X-SET: An Efficient Graph Pattern Matching Accelerator With Order-Aware Parallel Intersection Units

Chenxi Xu
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
cxu930@connect.hkust-gz.edu.cn

Tianhui Shi
QingCheng.AI
Beijing, China
shitianhui@qingcheng.ai

Shixuan Sun
Shanghai Jiao Tong University
Shanghai, China
sunshixuan@sjtu.edu.cn

Jidong Zhai
Tsinghua University
Beijing, China
zhaijidong@tsinghua.edu.cn

Xinyu Chen*
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
xinyuchen@hkust-gz.edu.cn

Abstract

Graph Pattern Matching (GPM) is a critical task in a wide range of graph analytics applications, such as social network analysis and cybersecurity. Despite its importance, GPM remains challenging to accelerate due to its inherently irregular control flow and heavy reliance on set operations, which dominate execution time and introduce data dependencies that limit parallelism. While recent GPM accelerators attempt to improve performance, they often overlook the ordered nature of input data, resulting in redundant computations and inefficient hardware utilization.

This paper presents X-SET, a GPM accelerator that overcomes these limitations by introducing two key innovations. First, we propose an Order-Aware Set Intersection Unit (SIU), which exploits input ordering to reduce the hardware complexity of parallel set intersection from $O(N^2)$ to $O(N \log N)$, achieving high throughput and significant area savings by avoiding unnecessary comparisons. Second, we develop a barrier-free task scheduler that breaks traditional DFS scheduling constraints by enabling asynchronous, out-of-order task execution across different levels of the GPM search tree. X-SET is integrated into a RISC-V SoC, supporting end-to-end acceleration. Extensive experimental results show that X-SET outperforms state-of-the-art GPM accelerators, achieving **4.6**×-**142.9**× improvements in compute density, with a geometric mean of **13.7**×, and delivering **6.4**× geometric mean and **42.9**× maximum speedup in end-to-end performance. X-SET is open-sourced at github[1].

CCS Concepts

• **Computer systems organization** → **Parallel architectures**; • **Mathematics of computing** → **Graph enumeration**.

*Corresponding author
[1]https://github.com/CLab-HKUST-GZ/micro58-xset

MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756112

Keywords

Graph pattern matching, Hardware acceleration, Set intersection, Order-aware processing, Barrier-free scheduling, RISC-V

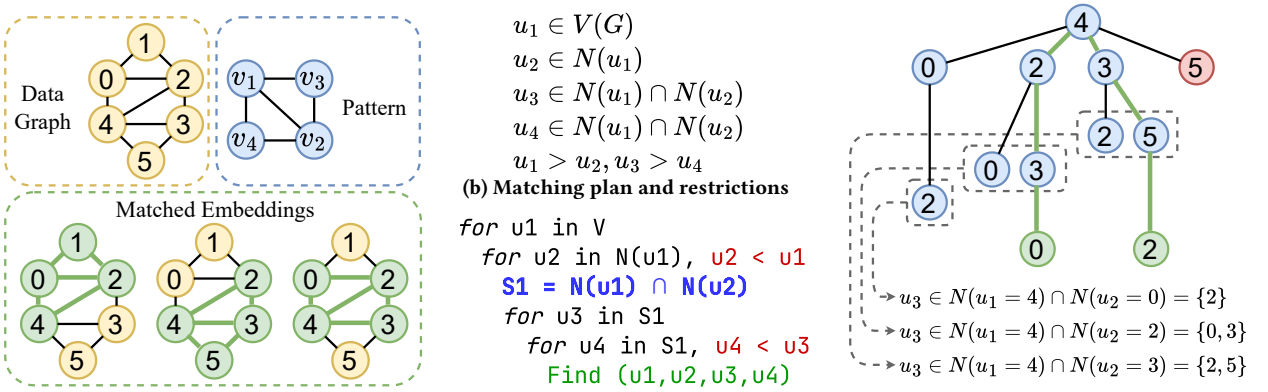
ACM Reference Format:
Chenxi Xu, Tianhui Shi, Shixuan Sun, Jidong Zhai, and Xinyu Chen. 2025. X-SET: An Efficient Graph Pattern Matching Accelerator With Order-Aware Parallel Intersection Units. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756112

1 Introduction

Graph Pattern Matching (GPM) is a fundamental operation in graph analysis that identifies all subgraphs in a large data graph structurally equivalent to a given query pattern. By discovering significant patterns like cliques [1] and motifs [16], GPM uncovers critical structural insights in complex networks. It is widely used in domains such as social networks [16, 18, 22, 23], bioinformatics [1, 2, 37, 53], cheminformatics [20, 28, 39], recommendation systems [34, 52] and web spam detection [19, 29].

However, the combinatorial complexity of GPM poses significant computational challenges. The need to explore a vast search space of potential matches renders general-purpose CPUs [12, 13, 24, 36, 41, 42, 48, 50, 54] and GPUs [9, 24, 32, 41] inefficient for large-scale GPM workloads. This performance gap has motivated the development of specialized hardware accelerators [11, 14, 26, 31, 44, 49, 51]. While these custom designs show promise, their performance remains constrained by two primary bottlenecks: inefficient set intersection and rigid task scheduling.

Many accelerators adopt a set-centric Depth-First Search (DFS) paradigm [24, 35, 36], where set intersection is the fundamental operation, consuming the majority of execution time [40] to perform the efficient pruning that gives DFS a key advantage over Breadth-First Search (BFS) [10, 46, 47]. However, existing hardware implementations for this operation are either inherently sequential (merge-based) [11, 14, 45] or computationally redundant (systolic-based) [15], limiting throughput. Furthermore, the DFS execution model in prior work enforces strict synchronization barriers between levels of the search tree [11], causing frequent stalls and underutilization of parallel hardware, particularly on graphs with irregular degree distributions.



(a) Example match with "diamond" Pattern (b) Matching plan and restrictions (c) Corresponding GPM algorithm (d) Example search tree for root vertex $u_1 = 4$

Figure 1: Overview of graph pattern matching (GPM) and the set-centric DFS algorithm

Table 1: Theoretical comparison of set intersection architectures, where N denotes the number of elements processed concurrently from two input sets.

Architecture	Throughput (#element/cycle)	Latency (#cycle)	Resource (#comparators)
Merge Queue [11]	1	$O(1)$	$O(1)$
Systolic Array [15]	N	$O(N)$	$O(N^2)$
Our Work	N	$O(\log N)$	$O(N \log N)$

To overcome these limitations, we propose X-SET, a scalable hardware accelerator designed for high-throughput GPM. Our work introduces two fundamental architectural innovations targeting the primary bottlenecks in existing GPM pipelines. First, we introduce a high-throughput, order-aware set intersection unit (SIU) built upon a bitonic merger network. By transforming inputs into a bitonic sequence, our design exploits inherent data ordering to eliminate redundant comparisons. This enables highly parallel intersection with $O(N \log N)$ complexity and $O(\log N)$ latency, granting X-SET a significant performance advantage over conventional merge- and systolic-based architectures, as quantified in Table 1. Second, to overcome the synchronization barriers of traditional DFS accelerators, we developed a barrier-free, out-of-order task scheduler. It tracks data dependencies across the entire search tree and dynamically dispatches any ready task to available SIUs, regardless of its level. This ensures high hardware utilization on irregular workloads and removes performance bottlenecks. We integrate X-SET into a RISC-V SoC via the standard RoCC interface, enabling acceleration of the full GPM workflow without software or compiler modifications and establishing it as a practical and powerful solution. This paper makes the following key contributions:

- We design a high-throughput, order-aware set intersection unit that leverages input order via a bitonic merger network to improve performance and area efficiency over existing designs.
- We propose a dynamic, out-of-order scheduler that removes synchronization barriers in DFS-based GPM, enabling fine-grained parallelism by executing ready tasks from different levels of the search tree concurrently.
- We integrate X-SET into a RISC-V SoC via the RoCC interface, demonstrating a practical path to end-to-end hardware acceleration without specialized compiler or software stacks.
- Our evaluations on real-world graphs show that X-SET achieves up to **142.9**× higher compute density (13.7× geometric mean) and delivers up to **42.9**× end-to-end speedup (6.4× average) over state-of-the-art GPM accelerators.

2 Background and Motivation

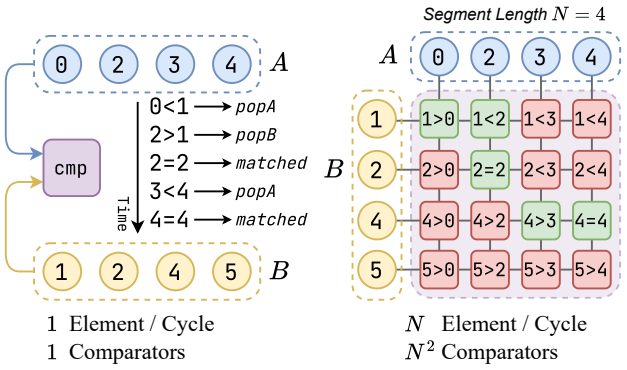
2.1 GPM Definition

Graph Pattern Matching (GPM) is a fundamental task in graph mining that involves identifying and enumerating all subgraphs or embeddings within a larger data graph G that are structurally identical (isomorphic) to a given pattern graph P. Formally, this task requires finding a bijective mapping $f : V(P) \rightarrow V(H)$ for a subgraph $H \subseteq G$, such that the adjacency relationship between vertices is preserved: $(u, v) \in E(P) \Leftrightarrow (f(u), f(v)) \in E(H)$, where $V(G)$ represents vertices of graph G and $E(G) \subseteq V(G) \times V(G)$ represents its edge set. Figure 1a illustrates this concept by showing a data graph G along with a target pattern graph P (a *Diamond* structure). Embeddings within G that match pattern P are highlighted with vertices and edges marked in green: specifically $(1, 2, 0, 3)$, $(2, 3, 1, 4)$, and $(3, 4, 2, 5)$, each of which preserves the connectivity defined by the pattern. While the general GPM problem is NP-complete, in many practical applications, the interested patterns are fixed while the data graph is dynamic [1, 2, 16]. For this scenario, which is our main focus, the problem exhibits polynomial-time complexity, making it amenable to hardware acceleration.

2.2 Set-Centric GPM Algorithm

Recent advancements in GPM systems have embraced a *Set-Centric* DFS programming model [8, 36, 41, 42]. In this model, the system methodically matches vertices in a defined sequence. The potential matches for each vertex, known as its *candidate set*, are determined by performing set operations (such as intersection and difference) on the neighbor sets of vertices that have already been matched, following the pattern's connectivity.

Figure 1b illustrates the corresponding matching plan for the "diamond" pattern depicted in Figure 1a, where $v_i \in V(P)$ denotes



(a) Simple merge queue (b) Systolic merge array [15]

Figure 2: Set intersection unit design of previous works

the vertices of the pattern, $u_i \in V(G)$ represents the vertices to be matched with v_i, and $N(u) \in V(G)$ indicates the neighbor set of u. To avoid enumerating duplicate embeddings due to automorphisms, symmetry-breaking restrictions are incorporated into the matching plan [35]. In the example pattern, v_1 is symmetrical with v_2, as is v_3 with v_4. Consequently, restrictions such as $u_1 > u_2$ and $u_3 > u_4$ are applied, as shown in Figure 1b. These constraints effectively prevent embeddings like $(2, 0, 4, 1)$ from being matched twice in different permutations: $(u_1, u_2, u_3, u_4) = (2, 0, 4, 1)$ and $(u_1, u_2, u_3, u_4) = (0, 2, 4, 1)$. This matching plan with its associated restrictions can be directly translated into nested `for` loops executed sequentially, as demonstrated in Figure 1c.

