71.7×, demonstrating significant performance improvements.

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

Additional Key Words and Phrases: Graph Pattern Matching, Processing in Memory

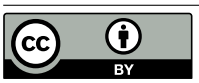
ACM Reference Format:
Shuangyu Cai, Boyu Tian, Huanchen Zhang, and Mingyu Gao. 2024. PimPam: Efficient Graph Pattern Matching on Real Processing-in-Memory Hardware . *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 161 (June 2024), 25 pages. https://doi.org/10.1145/3654964

1 INTRODUCTION

Graph mining is a powerful class of algorithms that has seen wide applications in many fields, including bioinformatics [14], cheminformatics [44], and social network [41]. Graph pattern matching represents an important problem in graph mining. Given a graph, it asks about the number of subgraphs (called embeddings) that are isomorphic to a certain pattern. Unfortunately, it is a well-known NP-Complete problem. The number of potential embeddings grows exponentially as the graph size increases.

To address the extensive computational complexity, general graph pattern matching systems [37, 52, 56] have been designed, allowing users to express pattern queries at a high abstraction level

Authors' addresses: Shuangyu Cai, caisy21@mails.tsinghua.edu.cn, Tsinghua University, Beijing, China; Boyu Tian, tby20@mails.tsinghua.edu.cn, Tsinghua University, Beijing, China; Huanchen Zhang, huanchen@tsinghua.edu.cn, Tsinghua University, Beijing, China and Shanghai Qi Zhi Institute, Shanghai, China; Mingyu Gao, gaomy@tsinghua.edu.cn, Tsinghua University, Beijing, China and Shanghai Qi Zhi Institute, Shanghai, China.



ACM 2836-6573/2024/6-ART161
https://doi.org/10.1145/3654964

and still benefit from performance-optimized low-level implementations. Unfortunately, these designs are still inefficient and appear orders of magnitude slower than hand-written programs. In recent years, specialized pattern-aware frameworks [27, 38, 39, 48] are proposed to narrow down this gap. By exploiting specific features of the to-be-matched patterns, these systems are able to automatically generate efficient algorithms that have comparable performance to hand-optimized implementations.

However, with continuously growing graph sizes, more complex connection structures, and highly irregular degree distributions, the memory access cost has become a significant bottleneck in graph pattern matching in addition to the computational complexity [4, 15, 51]. The aforementioned state-of-the-art frameworks require heavy memory accesses. Take triangle counting, one of the simplest patterns, as an example. Its data access volume could be up to 451 times of the original graph size [15], reaching tens of gigabytes on large real-world graphs. Such excessive data access demands severely stress the limited memory bandwidth in conventional hardware systems and lead to substantial energy consumption [49].

On the other hand, researchers are routinely exploring new types of hardware architectures to overcome the "memory wall". Processing-in-Memory (PIM) is one promising paradigm of such kind. It moves computation closer to data locations by adding computing units into memory, with the choices ranging from simple logic units [46, 47] to general-purpose cores [17]. Since data could now be processed locally in memory chips, data transfers are reduced, which saves both bandwidth and energy. Furthermore, a large number of PIM units could be placed in the memory, each with independent links to access its local memory region fully in parallel, without sharing the narrow processor-memory channels. This enables excessively high bandwidth for PIM computation. Recently, real PIM hardware platforms are becoming commercially available to the market [17, 24, 45], among which UPMEM [17, 55] is the first publicly accessible PIM system. This makes it possible to accelerate memory-intensive applications such as graph pattern matching on real PIM hardware.

However, fully exploiting the abundant computation and bandwidth resources offered by the UPMEM hardware is challenging, particularly with the complex and irregular graph pattern matching algorithms. For example, UPMEM contains thousands of PIM cores, each of which further supports tens of hardware threads. Efficiently making use of these many computational capabilities requires careful parallelization strategies with good load balance. Furthermore, despite the high access bandwidth, the local memory capacity of each PIM core is quite limited, only up to tens of megabytes. The large graph data must be meticulously partitioned and formatted so that all the required input data for each thread can fit in the local memory. Finally, the PIM cores can only run at relatively low speed. The performance of each single thread also needs to be improved by optimizing the key computation operations.

In this work, we propose PimPam, which to our best knowledge is the first graph pattern matching framework implemented on real-world, commercially available PIM hardware platforms, i.e., UPMEM [17]. PimPam incorporates several novel optimizations to achieve efficient resource utilization and to accommodate UPMEM's practical limitations. First, to ensure load balance, the *load-aware task assignment* phase in PimPam uses an accurate and simple prediction model to estimate the execution time of each pattern matching task, and statically assigns them to the PIM cores to achieve uniform loads. This is because UPMEM does not support direct communication between PIM cores, and thus dynamic work stealing is impossible. Second, PimPam leverages a *space-efficient subgraph format* to store the necessary input data compactly in the limited local memory of each PIM core, after a *parallel graph partitioning* phase that is also offloaded to the many PIM cores in order to reduce the pre-processing overheads. Third, during the actual pattern matching

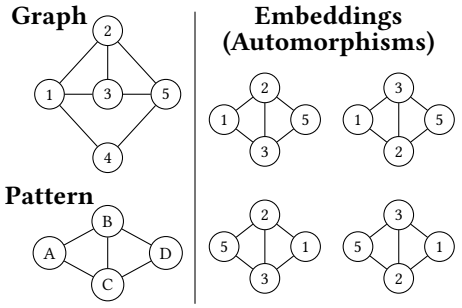


Fig. 1. An example graph pattern matching problem. Four embeddings are found, but all of them are automorphisms of the same subgraph. So they are counted only once.

```
1 for v_B ∈ V do
2   for v_C ∈ N(v_B) do
3     if id(v_C) > id(v_B)
        then break ;
4     S_BC ←
        N(v_B) ∩ N(v_C);
5     for v_A ∈ S_BC do
6       for v_D ∈ S_BC do
7         if id(v_D) >
            id(v_A) then
            break ;
8         ans++;
```

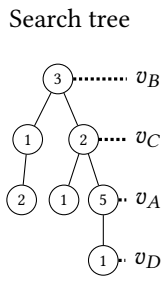


Fig. 2. Left: The pattern matching algorithm for the pattern in Figure 1. $N(v)$ denotes the neighbor list of v. Right: The search tree rooted at vertex 3.

phase, we propose an *adaptive thread collaboration* mechanism in PimPam, which automatically selects the best level and amount of parallelism according to the current task's structure and load. It allows for efficient use of the multiple hardware threads in one PIM core even in the presence of excessively large tasks under skewed degree distributions. Fourth, we also design a *dynamic bitmap structure* to optimize the computation efficiency for clique patterns. Only when beneficial, it dynamically builds bitmap data structures for a small confined part of the graph, to accelerate the key operations of set intersection in the algorithm.

We evaluate PimPam and demonstrate its superior performance by comparing it to the state-of-the-art graph pattern matching system on general-purpose CPUs [48]. PimPam outperforms the baseline by 22.5× on average and up to 71.7×, across five patterns on six real-world graphs. The combination of the above optimizations ensures efficient and load-balancing processing, resulting in such significant improvements. Although the pre-processing in PimPam is more challenging, the corresponding overheads of task assignment and data partitioning are well minimized, due to our design choice and offloading technique. The pre-processing time is comparable to the CPU baseline. Finally, we show that PimPam has comparable performance with existing GPU-based graph pattern matching frameworks [12], while being free from out-of-memory issues.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce the background of the graph pattern matching problem and its state-of-the-art algorithm solutions, as well as the commercial processing-in-memory (PIM) architecture we use in this work. Then, we present the technical challenges of accelerating graph pattern matching on PIM.

2.1 Graph Pattern Matching Algorithms

Problem definition. In the graph pattern matching problem, a graph $G(V, E)$ and a pattern $P(V_p, E_p)$ are given, where V, V_p denote the vertex sets and $E \subseteq V \times V$, $E_p \subseteq V_p \times V_p$ denote the edge sets. In this work, we focus on the case where both G and P are undirected, simple (i.e., without self-loops or multiple edges), and unlabeled graphs. An **embedding** of P in G is a bijection $f : V_p \rightarrow V' \subseteq V$ such that $\forall\ (u, v) \in E_p$, $(f(u), f(v)) \in E$. Each embedding therefore maps to a subgraph in the full graph G.

The graph pattern matching problem aims to discover all **distinct** embeddings in G given the pattern P and count the total number of matches. By "distinct", we mean that automorphisms of the same subgraph should be eliminated and counted only once. Figure 1 gives an example. The diamond pattern itself has four automorphisms, which means $[A, B, C, D]$ can be remapped to $[A, B, C, D]$, $[A, C, B, D]$, $[D, B, C, A]$, $[D, C, B, A]$, but still preserves the shape. Therefore, although four embeddings can be found, there is only one distinct embedding that counts.

State-of-the-art algorithms. GraphPi [48] represents the state-of-the-art pattern matching systems executing on CPUs, which is optimized upon AutoMine [39] and GraphZero [38]. It takes in a graph and a pattern, and automatically generates optimized code to complete the task. A typical algorithm generated by GraphPi is shown in Figure 2 (left), which solves the problem in Figure 1. Vertices in P are matched to G one by one, following a specifically determined order called the **schedule**, e.g., $B \rightarrow C \rightarrow A \rightarrow D$ in this example. First, B is matched to a vertex v_B in G (Line 1). Next, C must be matched to a neighbor of v_B (Line 2), as C and B are connected in the pattern (see Figure 1). Similarly, A and D must be matched to the common neighbors of v_B and v_C according to the pattern, so an intersection of the two neighbor lists should be done first (Line 4), whose result is the **candidate set** of the vertices that may be matched to A and D (Lines 5 and 6). The graph is stored in the Compressed Sparse Row (CSR) format, with the neighbor list of each vertex sorted, so intersection can be done by sequentially scanning and merging two lists in linear time.

