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

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

\textsuperscript{1} ETH Zurich \textsuperscript{2} IIT Tirupati \textsuperscript{3} King Mongkut’s University of Technology North Bangkok \textsuperscript{4} Technical University of Ostrava \textsuperscript{5} AGH-UST

\section*{ABSTRACT}
Simple graph algorithms such as PageRank have recently 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 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 \(10 \times\) 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.

\section{INTRODUCTION}
Research in graph analytics in the architecture community has mostly targeted graph algorithms based on vertex-centric formulations \cite{5,6,13,26,73,93,123,131,154,196}. Some works also focus on edge-centric or linear algebra paradigms \cite{97,145,162,164}. Such algorithms have complexities described by low-degree polynomials \cite{98}, for example Breath-First Search (BFS) \cite{47} \((O(n + m))\) or iteration-based PageRank (PR) \cite{25} \((O(m \cdot \#iterations))\), 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 \cite{28,45,89,148,170} that received little or no attention in the architecture community. One large class is graph pattern matching \cite{89}, which focuses on finding certain specific subgraphs (also called motifs or graphlets). Examples of such problems are \(k\)-clique listing \cite{53}, maximal clique listing \cite{33,36,62,172}, \(k\)-star-clique mining \cite{84}, and many others \cite{45}. Another class is broadly referred to as graph learning \cite{45}, with problems such as unsupervised learning or clustering \cite{86}, link prediction \cite{8,109,113,168}, or vertex similarity \cite{104}. All these problems are used in social sciences \cite{62}, bioinformatics \cite{62}, computational chemistry \cite{166}, medicine \cite{166}, cybersecurity \cite{59}, healthcare \cite{171}, web graph analysis \cite{90}, and many others \cite{37,45,80,89}. These problems often run in time at least quadratic in the number of vertices, and many problems are NP-complete \cite{33,45,53,173}. 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 \cite{117}, is not the right tool for expressing graph mining problems. This paradigm 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 \cite{45}. Obtaining such knowledge in the vertex-centric paradigm is hard or infeasible, as noted by Kalavri et al. \cite{93} (“(...) graph algorithms, like triangle counting, are not a good fit for the vertex-centric model”) and many others \cite{100,110,147,187}. Similar arguments apply to other paradigms such as GraphBLAS \cite{97,145} and to frameworks such as Ligra \cite{157}. None of them supports a wide selection of graph mining problems (e.g., GraphBLAS only enables subgraph isomorphism assuming patterns are trees); we show it in detail in Table 1 and Section 4.

Several graph mining software frameworks (Peregrine \cite{85} and others \cite{40,41,58,83,91,121,122,170,186,188,198}) have been proposed. Yet, they focus exclusively on a few graph pattern matching problems. Moreover, these frameworks usually do not offer theoretical guarantees (unlike parallel graph algorithms for specific mining problems). Overall, there is a need for a graph mining paradigm that would enable expressing many graph mining problems, and ideally offer competitive theoretical guarantees on their runtimes.

Moreover, past works illustrated that graph mining algorithms are memory bound \cite{42,60,85,193,197}. This is because these algorithms generate and heavily use large intermediate structures, but – similarly to algorithms such as PageRank – they are not compute-heavy \cite{62,85,194}. We show this in Figure 1. When increasing the number of parallel threads,
these instructions are offloaded to PIM units. We call these instructions SISA as they form a “Set-centric” ISA extension that enables 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), the actual set-centric ISA extension with its instructions, implemented set operations, set organization, and a thin software layer (contribution #2), and PIM acceleration (contribution #3).

Using set algebra as a basis for algorithm design ensures that SISA set-centric algorithms are succinct, applicable to many problems, and theoretically efficient. Our set-centric paradigm is the first to use set operations as fundamental general building blocks for both algorithmic formulations and their execution. Next, when mapping set-centric formulations to the SISA code, one can use different set representations (e.g., a sparse integer array or a dense bitvector), and set operations such as intersection can be executed using different set algorithms (e.g., merge or galloping intersection) [74]). These choices enable flexibility as they come with different performance/storage tradeoffs, which we analyze in detail.