Figure 1d illustrates the search process starting with root vertex $u_1 = 4$. At level 2, a neighbor u_2 of u_1 is chosen. The candidate $u_2 = 5$ is pruned by the symmetry-breaking constraint $u_2 < u_1$. At level 3, the candidate sets for u_3 and u_4 are computed. As pattern vertices v_3 and v_4 are both connected to v_1 and v_2, their corresponding data vertices, u_3 and u_4, must be selected from the intersection of $N(u_1)$ and $N(u_2)$. For the branch $u_2 = 0$, this intersection $N(4) \cap N(0) = \{2\}$ is a singleton, so no valid pair (u_3, u_4) with $u_4 < u_3$ can be formed. For $u_2 = 2$, the intersection $N(4) \cap N(2) = \{0, 3\}$ yields the valid pair $(u_3, u_4) = (3, 0)$, resulting in the match $(4, 2, 3, 0)$. Similarly, for $u_2 = 3$, the intersection $N(4) \cap N(3) = \{2, 5\}$ produces the pair $(u_3, u_4) = (5, 2)$ and the match $(4, 3, 5, 2)$.

2.3 Challenges of GPM Accelerator Design

Efficient hardware acceleration of GPM is essential due to the computational intensity and irregularity of memory access patterns in graph analytics workloads. These properties often exceed the capabilities of general-purpose processors [8, 11, 15]. However, effective GPM acceleration involves addressing two critical algorithmic components: intensive set intersection computations and efficient task scheduling within a DFS-based search traversal.

Challenge #1: Design of Efficient Set Intersection Unit (SIU) Set operations, including set intersection and set difference, are repeatedly applied to prune candidate vertex sets during the DFS-based traversal, accounting for the majority of GPM execution time (up to 96.4%)[40]. While set difference is often implemented via set intersection using the relation $A - B = A - (A \cap B)$, an efficient set intersection unit (SIU) that compares elements from two

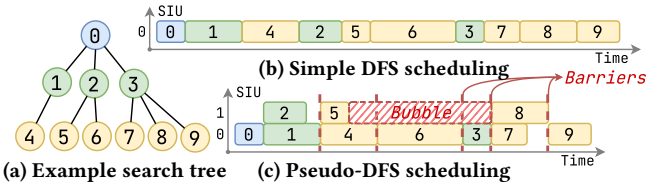


Figure 3: Previous scheduling strategies for GPM

sets to identify common elements is the key to GPM accelerators. However, set intersection process inherently requires sequential comparisons, as the presence of an element in the intersection set depends on its presence in all input sets. Such data dependencies limit opportunities for parallel execution, making it difficult to customize parallel and efficient hardware architectures.

Current designs for SIUs face challenges in optimizing parallelism without sacrificing hardware efficiency. Figure 2 visualizes the computation in existing SIU designs with two example input sequences. Most existing works [11, 14, 45] adopt a simple merge-based SIU (shown in Figure 2a), where elements from two sets ($\{1, 2, 4, 5\}$ and $\{0, 2, 3, 4\}$) are compared sequentially through a single comparison unit. This approach processes one comparison at a time, as shown in the time sequence of operations. While architecturally simple, it results in limited throughput (one element per cycle) due to its inherently sequential nature.

To increase parallelism, DIMMining [15] adopts a systolic merge array that processes segments of input sets concurrently by structuring all possible element comparisons in a systolic array (Figure 2b for segment length $N = 4$). Although this design boosts throughput and scalability, it performs an exhaustive all-to-all comparison, disregarding the inherent order of the input segments. This leads to considerable redundancy, as illustrated by unnecessary comparisons (e.g., if 2 from sequence B is already less than 3 from sequence A, comparing 1 from B with 3 is superfluous). As a result, this method requires N^2 comparators for sequences of length N, escalating hardware complexity and resource usage.

Challenge #2: Efficient Task Scheduling for SIUs

While DFS-based scheduling effectively controls memory usage, it presents significant challenges in leveraging fine-grained parallelism across multiple SIUs due to its irregular memory access patterns and complex control flow. To exploit search-tree-level parallelism, search trees with different root vertices are usually distributed to available processing elements (PEs). However, most existing implementations [14, 15, 42, 45] only equip a single SIU in a PE and execute tasks sequentially, as illustrated in Figure 3b, where the search tree of root vertex 0 (Figure 3a) is executed with a single SIU. As a result, parallelism within the search tree remains unexploited, and the entire PE sits idle while the sole SIU awaits outstanding memory requests, leading to suboptimal utilization.

Although various strategies have been proposed to exploit the fine-grained subtree-level parallelism and achieve better utilization of multiple SIUs, their effectiveness remains constrained. The pseudo-DFS scheduling approach introduced by FINGERS [11] enables concurrent execution of sibling tasks (those at the same level in the search tree) within predefined execution windows, where the window size determines the maximum number of tasks that can be executed in parallel. Nevertheless, this approach requires that

all tasks in the current execution window must complete before tasks in the next window can begin, which introduces additional synchronization barriers.

Figure 3c illustrates this scheduling with an execution window size of two. The task ⑥ must wait for both tasks ④ and ⑤ to finish even when task ⑤ completes early, leaving SIU 1 idle. Furthermore, pseudo-DFS scheduling prohibits the simultaneous execution of tasks from different hierarchical levels within a single execution window. This further reduces SIU utilization when the execution window is under-saturated. For instance, while task ⑥ is executing, SIU 1 remains completely idle, missing the opportunity to process task ③ concurrently.

Shogun [49] builds on FINGERS [11] by introducing an incremental task scheduler for out-of-order execution to reduce inter-depth barriers. However, it retains low-throughput, merge-based SIUs, requiring many SIUs and task dividers, which increases hardware complexity and impacts cache locality because of fine-grained input partitioning. To mitigate this, Shogun's scheduler incorporates centralized control and a locality-aware mode that adds synchronization barriers, essentially restricting parallelism.

3 The Proposed Solution

To address the challenges in GPM accelerator design, we propose an innovative hardware architecture integrated with advanced set operation mechanisms and a barrier-free scheduling approach. We first introduce Order-Aware Parallel Set Intersection Unit to improve the efficiency of computing intersections by exploiting the inherent order within input segments. Then, we propose a barrier-free scheduling method that dynamically manages task dependencies, thereby enhancing parallelism and utilization.

3.1 Order-Aware Parallel Set Intersection Unit

We propose a novel *Order-Aware Parallel Set Intersection Unit* that leverages the inherent order of input data to minimize computational redundancy and hardware complexity. This approach is inspired by two key insights related to the order of sequences.

Insight 1: we can achieve linear complexity for computing set intersection with ordered sequences. This realization comes from observing that the Simple Merge Queue (shown in Figure 2a) closely resembles the merge step of classical merge sort. Formally, let S be a sequence containing all elements from sets A and B, where each element $S_i = (x_i, F_i)$ consists of a value x_i and a flag $F_i \in \{L, R\}$ indicating whether the element belongs to set A (when $F_i = L$) or set B (when $F_i = R$). We define a total ordering on S such that $S_i < S_j \Leftrightarrow x_i < x_j$ or ($x_i = x_j$ and $F_i = L$). When S is sorted according to this ordering, set intersection can be computed by examining adjacent elements: $x_i \in A \cap B \Leftrightarrow x_i = x_{i+1}$ and $F_i \neq F_{i+1}$ and for difference $x_i \in A - B \Leftrightarrow F_i = L$ and ($x_i \neq x_{i+1}$ or $F_{i+1} = L$). This enables us to use linear scan to compute the intersection and difference and those comparisons can be executed in parallel as there are no data dependencies between them.

Insight 2: we can achieve $O(N \log N)$ hardware complexity for sorting two ordered sets in parallel. Leveraging the inherent order of neighbor sets allows for a highly efficient merging process. The key maneuver involves concatenating one set with the reverse of the other, strategically creating a *bitonic sequence*, which

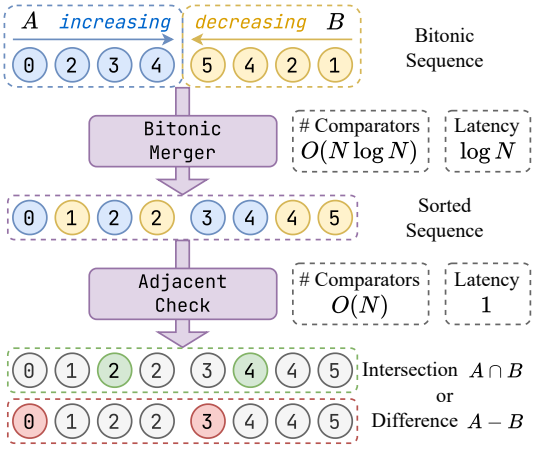


Figure 4: Order-aware SIU

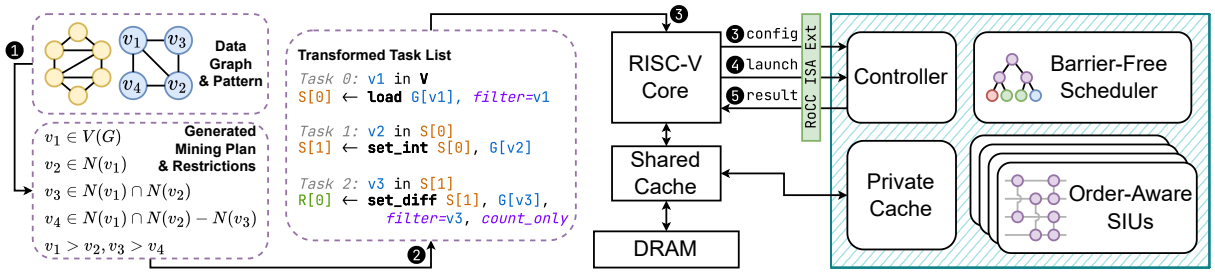
is a sequence that first increases and then decreases. This special structure is perfectly suited for sorting by a Bitonic Merger [33]. The power of the merger lies in its recursive strategy: it repeatedly splits a bitonic sequence of length $2L$ into two smaller bitonic halves, using L comparators to guarantee all elements in the left half are smaller than those in the right. This elegant method sorts the sequence with only $O(N \log N)$ comparators and achieves a latency of $O(\log N)$ cycles [38].

Our Order-Aware SIU architecture is engineered to exploit this insight. The process begins by transforming the input sets into a bitonic sequence. This sequence is then channeled into the Bitonic Merger, which employs a parallel compare-and-swap mechanism for rapid sorting. In the final stage, the set intersection and difference are resolved through parallel comparisons between adjacent elements in the perfectly sorted sequence. To crystallize this concept, consider the example in Figure 4 with sets $A = \{0, 2, 3, 4\}$ and $B = \{1, 2, 4, 5\}$. Denoting an element x_i with its origin flag F_i as $x_i^{F_i}$, we form the bitonic sequence $C_1 = (0^L, 2^L, 3^L, 4^L, 5^R, 4^R, 2^R, 1^R)$ by concatenating A with a reversed B. The Bitonic Merger then transforms C_1 into the sorted sequence $S = (0^L, 1^R, 2^L, 2^R, 3^L, 4^L, 4^R, 5^R)$. From this output, parallel adjacent comparisons cleanly extract the intersection $A \cap B = \{2, 4\}$ and the difference $A - B = \{0, 3\}$.

