NDMiner: Accelerating Graph Pattern Mining Using Near Data Processing

Nishil Talati
University of Michigan
Ann Arbor, Michigan, USA
talatin@umich.edu

Haojie Ye
University of Michigan
Ann Arbor, Michigan, USA
yehaojie@umich.edu

Yichen Yang
University of Michigan
Ann Arbor, Michigan, USA
yangych@umich.edu

Leul Belayneh
University of Michigan
Ann Arbor, Michigan, USA
leulb@umich.edu

Kuan-Yu Chen
University of Michigan
Ann Arbor, Michigan, USA
knyuchen@umich.edu

David Blaauw
University of Michigan
Ann Arbor, Michigan, USA
blaauw@umich.edu

Trevor Mudge
University of Michigan
Ann Arbor, Michigan, USA
tnm@umich.edu

Ronald Dreslinski
University of Michigan
Ann Arbor, Michigan, USA
rdreslin@umich.edu

ABSTRACT

Graph Pattern Mining (GPM) algorithms mine structural patterns in graphs. The performance of GPM workloads is bottlenecked by control flow and memory stalls. This is because of data-dependent branches used in set intersection and difference operations that dominate the execution time.

This paper first conducts a systematic GPM workload analysis and uncovers four new observations to inform the optimization effort. First, GPM workloads mostly fetch inputs of costly set operations from different memory banks. Second, to avoid redundant computation, modern GPM workloads employ symmetry breaking that discards several data reads, resulting in cache pollution and wasted DRAM bandwidth. Third, sparse pattern mining algorithms perform redundant memory reads and computations. Fourth, GPM workloads do not fully utilize the in-DRAM data parallelism.

Based on these observations, this paper presents NDMiner, a Near Data Processing (NDP) architecture that improves the performance of GPM workloads. To reduce in-memory data transfer of fetching data from different memory banks, NDMiner integrates compute units to offload set operations in the buffer chip of DRAM. To alleviate the wasted memory bandwidth caused by symmetry breaking, NDMiner integrates a *load elision unit* in hardware that detects the satisfiability of symmetry breaking constraints and terminates unnecessary loads. To optimize the performance of sparse pattern mining, NDMiner employs *compiler optimizations* and maps reduced reads and composite computation to NDP hardware that improves algorithmic efficiency of sparse GPM. Finally, NDMiner proposes a new *graph remapping* scheme in memory and a *hardware-based set operation reordering* technique to best optimize bank, rank, and channel-level parallelism in DRAM. To orchestrate NDP computation, this paper presents design modifications at the host ISA, compiler, and memory controller. We compare the performance of NDMiner with state-of-the-art software and hardware baselines using a mix of dense and sparse GPM algorithms. Our evaluation shows that NDMiner significantly outperforms software and hardware baselines by 6.4× and 2.5×, on average, while incurring a negligible area overhead on CPU and DRAM.

ISCA '22, June 18–22, 2022, New York City, NY
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527437

CCS CONCEPTS

• **Hardware → Emerging architectures**.

KEYWORDS

Graph pattern mining, near data processing, hardware-software co-design

ACM Reference Format:
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 *Proceedings of The 49th Annual International Symposium on Computer Architecture (ISCA '22)*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3470496.3527437

1 INTRODUCTION

Graph Pattern Mining (GPM) algorithms are used in numerous applications, including bioinformatics [14], cyber-security [18, 42], social network analysis [55, 57], and spam detection [28]. Despite their prevalence, GPM workloads are severely stalled on modern hardware platforms [8, 12, 60]. A majority of this performance slowdown is attributed to the irregular memory and complex data-dependent branch instructions used in set intersection and difference operations that dominate GPM workload execution times.

Prior hardware works have addressed the inefficiencies of GPM workloads either by proposing domain-specific accelerators [12, 60] or Near Data Processing (NDP) [8]. These works, however, can be significantly improved. While accelerators like FlexMiner [12]

Challenges	Proposed Optimizations
Wasted DRAM bandwidth due to symmerty breaking	Load Elision Unit Breaks symmerty in hardware
Unnecessary reads when mining sparse patterns	Compiler optimizations to improve algorithmic efficiency
Unexploited in-DRAM data parallelism	Graph remapping and set operation reordering

*This also includes multiple near-data PEs.

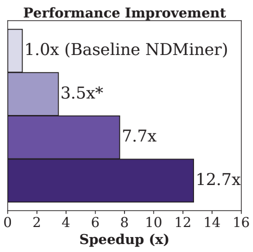


Figure 1: NDMiner optimizations and corresponding performance improvements inspired by the challenges of accelerating GPM workloads. Optimizations are cumulative as the bars move down.

employ application-specific control and data paths, the general-purpose nature of their memory subsystems suffers from unnecessary data movement. On the other hand, SISA [8] optimizes GPM software by using a set-centric ISA and improved intersection algorithm. SISA, however, maps GPM computation to generic NDP architectures, *e.g.,* Ambit [46], without specialization. Therefore, GPM performance can be further improved by employing domain-specific techniques to design NDP architectures. To best design a domain-specific NDP solution, it is important to first understand the unique characteristics of GPM workloads.

To this end, we conduct a systematic characterization of GPM workloads to understand their sources of inefficiencies. This leads to four unique takeaways. First, because of the irregular graph data layout in memory, GPM workloads read data from different DRAM banks to compute set operations. Second, the symmetry breaking optimization used in modern GPM workloads discards most vertices fetched from memory in each iteration, resulting in cache pollution and wasted DRAM bandwidth. Third, sparse pattern mining algorithms perform several redundant reads and computations, leading to low algorithmic efficiency. Fourth, the size-limited memory controller queue does not allow GPM workloads to fully utilize internal DRAM data parallelism.

In this paper, we present NDMiner—an NDP architecture to accelerate GPM workloads. In addition to tapping the abundant in-memory data bandwidth, the goal of this design is to exploit the presented domain-specific insights for optimization. NDMiner proposes architectural innovations to a general-purpose system with low-cost compute units within a DIMM-based DRAM and CPU to effectively execute costly set operations in GPM. To support NDP operations, we also present a hardware-software interface that (a) extends the host ISA to include NDP instructions, (b) transforms GPM source code to use these NDP instructions, and (c) extends the memory controller design to orchestrate in-DRAM compute.

We further optimize NDMiner using domain-specialization as shown in Fig. 1. First, NDMiner integrates a new **load elision unit** in hardware to alleviate the DRAM bandwidth wastage due to symmetry breaking. This unit terminates unnecessary loads by breaking symmetry in hardware. Second, NDMiner employs **compiler optimizations** to improve the algorithmic efficiency of sparse pattern mining algorithms. This avoids redundant data loads and compute operations by flattening the loop nest into composite set operations and hoisting loop invariant computations out of the loops. We also present how to map these computations to NDP hardware. Third, NDMiner reorders set operations at runtime to exploit internal data parallelism in DRAM. To make this reordering possible at low-cost, we first propose a novel **graph data remapping** scheme in DRAM. Based on this remapping, we design a new **vertex ID–based reordering** hardware that examines a large window (*e.g.,* 1024 entries) of set operations and reorders them to insert requests into a size-limited memory controller. The goal of this reordering is to exploit bank, rank, and channel-level parallelism in DRAM.

We rigorously evaluate NDMiner using seven GPM algorithms that mine cliques, user-defined subgraphs, and motifs on five real-world graphs. The input patterns contain a mix of both sparse and dense patterns. We first evaluate the effectiveness of various design optimizations by comparing NDMiner configurations with a baseline NDP architecture that integrates one set operation unit per channel. As shown in Fig. 1, proposed optimizations significantly improve the performance of this baseline design by 12.7× and reduces energy consumption by 5.1×, on average (more results in §8). We also compare NDMiner with the state-of-the-art GPM software (*i.e.,* GraphPi [48] and Pangolin [10]) and hardware (*i.e.,* FlexMiner [12]). We show that, on average, NDMiner significantly outperforms software and hardware baselines by 6.4× and 2.5×. Post-synthesis estimation of proposed circuits shows that NDMiner achieves these improvements at a negligible area cost.

In summary, we make the following novel contributions.

- A detailed analysis of GPM workloads uncovering new opportunities for performance optimization.
- *Load elision unit:* a novel design that breaks symmetry in hardware to avoid unnecessary loads.
- *Compiler optimizations:* a collection of software techniques and corresponding hardware mapping to reduce redundant loads and computations in sparse GPM.
- *Graph remapping and set operation reordering:* novel techniques to reorder computation in GPM to exploit internal data parallelism in DRAM.
- *NDMiner:* an end-to-end system that combines aforementioned optimizations that significantly improves the performance of the state-of-the-art GPM hardware accelerator by 2.5×, on average, at negligible silicon cost.

2 BACKGROUND

This section briefly discusses the background on GPM and NDP.

2.1 Graph Pattern Mining (GPM)

GPM problem finds all *unique* subgraphs (also known as *embeddings*) in an input graph that are *isomorphic* to a given input pattern. A pattern is isomorphic to a subgraph if there exists a one-to-one mapping of all the vertices and edges between the pattern and a subgraph. Permuting vertices and edges of a given subgraph generates equivalent subgraphs, also called *automorphic* embeddings.