To eliminate repeated counting of automorphisms, specific **restrictions** are added such that each distinct embedding is counted exactly once [38]. This is called **symmetry breaking**. For example, two restrictions $\mathrm{id}(v_C) \not> \mathrm{id}(v_B)$ and $\mathrm{id}(v_D) \not> \mathrm{id}(v_A)$ are enforced at Lines 3 and 7. In this way, the symmetry between B and C and between A and D can be broken, so only one of the four automorphisms in Figure 1 would be counted.

The combination of a schedule and a set of restrictions is called a **configuration**. Different configurations result in significantly different performance [48]. Given the pattern, GraphPi enumerates through all possible configurations (with some pruning to reduce the search space), and uses a cost model to select the best one.

The whole process above is usually described as traversing **search trees** in a depth-first search (DFS) manner [38, 39, 48]. A search tree with the above configuration is shown in Figure 2 (right), where B is matched to vertex 3 in G. Different matches to the root v_B (i.e., each iteration of the loop in Line 1) correspond to different search trees. At each tree node, the algorithm computes the candidate set for the next vertex (mainly using intersection operations). For example, vertex 3 has neighbors 1, 2, and 5 in G, but only 1 and 2 satisfy the restriction in Line 3, so tree node 3 has child nodes of 1 and 2. The algorithm first explores branch 1, but the next candidate set S_{BC} has only one element 2, which cannot be matched with both A and D. So no embedding is found along this branch. Then it explores branch 2, now with $S_{BC} = \{1, 5\}$, and thus it is extended with two children, and one of them leads to a valid embedding.

Note that it is also possible to traverse the search trees in the breadth-first search (BFS) manner [50, 54, 58, 61]. BFS could provide more available parallel workloads during the processing, by simultaneously exploring multiple tree branches in parallel. However, BFS also generates a large amount of intermediate data along all these branches, which significantly increases the memory footprint. Therefore excessively exploiting BFS may not be desired.

Additional optimizations. We next introduce two additional optimizations in graph pattern matching algorithms. Our design also supports them to improve performance.

The first one is called **orientation**. For a vertex, the number of its neighbors with smaller IDs than the vertex itself is called the vertex's **oriented degree**. With symmetry breaking, the oriented degree of a previously matched vertex sometimes determines the number of loop iterations for a

vertex matched next. For example, in Figure 2, the number of valid v_C choices at Line 2 is just the oriented degree of v_B due to the restriction at Line 3. The orientation optimization pre-processes the graph G to re-number its vertices in the order of their degrees, such that higher-degree vertices have smaller vertex IDs [38, 39]. Although it does not reduce the total oriented degrees, orientation effectively reduces the maximum oriented degree. Because the workload associated with each vertex is superlinear to the oriented degree, the total complexity is reduced.

The second optimization is to apply the **Inclusion-Exclusion Principle (IEP)** to directly calculate the number of embeddings without iterating the loops [48]. For a certain nested loop structure, if there are at least two innermost `for` loops that do not conduct intersections (e.g., Lines 5 and 6 in Figure 2), then the total number of embeddings can be directly counted by IEP. Specifically, Lines 5 to 8 in Figure 2 can be replaced with `ans +=` $|S_{BC}| \times (|S_{BC}| - 1)/2$.

2.2 PIM Hardware — UPMEM

In recent years, energy efficiency has become a major bottleneck that impedes the development of computing systems, a large fraction of which is caused by data movement between the processor and the memory system [5, 42], known as the "memory wall" problem. **Processing-in-Memory (PIM)** is found to be a promising way to deal with this issue. By adding various computing units into or close to the memory, ranging from simple logic units [46, 47] to general-purpose cores [17], data could be processed locally in the memory chips without being transferred on the processor-memory bus. Not only energy consumption is reduced, but also higher bandwidth can be obtained since memory accesses by the PIM cores can be made in parallel and independently, without congesting the narrow and shared bus.

Although the idea of PIM has been extensively discussed in academia for decades, real PIM hardware was not commercially available until recently [17, 24, 45]. Among them, UPMEM [17, 55] is the first publicly accessible PIM hardware, and we use it as our PIM platform in this work. As shown in Figure 3, in the UPMEM system, the host CPU has extra PIM-enhanced memory in addition to the traditional DDR4 memory. The PIM memory consists of PIM-enhanced DRAM chips, organized in the standard DIMM (Dual In-line Memory Module) form factor. Inside each PIM chip, in addition to the DRAM banks, several general-purpose cores called **DPUs** are added as the PIM units. Each DPU has private connection to the local memory banks, so the aggregated bandwidth linearly scales with the number of DPUs, and is not limited by the shared memory channels as in conventional architectures. UPMEM can hold up to 20 DIMMs, 16 chips per DIMM, and 8 DPUs per chip. Therefore, a maximum of 2560 DPUs can work simultaneously with up to 7.2 TB/s data access bandwidth, providing abundant computing parallelism as well as excessive memory bandwidth to these internal DPUs.

Each DPU in UPMEM is an in-order 32-bit core that uses a RISC-style instruction set architecture, and supports up to 24 hardware threads running at 350 MHz. At least 11 active threads are needed to fully utilize its 14-stage pipeline; otherwise the computation resources may be wasted. Therefore it is critical to perform sufficient multi-threading processing. Each DPU has a 64 MB private local memory bank as its main memory (**MRAM**), as well as a 64 kB data cache (**WRAM**) and a 24 kB instruction cache (**IRAM**). The WRAM can be accessed by load/store instructions in one cycle, but the DPU has no direct access to the MRAM. A DMA engine, which is triggered by software using the runtime API functions `mram_read` and `mram_write`, is responsible for data transfers between MRAM and WRAM. The measured DMA latency is $(\alpha + \beta \cdot \text{size})$ cycles, where α is 77 for reads and 61 for writes, and β is 0.5 per byte [20]. This latency can be hidden if multiple threads are actively executing and time multiplexed on the DPU. Furthermore, a DPU has no way to directly access the MRAM of another DPU, i.e., no inter-DPU communication. Such data communication must by facilitated by the host CPU.

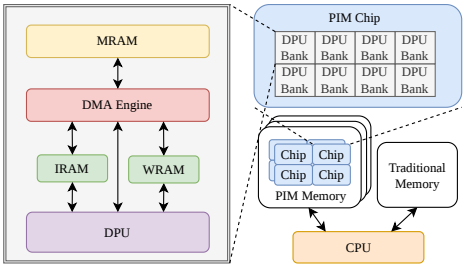


Fig. 3. UPMEM hardware architecture.

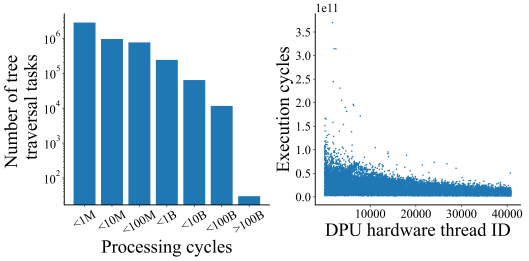


Fig. 4. Load imbalance example. Left: Histogram of cycles taken by each task. Right: Execution time of each thread under round-robin task assignment.

The CPU offloads computation to the DPUs by launching a kernel. In the current UPMEM system, once the kernel is launched, the CPU cannot control or communicate with the DPUs until the kernel is completed. Hence all data preparation and communication must take place before kernel launching. There are three ways for CPU-DPU data communication: (1) broadcast the same data to all DPU MRAMs through `dpu_broadcast_to`; (2) simultaneously transfer different data of the same size to or from DPUs through `dpu_prepare_xfer` and `dpu_push_xfer`; (3) transfer with a single DPU through `dpu_copy_to` and `dpu_copy_from`. The bandwidth of (1) is almost linear to the number of DPUs, while that of (2) is sublinear, and clearly (3) gives the lowest bandwidth. A big transfer size is preferred due to the DMA triggering overhead [20].

2.3 Design Challenges

Although UPMEM provides plentiful computation units (up to 2560 DPUs) with abundant memory bandwidth, fully exploiting these resources to maximally accelerate graph pattern matching is not trivial. We list the potential design challenges below.

First, with thousands of parallel DPUs, it is crucial to evenly assign workloads to them to ensure good *load balancing*. This is particularly challenging for graph pattern matching, not only because the graph itself is highly irregular, sparse, and skewed [19], but the complex search tree also results in various workloads for each vertex. We illustrate the issue of load imbalance in Figure 4, which runs the House pattern (Figure 6) on the `LiveJournal` [30] graph. On the left, we see that the processing time of search tree traversals starting from different vertices in the graph can significantly differ by many orders of magnitude, exhibiting a highly skewed distribution. If we naively assign these tasks to the DPUs in a simple round-robin fashion (assuming 2560 DPUs and 16 threads per DPU), the few large tasks will cause some threads to execute for excessively long time, while other threads remain idle and wasted, as in Figure 4 (right). Note that there are two levels of load balancing that we need to address in UPMEM. The first level is among different DPUs, which must be done statically due to the inability to transfer data across DPUs during processing. The second level is among the multiple hardware threads within one DPU, which allows more flexible and dynamic policies at runtime.

Second, because of the limitations in DPU-CPU and inter-DPU communication, all the input data needed by each DPU must be properly loaded into the DPU's MRAM before the processing starts. This *data preparation* time cannot be overlapped with data processing, and thus its overhead must be minimized. Note that due to the small capacity of MRAM, we cannot blindly transfer the entire input graph to each DPU. The data partitioning method must be both time- and space-efficient.

Third, the abundant bandwidth available to each DPU not only alleviates the memory access overheads, but actually may shift the bottleneck back to the *computation efficiency*, especially with the relatively low frequency of DPUs. The key operations in graph pattern matching are set

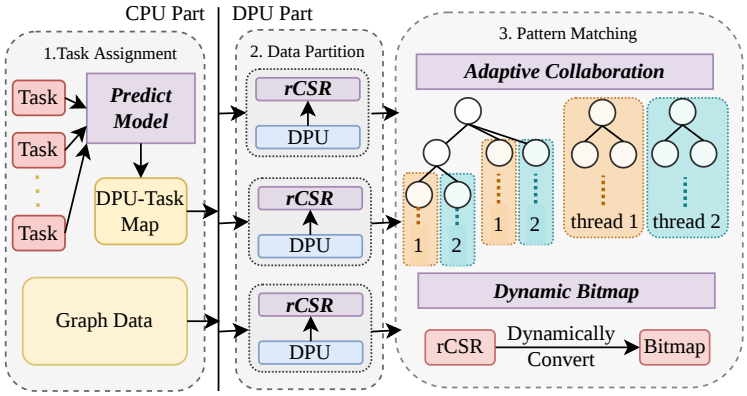


Fig. 5. Overview of PimPam. The key optimizations are marked in purple.

intersections. It might be necessary to improve the intersection algorithm when possible, to tolerate the limited speed of DPUs.

