SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems

Maciej Besta[1], Raghavendra Kanakagiri[2], Grzegorz Kwasniewski[1], Rachata Ausavarungnirun[3], Jakub Beránek[4], Konstantinos Kanellopoulos[1], Kacper Janda[5], Zur Vonarburg-Shmaria[1], Lukas Gianinazzi[1], Ioana Stefan[1], Juan Gómez-Luna[1], Marcin Copik[1], Lukas Kapp-Schwoerer[1], Salvatore Di Girolamo[1], Nils Blach[1], Marek Konieczny[5], Onur Mutlu[1], Torsten Hoefler[1]

[1]ETH Zurich, Switzerland [2]IIT Tirupati, India [3]King Mongkut's University of Technology North Bangkok, Thailand [4]Technical University of Ostrava, Czech Republic [5]AGH-UST, Poland

ABSTRACT

Simple graph algorithms such as PageRank have been the target of numerous hardware accelerators. Yet, there also exist much more complex graph *mining* algorithms for problems such as clustering or maximal clique listing. These algorithms are memory-bound and thus could be accelerated by hardware techniques such as Processing-in-Memory (PIM). However, they also come with non-straightforward parallelism and complicated memory access patterns. In this work, we address this problem with a simple yet surprisingly powerful observation: operations on sets of vertices, such as intersection or union, form a large part of many complex graph mining algorithms, and can offer rich and simple parallelism at multiple levels. This observation drives our cross-layer design, in which we (1) expose set operations using a novel programming paradigm, (2) express and execute these operations efficiently with carefully designed *set-centric* ISA extensions called SISA, and (3) use PIM to accelerate SISA instructions. The key design idea is to alleviate the bandwidth needs of SISA instructions by mapping set operations to two types of PIM: in-DRAM bulk bitwise computing for bitvectors representing high-degree vertices, and near-memory logic layers for integer arrays representing low-degree vertices. Set-centric SISA-enhanced algorithms are efficient and outperform hand-tuned baselines, offering more than 10× speedup over the established Bron-Kerbosch algorithm for listing maximal cliques. We deliver more than 10 SISA set-centric algorithm formulations, illustrating SISA's wide applicability.

CCS CONCEPTS

• **Hardware** → Emerging architectures; Memory and dense storage; Application-specific VLSI designs; Application specific instruction set processors; • **Computer systems organization** → **Architectures**; • **Theory of computation** → *Design and analysis of algorithms*; *Graph algorithms analysis*; Data structures design and analysis; Parallel algorithms; • **Mathematics of computing** → **Graph algorithms**; • **Information systems** → **Data mining**; Clustering; • **Computing methodologies** → **Parallel computing methodologies**.

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480133

KEYWORDS

Graph Mining, Graph Pattern Matching, Graph Learning, Clique Mining, Clique Listing, Clique Enumeration, Subgraph Isomorphism, Parallel Graph Algorithms, Processing In Memory, Processing Near Memory, Graph Accelerators, Instruction Set Architecture

ACM Reference Format:
Maciej Besta[1], Raghavendra Kanakagiri[2], Grzegorz Kwasniewski[1], Rachata Ausavarungnirun[3], Jakub Beránek[4], Konstantinos Kanellopoulos[1], Kacper Janda[5], Zur Vonarburg-Shmaria[1], Lukas Gianinazzi[1], Ioana Stefan[1], Juan Gómez-Luna[1], Marcin Copik[1], Lukas Kapp-Schwoerer[1], Salvatore Di Girolamo[1], Nils Blach[1], Marek Konieczny[5], Onur Mutlu[1], Torsten Hoefler[1]. 2021. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3466752.3480133

1 INTRODUCTION

Research on graph analytics in computer architecture has mostly targeted graph algorithms based on vertex-centric formulations [6, 7, 12, 22, 65, 88, 113, 120, 142, 177, 183]. Some works also focus on edge-centric or linear algebra paradigms [90, 134, 149, 151]. Such algorithms have complexities described by *low-degree* polynomials [91], e.g., $O(n + m)$ for Breadth-First Search (BFS) [42] and $O(m \cdot \#\text{iterations})$ for iteration-based PageRank (PR) [21], where n and m are numbers of vertices and edges, respectively.

Yet, there are numerous important problems and algorithms in the area of ***graph mining*** [23, 39, 84, 137, 156] that received little or no attention in computer architecture. One large class is ***graph pattern matching*** [84], which focuses on finding certain specific subgraphs (also called motifs or graphlets). Examples of such problems are k-clique listing [44, 58], maximal clique listing [26, 29, 51, 158], k-star-clique mining [79], and many others [39]. Another class is broadly referred to as ***graph learning*** [39], with problems such as unsupervised learning or clustering [81], link prediction [8, 102, 105, 155], or vertex similarity [98]. All these problems are used in social sciences [51], bioinformatics [51], computational chemistry [153], medicine [153], cybersecurity [49], healthcare [157], web graph analysis [85], and many others [30, 39, 74, 84]. These problems often run in time at least quadratic in the number of vertices, and many problems are NP-complete [26, 39, 44, 159]. Thus, they often differ significantly in their performance properties from "low-complexity" problems such as BFS or PageRank.

Importantly, the established vertex-centric model, originally proposed in the Pregel graph processing system [108], does *not* effectively express graph mining problems. It exposes only the local graph structure: A thread executing a vertex kernel for any vertex v can only access the neighbors of v. While this suffices for algorithms such as PageRank, graph mining often requires non-local knowledge of the graph structure [39]. Obtaining such knowledge in the vertex-centric paradigm is hard or infeasible, as noted by Kalavri et al. [88] (*"(...) graph algorithms, like triangle counting, are not a good fit for the vertex-centric model"*) and many others [93, 103, 136, 172]. Similar arguments apply to other paradigms such as GraphBLAS [90, 134] and to frameworks such as Ligra [145]. They do not support many graph mining problems, and we discuss in Table 1 and Section 4.

Several graph mining software frameworks (Peregrine [80] and others [33, 35, 48, 78, 86, 111, 112, 156, 171, 173, 179]) were proposed. Unfortunately, they (1) focus exclusively on only *a few* graph pattern matching problems, and (2) usually do *not* provide theoretical guarantees on total work [24] (unlike parallel graph algorithms for *specific* mining problems). Overall, there is a need for a graph mining paradigm that would (1) enable expressing many graph mining problems, and (2) offer competitive theoretical work guarantees [24].

Moreover, past works illustrated that graph mining algorithms are memory bound [37, 50, 80, 175, 178]. This is because these algorithms generate and heavily use large intermediate structures, but, similarly to algorithms such as PageRank, they are not compute-intensive [51, 80, 176]. We show this in Figure 1: When we increase the number of parallel threads, runtime decrease flattens out and stalled CPU cycle count increases. This motivates using processing-in-memory (PIM) to obtain the much needed speedups in graph mining. While PIM is not the only potential solution for hardware acceleration of graph mining, we select PIM because it (1) represents one of the most promising trends to tackle the memory bottleneck [56, 117] outperforming various other approaches [141], (2) offers well-understood designs [118], and (3) brings very large speedups in *simple* graph algorithms such as BFS or PageRank (see more than 15 works in Table 7). Yet, graph mining algorithms are *much more complex* than PageRank, BFS, and similar: they employ deep recursion, create many intermediate data structures with non-trivial inter-dependencies, and have high load imbalance [51, 171]. As we show in Section 10, *no existing HW design targets broad graph mining (i.e., both graph pattern matching and graph learning), or explores PIM techniques for accelerating broad graph mining.*

To address all these issues, we propose a novel design that is high-performance (empirically *and* theoretically), applicable to many

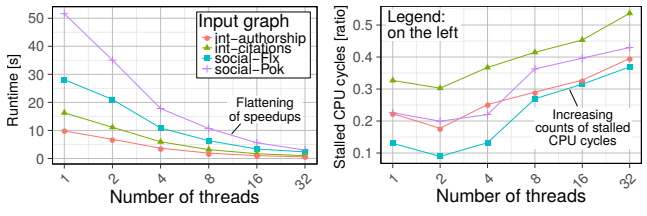


Figure 1: Runtimes and stalled CPU cycle count, for various numbers of parallel threads, using the Bron-Kerbosch algorithm for listing maximal cliques in different input graphs (Section 9 describes our evaluation methodology).

Abstraction or programming model	A?	Pattern M.			Learning			"Low-c."			Remarks
		mc	kc	ds	si	vs	lp	cl	av	tc	bf	cc	pr	
Vertex-centric (ver-c)	✖	✖	✖	✖	✖	✖	✖	✖	✖	🔋*	🔋	🔋	🔋	*High comm. costs
Edge-centric (edge-c)	✖	✖	✖	✖	✖	✖	✖	✖	✖	🔋*	🔋*	🔋*	🔋	*High work and depth
Array maps	✖	✖	✖	🔋	✖	✖	✖	🔋*	✖	🔋	🔋	🔋	🔋	*Only low-diameter decomp.
GraphBLAS [90]	🔋 (L)	✖	✖	✖	🔋*	✖	✖	✖	✖	🔋	🔋	🔋†	🔋	*The only existing SI scheme only uses trees as patterns [34]
Neural message passing, graph networks [13, 62]	🔋 (L)	✖	✖	✖	🔋†	🔋	🔋	🔋	🔋	✖	✖	✖	🔋	†GNNs are as powerful as the Weisfeiler-Lehman test [170].
Pattern matching	✖	🔋*	🔋*	🔋*	🔋*	✖	✖	✖	✖	🔋*	✖	✖	✖	*No bounds, low performance
Joins [36]	🔋 (R)	✖	🔋*	🔋*	✖	🔋*	🔋*	🔋*	✖	🔋*	✖	🔋*	🔋*	*No bounds, low performance
Set-Centric / SISA	🔋 (S)	🔋	🔋	🔋	🔋	🔋	🔋	🔋	🔋	🔋	✖	✖	✖	

Table 1: Comparison of the set-centric programming approach and SISA to existing graph processing abstractions/programming models, focusing on support for selected graph mining problems (pattern matching, learning), and for "low-complexity" graph problems. A?: Underlying algebra? **L**: linear, **R**: relational, **S**: set. "🔋": Support / significant focus. "🔋": Partial support / some focus. "✖": no support / no focus. **Pattern M.**: selected graph pattern matching problems, **mc**: maximal clique listing, **kc**: k-clique listing, **ds**: dense subgraph, **si**: subgraph isomorphism, **Learning**: selected graph learning problems, **vs**: vertex similarity, **lp**: link prediction, **cl**: clustering or community detection, **av**: accuracy verification (of link prediction outcomes), **"Low-c."**: selected "low-complexity" problems targeted by vast majority of existing works on graph processing. **tc**: triangle counting, **bf**: BFS, **cc**: connected components, **pr**: PageRank. **The analysis in this table is extended in Section 10 and Table 7** by detailing specific HW accelerators for graph processing.

graph mining problems, and easily amenable to PIM acceleration. We first observe that large parts of many graph mining algorithms can be expressed with simple set operations such as intersection $\cap$ or union $\cup$, where sets contain vertices or edges. This drives our **set-centric programming paradigm**, in which the developer identifies sets and set operations in a given algorithm. These set operations are then mapped to a small and simple yet expressive group of instructions, offering a rich selection of storage/performance tradeoffs. These instructions are offloaded to PIM units. We call these instructions ***SISA*** as they form *"**S**et-centric" **ISA*** extensions that enable a simple interface between numerous graph mining algorithms and PIM hardware. Overall, our cross-layer design consists of three key elements: a new set-centric programming paradigm and formulations of graph algorithms (contribution **#1**), set-centric ISA extensions with its instructions, implemented set operations, and set organization (contribution **#2**), and PIM acceleration (contribution **#3**).

Overall, we advocate using set algebra as a basis for the design of graph mining algorithms. Our set-centric paradigm is the first to use set operations as fundamental general building blocks for both algorithmic formulations *and* their execution. Using set algebra ensures that SISA set-centric algorithms are succinct, applicable to many problems, and theoretically efficient.

For the in-memory acceleration of SISA, we investigate which types of PIM are beneficial for which set operations. We process sets stored as bitvectors using in-situ PIM [57, 118], as offered in Ambit [64, 141], ELP2IM [168], DRISA [100], or ComputeDRAM [53], for highest performance and energy efficiency ("**SISA processing using memory**" – **SISA-PUM**). In contrast, while sets stored as sparse arrays cannot be simply processed in situ with today's technology, they can use the high throughput and low latency of near-memory PIM [57, 104, 118, 122] as offered in the 2D UPMEM architecture [63, 96] or logic layer of 3D DRAM such as Hybrid Memory Cube (HMC) [83] ("**SISA processing near memory**" – **SISA-PNM**). For even higher performance, we provide a small HW controller that selects the best variant of a set instruction to be executed on-the-fly.