GPM algorithm. It uses a search tree to enumerate embeddings in an input graph G matching a user-defined pattern P. From all single-vertex subgraphs, the tree visits one node/edge at a time to expand the embedding in each level. The isomorphism test is performed after all the embeddings reach a desired tree depth (*i.e.,* size of the embedding), where the number of vertices in expanded

Algorithm 1 Pseudocode for Triangle Counting (TC)

```
1:  procedure GPM_TC(G, P)                    ▷ G: graph, P: pattern (triangle in this case)
2:      num_trialges = 0;
3:      for u ∈ V do                          ▷ V: Vertex set of G, {u}: single-vertex embedding
4:          N_u = G.out_neighbors(u);         ▷ Neighborhood expansion
5:          for v ∈ N_u do                    ▷ {u, v}: two-vertex embedding
6:              if v ≥ u then                 ▷ Neighborhood filtration for symmetry breaking
7:                  break;
8:              N_v = G.out_neighbors(v);     ▷ Neighborhood expansion
9:              N_uv = Intersection(N_u, N_v);   ▷ Set intersection
10:             for w ∈ N_uv do               ▷ {u, v, w}: three-vertex embedding
11:                 if w ≥ v then             ▷ Intersection filtration for symmetry breaking
12:                     break;
13:                 num_triangles++;
14:     return num_triangles;
15:
16: procedure Intersection(SetA, SetB)        ▷ Set intersection procedure
17:     intersection_result = [];
18:     while i < SetA.size() and j < SetB.size() do
19:         if SetA[i] < SetB[j] then         ▷ Data-dependent control flow
20:             i++;
21:         else if SetA[i] > SetB[j] then    ▷ Data-dependent control flow
22:             j++;
23:         else                              ▷ SetA[i] = SetB[j]
24:             intersection_result.insert(SetA[i]);
25:             i++; j++;
26:     return intersection_result;
```

subgraphs matches the number of vertices in P. Following the terminology in Peregrine [22], GPM algorithms can be broadly classified in two categories: (a) pattern-oblivious, and (b) pattern-aware. Peregrine concludes that pattern-aware GPM algorithms outperform their pattern-oblivious counterparts by eliminating redundant computations. Therefore, we use pattern-aware algorithms.

Algorithm 1 shows the pseudo-code of triangle counting. Starting from single-vertex embeddings shown in line 3, the algorithm expands them to two-vertex embeddings (line 5) by finding their outgoing neighbors (line 4). The graph is typically stored in a Compressed Sparse Row (CSR) format in memory. A node's neighbor list is found by first indexing into the offset list and then into the edge list. These embeddings are further expanded by finding common neighbors amongst its vertices. The *intersection* (line 9) of vertex neighborhood sets is employed to find common neighbors. With this expansion, embeddings isomorphic to a desired pattern (triangle) are found. Lines 16–26 present the pseudocode for performing the intersection operation. Similar to state-of-the-art graph frameworks [6], we assume that neighbors of any node stored in the edge list are sorted by their vertex IDs. This allows for completion of intersection in linear time. Notably, the pattern-aware GPM algorithms only find embeddings isomorphic to P. In other words, the isomorphism test is encoded into the algorithms, precluding the necessity for explicit isomorphism tests after search tree expansion.

GPM algorithm optimizations. Pattern-specific GPM algorithms enable several performance optimizations. We briefly discuss (a) optimized schedule, and (b) symmetry breaking restrictions optimizations used in this paper, and refer the reader to prior works [9, 10, 22, 23, 30, 31, 48] for other optimizations. The *schedule* of a GPM algorithm determines the order at which each vertex of a pattern is searched. When searching for patterns, *restrictions* are applied to vertex IDs to avoid redundant computation. This is also known as symmetry breaking/search tree pruning as it avoids expanding unnecessary tree branches that cannot lead to P.

GraphPi [48] shows that there is a large design space to find the optimal schedule and restrictions that can affect performance by up to three order of magnitude. This is because the schedule and restrictions define the size and pruning level of the search tree that lead to significant performance differences. Lines 6 and 11 show the instances of the *filtration operation* applied to triangle counting for search tree pruning. Because a triangle is a symmetric pattern, the order at which the vertices are searched makes no difference, leading to only one schedule. However, large asymmetric patterns can benefit significantly from schedule optimizations. This paper adopts optimal schedule and restrictions from GraphPi.

2.2 Near Data Processing

Near Data Processing (NDP)[1] improves the performance of memory bound workloads by reducing the amount of costly off-chip data transfers and exposing high internal memory bandwidth to compute units. The early efforts in this direction date back to the '90s [17, 19, 36, 38, 39] that integrate logic units in DRAM. More recent NDP architectures include computing in DRAM [2, 7, 15, 25, 27, 61, 62] and emerging memory technologies [13, 29, 47, 50, 52].

NDP proposals can be broadly classified into three categories based on the proximity of compute units from data. This classification is crucial to determining the design choices while designing novel NDP architectures. Approaches similar to MAGIC [52] process data within a memory mat/subarray without reading them out. Such proposals enjoy high internal data bandwidth if the operands are aligned in two memory rows/columns. Other approaches process data at local/global row buffer (*e.g.,* a recent industrial proposal from Samsung [27]). While these proposals do not require the operands to be aligned within memory rows, they can be best utilized when the operands are present in the same bank. Although it is possible to move data internally within the memory from one bank to another using RowClone [45], frequent data movement can limit the benefit of near data processing. Lastly, other proposals place computation within the buffer chip or logic layers of the memory (*e.g.,* RecNMP [25] for DIMM, Teserract [2] for HMC). These approaches can avail data from different banks, however, their bandwidth is limited by the data acquisition bandwidth at the buffer chip or the TSVs in 3D DRAM. In sum, **where** to place compute units within memory depends on the workload characteristics.

3 FINDING OPTIMIZATION OPPORTUNITIES FOR GPM

This section presents unique GPM workload characteristics to motivate NDMiner design. We divide these findings into well-known GPM characteristics and new findings based on our profiling results.

3.1 Well-Known GPM Characteristics

Prior optimization works [8, 12, 43, 60] find several unique characteristics of GPM workloads. We summarize them below.

[1]Without losing generality, we refer to computing in/near memory approaches to Near Data Processing (NDP).

Takeaway 1. Set intersection and difference operations dominate the execution times of GPM workloads.
Takeaway 2. GPM workloads use simple arithmetic compute instructions (*e.g.,* shape count increments) that do not contribute to stall cycles.
Takeaway 3. The irregular memory accesses and their dependent control flow operations are the major sources of bottlenecks in GPM workloads.
Takeaway 4. GPM algorithms mostly use read-only data structures offering the opportunity for massive parallelism without needing synchronization.

3.2 Novel GPM Characteristics

In addition to validating well-known characteristics of GPM workloads, this work finds the following novel characteristics that we employ for NDMiner hardware design.

Distribution of input sets in memory. To better understand the workload behavior of GPM, we examine the memory locations of set operation inputs used in computing difference and intersection. Fig. 2 shows this distribution classified into four categories: (a) same bank, (b) different banks in the same bank group, (c) different bank groups on the same rank, and (d) different ranks. The figure shows that a majority of the time, the set operands are present in different banks. Because these workloads perform a large number of set operations that choose inputs interleaved between different banks/ranks based on vertex IDs, there is less than 5% difference in their operand distributions. This result offers insight into where to best place NDP compute logic to optimize GPM workloads.

Takeaway 5. GPM workloads fetch data from different DRAM banks to compute set operations.

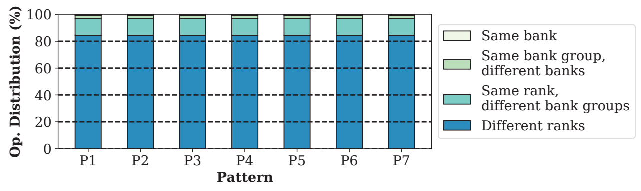
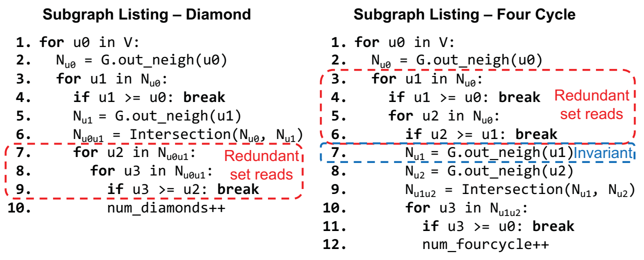


Figure 2: Distribution of locations of set operation inputs showing that GPM workloads mostly fetch operands from different banks.

	Dense Patterns			Sparse Patterns		Mixed Patterns	
	P1	P2	P3	P4	P5	P6	P7
wiki-vote	2.4%	1.2%	0.7%	37.8%	5.9%	26.1%	47.8%
pokec	1.3%	1.0%	0.9%	14.6%	1.5%	25.5%	36.5%
patents	4.0%	3.0%	2.6%	13.8%	6.4%	26.4%	42.7%
livejournal	2.5%	5.4%	6.4%	45.4%	7.1%	26.1%	39.9%

Table 1: Percentage of vertices utilized in the next search levels out of all fetched vertices because of symmetry breaking.


