PSMiner: A Pattern-Aware Accelerator for High-Performance Streaming Graph Pattern Mining

Hao Qi*, Yu Zhang*†, Ligang He‡, Kang Luo*, Jun Huang*, Haoyu Lu*, Jin Zhao*†, Hai Jin*
*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab,
Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China
†Zhejiang-HUST Joint Research Center for Graph Processing, Zhejiang Lab, China
‡Department of Computer Science, University of Warwick, United Kingdom
{theqihao, zhyu, luokang2000, jun_huang, hyl, zjin, hjin}@hust.edu.cn ligang.he@warwick.ac.uk

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247902

***Abstract*—Streaming *Graph Pattern Mining* (GPM) has been widely used in many application fields. However, the existing streaming GPM solution suffers from many unnecessary explorations and isomorphism tests, while the existing static GPM ones require many repetitive operations to compute the full graph. In this paper, we propose a pattern-aware incremental execution approach and design the first streaming GPM accelerator called PSMiner, which integrates multiple optimizations to reduce redundant computation and improve computing efficiency. We have conducted extensive experiments. The results show that compared with the state-of-the-art software and hardware solutions, PSMiner achieves the average speedups of 770.9× and 60.4×, respectively.**

I. Introduction

Graph Pattern Mining (GPM) locates all subgraphs that are isomorphic to the specific patterns in a given graph. It is widely used in many fields, such as social science [9], bioinformatics [13], and cheminformatics [10]. GPM suffers from high algorithmic complexity and large-volume of memory accesses. Therefore, many systems [7], [11], [14] have been proposed to process GPM efficiently.

Early GPM systems [7], [16] adopt the pattern-oblivious approach. The approach starts from a subgraph (one vertex initially), keeps expanding it, and then checks whether the resulting subgraph is isomorphic to the pattern. However, the approach suffers from unnecessary explorations and isomorphism tests. The pattern-aware approach generates the matching order to avoid the isomorphism tests and also generates the symmetry order to prune the exploration space. Therefore, the pattern-aware approach typically achieves better performance and is adopted by most recent GPM software systems [11], [14] and hardware accelerators [4], [5].

In the real world, the graphs are often streaming graphs. When a small number of vertices or edges are updated, it is expensive and unnecessary to recompute all matching subgraphs in the entire graph. Therefore, Tesseract [3] proposes an incremental execution approach, in which it adopts the update-based exploration model. However, it uses the pattern-oblivious approach, which suffers from unnecessary explorations and expensive isomorphism tests. To address this problem, we design a pattern-aware incremental execution approach to improve the efficiency of streaming GPM.

However, the proposed approach contains many set computations (such as intersection of two sets), which accounts for more than 89% of the execution time according to our benchmarking. We analyzed the runtime characteristics of the set operations in streaming GPM and made two important observations: i) there are a large amount of redundant set computations, and ii) the lengths of the two sets in pattern-aware streaming GPM are very different. The reason is because the graph updates are usually associated with a small number of vertices, and the set operations are linked to these vertices. Based on the two observations, we further design the redundancy detection algorithm and a hybrid set computation mechanism.

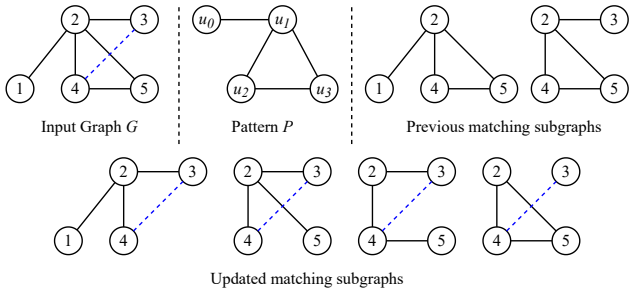

Fig. 1. An example to illustrate streaming GPM. The blue dashed line represents added edge.

Since streaming GPM is usually conducted concurrently by multiple processing elements (e.g., multithreading), detecting the redundant set computation at runtime may cause high lock overhead. Moreover, it is difficult to process the set operations efficiently on general-purpose processors (e.g. CPU) [5]. In addition, the existing GPM accelerators [4]–[6] are designed for static GPM, and are inefficient for streaming GPM because they search the entire graph for the given pattern. Therefore, we propose PSMiner, a pattern-aware accelerator for streaming GPM, to efficiently reduce redundant set operations and compute the sets with very different sizes (called hybrid sets).

The contributions of this paper are summarized as follows. i) We propose a pattern-aware incremental execution approach for streaming GPM. By analyzing the runtime characteristics of the approach, we propose two optimization techniques to reduce redundant set computations and improve the computation efficiency on the hybrid sets. ii) We design a pattern-aware accelerator, which implements a redundancy detector and the hybrid set computation mechanism. In the set computation mechanism, we design a novel data-parallelism and pipeline-based method for accelerating the binary search desired in the hybrid set computations. iii) We implement PSMiner and compare it with the state-of-the-art CPU- and ASIC-based solutions. The results show that PSMiner can achieve significant improvement.