Overall, our results show that graph mining algorithms, although complex and lacking straightforward parallelism, greatly benefit from PIM. Our key solution is using parallelism offered by set operations and exposed with the set-centric approach. This solution

harnesses parallelism at the level of bits, DRAM subarrays, and vaults. We show that SISA-enhanced algorithms are theoretically efficient (contribution **#4**) *and* empirically outperform tuned parallel baselines (contribution **#5**), for example offering more than 10× speedup for many real-world graphs over the established Bron-Kerbosch algorithm for listing maximal cliques [51]. Finally, for usability, we integrate SISA with the RISC-V ISA [166].

2 NOTATION AND BACKGROUND

Graphs We model an undirected graph G as a tuple (V, E); V and $E \subseteq V \times V$ are sets of vertices and edges; $|V| = n$, $|E| = m$. Vertices are modeled with integers ($V = \{1, ..., n\}$). $N(v)$ denote the neighbors of $v \in V$; d and $d(v)$ denote G's maximum degree and a degree of v. In some cases, we consider *labeled* graphs $G = (V, E, L)$; L is a labeling function that maps a vertex or an edge to a label.

Set Representations SISA heavily uses sets. Consider a set of k vertices $S = \{v_1, ..., v_k\} \subseteq V$ (we focus on vertex sets, but SISA also works with edges). One can represent S as a simple contiguous **sparse array (SA)** with integers from S ("sparse" means that only non-zero elements are explicitly stored). SA's size is $W|S|$ [bits] where W is the memory word size (we assume that the maximum vertex ID fits in one word). One can also represent S with a **dense bitvector (DB)** of size n [bits]: the i-th set bit indicates that a vertex $i \in S$ ("dense" means that all zero bits are explicitly stored).

3 OVERVIEW & CROSS-LAYER DESIGN

We now overview SISA's cross-level design, see Figure 2.

(a) Set-Centric Formulations [Section 5 & 5.1] SISA relies on set-centric formulations of algorithms in graph mining. While some algorithms (e.g., Bron-Kerbosch [51]) by default use rich set notation, many others, such as k-clique listing by Danisch et al. [44], do not. In such cases, we develop such formulations. Details on deriving set-centric formulations are in Section 5.1; the key common step is to express two nested loops, commonly used to identify connections between two sets of vertices, with a single intersection of these sets.

A set can be represented in different ways, and a set operation can be executed using different set algorithms. A set-centric formulation

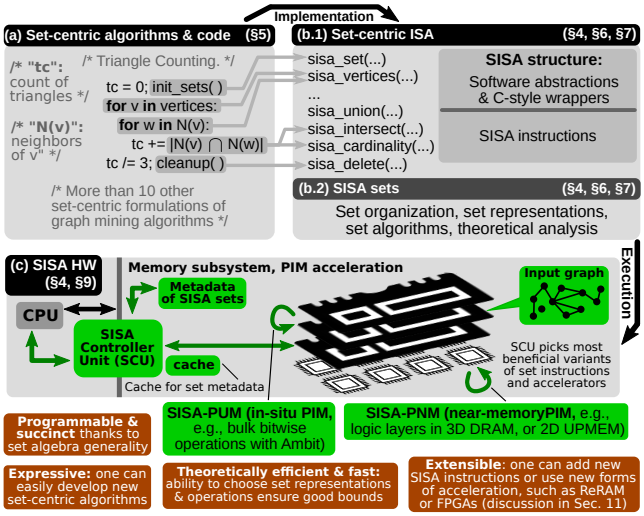


Figure 2: The overview of SISA with a summary of new introduced architecture and graph representation elements (green) and advantages (brown).

hides these details, focusing on *what* a given graph algorithm does, and not *how* it is done.

(b.1) Set-Centric ISA (Instructions) [Section 6] Our ISA extension implements set operations. These instructions support all variants of operations, for example there is an instruction for both merge and galloping set intersection (details in Section 6). We also provide a thin software layer: iterators over sets and C-style wrappers for SISA instructions. For programmability and performance, many SISA instructions automatize selecting the best set operation variant on-the-fly.

(b.2) Set-Centric ISA (Organization of Sets) [Section 6] We represent sets as DBs or SAs. The former are processed by bulk bitwise in-situ PIM, harnessing huge internal DRAM bandwidth (SISA-PUM). The latter use near-memory PIM, for example DRAM cores in the UPMEM architecture, or logic layers in 3D stacked DRAM, harnessing the large through-silicon via (TSV) bandwidth (SISA-PNM).

(c) HW Implementation Details [Section 8] For maximum programmability and performance, we use hardware to automatically decide between SISA-PUM and SISA-PNM, or a set algorithm variant (merge vs. galloping). For this, we use a dedicated unit called the SISA Controller Unit (SCU).

4 GENERAL & FAST GRAPH MINING

The set-centric approach is superior to other graph programming paradigms in that (1) it supports many graph mining problems and (2) it enables algorithms with competitive theoretical bounds on performance (we discuss (2) in Section 7; this is often a key to low runtimes [46, 91]). The analysis results for (1) are in Table 1.

To illustrate the above points, we first extensively examined the related literature to identify representative **graph mining problems** and important **graph processing paradigms** [4, 9, 30, 52, 84, 97, 98, 102, 105, 126, 128, 130, 154, 163]. For the former, we pick four problems from both graph pattern matching and graph learning areas (maximal clique listing [26], k-clique listing [38], dense subgraph discovery [61, 97], subgraph isomorphism [159], vertex similarity [98, 131], link prediction [8, 102, 105, 155], graph clustering [81, 137], verification of prediction accuracy [162]). For fairness, we also consider four popular "low-complexity" problems, targeted by many past works (triangle counting, BFS, connected components, and PageRank). For the latter, we first select *vertex-centric* [108] and *edge-centric* [134], two established graph processing paradigms implemented in the Pregel and X-Stream systems. Second, we pick *vertex/edge array maps* from Ligra [145], an approach for developing graph algorithms based on transforming arrays of vertices or edges according to a specified map. Third, we consider *GraphBLAS* and its linear algebraic approach [90], where graph algorithms are expressed with linear algebra building blocks such as matrix-vector products. Moreover, we consider *pattern matching frameworks* [52] that usually employ some form of exploring neighbors of each vertex, combined with user-specified filtering, to search for specified graph patterns. For completeness, we also consider recent attempts at solving graph problems with novel deep learning [15] paradigms such as *graph neural networks (GNN)* [17, 167] and others [59], as well as *joins* and principles from relational databases and the associated algebra [180].

The analysis results are in Table 1. Overall, no single paradigm, except for the set-centric approach, enables efficient graph mining

Problem	Algorithm	Used set operations
Maximal clique list.	Bron-Kerbosch [51]	$A \cup B$, $A \cap B$, $A \setminus B$
k-clique listing	Danisch et al. [44] + **[This work]**	$A \cap B$
4-clique counting	**[This work]**	$A \cap B$, $\|A \cap B\|$
Triangle counting	[well-known]	$\|A \cap B\|$
k-clique-star listing	Jabbour et al. [79]	$A \cap B$, $A \cup B$
k-clique-star listing	**[This work]**	$A \cap B$
Subgr. isomorphism	**[This work]**	$A \cap B$, $\|A \cap B\|$, $A \cup B$, $A \setminus B$
Vertex similarity	Jaccard coeff., others [19, 131]	$\|A \cap B\|$, $\|A \cup B\|$
Clustering	Jarvis-Patrick [81]	$\|A \cap B\|$, $\|A \cup B\|$
Link prediction (LP)	Jaccard coeff., others [131]	$\|A \cap B\|$, $\|A \cup B\|$
LP accuracy testing	Wang et al. [162]	$A \setminus B$, $\|A \cap B\|$
Approx. degeneracy	Besta et al. [16]	$A \setminus B$

Table 2: Overview of set-centric graph algorithms. In maximal clique listing, subgraph isomorphism, and clustering, one also uses variants of union and difference where one set is always a single-element set (i.e., $A \cup \{b\}, A \setminus \{b\}$). Bolded text indicates algorithms with set-centric formulations derived in this work.

algorithms for the considered problems. Some paradigms, such as the vertex-centric or the edge-centric model, do not focus on such problems at all. Other paradigms, for example array maps or GNNs, address only certain problems. Finally, graph pattern matching or RDBMS can solve different graph mining problems, but they do not offer formal guarantees, as indicated by past work.

5 SET-CENTRIC GRAPH ALGORITHMS

We now present set-centric formulations of graph mining algorithms. The used set operations are in Table 2.

Notes on Listings Set operations accelerated by SISA are marked with the gray color. "`[in par]`" indicates that in a given loop one can issue set operations in parallel. We ensure that the parallelization does not involve conflicting memory accesses. 1111 We now focus on formulations and we discuss set representations, instructions, and parallelization later. For clarity, we exclude unrelated optimizations from the listings.

Maximal Cliques Listing A clique is a fully-connected subgraph of an input graph; a maximal clique is a clique not contained in a larger clique. Finding all maximal cliques is an important NP-hard problem [45, 129, 150, 164]. Algorithm 1 shows the widely used recursive backtracking Bron-Kerbosch algorithm (BK) [26, 29, 51]. The main recursive function `BKPivot` (Line 4) has three arguments that are dynamic sets containing vertices. R is a partially constructed, non-maximal clique c, P are candidate vertices that may belong to c but are yet to be tried, and X are vertices that definitely do not belong to c. The algorithm recursively calls `BKPivot` for each new candidate vertex, checks if this gives a clique, and updates accordingly P and X. Some optimizations need more set operations, but they reduce the search space of potential cliques [158]. For example, the set of candidates (for extending a clique c) is $P \setminus N(u)$ instead of P, where $u \in P \cup X$. Overall, BK is non-trivial, with many different set operations, *including non anti-monotonic ones such as union*. Thus, it shows SISA's ability to accelerate complex algorithms.

```
1 /* Input: A graph G. Output: Maximal clique R (R ⊆ V).*/
2 P = V; R = ∅; X = ∅; //Init sets appropriately.
3 for v ∈ V [in par] do: BKPivot({v}, P, X);
4 function BKPivot(R, P, X):
5   if |P| == 0 and |X| == 0: return R; //Found a maximal clique
6   u = /* Choose a pivot vertex from P∪X */
7   for v ∈ P\N(u) do: BKPivot( R∪{v} , P∩N(v) , X∩N(v) )
8     P = P\{v} ; X = X∪{v}
```

Algorithm 1: Maximal Clique Listing (Bron-Kerbosch) [26, 29].

k-Clique-Star Listing A k-clique-star is a k-clique with additional adjacent vertices that are connected to all the vertices in the clique. k-clique-stars relax the restrictive nature of cliques [79]. Algorithm 2 shows the scheme. We first find k-cliques. Then, for each k-clique, one finds additional vertices that form stars with intersections and a union.

```
1 /* Input: A graph G. Output: All k-clique-stars, S.*/
2 C = /* First, find k-cliques (e.g., with Table 3)*/
3 S = ∅ //S is a set with identified k-clique-stars.
4 foreach c = (V_c, E_c) ∈ C do: //For each k-clique...
5   X = ⋂_{u∈V_c} N(u) //Intersect all N(u) such that u ∈ V_c
6   G_s = X ∪ V_c //Derive the actual k-clique-star
7   S = S ∪ {G_s} //Add an identified k-clique-star to S
8 //At the end, remove duplicates from S
```

Algorithm 2: k-clique-star listing [79].

Subgraph Isomorphism Subgraph isomorphism (SI) is a key graph problem where one checks whether a given (usually small) graph G_2 is a subgraph of a graph G_1. Here, we consider an established VF2 algorithm [41]. Due to its complexity, in Algorithm 3, we only provide the most important part that recursively constructs a candidate set of vertices from G_1, and verifies if it matches the pattern G_2.