Subgraph Listing – Diamond

```
1.  for u0 in V:
2.    N_u0 = G.out_neigh(u0)
3.    for u1 in N_u0:
4.      if u1 >= u0: break
5.      N_u1 = G.out_neigh(u1)
6.      N_u0u1 = Intersection(N_u0, N_u1)
7.      for u2 in N_u0u1:
8.        for u3 in N_u0u1:
9.          if u3 >= u2: break
10.         num_diamonds++
```

Subgraph Listing – Four Cycle

```
1.  for u0 in V:
2.    N_u0 = G.out_neigh(u0)
3.    for u1 in N_u0:
4.      if u1 >= u0: break
5.      for u2 in N_u0:
6.        if u2 >= u1: break
7.        N_u1 = G.out_neigh(u1)
8.        N_u2 = G.out_neigh(u2)
9.        N_u1u2 = Intersection(N_u1, N_u2)
10.       for u3 in N_u1u2:
11.         if u3 >= u0: break
12.         num_fourcycle++
```

Figure 3: Examples of redundant load and computation in sparse pattern mining algorithms (*i.e.,* subgraph mining for diamonds and four cycles).

Adverse effect of symmetry breaking. As presented in §2.1, advanced GPM algorithms use symmetry breaking to avoid redundant computation. For triangle counting, this is reflected in lines 6 and 11 of Algorithm 1. In effect, only a fraction of the computed neighborhood or set operation results (lines 4 and 9) are used in the next phase of computation, which we call the *filter operations*. To understand the effect of filter operations, we calculate the fraction of vertices used in the current GPM iteration out of all the vertices fetched in the previous iteration to compute neighborhoods/set operations. Table 1 shows that 66.5% of the vertices fetched in a previous iteration are discarded in the current iteration. Sparse patterns are defined as graph patterns where most nodes are not connected to all other nodes. Conversely, fully connected patterns (*e.g.,* cliques) are called dense patterns. Intuitively, dense input patterns utilize a smaller fraction of vertices compared to sparse patterns. This is because dense pattern mining algorithms employ more constraints than their sparse counterparts because of their dense connectivity structures. While this improves the efficiency of GPM algorithms by avoiding redundant computation, it pollutes the CPU caches and squanders useful DRAM bandwidth.

Takeaway 6. Symmetry breaking discards most vertices fetched from memory in each iteration, leading to cache pollution and wasted DRAM bandwidth.

Redundant reads and computations for mining sparse patterns. Fig. 3 shows the pseudocode for mining two sparse patterns, *i.e.,* diamond and four cycle. The figure shows that, for diamond mining in lines 7–9, vertices $u2$ and $u3$ are found by iterating over the same candidate sets, *i.e.,* N_{u0u1}. The same trend exists for vertices $u1$ and $u2$ in four cycle mining algorithm (lines 3–6). Furthermore, line 7 of four cycle mining algorithm shows that neighborhood computation N_{u1} is invariant to $u2$. These properties of sparse GPM lead to redundant reads and computation. While we use two example shapes to demonstrate this concept, this redundancy is common across a wide range of sparse GPM algorithms.

Takeaway 7. Sparse pattern mining algorithms involve redundant reads and computations.

Set Operation reordering opportunity. While most prior

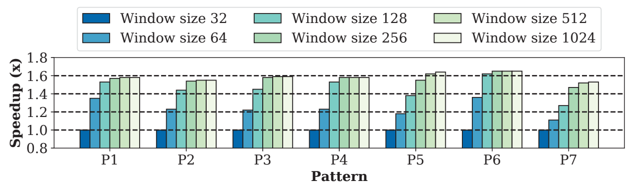


Figure 4: Speedup of GPM workloads for different memory controller reorder window sizes. Results are normalized to 32 window (memory controller read queue) size.

GPM works typically process vertices in an input graph in the order of their vertex IDs, we design an experiment to find if there is an opportunity to gain performance by reordering the GPM memory accesses. First, we reorder an input graph in software by using three graph reordering techniques, *i.e.,* DegreeSort, HubCluster, and HubSort based on a prior work [4]. This, however, does not affect the performance of GPM workloads. Second, we reordered the set operations computed in hardware by artificially increasing the memory controller read queue size. Fig. 4 shows the effect of using larger memory controller reordering window sizes on GPM performance, normalized to a realistic size of 32. The figure shows that a larger reordering window improves the workload performance by up to 1.6×. This is because a smaller reordering window is congested by the requests to the same bank, reducing reordering and data-parallelism opportunity. Larger windows, on the other hand, find requests to better exploit data-parallelism by sending concurrent requests to multiple banks, ranks, and channels. Notably, this result does not contradict Takeaway 5 because Fig. 2 shows operand distribution for a *single* set operation, whereas Fig. 4 is an effect of operand distribution of *multiple* set operations.

Takeaway 8. GPM workloads do not fully exploit abundant data-parallelism in DRAM because of size-limited memory controller queues.

3.3 Why NDP for GPM?

As discussed in §2.2, NDP alleviates the performance and energy overheads of costly off-chip data transfers between the CPU and DRAM. This can be used to alleviate the wasteful data transfer in GPM algorithms because of symmetry breaking (Takeaway 6). NDP has the potential to reduce cache thrashing and energy wasted on off-chip data transfer. Additionally, NDP exposes high internal memory bandwidth that can be exploited by GPM algorithms as they offer ample parallelism (Takeaway 4).

In-DRAM compute parallelism can be best utilized by simple compute units that can be integrated within the memory in a cost-effective manner. GPM algorithms mostly use adder and comparator logic to perform most of their computations (Takeaway 2). The simplicity of these operations allows their cost-efficient integration within the memory. Resolving load-dependent control flow operations at NDP precludes the need for using expensive branch resolution mechanisms on the CPU. Moreover, irregular accesses to graph data structures resulting in high memory latency and/or bandwidth [33, 54] can be better serviced near memory at a low latency and high available bandwidth, addressing the two main bottlenecks in GPM workloads (Takeaway 3). *In summary, NDP is an attractive candidate for accelerating GPM workloads.*

3.4 How To Best Design NDP For GPM?

The next task is to find where to place the compute unit within memory? As discussed in §2.2, the best place depends on the workload characteristics. As set intersection/difference operations dominate the execution time of GPM workloads (Takeaway 1), we offload them to NDP units. Furthermore, Takeaway 5 shows that GPM workloads mostly fetch data from different banks. Therefore, placing compute units inside the bank would incur significant in-DRAM data transfer. Hence, we make a design decision to place the compute units at the buffer chip of DIMMs in NDMiner. While we use DIMM in this paper, similar design principles can also be applied to the logic layer of HMC/HBM.

4 HARDWARE-SOFTWARE INTERFACE

This section discusses the hardware-software interface of NDMiner to support NDP operations for GPM acceleration.

4.1 Supported NDP Operations

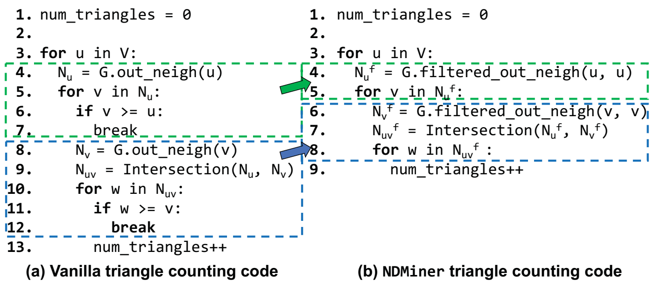
Based on Takeaway 1, NDMiner offloads set intersection and difference operations to the NDP units. Additionally, the primary goal of NDP design is to alleviate the cost of data movement in GPM workloads. As presented in Takeaway 6, symmetry breaking results in wasteful data movement. By using NDP, it is possible to identify and terminate loads filtered by breaking symmetry in hardware. This helps improving the overall efficiency of the program by eliding useless loads that prevents cache pollution. Therefore, NDMiner also offloads load elision operations to memory. In total, NDMiner supports five NDP operations: (a) complete set intersection, (b) complete set difference, (c) filtered set intersection, (d) filtered set difference, (e) load filtered set.

4.2 ISA Extensions

```
filtered_intersect   addr0, len0, addr1, len1, u_th // u_th=-1 if no filter
filtered_difference  addr0, len0, addr1, len1, u_th // u_th=-1 if no filter
      filtered_load  addr0, len0, u_th             // u_th=-1 if no filter
```

Figure 5: Host ISA instructions to support NDP.

To enable software to communicate NDP operations to memory through the host CPU, NDMiner introduces three instructions in the ISA as shown in Fig. 5. To support symmetry breaking in hardware (more details in §6.1), these instructions support filtering of input sets. A threshold vertex ID is specified (*i.e.,* u_th) that is determined at runtime by the CPU and communicated to the NDP units. If load elision is not applied, the values of u_th is specified as −1. The memory address ranges of input sets are indicated by the base address and length of sets. Similar to recent academic NDP proposals [2, 25, 61] and an industrial product [27], we assume that the data allocated for NDP uses physically contiguous memory blocks. Contiguous mapping ensures that NDP instructions only have to translate one address, and the rest of the addresses can be obtained using the address range, even if the addresses rarely



```
1. num_triangles = 0
2.
3. for u in V:
4.   N_u = G.out_neigh(u)
5.   for v in N_u:
6.     if v >= u:
7.       break
8.     N_v = G.out_neigh(v)
9.     N_uv = Intersection(N_u, N_v)
10.    for w in N_uv:
11.      if w >= v:
12.        break
13.        num_triangles++
```

(a) Vanilla triangle counting code

```
1. num_triangles = 0
2.
3. for u in V:
4.   N_u^f = G.filtered_out_neigh(u, u)
5.   for v in N_u^f:
6.     N_v^f = G.filtered_out_neigh(v, v)
7.     N_uv^f = Intersection(N_u^f, N_v^f)
8.     for w in N_uv^f :
9.       num_triangles++
```

(b) NDMiner triangle counting code

Figure 6: Code transformations to utilize NDP instructions.

cross the OS page boundaries. This, however, is not a fundamental limitation of NDMiner as it is also compatible with the current OS page mapping scheme, which would rarely require more than one address translations per NDP instruction when set inputs span multiple pages. Furthermore, while ISA extensions simplify the design parameters and programming model, computation offloading to NDP can be alternatively achieved by using load/store instructions to memory-mapped registers.

