DIMMining: Pruning-Efficient and Parallel Graph Mining on Near-Memory-Computing

Guohao Dai*†
Tsinghua University
BNRist, Beijing
China

Zhenhua Zhu*
Tsinghua University
BNRist, Beijing
China

Tianyu Fu
Tsinghua University
BNRist, Beijing
China

Chiyue Wei
Tsinghua University
BNRist, Beijing
China

Bangyan Wang
University of California, Santa Barbara, CA
USA

Xiangyu Li
Tsinghua University
BNRist, Beijing
China

Yuan Xie
University of California, Santa Barbara, CA
USA

Huazhong Yang
Tsinghua University
BNRist, Beijing
China

Yu Wang†
Tsinghua University
BNRist, Beijing
China

ABSTRACT

Graph mining, which finds specific patterns in the graph, is becoming increasingly important in various domains. We point out that accelerating graph mining suffers from the following challenges: (1) Heavy comparison for pruning: Pruning technique is widely used to reduce search space in graph mining. It applies constraints on vertex indices and involves massive index comparisons. (2) Low parallelism of set operations: The typical graph mining algorithms can be expressed as a series of set operations between neighbors of vertices, which suffer from low parallelism if vertices are streaming to the computation units. (3) Heavy data transfer: Graph mining needs to transfer intermediate data with two orders of magnitude larger than the original data volume between CPU and memory.

To tackle these challenges, we propose DIMMining with four techniques from algorithm to architecture perspectives. The **Index Pre-comparison** scheme is proposed for efficient pruning. We introduce the self anchor and neighbor partition to enable pre-comparison for vertex indices. Thus, we can reduce comparisons during runtime. We propose a **Flexible BCSR** (**B**itmap with **C**ompressed **S**parse **R**ow) format to enable parallelism for set operations from the data structure perspective, which works on continuous vertices without memory space overheads. The **Systolic Merge Array** is designed to further explore the parallelism on discontinuous vertices from the architecture perspective. Then, we propose a **DIMM-based Near-Memory-Computing** architecture, which eliminates the large-volume data transfer between the computation and the memory. Extensive experimental results on real-world graphs show that DIMMining achieves 222.23× and 139.51× speedup compared with FPGAs and CPUs, and 3.61× speedup over the state-of-the-art graph mining architecture.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Hardware** → **Memory and dense storage**.

KEYWORDS

Graph Mining, Near-Memory-Computing, Systolic Merge Array.

ACM Reference Format:
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 *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3470496.3527388

1 INTRODUCTION

As we are now in the big data era, graph processing has been widely used in many domains, such as social network analysis [1–3], machine learning [4–6], recommendation system [7–9], etc [10–18]. Previous studies have proposed several systems and architectures to accelerate conventional graph traversing problems [18–27]. Recent studies are paying more attention to emerging graph mining problems, which explore specific patterns in graphs [28–33].

*Both authors contribute equally to this work.
†Corresponding: daiguohao1992@gmail.com, yu-wang@tsinghua.edu.cn.

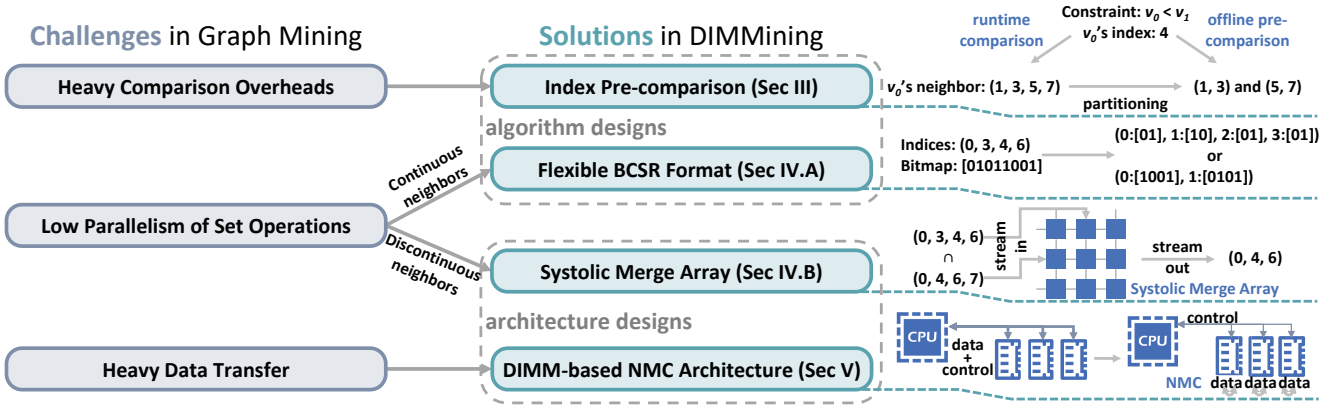


Figure 1: Graph mining problems suffer from challenges of poor locality, low parallelism, and heavy data transfer. We propose 4 solutions to tackle these challenges, illustrated on the right and detailed in corresponding sections.

Table 1: Graph Datasets, and the Time Ratio of Comparison Operation for Different Mining Tasks

Graph	$\|\mathbb{V}\|$	$\|\mathbb{E}\|$	3-CF	4-CF	5-CF	3-MC*
P2P(PP)[34]	11K	40K	25%	23%	23%	22%
Astro(AS)[34]	18K	0.2M	20%	27%	35%	19%
Mico(MI)[35]	0.1M	1.1M	24%	35%	45%	20%
Patents(PA)[36]	2.7M	14.0M	41%	41%	44%	33%
Youtube(YT)[37]	7.1M	57.1M	8%	6%	10%	38%
LiveJournal(LJ)[34]	4.8M	42.9M	20%	33%	53%	27%

*3-CF: 3 Clique Finding, 4-CF: 4 Clique Finding
5-CF: 5 Clique Finding, 3-MC: 3 Motif Counting

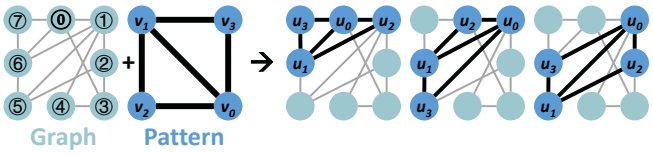
Table 2: Transferred Size for 3-CF

Graph	|Graph|	|Transferred Data|	|Transferred Data|/|Graph|
PP	240.6KB	6.042MB	26×
AS	1.660MB	86.73MB	52×
MI	9.028MB	420.4MB	47×
PA	147.3MB	64.92GB	451×
YT	485.2MB	74.86GB	158×
LJ	293.4MB	92.59GB	323×

However, accelerating graph mining suffers from the following challenges: (1) **Heavy comparison for pruning**: Pruning is widely used to reduce search space in graph mining. It applies constraints on indices of symmetry and isomorphism vertices in the pattern, which involves massive vertex index comparison operations. Table 1 shows the ratio of comparison time in different graph mining problems, which takes up 6% to 53% of total runtime. (2) **Low parallelism of set operations**: Graph mining performs set operations between neighbors of vertices. There are two typical formats to represent neighbors in the graph, the Compressed Sparse Row (CSR) and the bitmap. Using CSR enables streaming read and write to the memory, but it is of low parallelism because neighbors of different vertices need a sequential comparison. The bitmap format exposes high parallelism because neighbors are aligned and the set operations can be expressed as bit vector logic operations. However, due to graph sparsity, it is impossible to deploy bitmaps to the graph mining problem because of large memory space and low memory utilization. (3) **Heavy Data Transfer**: Graph mining requires repetitive access to graph data. Table 2 shows the comparison between graph size and transferred data when processing a 3-CF (clique finding) problem. The data volume is measured by running GraphPi [29] using likwid-perfctr [38] with detailed setups in Sec. 6. As we can see, the transferred data volume is 26× to 451× larger than the original graph size, leading to heavy data transfer between CPU and memory.

To tackle all these challenges and accelerate graph mining problems, we propose DIMMining in this paper, which is based on a Dual-Inline Memory Module (DIMM)-based Near-Memory-Computing (NMC) architecture. Figure 1 shows challenges and contributions proposed in DIMMining, including:

- We introduce an index pre-comparison scheme to reduce the vertex index comparison for pruning during runtime. We use the self anchor to label neighbors with larger indices than itself for each vertex, and neighbor partitions to divide neighbors into ordered subsets. Then, indices can be pre-compared, and comparison operations can be reduced during runtime, achieving up to 2.01× speedup.
- We propose a flexible BCSR (bitmap with CSR) format to enable parallel set operations. The flexible BCSR format encodes continuous neighbors using a variable-length bitmap, which combines the advantages of set operation parallelism of bitmap and memory utilization of CSR. BCSR optimizes index encoding and saves up to 40.6% memory space over CSR. BCSR also achieves up to 2.30× higher parallelism over CSR, leading to an average of 1.25× speedup on CPUs.
- We propose the systolic merge array architecture to explore the parallelism for streaming set operations



(a) Problem Definition (Sec II.A)

```
for u0 ∈ V // we assign vi to ui for each i
  for u1 ∈ N(u0)
    for u2 ∈ N(u0) ∩ N(u1)
      for u3 ∈ N(u0) ∩ N(u1) - N(u2)
        // Find a valid pattern
        counter++;
```

(b) DFS-based Mining (Sec II.B)

Figure 2: Overview of the graph mining problem. (a) An example graph and the mining pattern. (b) DFS-based mining algorithm proposed in Automine [28].

on discontinuous indices. Compared with the conventional ordered queue and crossbar architectures, the systolic merge array achieves 7.00× and 7.37× higher throughput for set operations against ordered queue and crossbar, respectively.

- We propose DIMM-based Near-Memory-Computing architecture to alleviate heavy data transfer. Parallel set operations are offloaded to the memory side during runtime. Comprehensive experiments on real-world graphs show that DIMMining achieves **222.23**× and **139.51**× speedup compared with FPGAs and CPUs, and **3.62**× speedup against the state-of-the-art graph mining architecture.

2 PRELIMINARIES

2.1 Problem Definition

The graph mining problem is to find a given pattern $G = (V, E)$ in the input graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$. Figure 2(a) shows an example, where $V = \{v_0, v_1, v_2, v_3\}$ and $\mathbb{V} = \{⓪, ..., ⑦\}$ For the graph in cyan and the pattern in blue, there are three patterns in the graph, shown on the right. Typical graph mining problems include Clique Finding (CF), Motif Counting (MC), etc [28–30].

2.2 DFS-based Set-centric Graph Mining Model

Recent graph mining systems such as Automine [28], GraphPi [29], and GraphZero [30] adopt a DFS-based (Depth-first Search) model for mining problems, where they enumerate embeddings in one branch before others are explored. The model achieves less memory consumption and better performance compared with previous systems using a BFS-based (Breadth-first Search) model [39, 40]. A typical way to implement this model is using a set-centric model, where new vertices are calculated based on neighbor sets of explored vertices. Figure 2(b) shows an example of this DFS-based set-centric graph mining model. The pattern in Figure 2(a) is first modified into a complete pattern (Figure 2(b) left). The red edge represents there is no edge in the original pattern. Vertices in the pattern are explored from v_0 to v_3, and the search range of each vertex is based on the set intersection or subtraction results of previous vertices' neighbors. For the i-th `for` loop ($i > 1$) in the pseudo-code of Figure 2(b), the search range of vertex v_{i-1} is a subset of $\mathbb{V}$ (vertices in $\mathbb{G}$):

$$GetSet(u_0, ..., u_{i-2}) = \bigcap_{v_b \in black} N(u_b) - \bigcup_{v_r \in red} N(u_r) \quad (1)$$

In Equation (1), each pattern vertex v_j ($0 \le j < i-1$) is assigned to a certain graph vertex u_j in previous `for` loops. The right side of the equation traverses $v_0, ..., v_{i-2}$ and divides them into two sets, *black* and *red*. A v in the *black*/*red* set represents there is a black/red edge (Figure 2(b) left) between v_{i-1} and v. $N(u_j)$ outputs the neighbor set of u_j in the graph when v_j is assigned to u_j. Equation (1) generates a subset of $\mathbb{V}$ for v_{i-1} to traverse.

3 EFFICIENT PRUNING WITH INDEX PRE-COMPARISON

Previous researches propose pruning techniques to reduce the search space in graph mining, while involving massive comparison operations on vertex indices. We propose several index pre-comparison techniques to reduce runtime comparison overheads and enable efficient pruning for graph mining.

3.1 Pruning in Graph Mining

Pruning is a widely adopted optimization technique to reduce the search space in graph mining problems [29, 30]. The motivation of pruning is to find symmetry and isomorphism in the pattern and add constraints to vertex indices. Take the pattern in Figure 2(a) as an example, we find that v_0 and v_1 are symmetrical in the pattern (so are v_2 and v_3). Thus, when we find a valid pattern like $(v_0, v_1, v_2, v_3) = (①, ⑤, ②, ⑥)$, we will also find other 3 valid patterns, *i.e.*, $(⑤, ①, ②, ⑥)$, $(①, ⑤, ⑥, ②)$, $(⑤, ①, ⑥, ②)$, by permuting v_0 with v_1 and v_2 with v_3. Thus, to reduce the search space in Figure 2(b), previous designs introduce pruning by applying constraints on indices for symmetrical vertices in the pattern. In Figure 3(a), the constraints include $u_0 < u_1$ and $u_2 < u_3$ (recall that v_i is assigned to u_i). Then, only the pattern $(v_0, v_1, v_2, v_3) = (①, ⑤, ②, ⑥)$ is searched, while other three patterns can be directly induced.

Such a pruning technique requires comparison on vertex indices during runtime. According to the profiling results in Table 1, such a comparison operation accounts for up to 53% of total execution time in typical mining problems, which is inefficient when adopting the pruning technique.

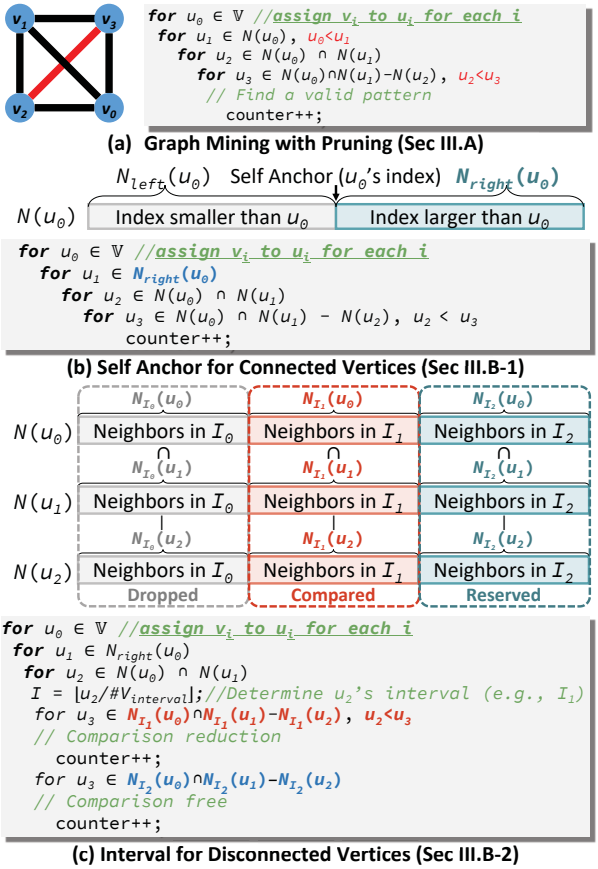


Figure 3: Efficient pruning with index pre-comparison. (a) Constraints on symmetry and isomorphism vertices in the pattern. (b) Using self anchor to eliminate index comparison for connected vertices in the pattern. (c) Using intervals to reduce comparison for disconnected vertices in the pattern.

3.2 Index Pre-comparison

To tackle inefficient pruning caused by index comparison during runtime, we point out that the constraints on vertex indices can be pre-computed. Thus, the comparison operations can be reduced during runtime.

3.2.1 Self Anchor for Vertices Connected in the Pattern. The main idea of the self anchor is that the index order constraint applied on two connected vertices will divide the neighbor set of a vertex into two disjoint sets. For example, we follow the constraint $u_0 < u_1$ in the second `for` loop in Figure 3(a). When u_1 traverses $N(u_0)$ in this loop, we can divide $N(u_0)$ into two disjoint sets, $N_{left}(u_0)$ and $N_{right}(u_0)$, where $N_{left}(u_0)$ contains all u_0's neighbor with smaller indices than u_0, and $N_{right}(u_0)$ contains all u_0's neighbor with larger indices than u_0 ($N_{left}(u_0) \cap N_{right}(u_0) = \emptyset$, $N_{left}(u_0) \cup N_{right}(u_0) = N(u_0)$). Then, u_1 only needs to traverse $N_{right}(u_0)$ (rather than whole $N(u_0)$) with no comparison during runtime, shown in Figure 3(b).

The self anchor technique works for all index comparisons applied on two vertices connected in the pattern. The neighbor set of each vertex is divided into two sets in advance, which can be easily applied to existing graph data formats. For example, we can store an additional self anchor pointer array to record the offset of the N_{right} set besides the row and column array in a common Compressed Sparse Row (CSR) format, causing $O(|\mathbb{V}|)$ memory overhead ($|V|$ is the number of vertices). Under this circumstance, the self anchor brings 15% storage overheads on average compared with original graphs in Table 1.

3.2.2 Neighbor Partitions for Vertices Disconnected in the Pattern. The self anchor is not always applicable because it cannot handle the index comparison of two vertices disconnected in the pattern, which contains two cases. Here we use two examples for demonstration:

- $u_3 \in N(u_0) \cap N(u_1), u_3 > u_2$: In this case, the neighbor set of the compared vertex (*i.e.*, u_2) does not participate in the set operations ($N(u_0) \cap N(u_1)$). The self anchor of u_0 (or u_1) divide the neighbor set according to its own index, which cannot be used to directly access the sub-neighbor set with indices larger than u_2.
- $u_3 \in N(u_0) - N(u_2), u_3 > u_2$: In this case, the neighbor set of the compared vertex (u_2) is used as the subtrahend. Even if we can get $N_{right}(u_2)$ for subtraction, we still need to filter out vertex indices larger than u_2 in the neighbor set of u_0, which requires index comparisons.

To tackle these scenarios, we introduce the idea of interval data structure to graph mining problems, which has been widely used in conventional graph traverse problems [18, 20, 23, 24]. The main idea of introducing the interval is to divide neighbor sets into several partitions, and the set operation can be applied to partitions in a certain range rather than applied on the whole set. Here we take the fourth `for` loop in Figure 3(a) as an example. Vertices are divided into three intervals, I_0, I_1, and I_2. Thus neighbors of each vertex are also divided into three subsets (*e.g.*, $N(u_0)$ is divided into $N_{I_0}(u_0)$, $N_{I_1}(u_0)$, and $N_{I_2}(u_0)$, where N_{I_i} represents neighbors in the i-th interval). Then, we can modify the range of u_3 in the fourth `for` loop according to:

$$\begin{aligned} & N(u_0) \cap N(u_1) - N(u_2) \\ =& [N_{I_0}(u_0) \cap N_{I_0}(u_1) - N_{I_0}(u_2)] (\in I_0) \\ \cup& [N_{I_1}(u_0) \cap N_{I_1}(u_1) - N_{I_1}(u_2)] (\in I_1) \\ \cup& [N_{I_2}(u_0) \cap N_{I_2}(u_1) - N_{I_2}(u_2)] (\in I_2) \end{aligned} \tag{2}$$

Equation (2) can be derived from Equation (1).

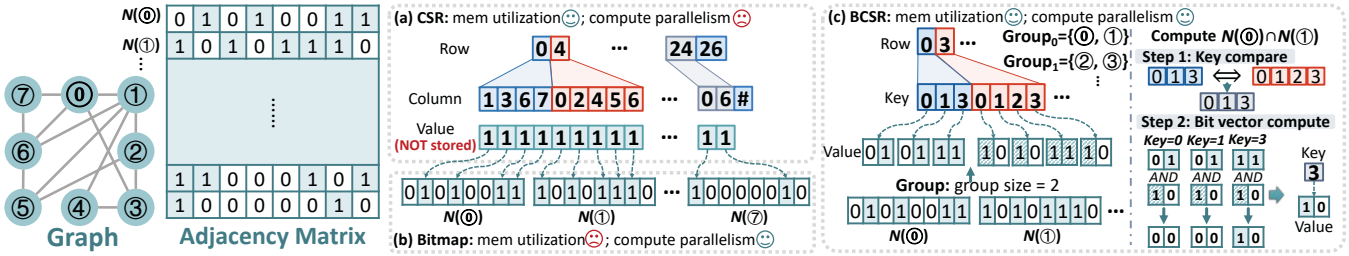


Figure 4: Different formats to represent neighbors of the graph in Figure 2. (a) CSR: neighbors are stored using the index in the column array. (b) Bitmap: all vertices are encoded with one bit. (c) Flexible BCSR (bitmap+CSR): vertices are divided into groups, neighbors of a vertex are encoded with the group index using a key array (similar to the column array in CSR), attached with a value array representing the bitmap in the group.

Proof. For $\forall i \neq j$, and $\forall u, u' \in \mathbb{V}$, we have $N_{I_i}(u) \subset I_i$ and $N_{I_j}(u') \subset I_j$, thus $N_{I_i}(u) \cap N_{I_j}(u') = \emptyset$. Thus, when vertices are divided into P intervals, Equation (1) can be modified into:

$$
\begin{aligned}
&\bigcap_{v_b \in black} N(u_b) - \bigcup_{v_r \in red} N(u_r) \\
&= \bigcap_{v_b \in black} \left(\bigcup_{i<P} N_{I_i}(u_b) \right) - \bigcup_{v_r \in red} \left(\bigcup_{i<P} N_{I_i}(u_r) \right) \\
&= \left[\bigcup_{i<P} \left(\bigcap_{v_b \in black} N_{I_i}(u_b) \right) \right] \bigcup \\
&\quad \left[\bigcup_{\forall i,j<P, i \neq j} \left(\bigcap_{v_b, v'_b \in black} (N_{I_i}(u_b) \bigcap N_{I_j}(u'_b)) \right) \right] - \bigcup_{i<P} \left(\bigcup_{v_r \in red} N_{I_i}(u_r) \right) \\
&= \bigcup_{i<P} [\bigcap_{v_b \in black} N_{I_i}(u_b)] - \bigcup_{i<P} [\bigcup_{v_r \in red} N_{I_i}(u_r)] \\
&= \bigcup_{i<P} [\bigcap_{v_b \in black} N_{I_i}(u_b) - \bigcup_{v_r \in red} N_{I_i}(u_r)]
\end{aligned}
\tag{3}
$$

Thus, we get Equation (2), where the set operation can be performed on each subset individually. □

We assume that $u_2 \in I_1$ (v_2 is assigned to u_2) in the third `for` loop. According to the constraint $u_2 < u_3$, the results in the second line of Equation (2) can be dropped ($\forall u \in I_0$, we have $u_2 > u$, which does not satisfy the constraint), and all results in the fourth line will be reserved ($\forall u \in I_2$, we have $u_2 < u$, which meets the constraint). Only the results in the third line require an index comparison between u_2 and u_3. In this way, the number of comparison operations is bounded by the size of an interval rather than the whole neighbor set, shown in Figure 3(c). Similar to the self anchor technique, we only need to store another P-1 pointers to distinguish neighbors of each vertex into P intervals. Thus the storage overheads are still $O(|\mathbb{V}|)$.

To reduce the storage overhead of neighbor partitions, we further analyze the characteristics of memory access during graph mining. Due to the power-law distribution[19] of vertex degree (*i.e.,* the number of neighbors), we observe that the top 10% largest neighbor set takes up 44% of the neighbor set access in 3-MC mining problems. Thus, we only apply the neighbor partitions to the vertices with largest degrees. For graphs in Table 2, applying neighbor partition onto 10% vertices only poses 12% additional memory compared to the graph size.

4 PARALLEL SET OPERATION IN DIMMINING

There are two main set operations, intersection (**INT**) and subtraction (**DIFF**), on two neighbor sets A and B in graph mining. To explore parallelism in these set operations, we introduce the flexible BCSR data format from the data structure perspective in Sec. 4.1, and the Systolic Merge Array (SMA) from the hardware perspective in Sec. 4.2.

4.1 Flexible BCSR Format

The CSR and bitmap are two major formats to represent a graph. Figure 4(a) and (b) show examples of these two formats. In this example, because the graph contains eight vertices, the bitmap format uses 8-bit for each vertex to represent its neighbors. The CSR format stores indices of neighbors for each vertex using a column array, which is indexed by a row array. The set operations on the bitmap format are of high parallelism, because the neighbor with the same index is aligned and can be executed using bitwise operations. However, due to the sparsity of graphs, the bitmap format involves many "0"s and leads to inefficient memory utilization. In contrast, the CSR format stores neighbors in a compact way, leading to efficient memory utilization. But, the parallelism of the CSR format is low because the neighbors of different vertices are not aligned and need sequential comparisons for set operations.

Neither bitmap nor CSR achieves both high memory utilization and set operation parallelism. To benefit from the advantages of both formats, we propose our flexible BCSR (bitmap + CSR) format in Figure 4(c). In the flexible BCSR format, continuous vertices are first combined into a group. For

Table 3: Memory space (Byte) comparison on different formats, (a+b) means (key bit + value bit)

Format / Graph, $\mathbb{G}$	Bitmap	CSR	BSR [41]	Flexible BCSR				Optimal	Optimal BCSR
	0+|V|	32+0	32+32	30+2	28+4	24+8	16+16	BCSR	/CSR
PP	8.23E+06	2.41E+05	4.12E+05	2.28E+05	2.21E+05	2.15E+05	**2.11E+05**	**2.11E+05**	87.7%
AS	4.40E+07	1.66E+06	1.77E+06	1.46E+06	1.28E+06	1.12E+06	**9.86E+05**	**9.86E+05**	59.4%
MI	1.17E+09	9.03E+06	1.39E+07	8.35E+06	7.81E+06	7.43E+06	**7.15E+06**	**7.15E+06**	79.2%
PA	1.78E+12	1.47E+08	2.79E+08	1.44E+08	1.42E+08	**1.41E+08**	-	**1.41E+08**	95.7%
YT	6.30E+12	4.85E+08	8.18E+08	4.47E+08	4.24E+08	**4.07E+08**	-	**4.07E+08**	84.0%
LJ	2.00E+12	2.93E+08	4.47E+08	2.72E+08	2.56E+08	**2.43E+08**	-	**2.43E+08**	82.9%

example, two vertices with continuous indices form a group in Figure 4(c). Then, the column array in the CSR format turns into a **key** array, and each key in the array represents a non-zero group index. The key array is also indexed by a row array like CSR. Because there are multiple vertices in a group, each element in the key array is attached with a bitmap, representing whether a vertex in this group is a neighbor of the source vertex. These bitmaps form a **value** array in BCSR.

In our BCSR format, neighbors of a vertex are represented with several ⟨Key, Value⟩ Pairs (*KVP*s). In these *KVP*s, the key represents a group index, and the value represents a bitmap. The flow for the set operation on the BCSR format can be described as a combined flow of bitmap and CSR. To perform a set operation between neighbors of two vertices using the flexible BCSR format, we first compare keys to get groups with common indices. Then, the values of these groups are processed according to the same rules on the bitmap format. An example is shown on the right of Figure 4(c). To calculate $N(⓪) \cap N(①)$, common indices (0, 1, 3) are got using a similar way in the CSR format, and then a bitwise AND operation is applied to values attached to these groups. The final result consists of a group with key (3), and the corresponding value "10", representing $N(⓪) \cap N(①) = \{⑥\}$. $Note: vertex\ index = key \times group\ size + position\ of$ "1" $in\ value = 3 \times 2 + 0 = 6$.

The flexibility of the BCSR format is reflected in the adjustable group size of the value array (*i.e.,* bitmap width). Assuming we use a bits to store a key, and b bits to store a value, we can store a graph with maximum $2^a \times b$ vertices. The flexible BCSR format is a unified representation for both bitmap and CSR by adjusting the value of a and b. In the bitmap format, we have $a = 0$, and $b = |\mathbb{V}|$ ($|\mathbb{V}|$ is the number of vertices). In the CSR format, we have $a = \lceil log_2(|\mathbb{V}|) \rceil$, and $b = 1$. The flexible BCSR format inherits the advantages of high parallelism in bitmap and high memory utilization in CSR. In our implementation, we set $a + b = 32$ for DRAM alignment purpose with several typical BCSR format configurations of $(a, b) = (16, 16), (24, 8), (28, 4)$, and $(30, 2)$. It is obvious that a larger b leads to higher parallelism of set operations. Therefore we choose the largest b under the constraint of $2^a \times b \geq |\mathbb{V}|$ when storing a graph using BCSR.

Table 3 shows memory space on different BCSR configurations and other graph data formats. In CSR, each 32-bit column represents one neighbor. While in BCSR, each 32-bit *KVP* represents $1 \sim b$ neighbors, showing better memory efficiency. As shown in Table 3, BCSR can save 4.3% to 40.6% memory space compared with CSR. Moreover, our format saves 1 to 4 orders of magnitudes on the memory space compared with the bitmap format, making it practical to use bitwise operations in graph mining. Furthermore, BCSR achieves 1.01× to 2.30× higher computing parallelism because BCSR exploits the parallelism of bitwise operation in its value part.

4.2 Systolic Merge Array

The key array in the flexible BCSR format (column array in CSR) stores neighbors with discontinuous indices, and we design the **Systolic Merge Array (SMA)** to compute set operations between two key arrays in parallel. Two *KVP*s have non-empty intersections only when their keys are equal and the bitwise AND of their values are not all zeros. Thus, SMA needs to (1) pick out all pairs of *KVP*s with matched keys from the two *KVP* lists; (2) compute bitwise AND of values in parallel for each pair of *KVP*s with the matched key; (3) filter out *KVP* in the result whose value is all zeros. Because $A - B = A \cap \bar{B}$ and the duality between $\cap$ and $\cup$, we use **INT** as the example in this section, while **DIFF** can be derived in a similar way.

Before introducing SMA, we first consider a straightforward queue-based merging scheme that finds *KVP*s with matched keys in sequence. As shown in Figure 5(a), we write multiple *KVP*s into two shift registers (ordered queue) in the ascending order of the key. Then, in each merging step, we compare the keys of the two queues' heads. If one key is smaller than the other, the *KVP* with smaller key will be popped from its queue, and other *KVP*s in this queue will move rightward. When two keys match with each other, both two pairs will be popped, and the merging unit will calculate the bitmap merging results. Two queues shift to the right and receive new *KVP* at the same time. The ordered queue has the advantage of lightweight hardware (*i.e.,* one comparator, one merging unit, and two register-based ordered

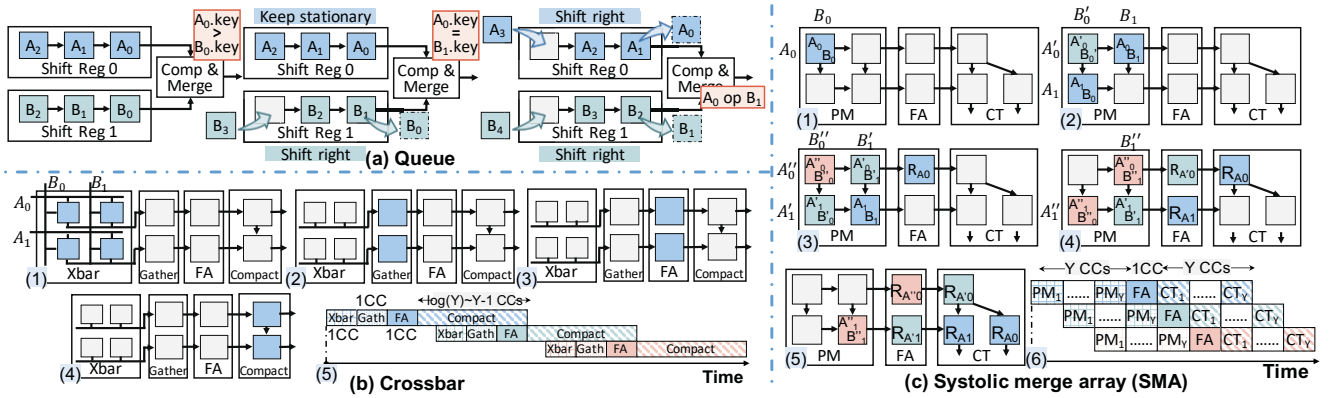


Figure 5: Different hardware implementation for BCSR-based set operation: (a) ordered queue based implementation; (b) crossbar-based implementation; (c) SMA. *PM, FA, CT*, **and** *CC* **mean** *processing matrix*, *filter array*, *compaction triangle*, **and** *clock cycle*. Y **represents the input vector size.** $A_{0,1,2,3}$, $B_{0,1,2,3}$, $A''_{0,1}$, **and** $B''_{0,1}$ **are different KVPs.**

queues). However, it suffers from low throughput with only one *KVP* output per cycle at most.

To improve the throughput, a way is to use a crossbar matrix for parallel comparison between keys, at the cost of additional hardware resources. In the crossbar matrix, the two lists of *KVP*s are split into multiple vectors ($A^{(0)}, A^{(1)}, \ldots$ and $B^{(0)}, B^{(1)}, \ldots$) that each contains Y *KVP*s (*e.g.*, $A^{(0)} = \{A_0^{(0)}, A_1^{(0)}, \ldots, A_{Y-1}^{(0)}\}$), and we use a $Y \times Y$ crossbar to perform an all-to-all comparison for the keys in two vectors in parallel. Ideally, such an implementation achieves the throughput of Y *KVP* outputs per cycle. However, two new problems will emerge after dividing *KVP*s into vectors:

(1) ***Problem 1:*** Y keys from one vector $A^{(n)}$ may be found in multiple vectors $B^{(m)}, \ldots, B^{(m+k)}$. Each matrix comparison only gets a partial result (*e.g.*, $A^{(n)} \cap B^{(m+i)}$) instead of full result (*e.g.*, $A^{(n)} \cap (B^{(m)} \cup \ldots \cup B^{(m+k)})$). We need to calculate all these partial results and combine them together to get the full result. For example, $A^{(0)}.key = [1, 6]$, $B^{(0)}.key = [1, 4]$, and $B^{(1)}.key = [5, 6]$. We need to calculate $[1, 6] \cap ([1, 4] \cup [5, 6])$.

Solution 1: We use a simple "compare-and-advance" scheme. Suppose we send two vectors $A^{(i)}$ and $B^{(j)}$ to the processing matrix in the first cycle, we compare the last keys of two input vectors $A^{(i)}$ and $B^{(j)}$. If $A^{(i)}_{Y-1}.key > B^{(j)}_{Y-1}.key$, we keep $A^{(i)}$ unchanged and input $B^{(j+1)}$ in the second cycle. Otherwise, $A^{(i)}$ is replaced by $A^{(i+1)}$ and $B^{(j)}$ keeps unchanged. After getting all the partial results for $A^{(n)}$ (when the $A^{(n)}$ is replaced), they are combined according to following rules:

$$A^{(n)} \cap (B^{(m)} \cup \ldots \cup B^{(m+k)}) = (A^{(n)} \cap B^{(m)}) \cup \ldots \cup (A^{(n)} \cap B^{(m+k)})$$
$$A^{(n)} - (B^{(m)} \cup \ldots \cup B^{(m+k)}) = (A^{(n)} - B^{(m)}) \cap \ldots \cap (A^{(n)} - B^{(m+k)}) \quad (4)$$

(2) ***Problem 2:*** The hardware matrix generates Y output *KVP*s at the same time, but some pairs may be invalid (*e.g.*, key unmatched or the value field of output *KVP* is zero), making the result vector becomes sparse.

Solution 2: We attach a compaction module after the crossbar matrix. The module removes the invalid *KVP* in the output vector, arranging valid elements one by one.

A naïve crossbar-based implementation is shown in Figure 5 (b). Two input vectors (*e.g.*, $A^{(n)}$ and $B^{(m)}$) are fed into each row and column of the crossbar, respectively. All processing units at the crossbar intersections perform the BCSR-based set operation in parallel and generate $Y \times Y$ output *KVP*s in one cycle. Only at most one out of Y *KVP* outputs is valid for each row. Therefore, we send the results of crossbar to a gather array to choose the valid output of each row as the partial result (*i.e.*, $A^{(n)} \cap B^{(m)}$). After that, the partial result is fed into a filter array (FA) to combine with other partial results following Equation (4). Then, the full results generated by FA are sent to the compaction array to remove the invalid *KVP*, as discussed in ***Solution 2***. Theoretically, except for the compaction array, other modules all take one clock cycle to process one input *KVP* vector. Since the crossbar outputs Y *KVP*s at a time, the compaction array needs $\log(Y) \sim (Y-1)$ cycles to rearrange the valid *KVP* in the output vector depending on the specific hardware implementation. Therefore, when using the crossbar-based implementation in a pipeline manner, each pipeline stage takes $\log(Y) \sim (Y-1)$ cycles, achieving the throughput of $Y/(Y-1) \sim Y/(\log(Y)-1)$ *KVP*s per cycle.

Because the crossbar-based implementation needs to choose one valid output from Y outputs in each crossbar row, the crossbar size is limited by realistic circuits constraints (*e.g.*, fan-in of the gather unit). Moreover, the lower bound of the pipeline stage is $\log(Y)$ due to the compaction array. To tackle these two problems, we propose a fully-pipelined, high throughput, and easy-to-scale crossbar-based implementation, Systolic Merge Array (SMA). SMA consists of three main components: the processing matrix (PM), the filter array (FA), and the compaction triangle (CT), which are depicted in Figure 5(c). As the name implies, SMA works like

Table 4: Hardware complexity/throughput comparison

	SMA	Queue	Crossbar
Storage Hardware	$O(Y^2)$	$O(Y)$	$O(Y^2)$
Compute Hardware	$O(Y^2)$	$O(1)$	$O(Y^2)$
Throughput (pairs/cycle)	Y	1/4∼1	$\frac{Y}{Y-1} \sim \frac{Y}{logY-1}$

a traditional systolic array. The PM reads two vectors (A and B) in each cycle and transfers the results to the right. We also need a filter array and a compaction module to generate and compact the final full results. An example is shown in Figure 5(a-1∼a-5). It shows the procedure of merging three vector pairs: $\{A \text{ op } B\}$ (in blue), $\{A' \text{ op } B'\}$ (in green), and $\{A'' \text{ op } B''\}$ (in orange).

Different from the crossbar structure, only the first row and column receive the input data, and each processing element (PE) in PM sends its upper input to the lower neighbor PE, and left input together with output to its right neighbor PE, which guarantees the scalability of SMA. Besides, all the input vector elements flow into the PM in a "space-time-offset" manner, *e.g.,* for the n^{th} input vector, the i^{th} element appears in row i at cycle $(n + i)$. In this manner, the element of A takes Y cycles to flow from left to right and performs computations with all Y elements of B. We also split the compaction phase into multiple fully pipelined stages, *i.e.,* takes Y one-clock-cycle stages for compaction instead of one $\log(Y)$-clock-cycle stage. Thus, we can improve the throughput to Y within the same area, as shown in Figure 5(c-6). The hardware implementations of SMA will be discussed in Sec. 5. Table 4 summarizes all three hardware implementations mentioned in this section. Considering the simple calculations in BCSR merging operations (*i.e.,* comparisons and bitwise operations), a little larger hardware complexity is still acceptable. Therefore, from the performance perspective, we choose the SMA with the highest throughput as our hardware design.

5 DIMMINING ARCHITECTURE

To tackle huge data movements in graph mining, we propose NMC-based DIMMining. DIMMining is built with rank-level NMC processors in Load-Reduced DIMM (LRDIMM) without modifying the design of DRAM devices, which fully exploits the large internal bandwidth and ensures the rank-level computing parallelism by avoiding the DIMM bus contention.

5.1 Architecture Overview

The overall architecture of DIMMining is shown in Figure 6(a). Compared with traditional storage-oriented LRDIMM, we modify the Registering Clock Driver (RCD) for NMC instruction decoding and computation-oriented chip selection (CS), and add two DIMMining NMC modules in each rank for performing BCSR-based vertex merge operations. In order to realize NMC on the premise of retaining the original memory function, DIMMining supports two work modes, *i.e.,* memory mode and NMC mode. In the memory mode, NMC modules are disabled and bypassed, and the DRAM chips are connected with Data Buffers (DBs) directly as in the traditional LRDIMM. In the NMC mode, one rank is split into two sub-ranks, each of which is attached to one DIMMining NMC module and provides 32-byte data to NMC modules independently. The DIMMining NMC module receives/sends the input/output BCSR data from/to DRAM chips within the same rank directly or other ranks through DBs. The scalability of DIMMining can be achieved by connecting the DB buses of different ranks or realizing inter-DIMM communications through the standard DDR channel.

DRAM chips (Figure 6(b)). At the DRAM chip level, we only restrict the usage rules of each memory bank without modifying the internal circuit design of the DRAM device, which ensures the applicability of DIMMining design to different DRAM devices. We split each DRAM chip into three regions for storing the *KVP* data for each vertex in BCSR, the starting address to read corresponding *KVP* data for each vertex, and intermediate data, respectively. Since the neighbor number of vertices varies greatly, the *KVP* number is different for different vertices. Therefore, we store the starting memory address of BCSR data for each vertex in the address region to realize address-index-based fast access. In each sub-rank, all *KVP*s in one BCSR vector are evenly stored in multiple DRAM chips to improve the readout parallelism.

Modified RCD (Figure 6(c)). In LRDIMMs, RCD is used for driving the command/address (C/A) signals from the memory controller to its DRAM chips. Based on this, we add an NMC instruction buffer & decoder and an address generator in RCD to strengthen the C/A signal driving in the NMC mode. The buffer & decoder receives NMC instructions and sends key instruction parameters (will be discussed in Sec. 5.2) to the address generator, which generates the address for NMC modules within the same rank.

DIMMining NMC (Figure 6(d)). In DIMMining, two NMC modules are set in each rank, which contains the data forwarding unit, the mining processor, and an NMC controller (NMC Ctrl). The NMC Ctrl is responsible for managing other NMC modules, configuring mining processor functions according to the operation queue, and accessing the next vertex data according to the address. The data forwarding unit (Figure 6(e)) forwards the BCSR data from DRAM chips, DBs, or cache to the input queue of the mining processor. Because of the power-law distribution of vertex degree, a few vertices with much more neighbors are accessed more frequently during the mining, showing a certain degree of spatial locality. Therefore, we set an SRAM-based cache in the data forwarding unit to store intermediate data and neighbor information of frequently used vertices. The mining processor

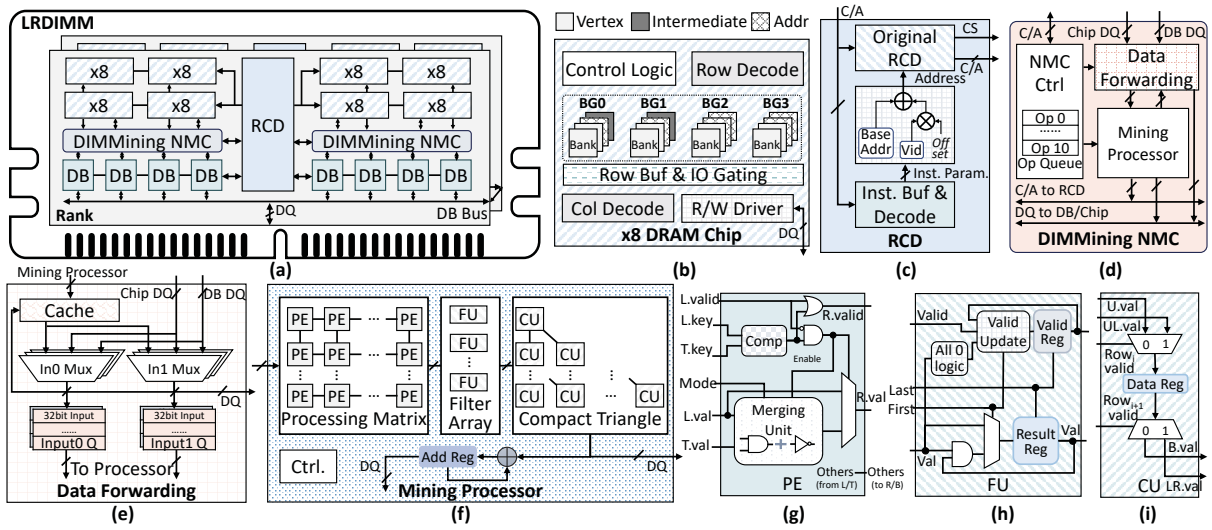


Figure 6: (a) Architecture of DIMMining; (b) DRAM chip; (c) Modified Registering Clock Driver (RCD); (d) DIMMining NMC module; (e) Data forwarding unit in DIMMining NMC; (f) Mining processor design with (g) Processing elements (PE), (h) Filter units (FU), and (i) Compaction units (CU). In this figure, L, R, T, D, UL, **and** LR **mean** $left$, $right$, $upper$, $lower$, $upper\ left$, **and** $lower\ right$.

(Figure 6(f)) performs the BCSR-based vertex merging operation, which is the hardware realization of SMA. The mining processor is composed of four parts: processing matrix (PM), filter array (FA), compaction triangle (CT), and counting & addition (CA) module.

Processing matrix (PM). The PM contains $Y \times Y$ connected **processing elements** (PE, Figure 6(g)) when processing BCSR vectors with the length of Y. The PE executes three functions: (1) Input transfer: each PE transfers the input data from the left (upper) to its right (lower) PE neighbor; (2) Valid signal transfer: each PE transfers the output result valid signal to its right PE neighbor. The result valid signal indicates whether the two input keys match; (3) Vertex merging: if the input valid signal is "1", the vertex merging operation is skipped, and PE transfers the merging results from left to right. Otherwise, PE compares two keys and performs bitmap-based set operations when matched.

Filter array (FA). FA filters the invalid data and combines all partial results in the "compare-and-advance" scheme. It contains Y **filtering units** (FU, Figure 6(h)), and each FU is connected to the last PE of the same row in PM. Compared with PE, FU has two additional control inputs: *First* and *Last*. *First* indicates whether the input vector appears for the first time (*e.g.,* $A^{(n)}$ appears for the first time when calculating with $B^{(m)}$ in Equation (4)). If so, the result/valid registers are initialized according to PE outputs in the same row. Similarly, *Last* represents whether the input vector is the last occurrence (*e.g.,* $A^{(n)}$ appears for the last time when calculating with $B^{(m+k)}$ in Equation (4)). If so, the FU has combined all partial results and will send the result to CT. Otherwise, the result/valid registers are updated according to the historical merging results and current PE outputs.

Compaction triangle (CT). The CT aims to skip the invalid merging results (*e.g.,* mismatched keys or result value is all-zero) in the output vector and make the vector more compact. To cooperate with PM and sustain the high throughput, multiple **compaction units** (CU, Figure 6(i)) are connected to a triangular array with the data flow from left to bottom. For each row, if the FU in this row outputs a valid *KVP*, the leftmost CU receives this valid *KVP*, and other CUs receive data from its upper-left CU neighbor. Otherwise, all CUs receive data from their upper CU neighbor. Finally, the compact output vector is obtained in the CT lower row. According to whether the current computation is the last vertex in the pattern to be mined, the CT results will be sent to the CA module or written back to the cache. The CA module consists of a counter register and an adder that record the inquired pattern number.

5.2 DIMMining Instruction Design

Figure 7 illustrates the DIMMining instruction design, which is designed to be compatible with conventional memory functions and reduce the instruction transmission costs during graph mining. The mode code manages the work mode of DIMMining and determines the following field formats.

In the memory mode, all DRAM chips in a rank share the same C/A address and provide 64-byte data in lockstep. The instruction design is similar to the original storage-oriented LRDIMM, which is decoded to standard DDR instructions by the memory controller. The typical memory address format is depicted in Figure 7(a).

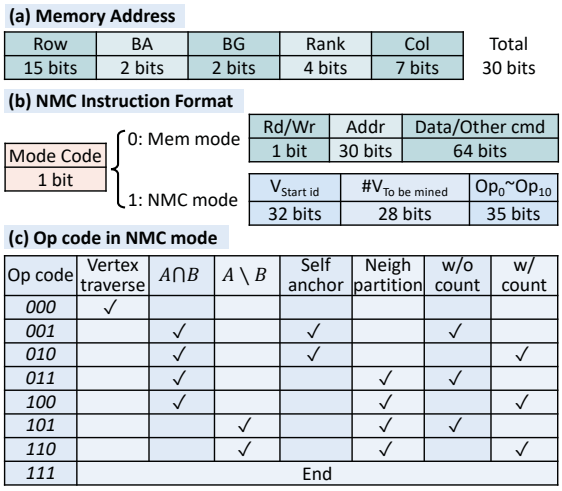


Figure 7: (a) Original LRDIMM memory address format; (b) DIMMining instruction format; (c) Operation code in instruction format.

In the NMC mode, the NMC modules in each rank need to be aware of the vertex data address and specific operations. However, because of the massive transferred data and mining computations, transferring memory address and operation code vertex-by-vertex for each rank brings a heavy bandwidth burden to C/A bus. To handle this problem, we propose the macro NMC instruction format for DIMMining, as shown in Figure 7(b). Instead of sending instructions vertex-by-vertex, the proposed macro NMC instruction format splits and distributes the entire mining workload to different ranks at one time. We split the mining workload according to the index of the start vertex during the mining (*i.e.,* the vertex in the first `for` loop). Then these ranks decode the instruction and generate control signals by themselves, without the need for host-side control. The NMC instruction format mainly contains three fields:

(1) **Start vertex index** indicates the index of the first start vertex for this rank. For example, when restricting $Rank_0$ to mining under the premise that the first vertex range is {⓪, ⑯, ㉜}, this field for $Rank_0$ is 0.
(2) **Number of start vertex to be mined** represents the execution times of the first `for` loop in this rank. It also reflects the workload of this rank. In the former example, this field for $Rank_0$ is 3. It should be noted that in order to realize workload balance, the first vertices assigned to each rank are discontinuous. The first vertex index is accumulated in steps of the total number of ranks (*i.e.,* 16 ranks in the former example).
(3) **Operation code sequence** represents the execution sequence of the `for` loop, detailed in Figure 7. For the example shown in Figure 3, the operation code sequence can be expressed as follows.

$$000 - 000 - 001 - 000 - 110 - 111$$

6 EXPERIMENTAL RESULTS

6.1 Experimental Setup

6.1.1 DIMMining setup. Table 5 shows the configurations. We setup the simulation framework with: (1) We design a behavior level BCSR-based graph mining simulator, which also generates the physical memory access trace. (2) We build a trace-based cache simulator to simulate the cache behavior in DIMMining. We set the associativity to four and leverage LRU cache replacement policy. The latency, area, and power data of the cache are derived from CACTI [42] under 32nm technology node. (3) We use Ramulator [43] to generate the cycle-level evaluation results of each DDR4 rank. DRAM energy data are given by DRAMPower [44]. (4) For the NMC module, we estimate the latency, area, and power using Synopsys Design Compiler with TSMC 65nm technology library at the frequency of 500MHz. We scale these performance results to 32nm technology node according to [45]. We combine the above four components and one workload distribution simulator to estimate the end-to-end performance of DIMMining.

6.1.2 Baseline. We select following designs for comparison:

- **Gramer** [33] is a graph mining accelerator implemented on an XCU250-2LFIGD2104E FPGA chip with four 16GB DDR4 memories. The FPGA chip provides 1.68M LUTs, 3.37M registers, and 11.8MB BRAMs. We use the results reported in its paper.
- **GraphPi** [29] is a graph mining system on CPUs. The evaluation is carried out on a CPU machine with 2 Intel(R) Xeon(R) Silver 4210 CPUs running at 2.2GHz with 13.75MB LLC and 64GB DDR4 memory. We run GraphPi with the source code [46] using 24 threads.
- **FlexMiner** [32] is a graph mining accelerator. The processing elements (PEs) in FlexMiner run at 1.3GHz, with 32kB private cache and 4MB shared cache, using 15nm process. The `c-map` enabled FlexMiner data are provided by its authors, and we scale the results to 32 PEs configuration (same PE number as DIMMining while consuming more area/energy compared with DIMMining).

6.1.3 Datasets. We carry out our evaluation on six real-world graphs, shown in Table 1 with detailed information of each graph (We use the same YT graph used in FlexMiner [32], which is a superset of that used in Gramer [33] ($|V|$=4.6M, $|E|$=44.0M)). We select these graphs for performance comparison because they are also used in previous graph mining systems and architecture designs. The BCSR format can compress the memory space of these graphs to less than 1GB (Table 3). So we duplicate and store the entire graph in each sub-rank, avoiding the graph data transmissions among

Table 5: DIMMining configurations

Memory configurations	
Memory capacity	64GB
Channels/Rank per channel	2/8
NMC per rank	2
DDR4 DRAM parameters	
DDR4 configurations	4Gb x8 2400R
Clock frequency	1200MHz
tRCD-tCAS-tRP	16-16-16
NMC module configurations	
Technology node	32nm
Size of Systolic Merge Array	8
Cache size	128KB per NMC

different ranks. We also show the comparison of splitting graphs into partitions in the latter experiments.

6.2 Overall Performance

We compare the performance of DIMMining with three baseline designs under the configuration introduced in Sec. 6.1. The comparison results are shown in Table 6. Because the speedup is the ratio between the execution time of two designs, we use the geometric mean metric. DIMMining achieves an average of **222.23**× and **139.51**× speedup against Gramer [33] on FPGAs and GraphPi [29] on CPUs. By utilizing optimizations proposed in this paper, DIMMining can also achieve an average of **3.62**× speedup against the state-of-the-art graph mining architecture FlexMiner with 32 PEs [32].

6.3 Ablation Study of DIMMining Designs

6.3.1 Benefit of Index Pre-comparison. By using our Pre-comparison techniques, comparison for pruning is eliminated for 3,4,5-CF and reduced for 3-MC. We add a self anchor and apply neighbor partition on 10% vertices for all graphs. Figure 8 shows the speedup over naïve pruning. Index pre-comparison on 5-CF achieves 1.46× speedup over naïve pruning for CSR, and 1.65× for BCSR. Besides, we also calculate the theoretical speedup bounds, *i.e.,* the speedup when all the comparisons are eliminated. Most of the CF experiments show the speedup closed to the theoretical bound, revealing that our techniques successfully eliminate comparison overhead for connected vertices. For 3-MC, though only 10% of large-degree vertices store neighbor partition, they take up 44% of neighbor set access on average. Respective percentage of comparison is completely ignored to give the speedup bound. Actual speedups proportionally align with the bound, showing effective comparison reduction for disconnected vertices.

6.3.2 Benefit of Flexible BCSR Format. We compare runtime performance of BCSR with conventional CSR on 4-CF task. Figure 9(a) and (b) profile hardware-irrelevant algorithmic Performance. By offering more information within a

Table 6: Graph mining systems and architectures comparison (/seconds)

Pattern	$\mathbb{G}$	Gramer	GraphPi	FlexMiner	DIMMining
3-CF (3-clique)	PP	0.010	0.098	-	**1.594E-05**
	AS	0.028	0.093	0.0005	**2.564E-04**
	MI	0.110	0.150	0.0023	**1.574E-03**
	PA	3.090	0.640	0.0784	**1.476E-02**
	YT	13.010	1.605	0.1862	**1.159E-02**
	LJ	17.810	2.320	0.2096	**6.899E-02**
4-CF (4-clique)	PP	0.011	0.065	-	**1.708E-05**
	AS	0.270	0.130	0.0021	**1.507E-03**
	MI	6.860	1.170	0.0343	**2.224E-02**
	PA	3.740	0.660	0.0946	**1.798E-02**
	YT	17.300	5.490	0.3213	**1.541E-02**
	LJ	30.890	15.810	-	**4.376E-01**
5-CF (5-clique)	PP	0.012	0.067	-	**1.717E-05**
	AS	1.460	0.360	0.0139	**1.011E-02**
	MI	274.410	44.490	-	**7.764E-01**
	PA	4.060	0.690	0.5271	**1.936E-02**
	YT	16.250	6.160	-	**2.880E-02**
	LJ	29.680	535.210	-	**1.347E+01**
3-MC (3-motif)	PP	0.033	0.2	-	**4.744E-05**
	AS	0.110	0.190	0.0011	**9.076E-04**
	MI	0.360	0.310	0.0081	**6.181E-03**
	PA	4.170	1.280	0.2311	**4.791E-02**
	YT	16.250	2.460	0.5473	**1.162E-01**
	LJ	29.680	4.640	-	**3.288E-01**

4 Byte computation, BSCR reduces the total operations by 42.49%. Data load and store requests are reduced by 43.40% and 28.44%, respectively. However, on current CPUs which usually take 32-bit operand, we need to treat a BCSR's key and value as two operands. Despite such drawbacks, BCSR still shows a 1.25× speedup over CSR (Figure 9(c)). Our SMA design can further harvest BCSR's efficiency and parallelism by enforcing multiple operations on different bits at the same time. Additionally, BCSR saves 16.57% memory on average (Figure 9(d)), which can be used to compensate for the additional overhead of index pre-comparison. So BCSR can benefit from both high parallelism and efficient pruning with approximately the same memory overhead as CSR.

6.3.3 Benefit of Systolic Merge Array. DIMMining proposes a novel SMA to perform set operations on two CSR format arrays in parallel. Figure 10 shows the comparison between SMA and conventional ordered queue and crossbar. Our SMA

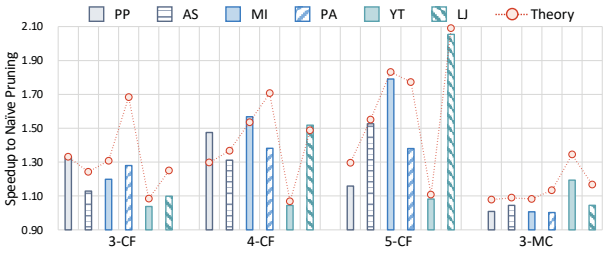


Figure 8: Index Pre-comparison Speedup over Naive Pruning on CSR, measured on CPUs

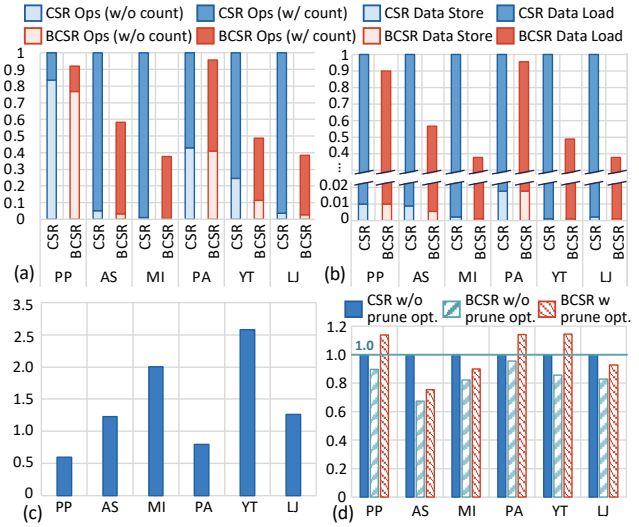


Figure 9: (a) Operation counts and (b) Data request volume of BCSR (normalized to CSR). (c) CPU runtime of BCSR normalized to CSR with naïve pruning. (d) Memory overhead normalized to CSR on 4-CF.

achieves an average of 7.37× and 7.00× speedup against ordered queue and crossbar, while the end-to-end DIMMining mining speedup is 3.91× and 3.75× on average.

6.3.4 Benefit of Near Memory Computing. We also analyze the performance gain brought by the thought of near memory computing. For a fair comparison, we propose a baseline design w/o using NMC which has the same PE number and memory configuration as DIMMining. The only difference is that the baseline w/o NMC needs data communications between PEs and off-chip DRAM, and the off-chip memory bandwidth is assumed to be 38.4GB/s (19.2GB/s per channel). The results are shown in Figure 11. Compared with the baseline design w/o NMC, DIMMining can achieve 1.21× to 5.39× and 3.98× to 5.35× speedup on different mining patterns for LJ and PP, respectively. The speedup comes from two aspects: (1) in NMC design, all computations happen in each rank, eliminating the data movements between PEs and off-chip memory. And memory ranks do not need to compete with others for DDR bandwidth. (2) DIMMining splits the rank into two independent sub-ranks, improving the DRAM access parallelism of graph mining.

6.4 Area and Power Analysis

We show the area cost of DIMMining in Table 7. As we can see, one NMC module is less than $0.38mm^2$ under the 32 nm process (2 modules/rank). As a contrast, the typical area of a DRAM chip is ~ $100mm^2$. The power of each module takes 105mW, which is much smaller than the power of entire LRDIMM (*e.g.,* ~10W) . For reference, each PE of FlexMiner takes $0.18mm^2$ at 15nm technology node (FlexMiner does not provide power data). The effective area of DIMMining NMC module is $0.14mm^2$ when scaling to 15nm, which is similar to FlexMiner. We also simulate the end-to-end energy consumption, and here we take 4-CF pattern mining on LJ as an example. The end-to-end graph mining procedure consumes 1.002J in total, DRAM access, cache read/write, and computation, occupy $349mJ$, $366mJ$, and $287mJ$, respectively.

Table 7: DIMMining Design Overhead of One NMC Module (PM: processing matrix, FA: filter array, CT: compaction triangle, CA: counting & addition module.)

	Computing Units				Cache	Total
	PM	FA	CT	CA		
Area (μm^2)	33194	3247	10162	679	334815	382097
Power (mW)	10.20	1.27	2.49	0.10	91.76	105.82

6.5 Complex Patterns

In Table 8, we show the comparison results on more complex patterns: "Diamond" is the pattern in Figure 2(a), and we take the mining order provided by FlexMiner [32]; "House" is the pattern P1 in GraphPi [29]'s paper and FlexMiner does not provide the House mining results. The results show that DIMMining achieves 23.91× and 1.51× speedup compared with GraphPi and FlexMiner on complex patterns.

6.6 Discussions

6.6.1 Cache Size. The cache in DIMMining provides an efficient way to access frequently-used intermediate data, improving the mining performance. Figure 12 shows the trend of latency and cache hit rate when cache size changes in two representative graphs. As the cache hit rate increases, the DRAM access volume drops improving the mining performance. However, a larger cache will introduce longer access latency and higher hardware costs, degrading DIMMining performance. Since both the graph scale and the inquired pattern determine the intermediate data volume and memory access characteristics, we choose the 128KB cache as an optimal design in this paper.

6.6.2 Workload Balance. In DIMMining, we distribute the mining workloads to different ranks evenly due to the index of the start vertex of mining. Figure 13 shows the workloads of different ranks. Here the workload is defined by operation

Table 8: Complex Patterns Comparison (/Seconds)

	Diamond			House	
𝔾	GraphPi	FlexMiner	DIMMining	GraphPi	DIMMining
PP	0.072	-	**3.98E-05**	0.062	**4.96E-03**
AS	0.077	0.0009	**6.19E-04**	0.726	**3.89E-02**
MI	0.124	0.0071	**4.92E-03**	9.686	**6.21E-01**
PA	0.674	0.0810	**3.69E-02**	7.144	**5.30E-01**
YT	1.188	0.2183	**1.28E-01**	51.94	**6.02E+00**
LJ	2.679	**0.3182**	3.21E-01	299.4	**2.90E+00**

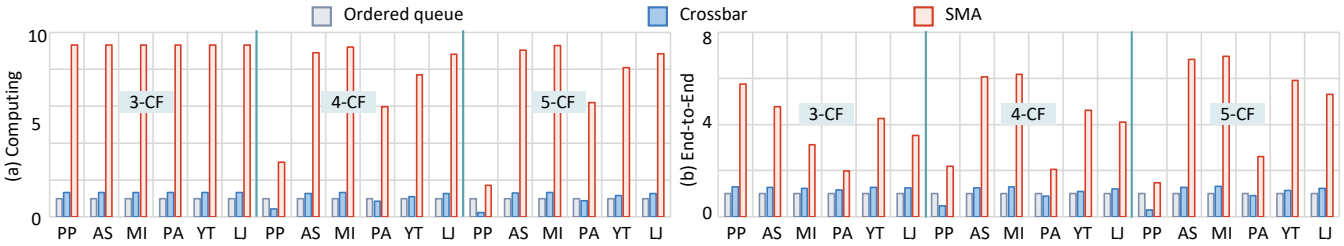


Figure 10: (a) Computing part and (b) End-to-end mining speedup of SMA and crossbar (over ordered queue).

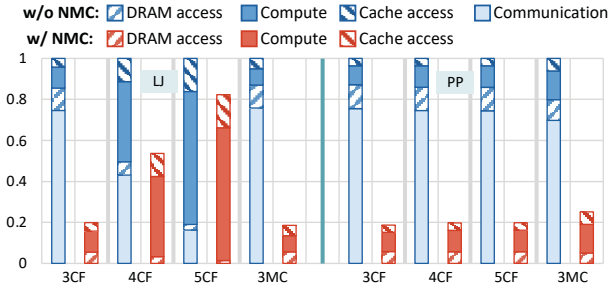


Figure 11: Latency comparison of DIMMining and baseline design w/o using NMC on LJ (left) and PP (right).

counts and data access volume, and the results are normalized to the average workload of these 16 ranks. Experimental results show that for most cases, the fluctuation of workload is less than 10%, revealing a good workload balance of different ranks. For the same mining pattern, the workload imbalance of small graphs is worse than that of large graphs. While for the same graph, a more complex pattern usually brings a worse workload imbalance.

6.6.3 Scalability. Figure 14(a) displays how the DIMMining performance scales with the number of ranks (on the premise that the memory space of graphs is less than 1GB which makes it possible to duplicate them in each rank). For small graphs (*e.g.,* PP), the processing time is quite short. When deploying such graph mining tasks on multiple ranks, the communication overhead (*e.g.,* C/A transfer) becomes non-negligible and causes sub-linear scaling with multiple ranks. On the other hand, for other graphs whose mining time is much longer, the performance increases linearly with the number of ranks, showing good scalability.

We also conduct an experiment that splits LJ onto 16 ranks for the mining on large-scale graphs, where the computation requires inter-rank graph data transmissions. We assume the inter-rank data transfer rate is $2400MT/s$. Results in Figure 14(b) show that the inter-rank communications only take 19.50% runtime overhead compared with the graph-copy-based mining (best case without inter-rank communications), guaranteeing good scalability for large-scale graphs.

6.6.4 Comparison with SISA. SISA is an instruction set architecture for graph mining [47] based on Processing-In-Memory. The major differences between DIMMining and SISA contain two aspects. (1) SISA pays more attention to the set-centric programming paradigm and formulations of graph algorithms, while DIMMining focuses on optimizing graph mining performance from both algorithm and hardware perspectives. We run the 3-CF mining on the graphs mentioned in SISA [17]. We assume SISA runs at 5GHz and use the clock cycle number given by SISA paper to estimate the mining time of SISA. Figure 15 shows that DIMMining achieves 5.78× to 12.32× speedup compared with SISA. (2) SISA is designed based on Hybrid Memory Cube (HMC) and modifications of DRAM peripheral circuits, while DIMMining does not change the DRAM internal circuit and can be applied to arbitrary DDR technology.

7 RELATED WORKS

7.1 Graph Mining

Previous designs [28–33, 39–41, 48–57] have proposed several techniques to improve the performance of graph mining problems. RStream [40] uses tuple streaming and partitioning to reduce I/O overhead and preserve the locality. GraphZero [30] and GraphPi [29] automatically generate and enforce a set of restrictions to break symmetry, and thus eliminate computation redundancy. GraphZero and GraphPi also address the significance of scheduling. Different schedules vary a lot in search space and data reusability. Gramer [33] introduces the idea of memory priority to optimize data access. The high-priority memory is used to permanently store the frequently accessed data to improve the locality. IntersecX [31] treats neighbors of vertex as a string and designs specialized S-cache to minimize data transfer. FlexMiner [32] generates pattern-specific execution plans and utilizes on-chip storage for data reuse, leading to 10× speedup compared with GraphZero. FINGERS [48] explores parallelism of different levels in the graph mining problem. In conclusion, optimizations like ensuring locality and improving parallelism are proved to be effective in mining problems, which are studied with novel designs in DIMMining.

7.2 Near-Memory-Computing

With the data volume sharply growing in recent years, the memory wall problem has become the main performance bottleneck in various memory-centric application scenarios.

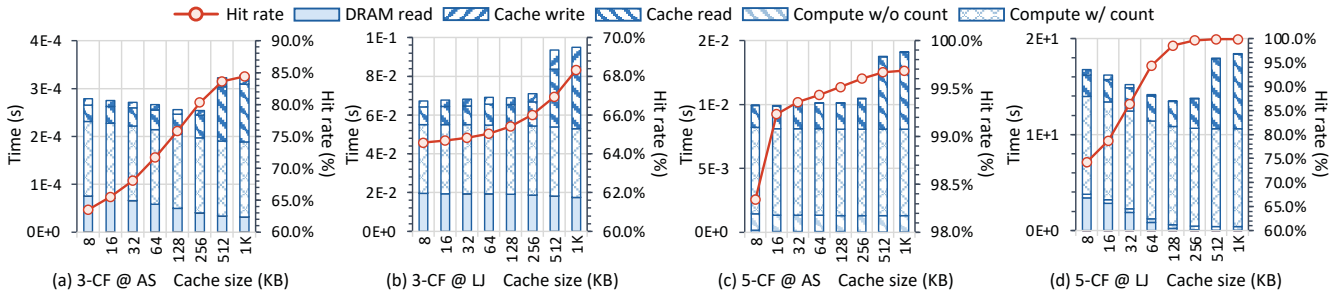


Figure 12: Latency breakdown and cache size design space exploration.

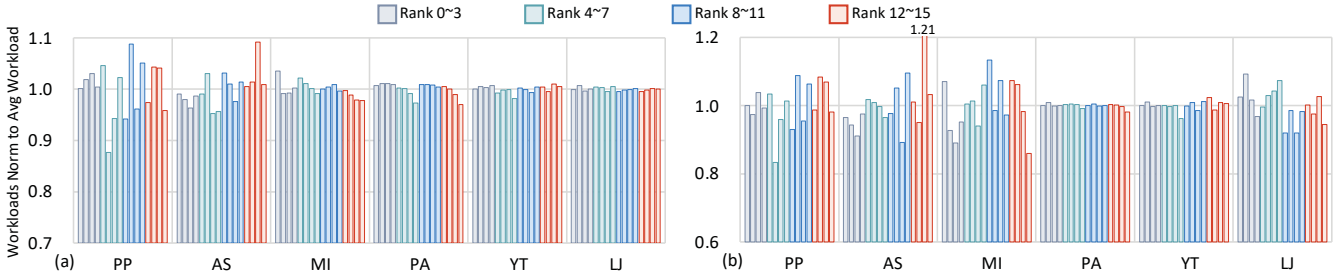


Figure 13: Normalized workloads of different ranks. (a) 3-CF (b) 5-CF.

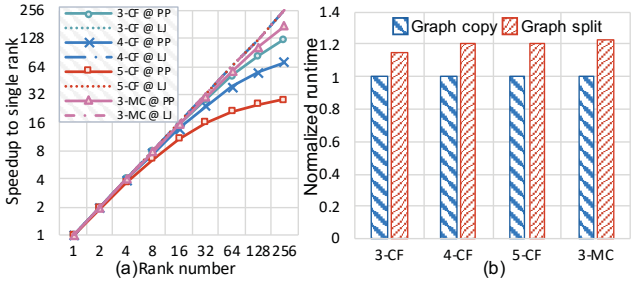


Figure 14: (a) Speedup to single rank under different rank number configurations. (b) Runtime @ LJ of duplicating the graph on 16 ranks (graph copy) and equally splitting the graph to 16 ranks (graph split).

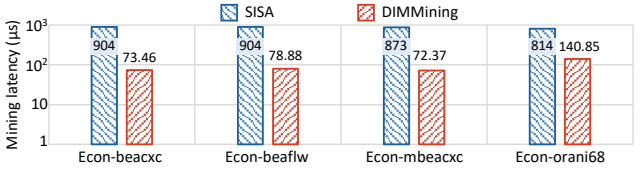


Figure 15: 3-CF mining latency of SISA and DIMMining.

Near-Memory-Computing (NMC) which puts computation units closer to the memory shows great potential to accelerate data-intensive applications because of low latency and less distance for data movement [58–66]. RecNMP [58] implements a sparse operation engine near the memory to aggregate feature vectors in recommendation systems, leading to over 4× throughput with conventional designs. TensorDIMM [59] introduces the NMC architecture to GPUs for deep learning-based recommendation systems, leading to the speedup of one order of magnitude. NEST [62] and Medal [63] introduce NMC for sequence processing in the domain of bio-informatics. All these studies show the potential of NMC for memory-centric applications, and motivate us to design DIMMining for the graph mining problems.

8 CONCLUSION

In this paper, we propose DIMMining, a DIMM-based graph mining architecture. DIMMining targets three challenges in the mining problem: heavy comparison overheads, low set operation parallelism, heavy data transfer, and proposes four novel techniques to solve these challenges. We propose index pre-comparison scheme with self anchor and neighbor partition to reduce comparison overheads during runtime. The set operation is accelerated by our BCSR (bitmap+CSR) format and a novel systolic merge array architecture. We implement DIMMining with the near-memory-computing architecture to alleviate heavy data movements. All these extensive experimental results show that DIMMining achieves **222.23**× and **139.51**× speedup compared with FPGAs and CPUs, and **3.62**× speedup against the state-of-the-art graph mining architecture.

ACKNOWLEDGEMENT

This work was supported by National Natural Science Foundation of China (No. U19B2019, 62104128, 61832007); China Postdoctoral Science Foundation (No. 2019M660641); National Key R&D Program of China (No. 2017YFA02077600); Tsinghua EE Xilinx AI Research Fund; Beijing National Research Center for Information Science and Technology (BNRist); Beijing Innovation Center for Future Chips. This work will be included in dgSPARSE project[1].

[1]The open source dgSPARSE project: https://dgsparse.github.io/

REFERENCES

[1] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[2] Seth A Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. Information network or social network? the structure of the twitter follow graph. In *International Conference on World Wide Web (WWW)*, pages 493–498, 2014.

[3] Alexandra Duma and Alexandru Topirceanu. A network motif based approach for classifying online social networks. In *IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 311–315. IEEE, 2014.

[4] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. Tux^2: Distributed graph computation for machine learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 669–682, 2017.

[5] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[6] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *International Conference on Neural Information Processing Systems (NeurIPS)*, pages 1025–1035, 2017.

[7] Peng-Cheng Lin and Wan-Lei Zhao. A comparative study on hierarchical navigable small world graphs. *Computing Research Repository (CoRR) abs/1904.02077*, 2019.

[8] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 42(4):824–836, 2018.

[9] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (SIGKDD)*, pages 974–983, 2018.

[10] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S Cenk Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):i241–i249, 2008.

[11] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. A selectivity based approach to continuous pattern detection in streaming graphs. *arXiv preprint arXiv:1503.00849*, 2015.

[12] Mohammed AlQuraishi. Alphafold at casp13. *Bioinformatics*, 35(22):4862–4865, 2019.

[13] Ichigaku Takigawa and Hiroshi Mamitsuka. Graph mining: procedure, application to drug discovery and recent advances. *Drug discovery today*, 18(1-2):50–57, 2013.

[14] Bernardete Ribeiro, Ning Chen, and Alexander Kovacec. Shaping graph pattern mining for financial risk. *Neurocomputing*, 2017.

[15] Leman Akoglu and Christos Faloutsos. Anomaly, event, and fraud detection in large network datasets. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 773–774. ACM, 2013.

[16] Balázs Adamcsek, Gergely Palla, Illés J. Farkas, Imre Derényi, and Tamás Vicsek. Cfinder: locating cliques and overlapping modules in biological networks. *Bioinformatics*, 22(8):1021–1023, 2006.

[17] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2015.

[18] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. Nxgraph: An efficient graph processing system on a single machine. In *International Conference on Data Engineering (ICDE)*, pages 409–420, 2016.

[19] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[20] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annual Technical Conference (ATC)*, pages 375–386, 2015.

[21] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 599–613, 2014.

[22] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2015.

[23] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 217–226, 2017.

[24] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a {PC}. In {*USENIX*} *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[25] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, volume 20, 2010.

[26] Guohao Dai, Tianhao Huang, Yu Wang, Huazhong Yang, and John Wawrzynek. Graphsar: A sparsity-aware processing-in-memory architecture for large-scale graph processing on rerams. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 120–126, 2019.

[27] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (IEEE TCAD)*, 38(4):640–653, 2019.

[28] Daniel Mawhirter and Bo Wu. Automine: harmonizing high-level abstraction and high performance for graph mining. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 509–523, 2019.

[29] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1418–1431. IEEE Computer Society, 2020.

[30] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. Graphzero: Breaking symmetry for efficient graph mining. *arXiv preprint arXiv:1911.12877*, 2019.

[31] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. Intersectx: An efficient accelerator for graph mining. *arXiv preprint arXiv:2012.10848*, 2020.

[32] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, and Chanwoo Chung Arvind. Flexminer: A pattern-aware accelerator for graph pattern mining. In *Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2021.

[33] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. A locality-aware energy-efficient accelerator for graph mining applications. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 895–907. IEEE, 2020.

[34] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[35] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 7(7):517–528, 2014.

[36] Bronwyn H Hall, Adam B Jaffe, and Manuel Trajtenberg. The nber patent citation data file: Lessons, insights and methodological tools, 2001.

[37] Xu Cheng, Cameron Dale, and Jiangchuan Liu. Statistics and social network of youtube videos. In *Interntional Workshop on Quality of Service*, pages 229–238. IEEE, 2008.

[38] likwid-perfctr: Measuring applications' interaction with the hardware using the hardware performance counter. [Online]. https://github.com/RRZE-HPC/likwid/wiki/likwid-perfctr.

[39] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 425–440, 2015.

[40] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 763–782, 2018.

[41] Shuo Han, Lei Zou, and Jeffrey Xu Yu. Speeding up set intersections in graph algorithms using simd instructions. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1587–1602, 2018.

[42] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009.

[43] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2015.

[44] Karthik Chandrasekar, Christian Weis, Yonghui Li, Benny Akesson, Norbert Wehn, and Kees Goossens. Drampower: Open-source dram power & energy estimation tool. *URL: http://www.drampower.info*, 22, 2012.

[45] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm. *Integration*, 58:74–81, 2017.

[46] Graphpi. [Online]. https://github.com/thu-pacman/GraphPi.

[47] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, et al. Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 282–297, 2021.

[48] Qihang Chen, Boyu Tian, and Mingyu Gao. Fingers: Exploiting fine-grained parallelism in graph mining accelerators. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 43–55, 2022.

[49] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: a pattern-aware graph mining system. In *European Conference on Computer Systems (EuroSys)*, pages 1–16, 2020.

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

[51] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: an efficient task-oriented graph mining system. In *European Conference on Computer Systems (EuroSys)*, pages 1–12, 2018.

[52] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng Guo. Kaleido: An efficient out-of-core graph mining system on a single machine. In *International Conference on Data Engineering (ICDE)*, pages 673–684. IEEE, 2020.

[53] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proceedings of the VLDB Endowment*, 13(8):1190–1205, 2020.

[54] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Sandslash: a two-level framework for efficient graph pattern mining. In *ACM International Conference on Supercomputing (ICS)*, pages 378–391, 2021.

[55] Xuhao Chen et al. Efficient and scalable graph pattern mining on gpus. *arXiv preprint arXiv:2112.09761*, 2021.

[56] Daniel Mawhirter, Samuel Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. Dryadic: Flexible and fast graph pattern matching at scale. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 289–303. IEEE, 2021.

[57] Chuangyi Gui, Xiaofei Liao, Long Zheng, Pengcheng Yao, Qinggang Wang, and Hai Jin. Sumpa: Efficient pattern-centric graph mining with pattern abstraction. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 318–330. IEEE, 2021.

[58] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. Recnmp: Accelerating personalized recommendation with near-memory processing. In *Annual International Symposium on Computer Architecture (ISCA)*, pages 790–803, 2020.

[59] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 740–753, 2019.

[60] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Gather-scatter dram: In-dram address translation to improve the spatial locality of non-unit strided accesses. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 267–280, 2015.

[61] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[62] Wenqin Huangfu, Krishna T Malladi, Shuangchen Li, Peng Gu, and Yuan Xie. Nest: Dimm based near-data-processing accelerator for k-mer counting. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[63] Wenqin Huangfu, Xueqi Li, Shuangchen Li, Xing Hu, Peng Gu, and Yuan Xie. Medal: Scalable dimm based near data processing accelerator for dna seeding algorithm. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 587–599, 2019.

[64] Weiyi Sun, Zhaoshi Li, Shouyi Yin, Shaojun Wei, and Leibo Liu. Abc-dimm: Alleviating the bottleneck of communication in dimm-based near-memory processing with inter-dimm broadcast. In *Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[65] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and TN Vijaykumar. Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 372–385. IEEE, 2020.

[66] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Syncron: Efficient synchronization support for near-data-processing architectures. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 263–276. IEEE, 2021.