Our Order-Aware SIU significantly reduces the number of comparators and eliminates redundant operations in systolic merge array [15]. As a result, the Order-Aware SIU achieves $O(N \log N)$ hardware complexity, $O(\log N)$ latency, for N elements per cycle throughput, substantially outperforming traditional merger and systolic array in both efficiency and scalability.

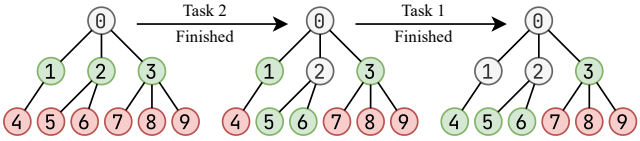
3.2 Barrier-Free Task Scheduler

To maximize parallelism during DFS-based GPM [36], we introduce a barrier-free task scheduler that eliminates global synchronization constraints and enables dynamic, fine-grained task scheduling across SIUs. Unlike previous designs [11, 14] that impose level-wise barriers, forcing all sibling tasks to complete before progressing, our approach dynamically dispatches ready tasks based on actual data dependencies encoded in the task tree. The key insight of our method is that ***the search tree structure naturally encodes task dependencies, allowing tasks to execute as soon as their***

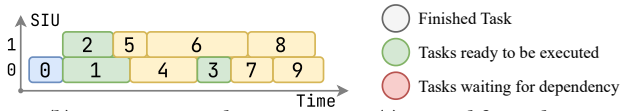


(a) Accelerated GPM workflow with X-SET (b) X-SET process element (PE)

Figure 5: X-SET system overview



(a) Parallelizable tasks with execution progress



(b) Executing timeline (c) Legend for task tree

Figure 6: Barrier-free scheduling for GPM

parent task completes, without waiting for the execution of other branches to reach the same depth.

As illustrated in Figure 6a, GPM computation can be conceptualized as a hierarchical task tree where each node represents a specific set operation. Our barrier-free scheduling approach dynamically manages task execution based on dependency resolution rather than rigid synchronization barriers. The figure demonstrates how tasks transition from waiting (red) to ready (green) states as their dependencies are satisfied. Following the completion of the root task, tasks ①, ②, and ③ become immediately eligible for parallel execution. As these tasks complete, their respective child tasks become available for processing, enabling cross-level parallelization without artificial synchronization points.

Figure 6b presents the execution timeline for the same task tree of our approach, with time progression on the horizontal axis and SIU execution slots on the vertical axis. This visualization highlights the efficiency gains compared to traditional methods shown in Figure 3. For instance,Tasks ⑤ and ⑥ begin immediately after task ② completes, without waiting for other same-level tasks. Similarly, task ④ starts promptly after task ① finishes.

Our barrier-free scheduling algorithm continuously monitors task completion and dependency satisfaction, adapting to the actual computation progress rather than enforcing rigid synchronization points. This adaptive approach is particularly beneficial for GPM workloads characterized by irregular computation patterns and varying task execution times.

4 X-SET Overview

4.1 System Architecture

Figure 5 illustrates the system architecture of X-SET, which is built on a Chipyard-generated SoC [4], where each PE is implemented as a RoCC accelerator alongside a Rocket core [5]. This design supports instruction extensions for efficient software-hardware co-design, managing configuration, task execution, and result collection. As illustrated in Figure 5b, each PE contains specialized components. A Controller interfaces with the RISC-V core via RoCC instructions to manage PE states. Multiple Order-Aware SIUs execute set operations, leveraging data order to optimize performance as elaborated in Section 5. A Barrier-Free Scheduler, detailed in Section 6, dynamically dispatches tasks from the task tree to maximize parallelism, while a Private Cache stores the data graph and intermediate results for access by both the SIUs and the scheduler. All data resides in DRAM and is accessed through a shared cache.

4.2 Execution Flow

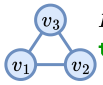
X-SET provides an integrated software-hardware flow for GPM acceleration from generating matching plans to execution on hardware. Figure 5a depicts the workflow with X-SET. The GPM process begins with the generation of a GPM plan using a system such as GraphPi [42] (①). This plan is subsequently transformed into an executable task list (②). Each task within this list is designed to identify a specific vertex of the pattern graph and perform associated set operations, including set intersection (`set_int`), set difference (`set_diff`), and data loading from global memory (`load`). Additionally, each task determines the candidate set for initiating subsequent tasks in the next processing level. Notably, all set operations incorporate functionalities for output filtering based on a defined upper limit and a count-only mode, specifically optimized for pattern counting workloads. Following these preprocessing stages, the RISC-V CPU core initiates the offloading of GPM execution to a dedicated PE through customized RoCC instructions. This offloading process consists of several key steps: first, the data graph and the prepared task list are configured within the PE (③); second, execution is initiated with specification of the maximum vertex to be processed (④); finally, the RISC-V core polls the PE to retrieve the results (⑤).

The corresponding CPU code with RoCC ISA extensions [5, 21] is shown in Figure 7a, where custom instructions (which are bolded and prefixed with `xset_`) are annotated with numbers corresponding to each stage in the workflow. Integration with RISC-V cores enables flexible support for advanced result collection logic, allowing X-SET to handle a wide range of GPM workloads, including matching plans enhanced by Intersection Expression Pruning (IEP) [41, 42]. For instance, while triangle counting (`3CF`, Figure 7b) involves straightforward accumulation, the diamond pattern (`DIA`,

```
xset_setup_addr(addr_arr);
xset_setup_len(len_arr); ❸
xset_setup_plan(task_list);
xset_launch(N_vertices); ❹

int total = 0;
xset_res R;
while(xset_poll_res(&R)) ❺
  { total += R[0]; }
```

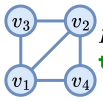
(a) Example CPU code



$R_0 = |N(v_1) \cap N(v_2)|$

```
total += R[0];
```

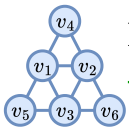
(b) Result collection for "3CF"



$R_0 = |N(v_1) \cap N(v_2)|$

```
total += R[0] *
  (R[0] - 1) / 2;
```

(c) Result collection for "DIA"



$R_0 = |N(v_1) \cap N(v_2)|$ $R_1 = |N(v_1) \cap N(v_3)|$
$R_2 = |N(v_2) \cap N(v_3)|$ $R_3 = |N(v_1) \cap N(v_2) \cap N(v_3)|$

```
total += (R[0] - 1) * (R[1] - 1) * (R[2] - 1)
       - R[3] * (R[0] + R[1] + R[2] - 5);
```

(d) Result collection for "TRI6"

Figure 7: Example CPU code for offloading GPM to X-SET with RoCC extensions and supported IEP-enabled result collection for different patterns

Figure 7c) requires more sophisticated result collection. In this case, both v_3 and v_4 must connect to v_1 and v_2, and the result is computed as $|S| = |\{(v_3, v_4) \mid v_3, v_4 \in N(v_1) \cap N(v_2), v_3 < v_4\}|$, which simplifies to $A(A-1)/2$, where $A = |N(v_1) \cap N(v_2)|$. Even more complex logic, including arbitrary IEP expressions, can be easily supported by X-SET, as demonstrated by the pattern `TRI6` in Figure 7d with the result collection expression introduced by GraphSet [41].

Another key advantage of integrating a RISC-V core with an X-SET PE is the ability to handle patterns with arbitrary size. For complex patterns with a vertex count beyond the hardware scheduler's limit, the CPU executes the initial stages of the GPM plan to generate and refine candidate sets, which are then delegated to the PEs for accelerated analysis of the remaining stages. This tight integration ensures greater flexibility in handling diverse GPM workloads compared to conventional hardware accelerators [11, 14, 15, 49].

5 Order-Aware Set Intersection Unit

The Order-Aware Set Intersection Unit (SIU) forms the core of our proposed architecture, designed to efficiently process set intersection and difference on ordered neighbor sets of arbitrary length. In this section, we provide a comprehensive overview of its micro-architecture, followed by a detailed explanation of how to exploit the internal order of inputs to achieve efficient processing.

5.1 Micro-Architecture Overview

Figure 8 illustrates the micro-architecture of our proposed Order-Aware SIU. The architecture comprises four primary components supporting the core pipeline: control unit (`Ctrl`), memory requester (`Mem`), input filter, and buffers for both input and output data.

The Control unit orchestrates the pipeline by issuing memory requests through the Memory Request module (①) to fetch neighbor sets. The Input Filter (③), configured by the Control unit (②), performs initial filtering to reduce computation by eliminating vertices and breaking pattern symmetry—a crucial factor for GPM efficiency [35]. Filtered data is then segmented and fed into N parallel input FIFOs (④) in round-robin, where the k-th FIFO stores elements $k, k+N, \ldots$. This fully pipelined approach supports arbitrary set lengths. After processing by the core pipeline, results are aligned

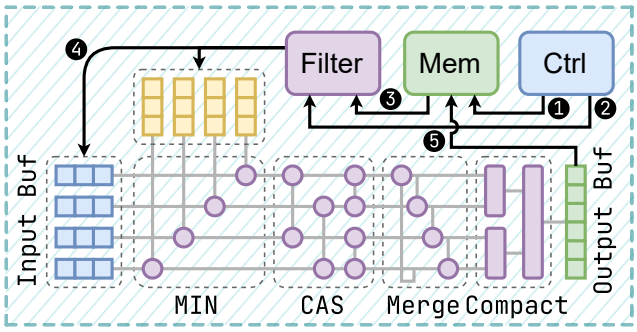


Figure 8: Architecture of order-aware SIU, with $N = 4$

into segments by the Output Buffer (⑤) for efficient write-back via the Memory Request module. N represents the pre-configured segment length, balancing both throughput and hardware complexity. For clarity, we use $N = 4$ in this section, while $N = 8$ is used in evaluation to match DRAM access granularity.

5.2 Core Processing Pipeline

The core of our Order-Aware SIU is a novel, fully-pipelined streaming architecture designed to process sets of arbitrary length. Instead of directly adopting a standard bitonic sorter, we leverage a bitonic merger network [38], which uses MIN and CAS stages to merge two input sets into a single sorted stream in segments. This stream is then processed by a parallel Merge stage for adjacent vertex comparison and BitmapCSR updates, followed by a Compact stage to produce a dense output. This segmented, streaming approach ensures scalability to arbitrary set sizes.

BitmapCSR is a hybrid format where each 32-bit element represents multiple vertices. This is achieved by dividing the element into an b-bit bitmap (v) and a $32-b$-bit index (k). For a given vertex x, the index is set to $k = x/b$, and the corresponding bit in the bitmap is set to 1 at position $x \pmod b$. This allows a single element to represent up to b distinct vertices, leading to optimized storage memory usage and intra-element parallelism.