II. Background and Motivation

A. Streaming Graph Pattern Mining

The streaming graph updates, which include the addition and deletion of edges (an edge is a triple: two vertices and a timestamp indicating the time of the edge update), arrive constantly and are buffered in batches [12]. These buffered updates will be applied to previous graph snapshots to generate the latest graph snapshot.

This paper is supported by National Key Research and Development Program of China under grant No. 2022YFB2404202, NSFC (No. 62072193), Huawei Technologies Co., Ltd (No. YBN2021035018A5), and the Young Top-notch Talent Cultivation Program of Hubei Province. Yu Zhang (zhyu@hust.edu.cn) is the corresponding author of this paper.

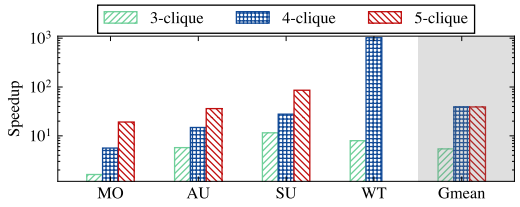

Fig. 2. Speedup of GraphPi over Tesseract

After the graph is updated, the matching subgraphs (also called embeddings) also need to be updated. For example, consider the example of mining a tailed triangle shown at the top of Fig. 1 (i.e., "Pattern P"). There are two matching subgraphs in the "Input Graph G", which are the subgraph consisting of the vertices $\{1, 2, 4, 5\}$ and the one of $\{3, 2, 4, 5\}$. The bottom row of Fig. 1 shows the four new matching subgraphs (embeddings) after adding a new edge between vertices 3 and 4. In order to obtain the updated matching subgraphs, Tesseract [3], which is a streaming GPM system, has been developed recently to process the latest snapshot.

B. Static Graph Pattern Mining Execution Model

Existing GPM systems can be classified into two categories in general: *pattern-oblivious* and *pattern-aware*. The *pattern-oblivious* approach suffers from high overhead due to unnecessary explorations and isomorphism tests. The pattern-aware solution avoids the expensive isomorphism tests by specifying the matching order, and eliminates the repetitive enumerations of identical subgraphs by specifying the symmetry order [11].

Matching order. In GPM, the matching order is a total order of vertices in the pattern. For example, in Fig. 1, $\{u_0,\ u_1,\ u_2,\ u_3\}$ is a matching order of tailed triangle, and indicates that u_i is matched (processed) before u_j only when $i < j$. Since the subgraphs finally obtained by following the matching order will always match the pattern, isomorphism tests are not necessary.

Symmetry order. In Fig. 1, u_2 and u_3 can be mapped to the subgraphs $\{4, 5\}$ and $\{5, 4\}$, which are actually identical subgraphs. To avoid such redundant explorations, a partial order (also called symmetry order) can be defined for symmetry breaking [11]. For example, the subgraph $\{4, 5\}$ can be pruned by the restriction of $u_2 > u_3$ in the symmetry order.

C. Limitations of Existing Solutions

In streaming GPM, a newly added edge, especially added to the high-degree vertices, can create a large number of matching embeddings. It is a very time-consuming task to simply enumerate them. Also, it is very costly and unnecessary to recompute all matching embeddings in the entire graph.

With the incremental computation of streaming GPM, Tesseract [3] enumerates all changes using an update-based exploration approach, which starts from the graph updates (e.g., a newly added edge) to explore the subgraphs. However, its approach is pattern-oblivious, which may lead to significant overhead from unnecessary explorations and isomorphism tests. To demonstrate the problem, we conducted the experiments to compare Tesseract with GraphPi [14], which is the state-of-the-art static GPM system and uses the pattern-aware approach. The details of the platform and datasets used in the comparison experiments are presented in Section V. GraphPi performs GPM in the entire graph, while Tesseract performs incremental computation with 10 K updates. As illustrated in Fig. 2 (note that we do not show the 5-clique algorithm on WT with Tesseract because its run-time is too long), GraphPi outperforms Tesseract by up to 1020.1x when running the clique algorithms. This is because GraphPi adopts the pattern-aware approach.

Algorithm 1: Pattern-Aware Incremental Execution Approach for Streaming GPM

Input: P: pattern; UE: updated edges; G: graph data
Output: UME: updated matching embeddings

```
1 SO = GenSymmetryOrder(P);
2 PMOS = GenPrunedMultiMatchingOrders(P, SO);
3 foreach edge ue ∈ UE do
4     foreach matching order mo ∈ PMOS do
5         UME ∪= PatternMatching(ue, mo, SO, G);
```

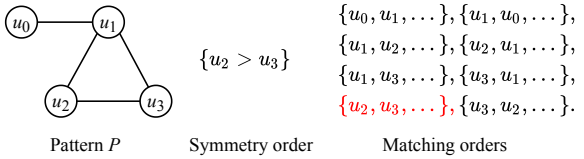

Fig. 3. The set of matching orders generated from the tailed triangle pattern. The matching order in red is redundant based on the symmetry order.