Note that the above challenges for UPMEM are different from previous designs targeting multi-core and GPU systems. First, each DPU can only access its limited private memory, which is a limitation not existing in CPU/GPU systems. Second, there is no direct communication between DPUs, so task assignment and data partitioning should be pre-determined, and runtime work stealing is infeasible. Third, the number of DPUs is extremely large, but the capacity of each DPU is low, calling for more balanced workload assignment.

3 PIMPAM OVERVIEW

In this paper, we propose PimPam to solve the challenges above and accelerate graph pattern matching on the UPMEM system. To the best of our knowledge, PimPam is the first graph pattern matching algorithm implemented on real-world PIM architectures.

Figure 5 shows the overall execution flow of PimPam, as well as our key techniques (marked in purple) to optimize its performance. The execution is composed of three phases, namely **task assignment**, **data partitioning**, and **pattern matching**. The first phase executes on the CPU, while the other two are offloaded to the DPUs to enjoy the abundant parallel computation resources. Note that DPUs in UPMEM can be launched in an asynchronous manner. Other jobs can run on the CPU simultaneously with DPU processing to avoid idle resources, or multiple jobs can pipeline between the CPU and DPUs.

The basic unit of workload assignment among DPUs in PimPam is a *task*, which traverses the entire search tree rooted at a specific vertex in the graph, e.g., Figure 2 (right) with $v_B = 3$. Different tasks can be executed fully independently. We choose such a workload splitting level because it exposes sufficient parallelism among different root vertices, and also it avoids excessive memory footprint overheads that may not fit in the limited DPU MRAM. Note that theoretically we could use more fine-grained tasks, such as dividing a search tree into several subtrees following different branches from the root. However, all these subtrees would require similar graph data around the same root vertex to process. Distributing these subtrees across different DPUs leads to extensive data duplication that increases the total memory footprint. We discuss this capacity issue in Section 4.2 in more details. Nevertheless, the more fine-grained parallelism within one search tree is beneficial for load balancing within one DPU, as will be elaborated in Section 4.3.

We next describe the overall execution flow of PimPam. First, with each vertex in the graph representing the search tree root of a task, the host CPU first executes the task assignment phase

(Section 4.1) to decide the mapping between tasks and DPUs. To ensure load balance among DPUs, our assignment is aware of the various loads of the tasks. We propose a *task time prediction model*, which is both accurate to ensure good load balance, and also simple to evaluate without expensive graph traversals, so that the CPU phase does not become a significant performance bottleneck compared to the later parallel phases on the DPUs.

With the list of tasks assigned to each DPU, the next data partitioning phase prepares the input data needed by the DPU and stores them into the DPU MRAM (Section 4.2). Instead of using the CPU, this phase is designed to be *parallel* and to leverage the abundant DPU resources, so that each DPU discovers its required subgraph in a fully parallel manner. We propose an *rCSR format* as a space-efficient way to store each DPU input subgraph in the limited MRAM. By exploiting the specific structure of the pattern, we are able to reduce the required memory size by a factor of the graph's average degree, which is up to 20 in large real-world graphs.

After that, the pattern matching kernels are launched on the DPUs. To fully utilize the multiple hardware threads within a DPU and ensure load balance among them, we propose an *adaptive thread collaboration* mechanism that can automatically select the best level and amount of parallelism in the task's search tree (Section 4.3). This further alleviates the issue caused by excessive large tasks, by splitting their processing to multiple threads in a DPU. Finally, to improve computation efficiency, we use *dynamic bitmap structures* to accelerate clique pattern matching (Section 4.4). Rather than creating bitmaps for the entire graph which would be impractically large, our optimization dynamically identifies a small part of the graph with a well bounded size to construct bitmaps, so that intersections can be done more efficiently than linear merging.

We emphasize our work is orthogonal to the previous work [38, 39, 48], where they mainly focused on how to generate optimized configurations (schedules, restrictions, etc.) for given patterns and graphs. PimPam is compatible with these optimized configurations, and could directly take them as input and follow the discovered configurations.

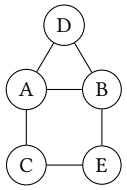
4 PIMPAM DESIGN

We describe the detailed designs of the four optimizations in PimPam: load-aware task assignment, space-efficient and parallel data partitioning, adaptive thread collaboration, and dynamic bitmaps.

4.1 Load-Aware Task Assignment

Recall from Section 3 that in PimPam, a task is defined as an entire search tree rooted at some vertex, and is the smallest unit of workload assignment. When tasks are assigned to DPUs, the total execution time is bounded by the most loaded DPU, so it is desired to assign tasks as evenly as possible in terms of their processing time. To estimate the task processing time, we need a prediction model that is both accurate and also simple to calculate. However, *the accuracy and complexity goals exhibit a challenging tradeoff*. On the one hand, the time of a task is determined by the irregular structure of the search tree (see Figure 2), where the number of branches that could be spawned from each tree node follows the cardinality of the candidate set as a result of the intersection of vertex neighbor lists, whose sizes are in turn determined by the irregular input graph topology. Thus a naive model would unlikely work well. On the other hand, note that the actual pattern matching phase is parallelized across thousands of DPUs, but the task assignment phase executes on the CPU. Therefore the assignment must be sufficiently fast to avoid dominating the overall performance, which forbids overly complex models.

In our model, we limit the available input information to the statistics of the search tree root itself (i.e., the degree d and the oriented degree d_o), and the overall statistics of the whole graph (i.e., the average degree d_{avg}). The root degrees are the only information that can be immediately used for each task; any other statistics (e.g., for a neighbor) would need extra graph traversal hops.




```
1 for v_A ∈ V do
2   for v_B ∈ N(v_A) do
3     if id(v_B) > id(v_A) then break ;
4     S_AB ← N(v_A) ∩ N(v_B);
5     for v_C ∈ N(v_A) do
6       S_BC ← N(v_B) ∩ N(v_C);
7       ans += |S_AB| × |S_BC| − |S_AB ∩ S_BC|;
```

Fig. 6. The "House" pattern and the corresponding algorithm. The schedule is $[A, B, C, D, E]$ with a restriction $\mathrm{id}(v_B) \not> \mathrm{id}(v_A)$. The last line uses IEP to directly count the number of combinations of D and E.

The average degree only needs to be calculated once and can be reused across all tasks. Therefore, this choice of input data represents a minimum overhead. Our evaluation in Table 4 shows that this limited available information already allows us to build a sufficiently accurate prediction model, and using additional statistics only brings marginal benefits (and sometimes unstability) but significantly increases the cost of task assignment itself, and thus may be not worth it.

Our prediction model is built upon two assumptions. First, the processing time is dominated by the innermost loop body since it will be executed the most times. When IEP is not applied, the innermost loop body is just a simple counting (e.g., Figure 2), so we only need to calculate the total number of loop iterations. With IEP, the innermost loop body usually contains intersection operations (e.g., Figure 6 Lines 6 and 7), whose cost must also be considered.

Second, we assume the vertices other than the root have the average degree d_{avg}, and the edges not neighboring to the root are uniformly distributed. With the limited input information described above, this is the best we can know about the other vertices. Although it seems overly simplified, we note that our goal here is to estimate the cardinalities of the candidate sets which are intersections of neighbor lists. Even with real-world power-law graphs whose vertex degree distribution is highly skewed [19], this assumption is still reasonable in that the cardinality of the intersection result mostly depends on the smaller list instead of the excessively large one. So the impact of high-degree vertices is minor.

Task time prediction model. Now we describe our prediction model. First, we estimate the cardinality of the candidate set for a vertex, which is also the number of iterations for a loop in a specific level in the algorithm, e.g., Line 5 in Figure 2. Such a candidate set is typically a neighbor list or the intersection of several neighbor lists. For a vertex in the pattern (e.g., A in Figure 2), if $b \in \{0, 1\}$ denotes whether it is a neighbor of the root in the pattern, and k denotes the number of vertices other than the root it is neighboring to in the pattern, then the probability for a vertex in the graph to become a candidate is given as

$$\left(\frac{d}{|V|}\right)^b \times \left(\frac{d_{\text{avg}}}{|V|}\right)^k \propto d^b \tag{1}$$

where $|V|$ is the number of vertices in the graph. For example, A at Line 5 in Figure 2 has $b = 1$ and $k = 1$. Furthermore, if there is a restriction requiring the vertex ID to be smaller than the root, d_{o} should be used instead of d, such as Line 2 in Figure 6.

When IEP is not applied, we directly multiply together the estimated cardinality of each `for` loop to compute the workload. Here we can simplify all constants like $|V|$ and d_{avg} into an opaque constant factor without actually using their concrete values.

If IEP is used, we also need to consider the time for the intersections in the innermost loop. Since the neighbor lists are sorted, the intersection cost is $O(n_1 + n_2)$, and we just need to estimate the

cardinalities n_1 and n_2 of the two sets involved, with the same approach above. The only difference is that we cannot omit the constants now. For example, Lines 6 and 7 in Figure 6 involve two intersections. $|N(v_B)|$ and $|N(v_C)|$ are estimated using d_{avg}. $|S_{AB}|$ and $|S_{BC}|$ can be estimated using $d \cdot d_{\text{avg}}/|V|$ and $d_{\text{avg}}^2/|V|$, respectively.