Figure 9 illustrates the detailed dataflow of the core pipeline when processing the intersection of two sets $A = \{0, 1, 3, 4, 5, 6, 7\}$ and $B = \{0, 2, 3, 6, 7\}$ with segment length $N = 4$. Note that the following mentioned comparisons are all based on the index of the BitmapCSR element, which further reduces the width to compare for comparators and reduces hardware area.

5.3 Exploiting Order in Input Sets

The order inherent in the input sets is exploited by the Bitonic Merge network to efficiently sort two ascending input sets, enabling subsequent parallel set operations on sorted elements.

5.3.1 MIN Stage. The first stage of the bitonic merge network is the MIN stage, designed to extract the smallest N elements from the two sorted input sets and form output as a bitonic segment, preserving order information of original inputs. The MIN stage compares the top element of the i-th ($1 \le i \le N$) FIFO of the first input set with the top element of the $(N-i+1)$-th FIFO of the second input set and outputs the smaller one as the i-th output element. Formally, the output segment is $C_i = \min\{A_i, B_{N-i+1}\}$ for two input segment A, B. The resulting segment is bitonic because

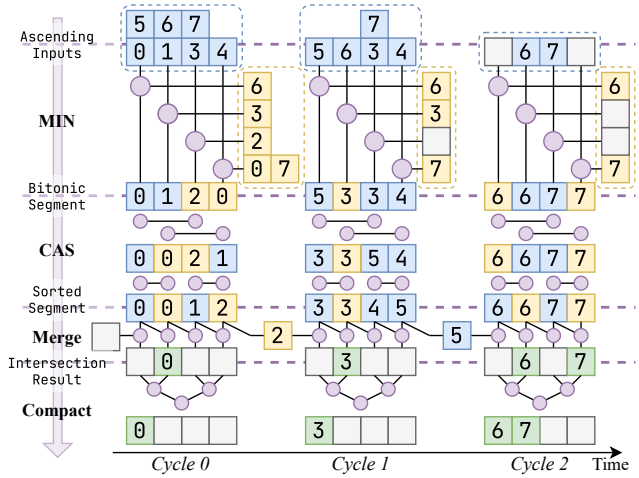


Figure 9: Example of order-aware SIU processing intersection of $A = \{0, 1, 3, 4, 5, 6, 7\}$ (top) and $B = \{0, 2, 3, 6, 7\}$ (right)

MIN stage always extracts consecutive elements from inputs and appends them in reversed order.

As shown in Figure 9, in cycle 0, the MIN stage compares 0 from A with 6 from B and outputs 0, compares 1 from A with 3 from B and outputs 1, and so on. Finally, $(0, 1, 2, 0)$ are selected as the smallest-4 elements and are popped out from the input buffer. Other elements remain in the input buffer, and new elements become the top elements. Because the input data are pushed to FIFOs in a round-robin manner, the input segment of A for cycle 1 becomes $(5, 6, 3, 4)$ and that of B becomes $(6, 3, \times, 7)$, where $\times$ represents an empty element and is treated as larger than any valid element. The output of the MIN stage in cycle 0 is $(0, 1, 2, 0)$, which is bitonic because it increases first then decreases. This property enables efficient processing in the following CAS stage.

5.3.2 CAS Stage. The second stage is the CAS (compare and swap) stage, designed to split one bitonic segment into two halves of bitonic segments with the same length, where all elements of one half are smaller than those of the other half. When the subsegments contain only one element each, the whole segment is fully sorted. For the CAS stage processing L elements, it uses $L/2$ comparators to perform CAS operations on the i-th and $(i + L/2)$-th elements ($i \in \{1, \cdots, L/2\}$) in parallel.

As shown in Figure 9, at cycle 0, the first CAS stage uses 2 comparators to split the bitonic segment $(0, 1, 2, 0)$ into two bitonic sub-segments $(0, 0)$, $(2, 1)$. Next, two CAS stages with 1 comparator each are recursively applied to the two sub-segments, generating the final sorted segment $(0, 0, 1, 2)$. The utilization of bitonic properties across multiple CAS stages enables efficient sorting with only $N \log_2 N$ comparators and $\log N$ cycles.

Furthermore, the standard CAS operations are enhanced with a flag m_i that indicates whether the i-th element matches another element during sorting. Specifically, when a CAS unit is applied to elements x_i and x_j, their corresponding flags are updated according to the formula $m_i' = m_i \vee (x_i = x_j), m_j' = m_j \vee (x_i = x_j)$. This flagging mechanism efficiently eliminates the need for N additional comparators in the final merge stage.

5.4 Parallel Set Operation on Sorted Segments

5.4.1 Merge Stage. After sorting, the Merge stage efficiently performs set operations on the sorted segments by leveraging the match flag from the CAS stage. When an element from the first input set has its match flag set, it is guaranteed to match with the next element in the sorted segments. This optimization is significant as it replaces the N comparators with N 1-bit multiplexers for final adjacent checking, substantially reducing hardware costs.

A key exception in this process is the comparison between the last element of the current segment and the first element of the next, which the match flag does not support. To address this, a single register preserves the last element for this subsequent comparison. As shown in Figure 9, element 2 from cycle 0 is retained for comparison with element 3 at cycle 1, and element 5 from cycle 1 is held for element 6 at cycle 2. This mechanism ensures continuous and accurate set operations across segment boundaries.

Additionally, the Merge stage is designed to handle bitmap updates for the BitmapCSR format. If adjacent elements have equal indices, the Merge stage applies a bitwise AND to their bitmaps for intersection, or a bitwise AND between the bitmap of the first element and the negation of the bitmap of the second for difference.

5.4.2 Compact Stage. Finally, a compact stage removes any empty elements from the output of the Merge stage and generates consecutive output data to be aligned by the circular output buffer. Our compact stage is implemented as an efficient binary-tree-like recursive reducer, with additional accumulation functions for element count and segment length. Compared with the systolic merge array, which is constrained by the timing of the systolic array and requires $N^2/2$ hardware resources and N cycles of latency for the final compact triangle, our compact stage only requires $\log_2 N$ cycles of latency and $N \log_2 N$ hardware resources, making it more efficient and scalable for the compact operation.

6 Barrier-Free GPM Scheduling

To fully exploit parallelism within the DFS tree for GPM, we propose a barrier-free scheduler that eliminates traditional synchronization constraints while dynamically managing parallel tasks. The proposed barrier-free scheduler architecture, illustrated in Figure 10, comprises hierarchical structures to manage data dependencies of tasks and key components to execute tasks in parallel.

6.1 Task Management and Spawning

At the core of our scheduler is the Task Tree, shown in Figure 10a, which implements a hierarchical organization of tasks arranged in layers from 0 to N. Layer 0 serves as the root of the tree, with each subsequent layer containing one or more Task Sets. This hierarchical structure enables efficient management of complex task dependencies and parallel execution. The Task Set is the fundamental component of dependency management within the task tree. Each Task Set is assigned to one parent task and represents all its subtasks. When a task completes its execution with a non-empty candidate set, an available Task Set is allocated to it and configured to spawn tasks with the vertices in the candidate set. This mechanism ensures proper task succession and dependency resolution throughout the execution process. As illustrated in Figure 10b, each

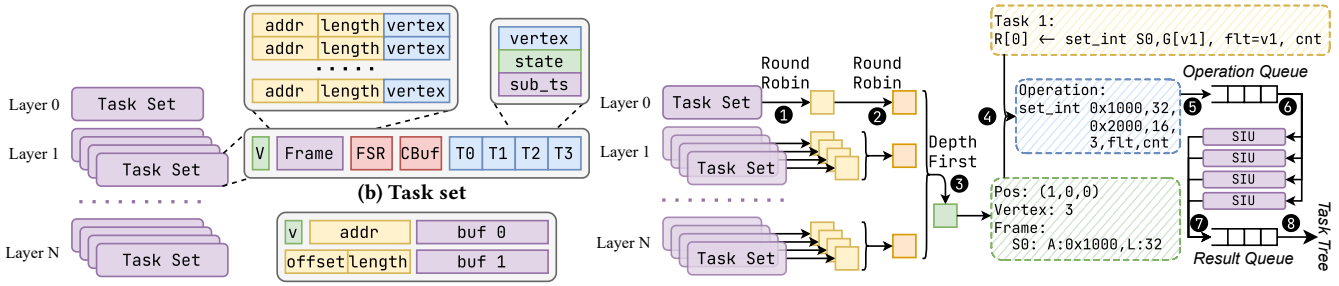


Figure 10: Key structures and processes of barrier-free scheduler

Task Set contains several key components. A valid bit (V) indicates whether this task set can be allocated. The Task Frame (Frame) tracks information of intermediate sets and vertices. For efficient task creation, the Task Set includes a Fast Spawning Register (FSR) and a CBuf index (CBuf). Additionally, each Task Set maintains a list of subtasks (v0-v3) with their respective status, vertex, and sub-task set ID stored for tracking purposes.

The Candidate Buffer (CBuf), depicted in Figure 10c, plays a crucial role in the task spawning process. It consists of multiple items, each comprising a valid bit (V), address and length information of the candidate set, and a ping-pong buffer for storing segments of the candidate set. When a Task Set is allocated for a parent task, an available CBuf item is also assigned and configured with the candidate set information. To optimize performance, the CBuf continuously fetches segments to empty ping-pong buffer slots, effectively overlapping the latency of data fetching with task spawning operations. For symmetric breaking in Graph Pattern Matching (GPM), the system employs an upper limit filter (Flt), which functions similarly to the one in Order-Aware SIU. Furthermore, to leverage the BitmapCSR format during task spawning, the Fast Spawning Register (FSR) stores one BitmapCSR element fetched from CBuf. The system first checks the FSR for available vertices to spawn tasks, then unsets the corresponding bit in its bitmap after spawning. Only when the FSR's bitmap becomes empty does the CBuf receive requests for subsequent elements, ensuring efficient resource utilization and task management.

6.2 Task Issuing and Execution

The scheduler employs a sophisticated Task Issuing Policy, shown in Figure 10d, which exploits the parallelism inherent in the task tree structure while maintaining cache locality. At the Task Set level, each Task Set selects one of its ready subtasks for issuing in a round-robin manner (①). At the Layer level, further round-robin selection is made among valid task sets (②). This round-robin policy applied at these two levels maintains cache locality, ensuring that tasks with similar data dependencies are executed in close proximity. However, at the task tree level, a Depth-First policy is used for final selection to minimize the memory footprint.

The Operation Dispatcher, depicted in Figure 10e, accepts the task selected by the task tree and transforms it into set operations before dispatching them to available SIUs. As shown in Figure 10e, the position of issued task (1,0,0) indicates it is from the second layer. The dispatcher reads the task information of corresponding layer, which is R[0] <- set_int S0, G[v1], filter=v1, count_only. It means that the task contains one set intersection operation with intermediate set S0 and global neighbors set of v1, with v1 as filter and counting the result elements only. The information of S0 is obtained from the frame with the task, and the information of the neighbor set is fetched from memory through the cache. Finally, the generated operation set_int 0x1000, 32, 0x2000, 16, 3, flt, cnt(④) is pushed to the operations queue (⑤), waiting for available SIUs to fetch and execute in parallel(⑥). The execution results are pushed to the Result Queue (⑦) and then committed back to update the Task Tree (⑧). This isolated operations dispatching and result collection allows the scheduler to execute tasks asynchronously, eliminating the need for synchronization barriers that typically hinder performance in GPM systems.