The existing static GPM software systems [11], [14] and hardware accelerators [4]–[6] compute the entire graph. In streaming graphs, recomputing all matching embeddings over the whole graph for each update batch is expensive and unnecessary. It is expensive because streaming graph updates arrive frequently and the graphs in the real-world are usually large. It is unnecessary mainly because only a small portion of the graph is affected by the graph updates [8].

III. Pattern-Aware Incremental Execution Approach

The limitations of the existing solutions drive us to design a pattern-aware incremental execution approach for streaming GPM. We aim to reduce repetitive computations by the incremental approach, while pruning unnecessary subgraph explorations and avoiding isomorphism tests by the pattern-aware approach.

A. The Approach

We observed that *each of the updated matching embeddings in streaming GPM must contain at least one updated edge, and each updated edge can be matched to any edge of the pattern.* For example, in Fig. 1, the updated matching embeddings all contain the updated edge (3, 4). The updated edge in the input graph can be matched to any edge in the pattern P. The incremental approach aims to construct an embedding (a subgraph) isomorphic to pattern P by starting from the updated edge to explore other neighboring edges in the input graph. Note that although GraphPi only generates one matching order, the first vertex in the matching order has to be mapped to every vertex in the entire graph.

The main steps of our pattern-aware incremental execution approach is outlined in Algorithm 1. In the algorithm, a set of matching orders are generated and then pruned based on the symmetry order (lines 1-2). Next, the algorithm starts from the updated edges to explore the matching embeddings by processing these matching orders (lines 3-5).

The matching orders of a given pattern are generated in the following way. Since a newly updated edge may match any edge in a given pattern, the first two elements of a matching order are two vertices of an edge in the pattern. Since the order of the vertices matters, two matching orders will be generated for an edge. For example, two matching orders are generated by the edge connecting u_0 and u_1 in Fig. 3, one starting with $\{u_0, u_1, ...\}$ and the other with $\{u_1, u_0, ...\}$. If the number of edges in the pattern is ep, $2 \times ep$ matching orders will be generated.

Further, when two vertices u_i and u_j in a pattern are symmetric, the matching orders $\{u_i, u_j, ...\}$ and $\{u_j, u_i, ...\}$ will induce the identical

Algorithm 2: The Streaming Pattern Matching Algorithm for the Tailed Triangle Pattern

Input: ue: updated edge (s,d,t); mo: $\{u_0, u_1, u_2, u_3\}$; SO: $u_2 > u_3$; $N(v)$: neighbors of vertex v
Output: UME: updated matching embeddings

1 $v_0 = ue.s,\ \ v_1 = ue.d$;
2 **foreach** vertex $v_i \in N(v_1)$ **do**
3 *// $e_{v_1 v_i}$ denotes the edge from v_1 to v_i;*
4 **if** $e_{v_1 v_i}.t > ue.t$ **then**
5 continue with next iteration;
6 **foreach** vertex $v_j \in (N(v_1) \cap N(v_i))$ **do**
7 $t_j = max(e_{v_1 v_j}.t,\ e_{v_i v_j}.t)$;
8 **if** $(t_j > ue.t)$ *or* $((v_i < v_j)$ *from* $SO)$ **then**
9 continue with next iteration;
10 $UME \cup = \{v_0, v_1, v_i, v_j\}$;

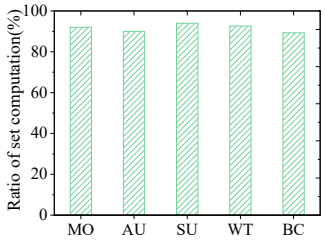

Fig. 4. The time proportion of set computations

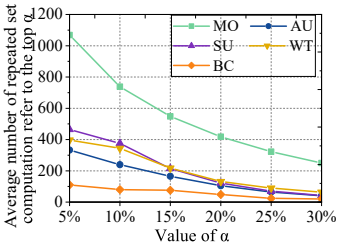

Fig. 5. The number of times that the top α of set computations are repeated

subgraphs. The matching order $\{u_2, u_3, ...\}$ in the tailed triangle can be removed when the new edge is (3, 4) and the symmetry order contains $u_2 > u_3$. After *redundant matching orders are pruned*, the pattern-aware incremental algorithm starts from the updated edges to run these matching orders (lines 4-5 in Algorithm 1).

Algorithm 2 outlines the steps of finding the matching embeddings for the tailed triangle pattern by taking as input the matching order $\{u_0, u_1, u_2, u_3\}$ and the symmetry order $\{u_2 > u_3\}$ in Fig. 3. Algorithm 2 corresponds to the PatternMatching procedure (Line 5) in Algorithm 1,

The elements u_0 and u_1 in the matching order are mapped to the two vertices (v_0 and v_1) of the updated edge (line 1) (note that u denotes the vertices in the pattern while v the vertices in the input graph). Then, the neighbors (denoted by v_i) of v_1 are traversed to match u_2 (line 2). u_3 is matched by the intersection of v_1's neighbors and v_i's neighbors (line 6) because u_3 is the common neighbor of u_1 and u_2 in the pattern. We also apply the timestamp constraint (lines 4 and 8) similar to Tesseract [3], and the symmetry order (line 8) to reduce redundant explorations.