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 [70], as offered in Ambit [153], ELPI2IM [183], DRISA [107], or ComputeDRAM [65], 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 of near-memory PIM [112] as offered in the 2D UPMEM architecture [101] or logic layers of 3D DRAM such as Hybrid Memory Cube (HMC) [88] (“SISA processing near memory” – SISA-PNM). For even higher speedups, we provide a small HW controller that selects the best variant of a set instruction to be executed on-the-fly.

Overall, we show that graph mining algorithms, despite being complex and lacking straightforward parallelism (unlike PageRank and similar), benefit from PIM. Our key solution is using parallelism offered by set operations and exposed with the set-centric approach. This harnesses parallelism at the level of bits, DRAM subarrays, and vaults. We build upon recent HW developments and are the first to comprehensively implement graph mining algorithms with PIM. We integrate SISA with the RISC-V ISA [181] and 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 [62].

2. NOTATION AND BACKGROUND

We first describe background and notation, see Table 2.

**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 $1, \ldots, n$ and $V = \{1, \ldots, n\}$. $N(v)$ denote the neighbors of $v \in V$; $d$ and $d(v)$ denote $G$’s maximum degree and a degree of $v$.

**Set Representations** SISA heavily uses sets. Consider a set of $k$ vertices $S = \{v_1, \ldots, v_k\} \subseteq V$ (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).

### Set Operations
SISA uses fundamental set operations: intersection $A \cap B$, union $A \cup B$, difference $A \setminus B$, cardinality $|A|$, and membership $\in$. We use different algorithms to implement these operations (described later in the paper).

### 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 [62]) by default use rich set notation, many others, such as $k$-clique listing by Danisch et al. [53], 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 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 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).

### 4. PROVABLY FAST GRAPH MINING

We first show that the set-centric approach is superior to existing graph programming paradigms as (1) it supports many graph mining problems and (2) it enables algorithms with theoretical guarantees on performance (e.g., work/depth [31]) that are competitive to those of tuned algorithms. Such provable guarantees are often key to low runtimes and scalability [56, 98]. The analysis results 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, 37, 64, 89, 103, 104, 109, 113, 138, 139, 141, 167, 178]. For the former, we pick four problems from both graph pattern matching and graph learning areas (maximal clique listing [33], $k$-clique listing [44], dense subgraph discovery [72, 103], subgraph isomorphism [173], vertex similarity [104, 142], link prediction [8, 109, 113, 168], graph clustering [86, 148], verification of prediction accuracy [177]). 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 [117] and edge-centric [145], two established graph processing paradigms implemented in the Pregel and X-Stream systems. Second, we pick vertex/edge array maps from Ligra [157], an approach for developing graph algorithms based on transforming arrays of vertices or edges according to a specified map. Third, we consider Graph- BLAS and its linear algebraic approach [97], where graph algorithms are expressed with linear algebra building blocks such as matrix-vector products. Moreover, we consider pattern matching frameworks [64] 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
### 5. SET-CENTRIC GRAPH ALGORITHMS

We present set-centric formulations of graph mining algorithms. A list of algorithms and set operations used in each algorithm is in Table 3. Due to space constraints, we provide a few selected key formulations.

**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. 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 [Pattern Matching]** 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 [54, 140, 163, 179]. Listing 1 shows the widely used recursive backtracking Bron-Kerbosch algorithm [BK] [33, 36, 62]. BK heavily uses different set operations. The main recursive function BKPivotv (Line 5) has three arguments that are dynamic and principles from relational databases and the associated algebra [199].

```
Algorithm 1: Maximal Clique Listing (Bron-Kerbosch) [33, 36].
```

```
2. P = V; R = 0; X = ∅; //Init sets appropriately.
3. for v ∈ V [in par] do: BKPivotv(v, P, X);
4. function BKPivotv(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) ∪ R
7. for v ∈ (P \ u) ∩ X do: BKPivotv(R \ v, P \ u, X \ v);
8. P = P \ u; X = X \ u;
```

Algorithm 2: k-clique-star listing [84].

```
Algorithm 2: k-clique-star listing [84].
```

**Vertex Similarity & Clustering [Graph Learning]** Various measures assess how similar two vertices v and u are, see Listing 3. 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.

```
Algorithm 3: Vertex similarity measures.
```

**“Low-Complexity” Graph Algorithms: Discussion** SISA does not target the “low-complexity” algorithms such as PageRank, as these algorithms offer few straightforward opportunities for set-centric acceleration. For example, in PageRank, one iterates over two nested loops, and updates vertex ranks, which is not easily expressible with set operations. We analyzed many other such algorithms, including Dijkstra’s SSSP [160], A-Stepping [124], Bellman-Ford [47], Betweenness Centrality schemes [161], traversals [24], Connected Components algorithms [71, 156, 165, 187], Low-Diameter Decomposition [125], or Boruvka’s Minimum Spanning Tree [32]. Some of them use set notation, but these are not “PIM-friendly” operations such as bulk set intersections of vertex sets, and are thus not easily accelerated with SISA (we list such operations in Table 3). While this may be possible, we leave this direction for future work.