Previous designs like AutoMine [39] and GraphPi [48] also used prediction models, but their goal was to estimate the performance of the entire execution given a configuration, not specific to each individual task. Therefore their models were more coarse-grained and less accurate. Specifically, AutoMine pre-defined a constant filtering factor for intersection operations, which was less accurate than ours. GraphPi used the number of triangles to predict the intersection results. Although it was accurate, it is not suitable for PimPam since it would now require the number of triangles rooted at each individual vertex, causing unacceptable overheads.

Task assignment algorithm. After the processing time of each task is estimated, tasks should be assigned to DPUs as balanced as possible. The question becomes, given n numbers $w_1, w_2, \ldots, w_n$, divide them into m groups such that the maximum sum is minimized. This problem has long been proved to be NP-complete, so we use the following approximated algorithm. All tasks are sorted in the decreasing order of their predicted processing time. Then, iterate through each task and assign it to the currently least loaded DPU, until all tasks are assigned. In practice, considering that each task execution has some extra overheads, a small empirical constant is added to the predicted processing time. This prevents large amounts of extremely small tasks from being assigned to the same DPU, which may overwhelm the DPU with the unmodeled secondary overheads.

4.2 Space-Efficient & Parallel Data Partitioning

With the task assignment result from Section 4.1, we will next prepare the input data for each DPU. Note from Section 2.2 that each MRAM has only 64 MB capacity, far less than a typical graph size of several GBs. Moreover, the DPUs cannot communicate with each other to exchange data. Therefore we must properly partition the input graph so that all "necessary" data of the assigned tasks can be found in the local MRAM of each DPU. The data partitioning method should satisfy two requirements: (1) in terms of *space*, each data partition for a DPU must be minimized so that it can fit in the limited MRAM even for large graphs; (2) in terms of *time*, the partitioning and data transfers should be sufficiently fast, ideally in a parallel manner, to avoid becoming the performance bottleneck. We discuss how PimPam meets these two goals below.

Space-efficient rCSR format. We need to first determine what data are "necessary" for a task. Given a pattern P with a schedule $[v_1, v_2, \ldots, v_k]$, we consider a task rooted at u_1, i.e., $u_1 = f(v_1)$ is the vertex in the graph G that maps to v_1.[1] Apparently, if the maximum distance between v_1 and any other vertices in P is $D_{\max}$, then only the vertices in G within distance $D_{\max}$ to u_1 may be included in the embedding, and only their information is required to complete the task. A naive method is thus to generate the subgraph induced by those vertices following the conventional CSR format. Unfortunately, this approach is not sufficiently space-efficient for large graphs. In our experiments, DPUs run out of memory with the `LiveJournal` graph [30] even for simple patterns of $D_{\max} = 2$. Further optimizations are thus required.

In our design, we make a key observation that, not all the neighbor lists of those vertices within the $D_{\max}$-neighborhood are necessary. For example, in Figure 6, the neighbor lists of vertices v_D and v_E are never used by the algorithm. Actually, to derive the candidate set for v_i, the intersection only requires the neighbor lists of those vertices that (1) are neighbors of v_i in P, and (2) appear

[1] In this subsection, we use v to denote a vertex in the pattern P, and u to denote a vertex in the graph G.

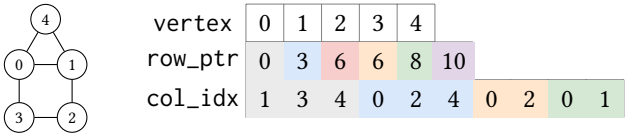


Fig. 7. An example of the proposed rCSR format to store the graph on the left. Vertex 0 is the root. $D_{\max} = 2, \hat{D} = 1$. The neighbor list of vertex 2 is not stored.

before v_i in the schedule. Formally speaking in another word, the neighbor list of $u_a = f(v_a)$ is required if and only if $\exists\ (v_a, v_b) \in E_P$ such that v_b appears after v_a in the schedule.

Therefore, if there are no edges among the vertices at distance $D_{\max}$ in the pattern P, then there exists a schedule where only the neighbor lists of vertices within distance $\hat{D} = D_{\max} - 1$ are needed. Such a schedule will first match all the vertices within distance $\hat{D}$, and then use their neighbor lists to compute the candidate sets for the remaining vertices. This is also a natural choice to match vertices from near to far; in practice, the optimized schedules generated by the system typically follow this way [48]. In this case, the space requirement could be reduced by a factor of the average degree d_{avg} of the graph, which can be as large as 20 in large graphs. In fact, with each hop expanding $O(d_{\text{avg}})$ vertices, the naive methods requires $O({d_{\text{avg}}}^{D_{\max}})$ vertices, each with a neighbor list of length $O(d_{\text{avg}})$. The total space reaches $O({d_{\text{avg}}}^{D_{\max}+1})$. By employing the above method, only $O({d_{\text{avg}}}^{D_{\max}-1})$ neighbor lists are required, reducing space to $O({d_{\text{avg}}}^{D_{\max}})$.

It may at first seem that the above condition is strict. But actually, all patterns of 3, 4, or 5 vertices except Cycle-5 (a cycle of 5 vertices) satisfy $\hat{D} \leq 1$, either because $D_{\max} \leq 1$ already, or $D_{\max} \leq 2$ with the above condition satisfied. This is also true in most common patterns with 6 vertices or more. The patterns that are common to match in real applications are usually not too sparse and have limited $D_{\max}$; otherwise the number of embeddings would grow exponentially, making pattern matching impractical.

To realize the space saving, we extend the CSR format into a reduced one (rCSR) to store the subgraph for a task. First, all vertices within distance $D_{\max}$ are re-numbered starting from 0. This re-numbering preserves the vertex order so that symmetry breaking and orientation are not affected. Similar to CSR, in rCSR, an array `col_idx` is used to store all the neighbor lists sequentially, and an array `row_ptr` points to the starting position of each vertex's neighbor list. Furthermore, the unnecessary neighbor lists of the vertices outside distance $\hat{D}$ are removed from `col_idx`, and the remaining ones are compacted. For example, in Figure 7, `col_idx` excludes the neighbor list of vertex 2. On the other hand, we choose to keep these vertices in `row_ptr`, so that indexing in the two arrays is not affected and can still be done in a simple way. In Figure 7, vertex 2 has its starting position as 6 in `row_ptr`, the same as vertex 3. The small redundancy in `row_ptr` slightly trades off space efficiency for fast indexing.

In practice, multiple tasks are assigned to the same DPU. We take the union of all the required vertices within distance $D_{\max}$ from the corresponding roots, and the neighbor lists of those within distance $\hat{D}$. Then we construct them as one subgraph in the rCSR format. This further saves space compared to storing the local subgraph of each root separately.

We would like to emphasize that no embedding could be missed due to being split at the boundaries of partitioned subgraphs. Intuitively, we partition the graph with overlapping at the boundaries, so embeddings will not be split. More rigorously, embeddings are grouped by root into tasks. After assigning a task rooted at some vertex to a DPU, all the neighbors within the pattern diameter $D_{\max}$ are included in the DPU's subgraph to ensure no embeddings are missing.

Parallel partitioning. Given thousands of DPUs in the system that each requires a different subgraph, using the CPU to partition and prepare the input data for each DPU would cause excessively long time, likely even outweighing the parallel pattern matching phase. To avoid this issue, PimPam also offloads the data partitioning phase onto the DPUs, so that all DPUs extract their own data partitions fully in parallel.

We design a multiple-pass algorithm as illustrated in Algorithm 1, which is executed on each DPU independently to construct its input subgraph. The DPU is given the assigned task list (Line 1). In each pass, the DPU calculates a bitmap of the vertices within distance D. The vertex neighbor lists are streamed in (Line 4), and the neighbors of vertices at distance D are used to update the bitmap of distance $D+1$ (Line 5). Eventually the bitmap of $D_{\max}$ contains all the vertices needed for the tasks, which are then re-numbered (Line 7). Finally, the graph is streamed in (Line 8), and the DPU extracts the relevant neighbor lists to build the rCSR format (Lines 9 and 10).

The main difficulty in the implementation is how the CPU feeds the graph data into the DPU MRAM. As mentioned before, the MRAM is not large enough to fit the entire graph, and once the DPU kernel is launched, the CPU cannot transfer additional data to the DPUs. We thus divide the graph vertices (together with their neighbor lists) into small blocks that each can fit in the MRAM size. To iterate over all vertices at Lines 4 and 8, the CPU transfers one block to the DPUs at a time and launches the kernel for that block, until all blocks are processed. Our partitioning algorithm consumes the graph data in such a stream manner.

Algorithm 1: Parallel data partitioning on each DPU.

```
1  D ← 0; bitmap_0 ←task_list;
2  while D < D_max do
3     bitmap_{D+1} ← bitmap_D;
4     for v ∈ V do
5        if v ∈ bitmap_D then  Add N(v) to bitmap_{D+1} ;
6     D ← D + 1;
7  Re-number vertices in bitmap_{D_max} starting from 0;
8  for v ∈ V do
9     if v ∈ bitmap_D̂ then
10       Use N(v) to update row_ptr and col_idx;
```

4.3 Adaptive Multi-Thread Collaboration

The UPMEM system not only provides a large number of DPUs for parallel processing, but each DPU also supports hardware multi-threading execution that exploits thread-level parallelism to hide long memory access latencies. The task assignment policy in Section 4.1 deals with the load balancing among DPUs. Now we discuss how to realize efficient multi-threading parallelism inside one DPU, through adaptive thread collaboration.

Recall that each DPU is assigned a set of tasks, where each task traverses a search tree rooted at a vertex in the graph G. All the hardware threads on the DPU should collaboratively process these tasks. The simplest approach is to collaborate *at the root level*, i.e., each task is given to a thread as a whole, and each thread is responsible for a disjoint set of tasks, as in Figure 8 (left). However, as shown in Section 2.3, the threads with the largest tasks require far more cycles than others, causing load imbalance and waste of hardware resources within a DPU, and degrading performance.