B. Challenges in the Approach

We implemented our pattern-aware incremental approach based on GraphPi [14], which we call GraphPi-S. As discussed in introduction, there are a large number of set computations (set intersection and set difference) in streaming GPM, as shown in Fig. 4. In the benchmarking experiments, we made two important observations about the runtime characteristics of the set computations in streaming GPM, which represent the challenges we face to further improve the performance of our pattern-aware incremental approach.

Numerous Redundant Set Computations. Fig. 5 shows the percentage of set computations that are repeated in GPM. It can be seen that 30% of the set computations are repeated from tens of times to thousands of times. The high repetition of set computations in our approach can be explained as follows. First, our incremental approach

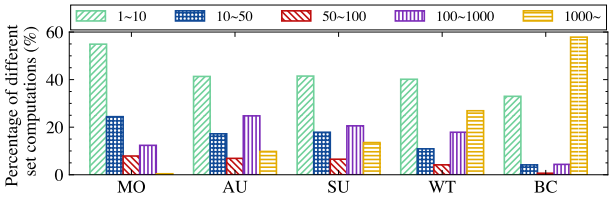

Fig. 6. The length ratio of the longer set to the shorter set

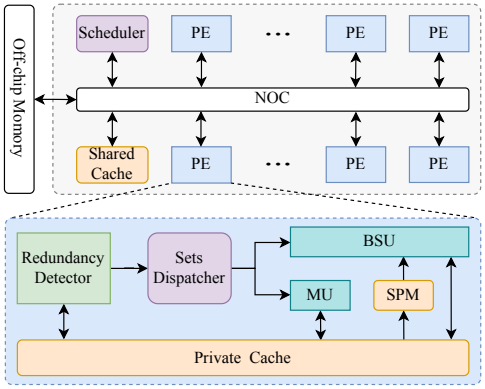

Fig. 7. PSMiner architecture overview

generates multiple matching orders, which all start from the updated edges to explore the matching subgraphs. So the set computations are performed around the same set of edges. Second, streaming graph updates usually refer to a small fraction of high-degree vertices in the input graph [17], which causes most pattern matching operations to be performed around these same vertices.

In this work, we develop an optimization technique to reduce repeated set computations. In this technique, we count at runtime the number of repeated times for the same set computation. The result of a set computation will be cached when the repeated times reach a threshold. When the same set computation is invoked in future, the result is directly fetched from the cache. However, it may cause high lock overhead to count the repeated times of a set computation at runtime on the general-purpose cores (such as CPU cores in a multicore computer). This is because multiple threads need to access the same memory location where the repeated times of a set computation is cached. Our experimental results show that the lock overhead takes up 76.6% of the execution time, which greatly diminishes the benefits brought by our approach. In addition to the lock overhead, there are also the overheads of repeat counting itself and indexing the saved results. These significant overheads eventually motivate us to develop a specialized hardware unit to efficiently detect and reduce redundant set computations.

Various Set Computations. The set operands in the set computations in streaming GPM can have very different lengths (i.e., the number of elements in the sets). As shown in Fig. 6, most of the set length ratios (i.e., the ratio of the longer length to the shorter length) exceed 50, and even more than 1000 in some cases. Given two sets L and S in a set computation, when the lengths of the two sets are very different (e.g., $|L| >> |S|$), the merge-based set computation algorithm [5], which iterates through the two sets and detects the common elements, is much less efficient than the binary search-based algorithm [1], which iterates over the elements in the shorter set and uses the binary search to determine if the element is in the longer set. This is because time complexity of the binary search-based algorithm is $O(log(|L|) * |S|)$ while that of the merge-based algorithm is $O(|L| + |S|)$. When $|L| >> |S|$, $log(|L|) * |S|$ is much smaller than $|L| + |S|$. However, the data dependencies and random

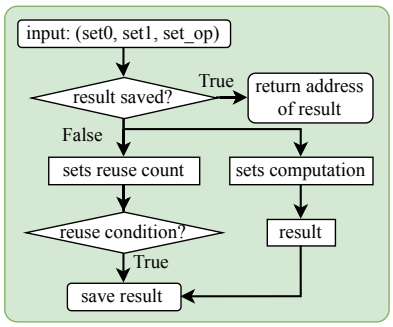

Fig. 8. The execution flow of redundancy detector

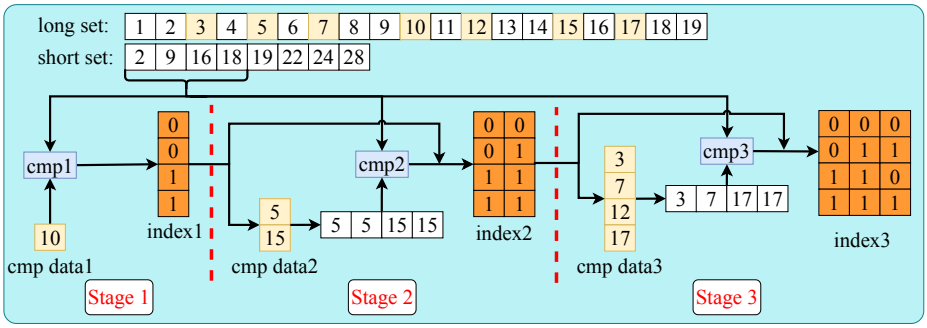

Fig. 9. Illustration of the pipeline mechanism of the binary search

memory accesses in the binary search algorithm lead to inefficient executions in the general-purpose hardware. In addition, most static GPM accelerators [4], [5] only use and optimize the merge-based algorithm. These challenges also motivate us to optimize the design of set computations in the specialized hardware accelerator.