We use SI as an example of how SISA supports **labeled graphs**. In VF2 [41], for each transition between states, one first verifies if the structure of G_2 matches that of G_1 (Line 11). Then, label matching is verified independently (Lines 12-13). Checking if vertex labels match, i.e., if $L(v_1)$ equals $L(v_2)$, is trivial. Yet, a graph may also contain edge labels that need to be matched. This could be done with a standard approach without set operations [41]. However, the generality of set notation also enables supporting label verification. For this, we first identify all edges in G_1 where one endpoint is the newly matched vertex v_1 and the other endpoint v_1' is already matched (i.e., $v_1' \in M_1(s)$). This is done with an intersection $N_1(v_1) \cap M_1(s)$. Then, we find the vertex with which v_1' is matched, see the second loop in Line 17. Finally, we verify that the respective labels match (Line 18).

```
1 /* Input: target graph G_1, pattern G_2. Output: mapping between graphs.*/
2 s_0 = {}; M(s_0) = ∅; // Initial state
3 Match(s_0); // Algorithm start
4 function Match(s):
5  if M(s) covers all nodes in pattern graph: output M(s); return;
6  P(s) = /* compute set of candidate pairs to be added to M(s) */
7  for (v_1, v_2) ∈ P(s) do:
8    checkCore = /* original R_core rule */
9    checkTerm = |N_1(v_1) ∩ T_1(s)| ≥ |N_2(v_2) ∩ T_2(s)|
10   checkNew = |N_1(v_1) \ (M_1(s) ∪ T_1(s))| ≥ |N_2(v_2) \ (M_2(s) ∪ T_2(s))|
11   checkFeasibility = checkCore ∧ checkTerm ∧ checkNew
12   checkSemantic = verify_labels(v_1, v_2, s) //If we use labels.
13   checkFeasibility = checkFeasibility ∧ checkSemantic //If we use
       labels.
14   if checkFeasibility : s' = NewState(s, v_1, v_2); Match(s')
15 //Check if labeling of v_1 and v_2 and their neighborhoods matches:
16 bool verify_labels(v_1, v_2, s):
17  forall v_1' ∈ N_1(v_1) ∩ M_1(s): forall (v_1', v_2') ∈ M(s):
18    if (L(v_1) != L(v_2)) or (L(v_1, v_1') != L(v_2, v_2')): return false
19  return true
```

Algorithm 3: Subgraph isomorphism [41]. M_1 and M_2 denote the current partial mappings associated with G_1 and G_2, respectively. T_1 and T_2 denote sets of vertices adjacent to the ones in M_1 and M_2, respectively. N_1 and N_2 denote neighborhoods within G_1 and G_2, respectively. `verify_labels` is used if graphs are labeled.

For **Frequent Subgraph Mining (FSM)**, we use an established *Apriori-based* scheme [5],[84, Algorithm 3.1]. We show it in Algorithm 4. It first generates candidate subgraphs C_k (Line 6) and then checks their counts `cnt` in the input graph (Line 8) using subgraph

isomorphism (SI) as a fundamental kernel [84] (combining candidate generation and occurrence verification is a very popular FSM approach [5, 66, 94, 95], also see other references in [84]). If the count is above a certain user selected threshold ($\sigma \cdot n$), a candidate is added as a found frequent subgraph (Line 9). VF2, an SI algorithm covered in this section, was found to be an efficient kernel for FSM; all SISA operations in SI are reused. Generation of candidate subgraphs (`candidate_gen`) is less time-consuming than SI [84]. Still, it also benefits from set operations; for example, joining trees that represent candidates, a key operation in a kernel by Hido and Kawano [72], is done using set union [84]. These trees can be implemented with either n-bit dense bitvectors or sparse arrays, benefiting from SISA-PUM or PNM (user's choice).

```
1 /* Input: target graph (G), minimum support / count of a found pattern (σ).
2  * Output: sets of frequent subgraphs of sizes 1, 2, ..., k (F_1, F_2, ..., F_k).*/
3 F_1 = V; k = 2 //k = 2 means we start recursion from edges.
4 //Use all subgraphs in F_{k-1} to generate candidates of size k:
5 while F_{k-1} ≠ ∅ do: //C_k (below) are candidate subgraphs of size k
6   F_k = ∅; C_k = candidate_gen(F_{k-1}) //Use any selected kernel[84]
7   foreach g ∈ C_k do:
8     cnt = SI(g, G) //For set operations in SI, see Algorithm 3
9     if cnt ≥ σn and g ∉ F_k: F_k ∪= g
10  k++
```

Algorithm 4: Frequent subgraph mining [84].

Vertex Similarity & Clustering Various measures assess how similar two vertices v and u are, see Algorithm 5. They can be used on their own, or as a main building block of more complex algorithms such as clustering. In clustering, one iterates over all adjacent vertex pairs, and uses their similarity to decide if the pair belongs to a cluster.

```
1 /* Input: A graph G. Output: Similarity S ∈ ℝ of neighborhoods
2  * N(u) and N(v) of some vertices u and v. */
3 S_J(v,u) = |N(v) ∩ N(u)| / |N(v) ∪ N(u)| /* Jaccard Similarity */
```

Algorithm 5: Vertex similarity measures.

Finally, SISA does not target the **"low-complexity" algorithms**, as they offer few opportunities for set-centric acceleration [20, 25, 42, 60, 114, 115, 144, 147, 148, 152, 172]. For example, in PageRank, one updates vertex ranks in two nested loops, which is not easily expressible with set operations. Our work is already more general than other pattern matching accelerators / frameworks, as it supports many more problems beyond simple pattern matching.

5.1 Deriving a Set-Centric Formulation

Often, algorithms use set notation, and one may simply pick operations for memory acceleration. This is the case with, for example, Jarvis-Patrick clustering. Still, one may need to apply more complex changes to "expose" set instructions. The general rule is to associate used data structures with sets, and then identify respective set operations. As an example, we compare a traditional snippet for deriving the count of all 4-cliques `cnt`, a derived set-centric algorithmic formulation, and the corresponding SISA snippet in Table 3. The key algorithmic change is using set intersections instead of explicitly verifying if vertices are connected. For example, instead of iterating over all neighbors of v_1-v_3 (Lines 4-6, the top snippet), in SISA, we intersect neighborhoods of v_1-v_3 (Line 4 & 6, the middle snippet) to filter 4-cliques.

6 SISA: DESIGN, SYNTAX, SEMANTICS

We now detail SISA's design, see Figure 3.

```
1 //Non set-centric code:
2 CSR_Graph g(G); //Standard codes often use some form of CSR
3 #pragma omp parallel for
4 for (auto v1: g.V()) //For all vertices in parallel.
5   for (auto v2: g.N_out(v1)) //Explore neighborhoods of v1-v4...
6     for (auto v3: g.N_out(v2)) //...searching for a 4-clique
7       for (auto v4: g.N_out(v3)) //If v1-v4 are connected pairwise
8         if(g.edge(v1,v3) && g.edge(v1,v4) && g.edge(v2,v4)) ++cnt;
```

```
1 //A set-centric algorithmic formulation:
2 for v_1 ∈ V in parallel do: //For all vertices in parallel.
3   for v_2 ∈ N(v_1) do: //For each neighbor of v_1...
4     S_1 = N(v_1) ∩ N(v_2) //Find common neighbors of v_1 and v_2.
5     for v_3 ∈ S_1 do: cnt += |S_1 ∩ N(v_3)|
```

```
1 //SISA (simplified) set-centric code:
2 SetGraph g = SetGraph(G);
3 #pragma omp parallel for
4 for (auto v1: g.V()) for (auto v2: g.N_out(v1)) {
5     auto S1 = intersect(g.N_out(v1), g.N_out(v2));
6     for (auto v3: S1) cnt += intersect_card(S1, g.N_out(v3)); }
```

Table 3: Finding all 4-cliques: a traditional (non-set-centric) snippet, a set-centric algorithmic formulation derived in this work, and a SISA set-centric snippet.

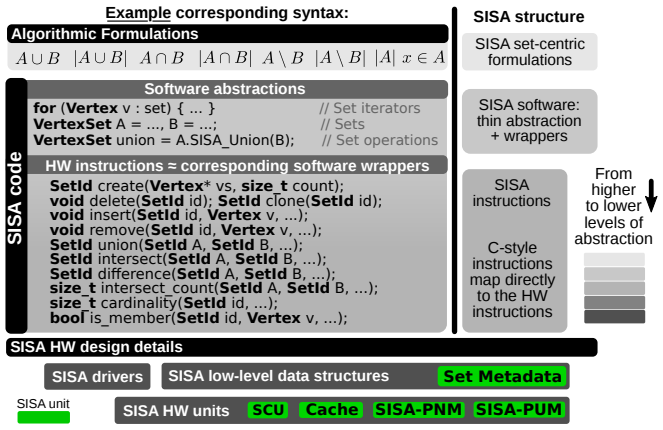


Figure 3: Overview of SISA instructions and syntax at different levels of abstraction.

6.1 Representation of Sets

The first key question is how to represent sets: SISA's "first-class citizens". We observe that – in each graph algorithm – there are two fundamentally different classes of data structures. One class are **(1) vertex neighborhoods** $N(v)$ that maintain the structure of the input graph. There are n such sets, their total size is $O(m)$, and each single neighborhood is static (we currently focus on static graphs) and sorted (following the established practice in graph processing [109]). Another class are **(2) auxiliary structures**, for example P in Bron-Kerbosch (Listing 1). These sets are used to maintain some algorithmic state. They are usually dynamic, they may be unsorted, their number (in a given algorithm) is usually a (small) constant, and their total size is $O(n)$. While SISA enables using any set representation for any specific set, we offer certain recommendations to maximize performance.

SAs should be used for small neighborhoods and DBs for the large ones (in the evaluation, we vary the threshold so that 5%-30% largest neighborhoods use DBs). This approach is memory efficient. For example, for $|N(v)| = n/2$, a DB takes only n bits while an SA uses $16n$ bits (for a 32-bit word size).

Auxiliary sets benefit from being stored as dense bitvectors. This is because such sets are often dynamic, and updates or removals take $O(1)$ time. As in practice there is usually a small constant number of such sets in considered algorithms, the needed storage is not excessive, e.g., less than 3% on top of a CSR for a graph with the average degree 100 (such as orkut), assuming using 32 threads and the Bron-Kerbosch algorithm, with auxiliary sets P, X, and R (the space complexity is $O(Tn)$ where T is #threads). We analyze and confirm it for other algorithms and datasets. For example, in SI, the storage complexity is (TnP) (where P is the size of the subgraph), which is also negligible in practice as P is usually small. To control space usage, the user may pre-specify that, above a certain number of DBs, SISA starts to use SAs only.

The user controls selecting a set representation. For programmability, SISA offers a predefined graph structure, where small and large neighborhoods are **automatically** created (when a SISA program starts) as sparse arrays and dense bitvectors, respectively. A given neighborhood $N(v)$ is stored as a DB whenever $|N(v)| \geq t \cdot n$ ($t \in (0;1)$ is a user parameter that controls a "bias" towards using DBs or SAs) and it does not exceed a storage budget limit set by the user (SISA by default uses a limit of 10% of the additional storage on top of the graph size when stored only with SAs). For example, $t = 0.5$ indicates that each vertex connected to at least 50% of all vertices has its neighborhood stored as a DB.

ins	Set op.	A and B represent.	Set algorithm	S?	Time complexity	Input size [bits]		Main form of data transfer (§ 8.2)
0x0	$A \cap B$	SA ∩ SA	Merge	👍, 👍	$O(\|A\| \quad \|B\|)$	$W\|A\|$	$W\|B\|$	Streaming
0x1	$A \cap B$	SA ∩ SA	Galloping	👍, 👍	$O(\|A\| \log \|B\|)$	$W\|A\|$	$W\|B\|$	Random accesses
0x2	$A \cap B$	SA ∩ SA	Merge vs. gallop.	👍, 👍	cf. 0x0 and 0x1	$W\|A\|$	$W\|B\|$	cf. 0x0 and 0x1
0x3	$A \cap B$	SA ∩ DB	Galloping	👍, na	$O(\|A\|)$	$W\|A\|$	n	Random accesses
0x4	$A \cap B$	DB ∩ DB	Bitwise AND	na, na	$O(n/(qS))$	n	n	In-situ row copies
0x5	$A \cup \{x\}$	DB ∪ $\{x\}$	Set bit	na, na	$O(1)$	n	W	Random access
0x6	$A \setminus \{x\}$	DB \ $\{x\}$	Clear bit	na, na	$O(1)$	n	W	Random access

Table 4: Overview of SISA instructions, one row describes one specific set operation variant. Set elements are vertices (A, $B \subseteq V$, $x \in V$). "👍" means "yes". "na" means "not applicable". "ins" is a proposed instruction opcode. "S (Sorted)" indicates if an instruction assumes set representations of A and B to be sorted (thus two columns).

Figure 4 shows an SA and a DB built from the same vertex set. Then, it illustrates an example SISA graph representation where some neighborhoods are DBs and some are SAs.

6.2 High-Performance Set Operations

The second key challenge in SISA is how to apply set operations for highest performance. For this, we detail the algorithmic aspects, a summary is in Table 4. HW details (used PIM and a performance model) are discussed in Section 8. An overview of the structure of SISA is in Figure 3.

Set Intersection $A \cap B$ is a key operation in SISA, because our analysis illustrates that it is used in essentially all considered graph algorithms. We now briefly discuss the most relevant variants of ∩, a summary is in Figure 4.

- **SA [sorted]** $A \cap$ **SA [sorted]** B The intersection of two sorted SAs is commonly used when processing two neighborhoods. It comes in two "flavors". If A and B have similar sizes ($|A| \approx |B|$), one prefers the **merge** scheme where one simply iterates through

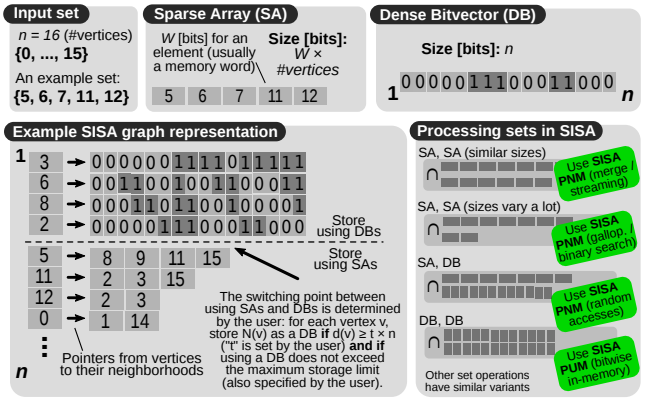


Figure 4: SISA representations of sets and graphs, and processing SISA sets.