To alleviate the above problem, PimPam further makes threads to collaborate *at the branch level*, i.e., multiple threads can work on different branches of one search tree, as in Figure 8 (right). The

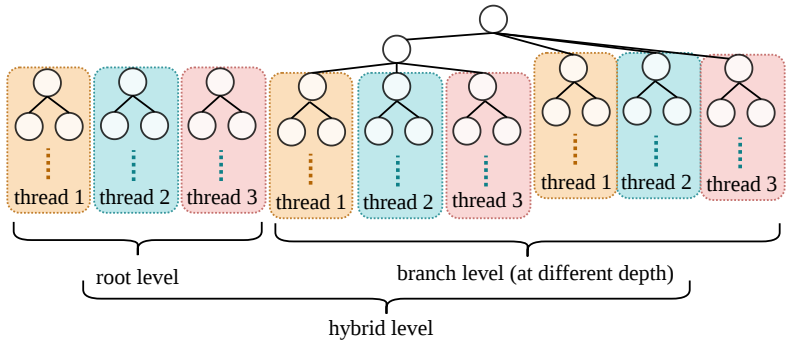


Fig. 8. Two types of thread collaboration: root-level and branch-level. PimPam adaptively uses a hybrid combination of the two types.

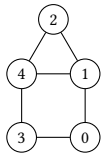
branch-level collaboration can be further extended to deeper tree levels. However, while it is useful to parallelize large tasks, for small tasks the degree of branch-level parallelism within a task may be limited (bounded by vertex degrees). If there are few branches, some hardware threads will be idle.

To achieve the best of both worlds, we propose to *adaptively collaborate among threads with a hybrid of root and branch levels, according to the characteristics of the tasks*. For convenience, we mark each `for` loop in the algorithm with a *depth* starting from 0. In this way, a `for` loop just explores the branches at a search tree level with the same depth. For example, the `for` loops at Lines 1, 2 and 5 in Figure 6 have depths of 0, 1, and 2, respectively. Now the root-level collaboration can be seen as a branch-level collaboration at depth 0, so the two approaches are represented in a unified way. Then the key question is to decide at which depth we shall apply multi-threading execution to parallelize the branches.

Algorithm 2: Adaptive thread collaboration.

```
1 function execute(C, depth)
2   /* Execute a for loop at a certain depth with the candidate set C, starting with a single thread. */
3   for v ∈ C do
4     Compute the next candidate set C′ with this v;
5     if |C′| < S_th or depth > D_th then
6       Partition remaining elements in C to all threads;
7     else
8       execute(C′,depth + 1)
```

Algorithm 2 shows our adaptive mechanism. We explain the algorithm using the House pattern in Figure 6. A threshold S_{th} is set on the cardinality of candidate sets. At each `for` loop level starting from depth 0, we first compute the candidate set of the next loop level, e.g., $N(v_A)$ for a specific v_A in Line 1 of Figure 6. If its cardinality is below the threshold, then the workload is not large, and we choose to use all the threads to process different vertices in the candidate set of this level (Line 6). If it is the first level, i.e., depth 0, then this is just the root-level collaboration. Note that the workloads of root vertices are naturally sorted according to our task assignment method in Section 4.1, so if an early task size is below the threshold, the following tasks would be even smaller. After parallelized, within one thread the tree traversal proceeds following the DFS order instead of BFS, in order to accommodate within the limited MRAM space, and because parallelism is now less concerned within a single thread.



	bit 1	bit 2	bit 3
Bitmap of 1	0	1	0
Bitmap of 2	1	0	0
Bitmap of 3	0	0	0

Fig. 9. An example of dynamic bitmaps for the task rooted at vertex 4. Only its oriented neighbors 1, 2, 3 are involved in the bitmaps.

Otherwise, if the cardinality exceeds the threshold, meaning that this task is relatively large, we proceed to the next loop level to split its workload, i.e., the v_B loop at Line 2 in Figure 6. We recursively apply the criterion at the new level for each v_B, by obtaining the candidate set for v_C and checking its cardinality. If the workload is small enough now, we parallelize into multiple threads at this level; otherwise we continue to deeper levels. Since the adaptive mechanism itself introduces some overheads, a depth threshold D_{th} beyond which no further workload splitting is allowed is set according to each pattern, in order to avoid overly fine-grained partitioning.

The final question is how to set the thresholds. For the cardinality threshold, we use 16 by default. This is because a DPU requires at least 11 threads to fully utilize its pipeline, and we use 16 threads. If it is set smaller, there may not be enough workloads for all threads and some threads will remain idle. If it is too large, the collaboration between threads would become more coarse-grained, leading to load imbalance. For the depth threshold, we trivially set it to 1, as for most simple patterns, usually there are not so many `for` loops. For example, a 4-node pattern has only two `for` loops. Very complex patterns with deeper loop structures may need a higher threshold, which is left to be explored future work.

4.4 Dynamic Bitmap Structures

PimPam mainly stores graph vertex neighbors as sorted lists, and intersections between two neighbor lists are implemented using linear merge operations. Another common way to do set intersections is to store input data using bitmaps and do bitwise AND between them, e.g., through DPUs' 32-bit logic instructions. The bitmap approach could outperform the sorted list approach if the input data are relatively dense. However, constructing the bitmaps of the whole graph is impractical in terms of both memory and computation. It requires $O(|V|^2)$ space, quickly exceeding the limitation of MRAM. Moreover, typically the graph is very sparse. Most of the elements in the bitmaps would be zero, leading to wasted computation.

Nevertheless, PimPam is able to exploit an opportunity to use bitmaps to accelerate clique pattern matching tasks. When processing a task, we *dynamically construct task-specific bitmaps* for the relevant vertices in the graph. By task-specific we mean that the constructed bitmaps can only be used for this task with the specific root. We only apply this technique to clique patterns where all other vertices are neighbors of the root. With symmetry breaking, they should all have smaller IDs than the root. Therefore, *both the number of bitmaps and the bitmap length are effectively bounded by the oriented degree of the root*, i.e., the number of neighbors whose IDs are smaller than the root. With the orientation optimization, the maximum oriented degree in the graph is significantly reduced. For example, there are about four million vertices in `LiveJournal` [30] and the maximum degree is 20333, but the maximum oriented degree is less than one thousand. Therefore, we only need to construct the bitmaps of a very small part of the graph.

Figure 9 shows an example. Suppose the current task is rooted at vertex 4. As said above, only its oriented neighbors 1, 2, 3 are relevant (0 is not a neighbor), so we only construct three bitmaps for vertices 1, 2, 3. The construction method is straightforward, by intersecting the vertex neighbor list

with the root's neighbor list and setting the corresponding bits. The rest of the matching procedure proceeds as usual, but uses the bitmaps for intersections. For example, matching cliques of 4 vertices now becomes to discover triangles (besides the root) using these bitmaps.

5 IMPLEMENTATION

We implement PimPam in about 2000 lines of code written in the C language. For the kernels running on the DPUs, we make use of the standard runtime library provided in the UPMEM SDK [55]. The timing and performance measurement functions follow the same approach as in [20]. More specifically, the task assignment phase is implemented as normal C programs that execute on the host CPU. After that, the CPU-DPU data communication API functions mentioned in Section 2.2 are invoked to transfer the input graph data to the DPUs, who process them to extract their own necessary subgraphs following Algorithm 1. For the pattern matching kernel implementation, we iterate over the tasks and decide their thread collaboration levels as in Algorithm 2. For root-level collaboration, all threads work independently. For branch-level collaboration, threads collaborate on a task and are synchronized using barriers. After all tasks finish, the results are transferred back to the CPU.

We pay special attention to the implementation of intersection operations in PimPam, as they represent the majority of the processing. Specifically, we optimize for the WRAM usage. Recall that there is no direct access from the DPU to the MRAM. The UPMEM compiler provides a default behavior that uses DMA to transfer each data element from MRAM to WRAM each time it is used, without caching or prefetching, thus incurring high overheads. We implement manual prefetching and caching for data accesses during intersection operations. Three buffers in the WRAM are allocated to store the two input lists and the result list, respectively. Each DMA transaction will read or write a chunk of data equal to the buffer size. In our implementation, we set each buffer size to 128 bytes, which is sufficient to amortize the DMA triggering overheads.

6 EVALUATION

6.1 Experimental Setup

We use an UPMEM server in our experiments. It has two Intel Xeon Silver 4216 processors with 64 physical cores in total, running at 2.1 GHz. The CPUs can access 4 DIMMs of traditional DDR4 memory, which provide 256 GB capacity and 38.4 GB/s in total. Besides the normal memory, the PIM memory features 20 DIMMs, with totally 2560 DPUs running at 350 MHz. However, 6 DPUs are found in bad condition,[2] so 2554 DPUs can be actually used by PimPam, with 16 threads per DPU. The details of the other DPU configurations are specified in Section 2.2. The server runs Debian GNU/Linux 10 with kernel version 4.19. The compiler for the host program is gcc 8.3.0, and the DPU programs are developed using the DPU runtime libraries and UPMEM SDK 2023.2.0, which are built based on clang 12.0.0.

In the evaluation, we mainly compare PimPam with the baseline GraphPi [48]. PimPam leverages all the available DPUs in the above UPMEM system. GraphPi executes on the host CPUs of the same server, but only uses the traditional DRAM. We set the number of threads to 64 when running GraphPi, matching the number of physical cores. We find that turning on hyperthreading does not offer benefits, due to the fact that graph pattern matching is memory-bound. We also compare PimPam with state-of-the-art systems that leverage GPUs and distributed execution, namely Pangolin [12] and Khuzdul [8]. Pangolin runs on a server with two Intel Xeon Gold 5218R processors (40 physical cores in total) and four NVIDIA RTX 3090 GPUs. Currently Pangolin only uses a single GPU. Each GPU has 24 GB device memory. The CUDA version is 12.2. Khuzdul runs

[2]This is common and also found in previous work [20].

Table 1. Graph datasets.