IV. The Architecture of PSMiner

The challenges of software-based implementation of our approach motivate us to design our hardware accelerator PSMiner, which can efficiently detect redundant set computations in streaming GPM, and implement a hybrid set computation processing mechanism, which allocates the set computations to the merge-based or the binary search-based units according to their estimated runtimes.

A. Overview

Fig. 7 illustrates the PSMiner architecture. PSMiner contains a shared cache, a scheduler, and multiple *Processing Elements* (PE). The scheduler dynamically assigns the updated edges in a graph snapshot to the PEs. Each PE starts from an edge to explore the matching subgraphs. This architecture is similar to FINGERS [4]. Therefore, we exploit its programming interface, execution flow, and system integration solutions. The key innovation in our PSMiner resides in the internal design of a PE. Each PE contains the following components. The redundancy detector detects whether the computation result on a pair of sets has been saved and if not, whether the result should be saved. The sets dispatcher is responsible for assigning a set computation task to the *binary search unit* (BSU) or the *merge unit* (MU), which performs the set computation by the binary search-based algorithm or the merge-based algorithm, respectively. A hash map resides inside the *scratchpad memory* (SPM) to hold the frequently read data for BSU.

B. Reducing Redundant Set Computations

The redundancy detector is designed to detect and reduce redundant set computations. The overall execution flow of the redundancy detector is illustrated in Fig. 8. The execution starts from a set computation, which includes two input sets (set0 and set1) and a set operation. If the result of the set computation has been saved, the redundancy detector will return the address of the result. Otherwise, the set computation will be sent to the sets dispatcher and record the number of times it has been performed. If the result has been saved, its address is saved in a hash table, whose $<key,value>$ is $<(id(set0), id(set1), set\_op), result\_address>$.

However, it is expensive and unnecessary to record the number of times that every set computation is performed. A large number of temporary sets will be generated when performing streaming GPM. For example, to match the last vertex in the 4-clique algorithm, two set computations must be performed. The result of the first set computation is a temporary set. Comparatively, the original set computation, which consists of the vertices' neighbours, has a higher probability to be reused. Therefore, we only count the original set computations. According to our experimental records, the set computations over temporary sets account for around 50% of all set computations. Further, we do not save the results of all original set computations. Rather, we only save the results of the top 30% of most repeated set computations in a batch of updates by default. The value of 30% is determined by analyzing the reuse of set computations in the update batches. Our experimental results show that the reuse count usually tails off when it is beyond top 30%.

To avoid the lock overhead, we assign to the same PE the updated edges linked to the same vertex, and allocate a private hash table for each PE. Specifically, we create a task queue for each PE, and put all updated edges of a vertex in the queue. Then we iterate over all the edges in the queue to check their neighbors. If an updated edge has not been assigned, we add it into the queue. The process repeats until all updated edges have been assigned.

C. Hybrid Set Computation Units

To efficiently support diverse set computations, we design two set computation units: MU and BSU. MU executes the merge-based algorithm while BSU runs the binary search-based algorithm. The design of MU unit is similar to the state-of-the-art static GPM accelerator FINGERS. However, the existing accelerators have not yet considered the optimization of binary search-based algorithm. In PSMiner, we design a data-parallelism pipeline mechanism to improve the performance of BSU.

The pipeline mechanism is illustrated in Fig. 9. In this example, we assume that four elements in the short set are processed in a pipeline stage. The first four elements to be processed are 2, 9, 16, and 18. In each pipeline stage, we use the comparison unit (such as cmp1 in Stage 1) to compare the elements in the short set against the comparison data, which is taken from the long set based on the binary search rule. For example, the comparison data (i.e., cmp data1) of Stage 1 is 10, which is the middle element of the long set, and the cmp data2 of Stage 2 are 5 and 15, which are the middle elements between 1 and 10, and between 10 and 19, respectively. The output of the comparison unit (e.g., the four orange numbers 0011 in Stage 1) in one pipeline stage is combined with the output of the subsequent stages, which is used as the index to locate the address of the comparison data. For example, index1 0011 locates the address of the cmp data2. 0 is obtained by Stage 1 because the elements 2 and 9 in the short set are less than the cmp data1 (10), which also suggests that the comparison data (element) in next stage should be at the left of cmp data1, which locates the comparison data 5. The comparison unit in Stage 2 (i.e., cmp2) generates 0111, which connects to the output in the previous stage. The connected indices will together locate the comparison data (cmp data3) in Stage 3.