A and B, identifying common elements (time $O(|A| + |B|)$). If one set is much smaller than the other ($|A| \ll |B|$), it is better to use the **galloping** scheme [1], in which one iterates over the elements of a smaller set and uses a binary search to check if each element is in the bigger set (time $O(|A| \log |B|)$). SISA offers both variants, and a variant that automatically selects the best variant with a performance model (described in § 8.2).

- **SA [unsorted or sorted]** $A \cap$ **DB** B Iterate over A ($O(|A|)$) and check if each element is in B ($O(1)$). This variant is often used to intersect a neighborhood with an auxiliary set represented as a bitvector, for example $X \cap N(v)$ in Listing 1.
- **DB** $A \cap$ **DB** B Apply bitwise AND over both input DBs (they both have sizes of n bits, giving $O(n/C)$ time, where C is the maximum chunk of bits that can be processed in $O(1)$ time using bit-level parallelism). This variant is used for example when intersecting two dense neighborhoods.

Set Union $A \cup B$**, Set Difference** $A \setminus B$ $A \setminus B$ and $A \cup B$ have variants similar to those for ∩, there are also corresponding merge and galloping variants.

Set Membership $x \in A$**, Set Cardinality** $|A|$ Set membership takes $O(|A|)$ time for an unsorted SA (linear scan), $O(\log |A|)$ time for a sorted SA (binary search), and $O(1)$ for a DB (a single access to verify if x-th bit is set). As for set cardinality, we keep $|A|$ for any set. This incurs only $O(1)$ storage overhead (per set) as well $O(1)$ time overhead needed to update the size, but it enables $O(1)$ time to resolve any set cardinality operation. Finally, SISA provides dedicated instructions for computing cardinalities of the results of set operations, for example $|A \cap B|$. This enables speedups as SISA avoids creating any intermediate structures needed for keeping the results of operations such as intersection.

Adding & Removing Elements Auxiliary sets often grow and shrink by one element. Both add and remove straightforwardly take $O(1)$ time for a DB (setting or zeroing a corresponding bit) and $O(|A|)$ for an SA (moving data for a sorted SA). Thus, in general, we advocate using DBs for auxiliary sets; the size is n bits

6.3 Additional Details of SISA Design

We detail several aspects of SISA's design; cf. Figure 3.

Labeled Graphs As a baseline, we propose to use a sparse array to maintain labels, indexed by vertex IDs, similarly to other

	Triangle Counting [146]	k-Clique Listing [44]	k-Star-Clique Listing [79]	Maximal Cliques Listing [26, 51]	Link Prediction†	Link Prediction‡	Link Prediction§	Jarvis-Patrick Clustering [81]
SISA + merging intersection	$O(mc)$★	$O(km(c/2)^{k-2})$★	$O(k^2m(c/2)^{k-1})$★	$O(cdn3^{c/3})$	$O(md)$	$O(n^2 \quad md)$	$O(n^2)$★	$O(md)$
SISA + galloping intersection	$O(mc\log c)$	$O(km(c/2)^{k-2}\log c)$	$O(k^2m(c/2)^{k-1}\log c)$	$O(cn3^{c/3})$★	$O(mc\log c)$★	$O(n^2 \quad mc\log c)$★	$O(n^2)$★	$O(mc\log d)$★

Table 5: The impact of set intersection schemes (merging vs. galloping) on the runtime of graph mining algorithms.. "★" means that a given SISA variant matches asymptotically the best known non-set-centric baseline, referenced in the top row. k, c, and d denote the size of the mined pattern, the graph degeneracy (a popular measure of graph sparsity) and the maximum vertex degree, respectively (other symbols are described in Section 2). Link prediction complexities are valid for the following vertex similarity measures: †Jaccard, Overlap, Adamic Adar, Resource Allocation, Common Neighbors; ‡Total Neighbors; §Preferential Attachment [98, 121].

works [41]. This form benefits from SISA-PNM. The SISA user can also implement labels with a one-hot encoding and use bit vectors. This would harness SISA-PUM.

SISA Instructions SISA offers instructions that package the described set operations in all the considered variants, including instructions that automatically select merge or galloping set algorithms (cf. § 6.2). Finally, SISA also provides instructions for creating and deleting sets.

Programming Interface (Set Iterators & Wrappers) For programmability, SISA offers a thin software layer on top of high-level instructions that consists of abstractions and wrappers. In the former, we provide an opaque type `Set` that is a reference to a SISA set; this enables using C++ iterators over sets, see left side of Figure 3. In the latter, SISA provides functions that directly map to SISA set instructions.

RISC-V Compliant Encoding SISA can be integrated with the RISC-V ISA [166]. To enable modularity and flexibility, SISA's new instructions are encoded using the custom opcode set [165]. We encode the opcode and functionality of custom RISC-V instructions using bits [31..25] and [6..0], see Figure 5. The former represent the different SISA instructions (up to 128). The latter are set to 0x16 to represent the custom characteristic of the instruction. Fields `rs1`, `rs2`, and `rd` indicate registers with IDs of input sets and the output set, respectively. In Table 4, we assign ISA codes (bits [31..25]) to respective instructions. The number of SISA instructions is less than 20, leaving space for potential new variants.

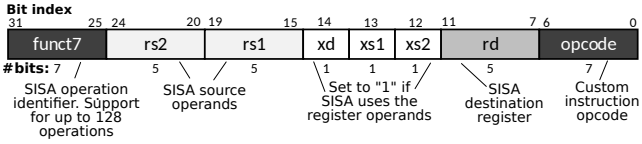


Figure 5: Encoding of SISA instructions.

7 THEORETICAL ANALYSIS

We now show that SISA-enhanced algorithms are *theoretically efficient*, i.e., their time complexities match those of hand-tuned graph mining algorithms. This is enabled by SISA's ability to control used set representations and set operations. To show this, we analyze how varying a used set intersection variant (merge vs. galloping) impacts the runtime of set-centric algorithms, see Table 5. We focus on intersection as it is prevalent in considered algorithms. Crucially, *all set-centric variants are able to match the competitive time complexities of considered tuned graph mining algorithms.*

8 HARDWARE IMPLEMENTATION

SISA-PUM First, the intersection, union, and difference of sets represented as DBs are processed with SISA-PUM that relies on in-situ DRAM bulk bitwise schemes. For concreteness, we pick Ambit [141], a recent design that enables energy-efficient bulk bitwise operations fully inside DRAM, by small extensions to the DRAM circuitry but without any changes to the DRAM interface. However, SISA is generic and other designs could also be used (e.g., ELP2IM [168], DRISA [100], ComputeDRAM [53], PCM (Pinatubo) [101]). The key extension in Ambit (for in-situ processing) is to modify a decoder for three selected DRAM rows (that share the same set of sense amplifiers) in such a way that one amplifier connects directly to three DRAM cells. This enables logical AND and OR over two of such three rows, immediately computing the result in the third row (NOT is provided by including a single row of dual-contact DRAM cells [141]). *Importantly for SISA-PUM*, only three selected designated DRAM rows (per single DRAM subarray) are modified this way. Whenever the running code requests an in-situ memory operation, Ambit uses a recent RowClone technology [140] to copy (also in-situ) the rows that store input sets to these two designated rows, compute the result in-situ, and again use RowClone to copy the result to the destination (unmodified) DRAM row. Now, SISA-PUM uses Ambit's execution model and interface without any modifications: set intersection and union are processed with an in-situ AND and OR, respectively. Set difference is processed using set intersection, along with the well-known set algebra rule: $A \setminus B = A \cap B'$ [82].

SISA-PNM A set operation with no bulk bitwise processing uses SISA-PNM that relies on high bandwidth between processing units and DRAM (as in UPMEM [96], HMC [83], or Tesseract [6]). Adding or removing an element from a set stored as a DB ($A \cup \{x\}, A \setminus \{x\}$) is conducted with a single DRAM access to a specific memory cell. Other set operations on SAs that employ streaming or random accesses are also executed using small in-order cores.

8.1 SCU & Automatizing SISA Decisions

We use a small SISA Control Unit (SCU), cf. Section 3, to automatically decide on (1) selecting the PNM or PUM execution, and (2) merge or galloping. Once the host core decodes a SISA instruction, it passes it to the SCU. The SCU further decodes this instruction, and picks either PNM or PUM to execute the instruction. For deployment, SCU could either be added to the CPU or to the DRAM circuitry (see the feasibility discussion later in this section), or – to avoid any HW modifications – it can also be emulated by a single designated in-order logic layer core. SCU does not implement any complex logic (e.g., dynamic set modifications), it only decides on variants of schemes to execute.

SISA-PUM & SISA-PNM First, SCU decides whether to use SISA-PUM or SISA-PNM for given two sets. This decision is simple and is based on how sets are represented (this information is stored in a simple in-memory SM ("set metadata") structure and possibly cached in SCU's cache).

Variants of Set Operations Second, SCU automatically detects if it is best to use merge or galloping, and processes input sets using the corresponding variant. This decision is guided by our performance models.

8.2 Performance Models for Set Operations

The runtime of each SISA instruction variant is dominated by either streaming or random accesses.

Streaming takes place when two sets A and B stored as SAs are processed using merging. We model the runtime as $l_M + W \cdot \max\{|A|, |B|\} \cdot \min\{b_M, b_L\}$. l_M and b_M are latency and bandwidth of accessing DRAM, and b_L is bandwidth between cores. The model conservatively assumes that A and B may be located in memory locations attached to different cores (e.g., in different vaults), and thus (1) the overall bandwidth is bottlenecked by $\min\{b_M, b_L\}$, and (2) we can use $\max\{|A|, |B|\}$ as A and B are streamed in parallel.

To model **random accesses**, we simply count the number of performed operations and multiply it by the memory access latency. This gives $l_M \cdot \min\{|A|, |B|\} \cdot \log(\max\{|A|, |B|\})$ for a binary search over the larger of input sets, used when processing two SAs with galloping. Then, a specific variant is **selected automatically** to minimize the predicted runtime. To **parametrize** these models, SISA needs (1) the sizes of processed sets, (2) their representation types, and (3) b_M, b_L, l_M. (1) and (2) are maintained in the metadata structure. (3) describe the execution environment and are thus identical for each set; they are stored directly in the SCU. We instantiate (3) to reflect logic layers in Tesseract [6].

8.3 Details of SISA Hardware

Life Cycle of a Set A set is allocated with a standard malloc, augmented with setting the appropriate set information in the set metadata (SM) structure. Loading, processing, and storing sets is conducted by the respective existing elements such as logic layer cores; the SCU is only responsible for selecting the appropriate instruction variant to be executed. Once a set is deleted, the standard free call is used, together with removing the respective entry from the SM structure.

Set Metadata SM forms a simple associative structure that holds constant amount of data per set (set representation, set size). The total SM size is $O(n)$ as there are n neighborhoods and a constant number of auxiliary sets. Thus, while we conservatively assume that SM is an in-memory structure, in practice it fits completely in cache or a small scratchpad. This is because many datasets processed by graph mining algorithms have small n, in the order of hundreds or thousands [132]. These graphs pose computational challenges, but these challenges come from high computational complexities (e.g., listing maximal cliques is NP-hard) or from relatively high edge counts m (as some vertices may have high degrees [132]), but *not* (or to a smaller extend) from n. Whenever the given SM information is not cached, there is a single additional memory access for one set operation. Each SM entry describing one set also contains the set location. Now, entries in the SM structure are indexed by set IDs. A set ID is returned by a function creating a set, cf. Figure 3. Set IDs and set creation (and destruction) calls are used by a developer analogously to pointers and malloc/free calls.

Caching Set Metadata Depending on how SISA HW is deployed, the SM information can be cached in either a small dedicated scratchpad or cache (if the SCU is implemented as an additional circuitry), or in the standard cache of a logic layer core (if the SCU is emulated by a such designated core).

SISA-PNM and SISA-PUM Together Ambit fully preserves the DRAM interface: the sets are always stored in standard DRAM rows, and moved to the designated rows *only* for bulk bitwise processing [141]. SISA-PNM accesses run on unmodified DRAM banks (the modifications in PNM are only related to the high bandwidth, and the SCU in SISA). Thus, SISA-PNM and -PUM are seamlessly used together.

Harnessing Parallelism First, bit-level parallelism is enabled by using Ambit's bulk bitwise operations: bits in a row are ANDed or ORed in parallel. Second, pairs of bitvectors placed in different subarrays (or, e.g., DRAM banks) can be processed in parallel. Third, processing pairs of sets stored as integer arrays in different vaults can also be parallelized. Here, SISA benefits from the same effect of bandwidth scalability as the Tesseract graph accelerator [6].