4.3 Programming Model

To utilize aforementioned ISA instructions, an NDMiner compiler transforms the source code of GPM workloads. First, the compiler analyzes the source code to extract the instructions amenable to NDP acceleration. These instructions include set operation computations, neighborhood loads, and symmetry breaking constraints. These instances are then replaced with NDP instructions. Fig. 6 shows an example of source code transformations, where lines in the green and blue boxes in the original source code are replaced with filtered load operations. In this workload, the intersection operation is not modified as it receives filtered neighborhoods as input (line 7 in Fig. 6(b)). For workloads where neighborhoods are not filtered beforehand, **`filtered_intersect`** instruction can filter sets before computing the intersection. These code transformations are translated into the primitive ISA instructions (§4.2) by the compiler back-end. At runtime, CPU executes these instructions by forwarding them to the memory controller, bypassing the cache hierarchy. Because NDMiner only processes read-only data (Takeaway 4), bypassing the cache hierarchy does not affect the correctness of the program as all the cached data is always in clean state.

5 NDMINER HARDWARE ARCHITECTURE

Fig. 7 shows an overview of the NDMiner hardware design. Upon receiving an NDP instruction, the NDMiner memory controller front-end converts it into multiple composite loads and set operations for offloading to DRAM. This section goes over the details of NDMiner hardware design that includes the design of the memory controller, near-memory compute units, and the DRAM access protocol. NDMiner targets a minimally invasive design, where we aim to utilize the existing hardware resources as much as possible.

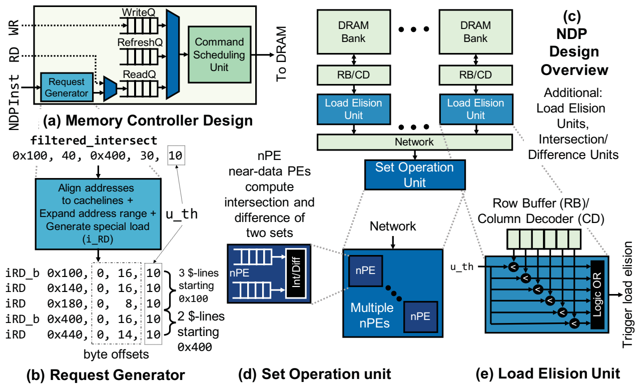


Figure 7: Hardware design overview of NDMiner.

5.1 NDMiner Memory Controller Front-end Design

Fig. 7(a) shows the NDMiner memory controller design. We introduce a front-end logic unit called the *request generator* that converts NDP instructions into DRAM requests. This unit accepts all three instructions discussed in §4.2 that perform different operations. Next, we take an example of filtered intersection to describe this hardware in detail. The incoming `NDPInst` specifies base addresses of two sets as `0x100` and `0x400`, and lengths of 40 and 30, respectively. Each element in a set is 4B long; there are 16 elements in a cache line. The instruction also indicates a threshold (`u_th`) of 10, *i.e.,* the intersection result must have element values less than 10. With this information, the request generator unit first aligns the addresses to cache line boundaries, and marks the range of byte offsets to read from each cache line. This unit also creates read requests with a unique opcode (*i.e.,* `iRD`) indicating intersection operations. The figure shows two opcodes: `iRD` and `iRD_b`. The latter one marks the beginning of a cache line for a set. The generated request also contains byte offsets and `u_th` as shown in Fig. 7(b). These requests are then enqueued into the memory controller queue.

5.2 NDMiner Memory-side Hardware Design

Fig. 7(c,d) show the set operation unit located at the buffer chip of DRAM based on Takeaway 5. It reads two sets from DRAM banks, and computes intersection or difference. As shown in 7(d), the near-data Processing Engines (nPEs) employ buffers to temporarily store the cache line of one set while the other set is being read from DRAM. After the first cache lines of both sets are read, simple comparator logic starts computing intersection/difference result. Each nPE employs the set operation logic similar to lines 16–26 in Algorithm 1. Sorted neighborhood sets (§2.1) preclude the necessity for all-to-all comparisons for set operations, and simplify the hardware design of nPEs. For each operation, the nPE is blocked until its completion. We name this design choice **NDMiner-Base**, where NDMiner employs one nPE per DIMM. While fetching two sets from different banks, NDMiner-Base can exploit as much as 2× compute bandwidth compared to moving data off-chip. Because GPM uses read-only data structures, lack of stores prevents memory consistency and coherency (between NDP and CPU caches) issues.

5.3 NDMiner Command Scheduling

This unit dequeues requests from the memory controller and issues commands to memory. In addition to issuing regular DRAM requests, the NDMiner command scheduler also issues NDP requests using unique opcodes. To support NDP at a minimal hardware overhead, NDMiner communicates compute operations in terms of DRAM commands, as opposed to a prior work [25] that issues composite operations.

All of the NDMiner operations are performed in conjunction with memory reads. For example, an intersection operation first reads operands from memory. Therefore, NDMiner issues compute commands following row activate and prior to row precharge. To issue commands for requests generated in Fig. 7(b), first, an `ACT` command opens a DRAM row. Then, an `iRD_b` command blocks an nPE for intersection and reads the first cache line to the set operation unit. On the address and data buses, the memory controller sends row/column addresses along with metadata for computation (*i.e.,* byte offsets and vertex threshold) in a time-multiplexed fashion. This obviates the need to add extra buses to support NDP. While discussed designs enable computation offloading to NDP, other non-GPM workloads can still use traditional request queues and command scheduling logic to access main memory.

6 DESIGN OPTIMIZATIONS

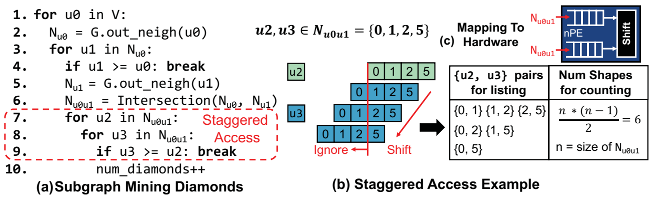
To further improve the performance of NDMiner, this section presents novel optimization techniques.

6.1 NDMiner-LoadElision: Eliding Unnecessary Loads

Based on Takeaway 6, symmetry breaking results in wasted DRAM bandwidth. To alleviate this effect, we propose Load Elision Unit (LEU) that **breaks symmetry in hardware**. Fig. 7(c,e) show near-memory compute logic for eliding loads. This unit compares data values read from DRAM with `u_th` and raises a signal when further loads need to be terminated. It employs a set of comparators as shown in Fig. 7(e). If a neighbor value read is higher than `u_th`, it triggers load elision. Because this unit directly uses cache line values read from DRAM, it is placed at the column decoder output. With 16 banks per rank and 2 ranks in a DIMM, the load elision unit can exploit the compute bandwidth as high as 2×16 = 32× on a single DIMM compared to moving data off-chip. We name this design choice as **NDMiner-LoadElision**. While NDP operations do not transfer data off-chip, we use the data bus response to indicate the termination of reads when the load elision is triggered. The memory sends a pre-encoded response (*e.g.,* `ff`) back to the memory controller indicating a load elision event. This response enables the memory controller to find the pending load requests for termination.

6.2 NDMiner-Overlap: Offloading Concurrent Instructions

With one nPE per DIMM, the near-memory set operation units can only exploit up to 2× compute bandwidth compared to processing data off-chip. While this is favorable, there is still 16× data bandwidth left unexploited. Moreover, a simple nPE design incurs low integration cost within the DRAM. To match the available data bandwidth, NDMiner integrates 16 nPEs per DIMM as shown in Fig. 7(d). We name this design **NDMiner-Overlap**, as multiple nPEs can overlap set operations. This includes (a) concurrently reading operands from multiple banks to exploit bank-level parallelism, and (b) concurrent set computation. While this is not a novel optimization, separating this design choice from NDMiner-Base helps us understand the potential of GPM workloads to exploit in-DRAM data parallelism.



```
1. for u0 in V:
2.   N_u0 = G.out_neigh(u0)
3.   for u1 in N_u0:
4.     if u1 >= u0: break
5.     N_u1 = G.out_neigh(u1)
6.     N_u0u1 = Intersection(N_u0, N_u1)
7.     for u2 in N_u0u1:
8.       for u3 in N_u0u1:
9.         if u3 >= u2: break
10.          num_diamonds++
```

{u2, u3} pairs for listing	Num Shapes for counting
{0, 1} {1, 2} {2, 5} {0, 2} {1, 5} {0, 5}	$\frac{n * (n-1)}{2} = 6$ n = size of N_{u0u1}

Staggered Access
(a)Subgraph Mining Diamonds
$u2, u3 \in N_{u0u1} = \{0, 1, 2, 5\}$
u2
u3
0 1 2 5
Ignore
Shift
(b) Staggered Access Example
(c) Mapping To Hardware
N_{u0u1}
nPE
Shift

Figure 8: Proposed compiler optimizations and corresponding computation mapping to hardware to improve the algorithmic efficiency of sparse GPM. Consecutive loops iterating over same sets are flattened to perform one set read and a composite computation (shift and record in this example).