The output of the entire pipeline, which is index3 in this figure, is used by the conventional binary search algorithm in the search unit

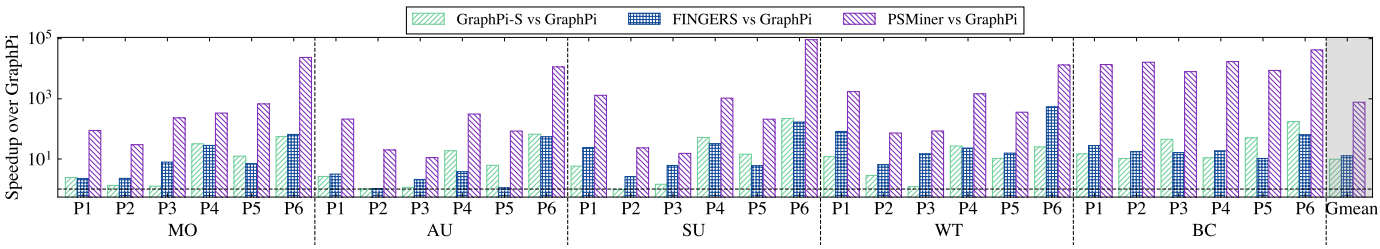

Fig. 10. Speedups of GraphPi-S, FINGERS, and PSMiner over GraphPi

TABLE I
REAL-WORLD STREAMING GRAPH DATASETS (*SEdges* IS THE EDGE NUMBER OF STATIC GRAPH)

Datasets	*#Vertices*	*#Edges*	*#SEdges*
mathoverflow (MO) [1]	24,818	506,550	239,978
askubuntu (AU) [1]	159,316	964,437	596,933
superuser (SU) [1]	194,085	1,443,339	924,886
wiki-talk-temporal (WT) [1]	1,140,149	7,833,140	3,309,592
soc-bitcoin (BC) [2]	24,575,382	122,948,162	60,489,096

[1] https://snap.stanford.edu/data/ [2] https://networkrepository.com/soc-bitcoin.php

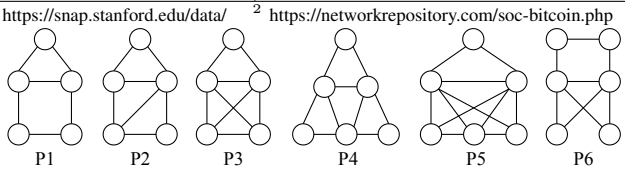

Fig. 11. Patterns for evaluation

as the index to locate the binary search range in the longer set. For example, the top row 000 in index3 means element 2 is less than 10, 5, and 3. Then the search unit will perform the conventional binary search algorithm, and continue to search for 2 between 1 and 3 in the long set. With the pipeline mechanism, the elements in the short set can be processed simultaneously. For example, after the processing of elements $\{2, 9, 16, 18\}$ move to Stage 2, the processing of elements $\{19, 22, 24, 28\}$ in the short set can be started in Stage 1.

The sets dispatcher decides which unit, MU or BSU, is better to run a given set operation, and dispatches the operation to the corresponding unit. The decision is based on the estimated runtime of the set operation on MU and BSU. We build the performance models based on the time complexity and the hardware performance to estimate the runtime of the set computation. Given two sets L and S ($|L| > |S|$), the runtime of the merge-based algorithm to compute the two sets is modelled by $(|L| + |S|)/(MP * F)$ (MP is the number of parallel processing units in MU, F is the clock frequency of PSMiner). Different from the merge-based algorithm, the binary search-based algorithm manifests random memory access. So its runtime can be modelled by multiplying the time complexity by the latency of memory access (i.e., the startup cost of memory access), i.e., $(LM*(|S|*(log(|L|)-PD))/BSP$ (LM is the latency of memory access, PD is the depth of the pipeline, and BSP is the number of search units). The sets dispatcher sends a set operation to the unit (MU or BSU) which offers shorter runtime. According to our previous discussions in Section III.B, binary search-based algorithm is much faster when the lengths of the two sets have big difference, which is also verified by our experimental results.

V. EVALUATION

Datasets and Benchmarks. In our experiments, we use the five streaming graph datasets listed in Table I. Each dataset consists of a collection of graph edges. An edge is represented by a triple (v_i, v_j, t), in which v_i and v_j are the two vertices of the edge and t is the edge's timestamp, i.e., the time when the edge update arrives. The batch size is set to 10 K by default. Six patterns, shown in Fig. 11, are used in the evaluation, and are from GraphPi [14] and Peregrine [11].

Baseline. We compared PSMiner with GraphPi [14] and FINGERS [4]. GraphPi is the state-of-the-art software-based GPM system while FINGERS is a hardware static GPM accelerator. They are static because they do not consider the streaming graphs, and mine the matching embeddings by searching the entire graph. We also implemented our pattern-aware incremental approach into GraphPi, which is called GraphPi-S. GraphPi and GraphPi-S are run on a 28-core 2-way hyper-threaded Intel Xeon E5-2680 v4 CPU with 256 GB DRAM. Note we do not present the comparison results against Tesseract [3]. This is because GraphPi outperforms Tesseract significantly as shown in Fig. 2.