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 (Section 5). Yet, sometimes 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 graph problems with graph neural (GNN) and convolution (GCN) networks, or general deep learning [15, 182], as well as joins and principles from relational databases and the associated algebra [199].

The analysis results are in Table 1. Overall, no single paradigm, except for the set-centric approach, enables efficient graph mining 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 connected to the k-clique actually form a (k + 1)-clique (together with this k-clique). Thus, to find k-clique-stars, we first mine (k + 1)-cliques. Then, we find k-clique-stars within each (k + 1)-clique using set union, membership, and difference.
Figure 3: Overview of SISA instructions and syntax at different levels of abstraction. The snippet for deriving the count of all 4-cliques `cnt`, a derived set-centric algorithmic formulation, and the corresponding SISA snippet in Table 4. 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 present the details of representing and processing sets used in set-centric formulations. This constitutes core parts of SISA’s design. We summarize SISA in Figure 3 and we detail key SISA instructions in Table 5.

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 \(O(n)\), and each single neighborhood is static (we currently focus on static graphs) and sorted (following the established practice in graph processing [118]). 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 16\(n\) 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. Simultaneously, 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% of the total storage needed 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 \(R\).)

We analyze and confirm it for other algorithms and datasets.

The user controls selecting a set representation. For programmability, SISA offers a predefined graph structure, where small and large neighborhoods are automatically stored as sparse arrays and dense bitvectors, respectively. This is guided by a simple model: 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.

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. The input sets \(A\) and \(B\) represent.

<table>
<thead>
<tr>
<th>Ins Set op.</th>
<th>A and B represent.</th>
<th>Set algorithm</th>
<th>S</th>
<th>Time complexity</th>
<th>Input size [bits]</th>
<th>Main form of data transfer (§4.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x4(A.B)</td>
<td>SA (\cup) SA Merge</td>
<td>(\bigcup) (O(\log B))</td>
<td>(W[A])</td>
<td>(W[B])</td>
<td>Streaming</td>
<td></td>
</tr>
<tr>
<td>8x1(A.B)</td>
<td>SA (\cup) SA Galloping</td>
<td>(\bigcup) (O(n\log B))</td>
<td>(W[A])</td>
<td>(W[B])</td>
<td>Random accesses</td>
<td></td>
</tr>
<tr>
<td>8x2(A.B)</td>
<td>SA (\cup) SA Merge</td>
<td>(\bigcup) (O(8n))</td>
<td>(W[A])</td>
<td>(W[B])</td>
<td>Random accesses</td>
<td></td>
</tr>
<tr>
<td>8x3(A.B)</td>
<td>SA (\cup) SA Galloping</td>
<td>(\bigcup) (O(nA))</td>
<td>(W[A])</td>
<td>(n + n)</td>
<td>Random accesses</td>
<td></td>
</tr>
<tr>
<td>8x4(A.B)</td>
<td>DB (\cup) DB Binary AND</td>
<td>(\bigcup) (O(n))</td>
<td>(W[A])</td>
<td>(n + n)</td>
<td>In-situ row copies</td>
<td></td>
</tr>
<tr>
<td>8x5(A.M)</td>
<td>DB (\cup) Set bit</td>
<td>(\bigcup) (O(n))</td>
<td>(n + W)</td>
<td>Random accesses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8x6(A.B)</td>
<td>DB (\cup) Clear bit</td>
<td>(\bigcup) (O(n))</td>
<td>(n + W)</td>
<td>Random accesses</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Overview of SISA instructions. One row describes one specific set operation variant. Set elements are vertices (\(A \subset \subset V \subset E\)). \(\bigcup\) means “yes”; "na" means “not applicable”. "time" is a proposed instruction opcode. "S (Sorted)" indicates if an instruction assumes set representations of \(A\) and \(B\) to be sorted (thus two columns).
operations for highest performance. For this, we detail the algorithmic aspects, a summary is in Table 5. 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 $\cap$, 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 $A$ 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 § 8.3).