Name	Vertices	Edges	Description
`Wiki-Vote` (WV) [31]	7.0k	98.4k	Wikipedia vote
`P2P` (PP) [32]	10.6k	39.0k	P2P network
`ASTRO-PH` (AP) [33]	18.3k	193k	Collaboration
`Youtube` (YT) [34]	1.1M	2.9M	Social network
`Patent` (PT) [35]	3.6M	15.8M	Patent citation
`LiveJournal` (LJ) [30]	4.6M	40.9M	Social network

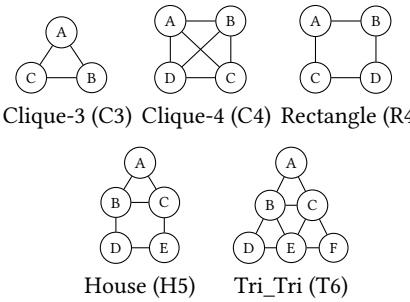


Fig. 10. Patterns.

Table 2. The schedule, the restrictions, and our task time prediction results on the evaluated patterns. The missing vertices in the schedule mean that IEP is used so that these vertices are not counted by `for` loops.

Name	Schedule	Restrictions	Prediction
C3	$[A, B, C]$	$A > B > C$	${d_o}^2$
C4	$[A, B, C, D]$	$A > B > C > D$	${d_o}^3$
R4	$[A, B, C, D]$	$A > B > C, A > D$	${d_o}^2$
H5	$[B, C, D]$	$B > C$	$d_o d d_{avg}(2 + (d + d_{avg})/\|V\|)$
T6	$[B, C, E]$	$B > C > E$	${d_o}^2(d + 3d_{avg} + (d + d_{avg})d_{avg}/\|V\|)$

Table 3. The pre-processing time of GraphPi and PimPam, together with the graph size.

	WV	PP	AP	YT	PT	LJ
PimPam (ms)	90.21	67.02	148.2	2131	25795	77048
GraphPi (ms)	79.25	44.32	163.5	3329	33678	59696
Graph size	816 kB	355 kB	1.6 MB	28 MB	141 MB	346 MB

on eight servers, each with two AMD EPYC 7H12 64-core processors at 1.5 GHz, and 512 GB main memory. These servers are connected using 100 Gb Ethernet with TCP/IP.

Table 1 summarizes the six real-world graphs used to evaluate PimPam, including both small and large graphs that are commonly used for graph pattern matching tasks in previous work [13, 15]. These datasets are first cleaned so that the graphs are transformed into undirected ones, and also duplicated edges, self loops, and empty vertices are removed.

We mainly use five patterns in our experiments, as shown in Figure 10. These patterns include both simple ones like `Clique-3` and more complex ones like `Tri_Tri`. For each pattern, we use the schedule and restrictions generated by GraphPi [48], so both PimPam and the baseline use the same configuration, as shown in Table 2. The table also includes the results of our task time prediction model (Section 4.1). Note that the last two patterns do not contain all vertices in the schedule, which means IEP is used to directly count the few missing vertices. Our prediction model also considers the application of IEP.

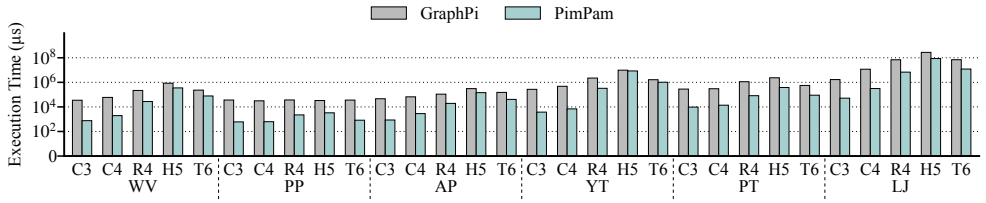


Fig. 11. Overall performance comparison between GraphPi on CPUs and PimPam on UPMEM.

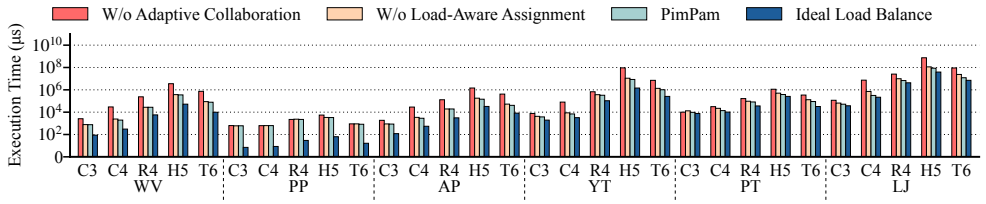


Fig. 12. Performance impacts of adaptive thread collaboration and load-aware task assignment.

6.2 Overall Performance

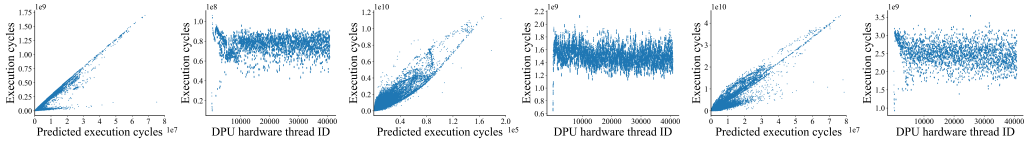
To make our evaluation more clear, we first remind the end-to-end picture of how PimPam works. The schedule and the set of restrictions are determined offline at compile time. This part is exactly the same as GraphPi and excluded in our comparison. After that, PimPam executes the following steps. 1) The prediction model is applied to assign tasks evenly to DPUs. 2) Data are partitioned and transferred to each DPU accordingly. Partitioning is skipped if the whole graph fits in each MRAM. 3) The pattern matching kernel is launched to compute the answer. Note that the final results are just a few integer numbers representing the pattern count in each subgraph. Collecting and summing them up are fast and parallelizable, incurring negligible cost.

We first compare the actual pattern matching time of PimPam and GraphPi, namely the third step above. Figure 11 shows the performance results on all five patterns and six graphs. Time is plotted in log scale. On average, PimPam achieves a 22.5× speedup over GraphPi. It outperforms GraphPi by 48.9×, 38.6×, 10.1×, 4.2×, and 10.6× respectively for the five patterns on different graphs. The maximum speedup is 71.7× for C3 on YT. We analyze these results in more details in Section 6.3.

We then compare the pre-processing cost of PimPam (the first two steps above) with the graph pre-processing time of GraphPi. Although this cost is usually omitted by previous work, we include it to make the evaluation more comprehensive. The pre-processing time depends on the graph size, but remains independent of the pattern. Table 3 demonstrates that, despite the much more complex pre-processing needed by PimPam, it has comparable pre-processing cost to GraphPi. This is because our design only uses simple prediction models and easy-to-obtain information, and offloads most pre-processing to the DPUs to benefit from parallel computation and abundant bandwidth. Furthermore, the pre-processing time roughly scales linearly with the graph size, as one could expect. Such linear scaling in both PimPam and GraphPi is mainly because reading and storing the graph have linear cost.

6.3 Detailed Performance Analysis

We now make detailed performance analysis of the key techniques in PimPam, and separate their performance contributions.



(a) Clique-4 (C4) (b) Rectangle (R4) (c) Tri_Tri (T6)

Fig. 13. Accuracy of the task time prediction model for task assignment. On the LJ graph.

Performance improvements of adaptive thread collaboration and load-aware task assignment. Figure 12 compares four systems: (1) PimPam without adaptive thread collaboration, i.e., only exploiting the root-level parallelism; (2) PimPam without load-aware task assignment, i.e., naively assigning tasks to DPUs in a round-robin manner; (3) the full PimPam design; and (4) an ideal design with perfect load balance, where the performance is calculated by summing up the execution time of all tasks and then dividing by the total number of threads (2554×16).

Without adaptive thread collaboration on each DPU, the performance degrades significantly, by 2.0×, 10.6×, 4.0×, 7.4×, and 6.5× for the five patterns on average. C3 sees a relatively small degradation because it is very simple and the workload of each task is small. In this case, load imbalance between threads is not severe, and root-level parallelism works well. The other more complex patterns heavily rely on the adaptive mechanism to achieve better load balance and exploit more parallelism. The maximum slowdown is 23.4× for C4 on LJ.

Regarding task assignment among DPUs, our prediction model is able to bring a speedup up to 2.3×, also for C4 on LJ. The benefits on small graphs are insignificant, e.g., only 1.09× and 1.03× on WV and PP. This is because there are insufficient tasks at the first place. For example, PP has 10.6k vertices but there are 2554×16 hardware threads. Some threads will remain idle regardless of the assignment. On the other hand, large graphs like LJ exhibit a 1.66× speedup.

The comparison against the ideal case illustrates the remaining room for load balance improvements. Note that both thread collaboration and task assignment affect load balance. On small graphs, as mentioned above, the insufficient tasks make it impossible to achieve balanced loads. On larger graphs like PT and LJ, PimPam has only 1.83× and 1.67× slowdown over the ideal case, which represents a fairly good result in such a challenging case.

Accuracy of the prediction model. We further illustrate the accuracy of our task time prediction model in Figure 13. We select LJ as an example. It is a highly skewed graph with the maximum degree of 20333 but the average is only 17.8. Thus, predicting the load of each task is quite difficult. We use three patterns, C4, R4, T6, including both simple and complex ones. In the left figure of each group, each point represents a task. We can see the good proportional relation between the predicted result and the actual time, following a clear trend of straight lines (recall that a constant factor is omitted in our model). In the right figures, each point represents a thread. The execution cycles of different threads are close to each other, representing balanced loads. This is in sharp contrast to Figure 4 which uses naive round-robin assignment.

Tradeoff made by the prediction model. Recall that we have decided to only use the statistics of the root vertex in the task time prediction model. Intuitively, if more information is used, e.g., the degrees of its immediate neighbors, the prediction should be more accurate, while obtaining these data would cause larger pre-processing cost. Here we quantify the cost and the benefit of refining our prediction model in such a way. We follow the edges of the root vertex to get its neighbors' degrees, and replace the corresponding d_{avg} terms in Equation (1). Table 4 shows that, the cost is significant, especially for large graphs, due to the need of traversing the graph to obtain the

Table 4. The cost and the benefit of more complex prediction. The cost is the extra time in pre-processing. The benefit is the time saved in processing (negative means increased time).