PSMiner Simulation and Configurations. We developed a cycle-accurate simulator to evaluate the performance of PSMiner. We use a DDR4-2666 DRAM of 64 GB with four channels, and a 4 MB shared cache. By default, each PE in PSMiner contains a redundancy detector, a sets dispatcher, 24 MU, a BSU (which contains a pipeline with a depth of 5 and 32 search units), a 32 KB private cache, and a 4 KB SPM.

We implemented the components of PE in Verilog and synthesized the Verilog code using the Synopsis Design Compiler in 28 nm, where PSMiner runs at the 1 GHz clock frequency, to estimate the chip area and the power. We used CACTI [2] to model the SRAMs in PE. The area of each PE is 1.15 mm^2 and its power is 217.1 mW. Each PE (its area is 0.93 mm^2 in 28 nm) of FINGERS also runs at 1 GHz, and contains 24 IU and a 32 KB private cache. To keep the iso-area with FINGEERS, we compared a 16-PE PSMiner chip to a 20-PE FINGERS chip in the evaluation.

A. Overall Results

Fig. 10 shows that GrpahPi-S, FINGERS, and PSMiner achieve the average speedups of 9.8×, 12.7×, and 770.9×, respectively, over GraphPi across six different patterns. Our software-based approach (GraphPi-S) already outperforms GraphPi (the maximum speedup is 222.7×). This is because our incremental execution approach avoids the full-graph computation, which brings GraphPi-S the significant advantage in processing large-scale graphs. This is also the reason why GraphPi-S achieves the speedup of 29.6× over GraphPi on BC while the speedup on MO is 6.7× (BC is much larger than MO).

PSMiner achieves the speedup of 9.8× to 1583.7× over GraphPi-S with the average speed up of 78.6×. Such a great performance improvement is attributed to the fact that PSMiner i) detects and reduces a large number of redundant set computations, and ii) designs the specialized hardware to accelerate the set computations. PSMiner speeds up the processing over FINGERS by 60.4× on average. FINGERS computes the entire graph, resulting in lots of unnecessary computations. In addition, FINGERS performs the set computations using the merge-based algorithm, while PSMiner adopts the hybrid set computations, in which the set computation involving the sets with very different lengths will be assigned to the BSU unit.

B. Contributions of Optimization Techniques

PSMiner includes two main optimization techniques: reducing redundant set computations and hybrid set computations. To further

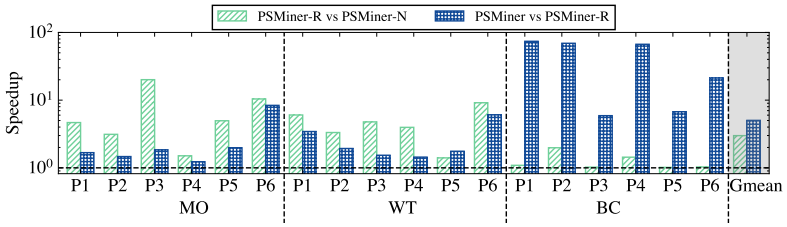

Fig. 12. Speedups from redundancy reduction and hybrid set computation

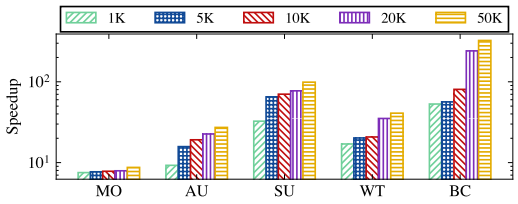

Fig. 13. Impact of batch size (PSMiner vs PSMiner-N)

investigate the performance gains contributed by each of them, we implemented PSMiner-N (PSMiner without any optimization) and PSMiner-R (PSMiner with only redundancy reduction). Fig. 12 shows the speedups of PSMiner-R vs PSMiner-N and PSMiner vs PSMiner-R. As shown in Fig. 12, PSMiner-R achieves up to 20.1× speedup over PSMiner-N, with an average speedup of 3.0×. The performance improvement of PSMiner-R mainly comes from detection and reduction of redundant set computations. Note that the performance improvement of PSMiner-R over PSMiner-N is not as significant on BC. This is mainly due to the fact that the updated edges of the large dataset are sparse compared to the smaller dataset. As a result, there are not so many redundant set computations.

Fig. 12 also shows that PSMiner achieves the 5.0× speedup over PSMiner-R on average. The performance improvement mainly originates from the optimization that accelerates the binary search-based algorithm through the pipeline-based BSU. We observe that the significant speedups are constantly achieved on BC across different patterns. This is because there are a large number of occasions where the sets' lengths differ significantly when the streaming GPM is performed on BC. For example, when mining pattern P1 on BC, 57.9% of set computations involve the input sets whose length ratio is more than 1000, which has been shown in Fig. 6.