- **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 Membership $x \in A$ and 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 maintain this information for any set. This incurs only $O(1)$ storage overhead for any 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, we note that 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 and 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 in case the SA is sorted). 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.

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 [181]. To enable modularity and flexibility, SISA’s new instructions are encoded using the custom opcode set [180]. 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 5, 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.

7. THEORETICAL ANALYSIS

We now support our claim that SISA-enhanced algorithms are *theoretically efficient*, i.e., their time complexities match those of hand-tuned graph mining algorithms. In general, this is enabled by SISA’s ability to control set representations and set operations, facilitates tuning performance and storage tradeoffs. To show this, we analyze how varying a set intersection variant (merge vs. galloping) impacts the runtime of set-centric algorithms. We focus on intersection as it is prevalent in considered algorithms. The analysis results are in Table 6 (proofs are in the report). Crucially, all set-centric variants are able to match the competitive time complexities of considered tuned graph mining algorithms.

We parametrize complexities with degeneracy $c$, a well-known measure of graph sparsity [120]. The degeneracy $c$ of a graph $G$ is the smallest number $x$ such that every subgraph in $G$ has a vertex of degree at most $x$ (i.e., every subgraph has at least one sparsely connected vertex). We consider degeneracy as it is used by many recent graph mining algorithms to enhance their time complexities [53, 62, 206].

8. HARDWARE IMPLEMENTATION

We now discuss details of SISA hardware implementation.

8.1 Processing-In-Memory for Sets

We start with how SISA uses PIM for set operations.

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 [153], 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., ELPI2IM [183], DRISA [107], ComputeDRAM [65], PCM (Pinatubo) [108]). The key extension in Ambit (for in-situ processing) is to modify a decoder for three selected DRAM
importance 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 [152] 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: 
\[ A \setminus B = A \cap B^c \] [87].

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 [101], HMC [88], or Tesseract [5]). 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.

8.2 Automatizing SISA Decisions

We use a small SISA Control Unit (SCU), cf. Section 3, to automatically decide which SISA instructions to run, and how. 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.

**Automatic Selection of 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 the SISA metadata structure and possibly cached in SCU’s cache). 

**Automatic Selection of 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.

8.3 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 
\[ l_M + W \cdot \max(|A|, |B|) \cdot \min(b_M, b_L) \]
where \( l_M \) 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 the overall bandwidth is bottlenecked by \( \min(b_M, b_L) \).

To model **random accesses**, we simply count the number of performed operations and multiply it by the memory access latency. This gives 
\[ c_M \cdot \min(|A|, |B|) \cdot \log(\max(|A|, |B|)) \]
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 (for each set) in a simple in-memory SM (“set 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 [5].

8.4 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 a respecting entry from the SM structure.

**Set Metadata** The SM structure is 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 [143]. 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 [143]), but not (or to a smaller extend) from \( n \). 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 designated such core).

Using SISA-PNM and SISA-PUM Together We rely on Ambit’s full compatibility with DRAM, as described in the original publication [153]. Specifically, 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. Thus, one can freely use standard DRAM accesses for any non-SISA-PUM set modifications.

Harnessing Parallelism SISA HW harnesses memory parallelism at different levels, enabling parallel execution of both a single set operation and different set operations. 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 can be processed in parallel. This applies to other parts of the DRAM hierarchy, for example banks. 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 [5].

Managing Concurrency SISA relies on established techniques (locks, lock-free protocols, and general parallel programming principles [78] and libraries such as OpenMP [39]) to manage concurrent accesses to the same set.

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).

8.5 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 [81] (a hybrid vertex-edge memory hierarchy that uses ReRAM and DRAM) or in GraphH [51] (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 [51].

9. EVALUATION

We illustrate example performance advantages from SISA.

9.1 Methodology, Setup, Parameters

Simulation Infrastructure We use Sniper [77] with the Pin frontend [114]. Sniper is a popular cycle-level simulator used in many works proposing various architectural extensions for both CPUs and memory subsystem [126, 174].

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 § 8.3. To simulate SISA-PUM, we model a run-time of in-situ operations with a delay $l_m + l_\ell \cdot \lceil n/(qS) \rceil$, where $l_m$ is the latency of accessing DRAM (to initiate the operation) and $l_\ell$ is the latency of executing a single 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 (cf. Table 2).

Simulated Platform for SISA & SISA Parametrization For concreteness, we set the platform for executing SISA instructions to match Tesseract [5] (for SISA-PNM) and Ambit [153] (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 of memory bandwidth to its core. Thus, we assume scalable bandwidth as proposed by Tesseract: using more vaults increases the total memory bandwidth. In the latter, we set the DRAM row rank size to 8 KB, following Ambit [153]. 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).

Simulated Platform for non-SISA Instructions & Baselines For any non-SISA instructions and comparison 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 that 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 [62], Danisch’s scheme for kcc-$k$ [53], enhanced Jabbour’s scheme for ksc-$k$ [84], parallel VF2 for $si-k$s [46], and cl-jac based on counting triangles in the GAP suite [14]. All used baselines have competitive work and depth complexities, cf. Table 6. 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 When possible, we compare to graph pattern matching frameworks: Peregrine [85] (a very recent and fast design that represents accelerators such as Gramer [194], cf. “pattern matching” in Table 1), and RStream [176] which represents accelerators such as TrieJax [94] based on relation algebra (cf. “joins” in Table 1). We stress that we focus on comparing to (much faster) hand-tuned parallel algorithms.

Graphs We select a broad set of input datasets from Network Repository [144], considering biological (bio-), interaction (int-), social (soc-), brain (br-), dynamic (D), web
(web-), economical (econ-), ecological (eco-), and structural (str-) networks. We pick graphs with different structural properties (low/high density, small/large maximum degree, low/high degree distribution, etc.).

**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 [53,62]. However, 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 [5] such as PageRank (limiting #iteration).

**Performance Measures & Summaries:** We focus on plain runtimes; this is recommended when measuring performance of parallel codes [79] as speedup may be misleading because it is higher on unoptimized baselines. However, for overview, we also summarize speedups (following [79]), i.e., we provide (1) speedups of average runtimes (“arithm”), and (2) geometric means of speedups of all data points (“geom”).

### 9.2 Discussion of Results

**Comparison to Hand-Tuned Algorithms** We first analyze run-times with all available cores, comparing SISA set-centric variants to non-set-based and set-based hand-tuned parallel baselines that 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 2x, often more than 10x (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 [144], 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

![Figure 6: Run-times with full parallelism. The bold red line indicates the cutoff of long simulation runtimes, used for readability (the bars reaching the line have much larger runtimes). No bar indicates the timeout of the respective baseline (>24h). The results for c1-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”.

```plaintext
Figure 6: Run-times with full parallelism. The bold red line indicates the cutoff of long simulation runtimes, used for readability (the bars reaching the line have much larger runtimes). No bar indicates the timeout of the respective baseline (>24h). The results for c1-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".
```
the high bandwidth setting. In other cases (e.g., bio-H5-LC), more vertices have large enough degrees to benefit from DBs and low latencies of SISA-PUM.

**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 [105] 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. Run-time 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 Pattern Matching Frameworks** We compare SISA set-centric algorithms to Peregrine and RStream. 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 is only able to 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 these frameworks 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.

**SCU cache** First, not using the SCU cache results in the loss of performance of $\approx 1.5 \times$ for $T = 1$ and $\approx 0.05-0.1 \times$ for $T = 32$. The lower performance loss for high $T$ is because, with more threads executing set operations, it becomes increasingly 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.

**Varying Fraction of Dense/Sparse Neighborhood** Second, we observe that – while using SISA-PUM is beneficial for the overall performance – too many neighborhoods stored as DBs result in slowdowns. This is because when sparse neighborhoods are also stored as DBs, processing such sets (which have always size $n$ bits) with SISA-PUM begins to take more time than processing them with SISA-PNM. Thus, it is relevant to not choose the 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 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 the 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.

We also analyze **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.