Pattern	YT		PT		LJ	
	Cost (ms)	**Benefit** (ms)	**Cost** (ms)	**Benefit** (ms)	**Cost** (ms)	**Benefit** (ms)
C3		−0.4		0.18		7.0
R4	75.9	61	1253	12.3	2012	1205
T6		−6250		−26		-8.0×10^4

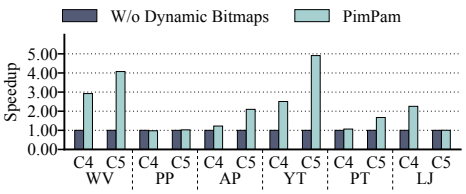


Fig. 14. Performance impact of dynamic bitmap structures.

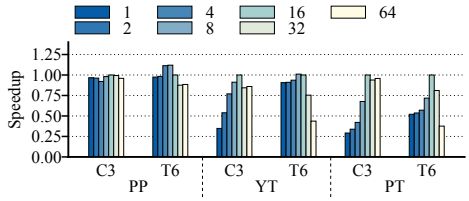


Fig. 15. Performance impact of the threshold of adaptive thread collaboration.

neighbor degrees. On the other hand, the saved benefits are marginal in most cases, and sometimes it even leads to substantial slowdown (e.g., T6 on LJ). There are several reasons. First, we have demonstrated in Figure 12 that our load balancing techniques already approach the ideal case, so the room for further improvements is small. Second, due to the power-law distribution, there is large uncertainty in the neighbor degree values. If a low-degree root happens to be neighboring to a high-degree vertex, the average degree is largely affected. However, as explained in Section 4.1, a single big vertex makes little impact to the workload.

Performance improvements of dynamic bitmap structures. Finally, we evaluate the impact of dynamic bitmaps for accelerating clique patterns. This optimization applies to cliques with more than 3 vertices, so we use Clique-4/5 (C4, C5) in Figure 14. The average speedups are 1.83× and 2.46× for C4 and C5, respectively, and up to 4.91× for C5 on YT. PP exhibits minor benefits because it has few cliques (only three 4-Cliques and no 5-Clique). This makes the bitmap elements mostly 0, and the advantages of such sparse bitmaps over neighbor list intersection vanish.

Threshold of adaptive thread collaboration. Figure 15 shows the performance of different values (from 1 to 64) for the cardinality threshold in Section 4.3, relative to the default value of 16 in PimPam. We use a simple C3 and a complex T6 patterns, on three graphs PP, YT, and PT from small to large sizes as representatives. Generally, choosing 8 or 16 achieves the best performance, where there are enough workloads to fully utilize the DPU pipeline, and the granularity is small enough to balance the load.

6.4 Comparison with GPUs

We also compare PimPam with the state-of-the-art GPU-based design, Pangolin [12]. Since Pangolin only implemented clique patterns, we use Clique-3/4/5 (C3, C4, C5) as the input patterns. The results are shown in Figure 16. Overall the two designs exhibit comparable performance. PimPam performs worse at C3 (0.75×), but better at C4 (1.99×). For C5, PimPam performs better on large graphs such as YT (2.4×), and Pangolin runs out of its 24 GB GPU memory on the large LJ graph.

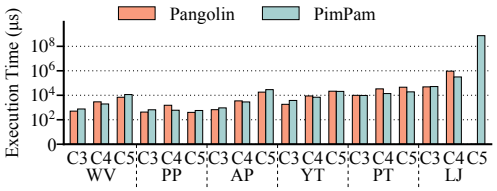


Fig. 16. Overall performance comparison between Pangolin on GPUs and PimPam on UPMEM. The missing data point is due to out-of-memory on the GPU.

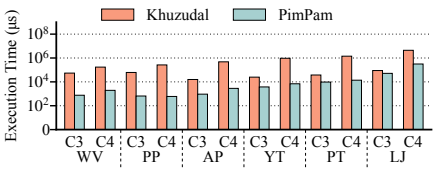


Fig. 17. Overall performance comparison between Khuzdul and PimPam.

For completeness, we also test the CPU version of Pangolin. PimPam outperforms it by 24.8× on average and up to 94.1×.

Note that these results are obtained under a huge gap between the peak computation resources of UPMEM and the GPU. The GPU has 10496 cores with 35.6 TFLOPS, while UPMEM can only deliver 896 GOPS of integer processing [20]. From the energy perspective, UPMEM reports a total power consumption of 383 W [55], while an RTX3090 GPU uses about 350 W. Considering that UPMEM has just come to the market and is mainly a proof-of-concept product, it currently has somewhat low computation throughput and high power consumption. Newer versions are expected to have higher frequency and better power efficiency.

Furthermore, PimPam is able to support larger graphs than the GPU, as shown by the out-of-memory case of C5 on LJ. Nevertheless, PimPam could also run out of memory on extremely large graphs, e.g., it fails on the `Twitter` graph with 1.9B edges. This limitation of PimPam is mostly related to the graph size and not sensitive to the pattern size, as analyzed in Section 4.2. We expect future PIM hardware to scale to larger capacity, and envision future work to combine PIM and distributed processing to handle larger graphs.

6.5 Comparison with Distributed Systems

We next compare PimPam with Khuzdul [8], the state-of-the-art distributed system for graph pattern matching. We use GraphPi as Khuzdul's single-machine engine, and launch 16 threads on each of the 8 nodes to be consistent with the original paper. We use Clique-3/4 (C3, C4) patterns. The results are shown in Figure 17. On the largest graph LJ, PimPam achieves 27.5× and 14.1× speedups on the two patterns, respectively. Note that our measurements are about 2× slower than the original paper [8], which may be attributed to the use of TCP/IP Ethernet instead of Infiniband. Even considering this difference, PimPam is still faster. On smaller graphs, the gains are even larger since the communication overhead in Khuzdul becomes relatively more severe. Particularly, the two most dense graphs WV and PT show higher speedups. Processing dense graphs incurs more communication cost than sparse ones, as retrieving a neighbor list from another node becomes more costly.

7 RELATED WORK

To the best of our knowledge, PimPam is the first graph pattern matching framework implemented on commercial PIM hardware. Previously there have been a large body of proposals optimizing graph pattern matching on CPUs, GPUs, and also using customized hardware accelerators. We briefly discuss them below. We also review other applications beyond graph pattern matching that have leveraged UPMEM, the real PIM platform.

CPU systems. Arabesque [52] built a distributed graph pattern matching system. It extended a subgraph in an edge-by-edge manner and filtered the candidates not meeting the requirements. Fractal [18] was another distributed framework with additional optimizations on various aspects, including load balancing, locality, memory efficiency, and programming expressiveness. G-thinker [59] and G-miner [6] adopted a novel subgraph-centric programming model that focused on computation-intensive processing. BIGJoin [1] and WCOJ [40], on the other hand, used the worst-case optimal join algorithm to process the graph and parallelized it on distributed systems. ASAP [26] instead only returned approximate results of pattern matching and thus showed substantially better performance. RStream [56] and Kaleido [62] were single-machine designs using out-of-core execution, focusing on orchestrating data movements from/to the disks.

AutoMine [39] was a seminal work who first used a compiler to automatically generate C/C++ code optimized for the given input patterns, and executed in a pattern-aware fashion. It achieved significant performance improvements. Later work thus all followed this pattern-aware approach. Peregrine [27], GraphZero [38], and GraphPi [48] were all based on AutoMine, each extracting more properties out of the input patterns (schedules, restrictions, etc.) to further accelerate the matching process. Later, aDFS [54] designed a distributed system that combined DFS and BFS orders to achieve higher parallelism. Sandslash [11] exposed low-level APIs to enable user-specific optimizations. DecoMine [7], mainly targeting pattern counting, used the idea of pattern decomposition to match partial patterns and combine them into large ones. Khuzdul [8] was a distributed execution engine with hybrid scheduling that enhanced GraphPi to support distributed processing of larger graphs.

GPU systems. Pangolin [12] was the first graph pattern matching system that supported both CPUs and GPUs. It focused on pruning the search space and eliding the isomorphism tests to alleviate the drawbacks of previous systems. RPS [22] was another GPU implementation which reused intermediate results to accelerate the algorithm. PBE [21] was based on RPS, equipped with the ability to partition graphs to fit into the GPU memory. Other systems like NEMO [36], GPSM [53], GunRockSM [57], GSI [61], cuTS [58], and EGSM [50] mainly focused on labeled graphs. They applied a filter phase to discard unpromising vertices, but the technique was not effective for unlabeled graphs as in our work.

Specialized hardware accelerators. TrieJax [28] was a hardware accelerator that used relational join operations to express graph matching algorithms. Gramer [60] was an early accelerator that targeted non-pattern-aware algorithms. Its main optimization was to divide the on-chip memory into high-priority and low-priority ones for hot and cold data, respectively. However, these designs failed to exploit the state-of-the-art software paradigm, and thus, were inefficient. On the other hand, FlexMiner [13] and FINGERS [10] introduced the pattern-aware algorithm breakthroughs into hardware, exploiting the abundant parallelism in the processing. SparseCore [43] extended the instruction set architecture (ISA) with stream instructions, making general-purpose CPUs more efficient to process graph pattern matching workloads as well as other sparse tensor programs. Focusing on PIM platforms, SISA [4] defined a new ISA, mostly targeting set operations. NDMiner [51] and DIMMining [15] combined the advantages of both software algorithms and PIM hardware, and proposed various architectural optimizations. Finally, GraphINC [23] leveraged in-network computing on SmartNIC devices to improve graph pattern matching.

Other applications on UPMEM.. PIM-tree [29] was a skew-resistant ordered index implemented on UPMEM. It dynamically decided whether a query should be executed on CPUs or on DPUs to fully utilize the advantages of both. PimDB [3] implemented a database management system on UPMEM and showed advantages over traditional processor-centric ones. Another database system [2] accelerated query processing on UPMEM and used a cost model to decide where data should be

placed. Others [16] implemented deep neural networks on UPMEM with different mapping schemes for different neural networks. TransPimLib [25] enabled support for transcendental functions on UPMEM. UpPipe [9] provided a pipelined design on UPMEM for RNA sequence quantification.