By systematically removing synchronization barriers, our scheduler significantly reduces idle time, improves the overall utilization of SIUs, and maximizes throughput for GPM operations. Our architecture design principles prioritize continuous computation flow, efficient resource utilization, and minimal waiting periods, making it particularly well-suited for complex GPM workloads that traditionally suffer from synchronization overhead.

7 Evaluation

7.1 Experimental Setup

7.1.1 Implementation and Configurations. Table 2 summarizes the system configurations for evaluation. We construct a detailed simulation framework consisting of the following components: (1) We build a cycle-accurate System-C-based event-driven simulator to reflect the key latency of hardware modules. (2) To model realistic memory performance, we integrate DRAMSys 5.0 [43], a cycle-accurate DRAM simulator. (3) We use CACTI [7] to model the latency, power, and area of cache in 32nm process node. (4) We implement the hardware design of our accelerator using Chisel [6] within the Chipyard [4] framework and synthesize the generated Verilog RTL using Synopsys Design Compiler under TSMC 28nm technology to obtain area and power metrics at 1GHz. We maintain similar configurations as in baselines to enable fair comparisons.

To facilitate performance evaluation on large, real-world data graphs, we employ a cycle-accurate SystemC-based simulator. This approach is necessary because RTL simulation is prohibitively slow for such workloads. Our simulator is approximately 200 times faster

Table 2: System configuration

PE	
#PE	16
Order-Aware SIU	4 SIUs per PE, Input width 8
Scheduler	TaskSet width 4, #TaskSet 96
Memory subsystem	
Private Cache	32KB per PE, LRU, 4 Banks, 4 Ways
Shared Cache	4MB total, LRU, 8 Banks, 8 Ways
Main Memory	16GB, 4 Channel, DDR4-2400 (76.84GB/s)
DDR Timing	(CL-tRCD-tRP) 16-16-16

Table 3: Graph datasets used in evaluation

Dataset	# Nodes	# Edges	Avg Deg.	Max Deg.	Skew
p2p-Gnutella04 (PP)	1.09E+4	4.00E+4	3.68	103	2.15
WikiVote (WV)	7.12E+3	1.04E+5	14.57	1065	5.14
AstroPh (AS)	1.88E+4	1.98E+5	10.55	504	3.85
MiCo (MI)	9.66E+4	1.08E+6	11.18	936	8.48
Youtube (YT)	1.13E+6	2.99E+6	2.63	28754	**232**
Patents (PA)	3.77E+6	1.65E+7	4.38	793	6.75
LiveJournal (LJ)	4.85E+6	6.90E+7	14.23	20333	30.9

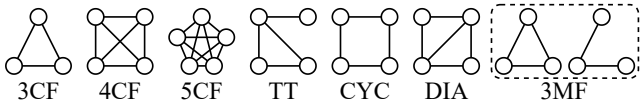


Figure 11: Patterns used in evaluation

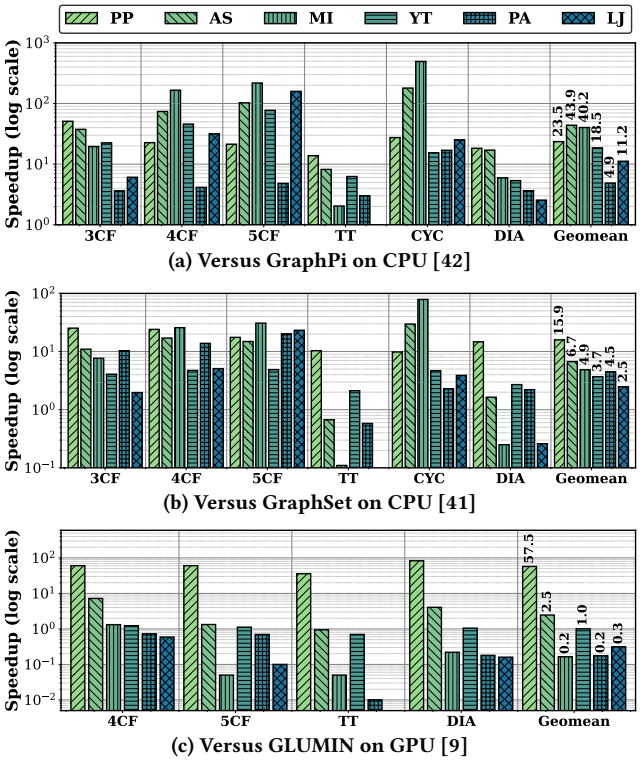
than the RTL equivalent and its fidelity is ensured through cross-validation against the RTL simulation using small test cases. While FPGA-accelerated RTL simulation, such as FireSim [27], presents a promising alternative, its integration remains future work.

7.1.2 Datasets and Benchmarks. We select widely adopted, real-world graph datasets and patterns that are excessively used in evaluating the performance of GPM systems in prior works. Graph datasets are listed in Table 3, where "MiCo" dataset is from [17], while others are from [30]. The *skew* column represents the measure of the asymmetry of degree distribution [25]. Figure 11 shows graph patterns used in the evaluation, which includes triangle (3CF), 4-clique (4CF), 5-clique (5CF), tailed-triangle (TT), 4-cycle (CYC), diamond (DIA). We also use 3-motif finding (3MF) to demonstrate our capability for multi-pattern matching.

7.1.3 Baselines. We compare X-SET against state-of-the-art CPU, GPU and accelerator-based GPM systems, respectively.

For **CPU baselines**, we compare X-SET against **GraphPi** [42], which effectively eliminates redundant computations, and **GraphSet** [41], which employs set-based transformations for mining plan optimization. For a fair comparison, we evaluated the open-source implementations of both baselines on a 96-core AMD EPYC 9654 processor with 1.5TB of DDR5-4800 memory and 384 MB shared cache. The max memory bandwidth is 921.6 GB/s. All cores were utilized at 3.55 GHz, with hyper-threading disabled to prevent performance degradation, consistent with the original studies [41, 42].

For the **GPU baseline**, we compare our approach against **GLUMIN** [9], which uses fast bitwise operations on dynamically generated Look-Up Tables. It is evaluated on an NVIDIA RTX 6000 Ada GPU with 48GB GDDR6 memory and 960 GB/s bandwidth.



(a) Versus GraphPi on CPU [42]

(b) Versus GraphSet on CPU [41]

(c) Versus GLUMIN on GPU [9]

Figure 12: Speedup of X-SET compared to software baselines

For **GPM accelerators**, we compare with **FlexMiner** [14], **FINGERS** [11], and **Shogun** [49]. These accelerators feature similar memory configurations, including 32KB private cache per PE, 4MB shared cache, and 4-channel DDR4-2666 memory providing 85GB/s maximum bandwidth, which is slightly larger than ours. FlexMiner deploys 40 PEs, while FINGERS and Shogun utilize 20 PEs to utilize available bandwidth. Performance data for FlexMiner is sourced directly from its original paper, while results for FINGERS and Shogun are computed based on their reported relative speedup over FlexMiner. We also compare our order-aware SIU with Systolic Merge Array in DIMMining [15] and Merge Queue in FINGERS [11]. We exclude DIMMining from end-to-end comparisons due to its near-memory architecture (based on DIMM modules), which operates under fundamentally different memory and bandwidth assumptions than other in-SoC accelerators.

7.2 Overall Performance Comparison

7.2.1 Comparison with software baselines. Figure 12 shows performance comparison with three software baselines across six real-world graphs and various pattern queries. All speedups are plotted on a logarithmic scale. As detailed in Figure 12a, X-SET significantly outperforms Graph-Pi [42], achieving speedups up to 494.26× (on 5CF) and geometric mean speedups from 4.9× (PA) to 43.9× (AS), due to its order-aware set operations and barrier-free scheduling. Against GraphSet [41], it maintains a competitive geometric mean speedup of 2.5× (LJ) to 15.9× (PP) (Figure 12b). The PA benchmark exhibits a modest speedup compared to GraphPi for two primary reasons. First, the graph's large working set exceeds shared cache capacity. Second, the high sparsity of the graph, as detailed in Table

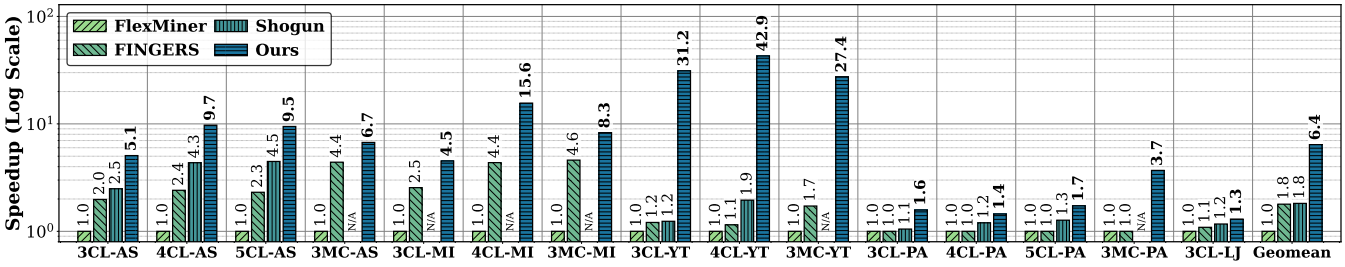


Figure 13: Speedup compared with FlexMiner, FINGERS, and Shogun

3, diminishes the throughput advantages of our specialized SIU hardware.

Figure 12c reveals that X-SET outperforms the GPU-based system GLUMIN [9]. The efficiency of our specialized hardware is highlighted by the fact that **X-SET achieves a 1.05× geometric mean speedup across all benchmarks while utilizing less than 10% of the memory bandwidth of the GPU.** We observe a trend where the performance advantage of X-SET diminishes for larger graphs. This is because larger workloads allow the GPU to more effectively saturate its massively parallel cores and leverage its high-bandwidth memory. It should also be noted that for the MI and PA graphs, the speedup is less pronounced. This outcome is explained by that the maximum vertex degrees in these specific graphs fall below the limit for GLUMIN's warp-level LUT generation, thereby allowing for enhanced parallelism on the GPU.

In summary, X-SET substantially outperforms state-of-the-art CPU and GPU-based systems by **43.9**× and **1.05**× correspondingly, while using less than 10% memory bandwidth than GPU.

7.2.2 Comparison with GPM accelerators. Figure 13 shows a comparative performance analysis of FlexMiner, FINGERS, Shogun and our X-SET, measured in speedup of logarithmic scale relative to FlexMiner. The x-asix represents the pattern-graph combination, sorted firstly by graph size then by pattern. We only compare data points with absolute time from FlexMiner paper [14] because FINGERS [11] and Shogun [49] only provide relative speedup.

X-SET consistently achieves the highest speedup for all benchmarks, with particularly impressive performance on 4CL-YT with maximum 42.9× speedup. Competing accelerators achieve lower performance due to their reliance on simple merge-based SIUs, which limit throughput. X-SET also performs better on more sparse graphs, benefiting from the ability of our barrier-free scheduler to sustain high SIU utilization under irregular workloads.