6.3 NDMiner-Compiler: Optimizing Algorithmic Efficiency

Based on Takeaway 7, mining sparse patterns involve redundant load and computation operations. For example, executing lines 7 and 8 in Fig. 8 would read N_{u0u1} several times to the NDP units redundantly loading the same data. To improve the algorithmic efficiency of these workloads, we propose the following compiler-based optimizations. First, the compiler identifies the existence of redundant reads by examining the candidate sets used in consecutive loops. As shown in Fig. 8(a), two loops in lines 7 and 8 iterate over the same candidate set N_{u0u1}. Furthermore, line 9 imposes a symmetry breaking constraint between $u2$ and $u3$.

Upon this identification, we propose to **flatten** the **loop nest** and convert it into one set read and a **composite computation**. For example, loop nest flattening in Fig. 8(a) is converted into a staggered access of N_{u0u1} as shown in Fig. 8(b). Symmetry breaking constraint is the reason for this type of access pattern because $u2$ cannot be greater than $u3$. We further map this computation in hardware to nPEs, where the same candidate set is replicated in two buffers, and $\{u2, u3\}$ pairs can be found by using a shift-and-record operation. While this is useful for pattern listing algorithms, pattern counting algorithms can directly compute the number of patterns by using simple accumulation equation as shown in Fig. 8(b). In addition to loop nest flattening, our compiler pass also hoists loop invariant computations outside the loop. This includes, for example, moving N_{u1} computation in line 7 in Fig. 3 before line 5 as neighborhood of $u1$ is independent of the value of $u2$. Applying compiler optimizations significantly improves algorithmic efficiency of sparse pattern mining; we name this design choice **NDMiner-Compiler**.

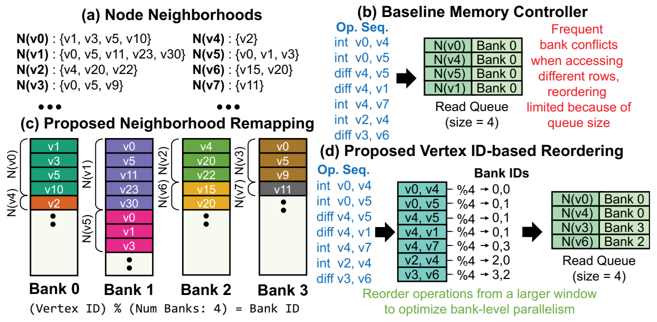


Figure 9: (a) Example graph's node neighborhoods, (b) baseline memory controller with a size-limited queue that leads to frequent bank conflicts when accessing different rows, (c) proposed neighborhood remapping scheme using a deterministic interleaving of neighborhoods across banks, and (d) vertex ID-based set operation reordering to exploit bank-level parallelism in DRAM.

DRAM Specification
DDR4-3200, 4Gb ×8, 4 Channels × 1 DIMM × 2 Ranks 32-entry RD/WR queue, FR-FCFS, Skylake address mapping [41]
DRAM Timing Parameters

tRCD=22, tCL=22, tRP=22, tBL=4. tCCD_S=4, tCCD_L=10,
tRRD_S=4, tRRD_L=8, tFAW=34, tRC=78

DRAM Energy Parameters
IDD and VDD parameters obtained from [44]
nPE, LEUs, and Reordering Unit Parameters
16 nPEs and 32 LEUs per channel @1.6GHz, 1024-entry vertex ID-based reordering unit on CPU

Table 2: NDMiner system parameters.

6.4 NDMiner-Reorder: Reordering Set Operations

Based on Takeaway 8, it is possible to improve the performance of GPM workloads by reordering set operations to exploit parallelism in DRAM. To further understand the reason behind this performance difference, consider an example where Fig. 9(a) shows neighborhoods of selected number of nodes in a hypothetical graph. Fig. 9(b) shows that a traditional memory controller falls short in identifying operation reordering opportunity because of its size-limited queues. This can result in frequent bank conflicts if row IDs of queued requests are different. One straightforward way to improve the performance is by increasing the size of the memory controller read queue and let the memory controller reorder a larger number of read requests. This, however, is not a practical design as it will significantly increase the latency of memory controller reordering logic, potentially hurting performance of other applications. Any other technique that uses addresses to reorder set operations would incur a similarly large overhead. Therefore, **we propose to raise the level of abstraction and reorder set operations based on vertex IDs at low cost.**

The intuition behind our proposal is to encode the vertex ID in the bank address to find a node's neighborhood. This allows us to compute the bank address of each set operation at a low-cost, obviating the necessity to decode an entire address. This can further be used to reorder operations from a *large window size* to maximize bank-level parallelism. Fig. 9(c,d) explain our design with an example. We propose to **remap** each node's neighborhood to different banks based on computing a simple hash function of a vertex ID. While this paper uses a modulo operation to map each vertex ID to a bank, this is not a fundamental limitation, and this operation can be replaced by a more sophisticated hash function, if necessary. The row and column addresses are then encoded to have a contiguous neighborhood mappings of two vertices without overwriting each others' data. A physical address from DRAM row, column, bank, bank group, rank, and channel coordinates is calculated based on a prior work [41]. Fig. 9(c) shows the resultant mapping of nodes v0-v7's neighborhoods. Notably, the graph is remapped only once as a pre-processing step, and it is agnostic to any specific pattern being mined. In practice, we find that the remapping cost is at least an order of magnitude smaller than workload execution, which can be amortized over multiple runs of GPM algorithms. Because remapping is a pre-processing step, it does not cause TLB shootdown during workload execution.

At runtime, this mapping information is used to intelligently **reorder** and selectively schedule set operations to maximize data parallelism in DRAM. Fig. 9(d) shows the functionality of reordering hardware located on the CPU, which takes an operation sequence as an input and computes bank addresses of neighborhoods used in each set operation. This is computed by simply applying a modulo function to vertex IDs. In hardware, modulo operation translates to simply selecting a few low significant bits, which can be executed in parallel efficiently. Based on the bank IDs, set operations are reordered to have distinct bank IDs in the consecutive operations in the reordered sequence. Based on this reordering, a subset of these operations are offloaded to the memory controller based on the empty slots in the queue. Because the proposed vertex-based reordering scheme enables bank address identification at an extremely low cost, it is possible to reorder operations from a much larger window size compared to a size-limited memory controller queue. As presented in Fig. 4, this reordering has a potential to result in a significant performance improvement. We name this design **NDMiner-Reorder**.

7 EVALUATION METHODOLOGY

7.1 Baseline CPU Hardware Platform

For the software baselines, we use an AMD EPYC 7742 processor with 64 physical cores (128 SMT threads). The aggregate Last Level Cache (LLC) size is 256MB. The main memory in the system is a 4-channel DDR4-3200 with a 512GB capacity. A prior work [12] shows that enabling hyperthreading for GPM workloads slows down performance scaling due to cache contention. Therefore, we use 64-thread implementations of our software baselines.

7.2 Simulation Infrastructure

We model the cycle-accurate NDMiner performance using Ramulator [26]. Ramulator is a DRAM simulator cross-validated against real DRAM devices, and extensively used by prior works [25, 59] to estimate the performance of NDP systems over real CPU baselines. We faithfully model NDP units and their latencies in Ramulator

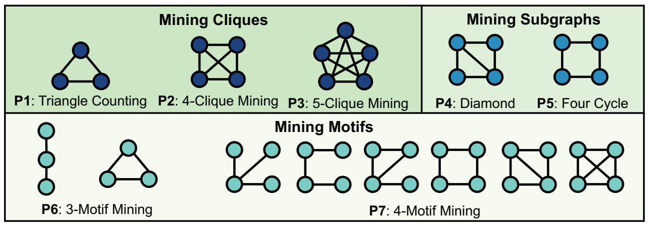


Figure 10: Input graph patterns used for evaluation.

Graph	#Vtx	#Edge	Size (MB)	Avg Degree	Description
wiki-vote (wi)	7.1k	103.7k	0.5	14.6	Voting network
pokec (po)	1.6M	30.6M	129.3	19.1	Social network
patents (pa)	3.7M	16.5M	91.8	4.4	Citation network
livejournal (lj)	4.0M	34.7M	162.8	8.7	Social network
orkut (or)	3.1M	117.8M	470.5	38.1	Social network

Table 3: Real-world graph datasets used for evaluation.

based on detailed RTL models. We also validate 1) the timing model of different input set sizes resulting in unique read vs. computation times, and 2) the scheduling decisions in presence of NDP constraints. The configuration of modeled memory system is shown in Table 2. We generate a trace of NDP instructions to feed into Ramulator and model the NDMiner hardware modifications presented above. As neighborhood set load, intersection, and difference operations take a majority of workload execution time, we model this computation in Ramulator and compare it with other baselines. Notably, in addition to this computation, GPM algorithms perform other simple computations including shape count increments. These operations are left to be performed efficiently using a multi-threaded host CPU. To estimate the latency, energy consumption, and area overhead of NDP logic, we model NDMiner using System Verilog HDL and synthesize using a commercial 28nm technology library using the Synopsys Design Compiler. For vector-based power estimation, we use Synopsys PrimeTime. While the nPEs can be clocked at a higher frequency in a logic process, we conservatively clock them at a lower frequency as they use slower transistors of the DRAM process.

7.3 Algorithms and Datasets

Algorithms. We mine seven patterns **P1–P7** of varying sizes and connectivity as shown in Fig. 10. The first six patterns are the same as what a prior work FlexMiner [12] used. In addition, we also use 4-motif counting (P7) for comprehensive evaluation. Among these patterns, the cliques (P1–P3) are dense, fully connected patterns, and P4–P5 are sparse patterns. Motif counting counts all possible patterns with a specified number of vertices (*i.e.,* two patterns for 3-MC and six patterns for 4-MC) that includes both dense and sparse patterns. While we choose these five patterns for evaluation, NDMiner is agnostic to any specific pattern, and it can work well for any arbitrary user-defined pattern. As detailed in prior works [8, 12], the simulation times for mining large patterns is quite high (*e.g.,* days to weeks); hence we mine patterns of up to five vertices.

	Num nPEs per channel	Load Elision	Loop Nest Flattening	Op Reorder
NDMiner-Base	1	✗	✗	✗
NDMiner-LoadElision	1	✓	✗	✗
NDMiner-Overlap	16	✓	✗	✗
NDMiner-Compiler	16	✓	✓	✗
NDMiner-Reorder	16	✓	✓	✓

Table 4: NDMiner configurations.

Datasets. We use five real-world graph datasets for evaluation as shown in Table 3. These datasets are diverse in terms of their sizes from small (wiki-vote) to large (orkut), and connectivity (*i.e.,* average degrees). Notably, the amount of simulation time grows exponentially with the graph size. Hence, we use similar sized datasets as prior works [12, 60]. We set a simulation timeout of 120 hours (five days) and do not include the results for workloads that do not finish execution in this time. This mostly includes mining large number of patterns (P7) on large datasets with slower baselines.

7.4 NDMiner Configurations

To present the benefit of proposed optimization techniques, we compare NDMiner configurations listed in Table 4.

7.5 State-of-the-art Baselines

We also rigorously compare NDMiner with the following software and hardware baselines. We run all software baselines on server-grade CPU (§7.1) for 10 times and use an average execution time to reduce noise in measurements.
GAPBS+GraphPi (software) extracts algorithms from GraphPi [48] including optimized schedules and symmetry breaking constraints and implements them onto GAPBS [6] data structures using a BFS-based search tree traversal. This baseline is validated against vanilla GraphPi using output shape counts. The purpose of this baseline is to evaluate GraphPi algorithms on optimized GAPBS graph data structures without framework overheads.
GraphPi (software) uses vanilla open-source GraphPi [48].
Pangolin (software) is a collection of open-source benchmarks [11] based on the implementations of state-of-the-art GPM frameworks including Pangolin [10] and Sandslash [9].
FlexMiner (hardware) is based on a GPM hardware accelerator [12]. To obtain FlexMiner execution time, we run the CPU baseline code open-sourced by authors in GraphMinerBench [11] on an Intel i9 machine (same as used in their paper), and multiply speedup factors reported in the paper for commonly evaluated algorithms and datasets.
SISA and IntersectX (hardware). We qualitatively compare NDMiner with these baselines [8, 43] as their open-source implementations are not available.

8 EVALUATION RESULTS

8.1 Performance Analysis

Comparison of different NDMiner configurations. We first compare the performance of various NDMiner baselines (§7.4) to estimate the effectiveness of proposed design optimizations. Fig. 11 shows the performance of NDMiner configurations normalized to NDMiner-Base. NDMiner-LoadElision outperforms NDMiner-Base by 2.1×, on average by breaking symmetry in hardware and

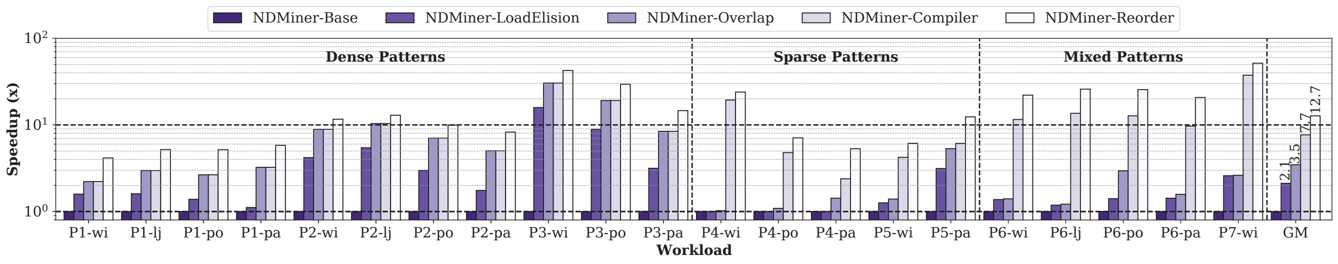


Figure 11: Performance comparison of NDMiner configurations showing the effectiveness of proposed optimizations. Workloads that do not simulate in 120 hours are excluded that mostly include P7 mining. All proposed optimizations together improves the performance of NDMiner-Base by 12.7×, on average.

avoiding unnecessary loads. NDMiner-Overlap further outperforms NDMiner-Base by 3.5×, on average, showing that adding extra nPEs marginally improve performance. This result also shows that merely adding 16× more NDP compute resources does not automatically offer significant performance, especially for sparse patterns. To best tap the potential of NDP, we need further optimizations.

Fig. 11 further shows that NDMiner-Compiler significantly improves the performance of sparse GPM algorithms (*i.e.,* P4–P7), resulting in an average improvement of 7.7×. Note that this optimization is not applicable to dense patterns P1–P3. The benefit of this optimization is attributed to the improved algorithmic efficiency, where NDMiner-Compiler avoids unnecessary load and compute operations. NDMiner-Reorder further improves the performance of GPM workloads by 12.7×, on average, compared to NDMiner-Base. This configuration outperforms all other baselines by introducing set operation reordering. This reordering fills up the size-limited memory controller queue by requests that can be serviced by different banks, ranks, and channels concurrently to optimize internal DRAM data parallelism.

NDMiner versus state-of-the-art baselines. Fig. 12 compares the performance of NDMiner with prior software and hardware optimizations for GPM. This comparison is conducted with our best-performing configuration, *i.e.,* NDMiner-Reorder. NDMiner significantly outperforms three strong software baseline, *i.e.,* GAPBS + GraphPi [6, 48], vanilla GraphPi [48], and Pangolin [10] by 7.4×, 6.4×, and 10.9×, on average. Our detailed investigation reveals that NDMiner uses the same traversal order and symmetry breaking constraints as other baselines. Therefore, these significant benefits are attributed to (a) reducing the off-chip data transfer using NDP, (b) hardware-based load elision with the knowledge of symmetry breaking constraints, (c) optimizing algorithmic efficiency of sparse patterns, and (d) exploiting high in-DRAM compute bandwidth by appropriately reordering set operation (and not because of better algorithms from GraphPi).

Fig. 12 also shows that NDMiner outperforms FlexMiner [12] on commonly evaluated algorithm-dataset pairs by 2.5×, on average. While FlexMiner improves GPM performance over CPU by domain-specialization, it uses a traditional memory architecture with on-chip caches and off-chip DRAM. Our profiling, however, shows that GPM workloads exhibit wasteful behavior on a traditional memory hierarchy, and can be significantly optimized by using NDP. NDMiner outperforms FlexMiner by offloading computation to NDP units, improving the algorithmic efficiency of sparse pattern mining, and reordering set operations to exploit abundant in-DRAM data parallelism.

	Dense Patterns			Sparse Patterns		Mixed Patterns	
	P1	P2	P3	P4	P5	P6	P7
Loads	4.1×	5.4×	4.9×	2.8×	1.6×	7.9×	12.7×
Comparisons	4.6×	5.0×	4.3×	1.0×	1.6×	4.6×	1.5×

Table 5: Reduction in loads and element-wise comparisons in set operations due to load elision and compiler optimizations. Results averaged over different datasets.

We qualitatively compare NDMiner with SISA [8] and IntersectX [43] because of their lack of available open-source implementations. While SISA efficiently maps GPM algorithms to set operations, it employs general-purpose NDP hardware (*e.g.,* Ambit [46]) to offload computation. NDMiner, on the other hand, employs domain-specialized NDP hardware design, circumvents unnecessary reads and computations, and reorders set operations to acquire additional performance from NDP. IntersectX optimizes GPM workloads on a CPU using a stream instruction set and its microarchitectural support. This, however, fetches data from off-chip DRAM that suffers from wasted DRAM bandwidth. Similar to FlexMiner, the performance of IntersectX can further be improved by NDMiner's domain-specialized NDP design.

Reduction in loads and computation. To better understand the performance benefits of NDMiner, Table 5 shows the reduction in the number of load and element-wise comparisons for computing set operations. NDMiner avoids unnecessary loads and element-wise comparisons by (a) hardware-based load elision (§6.1), and (b) software-based compiler optimizations using loop nest flattening and instruction hoisting (§6.3). Dense workloads only benefit from load elision that significantly improves their algorithmic efficiency. This is because dense patterns use a unique symmetry breaking constraint for each set operation, where load elision is effective. Sparse patterns, on the other hand, often compute a set operation once and reuse its result multiple times. Because each such usage might have a unique constraint, this sometimes precludes the employment of load elision because the entire set needs to be computed once. P4 (diamond) is one such pattern where intersection result is used several times with different constraints. This pattern, however, still benefits from our loop nest flattening technique and reduces the number of loads. Motif counting (P6-P7) algorithms

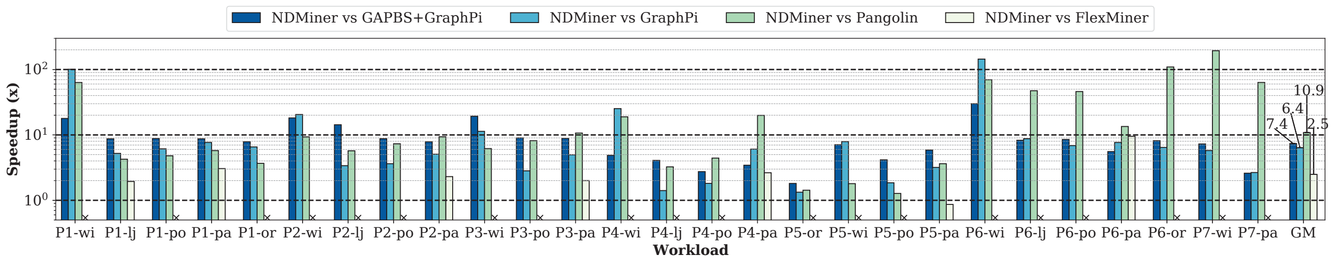


Figure 12: Performance comparison of state-of-the-art software and hardware baselines with NDMiner showing significant performance improvements. FlexMiner [12] is only compared against commonly evaluated datasets (others marked with "x" on x-axis). Workloads that time out are excluded.

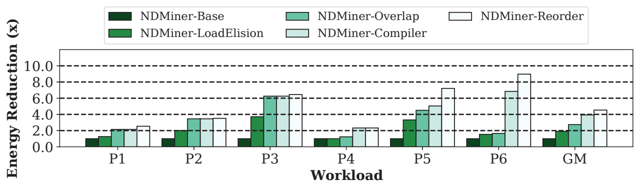


Figure 13: Energy consumption of NDMiner configurations normalized to NDMiner-Base on a representative patents dataset. P7 is excluded as its baseline simulation times out.

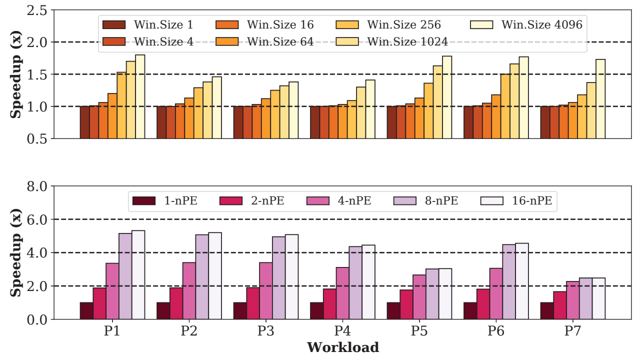


Figure 14: Performance sensitivity of NDMiner for different set operation reordering window sizes (top) and number of nPEs per channel (bottom) on a representative patents dataset.

	near-memory PE (nPE)	Load Elision Unit (LEU)	Operaration Reorder Unit
Location	DRAM	DRAM	CPU
Area (mm^2)	0.01237	0.00096	0.4147
Power (mW)	18.45	0.36	32.78

Table 6: Area and power estimates of NDMiner circuits.

mine several patterns, offering better opportunity for both load elision and compiler optimizations to be effective.

8.2 Energy Analysis

Fig. 13 compares the energy of different NDMiner configurations, normalized to NDMiner-Base, using a representative patents dataset. Energy of mining 4-motif (P7) is not reported as its simulation for NDMiner-Base times out. The figure shows that proposed optimizations improve the energy consumption of NDMiner-Base by 1.8×, 2.8×, 3.9×, and 4.7×, on average. This significant energy reduction is attributed to (a) improved memory traffic and algorithmic efficiency by load elision and compiler optimizations, and (b) reduction in static energy by speeding up the program execution by using multiple nPEs per channel and reordering set operations to exploit internal DRAM data parallelism.

8.3 Sensitivity Analysis

Fig. 14 shows the performance sensitivity of NDMiner compared to (a) different set operation reordering window sizes and (b) number of nPEs per channel. The top figure shows that increasing the window size from 1 to 4096 monotonically increases the performance by 1.6×, on average. Interestingly, there is a marginal performance increase from 1024 to 4096. The silicon and power costs, on the other hand, would increase significantly by increasing a window size by 4×. Therefore, NDMiner design employs a window size of 1024 that best trades off area and power costs with performance.

Fig. 14 (bottom) shows that the performance of NDMiner improves by 4.2× on average with an increase in the number of nPEs from 1 to 16. This improved performance shows the opportunity to overlap large portions of compute operations by availing ample in-DRAM compute bandwidth. This trend, however, slowly saturates beyond 8 nPEs, at which point, the workload gradually shifts from being compute bounded to memory bounded. Although using 16 nPEs marginally improves performance, the area and power overhead of integrating nPEs are minimal (discussed in §8.4), which informs our choice of using 16 nPEs per channel.

8.4 Overhead Analysis

Table 6 shows the post-synthesis area and power overheads of NDMiner hardware. While the table shows overheads of individual circuits, NDMiner design integrates 16 nPEs and 32 LEUs in a DRAM DIMM, and one set operation reordering unit on the CPU. The area and power of NDMiner is dominated by the reordering unit as it employs two 1024-entry buffers (one to store incoming NDP instructions and the other to store reordered instructions). The cost of these hardware units, however, is negligible compared to the performance benefit they provide. Compared to a $100mm^2$ [32] area of the DRAM buffer chip, NDMiner circuits add a minimal

	Symm. Break.	NDP	Load Elision	Loop Nest Flattening	Op Reorder
GraphZero [30]	✓	✗	✗	✗	✗
GraphPi [48]	✓	✗	✗	✗	✗
Gramer [60]	✗	✗	✗	✗	✗
FlexMiner [12]	✓	✗	✗	✗	✗
SISA [8]	✗	✓	✗	✗	✗
IntersectX [43]	✓	✗	✗	✗	✗
NDMiner	✓	✓	✓	✓	✓

Table 7: Comparison of NDMiner with related works.

area overhead of 0.23%. On the flip side, NDMiner significantly improves GPM performance by 7.4×, on average.

9 RELATED WORK

Table 7 provides a brief comparison of the most related works with NDMiner. A more detailed comparison follows.

GPM software systems. Numerous software frameworks efficiently utilize GPM algorithms on CPUs and GPUs. Early GPM systems [56] rely on enumerating all possible embeddings, and then ruling out redundant embeddings using isomorphism tests. Recent works [10, 22, 23, 30, 31, 48] avoid the expensive filter operations and prune out redundant embeddings during the search tree expansion. Other works strive to reduce the memory consumption of intermediate embeddings either by relying on SSD [58] or leveraging algorithmic techniques [16]. In addition to optimized software implementations, this paper shows that GPM performance can be further improved using hardware-based techniques.

NDP architectures. To alleviate the cost of data transfer over bandwidth-limited and energy-hungry CPU-memory bus, several NDP architectures are proposed. Of these works, OMEGA [1] and PHI [35] augment the CPU memory with low-cost compute units for graph processing. Other works [2, 7, 15, 61, 62] offload graph computations to the logic layer of HMC. These proposals, however, are suitable for graph processing and cannot be directly applied for GPM acceleration because of its unique workload characteristics. For GPM, SISA [8] proposes to offload computation on existing PIM architectures by proposing set-centric ISA and fast set intersection algorithm. NDMiner improves SISA using domain-specific optimizations (hardware load elision, compiler optimizations, and set operation reordering). Outside the context of graph computation, several other NDP architectures are proposed [13, 25, 27, 29, 40, 46, 47, 51–53].

Domain-specific accelerators. ExTensor [21] employs fast intersection circuits for tensor algebra that cannot be used for GPM out-of-the-box as it does not support key operations like pattern enumeration. Numerous graph processing accelerators aim at improving the irregular memory accesses via memory system optimizations [3, 5, 20, 33, 34, 37, 49, 54, 59]. As detailed in [60], graph processing and GPM workloads have distinct memory access patterns. Therefore, the effectiveness of graph processing accelerators might be limited when applied to GPM. Few recent works [12, 24, 43, 60] design specialized architectures for GPM. Out of these, FlexMiner [12] improves the performance and generality of prior accelerators [24, 60] by proposing a pattern-aware GPM accelerator. NDMiner outperforms FlexMiner by employing a domain-specific NDP architecture that includes novels optimizations like loop nest flattening and set operation reordering. IntersectX [43] optimizes GPM execution on a CPU by extending the ISA and architecture support. This approach, however, suffers from high on-chip storage requirement and unnecessary off-chip data transfers from DRAM. NDMiner offloads compute to low-cost NDP units augmented with domain-specific optimizations.

10 CONCLUSION

Irregular memory and complex data-dependent control flow instructions used in set operations dominate the execution time of GPM workloads. This paper presented NDMiner—a domain-specialized NDP architecture to accelerate GPM. NDMiner offloaded the costly set computations to NDP. NDMiner further improved performance by uncovering and applying domain-specific optimizations. NDMiner integrated a near-data load elision unit that broke symmetry in hardware and terminated unnecessary loads. NDMiner employed compiler optimizations and hardware mapping techniques that improved the algorithmic efficiency of sparse GPM workloads. NDMiner proposed a graph remapping scheme and set operation reordering hardware to optimize the bank, rank, and channel-level parallelism in DRAM. Using dense, sparse, and mixed pattern mining algorithms, we showed that NDMiner significantly outperforms the state-of-the-art software (GraphPi) and hardware (FlexMiner) baselines by 6.4× and 2.5×, on average, at a negligible cost.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful feedback and Arun Subramaniyan for detailed discussions. The material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

REFERENCES

[1] Abraham Addisie, Hiwot Kassa, Opeoluwa Matthews, and Valeria Bertacco. 2018. Heterogeneous Memory Subsystem for Natural Graph Analytics. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 134–145. https://doi.org/10.1109/IISWC.2018.8573480

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 105–117. https://doi.org/10.1145/2749469.2750386

[3] S. Ainsworth and T. M. Jones. 2016. Graph Prefetching Using Data Structure Knowledge. In *ICS* (Istanbul, Turkey). New York, NY, USA, Article 39, 11 pages. https://doi.org/10.1145/2925426.2926254

[4] V. Balaji and B. Lucia. 2018. When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs. In *IISWC*. 203–214.

[5] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads. In *HPCA*. 373–386. https://doi.org/10.1109/HPCA.2019.00051

[6] S. Beamer, K. Asanović, and D. Patterson. 2015. The GAP Benchmark Suite. In *arXiv:1508.03619 [cs.DC]*.

[7] Leul Wuletaw Belayneh and V. Bertacco. 2020. GraphVine: Exploiting Multicast for Scalable Graph Analytics. *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2020), 762–767.

[8] 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. *2021 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2021).

[9] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 378–391. https://doi.org/10.1145/3447818.3460359

[10] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proc. VLDB Endow.* 13, 8 (April 2020), 1190–1205. https://doi.org/10.14778/3389133.3389137

[11] Xuhao Chen and Tianhao Huang. 2021. GraphMinerBench open-source implementations. In *Github Repository*. https://github.com/chenxuhao/GraphMiner

[12] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and Arvind. 2021. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. 581–594.

[13] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 27–39. https://doi.org/10.1109/ISCA.2016.13

[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. https://doi.org/10.1109/TITB.2009.2028234

[15] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2019. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2019), 640–653. https://doi.org/10.1109/TCAD.2018.2821565

[16] Vinicius Dias, Carlos HC Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*. 1357–1374.

[17] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. Mckenzie. 1999. Computational RAM: implementing processors in memory. *IEEE Design Test of Computers* 16, 1 (Jan 1999), 32–41. https://doi.org/10.1109/54.748803

[18] Mojtaba Eskandari and Hooman Raesi. 2014. Frequent sub-graph mining for intelligent malware detection. *Security and Communication Networks* 7, 11 (2014), 1872–1886. https://doi.org/10.1002/sec.902 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.902

[19] M. Gokhale, B. Holmes, and K. Iobst. 1995. Processing in memory: the Terasys massively parallel PIM array. *Computer* 28, 4 (Apr 1995), 23–31. https://doi.org/10.1109/2.375174

[20] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783759

[21] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.

[22] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: A Pattern-Aware Graph Mining System. , Article 13 (2020), 16 pages. https://doi.org/10.1145/3342195.3387548

[23] Kasra Jamshidi and Keval Vora. 2021. A Deeper Dive into Pattern-Aware Subgraph Exploration with PEREGRINE. *SIGOPS Oper. Syst. Rev.* 55, 1 (June 2021), 1–10. https://doi.org/10.1145/3469379.3469381

[24] Oren Kalinsky, Benny Kimelfeld, and Yoav Etsion. 2020. The TrieJax Architecture: Accelerating Graph Operations Through Relational Joins. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1217–1231. https://doi.org/10.1145/3373376.3378524

[25] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 790–803. https://doi.org/10.1109/ISCA45697.2020.00070

[26] Y. Kim, W. Yang, and O. Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Comput. Archit. Lett.* 15, 1 (Jan. 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[27] Sukhan Lee, Shin haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, Seongil O, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*.

[28] Yutaka I. Leon-Suematsu, Kentaro Inui, Sadao Kurohashi, and Yutaka Kidawara. 2011. Web Spam Detection by Exploring Densely Connected Subgraphs. In *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 01 (WI-IAT '11)*. IEEE Computer Society, USA, 124–129. https://doi.org/10.1109/WI-IAT.2011.152

[29] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2897937.2898064

[30] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, , and B. Wu. 2019. Graphzero: Breaking symmetry for efficient graph mining. In *arXiv preprint arXiv:1911.12877*.

[31] Daniel Mawhirter and Bo Wu. 2019. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 509–523.

[32] P. J. Meaney, L. D. Curley, G. D. Gilda, M. R. Hodges, D. J. Buerkle, R. D. Siegl, and R. K. Dong. 2015. The IBM Z13 Memory Subsystem for Big Data. *IBM J. Res. Dev.* 59, 4–5 (July 2015), 4:1–4:11. https://doi.org/10.1147/JRD.2015.2429031

[33] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14. https://doi.org/10.1109/MICRO.2018.00010

[34] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2017. Cache-Guided Scheduling: Exploiting caches to maximize locality in graph processing. *AGP'17* (2017).

[35] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. *MICRO '52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 1009–1022. https://doi.org/10.1145/3352460.3358254

[36] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. 1998. Active Pages: A Computation Model for Intelligent Memory. *SIGARCH Comput. Archit. News* 26, 3 (April 1998), 192–203. https://doi.org/10.1145/279361.279387

[37] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 166–177. https://doi.org/10.1109/ISCA.2016.24

[38] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A Case for Intelligent RAM. *IEEE Micro* 17, 2 (March 1997), 34–44. https://doi.org/10.1109/40.592312

[39] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. 1997. Intelligent RAM (IRAM): chips that remember and compute. In *1997 IEEE International Solids-State Circuits Conference. Digest of Technical Papers*. 224–225. https://doi.org/10.1109/ISSCC.1997.585348

[40] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. 2019. Opportunistic computing in gpu architectures. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 210–223.

[41] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *25th USENIX security symposium (USENIX security 16)*. 565–581.

[42] B. Prakash. 2015. Graph Mining for Cyber Security. *Advances in Information Security* 56 (04 2015), 287–306. https://doi.org/10.1007/978-3-319-14039-1_14

[43] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. 2021. IntersectX: An Efficient Accelerator for Graph Mining. *arXiv:2012.10848v4* (2021).

[44] Samsung. 2018. DDR4 SDRAM Data sheet, 288-pin Load Reduced DIMM (LRDIMM)-based 8GB C-Die.

[45] 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 *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 185–197.

[46] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-Memory Accelerator for Bulk Bitwise

Operations Using Commodity DRAM Technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 273–287.

[47] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 14–26. https://doi.org/10.1109/ISCA.2016.12

[48] T. Shi, M. Zhai, Y. Xu, and J. Zhai. 2020. *GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination.* Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.

[49] Shreyas G. Singapura, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2017. OSCAR: Optimizing SCrAtchpad reuse for graph processing. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2017.8091070

[50] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating Graph Processing Using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 531–543. https://doi.org/10.1109/HPCA.2018.00052

[51] Nishil Talati, Ameer Haj Ali, Rotem Ben Hur, Nimrod Wald, Ronny Ronen, Pierre-Emmanuel Gaillardon, and Shahar Kvatinsky. 2018. Practical challenges in delivering the promises of real processing-in-memory machines. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1628–1633.

[52] Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsky. 2016. Logic design within memristive memories using memristor-aided loGIC (MAGIC). *IEEE Transactions on Nanotechnology* 15, 4 (2016), 635–650.

[53] Nishil Talati, Heonjae Ha, Ben Perach, Ronny Ronen, and Shahar Kvatinsky. 2019. Concept: A column-oriented memory controller for efficient memory and pim operations in rram. *IEEE Micro* 39, 1 (2019), 33–43.

[54] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. 2021. Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design. (2021), 654–667. https://doi.org/10.1109/HPCA51647.2021.00061

[55] Lei Tang and Huan Liu. 2010. Graph Mining Applications to Social Network Analysis. In *Managing and Mining Graph Data*.

[56] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 425–440.

[57] Johan Ugander, Lars Backstrom, and Jon Kleinberg. 2013. Subgraph Frequencies: Mapping the Empirical and Extremal Geography of Large Graph Collections. In *Proceedings of the 22nd International Conference on World Wide Web* (Rio de Janeiro, Brazil) *(WWW '13)*. Association for Computing Machinery, New York, NY, USA, 1307–1318. https://doi.org/10.1145/2488388.2488502

[58] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 763–782.

[59] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2019. Alleviating Irregularity in Graph Analytics Acceleration: A Hardware/Software Co-Design Approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 615–628. https://doi.org/10.1145/3352460.3358318

[60] P. Yao, L. Zheng, Z. Zeng, Y. Huang, C. Gui, X. Liao, H. Jin, and J. Xue. 2020. A Locality-Aware Energy-Efficient Accelerator for Graph Mining Applications. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 895–907. https://doi.org/10.1109/MICRO50266.2020.00077

[61] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 544–557. https://doi.org/10.1109/HPCA.2018.00053

[62] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. Graphq: Scalable pim-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 712–725.