C. Sensitivity Studies

To observe the impact of different batch sizes on the performance of our optimization techniques, we process P1 using different batch sizes (i.e., 1 K, 5 K, 10 K, 20 K, and 50 K) in all datasets. As shown in Fig. 13, as the batch size increases, the performance improvement becomes more prominent gradually, especially on larger datasets such as WT and BC. This is mainly because 1) the number of updated edges of the same vertex increases when the batch size is bigger, which results in more redundant set computations, and 2) larger graphs tend to have more high-degree vertices, which are more likely to be processed by the pipelined BSU.

VI. Related Work

Software GPM Systems. Arabesque [16] and Fractal [7] use the pattern-oblivious method to mine subgraphs, resulting in unnecessary explorations and isomorphism tests. To solve the problems, pattern-aware systems [11], [14] are developed. However, these static GPM systems compute the entire graph for each update batch. Tesseract [3] proposes an incremental approach for streaming GPM. However, it adopts the pattern-oblivious method and shows lower performance than the state-of-the-art pattern-aware systems, e.g., GraphPi [14].

Related Hardware Accelerators. Existing GPM accelerators all focus on static graphs. FlexMiner [5] and FINGERS [4] speed up the set computations based on the merge-based algorithm. DIMMining [6] and NDMiner [15] use the processing-in-memory architecture to reduce the costly off-chip data transfer. However, these hardware accelerators can not efficiently process streaming GPM because they compute the full graph for each batch of graph updates.

VII. Conclusion

In this paper, we proposed a pattern-aware incremental execution approach for streaming GPM. We identified the limitations of running streaming GPM on CPUs. In order to address these issues, we proposed a technique to dynamically detect redundant set computations, and designed the hybrid set computation mechanism. Based on the pattern-aware incremental execution approach and the above optimization techniques, PSMiner, the first streaming GPM accelerator, is designed in this paper. The experimental results show that PSMiner archives up to 948.5× speedup (60.4× on average) compared to the state-of-the-art accelerator.

References

[1] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *ACM Transactions on Database Systems*, vol. 42, no. 4, pp. 1–44, 2017.

[2] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 2, pp. 1–25, 2017.

[3] L. Bindschaedler, J. Malicevic, B. Lepers, A. Goel, and W. Zwaenepoel, "Tesseract: distributed, general graph pattern mining on evolving graphs," in *Proceedings of EuroSys*, 2021, pp. 458–473.

[4] Q. Chen, B. Tian, and M. Gao, "Fingers: Exploiting fine-grained parallelism in graph mining accelerators," in *Proceedings of ASPLOS*, 2022, pp. 43–55.

[5] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and A. Arvind, "Flexminer: A pattern-aware accelerator for graph pattern mining," in *Proceedings of ISCA*, 2021, pp. 581–594.

[6] G. Dai, Z. Zhu, T. Fu, C. Wei, B. Wang, X. Li, Y. Xie, H. Yang, and Y. Wang, "Dimmining: Pruning-efficient and parallel graph mining on dimm-based near-memory-computing," in *Proceedings of ISCA*, 2022, pp. 1–14.

[7] V. Dias, C. H. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy, "Fractal: A general-purpose graph pattern mining system," in *Proceedings of SIGMOD*, 2019, pp. 1357–1374.

[8] W. Fan, C. Hu, and C. Tian, "Incremental graph computations: Doable and undoable," in *Proceedings of SIGMOD*, 2017, pp. 155–169.

[9] K. Faust, "A puzzle concerning triads in social networks: Graph constraints and the triad census," *Social Networks*, vol. 32, no. 3, pp. 221–233, 2010.

[10] B. Gaüzere, L. Brun, and D. Villemin, "Two new graphs kernels in chemoinformatics," *Pattern Recognition Letters*, vol. 33, no. 15, pp. 2038–2047, 2012.

[11] K. Jamshidi, R. Mahadasa, and K. Vora, "Peregrine: a pattern-aware graph mining system," in *Proceedings of EuroSys*, 2020, pp. 1–16.

[12] M. Mariappan and K. Vora, "Graphbolt: Dependency-driven synchronous processing of streaming graphs," in *Proceedings of EuroSys*, 2019, pp. 1–16.

[13] T. Milenković, W. L. Ng, W. Hayes, and N. Pržulj, "Optimal network alignment with graphlet degree vectors," *Cancer informatics*, vol. 9, pp. CIN–S4744, 2010.

[14] T. Shi, M. Zhai, Y. Xu, and J. Zhai, "Graphpi: High performance graph pattern matching through effective redundancy elimination," in *Proceedings of SC*, 2020, pp. 1–14.

[15] N. Talati, H. Ye, Y. Yang, L. Belayneh, K.-Y. Chen, D. T. Blaauw, T. N. Mudge, and R. G. Dreslinski, "Ndminer: accelerating graph pattern mining using near data processing." in *Proceedings of ISCA*, 2022, pp. 146–159.

[16] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *Proceedings of SOSP*, 2015, pp. 425–440.

[17] Q. Wang, L. Zheng, Y. Huang, P. Yao, C. Gui, X. Liao, H. Jin, W. Jiang, and F. Mao, "Grasu: A fast graph update library for fpga-based dynamic graph processing," in *Proceedings of FPGA*, 2021, pp. 149–159.