The proposed accelerator significantly improves performance on the AS and MI datasets. Their small working sets fit entirely within the shared cache, enabling the Order-Aware SIUs to exploit data locality for higher throughput. The YT graph shows exceptional speedup due to its highly skewed degree distribution and large maximum degree. This allows the Order-Aware SIU to capitalize on long input streams and the barrier-free scheduler to efficiently manage the resulting sparse task trees. Conversely, the largest graphs, PA and LJ, exhibit more modest speedups. PA's low average and maximum degree limit the Order-Aware SIU's throughput, though our scheduler's effectiveness is still evident. LJ's performance is constrained by its large working set, which exceeds the shared cache and increases memory pressure.

Table 4: Area comparison of a single PE with previous works

	Tech	Total	Control	Compute	Cache
FINGERS [11]	28nm	0.934	0.069	0.115	0.332
Shogun [49]	28nm	0.971	0.106	0.115	0.332
FlexMiner [14]	15nm	0.180	Area breakdown not provided		
Ours	28nm	0.305	0.044	0.077	0.174

In summary, X-SET achieves speedups of 1.3×-42.9× compared to FlexMiner, 1.2×-37.3× over FINGERS, and 1.1×-25.1× when measured against Shogun. When evaluated using geometric mean, X-SET's performance advantages remain substantial, with 6.4× over FlexMiner, 3.6× compared to FINGERS, and 2.9× over Shogun.

7.3 Area and Compute Density Comparison

7.3.1 Area comparison. As detailed in Table 4, our PE, synthesized using Synopsys Design Compiler with a TSMC 28nm library at 1GHz, occupies a total area of 0.305 mm^2. The cache is the largest component at 57.0% of the PE area. The order-aware SIUs and the barrier-free Scheduler account for 25.4% (0.077 mm^2) and 14.4% (0.044 mm^2), respectively. For comparison, the control unit in FINGERS is 0.069 mm^2, and Shogun's optimized scheduler is 0.037 mm^2. Our barrier-free Scheduler, benefiting from high-throughput SIUs and a barrier-free design, achieves a 36.2% area reduction compared to FINGERS. While larger than Shogun's scheduler, our approach holistically considers the entire PE architecture for improved overall efficiency. The area and power analysis in Table 4 intentionally excludes the Rocket core. In our design, the PE operates as a co-processor, while the Rocket core functions as a controller. The Rocket core's role is confined to preparing the data payload and offloading the primary computation to the PE, after which it awaits the result. This architectural choice justifies a direct comparison of a single PE. Consequently, Figure 13 also omits the performance of the advanced IEP-enabled mining plan, as this process requires active processing by the Rocket Core.

7.3.2 Compute density comparison. As to compute density, while FlexMiner [14] reports a smaller per-PE area, it requires 40 PEs to achieve its reported performance. When examining performance per area metrics, X-SET delivers between 1.9× and **63.3**× speedup over FlexMiner with a geometric mean of **9.5**×, even without accounting for the advanced 15nm fabrication technology utilized by FlexMiner. Furthermore, X-SET's area efficiency extends beyond comparisons with other systems. X-SET demonstrates 4.6× to **142.9**× speedup over FINGERS with a geometric mean of **13.7**×. Similarly, when compared to Shogun, X-SET achieves 4.4× to **99.9**× speedup with a geometric mean of **11.5**× for performance per area.

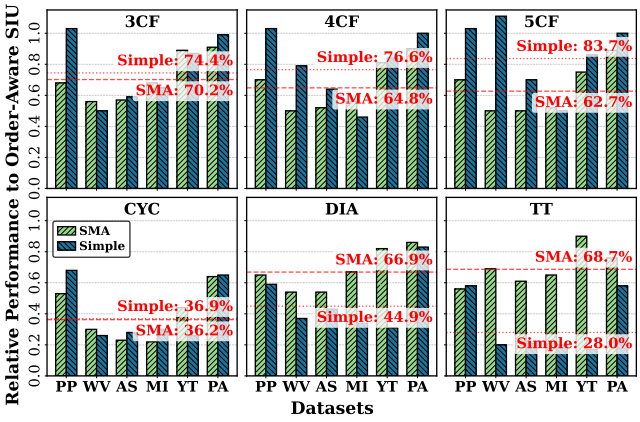


Figure 14: Performance of simple merge-based SIU and SMA [15] normalized to order-aware SIU

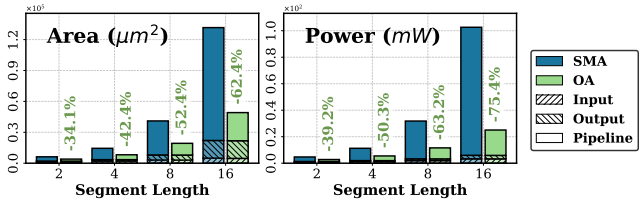


Figure 15: Area and Power breakdown of Order-Aware SIU and Systolic Merge Array for different segment length

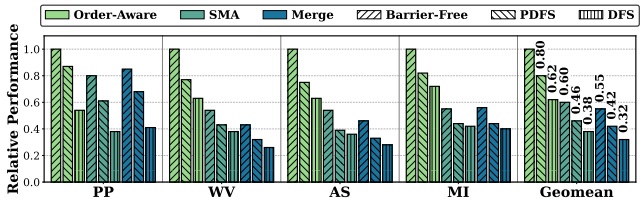


Figure 16: Performance comparison among different SIU and scheduler combinations

7.4 Evaluation on Order-Aware SIU

7.4.1 Performance. Figure 14 compares the end-to-end throughput of our Order-Aware SIU against Systolic Merge Array (SMA) from DIMMining [15] and simple merger adopted by many accelerators [11, 14]. To isolate each design's behavior, the evaluation uses a single PE and SIU. For a fair comparison, all designs incorporate BitmapCSR with a bitmap width of 8. Both the SMA and our architecture use a segment length of eight. Each subfigure shows the results for different patterns on a specific data graph. The efficiency of hardware-accelerated set operations depends on the input set length. Short sets are latency-sensitive due to pipeline setup overheads, while long sets are throughput-sensitive and require sustained processing. For instance, a merge-based SIU offers low latency, performing well on small neighbor sets in low-degree graphs (e.g., PP, WV, PA) or simple patterns (e.g., 3CF, 4CF), but its one-element-per-cycle limit makes it inefficient for large sets. In contrast, the SMA is optimized for high throughput but has higher setup latency, making it better for large sets and complex patterns (e.g., CYC, DIA, TT). Our Order-Aware SIU is architected to balance both latency and throughput and employs an $O(\log N)$ stages pipeline to deliver a throughput of N elements per cycle. This design ensures robust performance across diverse workloads. As a result, our Order-Aware SIU consistently superior performance across nearly all test cases, yielding an average speedup of 1.64× over SMA and 1.9× over a baseline Merge Queue on end-to-end GPM workloads.

7.4.2 Area and Power. Figure 15 compares the area and power consumption of our Order-Aware SIU (OA) and the Systolic Merge Array (SMA) across segment lengths of 2, 4, 8, and 16. Both total area and power are broken down into input, output, and processing pipeline components, with input/output costs held constant for each segment size. As shown in the left panel, the Order-Aware SIU consistently achieves better area efficiency than SMA, with reductions ranging from **34.1%** (at segment length 2) to **62.4%** (at length 16). Power improvements are even more pronounced, reaching **75.4%** at segment length 16, as seen in the right panel. These benefits stem from our SIU's $O(N \log N)$ hardware complexity, in contrast to SMA's $O(N^2)$. As a result, the Order-Aware SIU achieves **performance-per-area gains of 2.3×–9.5×** over SMA, and **3.0×–12.2×** over the Simple Merge Queue, with geometric mean throughput improvements of **3.5×** and **4.5×**, respectively.

7.5 Ablation Analysis

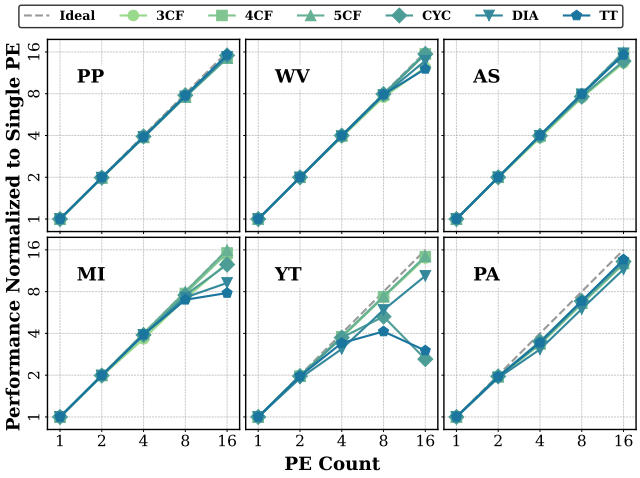
Figure 16 presents an ablation study evaluating nine configurations, combining three distinct SIU implementations (our Order-Aware SIU, SMA, and a simple merger) with three different scheduler policies (our barrier-free scheduler, DFS, and pseudo-DFS). Performance is normalized to the configuration with both of our optimizations.

The results underscore the criticality of both components. When using our Order-Aware SIU, performance degrades to 0.80× with the pseudo-DFS and further to 0.62× with a conventional DFS. Similarly, when paired with our barrier-free scheduler, replacing the SIU with an SMA or a simple merger reduces performance to 0.60× and 0.55×, respectively. Notably, the performance bottleneck from a suboptimal scheduler is nearly identical to that from a suboptimal SIU; using the conventional DFS with our advanced SIU (0.62×) is as detrimental as using the SMA-based SIU with our advanced scheduler (0.60×), demonstrating the effectiveness and robustness of our proposed optimizations.

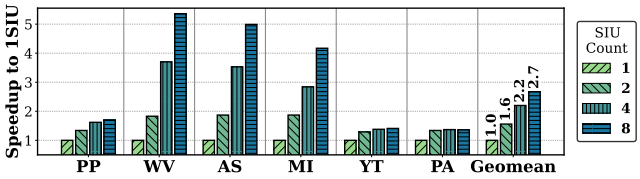
Furthermore, the relative impact of each optimization is data-dependent. For the small and sparse PP graph, performance is more sensitive to the scheduler policy, as the workload is not demanding enough to require a high-throughput SIU. Conversely, the larger and more complex WV, AS, and MI graphs are more profoundly impacted by the SIU implementation, as their execution is bound by the throughput of set operations.

7.6 Scalability Analysis

7.6.1 Number of PEs. Figure 17a illustrates the scalability of our design as the number of PEs increases exponentially. Each subgraph shows performance trends for different dataset-pattern pairs, with curves distinguished by color and markers. X-SET demonstrates near-linear performance scaling with increasing PEs for most test-cases, particularly evident for datasets PP, AS, WV, and patterns 3CF, 4CF, and 5CF. However, scalability degrades with complex patterns on the YT dataset. The YT graph is highly *skewed* as shown in Table 3, meaning its data is unevenly distributed. This causes



(a) Performance with increasing the number of PEs



(b) Performance with increasing the number of SIUs per PE

Figure 17: Scalability of X-SET

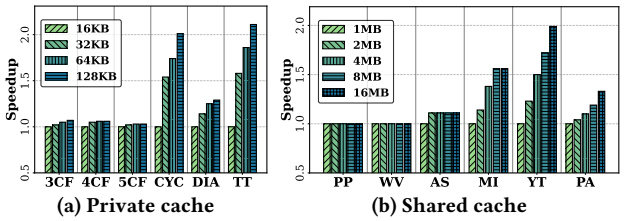
set difference (like in mining CYC and TT) to generate very large intermediate results. These temporary results then compete with the global graph data for limited cache memory, creating *cache contention* that degrades performance.