Managing Concurrency SISA relies on developers using established techniques (locks, lock-free protocols, general parallel programming principles [71] and libraries such as OpenMP [32]) to concurrently access the same set.

For **cache coherence in SISA-PUM**, we rely on mechanisms (provided by the memory controller) that flush dirty cache lines in source rows, and invalidate cache lines in destination rows. Existing schemes also rely on it, including Ambit [141], DMA accesses [40] and others [75, 140]. As in Ambit, SISA-PUM accesses are always row-wise, and thus we can also rely on Dirty-Block Index [139] and similar schemes for fast data flushing. Invalidations run in parallel with Ambit operations and thus do not incur overheads.

Memory Layout and Storage of Sets We ensure that storing SISA sets is feasible (i.e., a maximum-size neighborhood, represented as SA or DB, fits into a single vault).

8.4 SISA Hardware Cost and Feasibility

We also briefly discuss the hardware cost. First, the needed **DRAM chip modifications** are minimal and identical to those already discussed in Ambit. Second, as the **logic to be implemented in SCU** is straightforward decision making on what instruction variant to use, its costs are not prohibitive, as shown by many designs proposed in the past, for example in HyVE [76] (a hybrid vertex-edge memory hierarchy that uses ReRAM and DRAM) or in GraphH [43] (an accelerator that combines HMC with SRAM). Third, the code of all SISA instructions is also straightforward: a simple binary search (galloping), merging of two arrays (merge), or setting/clearing a DRAM cell (set element add/remove). Thus, they can be trivially deployed in in-order cores in the logic layer of 3D stacked DRAM, as shown by other designs [43].

9 EVALUATION

We illustrate example performance advantages from SISA.

9.1 Methodology, Setup, Parameters

Simulation Infrastructure We use Sniper [70] with the Pin frontend [106]. Sniper is a popular cycle-level simulator used in many

works proposing various architectural extensions for both CPUs and memory subsystem [116, 160].

SISA Implementation We simulate the SISA HW design and the ISA, instrumenting the code so that the simulation toolchain can distinguish between SISA and non-SISA instructions. To model each component of SISA, we add the respective set instructions and simulate the SCU (a small fixed delay), the cache in SCU (with the LRU policy), the SM structure (random memory accesses whenever the SCU cache is not hit), and the execution of all used set operations by appropriate delays in the simulation execution. For operations based on streaming and random memory accesses, we use the performance models described in § 8.2. To simulate SISA-PUM, we model a run-time of in-situ operations with a delay $l_M + l_I \cdot \lceil n/(qS) \rceil$; l_M is the latency to access DRAM (to initiate the operation) and l_I is the latency execute one in-situ instruction. $\lceil n/(qR) \rceil$ models a scenario when the bitvector size n exceeds the size of all DRAM rows that can be processed in parallel. q is the count of rows within a bank that can be used in parallel and R is the size of one row.

SISA Platform & Parameters For concreteness, we set the platform for executing SISA instructions to match Tesseract [6] (for SISA-PNM) and Ambit [141] (for SISA-PUM). The former has simple in-order cores (1 core/vault in its logic layer) with 32 KB L1 instruction/data caches, no L2, 16 8GB HMCs (128 GB in total), 32 vaults/cube, 16 banks/vault. Each vault offers 16 GB/s of memory bandwidth to its core. Thus, we assume scalable bandwidth as proposed by Tesseract: using more vaults increases the total memory bandwidth. We set the DRAM row rank size to 8 KB, following Ambit [141]. Next, we set the parameter $t \in [0; 1]$ (that controls the bias towards using DBs or SAs to store neighborhoods) to 0.4 (i.e., 40% of neighborhoods are stored as DBs); we also analyze other values. We ensure that the total storage used for neighborhoods does not exceed the size of the simple CSR graph storage by more than 10%. Finally, we set the size of SISA SCU's cache to be 32 KB (matching Tesseract's L1).

Platform for non-SISA Instructions & Baselines For any non-SISA instructions and baselines, we use a high-performance Out-of-Order manycore CPU. Each core has a 128-entry instruction window, a branch predictor, 32 KB L1 instruction/data caches, a 256 KB L2 cache. All cores share an 8 MB L3 cache. There is also a four-way associative 64-entry D-TLB, a 128-entry I-TLB, and a 512-entry S-TLB. For fair comparison, *we also use bandwidth scalability in this configuration, i.e., we increase the memory bandwidth with the number of cores, matching it with that of SISA-PNM.*

Considered Mining Problems The graph mining problems we consider are clustering with the Jaccard (`cl-jac`), overlap (`cl-ovr`), and total neighbors (`cl-tot`) coefficients, listing k-cliques (`kcc-`k, $k \in \{4, 5, 6\}$), k-clique-stars (`ksc-`k, $k \in \{4, 5, 6\}$), maximal cliques (`mc`), triangles (`tc`), and subgraph isomorphism (`si-`k`s` for k-stars).

Comparison Targets: Hand-Tuned Algorithms Our most important (the most challenging to outperform) baselines are hand-optimized parallel algorithms for each graph mining problem. Specifically, we use a tuned version from the GAP Benchmark Suite [14] for `tc`, Eppstein's version of BK for `mc` [51], Danisch' scheme for `kcc-`k [44], enhanced Jabbour's scheme for `ksc-`k [79], parallel VF2 for `si-`k`s` [41], and `cl-jac` based on counting triangles in the GAP suite [14]. All used baselines have competitive work and depth complexities, cf. Table 5. For fair comparison, all baselines benefit from the high bandwidth of PIM. We consider algorithms that do not explicitly use set algebra (denoted with `_non-set`) and their set-centric variants (denoted with `_set-based`).

Comparison Targets: Pattern Matching Frameworks SISA and its underlying paradigm do not aim to outperform specific accelerators but complement or reinforce them, by offering a novel set-centric paradigm and building blocks. Thus, we focus on comparing to the fundamental paradigms / algebras that underlie these accelerators: neighborhood expansion for pattern matching implemented in Peregrine [80] (which represents GRAMER [176]) and relational algebra implemented in RStream [161] (which represents TrieJax [89]). We stress that, while we consider these baselines for completeness, we focus on comparing to (much faster) hand-tuned parallel algorithms for solving specific problems.

Graphs We select different input datasets (Table 6) from Network Repository [133], considering biological (`bio-`), interaction (`int-`), brain (`bn-`), economics (`econ-`), social (`soc-`), scientific-computing (`sc-`), discrete-math (`dimacs-`), and wiktionary (`edit-`) networks. We pick graphs with different structural properties (low/high density, small/large maximum degree, low/high degree distribution skew, etc.).

Biological. Gene functional associations: (*bio-SC-GT*, 1.7K, 34K), (*bio-CE-PG*, 1.8K, 48K), (*bio-DM-CX*, 4K, 77K), (*bio-DR-CX*, 3.2K, 85K), (*bio-HS-LC*, 4.2K, 39K), (*bio-SC-HT*, 2K, 63K), (*bio-WormNetB3*, 2.4K, 79K). Human gene regulatory network: (*bio-humanGene*, 14K, 9M) **(large)**, (*bio-mouseGene*, 45K, 14.5M) **(large)**.
Interaction. Animal networks: (*int-antCol3-d1*, 161, 11.1K), (*int-antCol5-d1*, 153, 9K), (*int-antCol6-d2*, 165, 10.2K), (*intD-antCol4*, 134, 5K). Human contact network: (*int-HosWardProx*, 1.8k, 1.4k). Users-rate-users: (*int-dating*, 169K, 17.3M) **(large)**, (*edit-enwiktionary*, 2.1M, 5.5M) **(large)**.
Brain. (*bn-flyMedulla*, 1.8K, 8.9K), (*bn-mouse*, 1.1K, 90.8K).
Economic. (*econ-beacxc*, 498, 42K), (*econ-beaflw*, 508, 44.9K), (*econ-mbeacxc*, 493, 41.6K), (*econ-orani678*, 2.5K, 86.8K).
Social. Facebook: (*soc-fbMsg*, 1.9k, 13.8k). Orkut: (3.1M, 117M) **(large)**,
Scientific computing. (*sc-pwtk*, 217.9K, 5.6M) **(large)**,
Discrete math. (*dimacs-c500-9*, 501, 112K),

Table 6: Considered graphs[133]. **For each graph, we show its "(#vertices, #edges)".**

Tackling Long Simulation Runtimes Most benchmarks use relatively small graphs because (1) we run cycle accurate simulations, tracing all memory accesses, which is very time-consuming, and (2) the considered algorithms are computationally hard and even software codes use graphs much smaller than those used with algorithms such as PageRank [44, 51]. Yet, even this is often not enough to enable finishing simulations of algorithms such as Bron-Kerbosch. Thus, we usually also pre-specify a number of graph patterns to be found. Past work analogously handled long simulations graph algorithms [6] such as PageRank (limiting #iteration).

Performance Measures & Summaries: We focus on plain runtimes as recommended for parallel codes [73] as speedup may be misleading because it is higher on unoptimized baselines. However, for overview, we also *summarize speedups* (following [73]), i.e., we provide (1) speedups of *average runtimes* ("speedup-of-avgs"), and (2) *geometric means of speedups* of all data points ("avg-of-speedups").

9.2 Discussion of Results

Comparison to Hand-Tuned Algorithms We first analyze runtimes with all available cores, comparing SISA set-centric variants to non-set-based and set-based hand-tuned parallel baselines that

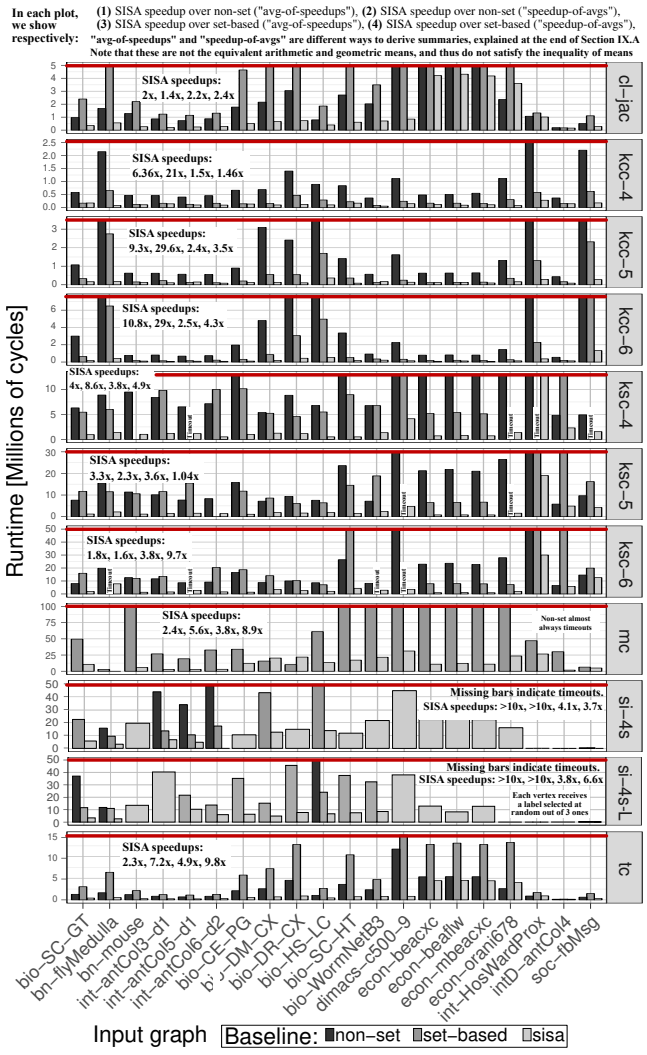


Figure 6: Run-times with full parallelism. The red line cuts off of long simulation runtimes for readability (the bars reaching the line have much larger runtimes). No bar indicates the timeout of the respective baseline (>24h). The results for `cl-jac` (clustering based on the Jaccard coefficient) are very similar to those that use other coefficients and for link prediction as well as vertex similarity. All 32 cores are used. Acronyms are stated in "Comparison Targets: Hand-Tuned Algorithms".

all benefit from high-bandwidth storage. The results are in Figure 6. SISA is almost always the fastest by a large margin of at least 2×, often more than 10× (than `non-set` schemes). The differences vary depending on the processed graphs and the considered problem. Gains are usually larger on graphs with large maximum degrees, such as brain graphs, where SISA-PUM is used more often to directly process sets inside DRAM, reducing the latency. Such graphs are prevalent in many computational domains [133], and this is the case for the majority of considered datasets.

Algorithmic vs. Architectural Speedups We also observe speedups from using only set-centric formulations (over non-set-based variants). Namely, speedups of "`_set-based`" schemes over the "`_non-set`" ones indicate gains from purely *algorithmic* (set-centric) changes, while speedups of "`_sisa`" schemes over the "`_set-based`" indicate gains only from *architectural* changes (i.e., from using PIM). First, the differences between `_set-based` and `_non-set` heavily depend on the targeted mining algorithm. These speedups are particularly visible for more complex algorithms such as `mc`, with multiple nested loops and/or recursion. Packaging different parts of such algorithms into, e.g., set intersections, and being able to control the used operation variant (e.g., merging based on streaming) helps to utilize features such as high sequential bandwidth. Contrarily, for certain simpler schemes such as clustering, the very tuned `_non-set` baseline outperforms `_set-based` (while still falling short of `_sisa`). Second, the difference between `_set-based` and `_sisa` depend more on the used graph. Here, in many cases, `_sisa` is only marginally faster than `_set-based`, because the graph structure (e.g., sizes of neighborhoods) favor using SAs rather than DBs, diminishing benefits from SISA-PUM (e.g., for `econ-` graphs) and equalizing the differences because both `_set-based` and `_non-set` take advantage from the high bandwidth setting. In other cases (e.g., `bio-HS-LC`), more vertices have large enough degrees to benefit from DBs and low latencies of SISA-PUM.

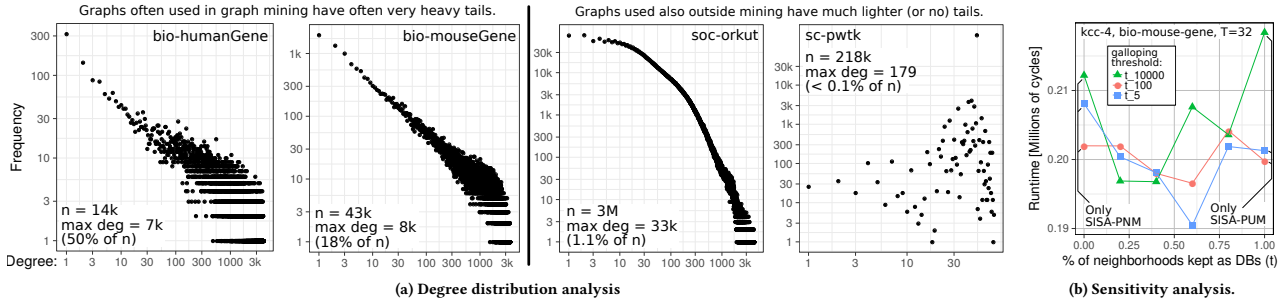
Labels We also analyze *labeled* SI. Most often, labeled graphs are faster to process. Despite more memory accesses, the labels form additional constraints, which eliminates some recursive calls earlier, resulting in performance gains.

Scalability We also analyze how run-times change when varying numbers of threads T, for a fixed graph size ("strong scaling"), and when increasing T proportionally to the graph size ("weak scalability"). To fix the used graph model, we use Kronecker graphs [99] and we vary the number of edges/vertex. SISA maintains its speedups, but they become less distinctive when T is small. This is expected because fewer threads exert less pressure on the memory subsystem, and there is overall smaller potential from using PIM in SISA.

Large Graphs We execute SISA on several large graphs, see Figure 8. Runtime benefits from SISA and the set-centric formulations are similar to those in smaller graphs in Figure 6. The only two graphs where SISA and non-SISA set baselines are comparable, are sc-pwtk and soc-orkut. This is because these networks, due to their origin (social and scientific) do not have large cliques or very dense clusters (unlike, e.g., genome graphs), somewhat lowering SISA benefits.

Comparison to Other Paradigms We compare SISA set-centric algorithms to neighborhood expansion and relational algebra paradigms, representing frameworks such as Peregrine or RStream, and accelerators such as GRAMER or TrieJax. Peregrine is able to express only listing k-cliques and subgraph isomorphism, and maximal clique listing in a limited way (i.e., it does not offer a native scheme for MC and we implemented it by iterating over possible clique sizes and listing maximal cliques of each size). RStream can only find k-cliques. In each case, SISA baselines are *much* faster: 10-100× than Peregrine (and more than 1,000× for `mc` due to Peregrine's inability to natively support `mc`), and more than 100× for RStream. This is because the underlying paradigms focus on programmability in the first place, sacrificing performance, while in SISA we start with tuned graph algorithms and only then restructure them with the set-centric paradigm.

Sensitivity Analysis & Design Exploration We investigate the impact from varying SISA parameters.



(a) Degree distribution analysis (b) Sensitivity analysis.

Figure 7: Figure 7a: Differences between degree distributions in graphs used mostly in graph mining and the ones used also outside graph mining (on the right). **Figure 7b:** Sensitivity analysis: the percentage of neighborhoods stored as dense bitvectors vs. different thresholds for using the galloping or the merging intersection.

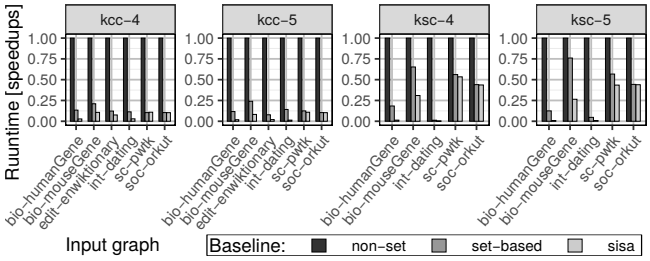


Figure 8: Run-times for large graphs. 8 cores are used.

SCU cache Not using the SCU cache lowers performance by ≈1.5× for $T = 1$ and ≈0.05-0.1× for $T = 32$. The lower performance for high T is because, with more threads executing set operations, it becomes more difficult to ensure high hit ratio. Overall, the behavior of the SCU cache is similar to that of other such units such as L1, including varying cache parameters such as size.

PNM vs. PUM & Sparse/Dense Neighborhoods PNM and PUM are synergistic in SISA. PNM cores handle sparse neighborhoods and SAs well, as they offer low latency and bandwidth proportionality. PUM is well-suited for large neighborhoods stored as DBs (common in considered graphs due to their degree distribution skews). Yet, SISA-PUM adds overheads when using it for sparse sets due to low utilization of very sparse rows. Thus, it is relevant to not choose the DB bias parameter to be too high. We find that 0.4 works well for most processed graphs. We illustrate this in Figure 7b, where we analyze how the performance changes when varying the fraction of largest neighborhoods stored as DBs. Smallest and largest fractions that correspond to *using only SISA-PNM or only SISA-PUM*, while technically feasible, give slowest runtimes. We also vary the "galloping threshold", i.e., the relative difference between two sets that causes the set operation to switch to the galloping variant. For example, the value of 5 indicates that galloping is used if any of the two sets is at least 5× larger than the other one. While this threshold influences performance, the general pattern stays the same.

We also analyze the **impact from degree distributions of datasets**, see Figure 7a. Graphs often used in graph mining, such as biological networks, that SISA focuses on, have often *very* heavy tails. This implies *many large neighborhoods and very dense large clusters, benefiting from SISA-PUM*. For example, the human genome graph has many vertices connected to more than 30% of all other vertices. Other graphs such as social networks have *much lighter tails*, cf. soc-orkut and sc-pwtk in Figure 7a. This is because these networks, due to their origin (social, scientific) do not have large cliques or very dense clusters. Such graphs benefit less from SISA-PUM. Still, using SISA-PNM enables high performance, outperforming tuned non-set-based baselines, cf. Figure 8.

Load balancing Figure 9a illustrates total fractions of time during which each parallel thread is stalled when executing a given algorithm. SISA stall times are low because its design implicitly tackles two types of load imbalance. First, SISA's performance models enable adaptive selection of the best variant of a set algorithm to be executed for any two sets. This minimizes load imbalance from processing two sizes that differ a lot in sizes. Second, load imbalance due to processing imbalanced *pairs* of sets (i.e., two very small and two very large sets) is alleviated by the fact that very large pairs of sets are processed with very fast SISA-PUM.

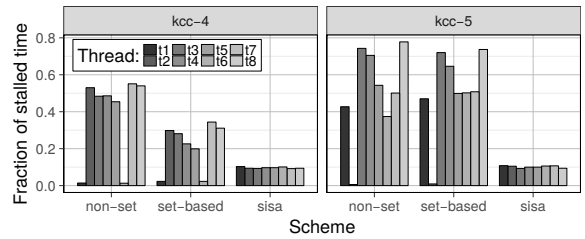
SCU cache: shared vs. private We also explore sharing the SCU cache among all the cores. While possibly increasing the hit rate, a single shared cache has higher access latency. This has a small (<1%) yet noticeable slowdown effect in our simulations. A potential remedy would be to include multiple SCU cache levels. To keep the core logic simple, we do not explore it further, and leave it for future work.

We also show that the reduced simulation runtimes do not artificially eliminate load imbalance. We gather traces of executed set operations in full vs. partial simulation executions, and we plot histograms of the sizes of processed sets, see Figure 9b. In both types of executions, we encounter large sets which are the primary source of load imbalance.

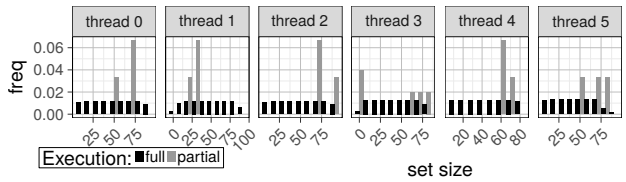
SISA Limitations For some graphs with small maximum degrees (e.g., `soc-fbMsg`) in Figure 6, SISA speedups are smaller, or even (in the extreme cases) result in slowdowns. This is because the benefits from SISA-PUM, or from the automatic selection of the most beneficial set operation variant, are out-weighted by having to process too many large bitvectors. This effect rare, and it can be alleviated by reducing the number of neighborhoods stored as DBs. In this case, the performance of SISA variants gradually converges towards that of standard CSR based set-centric algorithms. We plan on addressing it with advanced bitvector representations.

10 RELATED WORK

Related graph processing paradigms (Table 1) and software efforts are described in Section 1 [18, 18, 21, 107, 135]. We now briefly



(a) Total amounts of stall times of different 8 parallel threads.



(b) Histograms of sizes of processed sets of full vs. partial executions, for 6 parallel threads (the remainder of threads behave similarly). Graph: int-antCol3-d1. Problem: kcc-4.

Figure 9: Load balancing analysis.

Reference / Accelerator	Prob.	Key memory mechanism	Pattern M.						Learning			"Low-c."			is	xl	ab
			mc	kc	ds	si	vs	ls	lp	cl	av	bf	pr	cc			
[Pi] GaaS-X [31]	SpMV	**[e]** CAM/MAC	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[Pi] GraphiDe [10]	low-c	**[e]** DRAM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[Pc] Spara [181]	ver-c	**[e]** ReRAM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[Pc] GraphQ [183]	ver-c	**[e]** HMC	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[Pc] GraphS [11]	low-c	**[e]** SOT-MRAM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[Pc] RAGra [77]	ver-c	**[e]** 3D ReRAM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[Pc] GRAM [182]	ver-c	**[e]** ReRAM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[Pc] GraphR [149]	SpMV	**[e]** ReRAM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[Pc] GraphP [177]	ver-c	**[e]** HMC	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[Pc] Tesseract [6]	low-c	**[e]** HMC	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[Pc] PIM-Enabled [7]	low-c	**[e]** HMC	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[Pc] Gao et al. [54]	low-c	3D DRAM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[A] IntersectX [127]	pattern m.	**[e]** cache	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[A] Gramer [176]	pattern m.	DRAM, cache	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[A] TrieJax [89]	joins	DRAM, LLC	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[A] HyGCN [174]	GCN	eDRAM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[A] Outerspace [124]	SpMSpM	HBM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[A] Domino [169]	low-c	on-chip buffers	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[A] GraphPIM [120]	low-c	**[e]** HMC	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[A] Graphicionado [65]	ver-c	**[e]** eDRAM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[A] Ozdal et al. [123]	ver-c	**[e]** caches	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[A+Pc] EnGN [68]	GNN	**[e]** HBM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[A+Pc] OMEGA [2]	low-c	**[e]** Scratchpads	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[A+Pc+M] GraphH [43]	ver-c	**[e]** HMC	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[F+Pc] HRL [55]	ver-c	**[e]** 3D DRAM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[Pc+Pi] SISA [This work]	Graph mining	PIM	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖

Table 7: Comparison of SISA to graph-related accelerators, focusing on supported graph mining problems and offered architecture elements. "■": Support / significant focus. "◧": Partial support / some focus. "✖": no support / no focus. **Addressed problems:** see Table 1. **Graph problems and algorithms: as in Table 1**. **Considered architecture and stack elements:** "**is**": an ISA, or its extensions, "**xl**": a cross-layer design, "**ab**": a programming paradigm **Classes of accelerators: [Pi]**: *in-situ* PIM, **[Pc]**: *near memory* PIM (e.g., logic layers), **[A]**: ASIC, FPGA designs and little related memory hierarchy enhancements are excluded. **[e]** focus on extensions and modifications to the established (already proposed) HW technology, Note that the generality of SISA comes from harnessing all basic set algebra operations.

summarize other related areas. First, we conducted an exhaustive analysis of existing hardware accelerators as well as ISA designs for graph processing, see Table 7. The analysis indicates that SISA offers the only hardware acceleration for a broad family of problems such as maximal clique listing or clustering. The closest designs [89, 127, 176] only focus on selected pattern matching problems. Works orthogonal to SISA include HW accelerated dynamic (time-evolving) graph analytics [27, 28, 69], or external memory HW accelerated graph processing [47, 87, 110]. *One could use the latter as a SISA backend for external memory set instructions; we leave details for future work.*

Sets are used in different graph algorithms, to simplify operations on selected data structures [21, 26, 92, 114, 125, 138, 143]. For example, the BFS frontier can be modeled as a set. Here, SISA's main contribution is *not* to simply use set notation. Instead, from the algorithmic perspective, SISA is the first design that (1) uses set operations as the *primary building blocks*, which break down complex graph mining algorithms into simple units of parallel execution, and (2) identifies the "appropriate" set operations (i.e., operations that are easily accelerated with PIM) and reformulates selected algorithms so that they use such operations, cf. Table 2.

SISA vs. AutoMine [112] AutoMine [112] uses set operations to express finding graph patterns. It focuses on automatic compilation of set schedules into efficient code. This part is orthogonal to our work and AutoMine *could easily be combined with SISA* to, for example, generate code based on SISA's set-centric formulations. Note that SISA's set formulations are superior to those of AutoMine, because SISA (1) supports *all* set operations, including *non anti-monotonic ones* (not just intersection and difference), (2) it expresses whole algorithms with the set building blocks (not just pattern generation schedules), and (3) it targets broad graph mining (not just pattern matching).

SISA shows how to seamlessly integrate PUM and PNM capabilities in a single system. They work synergistically and produce significantly better results than working separately.

11 DISCUSSION AND CONCLUSION

We develop the first hardware acceleration for broad graph mining. First, we offer a set-centric programming paradigm, where one identifies and exposes set operations in graph mining algorithms. This enables competitive time complexities and succinct formulations. We support labeled graphs and non anti-monotonic set operations [89, 112, 127, 176].

Second, the set-centric algorithms are mapped to SISA, a small yet expressive "set-centric" ISA extension for graph mining. SISA could be extended with CISC-style set instructions that accept multiple arguments (e.g., $A_1 \cap ... \cap A_l$) to facilitate optimizations such as vectorization with loop unrolling. Due to the generality of set algebra, SISA can be used for problems beyond graph mining. Third, while we pick in-situ and logic layer PIM for hardware acceleration, SISA's set algebra interface could easily use other hardware backends, for example a GPU backend for fast SIMD-based set intersections [67], FPGAs [22], or even execution in caches [3, 119]. We leave this for future work.

Finally, our cross-layer architecture could also be extended in other directions, for example by providing compiler support for generating SISA programs from set-centric formulations. Here, one could use, e.g., AutoMine's [112] compiler generated schedules as input to some SISA programs.

ACKNOWLEDGMENTS

We thank Mark Klein, Hussein Harake, Colin McMurtrie, Angelo Mangili, and the whole CSCS team granting access to the Ault and Daint machines, and for their excellent technical support. We thank Timo Schneider for his immense help with computing infrastructure at SPCL. This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880).

REFERENCES

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–44.

[2] 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)*. IEEE, 134–145.

[3] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 481–492.

[4] Charu C Aggarwal and Haixun Wang. 2010. *Managing and mining graph data*. Vol. 40. Springer.

[5] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, Vol. 1215. Citeseer, 487–499.

[6] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*.

[7] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 336–348.

[8] Mohammad Al Hasan, Vineet Chaoji, Saeed Salem, and Mohammed Zaki. 2006. Link prediction using supervised learning. In *SDM06: workshop on link analysis, counter-terrorism and security*.

[9] Mohammad Al Hasan and Mohammed J Zaki. 2011. A survey of link prediction in social networks. In *Social network data analytics*. Springer, 243–275.

[10] Shaahin Angizi and Deliang Fan. 2019. Graphide: A graph processing accelerator leveraging in-dram-computing. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. 45–50.

[11] Shaahin Angizi, Jiao Sun, Wei Zhang, and Deliang Fan. 2019. GraphS: A graph processing accelerator leveraging SOT-MRAM. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 378–383.

[12] Omar Batarfi, Radwa El Shawi, Ayman G Fayoumi, Reza Nouri, Ahmed Barnawi, and Sherif Sakr. 2015. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing* 18, 3 (2015), 1189–1213.

[13] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261* (2018).

[14] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[15] Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, and Torsten Hoefler. 2019. A modular benchmarking infrastructure for high-performance and reproducible deep learning. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 66–77.

[16] Maciej Besta, Armon Carigiet, Zur Vonarburg-Shmaria, Kacper Janda, Lukas Gianinazzi, and Torsten Hoefler. 2020. High-performance parallel graph coloring with strong guarantees on work, depth, and quality. *arXiv preprint arXiv:2008.11321* (2020).

[17] Maciej Besta, Raphael Grob, Cesare Miglioli, Nicola Bernold, Grzegorz Kwasniewski, Gabriel Gjini, Raghavendra Kanakagiri, Saleh Ashkboos, Lukas Gianinazzi, Nikoli Dryden, et al. 2021. Motif Prediction with Graph Neural Networks. *arXiv preprint arXiv:2106.00761* (2021).

[18] Maciej Besta and Torsten Hoefler. 2015. Accelerating irregular computations with hardware transactional memory and active messages. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. 161–172.

[19] Maciej Besta, Raghavendra Kanakagiri, Harun Mustafa, Mikhail Karasikov, Gunnar Rätsch, Torsten Hoefler, and Edgar Solomonik. 2020. Communication-efficient jaccard similarity for high-performance distributed genome comparisons. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1122–1132.

[20] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 32–41.

[21] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 93–104.

[22] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. 2019. Graph Processing on FPGAs: Taxonomy, Survey, Challenges. *arXiv preprint arXiv:1903.06697* (2019).

[23] Maciej Besta, Zur Vonarburg-Shmaria, Yannick Schaffner, Leonardo Schwarz, Grzegorz Kwasniewski, Lukas Gianinazzi, Jakub Beranek, Kacper Janda, Tobias Holenstein, Sebastian Leisinger, et al. 2021. GraphMineSuite: Enabling High-Performance and Programmable Graph Mining Algorithms with Set Algebra. *VLDB* (2021).

[24] Guy E. Blelloch and Bruce M. Maggs. 2010. *Parallel Algorithms* (2 ed.). Chapman & Hall/CRC, 25.

[25] Otakar Boruvka. 1926. O jistém problému minimálním. (1926).

[26] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM* 16, 9 (1973), 575–577.

[27] Lázaro Bustio, René Cumplido, Raudel Hernández, José M Bande, and Claudia Feregrino. 2015. Frequent itemsets mining in data streams using reconfigurable hardware. In *International Workshop on New Frontiers in Mining Complex Patterns*. Springer, 32–45.

[28] Lázaro Bustio-Martínez, René Cumplido, Martín Letras-Luna, Claudia Feregrino Uribe, Raudel Hernández-León, and José M Bande-Serrano. 2017. Approximate frequent itemsets mining on data streams using hashing and lexicographie order in hardware. In *2017 IEEE 8th Latin American Symposium on Circuits & Systems (LASCAS)*. IEEE, 1–4.

[29] Frédéric Cazals and Chinmay Karande. 2008. A note on the problem of reporting maximal cliques. *Theoretical Computer Science* 407, 1-3 (2008), 564–568.

[30] Deepayan Chakrabarti and Christos Faloutsos. 2006. Graph mining: Laws, generators, and algorithms. *ACM computing surveys (CSUR)* 38, 1 (2006), 2.

[31] Nagadastagiri Challapalle, Sahithi Rampalli, Linghao Song, Nandhini Chandramoorthy, Karthik Swaminathan, John Sampson, Yiran Chen, and Vijaykrishnan Narayanan. 2020. GaaS-X: Graph Analytics Accelerator Supporting Sparse Data Representation using Crossbar Architectures. *ISCA* (2020).

[32] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan kaufmann.

[33] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 32.

[34] Langshi Chen, Jiayu Li, Ariful Azad, Lei Jiang, Madhav Marathe, Anil Vullikanti, Andrey Nikolaev, Egor Smirnov, Ruslan Israfilov, and Judy Qiu. 2019. A GraphBLAS approach for subgraph counting. *arXiv preprint arXiv:1903.04395* (2019).

[35] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2019. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *arXiv preprint arXiv:1911.06969* (2019).

[36] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, S Yu Philip, and Haixun Wang. 2008. Fast graph pattern matching. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 913–922.

[37] James Cheng, Linhong Zhu, Yiping Ke, and Shumo Chu. 2012. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1240–1248.

[38] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM Journal on computing* 14, 1 (1985), 210–223.

[39] Diane J Cook and Lawrence B Holder. 2006. *Mining graph data*. John Wiley & Sons.

[40] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux device drivers*. " O'Reilly Media, Inc.".

[41] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.

[42] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[43] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2018. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2018), 640–653.

[44] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 589–598.

[45] William HE Day and David Sankoff. 1986. Computational complexity of inferring phylogenies by compatibility. *Systematic Biology* 35, 2 (1986), 224–229.

[46] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2018. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. 393–404.

[47] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy Blelloch, Phillip Gibbons, and Julian Shun. 2020. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *PVLDB* (2020).

[48] 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*. ACM, 1357–1374.

[49] Sumeet Dua and Xian Du. 2016. *Data mining and machine learning in cybersecurity*. CRC press.

[50] John D Eblen, Charles A Phillips, Gary L Rogers, and Michael A Langston. 2012. The maximum clique enumeration problem: algorithms, applications, and implementations. In *BMC bioinformatics*, Vol. 13. Springer, S5.

[51] David Eppstein, Maarten Löffler, and Darren Strash. 2010. Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time. In *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part I*. 403–414. https://doi.org/10.1007/978-3-642-17517-6_36

[52] Brian Gallagher. 2006. Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching.. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*. 45–53.

[53] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. 2019. Computedram: In-memory compute using off-the-shelf drams. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 100–113.

[54] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 113–124.

[55] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 126–137.

[56] Saugata Ghose, Amirali Boroumand, Jeremie S Kim, Juan Gómez-Luna, and Onur Mutlu. 2019. Processing-in-Memory: A Workload-driven Perspective. *IBM JRD* (2019).

[57] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. 2019. The processing-in-memory paradigm: Mechanisms to enable adoption. In *Beyond-CMOS Technologies for Next Generation Computer Design*. Springer, 133–194.

[58] Lukas Gianinazzi, Maciej Besta, Yannick Schaffner, and Torsten Hoefler. 2021. Parallel Algorithms for Finding Large Cliques in Sparse Graphs. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*. 243–253.

[59] Lukas Gianinazzi, Maximilian Fries, Nikoli Dryden, Tal Ben-Nun, and Torsten Hoefler. 2021. Learning Combinatorial Node Labeling Algorithms. *arXiv preprint arXiv:2106.03594* (2021).

[60] Lukas Gianinazzi, Pavel Kalvoda, Alessandro De Palma, Maciej Besta, and Torsten Hoefler. 2018. Communication-avoiding parallel minimum cuts and connected components. *ACM SIGPLAN Notices* 53, 1 (2018), 219–232.

[61] David Gibson, Ravi Kumar, and Andrew Tomkins. 2005. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st international conference on Very large data bases*. 721–732.

[62] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*. PMLR, 1263–1272.

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

[64] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. 2021. SIMDRAM: a framework for bit-serial SIMD processing using DRAM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 329–345.

[65] 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 *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–13.

[66] J Han and M Kamber. 2006. Data Mining Concepts and Techniques (A. Stephan, Ed.), 2nd edn., vol. 40.

[67] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1587–1602.

[68] Lei He. 2019. EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks. *arXiv preprint arXiv:1909.00155* (2019).

[69] Eric Robert Hein. 2018. *Near-data processing for dynamic graph analytics*. Ph.D. Dissertation. Georgia Institute of Technology.

[70] Wim Heirman, Trevor Carlson, and Lieven Eeckhout. 2012. Sniper: Scalable and accurate parallel multi-core simulation. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*. High-Performance and Embedded Architecture and Compilation Network of . . . , 91–94.

[71] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The art of multiprocessor programming*. Newnes.

[72] Shohei Hido and Hiroyuki Kawano. 2005. AMIOT: induced ordered tree mining in tree-structured databases. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*. IEEE, 8–pp.

[73] Torsten Hoefler and Roberto Belli. 2015. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–12.

[74] Tamás Horváth, Thomas Gärtner, and Stefan Wrobel. 2004. Cyclic pattern kernels for predictive graph mining. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 158–167.

[75] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 25–32.

[76] Tianhao Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2018. HyVE: Hybrid vertex-edge memory hierarchy for energy-efficient graph processing. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 973–978.

[77] Yu Huang, Long Zheng, Xiaofei Liao, Hai Jin, Pengcheng Yao, and Chuangyi Gui. 2019. RAGra: Leveraging Monolithic 3D ReRAM for Massively-Parallel Graph Processing. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1273–1276.

[78] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. {ASAP}: Fast, Approximate Graph Pattern Mining at Scale. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 745–761.

[79] Said Jabbour, Nizar Mhadhbi, Badran Raddaoui, and Lakhdar Sais. 2018. Pushing the Envelope in Overlapping Communities Detection. In *International Symposium on Intelligent Data Analysis*. Springer, 151–163.

[80] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[81] Raymond Austin Jarvis and Edward A Patrick. 1973. Clustering using a similarity measure based on shared near neighbors. *IEEE Transactions on computers* 100, 11 (1973), 1025–1034.

[82] Thomas Jech. 2013. *Set theory*. Springer Science & Business Media.

[83] Joe Jeddeloh and Brent Keeth. 2012. Hybrid memory cube new DRAM architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*. IEEE, 87–88.

[84] Chuntao Jiang, Frans Coenen, and Michele Zito. 2013. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review* 28, 1 (2013), 75–105.

[85] Daxin Jiang and Jian Pei. 2009. Mining frequent cross-graph quasi-cliques. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 2, 4 (2009), 1–42.

[86] Aparna Joshi, Yu Zhang, Petko Bogdanov, and Jeong-Hyon Hwang. 2018. An Efficient System for Subgraph Discovery. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 703–712.

[87] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, and Shuotao Xu. 2018. GraFBoost: Using accelerated flash storage for external graph analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 411–424.

[88] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. 2017. High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering* 30, 2 (2017), 305–324.

[89] 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*. 1217–1231.

[90] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, and Henning Meyerhenke. 2016. Mathematical foundations of the GraphBLAS. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 1–9.

[91] Arijit Khan. 2016. Vertex-centric graph processing: The good, the bad, and the ugly. *arXiv preprint arXiv:1612.07404* (2016).

[92] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. K-core decomposition of large networks on a single PC. *Proceedings of the VLDB Endowment* 9, 1 (2015), 13–23.

[93] Seongyun Ko and Wook-Shin Han. 2018. Turbograph++: A scalable and fast graph analytics system. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 395–410.

[94] Michihiro Kuramochi and George Karypis. 2001. Frequent subgraph discovery. In *Proceedings 2001 IEEE international conference on data mining*. IEEE, 313–320.

[95] Michihiro Kuramochi and George Karypis. 2004. An efficient algorithm for discovering frequent subgraphs. *IEEE transactions on Knowledge and Data Engineering* 16, 9 (2004), 1038–1051.

[96] Dominique Lavenier, Jean-Francois Roy, and David Furodet. 2016. DNA mapping using Processor-in-Memory architecture. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 1429–1435.

[97] Victor E Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. 2010. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*. Springer, 303–336.

[98] Elizabeth A Leicht, Petter Holme, and Mark EJ Newman. 2006. Vertex similarity in networks. *Physical Review E* 73, 2 (2006), 026120.

[99] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research* 11, Feb (2010), 985–1042.

[100] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. Drisa: A dram-based reconfigurable in-situ accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 288–301.

[101] 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 *Proceedings of the 53rd Annual Design Automation Conference*. 1–6.

[102] David Liben-Nowell and Jon Kleinberg. 2007. The link-prediction problem for social networks. *Journal of the American society for information science and technology* 58, 7 (2007), 1019–1031.

[103] Siyuan Liu and Arijit Khan. 2018. An Empirical Analysis on Expressibility of Vertex Centric Graph Processing Paradigm. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 242–251.

[104] Gabriel H Loh. 2008. 3D-stacked memory architectures for multi-core processors. In *ACM SIGARCH computer architecture news*, Vol. 36. IEEE Computer Society, 453–464.

[105] Linyuan Lü and Tao Zhou. 2011. Link prediction in complex networks: A survey. *Physica A: statistical mechanics and its applications* 390, 6 (2011), 1150–1170.

[106] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[107] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry. 2007. Challenges in Parallel Graph Processing. *Par. Proc. Let.* 17, 1 (2007), 5–20.

[108] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.

[109] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. 2017. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 631–643.

[110] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture*. 116–128.

[111] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2019. GraphZero: Breaking Symmetry for Efficient Graph Mining. *arXiv preprint arXiv:1911.12877* (2019).

[112] 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*. ACM, 509–523.

[113] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 25.

[114] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[115] Gary L Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. 2015. Improved parallel algorithms for spanners and hopsets. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 192–201.

[116] Sparsh Mittal, Jeffrey S Vetter, and Dong Li. 2014. Improving energy efficiency of embedded DRAM caches for high-end computing systems. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 99–110.

[117] O. Mutlu et al. 2019. Processing Data Where It Makes Sense: Enabling In-Memory Computation. *MicPro* (2019).

[118] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2020. A Modern Primer on Processing in Memory. *arXiv preprint arXiv:2012.03112* (2020).

[119] Anirban Nag, CN Ramachandra, Rajeev Balasubramonian, Ryan Stutsman, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. 2019. Gencache: Leveraging in-cache operators for efficient sequence alignment. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 334–346.

[120] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. Graphpim: Enabling instruction-level PIM offloading in graph computing frameworks. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 457–468.

[121] Neo4j, Inc. 2019. The Neo4j Graph Algorithms User Guide v3.5. https://neo4j.com/docs/graph-algorithms/current.

[122] Geraldo F Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. 2021. DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks. *arXiv preprint arXiv:2105.03725* (2021).

[123] 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 *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 166–177.

[124] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[125] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, and Mario Méndez-Lojo. 2011. The tao of parallelism in algorithms. In *ACM Sigplan Notices*, Vol. 46. ACM, 12–25.

[126] T Ramraj and R Prabhakar. 2015. Frequent subgraph mining algorithms-a survey. *Procedia Computer Science* 47 (2015), 197–204.

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

[128] Saif Ur Rehman, Asmat Ullah Khan, and Simon Fong. 2012. Graph mining: A survey of graph mining techniques. In *Seventh International Conference on Digital Information Management (ICDIM 2012)*. IEEE, 88–92.

[129] Nicholas Rhodes, Peter Willett, Alain Calvet, James B Dunbar, and Christine Humblet. 2003. CLIP: similarity searching of 3D databases using clique detection. *Journal of chemical information and computer sciences* 43, 2 (2003), 443–448.

[130] Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. 2019. A Survey on Subgraph Counting: Concepts, Algorithms and Applications to Network Motifs and Graphlets. *arXiv preprint arXiv:1910.13011* (2019).

[131] Ian Robinson, Jim Webber, and Emil Eifrem. 2013. *Graph databases*. " O'Reilly Media, Inc.".

[132] Ryan A. Rossi and Nesreen K. Ahmed. 2016. An Interactive Data Repository with Visual Analytics. *SIGKDD Explor.* 17, 2 (2016), 37–41. http://networkrepository.com

[133] Ryan A Rossi and Nesreen K Ahmed. 2016. An interactive data repository with visual analytics. *ACM SIGKDD Explorations Newsletter* 17, 2 (2016), 37–41.

[134] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 472–488.

[135] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A Boncz, et al. 2020. The Future is Big Graphs! A Community View on Graph Processing Systems. *arXiv preprint arXiv:2012.06171* (2020).

[136] Semih Salihoglu and Jennifer Widom. 2014. Optimizing graph algorithms on Pregel-like systems. *Proceedings of the VLDB Endowment* 7, 7 (2014), 577–588.

[137] Satu Elisa Schaeffer. 2007. Graph clustering. *Computer science review* 1, 1 (2007), 27–64.

[138] Thomas Schank. 2007. Algorithmic aspects of triangle-based network analysis. *Phd in computer science, University Karlsruhe* 3 (2007).

[139] Vivek Seshadri, Abhishek Bhowmick, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2014. The dirty-block index. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 157–168.

[140] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, and Michael A Kozuch. 2013. RowClone: fast and energy-efficient in-DRAM bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 185–197.

[141] 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 *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 273–287.

[142] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR)* 50, 6 (2018), 81.

[143] Yossi Shiloach and Uzi Vishkin. 1980. *An O (log n) parallel connectivity algorithm*. Technical Report. Computer Science Department, Technion.

[144] Yossi Shiloach and Uzi Vishkin. 1982. An O (logn) parallel connectivity algorithm. *Journal of Algorithms* 3, 1 (1982), 57–67.

[145] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, Vol. 48. 135–146.

[146] Julian Shun and Kanat Tangwongsan. 2015. Multicore triangle computations without tuning. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 149–160.

[147] S Skiena. 1990. Dijkstra's algorithm. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley* (1990), 225–227.

[148] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2017. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 47.

[149] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 531–543.

[150] Victor Spirin and Leonid A Mirny. 2003. Protein complexes and functional modules in molecular networks. *Proceedings of the National Academy of Sciences* 100, 21 (2003), 12123–12128.

[151] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225.

[152] Michael Sutton, Tal Ben-Nun, and Amnon Barak. [n. d.]. Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling. ([n. d.]).

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

[154] Lei Tang and Huan Liu. 2010. Graph mining applications to social network analysis. In *Managing and Mining Graph Data*. Springer, 487–513.

[155] Ben Taskar, Ming-Fai Wong, Pieter Abbeel, and Daphne Koller. 2004. Link prediction in relational data. In *Advances in neural information processing systems*. 659–666.

[156] 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*. ACM, 425–440.

[157] Sutapat Thiprungsri and Miklos A Vasarhelyi. 2011. Cluster Analysis for Anomaly Detection in Accounting Data: An Audit Approach. *International Journal of Digital Accounting Research* 11 (2011).

[158] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. 2006. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.* 363, 1 (2006), 28–42. https://doi.org/10.1016/j.tcs.2006.06.015

[159] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23, 1 (1976), 31–42.

[160] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. 2013. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 177–187.

[161] 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.

[162] Liang Wang, Ke Hu, and Yi Tang. 2014. Robustness of link-prediction algorithm based on similarity and application to biological networks. *Current Bioinformatics* 9, 3 (2014), 246–252.

[163] Takashi Washio and Hiroshi Motoda. 2003. State of the art of graph-based data mining. *Acm Sigkdd Explorations Newsletter* 5, 1 (2003), 59–68.

[164] Stanley Wasserman and Katherine Faust. 1994. *Social network analysis: Methods and applications*. Vol. 8. Cambridge university press.

[165] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. 2011. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* 116 (2011).

[166] Andrew Shell Waterman. 2016. *Design of the RISC-V instruction set architecture*. Ph.D. Dissertation. UC Berkeley.

[167] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[168] Xin Xin, Youtao Zhang, and Jun Yang. 2020. ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 303–314.

[169] Chongchong Xu, Chao Wang, Lei Gong, Lihui Jin, Xi Li, and Xuehai Zhou. 2018. Domino: Graph Processing Services on Energy-Efficient Hardware Accelerator. In *2018 IEEE International Conference on Web Services (ICWS)*. IEEE, 274–281.

[170] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[171] Da Yan, Hongzhi Chen, James Cheng, M Tamer Özsu, Qizhen Zhang, and John Lui. 2017. G-thinker: big graph mining made easier and faster. *arXiv preprint arXiv:1709.03110* (2017).

[172] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1821–1832.

[173] Da Yan, Wenwen Qu, Guimu Guo, and Xiaoling Wang. 2020. PrefixFPM: A Parallel Framework for General-Purpose Frequent Pattern Mining. In *Proceedings of the 36th IEEE International Conference on Data Engineering (ICDE) 2020*.

[174] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 15–29.

[175] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. [n. d.]. A Locality-Aware Energy-Efficient Accelerator for Graph Mining Applications. ([n. d.]).

[176] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. 2020. A Locality-Aware Energy-Efficient Accelerator for Graph Mining Applications. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 895–907.

[177] 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 *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 544–557.

[178] Yun Zhang, Faisal N Abu-Khzam, Nicole E Baldwin, Elissa J Chesler, Michael A Langston, and Nagiza F Samatova. 2005. Genome-scale computational approaches to memory-intensive applications in systems biology. In *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, 12–12.

[179] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Xueqi Cheng. 2019. Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine. *arXiv preprint arXiv:1905.09572* (2019).

[180] Kangfei Zhao and Jeffrey Xu Yu. 2017. All-in-one: Graph processing in rdbmss revisited. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1165–1180.

[181] Long Zheng, Jieshan Zhao, Yu Huang, Qinggang Wang, Zhen Zeng, Jingling Xue, Xiaofei Liao, and Hai Jin. 2020. Spara: An Energy-Efficient ReRAM-Based Accelerator for Sparse Graph Analytics Applications. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 696–707.

[182] Minxuan Zhou, Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. GRAM: graph processing in a ReRAM-based computational memory.. In *ASP-DAC*. 591–596.

[183] 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.