8 CONCLUSION

We propose PimPam, the first graph pattern matching framework that leverages UPMEM, the real-world commercial processing-in-memory hardware. This allows PimPam to overcome the memory bandwidth bottleneck in traditional systems when processing large and irregular graph data. In order to improve load balance, data preparation, and computation efficiency, we further propose several novel techniques and integrate them in PimPam. Overall, PimPam is able to significantly outperform the state-of-the-art CPU baseline, and can scale to complex patterns on large graphs.

UPMEM is a new and promising architecture. Our design of PimPam provides some lessons for those who also want to leverage this hardware. Although UPMEM is a single server, the number of cores is extremely large. Load balance is highly critical, especially for realistic applications where the load is skewed for different subtasks. Furthermore, since there are no shared memory or communication mechanisms between cores, UPMEM is more suitable for tasks where most data can be perfectly partitioned or replicated in a read-only manner. People would need to think twice before using UPMEM if it is not such a case.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable suggestions. This work was supported by the National Natural Science Foundation of China (62072262) and Shanghai Qi Zhi Institute. Mingyu Gao is the corresponding author.

REFERENCES

[1] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas R. Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *Proceedings of the VLDB Endowment* 11, 6 (2018), 691–704.

[2] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Processing-in-Memory for Databases: Query Processing and Data Transfer. In *19th International Workshop on Data Management on New Hardware (DaMoN)*.

[3] Arthur Bernhardt, Andreas Koch, and Ilia Petrov. 2023. pimDB: From Main-Memory DBMS to Processing-In-Memory DBMS-Engines on Intelligent Memories. In *19th International Workshop on Data Management on New Hardware (DaMoN)*.

[4] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez-Luna, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. 2021. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[5] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[6] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: An Efficient Task-Oriented Graph Mining System. In *13th European Conference on Computer Systems (EuroSys)*.

[7] Jingji Chen and Xuehai Qian. 2022. DecoMine: A Compilation-Based Graph Pattern Mining System with Pattern Decomposition. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[8] Jingji Chen and Xuehai Qian. 2023. Khuzdul: Efficient and Scalable Distributed Graph Pattern Mining Engine. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[9] Liang-Chi Chen, Chien-Chung Ho, and Yuan-Hao Chang. 2023. UpPipe: A Novel Pipeline Management on In-Memory Processors for RNA-seq Quantification. In *60th ACM/IEEE Design Automation Conference (DAC)*.

[10] Qihang Chen, Boyu Tian, and Mingyu Gao. 2022. FINGERS: Exploiting Fine-Grained Parallelism in Graph Mining Accelerators. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[11] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining. In *34th ACM International Conference on Supercomputing (ICS)*.

[12] 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.

[13] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind. 2021. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *48th Annual IEEE/ACM International Symposium on Computer Architecture (ISCA)*.

[14] Young-Rae Cho and Aidong Zhang. 2010. Predicting Protein Function by Frequent Functional Association Pattern Mining in Protein Interaction Networks. *IEEE Transactions on Information Technology in Biomedicine* 14, 1 (2010), 30–36.

[15] Guohao Dai, Zhenhua Zhu, Tianyu Fu, Chiyue Wei, Bangyan Wang, Xiangyu Li, Yuan Xie, Huazhong Yang, and Yu Wang. 2022. DIMMining: Pruning-Efficient and Parallel Graph Mining on Near-Memory-Computing. In *49th Annual IEEE/ACM International Symposium on Computer Architecture (ISCA)*.

[16] Prangon Das, Purab Ranjan Sutradhar, Mark Indovina, Sai Manoj Pudukotai Dinakarrao, and Amlan Ganguly. 2022. Implementation and Evaluation of Deep Neural Networks in Commercially Available Processing in Memory Hardware. In *35th IEEE International System-on-Chip Conference (SOCC)*.

[17] Fabrice Devaux. 2019. The True Processing In Memory Accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*.

[18] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *2019 ACM SIGMOD International Conference on Management of Data*.

[19] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On Power-Law Relationships of the Internet Topology. *ACM SIGCOMM Computer Communication Review* 29, 4 (1999), 251–262.

[20] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2021. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. *arXiv preprint arXiv:2105.03814* (May 2021).

[21] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *2020 ACM SIGMOD International Conference on Management of Data*.

[22] Wentian Guo, Yuchen Li, and Lee Tan. 2020. Exploiting Reuse for GPU Subgraph Enumeration. *IEEE Transactions on Knowledge and Data Engineering* 34, 9 (2020), 4231–4244.

[23] Rana Hussein, Alberto Lerner, Andre Ryser, Lucas Bürgi, Albert Blarer, and Philippe Cudre-Mauroux. 2023. GraphINC: Graph Pattern Mining at Network Speed. In *2023 ACM SIGMOD International Conference on Management of Data*.

[24] Hynix. 2023. SK Hynix Develops PIM, Next-Generation AI Accelerator. https://news.skhynix.com/sk-hynix-develops-pim-next-generation-ai-accelerator/.

[25] Maurus Item, Geraldo F. Oliveira, Juan Gómez-Luna, Mohammad Sadrosadati, Yuxin Guo, and Onur Mutlu. 2023. TransPimLib: Efficient Transcendental Functions for Processing-in-Memory Systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[26] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[27] K. Jamshidi, R. Mahadasa, K. Vora, and Acm. 2020. PEREGRINE: A Pattern-Aware Graph Mining System. In *15th European Conference on Computer Systems (EuroSys)*.

[28] Oren Kalinsky, Benny Kimelfeld, and Yoav Etsion. 2019. The TrieJax Architecture: Accelerating Graph Operations Through Relational Joins. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[29] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2022. PIM-Tree: A Skew-Resistant Index for Processing-in-Memory. *Proceedings of the VLDB Endowment* 16, 4 (2022), 946–958.

[30] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/soc-LiveJournal1.html.

[31] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/wiki-Vote.html.

[32] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/p2p-Gnutella04.html.

[33] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/ca-AstroPh.html.
[34] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/com-Youtube.html.
[35] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/cit-Patents.html.
[36] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiao-Li Li. 2015. Network Motif Discovery: A GPU Approach. In *31st International Conference on Data Engineering (ICDE)*.
[37] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *2010 ACM SIGMOD International Conference on Management of Data*.
[38] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2019. GraphZero: Breaking Symmetry for Efficient Graph Mining. *arXiv preprint arXiv:1911.12877* (Nov 2019).
[39] Daniel Mawhirter and Bo Wu. 2019. Automine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *27th ACM Symposium on Operating Systems Principles (SOSP)*.
[40] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704.
[41] Mark E. J. Newman. 2003. The Structure and Function of Complex Networks. *SIAM Rev.* 45, 2 (2003), 167–256.
[42] Dhinakaran Pandiyan and Carole-Jean Wu. 2014. Quantifying the Energy Cost of Data Movement for Emerging Smart Phone Workloads on Mobile Platforms. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*.
[43] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. 2022. SparseCore: Stream ISA and Processor Specialization for Sparse Computation. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[44] Matthias Rupp. 2011. Graph Kernels for Chemoinformatics - A Critical Discussion. *Journal of Cheminformatics* 3, 1 (2011), 1–1.
[45] Samsung. 2023. HBM-PIM: Cutting-Edge Memory Technology to Accelerate Next-Generation AI. https://semiconductor.samsung.com/news-events/tech-blog/hbm-pim-cutting-edge-memory-technology-to-accelerate-next-generation-ai/.
[46] Vivek Seshadri, Kevin Hsieh, Amirali Boroum, Donghyuk Lee, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2015. Fast Bulk Bitwise AND and OR in DRAM. *IEEE Computer Architecture Letters* 14, 2 (2015), 127–131.
[47] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization. In *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[48] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination. In *32nd International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
[49] Jiya Su. 2022. PIMMiner: A High-Performance PIM Architecture-Aware Graph Mining Framework. *arXiv preprint arXiv:2306.10257* (Jun 2022).
[50] Xibo Sun and Qiong Luo. 2023. Efficient GPU-Accelerated Subgraph Matching. *2023 ACM SIGMOD International Conference on Management of Data*.
[51] Nishil Talati, Haojie Ye, Yichen Yang, Leul Belayneh, Kuan-Yu Chen, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2022. NDMiner: Accelerating Graph Pattern Mining Using Near Data Processing. In *49th Annual IEEE/ACM International Symposium on Computer Architecture (ISCA)*.
[52] 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 *25th ACM Symposium on Operating Systems Principles (SOSP)*.
[53] Ha-Nguyen Tran, Kim Jung-Jae, and He Bingsheng. 2015. Fast Subgraph Matching on Large Graphs using Graphics Processors. In *International Conference on Database Systems for Advanced Applications (DASFAA)*.
[54] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Ioan Haprian, Călin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. 2021. aDFS: An Almost Depth-First-Search Distributed Graph-Querying System. In *USENIX Annual Technical Conference (USENIX ATC)*.
[55] UPMEM. 2023. UPMEM Website. https://www.upmem.com/.
[56] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[57] Leyuan Wang and John Owens. 2020. Fast Gunrock Subgraph Matching (GSM) on GPUs. *arXiv preprint arXiv:2003.01527* (May 2020).
[58] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: Scaling Subgraph Isomorphism on Distributed Multi-GPU Systems Using Trie Based Data Structure. In *33rd International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
[59] Da Yan, Hongzhi Chen, James Cheng, M. Tamer Özsu, Qizhen Zhang, and John C.S. Lui. 2017. G-thinker: Big Graph Mining Made Easier and Faster. *arXiv preprint arXiv:1709.03110* (Sep 2017).
[60] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. 2022. A Locality-Aware Energy-Efficient Accelerator for Graph Mining Applications. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[61] Li Zeng, Lei Zou, M. Tamer Özsu, Lin Hu, and Fan Zhang. 2020. GSI: GPU-Friendly Subgraph Isomorphism. In *36th International Conference on Data Engineering (ICDE)*.
[62] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng Guo. 2020. Kaleido: An Efficient Out-of-Core Graph Mining System on A Single Machine. In *36th International Conference on Data Engineering (ICDE)*.

Received 16 October 2023; revised 21 January 2024; accepted 22 February 2024