7.6.2 Number of SIUs per PE. Figure 17b demonstrates the effect of increasing SIUs per PE, with bank count and private cache size scaled accordingly to maintain sufficient bandwidth. The speedup correlates strongly with computational intensity (average degree in Table 3). High-degree datasets (AS, MI, WV) achieve substantial speedups of 2.8× to 3.7× with 4 SIUs per PE, while sparser graphs (PP, PA, YT) show more modest gains of 1.4× to 1.6×. Overall, the four-SIU configuration delivers an average speedup of 2.2× across all datasets, optimally utilizing the 8-bank private cache in each PE and highlighting the effectiveness of the barrier-free scheduler.

7.7 Sensitivity Analysis

7.7.1 Impact of cache size. Figure 18a shows impact of private cache size, where each data point represents the geometric mean speedup across six datasets for a given pattern. Patterns such as 3CF, 4CF, 5CF, and DIA show minimal sensitivity to cache size, while CYC and TT see over 2× speedup when increasing private cache from 32KB to 128KB. This is because restrictive patterns generate smaller intermediate data, imposing less cache contention.

Figure 18b evaluates impact of shared cache, with each data point representing the geometric mean speedup across six patterns for a given dataset. Here, cache sensitivity is more dependent to dataset. Datasets like PP and WV fit well in even 1MB cache, showing stable performance. Others such as AS and MI reach peak performance at 2MB (1.11×) and 8MB (1.56×) respectively, while YT continues to benefit from increased cache capacity, indicating a large working



(a) Private cache (b) Shared cache

Figure 18: Sensitivity of private and shared cache size

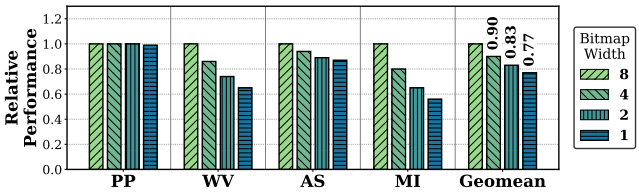


Figure 19: Sensitivity of bitmap width of BitmapCSR

set. Overall, our default configuration of 4MB shared cache offers a balanced trade-off between performance and area efficiency.

7.7.2 Bitmap width in BitmapCSR. Figure 19 demonstrates the influence of the bitmap width parameter on the performance of the BitmapCSR format. The y-axis represents performance relative to our default 8-bit width configuration. Each result is the geometric mean of performance across six distinct patterns for a given data graph. A bitmap width of 0 corresponds to the conventional CSR format. The results show that performance generally increases with the bitmap width. This improvement is credited to the increased intra-element parallelism enabled by a wider bitmap within a 32-bit word. However, the magnitude of this performance gain is affected by the sparsity of the data graph. While the adoption of BitmapCSR consistently shows a performance improvement at a geometric mean of 1.30× speedup compared to CSR format, its overall impact is relatively modest because real world graphs are highly sparse.

8 Conclusion

This paper presents X-SET, a high-throughput and scalable accelerator for GPM. It features two key innovations: an Order-Aware Set Intersection Unit (SIU) that exploits the inherent ordering of input sets to minimize redundant comparisons and a Barrier-Free Task Scheduler that enables fine-grained, out-of-order execution of DFS-based tasks across multiple processing elements. Integrated on a RISC-V SoC, X-SET supports end-to-end complex GPM acceleration without requiring software modifications. Experimental results confirm the effectiveness of X-SET, which achieves up to **142.9**× improvement in compute density (13.7× geometric mean) and delivers up to **42.9**× end-to-end speedup, with an average gain of **6.4**× over state-of-the-art GPM accelerators.

Acknowledgments

This work is supported by National Key Research and Development Program of China (No. 2024YFB4504200) and the Guangzhou-HKUST(GZ) Joint Funding Program (No.2025A03J3568). We also appreciate the AMD Heterogeneous Accelerated Compute Cluster (HACC) Program [3] for providing access to hardware resources.

References

[1] Balázs Adamcsek, Gergely Palla, Illés J. Farkas, Imre Derényi, and Tamás Vicsek. 2006. CFinder: locating cliques and overlapping modules in biological networks. *Bioinformatics* 22, 8 (April 2006), 1021–1023. https://doi.org/10.1093/bioinformatics/btl039

[2] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S. Cenk Sahinalp. 2008. Biomolecular network motif counting and discovery by color coding. *Bioinformatics* 24, 13 (July 2008), i241–i249. https://doi.org/10.1093/bioinformatics/btn163

[3] AMD Xilinx 2025. *Heterogeneous Accelerated Compute Cluster (HACC) at NUS*. https://xacchead.d2.comp.nus.edu.sg/.

[4] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[5] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016), 6–2.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221. https://doi.org/10.1145/2228360.2228584

[7] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization* 14, 2 (June 2017), 1–25. https://doi.org/10.1145/3085572

[8] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. 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)*. Association for Computing Machinery, New York, NY, USA, 282–297. https://doi.org/10.1145/3466752.3480133

[9] Weichen Cao, Ke Meng, Zhiheng Lin, and Guangming Tan. 2025. GLumin: Fast Connectivity Check Based on LUTs For Efficient Graph Pattern Mining. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '25)*. Association for Computing Machinery, New York, NY, USA, 455–468. https://doi.org/10.1145/3710848.3710889

[10] 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, Porto Portugal, 1–12. https://doi.org/10.1145/3190508.3190545

[11] Qihang Chen, Boyu Tian, and Mingyu Gao. 2022. FINGERS: exploiting fine-grained parallelism in graph mining accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 43–55. https://doi.org/10.1145/3503222.3507730

[12] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: a two-level framework for efficient graph pattern mining. In *Proceedings of the ACM International Conference on Supercomputing*. ACM, Virtual Event USA, 378–391. https://doi.org/10.1145/3447818.3460359

[13] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: an efficient and flexible graph mining system on CPU and GPU. *Proceedings of the VLDB Endowment* 13, 8 (April 2020), 1190–1205. https://doi.org/10.14778/3389133.3389137

[14] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind Arvind. 2021. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 581–594. https://doi.org/10.1109/ISCA52012.2021.00052 ISSN: 2575-713X.

[15] Guohao Dai, Zhenhua Zhu, Tianyu Fu, Chiyue Wei, Bangyan Wang, Xiangyu Li, Yuan Xie, Huazhong Yang, and Yu Wang. 2022. DIMMining: pruning-efficient and parallel graph mining on near-memory-computing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 130–145. https://doi.org/10.1145/3470496.3527388

[16] Alexandra Duma and Alexandru Topirceanu. 2014. A network motif based approach for classifying online social networks. In *2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*. IEEE, Timisoara, Romania, 311–315. https://doi.org/10.1109/SACI.2014.6840083

[17] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GraMi: frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment* 7, 7 (March 2014), 517–528. https://doi.org/10.14778/2732286.2732289

[18] Katherine Faust. 2010. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Social Networks* 32, 3 (2010), 221–233. https://doi.org/10.1016/j.socnet.2010.03.004

[19] Dima Feldman and Yuval Shavitt. 2008. Automatic Large Scale Generation of Internet PoP Level Maps. In *IEEE GLOBECOM 2008 - 2008 IEEE Global Telecommunications Conference*. IEEE, New Orleans, LA, USA, 1–6. https://doi.org/10.1109/GLOCOM.2008.ECP.466

[20] Benoit Gaüzère, Luc Brun, and Didier Villemin. 2012. Two new graphs kernels in chemoinformatics. *Pattern Recognition Letters* 33, 15 (Nov. 2012), 2038–2047. https://doi.org/10.1016/j.patrec.2012.03.020

[21] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. 2021. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 769–774.

[22] Mark S. Granovetter. 1973. The Strength of Weak Ties. *Amer. J. Sociology* 78, 6 (1973), 1360–1380. http://www.jstor.org/stable/2776392

[23] Paul W. Holland and Samuel Leinhardt. 1976. Local Structure in Social Networks. *Sociological Methodology* 7 (1976), 1. https://doi.org/10.2307/270703

[24] 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*. ACM, Heraklion Greece, 1–16. https://doi.org/10.1145/3342195.3387548

[25] D. N. Joanes and C. A. Gill. 1998. Comparing measures of sample skewness and kurtosis. *Journal of the Royal Statistical Society: Series D (The Statistician)* 47, 1 (1998), 183–189. https://doi.org/10.1111/1467-9884.00122 arXiv:https://rss.onlinelibrary.wiley.com/doi/pdf/10.1111/1467-9884.00122

[26] 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*. ACM, Lausanne Switzerland, 1217–1231. https://doi.org/10.1145/3373376.3378524

[27] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 29–42. https://doi.org/10.1109/ISCA.2018.00014

[28] Hisashi Kashima, Hiroto Saigo, Masahiro Hattori, and Koji Tsuda. 2011. Graph kernels for chemoinformatics. In *Chemoinformatics and advanced machine learning perspectives: complex computational methods and collaborative techniques*. IGI Global Scientific Publishing, 1–15.

[29] Yutaka I. Leon-Suematsu, Kentaro Inui, Sadao Kurohashi, and Yutaka Kidawara. 2011. Web Spam Detection by Exploring Densely Connected Subgraphs. In *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*. IEEE, Lyon, France, 124–129. https://doi.org/10.1109/WI-IAT.2011.152

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

[31] Zerun Li, Xiaoming Chen, and Yinhe Han. 2024. TMiner: A Vertex-Based Task Scheduling Architecture for Graph Pattern Mining. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1295–1308. https://doi.org/10.1109/MICRO61859.2024.00096 ISSN: 2379-3155.

[32] Zhiheng Lin, Ke Meng, Chaoyang Shui, Kewei Zhang, Junmin Xiao, and Guangming Tan. 2024. Exploiting Fine-Grained Redundancy in Set-Centric Graph Pattern Mining. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 175–187. https://doi.org/10.1145/3627535.3638507

[33] Kathy J. Liszka and Kenneth E. Batcher. 1993. A Generalized Bitonic Sorting Network. In *1993 International Conference on Parallel Processing - ICPP'93*, Vol. 1. 105–108. https://doi.org/10.1109/ICPP.1993.23

[34] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (April 2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[35] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. GraphZero: A High-Performance Subgraph Matching System. *SIGOPS Oper. Syst. Rev.* 55, 1 (June 2021), 21–37. https://doi.org/10.1145/3469379.3469383

[36] 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 (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 509–523. https://doi.org/10.1145/3341301.3359633

[37] Tijana Milenković and Nataša Pržulj. 2008. Uncovering biological network function via graphlet degree signatures. *Cancer informatics* 6 (2008), CIN–S680.

[38] Philippos Papaphilippou, Wayne Luk, and Chris Brooks. 2022. FLiMS: A Fast Lightweight 2-Way Merger for Sorting. *IEEE Trans. Comput.* 71, 12 (Dec. 2022), 3215–3226. https://doi.org/10.1109/TC.2022.3146509 Conference Name: IEEE Transactions on Computers.

[39] Liva Ralaivola, Sanjay J. Swamidass, Hiroto Saigo, and Pierre Baldi. 2005. Graph kernels for chemical informatics. *Neural Networks* 18, 8 (2005), 1093–1110. https://doi.org/10.1016/j.neunet.2005.07.009 Neural Networks and Kernel Methods for Structured Domains.