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 & DISCUSSION

We already extensively described related graph processing paradigms (Table 1) and various software related graph processing efforts (Section 1) [20, 20, 25, 115, 146]. We now briefly 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. Additionally, this work is an example of how to seamlessly integrate PUM and PNM capabilities in a single system. They work synergistically and produce significantly better results than working separately. Works orthogonal to SISA include HW accelerated dynamic (time-evolving) graph analytics [34, 35, 76], or external memory HW accelerated graph processing [57, 92, 119]. One could use the latter as a SISA backend for external memory set instructions; we leave details for future work.

While in the current SISA version we focus on implementing and executing set operations in set-centric algorithm

<table>
<thead>
<tr>
<th>Reference / Accelerator</th>
<th>Prob.</th>
<th>Key memory mechanism</th>
<th>Pattern M. Learning “Low-c” is slab</th>
</tr>
</thead>
<tbody>
<tr>
<td>[P1] G425-X [38]</td>
<td>SpMV</td>
<td>[M] CAM/MAC</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P2] FMA [54]</td>
<td>vec-c</td>
<td>[M] cachetAM</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P3] GraphHe [10]</td>
<td>low-c</td>
<td>[E] DRAM</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P4] GraphMA [106]</td>
<td>edge-c</td>
<td>[E] DRAM</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P5] Spada [200]</td>
<td>ver-c</td>
<td>[E] ReRAM</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P6] GraphQ [209]</td>
<td>ver-c</td>
<td>[E] HMC</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P7] ForeGraph [11]</td>
<td>low-c</td>
<td>[E] GPUs</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P8] RAGe [82]</td>
<td>ver-c</td>
<td>[E] 3D ReRAM</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P9] GraAM [201]</td>
<td>ver-c</td>
<td>[E] ReRAM</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P10] Graphf [162]</td>
<td>SpMV</td>
<td>[E] ReRAM</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P11] Graphp [196]</td>
<td>ver-c</td>
<td>[E] HMC</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P12] Ma [116]</td>
<td>low-c</td>
<td>[E] DRAM</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P13] PIM-Enabled [6]</td>
<td>low-c</td>
<td>[E] HMC</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P14] Gas et al. [66]</td>
<td>low-c</td>
<td>[E] 3D DRAM</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>[P15] LLM [207, 208]</td>
<td>SpMSM</td>
<td>[E] 3D DRAM</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
</tbody>
</table>

Table 7: Comparison of SISA to graph-related accelerators, focusing on supported graph mining problems and offered architecture elements. [M]: Support / significant focus. [E]: Partial support / some focus. "x": no support / no focus. Addressed problems: see Table 1. Graph problems and algorithms: as in Table 1. Architecture and stack elements that are considered and discussed: is an ISA, or six extensions, xF: a cross-layer design, ah: a programming paradigm. Classes of accelerators: [P]: in-situ PIM, [Pc]: near memory PIM (e.g., logic layers). [A]: ASIC. [M]: focus on memory hierarchy enhancements. [F]: FPGA, [e] focus on extensions and modifications to the established (already proposed) HW technology.
formulations using PIM, SISA could be extended into different directions. This includes parallel and distributed execution of set operations, and implementing them using high-performance techniques such as Remote Direct Memory Access [19,21,68,150]. One could also enable more efficient execution of set-centric graph mining algorithms in the context of modern complex heterogeneous architectures that may host massively parallel on-chip networks [18], NUMA and systems with locality effects [151,169], or FPGAs [17,26,55]. One could also incorporate various forms of graph compression and summarization [22,27,29,111].

Sets are used in different graph algorithms, to simplify operations on selected data structures [25,33,99,124,137,149,155]. For example, the BFS frontier can be modeled as we pick in-situ and logic layer PIM for hardware acceler-

also be extended in other directions, for example by provid-

SIMD-based set intersections [74], FPGAs [26], or even exe-

xations, SISA’s set algebra interface could easily use other

unrolling. Due to the generality of set algebra, SISA can

to facilitate optimizations such as vectorization with loop

structions that accept multiple arguments (e.g.,

mining. SISA could be extended with CISC-style set in-

a small yet expressive “set-centric” ISA extension for graph

operations on selected data structures [25,33,99,124,137,

and exposes set operations in graph mining algorithms. This

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 3.

11. DISCUSSION AND CONCLUSION

We develop the first hardware acceleration approach for
general graph pattern matching and learning. 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. 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., \( A_1 \cap \ldots \cap A_k \)) 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 [74], FPGAs [26], or even execution in caches [3,130]. 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.

REFERENCES