[40] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. 2022. SparseCore: stream ISA and processor specialization for sparse computation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 186–199. https://doi.org/10.1145/3503222.3507705

[41] Tianhui Shi, Jidong Zhai, Haojie Wang, Qiqian Chen, Mingshu Zhai, Zixu Hao, Haoyu Yang, and Wenguang Chen. 2023. GraphSet: High Performance Graph Mining through Equivalent Set Transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3581784.3613213

[42] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: high performance graph pattern matching through effective redundancy elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 100, 14 pages.

[43] Lukas Steiner, Matthias Jung, Felipe S. Prado, Kirill Bykov, and Norbert Wehn. 2020. DRAMSys4.0: A Fast and Cycle-Accurate SystemC/TLM-Based DRAM Simulator. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings* (Samos, Greece). Springer-Verlag, Berlin, Heidelberg, 110–126. https://doi.org/10.1007/978-3-030-60939-9_8

[44] Nishil Talati, Haojie Ye, Sanketh Vedula, Kuan-Yu Chen, Yuhan Chen, Daniel Liu, Yichao Yuan, David Blaauw, Alex Bronstein, Trevor Mudge, and Ronald Dreslinski. 2022. Mint: An Accelerator For Mining Temporal Motifs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1270–1287. https://doi.org/10.1109/MICRO56248.2022.00089

[45] Nishil Talati, Haojie Ye, Yichen Yang, Leul Belayneh, Kuan-Yu Chen, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2022. NDMiner: accelerating graph pattern mining using near data processing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 146–159. https://doi.org/10.1145/3470496.3527437

[46] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 425–440. https://doi.org/10.1145/2815400.2815410

[47] 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 *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 763–782.

[48] Yujiang Wang, Ying Cao, Zhaobo Zhang, Pingpeng Yuan, and Hai Jin. 2025. SymmPi: Exploiting Symmetry Removal for Fast Subgraph Matching. *Data Science and Engineering* 10, 2 (June 2025), 230–245. https://doi.org/10.1007/s41019-024-00271-w

[49] Yibo Wu, Jianfeng Zhu, Wenrui Wei, Longlong Chen, Liang Wang, Shaojun Wei, and Leibo Liu. 2023. Shogun: A Task Scheduling Framework for Graph Mining Accelerators. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3579371.3589086

[50] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M. Tamer Ozsu, Wei-Shinn Ku, and John C. S. Lui. 2020. G-thinker: A Distributed Framework for Mining Subgraphs in a Big Graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, Dallas, TX, USA, 1369–1380. https://doi.org/10.1109/ICDE48307.2020.00122

[51] 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)*. 895–907. https://doi.org/10.1109/MICRO50266.2020.00077

[52] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, London United Kingdom, 974–983. https://doi.org/10.1145/3219819.3219890

[53] Young-Rae Cho and Aidong Zhang. 2010. Predicting Protein Function by Frequent Functional Association Pattern Mining in Protein Interaction Networks. *IEEE Transactions on Information Technology in Biomedicine* 14, 1 (Jan. 2010), 30–36. https://doi.org/10.1109/TITB.2009.2028234

[54] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng Guo. 2020. Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, Dallas, TX, USA, 673–684. https://doi.org/10.1109/ICDE48307.2020.00064

A Artifact Appendix

A.1 Abstract

This artifact provides a comprehensive package to validate the findings of our work. It includes: (1) The Chisel RTL source for our X-SET hardware design; (2) VLSI scripts for synthesis and for Power, Performance, and Area (PPA) evaluation; (3) A cycle-accurate SystemC simulator for end-to-end performance evaluation; and (4) Python scripts and Jupyter notebooks for data extraction and plotting. These components enable the full reproduction of all tables and figures in the evaluation section.

A.2 Artifact check-list (meta-information)

- **Program:** Chisel RTL, SystemC simulator, Python scripts/Jupyter notebook
- **Compilation:** Chipyard[4], Synopsys Design Compiler, CMake, C++20 Compiler (e.g. Clang 19)
- **Binary:** A pre-built simulator binary included in Docker image
- **Data set:** Real-world graph datasets
- **Run-time environment:** Docker, Linux shell
- **Hardware:** Standard x86-64 machine. Multi-core processor are recommended for parallel execution.
- **Execution:** Shell scripts, GNU Parallel, Jupyter Notebooks
- **Metrics:** Area, power, execution time, speedup
- **Output:** Verilog files, synthesis reports (.rpt), CSV files, plots
- **Experiments:** Reproduction of Table 4 and Figures 12 to 19.
- **How much disk space required (approximately)?:** 20-30 GB
- **How much time is needed to prepare the workflow (approximately)?:** 20-30 minutes
- **How much time is needed to complete experiments (approximately)?:** Table 5 lists the estimated execution times for the cycle-accurate simulator on an AMD EPYC 9684X CPU. The total required time for all experiments exceeds 1500 CPU-core-hours.
- **Publicly available?:** Yes
- **Workflow automation framework used?:** Docker, CMake, shell scripts, GNU Parallel
- **Archived (provide DOI)?:** `10.5281/zenodo.17176841`

A.3 Description

A.3.1 How to access. The complete artifact, including all source code and datasets, is archived on Zenodo. For the latest updates, please refer to our AE GitHub repository[2]. The required graph data can be downloaded from Google Drive[3] and must be extracted into the `graphs/` directory at the root of the repository.

A.3.2 Hardware and Software dependencies. All experiments can be performed on a standard x86-64 machine equipped with Docker. We highly recommend using a server with a high CPU core count to accelerate the simulation of numerous benchmark cases in parallel. We also recommend using datasets other than LiveJournal (`LJ`) and Patents (`PA`) for early-stage verification. The area and power analysis portion of the evaluation requires Synopsys Design Compiler

[2] https://github.com/CLab-HKUST-GZ/micro58-xset

[3] https://drive.google.com/file/d/1d0muGAdL1zoal0cnMY4jf1HjQAnTM8J6/view

Table 5: Estimated time needed for simulator execution

	PP	AS	MI	YT	PA	LJ
3CF	1.23s	8.35s	50.74s	153.70s	31.89m	2.04h
3MC	2.63s	24.52s	140.70s	25.25m	57.61m	4.96h
4CF	1.33s	51.18s	17.73m	254.80s	34.96m	7.96h
5CF	1.33s	335.61s	7.29h	7.27m	38.25m	9.14d
CYC	5.07s	69.12s	11.17m	1.87h	1.81h	1.56d
DIA	1.49s	180.51s	1.31h	41.40m	1.17h	1.41d
TT	5.12s	8.38m	2.69h	5.61h	2.17h	4.57d

(V-2023.12) and TSMC 28nm Process Design Kit (PDK) for identical results.

If you do not have enough CPU power or time for full simulator execution or do not have access to Synopsys' commercial synthesis software, we have provided all results, including all simulator execution log, scripts for result gathering and plotting (`simulator-results`), generated Verilog code (`rtl-out`) and synthesis reports (`vlsi-results`), packed in `xset-results.zip` of the Zenodo archive.

A.3.3 Data sets. The experiments utilize several real-world graphs from established sources [24, 30]. All graphs are pre-processed and included in the artifact download.

A.4 Installation

The primary setup step involves building the Docker images from the `rtl/` and `simulator/` directories:

```
cd rtl
docker build -t micro58-xset-ae-rtl:v1.0 .
cd ../simulator
docker build -t micro58-xset-ae-sim:v1.0 .
cd ..
```

To bypass potential network issues during the build process, pre-built images are also available on DockerHub:

```
docker pull xsun2001/micro58-xset-ae-rtl:v1.0
docker pull xsun2001/micro58-xset-ae-sim:v1.0
```

A.5 Experiment workflow

The evaluation is divided into two parts: Area/Power Analysis and End-to-End Performance Evaluation. Detailed, step-by-step instructions are available in the main `README.md` file.

A.5.1 Area and Power Analysis (Table 4 and Figure 15).

(1) **Select Configuration:** Choose the target hardware configuration. For Table 4, use `XsetDefault`. For Figure 15, use `XsetSX` for the Systolic Merge Array and `XsetBX` for our Order-Aware SIU, where `X` is the segment width.
(2) **Generate RTL:** Launch the RTL container to generate Verilog design from Chipyard and Chisel source code. Then copy the Verilog codes to host directory.

```
docker run -it --name rtl micro58-xset-ae-rtl:v1.0 bash
> cd /opt/chipyard/sims/verilator
> make verilog CONFIG=<Config>

docker cp rtl:/opt/chipyard/sims/verilator/generated-src/\
    chipyard.harness.TestHarness.<Config>/gen-collateral \
    ./vlsi/
```

(3) **Run Synthesis:** In the `vlsi/` directory, edit `env.sh` to set the `CHIPYARD_TARGET` and `TOP_MODULE`. Then, execute the synthesis flow by sourcing the environment script (`.env.sh`) and running the main script (`run_synth.sh`). Results (`area.rpt`, `power.rpt`, `timing.rpt`, synthesized netlist) are generated to `vlsi/reports/<Config>`.

A.5.2 End-to-End GPM Performance (Figures 12 to 14 and 16 to 19).

(1) **Run Simulator:** Launch the simulation container (`micro58-xset-ae-sim`), mounting the graph datasets to `/opt/graphs/` via `-v./graphs/:/opt/graphs/`.
(2) **Execute Benchmarks:** Navigate to `/opt/xset/benchmarks`. Each figure in the paper has a corresponding execution script (e.g., `./fig12.sh`). These scripts leverage GNU Parallel to run all necessary simulations. Use `MAX_PROCS` environment variable to limit the maximum count of simulators running in parallel.
(3) **Plot Results:** Once simulations are complete, use the provided Jupyter notebooks in `/opt/xset/scripts` (`extractor.ipynb`, `draw.ipynb`, etc.) to parse the simulation logs, generate CSV files, and render the final plots.

A.6 Evaluation and expected results

The artifact is designed to faithfully reproduce the data presented in the paper.

- **Area/Power:** The synthesis reports, including generated `area.rpt` and `power.rpt`, should contain values that closely match those in Table 4 and Figure 15.
- **Performance:** The plots generated by the Jupyter notebooks should align with the trends and conclusions presented in Figures 12 to 14 and 16 to 19. Minor performance variations due to differences in host machine architecture are expected but should not alter the primary findings.

A.7 Experiment customization

The artifact is highly configurable, allowing for exploration beyond the specific experiments in this paper.

A.7.1 Hardware Customization. The X-SET hardware design uses the standard Chipyard parameterization framework. A comprehensive list of configurable parameters is located in `rtl/src/main/scala/xset/params.scala`. These can be modified by defining a new configuration class.

A.7.2 Simulator Customization. The simulator's behavior is controlled by a `.toml` configuration file. The baseline configuration is `fig12-overall.toml` in `simulator/benchmarks/` directory. New experiments can be created by modifying this file. The simulator is invoked via:

```
xset_systemc_simulator <dataset_idx> <pattern_name>
    [--log=<log_dir>] [--cfg=<config_file>]
```

Here, `dataset_idx` and `pattern_name` select the corresponding dataset and schedule listed in the given configuration file.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae