2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00052

FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining

Xuhao Chen*, Tianhao Huang*, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, Arvind
Massachusetts Institute of Technology
{xchen,tianhaoh,shuotao,bthom,cwchung,arvind}@csail.mit.edu
*Equal contribution

***Abstract*—Graph pattern mining (GPM) is a class of algorithms widely used in many real-world applications in bio-medicine, e-commerce, security, social sciences, etc. GPM is a computationally intensive problem with an enormous amount of coarse-grain parallelism and therefore, attractive for hardware acceleration. Unfortunately, existing GPM accelerators have not used the best known algorithms and optimizations, and thus offer questionable benefits over software implementations.**

We present FlexMiner, a software/hardware co-designed GPM accelerator that improves the efficiency without compromising the generality or productivity of state-of-the-art software GPM frameworks. FlexMiner exploits massive amount of coarse-grain parallelism in GPM by deploying a large number of specialized processing elements. For efficient searches, the FlexMiner hardware accepts *pattern-specific* execution plans, which are generated automatically by the FlexMiner compiler from the given pattern(s). To avoid repetitive computation on neighborhood connectivity, we provide dedicated on-chip storage to memoize reusable connectivity information in a *connectivity map* (`c-map`) which is implemented with low-cost yet high-throughput hardware. The on-chip memories in FlexMiner are managed dynamically using heuristics derived by the compiler, and thus are fully utilized. We have evaluated FlexMiner with 4 GPM applications on a wide range of real-world graphs. Our cycle-accurate simulation shows that FlexMiner with 64 PEs achieves 10.6× speedup on average over the state-of-the-art software system executing 20 threads on a 10-core Intel CPU.

***Index Terms*—accelerator, software/hardware co-design, graph pattern mining, pattern aware**

I. Introduction

Graph pattern mining (GPM) finds subgraphs that match certain pattern(s) in a given graph (Fig. 1). GPM is a compute-intensive building block in numerous important applications in chemical engineering [22, 48, 71], bioinformatics [7, 19, 60, 61, 70, 74], web spam detection [9, 26, 30, 36], social sciences [29, 31, 34, 41], and others [23, 35, 88, 92]. For example, GPM is used to predict the functionality of a new protein in a protein-protein interaction network, where vertices represent proteins labeled with their functionality and edges are interactions between these proteins. The prediction can be preformed by mining frequent subgraphs with similar interactions to the new protein [19]. We believe GPM would be used even more widely in applications if it were cheaper to compute.

Since writing efficient parallel GPM programs by hand is time-consuming and error-prone, many software GPM systems, such as, Arabesque [82], RStream [84], Fractal [25], Kaleido [95], AutoMine [58], Pangolin [16], Peregrine [44] have been proposed to improve productivity. In general, to solve a GPM problem, a solver program takes a data graph $\mathcal{G}$ and a pattern $\mathcal{P}$ of size k, enumerates all the possible subgraphs of size k in $\mathcal{G}$, and checks each subgraph to see if it is isomorphic to $\mathcal{P}$. This is computationally intensive even on moderate size graphs because of the massive combinatorial search space and the expensive graph isomorphism test. For example, to mine the `4-cycle` pattern (Fig. 3) in the `Friendster` [52] graph with 65 million vertices and 1.8 billion edges, AutoMine [58], one of the fastest GPM software systems, takes 15 hours on a 4-socket 14-core (56-core in total) Intel CPU machine. Therefore, hardware GPM accelerators like TrieJax [46] and Gramer [90] have been proposed to improve GPM's performance and energy-efficiency.

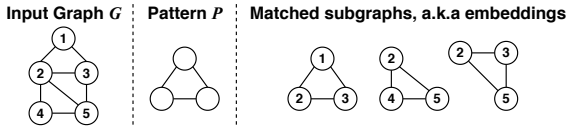

Fig. 1: Graph pattern mining example. The pattern $\mathcal{P}$ is a triangle, and 3 triangles are found in the data graph $\mathcal{G}$.

Subgraph enumeration to match a pattern can be modeled as building a *search tree*. The efficiency of any GPM solution, to the first order, depends upon the size of this *search tree*. *Pattern-aware* solutions exploit the specific properties of a pattern to drastically prune the search space. Unfortunately, existing hardware accelerators are inefficient because they use naive *pattern oblivious* search strategies, resulting in significantly larger search space than the state-of-the-art software solutions. Meanwhile, existing accelerators also have limited generality. TrieJax, unlike a software GPM system, supports only a small subset of GPM problems. Gramer supports more GPM problems but adopts an ad-hoc approach which requires implementing a specific solver for each specific GPM problem, significantly hampering its usability. The limited generality is a consequence of the fact that Gramer and TrieJax lack a well-defined software/hardware interface to configure/program the accelerator. Lastly, unlike GPM software systems, existing accelerators lack the support for memoization and thus *repetitively* check the connectivity of vertices during the search.

We propose FlexMiner, a software-hardware co-designed GPM system, which strives for both generality and performance without sacrificing the ease of programming. Fig. 2

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00052

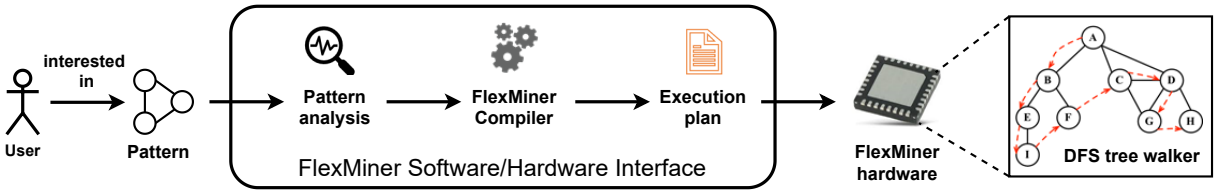

Fig. 2: An overview of the FlexMiner software/hardware co-designed system.

illustrates an overview of the FlexMiner system which contains the FlexMiner hardware and a software/hardware interface. The FlexMiner hardware is a special-purpose architecture to offload the GPM computations of an application running on the host. It does a depth-first search (DFS) over the search tree. The software/hardware interface is established between the user program and the hardware, to make FlexMiner **pattern-aware**. The user program only needs to specify the pattern(s) of interest, same as state-of-the-art software GPM frameworks. FlexMiner first does an analysis on the pattern(s), and then uses a compiler to automatically generate a pattern-specific *execution plan*. This *execution plan* is loaded by the host CPU to the FlexMiner hardware at the beginning of execution, and customizes the DFS search process in hardware. In this way, FlexMiner can flexibly mine any arbitrary pattern with high algorithmic efficiency. In addition to this flexibility, the FlexMiner hardware also provides much higher throughput than general purpose CPUs. The efficiency of the FlexMiner hardware stems from the following features:

Massive Multithreading: FlexMiner exploits the embarrassing amount of parallelism in GPM which comes from the fact that the searches starting from different vertices of $\mathcal{G}$ are mutually independent *tasks* and can be done concurrently. FlexMiner consists of a collection of processing engines (PEs) to exploit this massive parallelism to saturate the memory bandwidth and hide memory latency.

PE Specialization: Each PE contains a specialized unit that is optimized for efficient set intersection and set difference operations. These operations frequently compare neighbor list of vertices, and are the major bottlenecks in state-of-the-art software GPM frameworks.

Connectivity Memoization: The set operations access neighbor lists in $\mathcal{G}$ frequently and repeatedly, which results in a significant amount of redundant computation. We use a novel `c-map` structure (Section VI) to memorize the neighborhood information and avoid this redundancy with low-cost hardware. Besides, the FlexMiner compiler passes pattern-specific hints to the hardware to help efficiently manage `c-map` storage.

We prototype the PE of FlexMiner in RTL using Bluespec and synthesize the design with Silvaco's 15nm Open-Cell Library. Our PE with 32kB private cache and 8kB scratchpad runs at 1.3GHz and takes an area of 0.18mm^2. In comparison, a single high-end Intel SkyLake core with 1MB L2 cache is clocked around 4GHz and takes an area of 15mm^2 [46]. Detailed performance simulation shows that with 64 PEs (which take approximately the same area as a single Intel CPU core and operate at one third of its clock speed), FlexMiner achieves 10.6× speedup on average over the state-of-the-art software solution on a 10-core Intel i9-7900X CPU. In summary, this paper makes the following contributions:

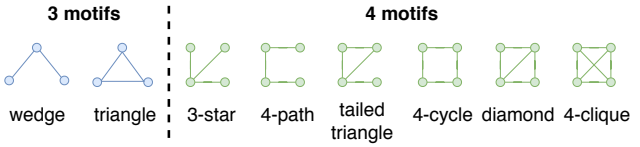

Fig. 3: 3-vertex (left) and 4-vertex (right) motifs.

- FlexMiner, *the first pattern-aware* GPM accelerator which supports a variety of GPM applications using techniques from state-of-the-art software algorithms for GPM;
- A *programming interface* which is the same as the state-of-the-art software GPM systems. We use a small compiler to generate the *execution plan* to configure the hardware;
- *Hardware support* for memoizing connectivity information to avoid redundant computation with low cost;
- *Evaluation* using representative GPM applications and diverse graphs to show that FlexMiner significantly improves performance over state-of-the-art software frameworks.

Paper organization: Section II defines GPM, introduces 4 GPM problems, and describes how a GPM problem is solved in state-of-the-art software GPM systems. In Section III we point out the limitations of existing hardware to motivate our work. Section IV introduces the FlexMiner architecture. Section V defines the software/hardware interface. We discuss our hardware support for `c-map` in Section VI. Section VII evaluates our design and Section IX concludes.

II. Background

A. Graph Pattern Mining Problems

Given a graph $G(V, E)$ and a vertex set $V' \subseteq V$, the *vertex-induced subgraph* is the graph G' whose vertex set is V' and edge set contains the edges in E whose endpoints are in V'. Given $G(V, E)$ and an edge set $E' \subseteq E$, the *edge-induced subgraph* is the graph G' whose edge set is E' and vertex set contains the endpoints in V of the edges in E'.

Given a data graph $\mathcal{G}$ and a set of patterns $S_p = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, GPM seeks to find all those subgraphs, i.e., *embeddings*, in $\mathcal{G}$ that are isomorphic to $\mathcal{P}_i$, $\forall \mathcal{P}_i \in S_p$. Fig. 1 shows an example of triangle embeddings in a graph. A GPM solution should guarantee *completeness*, i.e., every match of $\mathcal{P}$ in $\mathcal{G}$ should be found, and *uniqueness*, i.e., every distinct match should be reported only once. We consider the following GPM problems in this paper:

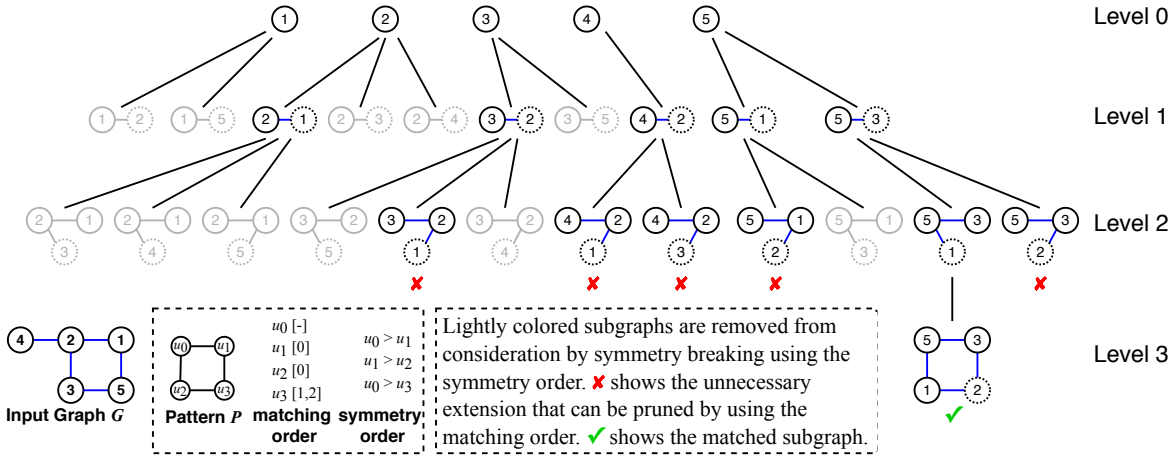

Fig. 4: A portion of a subgraph search tree with 4 levels (vertex extension used).

- *Triangle counting* (TC) counts the number of triangles in $\mathcal{G}$.
- *k-clique listing* (k-CL) lists all k-cliques in $\mathcal{G}$, where a k-clique of G is defined as a subgraph G' of G, such that G' has k vertices and every pair of vertices in G' is connected by an edge in G', i.e., G' is a complete graph.
- *Subgraph listing* (SL) enumerates all *edge-induced* subgraphs of $\mathcal{G}$ isomorphic to a user-defined pattern $\mathcal{P}$.
- *k-motif counting* (k-MC) counts the number of occurrences of the different patterns that are possible with k vertices. Each pattern is also called a *motif*. Fig. 3 shows all 3-motifs and 4-motifs. k-MC finds *vertex-induced* subgraphs.

A GPM problem to mine one single pattern is called a single-pattern problem, e.g., TC, k-CL and SL, whereas a problem to find multiple patterns simultaneously is called a multi-pattern problem, e.g., k-MC.

B. GPM Execution Model and Search Strategies

To solve a GPM problem, a solver program enumerates all the possible subgraphs of size k in $\mathcal{G}$, and checks each subgraph to see if it is isomorphic to $\mathcal{P}$. This solving process can be modeled as building a *search tree* in which each vertex represents a subgraph of $\mathcal{G}$. Fig. 4 shows a *search tree* with 4 levels. Level l of the tree represents subgraphs with $l+1$ vertices. Intuitively, subgraph $S_2{=}(W_2, E_2)$ is a child of subgraph $S_1{=}(W_1, E_1)$ in this tree if S_2 can be obtained by extending S_1 with a single vertex $v \notin W_1$ which is connected to some vertex in W_1 (v is said to be in the *neighborhood* of subgraph S_1); this process is called *vertex extension*. Formally, this can be expressed as $W_2{=}W_1 \cup \{v\}$ where $v \notin W_1$ and there is an edge $(v, u) \in E$ for some $u \in W_1$. It is useful to think of the edge connecting S_1 and S_2 in the tree as being labeled by v. Similarly, *Edge extension* extends a subgraph S_1 with a single edge (u, v) provided at least one of the endpoints of the edge is in S_1.

Based on the search tree model, we classify existing software GPM frameworks into two categories: *pattern-oblivious* and *pattern-aware*. The pattern-oblivious approach builds the search tree and checks each leaf to determine if it is isomorphic to $\mathcal{P}$. In contrast, a pattern-aware solution leverages the properties of $\mathcal{P}$ to prune the search tree and avoid isomorphism tests. The pattern is analyzed to generate a *matching order* [44] and a *symmetry order* [57] which are used to guide the search and prune the search space. We define *connected ancestor* and describe the two orders below. To avoid notational confusion, we call a vertex in $\mathcal{P}$ as a *pattern vertex*, denoted as u_i, and a vertex in $\mathcal{G}$ as a *data vertex*, denoted as v_i.

Connected ancestor: Given a subgraph $\mathcal{S}$ and a total order $\mathcal{TO}(\mathcal{S})$ on the vertices in $\mathcal{S}$, a *connected ancestor* of vertex u in $\mathcal{S}$ under $\mathcal{TO}$ is a vertex w in $\mathcal{S}$ such that $w < u$ and w is connected to u. The *connected ancestor set* that includes all the connected ancestors of u in $\mathcal{S}$ under $\mathcal{TO}$, is denoted as $\mathcal{CA}_{\mathcal{S},\mathcal{TO}}(u)$. For example, $\mathcal{CA}_{\mathcal{S},\mathcal{TO}}(u_3) = \{u_1,u_2\}$ means u_1 and u_2 are u_3's connected ancestors in $\mathcal{S}$ under $\mathcal{TO}$. For simplicity, we write $\mathcal{CA}(u)$ instead of $\mathcal{CA}_{\mathcal{S},\mathcal{TO}}(u)$ when $\mathcal{S}$ and $\mathcal{TO}$ are obvious from the context.

Matching order: It is a total order among pattern vertices that defines the order to match each data vertex to a pattern vertex. For example, in Fig. 4, suppose $\mathcal{P}$ is a `4-cycle`, and we generate its matching order $\mathcal{MO}$ and represent it as a list of connected ancestor set of u_i in $\mathcal{P}$ under $\mathcal{MO}$: $\{\mathcal{CA}(u_0){=}\{\}$, $\mathcal{CA}(u_1){=}\{u_0\}$, $\mathcal{CA}(u_2){=}\{u_0\}$, $\mathcal{CA}(u_3){=}\{u_1,u_2\}\}$, meaning (1) any vertex v_0 in V can be mapped to u_0; (2) v_1 mapped to u_1 must be a neighbor of v_0; (3) v_2 mapped to u_2 also must be a neighbor of v_0; (4) v_3 mapped to u_3 must be a common neighbor of v_1 and v_2. In level 2 of the tree in Fig. 4, subgraphs marked with $\times$ are pruned, because the vertex mapped to u_2 (i.e., v_2) is not a neighbor of the vertex mapped to u_0 (i.e., v_0). Note that using matching order eliminates the need to apply isomorphism tests at the leaves of the search tree, because the leaf subgraphs always match $\mathcal{P}$ (see level 3 in Fig. 4).

Symmetry order: A specific subgraph can occur in multiple places in the search tree. For example, in Fig. 4, the subgraph containing vertices 1 and 2 occurs in two places at level 1 of the tree. These identical subgraphs are called *automorphisms*. To avoid repetitive enumeration, only one of them, known as the *canonical* one, is kept, and extended further. This selection is known as symmetry breaking [57]. A well established

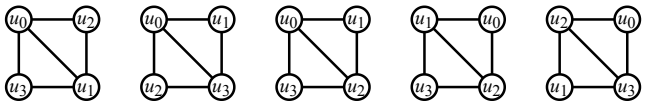

Fig. 5: Possible matching orders for `diamond`.

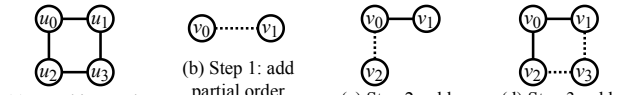

Fig. 6: Generating symmetry order for `4-cycle`.

approach for symmetry breaking is to define a partial order, known as a *symmetry order*, for candidate vertices and add only those subgraphs that satisfy the symmetry order.

In Fig. 4, suppose we define the symmetry order for the `4-cycle` pattern as $\{v_0 > v_1, v_1 > v_2, v_0 > v_3\}$. Using this order, the lightly colored subgraphs can all be pruned, which reduces the search space and guarantees uniqueness. Specifically, the subgraphs [1, 2] and [2, 1] are automorphisms, and the former can be pruned because $v_0 > v_1$.

Given $\mathcal{P}$, pattern-aware solutions use *pattern analysis* [58] to generate a matching order [44] and a symmetry order [57], which, in turn, can be used to generate a program to find $\mathcal{P}$ automatically. To generate a matching order, the pattern analyzer first enumerates all the possible matching orders of $\mathcal{P}$, and uses a set of rules to pick one that is likely to perform well in practice [49]. Fig. 5 lists all the possible matching orders for `diamond`. The first matching order searches for a triangle first (u_0, u_1, u_2 form a triangle), and the second matching order searches for a wedge first (u_0, u_1, u_2 form a wedge). Since the number of triangles is much fewer than the number of wedges in a sparse graph, the first matching order is likely to perform better than the second as it prunes many more candidates at the early stage. Using the rules in [49], we find that the first matching order is the best out of the five.

To generate a symmetry order for a pattern $\mathcal{P}$, we first take one its matching orders, ($\mathcal{MO}$) and build a subgraph incrementally in the order specified by $\mathcal{MO}$. At each step we also check for the symmetry (Fig. 6). The matching order in (a) results in the 3 steps shown in (b), (c) and (d). In (b), we add partial order $v_1 < v_0$ as v_0 and v_1 are *interchangeable*, meaning that if no order is enforced between v_0 and v_1 then any match of $\mathcal{P}$ in $\mathcal{G}$ will be found twice by permuting v_0 and v_1. Similarly, in (c) and (d) we add partial orders $v_2 < v_1$ and $v_3 < v_0$, respectively. In all, the symmetry order generated for `4-cycle` is $\{v_0 > v_1, v_1 > v_2, v_0 > v_3\}$.

C. Memoizing Connectivity Information

A major computation in AutoMine [58], Peregrine [44] and GraphZero [57] is set intersection/difference on edgelists. This requires frequent accesses to the edgelists of $\mathcal{G}$ during the search. For example, for pattern `4-cycle` in Fig. 6 (a), v_3 (match of u_3) is from the intersection of v_1's neighbors and v_2's neighbors. For every different v_2, the same v_1's neighbors are visited. To avoid repeated lookups in $\mathcal{G}$, we can memoize v_1's neighbors in a *connectivity map* (`c-map`) during the search. Each entry in the `c-map` is a key-value pair, where the key is a vertex ID (say v), and the value is a list of depths of vertices in the *current embedding* which are connected to v. This list is implemented as a bitset to save space. For example, assume that the current embedding has 3 vertices, the entry [w, '001'] means vertex w is connected to the first vertex, but not to the second and the third vertices in the current embedding. Once the `c-map` is constructed, all the set operations can be replaced by querying the `c-map`. For example, for `4-cycle`, the intersection is replaced by querying the `c-map` to check if each neighbor of v_2 is connected with v_1.

III. Limitations of Existing GPM Software Systems and Accelerators

Software GPM systems, such as Arabesque [82], RStream [84], Fractal [25], Kaleido [95], AutoMine [58], Pangolin [16], Peregrine [44], GraphZero [57] simplify GPM programming and apply algorithmic optimizations to improve performance. However, given the irregularity of the GPM computation, software systems often show poor performance. To get in-depth understanding of how software GPM systems behave on general-purpose processors, we evaluated AutoMine [58], a state-of-the-art software GPM system, on a 12-core machine with 100GB/s max DRAM bandwidth and hyper-threading (2 threads per core).

Fig. 7 shows the performance (left) and memory bandwidth (right) scaling of mining k-cliques in `orkut` (Table I). The performance scales linearly until 12-thread, after which hyper-threading kicks in and the scaling slows down probably because of cache contention [8]. We observe the same behavior using a 56-core machine, when we scale from 56-thread to 112-thread. Fig. 7 also shows that the memory bandwidth scales well beyond 12 threads, indicating there is potential to improve performance with more physical cores.

In another experiment using Intel VTune, we observe that 37%∼49% of pipeline slots are wasted due to branch mis-prediction. This is caused by the frequent comparison and branching in set intersection/difference, which account for a majority of computation in AutoMine. These observations suggest that an accelerator with a large number of physical cores with special support for set operations and local memory should be an effective way to scale GPM performance.

Unfortunately, existing hardware GPM accelerators, TrieJax [46] and Gramer [90], are not efficient. TrieJax is based on a variant of the Worst Case Optimal Join (WCOJ) algorithm. The `join` operations are used to perform set intersections. TrieJax introduces a specialized core which works as a co-processor with the CPU cores. The L1 and L2 caches of the specialized core are read-only, to avoid cache pollution by the writes. It also introduces a 4MB dedicated on-chip scratchpad, called the Partial Join Results (PJR) Cache, to buffer intermediate results of `join` operations. However, TrieJax has some limitations. First, it does not perform symmetry breaking, leading to a much larger search space than necessary. Second, it suffers from redundant computation on connectivity

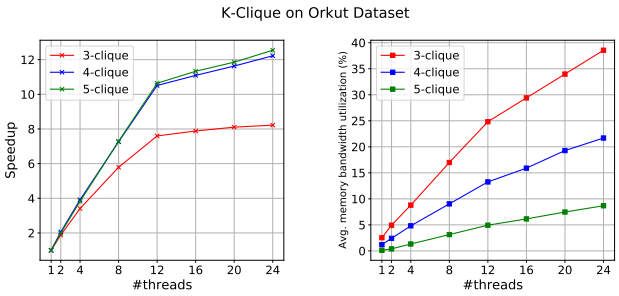

Fig. 7: Scalilibity of k-CL on 12 dual-threaded cores with maximum DRAM bandwidth of 100GB/s.

check. Third, it lacks an interface to configure the hardware and thus, is limited to solving single, edge-induced problems.

Gramer [90] proposed a data prioritization technique for GPM, which heuristically prioritizes the *hot*, i.e., frequently accessed, data in the cache. This technique improves performance over existing pattern-oblivious software systems, such as RStream [84] and Fractal [25]. However, Gramer employs a pattern-oblivious search strategy. Although a `filter` mechanism in Arabesque is leveraged by Gramer to remove irrelevant subgraphs, it is not sufficient enough [16] to prune the search space for an arbitrary pattern, and because of a lack of the matching order, Gramer requires expensive isomorphism tests. While Gramer supports solving more GPM problems than TrieJax, it requires re-synthesizing a hardware solver for each GPM problem due to a lack of programmable interface. Extra RTL programming efforts are expected from users, which could be a significant task for one without expertise in both hardware design and graph pattern mining.

IV. FlexMiner Design Overview

The limitations of existing pure software and pure hardware solutions motivate us to develop a SW/HW co-designed GPM system which can overcome these limitations.

As shown in Fig. 2, FlexMiner establishes a software/hardware interface (Section V), which takes the user specified pattern as input, does an analysis on the pattern, and invokes the FlexMiner compiler to generate a pattern-specific execution plan. The execution plan is fed into the FlexMiner hardware at the beginning of execution. The FlexMiner hardware can be considered as a template of DFS walker over the subgraph search tree, which is implemented as a finite-state machine (Section IV-B). This template is customized by the pattern-specific execution plan, and therefore the hardware is pattern aware. To provide high throughput, the hardware contains a collection of processing elements (PEs) specialized for GPM search. This architecture leverages the parallelism that comes from the fact that each search task starting from a vertex is independent from other tasks. Inside each PE, FlexMiner introduces a hardware hashmap (Section VI) to efficiently perform the major GPM computation. In case of hashmap overflow, a specialized set operation unit is invoked.

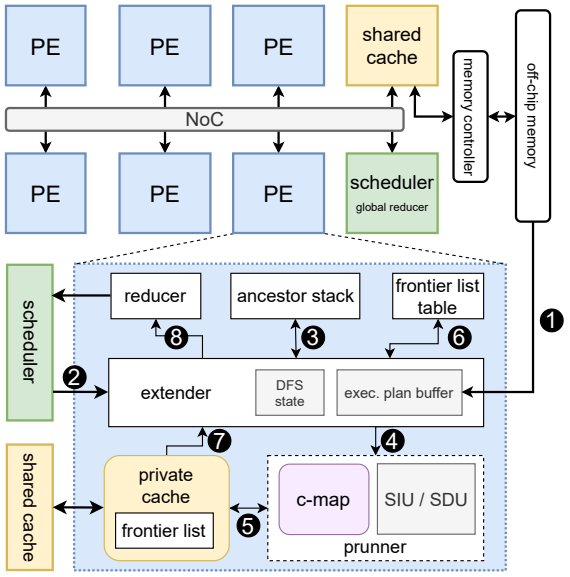

Fig. 8: The FlexMiner hardware architecture.

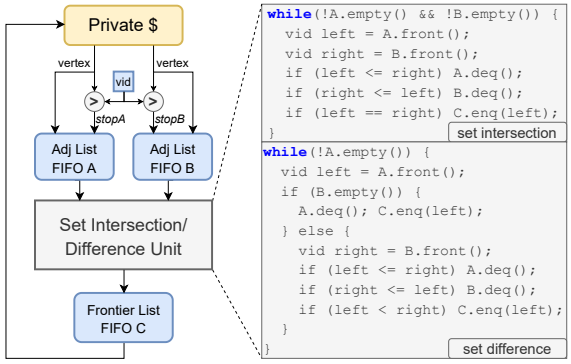

Fig. 9: Hardware SIU/SDU with upper bound pruning. `vid` is the vertex upper bound. A and B are input edgelists.

A. FlexMiner Hardware Architecture

Fig. 8 illustrates the FlexMiner architecture. FlexMiner consists of a `scheduler`, a `shared cache`, and a number of `processing elements` (PE), all connected with a network on chip (NoC). The `scheduler` dynamically assigns *tasks* to available idle PEs. The `shared-cache` buffers the vertex/edge data for all PEs, and the intermediate data spilled from PEs. There is no cache coherency in FlexMiner because each task is independent and there is no updates to shared data. Each PE is responsible for processing assigned tasks independently and does not require synchronization with other PEs.

Each PE contains the following components. The `extender` is a finite-state machine (detailed in Section IV-B) responsible for adding vertices one-by-one in the search tree. The `pruner` is used to prune the vertex candidates and is configured specifically for the pattern prior to the execution. It checks if the ID of the vertex being added is within the bounds of the symmetry order and queries `c-map` to check the connectivity constraints. Only when the `c-map` has overflowed, `SIU` and `SDU` are invoked to perform set intersection/difference operations

to compute the qualified vertex candidates. SIU/SDU uses the well-known merge-based algorithm [39, 42] and its hardware structure is shown in Fig. 9. Our specialized SIU and SDU perform one loop iteration (the while loop in Fig. 9) per cycle.

The reducer contains a group of counters, one for each pattern. It uses + as the reduction operation, but can be extended easily to support user-defined reduction operations. The ancestor stack is a set of registers to hold the current partial match during the DFS search. The private cache stores the edgelist data, and memoizes the frontier list which is a list of vertices that have already passed the constraint checking and are going to be reused (detailed in Section V-C). Once a frontier list is generated, its start address and size are stored in the frontier list table. The scratchpad accommodates a specialized hash map to memoize the neighborhood connectivity (detailed in Section VI). Next we show details of the state machine and the FlexMiner execution flow.

B. FlexMiner Execution Flow

Pattern-aware software solutions [44] use recursion, which is not suitable for direct implementation in hardware. Instead, FlexMiner uses the *iterative execution model* shown in Fig. 10 which can be implemented using a simple finite state machine.

A PE can be in one of the three states at runtime: Idle, Extending and IteratingEdges. d is the the depth counter, and emb is the ancestor stack in Fig. 8. Whenever the DFS steps to the next level, a vertex is pushed on to the stack. When the traversal finishes searching at the current level, the stack pops the vertex at the top and backtracks to the previous level. For each depth of the search tree, there are registers to hold the current vertex being extended and the index of edge used for extension (i.e., i-th neighbor).

The execution starts from a vertex v_{init}, which is a task assigned by the scheduler. v_{init} is pushed on to the stack, and the depth is set to 1. The state is switched to Extending. At this stage, if the depth has reached the size of the pattern (k), a match has been found in the stack and the matched subgraph is added to the output. After that the search backtracks to the previous level by decreasing d and popping emb. If the maximum depth not reached yet, a vertex v in emb is picked (according to the matching order $order$) as the new vertex for extension and the index is set to 0. Now the state switches to IteratingEdges. If the index has come to the end of the neighbor list, the search backtracks to the previous level (if the previous level is in depth 0, indicating the entire subtree is already traversed, it goes back to idle stage and waits for the next task from the scheduler). Otherwise, the next neighbor of the extender u is considered as a candidate vertex to be checked by the pruner. If u satisfies the constraints (i.e., the symmetry order and connectivity), u is pushed into emb and the search goes to the next level by increasing d and switching to the Extending state.

The overall flow of FlexMiner can be explained in Fig. 8 as follows. At the beginning of execution, the execution plan is loaded (❶). The scheduler then assigns a task (vertex v) to the PE (❷). The extender puts v in the ancestor stack

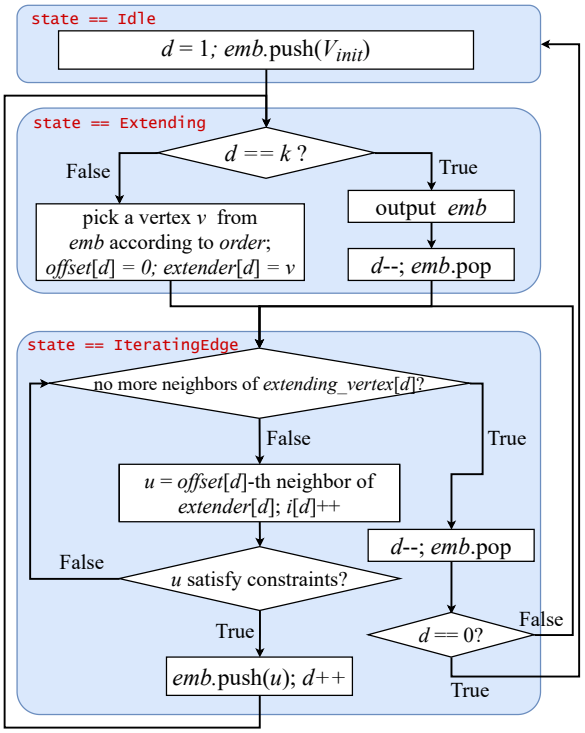

Fig. 10: The execution flow of FlexMiner (single-pattern).

(❸), and configures the pruner (❹) according to the execution plan. The pruner then starts to load v's edgelist from private cache (❺), checks vertex id bound and queries the c-map for connectivity constraints (if c-map overflows, SIU/SDU is invoked). The other intermediate data, frontier list, stays in the private cache, and is written to the shared cache when evicted from the private cache. Once a frontier list is generated, its corresponding information is updated in the table (❻). The frontier list is reused from the private cache when it is accessed in a deeper level (❼). Whenever the extender finds a match, the reducer increases the local count (❽), which is sent to the global reducer at the end.

V. Software/Hardware Interface

To efficiently support pattern-awareness, in FlexMiner, we define an interface to pass a pattern-specific *execution plan* to the hardware. We propose an intermediate representation (IR) to express the (1) matching order, (2) symmetry order and (3) the hints to manage on-chip storage, for a specific pattern. Fig. 11 (a) shows the *execution plan* for the 4-cycle pattern and its IR is given in Listing 1. We develop a *execution plan generator* (referred as a "compiler") to do the pattern analysis and generate the IR code automatically. IR code is pre-loaded to the hardware accelerator before the execution starts. Our compiler generates matching order and symmetry order using the same approach in prior software frameworks [44, 57, 58]. We describe how it generates the hints to manage on-chip storage and how multi-pattern problems are supported.

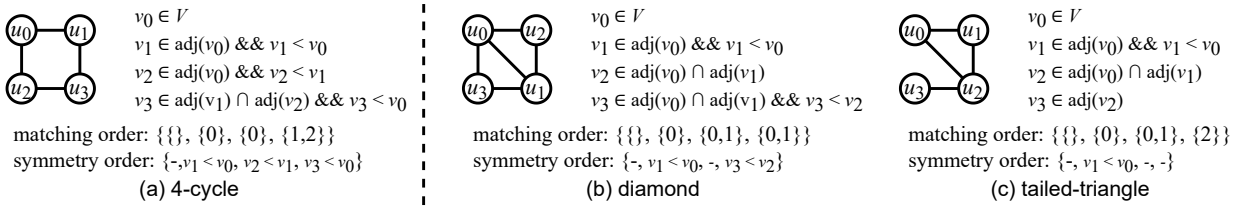

Fig. 11: Execution plans for finding edge-induced `4-cycle`, `diamond` and `tailed-triangle`. Errors in this figure fixed.

Listing 1: IR code for `4-cycle`

```
1  vertex:
2    v0 ∈ V      pruneBy (∞,    {})
3    v1 ∈ v0.N   pruneBy (v0.id, {})
4    v2 ∈ v0.N   pruneBy (v1.id, {})
5    v3 ∈ v2.N   pruneBy (v0.id, {v1})
6  embedding:
7    emb0 := v0
8    emb1 := emb0 + v1
9    emb2 := emb1 + v2
10   emb3 := emb2 + v3
```

Listing 2: IR code for finding `diamond` and `tailed-triangle`

```
11 vertex:
12   v0 ∈ V      pruneBy (∞,    {})
13   v1 ∈ v0.N   pruneBy (v0.id, {})
14   v2 ∈ v0.N   pruneBy (∞,    {v1})
15   v31 ∈ v0.N  pruneBy (v2.id, {v1})
16   v32 ∈ v2.N  pruneBy (∞,    {})
17 embedding:
18   emb0 := v0
19   emb1 := emb0 + v1
20   emb2 := emb1 + v2
21   emb31 := emb2 + v31    emb32 := emb2 + v32
```

A. IR Format

The IR consists of a *vertex section* and an *embedding section*. The *vertex section* describes how each vertex is extended and which vertices can be considered valid candidates for that extension step. For example, line 3 shows v_1 is extended from v_0. The candidates are confined by the `pruneBy` primitive, which accepts 2 parameters: a vid upper-bound and connected ancestor set. The vid upper-bound defines the upper bound of candidate vertex ID. The connected ancestor set contains all the vertices in the current embedding that must be connected to the candidate vertex. Only candidate vertices that conform to both constraints can be added to the embedding. For example, in line 5 of Listing 1, v_3's ID must be smaller than v_0's ID, and v_3 must be connected to v_1. The *embedding section* describes the dependencies between partially matched embeddings given by the matching order. For instance, when mining `4-cycle`, we will find the following partial embeddings in order: single-vertex embedding (emb_0), edge (emb_1), wedge (emb_2) and 4-cycle (emb_3). The embedding at the tail of the chain (emb_3) fully matches the pattern. Each link in a dependency chain is labeled with a `+` primitive which defines the *vertex-adding* action performed to extend the embedding. For example, line 9 shows that emb_1 is extended from emb_0 by adding v_1.

B. Multi-pattern Support

For a single-pattern problem, it is straightforward to use a compiler to generate the IR code from the matching order and symmetry order. In case of multiple patterns, each pattern needs one dependency chain. Since multiple chains may contain a common part, we merge multiple chains using a dependency tree whenever possible. This way, the embedding section can express the control flow of mining multiple patterns simultaneously, with common search paths merged to avoid repetitive enumeration. Fig. 11 (b) and (c) shows the execution plans of `diamond` and `tailed-triangle`. We find that for both patterns, v_0, v_1 and v_2 are from the same candidates set, i.e. the constraints of the first 3 vertices are the same for both patterns. Therefore, we merge the first three matching steps, and only the last step will diverge. Listing 2 shows the generated IR code. As illustrated in line 12, 13 and 14, the first three vertices are the same. This chain diverges when extending v_2 to v_3 where two different branches are represented by two vertices v_{31} and v_{32} in line 15 and 16 respectively. The embedding section shows the dependency tree. Line 23 shows the two branches emb_{31} and emb_{32}, both stemming from emb_2. In this way, FlexMiner can efficiently support any multi-pattern problem including k-MC.

C. Hints for Data Management

Different patterns can be optimized with different `frontier list` memoization techniques. For example, in the `diamond` pattern in Fig. 11 (b), v_2 and v_3 are from the same candidate set $adj(v_0) \cap adj(v_1)$, and the only difference is that $v_3 < v_2$. If we can memoize the result of $adj(v_0) \cap adj(v_1)$ in the PE-local cache, we can avoid repetitive computation. When analyzing the pattern, the compiler identifies which results are reusable and thus should be memoized, and indicates the hardware using a flag in the IR code. Similarly, the compiler also embeds information in the IR about how to manage the `c-map`. We describe it in Section VI.

In addition, the compiler does special optimization when detecting k-clique at pattern analysis, since symmetry breaking can be done by the *orientation* technique, i.e., converting the undirected data graph $\mathcal{G}$ into a directed acyclic graph (DAG) [16]. The idea is to establish an order between the endpoints of every edge in $\mathcal{G}$, which converts the originally undirected edge into directed. A commonly used approach is to enforce the vertex with smaller degree points to the vertex with larger degree. Vertex ID is used when there is a tie. After giving orientation, no symmetry order checking is needed at runtime. The preprocessing time is usually less than 1% of the

execution time, and once converted, the graph can be used for any k-CL.

D. Pattern-Aware Execution Flow

With execution model in Fig. 10, FlexMiner can mine any pattern of interest by customizing $order$ and $constraints$ at each level. The customization is achieved simply by downloading the IR code to the hardware. In the example of `4-cycle` in Fig. 11 (a), when FlexMiner extends a `wedge` to a `4-cycle` in depth 3 (executing $emb_3 := emb_2 + v_3$), it refers to the primitive $v_3 \in v_2$.N `pruneBy` (v_0.id, $\{v_1\}$) in the vertex section. This defines the $extender$ to be v_2 (i.e., the third vertex in the ancestor stack), and the constraints include that $\{v_3 < v_0\}$ and v_3 must be connected to v_1.

For single-pattern mining, the control flow (dependencies between partial embeddings) is a sequence, so the execution model can simply use depth d as the index into the *vertex section* to retrieve the primitives, without using the *embedding section*. For multi-pattern problems, however, the *embedding section* will be used to handle the divergence of the control flow due to different execution plans of the multiple patterns. For example, in line 23 of Listing 2, two branches (emb_{31} and emb_{32}) are explored sequentially, i.e., the DFS walk steps into level 3 with emb_{31} first; when search finished in level 3 and backtracks to level 2, it steps into level 3 again with emb_{32}.

VI. Hardware Support for Connectivity Map

In this section we explain our proposed hardware support for `c-map` optimization in FlexMiner. First, let's look into the details about how `c-map` is updated and queried at runtime.

A `c-map` can be constructed incrementally each time an embedding is extended with a new vertex. As shown in Fig. 12, the current embedding is [] → [1] → [1, 2] → [1, 2, 3]. The `c-map` is empty at the beginning. When vertex 1 is added to the current embedding, all neighbors of vertex 1 are inserted into the map (❶), and the value of these entries are all '001', indicating they are connected to vertex 1. When vertex 2 is added to the embedding, all neighbors of vertex 2 are inserted (❷) with value '010', indicating they are connected to vertex 2. Since vertex 4 and 5 already exist, their values are updated to '011', indicating they are connected to both vertex 1 and 2. Since vertex 2 is in the embedding, the entry for vertex 2 in the map become useless (no need to update). Similarly, when vertex 3 is added to the embedding, the neighbors of vertex 3 are inserted or updated (❸). Note that `c-map` is self-cleaned during backtracking, i.e. the connectivity information is resumed in a stack fashion. Therefore when a task is completed, all entries in `c-map` are invalidated.

When extending the current embedding with one more vertex (e.g., vertex 7), we want to check the connectivity of vertex 7 with vertex 1, 2, and 3. By querying the map with the key '7', we get the bitset '101', indicating vertex 7 is connected with vertex 1 and vertex 3, but not vertex 2. If the lookup key does not exist in the map, it means the vertex is not connected to any of the vertices in the current embedding.

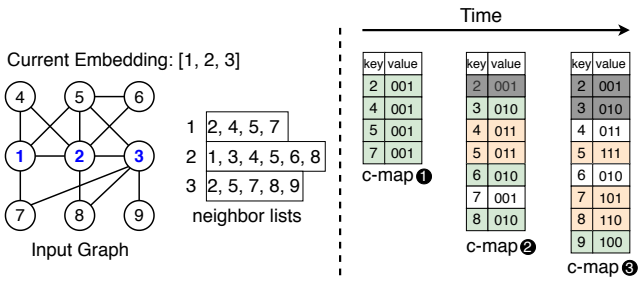

Fig. 12: An example of `c-map`. Green entries are newly inserted. Orange entries are updated. Grey entries are ones became useless.

Previous work uses a `vector` [15, 21] to implement a `c-map` in software. Instead of keeping only visited vertices in the map, the vector pre-allocates $|V|$ entries, one per vertex in $\mathcal{G}$. In such implementations, `c-map` queries are served in constant time, which leads to an average of 2.3× speedup for k-CL [21]. However, vector `c-map` only work well in a restricted setting. First, such implementation is not scalable as the size of the data graph $\mathcal{G}$. For example when counting the 4-cliques in a graph of a billion vertices of maximum degree 50, the working set can be bound by size 50, but the vector implementation will have to allocate a vector of size one billion (GB) per thread. Moreover, the L1 cache is particularly inefficient for such structures: each cache line holding `c-map` information may hold only a single byte of useful information.

A. Hardware `c-map` Design

We propose a hardware `c-map` which keeps the information in a compact form and provides fast accesses and low hardware complexity. We use a *simplified linear probing* scheme: the insertions and lookups are the same, but deletion is simpler than the conventional linear probing. In our `c-map`, to delete an entry, we simply find the entry and invalidate it (i.e., set the value to 0). This is functionally correct due to two key observations in the GPM algorithms: (1) updates on `c-map` work in a bulk fashion (a sequence of neighbors), thus the deletion of the entries inserted at the same level is done atomically (no lookups before all removed); (2) we never delete a key that does not exist in the map, thus the deletion operation will always find the entry and remove it.

To probe faster, we partition the `c-map` into m banks, allowing parallel probes to m successive entries. We prototype our hardware `c-map` design in Bluespec and test it with m=4 and a bank size of 512 lines of 5 bytes: 4 bytes of the key, one byte of the value (for a total of 2K entries). These parameters led to a design successfully synthesized for FPGA at 200MHz and ASIC at 1.3GHz. We empirically observe that the map should be properly sized (>4kB) to keep its occupancy below 75%, thus maintain a low expected access latency. In our design, most accesses take only a single cycle.

B. Support for `c-map` Space Management

Different patterns use the connectivity information differently. For `4-cycle`, line 5 in Listing 1 shows that v_3 is searched from v_2's neighbor list, and is checked if it is

Graph	Source	# V	# E	$\hat{d}$ (max degree)
As	ca-AstroPh [52]	18,772	0.2M	504
Mi	Mico [27]	0.1M	1.1M	1,359
Pa	Patent [37]	2.7M	14.0M	789
Yo	Youtube [17]	7.1M	57.1M	4,017
Lj	LiveJournal [52]	4.8M	42.9M	20,333
Or	Orkut [52]	3.1M	117.2M	33,313

TABLE I: Input graphs (symmetric, no loops or duplicate edges)

connected to v_1. The compiler annotates that only v_1's connectivity information is used. Therefore, when mining `4-cycle`, we only need to insert v_1's neighbors to `c-map`. Besides, in line 5 there is a vertex upper bound v_0, thus our compiler prevents any v_1's neighbor with VID larger than v_0 from being inserted into `c-map`, reducing the number of entries in `c-map` further.

Finally, we provide a fall-back mechanism to guarantee correct execution when `c-map` overflows. To decide when to fall back, the size of the neighbor list (i.e., the degree) is obtained before bringing the list, and then we compute how each vertex extension influence the `c-map` memory footprint. Hence, we dynamically estimate the occupancy of the `c-map` and keep it below our chosen threshold.

When we detect that estimated `c-map` footprint is above the threshold, the fall-back mechanism is activated, i.e., the `pruner` switches to invoke `SIU/SDU` instead of querying `c-map`. For example, for `4-cycle`, once `c-map` allocation for v_1's neighbors failed, at level 2, instead of extending v_2 and checking connectivity with v_1, the `pruner` sends requests of v_1's and v_2's edgelists to the L1 cache, and the two edgelists are sent to `SIU` to compute the intersection.

VII. Evaluation

A. Experimental Setup

Benchmarks and Graph Datasets: We test 4 GPM applications discussed in Section II-A, i.e., TC, k-CL, SL, k-MC. As listed in Table I, we use all the input graphs used in Gramer [90], except the two smallest graphs, which provide much higher speedups over software baseline and distort the results. All input graphs are symmetric, have no self-loops, and have no duplicated edges. We represent the input graphs in the compressed sparse row (CSR) format. The neighbor list of each vertex is sorted by ascending vertex ID.

Baseline: We evaluate the efficiency of FlexMiner against state-of-the-art software GPM systems, AutoMine [58] and GraphZero [57], as well as the existing hardware GPM accelerator, Gramer [90]. We are unable to compare FlexMiner with TrieJax because their simulator has not been released and there are no absolute numbers in the paper [46]. AutoMine and GraphZero are run on a 10-core Intel i9-7900X CPU (3.30GHz, Turbo 4.3GHz, 13.75 MB LLC) with 64GB DRAM. We report execution time on CPU as an average of 3 runs. Gramer was evaluated on a Xilinx Alveo U250 card with a XCU250 FPGA chip (1.68M LUTs, 3.37M registers, and 11.8MB BRAM) and four 16GB DDR4 memories.

Table II lists the running time (seconds) of Gramer (FPGA), AutoMine (CPU) and GraphZero (CPU). Note that Gramer results are reported from their paper [90], and the `Yo` graph

	Graph	Gramer [90] 4×8T	AutoMine [58] 2×10T	GraphZero [57] 2×10T
TC	As	0.028	0.017	**0.017**
	Mi	0.11	0.05	**0.03**
	Pa	3.09	0.21	**0.21**
	Yo	13.01	0.99	**0.45**
	Lj	17.81	2.51	**0.65**
4-CL	As	0.27	0.05	**0.03**
	Mi	6.86	1.28	**0.36**
	Pa	3.74	0.24	**0.24**
	Yo	17.30	1.90	**1.20**
	Lj	30.89	23.91	**6.52**
5-CL	As	1.46	0.22	**0.08**
	Mi	270.41	49.31	**11.03**
	Pa	4.06	**0.27**	0.32
	Yo	24.27	3.16	**2.22**
	Lj	**52.89**	815.66	192.53
3-MC	As	0.11	0.03	**0.02**
	Mi	0.36	0.13	**0.07**
	Pa	4.17	0.50	**0.36**
	Yo	16.25	2.89	**1.38**
	Lj	29.68	7.82	**2.60**

TABLE II: Comparing baseline systems: Gramer (4-thread 8-PU FPGA), AutoMine (20-thread CPU) and GraphZero (20-thread CPU). Bold numbers are the fastest of each row.

used in Gramer ($|V|$=4.6M, $|E|$=44.0M) is a subset of ours ($|V|$=7.1M, $|E|$=57.1M). SL is not evaluated because it is not supported in Gramer. As shown in the table, GraphZero is almost always faster than Gramer except 5-CL on `Lj`, with an average speedup of 8.3×, regardless of all the hardware specialization that Gramer did. This speedup is mainly due to the pattern awareness in GraphZero. Note that Gramer does outperform pattern-oblivious software GPM frameworks RStream [84] and Fractal [25], as demonstrated in the Gramer paper [90]. However, the pattern-awareness in AutoMine significantly prunes the search space, and makes it orders-of-magnitude faster [58] than RStream and Fractal. GraphZero adds symmetry breaking on top of AutoMine, and is thus faster than AutoMine. Therefore we use GraphZero as our CPU baseline, and only compare FlexMiner with GraphZero in the following evaluation. Note that the FlexMiner compilation time is similar to GraphZero, which is negligible compared to the mining execution time, since $\mathcal{P}$ is much smaller than $\mathcal{G}$.

FlexMiner Simulation and Configurations: For performance evaluation, we developed a custom cycle-accurate simulator, which models the microarchitecture behavior of each module described in Fig. 8. We conservatively use 1.3GHz PE frequency (2.38GHz used in TrieJax), 32kB private cache and 4MB shared cache. When space is not available in private scratchpad, FlexMiner uses its fallback mechanism (i.e. with `SIU` and `SDU`). We *o*nly report simulation results for benchmarks completed within 2 seconds by GraphZero in Table II due to the extremely slow cycle-accurate simulation. The simulator is integrated with DRAMsim3 [53] to simulate the cycle-accurate behavior of accesses to the off-chip memory, which is simulated as 64GB of DDR4-2666 DRAM with four channels (the same as our CPU baseline). We also integrated BookSim [63] for NoC, and a standard cycle-

accurate non-inclusive cache model for L2 cache. Note that the `frontier list` memoization (described in Section V-C) is always enabled in FlexMiner for a fair comparison with GraphZero, which has this technique implemented in software.

We implement our proposed PE using Bluespec, generate Verilog, and synthesize the logic using Synopsys Design Compiler and Silvaco's 15nm Open-Cell Library [1]. We give the synthesis tool an operating voltage of 0.8V and achieve a target clock period of 0.75ns, putting our design comfortably at 1.3GHz. We estimate area numbers of the SRAMs in a PE using CACTI [2]. We use the 22nm technology node (22nm is the closest from 15nm available in CACTI). The overall area of a PE in FlexMiner is only 0.18mm^2, while the area of an Intel SkyLake CPU core (14nm) is about 15mm^2 [46].

In the following evaluation, we only focus on ASIC design of FlexMiner. We first compare FlexMiner without `c-map` to the CPU baseline, GraphZero, to show the benefit of PE specialization and massive multithreading. We then evaluate FlexMiner with different sizes of `c-map`, to show the benefit of `c-map` and find out the reasonable size to use for the `c-map`. We change the number of PEs from 1 to 64 to demonstrate a more detailed performance scaling. Lastly NoC traffic is measured to show the impact of `c-map` on reducing the memory requests, i.e., memoization reduces accesses to the edgelists of $\mathcal{G}$.

B. Performance (no-cmap) Comparison with the Baseline

Fig. 13 compares FlexMiner (without `c-map`) performance with 20-thread GraphZero. As illustrated, FlexMiner with 10-PE already outperform GraphZero for most cases, although the clock frequency is much lower than CPU. This is due to the fact that the specialized execution units (i.e., `SIU` and `SDU`) in the PE are more efficient than general purpose CPU cores for set operations. There exists other algorithms [59, 64] for set intersection, but we use the same merge-based algorithm as that is used in GraphZero to make fair comparison with the CPU baseline. Note that same as GraphZero, FlexMiner supports the memoization of the `frontier list`, which avoids recomputation of the set intersections for patterns that have reusable intermediate results (e.g., k-clique and diamond). The support for this memoization in FlexMiner makes sure that it has the same algorithmic efficiency as software.

Since the performance improvement mostly comes from accelerating the set operations, the speedup for a specific case depends on the portion of memory stalls in the total cycles. The benchmark benefits most from PE specialization when it spends most of the time on computation. In contrast, if the benchmark mostly waits for memory requests, the speedup would be marginal. For example, TC has the least computation in all applications, and `Pa` and `Yo` are relatively large datasets than `As` and `Mi`, which leads to poor cache behavior (we observe 65.9% and 36.3% L2 cache miss rates for `Pa` and `Yo` respectively) and likely more time spent on memory accesses. Both `Lj` and `Yo` are large graphs, but `Lj` contains two times more triangles than `Yo`. Therefore, TC for `Pa` and `Yo` on FlexMiner is slower than the baseline.

Since each PE has much simpler logic and smaller private cache than a general purpose core in CPU (CPU has 32kB L1D and 1MB L2), we can put more PEs within the same area. By scaling the number of PEs from 10-PE to 40-PE, FlexMiner achieves even more speedups, thanks to the embarrassing amount of parallelism in these GPM applications. On average, FlexMiner with 10-PE, 20-PE and 40-PE outperform the CPU baseline by 1.56×, 2.93×, and 5.15×, respectively.

C. Performance Impact of the `c-map`

We evaluate the performance impact of the `c-map` in Fig. 14. We use different sizes of `c-map` from 1kB to 16kB. `cmap-unlimited` is the `c-map` with unlimited size, i.e., it represents the performance upper bound of `c-map` design (which is impractical). As shown, `cmap-unlimited` achieves significant performance improvement (up to 5.3×) over `no-cmap` for `4-cycle`, with an average speedup of 3.0×. This is expected as there is no `frontier list` reuse in `4-cycle` while `c-map` is reused heavily. As a proof, the read ratios, i.e., the percentage of reads of all accesses to `c-map`, are 93%, 98% and 86% in mining `4-cycle` in `As`, `mico` and `Pa`. These high ratios are translated directly to speedups in `4-cycle`. However, for k-CL and `diamond` where memoizing the `frontier list` is already quite effective, adding `c-map` on top of it does not bring as much performance improvement as it does for `4-cycle`. For TC, there is no `frontier list` reuse, and the `c-map` reuses are fewer than those in `4-cycle`. The read-write ratios for TC on `As`, `mico` and `Pa` are 90% and 93% and 74%. This reduced reuse (compared to `4-cycle`), ends up giving a speed up between `4-cycle` and k-CL.

We also observe that `Mi` constantly obtains good speedups across different applications. This is mainly due to the fact that `Mi` is the most dense graph in Table I (with an average degree of 21), and therefore there exists abundant `c-map` reuses (also demonstrated by the high read ratio of `Mi`).

Meanwhile, we observe that for most of the benchmarks, a 4kB `c-map` already captures most of the performance benefit of `cmap-unlimited`. This is because (1) the maximum degree $\hat{d}$ of $\mathcal{G}$ is small (see Table I) compared to the graph size (e.g., although with 57M edges, `Yo` has a $\hat{d}$ of 4,017), and high-degree vertices are rare due to power-law distribution; (2) `c-map` is well utilized with our compiler heuristics in Section VI-B. In practice, we choose a `c-map` size of 8kB for the default FlexMiner design to support larger graphs and patterns. This configuration achieves average speedups of 2.28×, 4.24× and 7.29×, over the CPU baseline, respectively.

D. Large Graphs and Large Patterns

We evaluate a larger graph `Or` with TC (3-clique). Our simulation shows that 20-PE FlexMiner achieves 2.5× speedup over GraphZero-20T. Due to the simulation speed, we can not finish running larger graphs using our simulator, but FlexMiner does support larger graphs as long as they fits in memory. To support graphs larger than memory capacity, we can add graph partitioning support [5, 40, 80] in our framework.

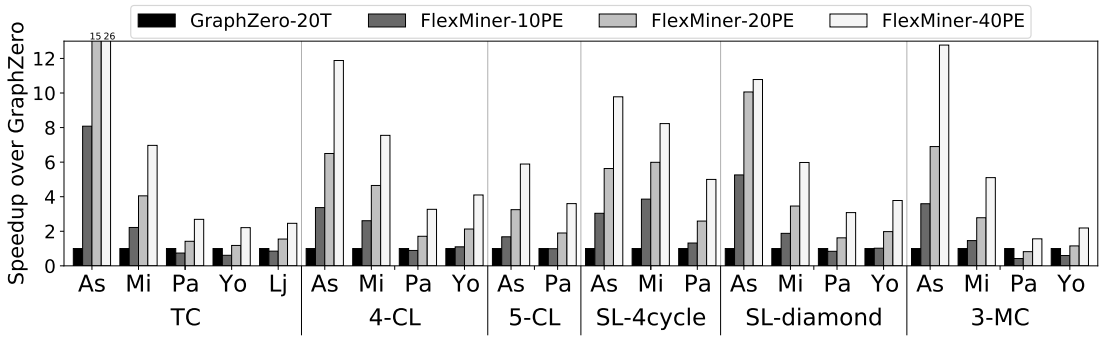

Fig. 13: FlexMiner (**without `c-map`**) performance compared with 20-thread GraphZero on CPU.

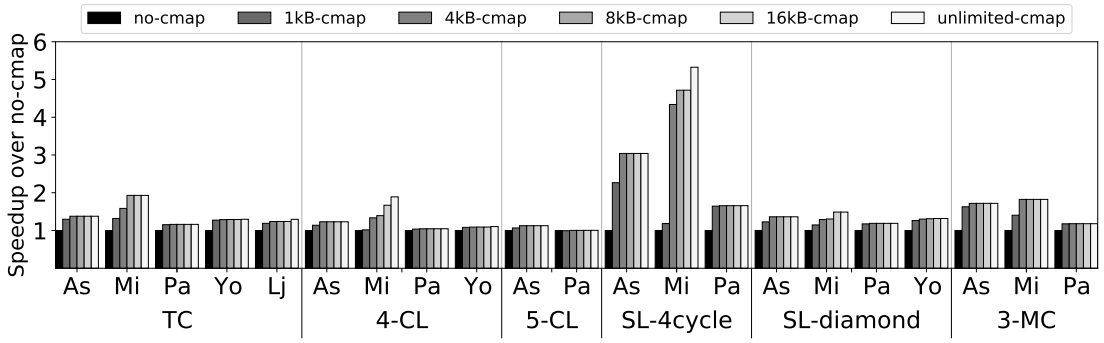

Fig. 14: FlexMiner performance using `c-map` with different sizes. All with 20-PE and normalized to FlexMiner without `c-map`.

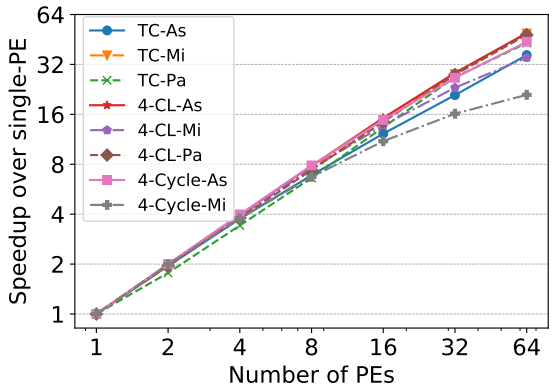

Fig. 15: FlexMiner with 8kB `c-map` performance scaling as the number of PEs increases from 1 to 64 (normalized to one-PE).

We evaluate large patterns on `Pa` using k-CL with $k \in [5, 9]$. 20-PE FlexMiner outperforms GraphZero by 1.7× to 1.9×. For a pattern of size k, `c-map` needs 32 bits for the key and $k-2$ bits for the value in each entry. We currently use 8-bit for the `c-map` value, and thus FlexMiner can fully benefit from `c-map` for patterns within 10-vertex. For patterns beyond size 10, `c-map` is partially used, i.e. for n-bit `c-map` value, FlexMiner uses `SIU/SDU` when DFS level above $n-2$ unless there is an early `c-map` overflow.

E. Scalability and NoC Traffic

Fig. 15 illustrates the FlexMiner with 8kB `c-map` performance as the number of PE increases from 1 to 64. Generally we observe linear scaling with more PEs, although different applications and datasets have different impact on the scaling factor. TC is the simplest and least irregular application among all. Therefore, TC performance scaling is almost perfect. However, `As` is the smallest dataset, and TC on `As` scales worse than the other two datasets because of much fewer tasks (i.e. parallelism). 4-CL on `As` scales better than TC on `As` likely due to more computation existed in 4-CL. On average, 64-PE FlexMiner achieves 10.60× speedup over GraphZero-20T.

In Fig. 16 we measure NoC traffic, i.e., the number of memory requests sent from the PEs to the NoC, to show the impact of `c-map` on memory subsystem. For the benchmarks which benefit from `c-map`, i.e. TC, `4-cycle` and `diamond`, the NoC traffic is significantly reduced by introducing the `c-map`. For example, 4kB `c-map` reduces nearly half of the NoC traffic for `4-cycle` on `As`. For k-CL, the NoC traffic stays the same, because the `frontier list` already cut down the same amount of memory requests for both `no-cmap` and `c-map` cases. As a result, the performance gain for 4-CL is less impressive by increasing the `c-map` size. However, we observe that `Mi` achieves close to 2× speedup in Fig. 14, which is purely achieved by `c-map` reducing set operations.

In summary, we demonstrate that FlexMiner achieves significant speedup over the CPU baseline, thanks to the PE specialization, massive multithreaing, and the support for memoization using `c-map`. More specifically, the performance speedup of 40-PE without c-map over CPU baseline is attributed to PE specialization (3.04×) and multithreading (1.76×). The adoption of `c-map` with a tiny 8kB scratchpad further improves

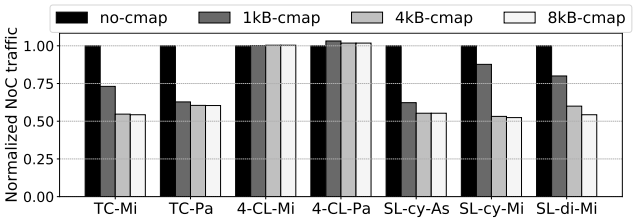

Fig. 16: Total number of NoC traffic (i.e., L2 accesses) and DRAM accesses. `cy`: `4-cycle`, `di`: `diamond`

the performance by 1.36× (up to 4.82× for some patterns), which is a reasonable tradeoff for the hardware design.

VIII. Related Work

Software GPM Systems: Arabesque [82], Fractal [25] and G-Miner [13] are distributed GPM systems, while RStream [84], Kaleido [95], Pangolin [16], AutoMine [58], GraphZero [57] and Sandslash [15] are GPM systems targeting single-machine. There are also graph querying systems like Graphflow [47, 59] and EmptyHeaded [4] which only solve single-pattern edge-induced GPM problems. Software GPM systems improve programmability, but achieves limited performance as we explained in Section III.

Hand-written Software GPM Applications: There are numerous hand-optimized GPM applications targeting various platforms, for TC [24, 32, 40, 42, 67, 68, 75, 79, 85, 91], k-CL [18, 21], k-MC [6, 69], SL [10, 11, 45, 49, 50, 51, 55, 72, 73, 77, 78, 83], and FSM [3, 27, 43, 80, 81, 86]. They employ sophisticated optimizations to improve algorithmic or/and architectural efficiency. However, they lack generality and thus require a lot more programming efforts than GPM systems. Meanwhile, our accelerator can achieve better performance than these hand-written applications on CPU or GPU.

Software Systems and Hardware Accelerators for Graph Analytics: Lots of graph processing frameworks have been proposed [33, 54, 56, 65, 94] to improve programmability for large-scale graph processing but have limited support for subgraph mining tasks. Many graph analytics accelerators [12, 14, 20, 28, 38, 62, 66, 76, 87, 89, 93, 96] have been proposed. They can improve performance over software systems when solving graph analytics problems. However, it has been demonstrated that graph analytics accelerators yield poor performance for GPM applications [46].

IX. Conclusion

GPM has been used widely in many real-world, compute-intensive applications. We present a pattern-aware hardware accelerator for GPM, which provides much higher performance than existing software and hardware accelerated solutions. Scalability of our design allows us to exploit massive parallelism in GPM by increasing the number of processing elements specifically designed for pattern-aware GPM. FlexMiner compiler automates the generation of execution plan for the hardware, which achieves pattern-awareness without any more programming effort than software GPM frameworks. It also provides hardware support for the memoization techniques, making a tradeoff between work efficiency and parallelism. Our evaluation demonstrates significant speedups over the state-of-the art software framework on CPU.

X. Acknowledgements

The research is funded by Samsung Semiconductor (GRO grants), NSF grant CCF-1725303, and NSFC grant 61802416.

References

[1] (2021) 15nm Open-Cell Library and 45nm FreePDK. https://si2.org/open-cell-library/.

[2] (2021) CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. https://github.com/HewlettPackard/cacti.

[3] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour, "Scalemine: Scalable parallel frequent subgraph mining in a single large graph," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 61:1–61:12.

[4] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *ACM Trans. Database Syst.*, vol. 42, no. 4, Oct. 2017.

[5] F. N. Afrati, D. Fotakis, and J. D. Ullman, "Enumerating subgraph instances using map-reduce," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 62–73.

[6] N. K. Ahmed, J. Neville, R. A. Rossi, and N. Duffield, "Efficient graphlet counting for large networks," in *ICDM*, 2015, pp. 1–10.

[7] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. Sahinalp, "Biomolecular network motif counting and discovery by color coding," *Bioinformatics*, vol. 24, no. 13, pp. 241–249, 2008.

[8] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, ser. IISWC '15. USA: IEEE Computer Society, 2015, p. 56–65.

[9] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2008, pp. 16–24.

[10] B. Bhattarai, H. Liu, and H. H. Huang, "CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: ACM, 2019, pp. 1447–1462.

[11] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1199–1214.

[12] N. Challapalle, S. Rampalli, L. Song, N. Chandramoorthy, K. Swaminathan, J. Sampson, Y. Chen, and V. Narayanan, "Gaas-x: Graph analytics accelerator supporting sparse data representation using crossbar architectures," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 433–445.

[13] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng, "G-miner: An efficient task-oriented graph mining system," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018.

[14] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive Cache Management for Energy-Efficient GPU Computing," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 343–355.

[15] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali, "Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining," in *Proceedings of the 35th ACM International Conference on Supercomputing*, ser. ICS '21, 2021.

[16] X. Chen, R. Dathathri, G. Gill, and K. Pingali, "Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU," *Proc. VLDB Endow.*, vol. 13, no. 8, Aug. 2020.

[17] X. Cheng, C. Dale, and J. Liu, "Dataset for statistics and social network of youtube videos," http://netsg.cs.sfu.ca/youtubedata/.

[18] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, Feb. 1985.

[19] Y.-R. Cho and A. Zhang, "Predicting protein function by frequent functional association pattern mining in protein interaction networks," *IEEE Transactions on Information Technology in Biomedicine*, vol. 14, no. 1, pp. 30–36, Jan. 2010.

[20] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 217–226.

[21] M. Danisch, O. Balalau, and M. Sozio, "Listing k-cliques in sparse real-world graphs*," in *Proceedings of the 2018 World Wide Web Conference*, ser. WWW '18. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2018, pp. 589–598.

[22] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis, "Frequent substructure-based approaches for classifying chemical compounds," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 8, pp. 1036–1050, Aug 2005.

[23] A. Deutsch, M. Fernandez, and D. Suciu, "Storing semistructured data with stored," in *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, 1999, pp. 431–442.

[24] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 393–404.

[25] V. Dias, C. H. C. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy, "Fractal: A general-purpose graph pattern mining system," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: ACM, 2019, pp. 1357–1374.

[26] C. Domshlak, S. Genaim, and R. Brafman, "Preference-based configuration of web page content," in *14th European Conference on Artificial Intelligence (ECAI 2000), Configuration Workshop, Berlin, Germany*, 2000, pp. 19–22.

[27] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "Grami: Frequent subgraph and pattern mining in a single large graph," *Proc. VLDB Endow.*, vol. 7, no. 7, pp. 517–528, Mar. 2014.

[28] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 234–248.

[29] 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.

[30] D. Feldman and Y. Shavitt, "Automatic large scale generation of internet pop level maps," in *IEEE GLOBECOM 2008-2008 IEEE Global Telecommunications Conference*. IEEE, 2008, pp. 1–6.

[31] O. Frank, "Triad count statistics," in *Annals of Discrete Mathematics*. Elsevier, 1988, vol. 38, pp. 141–149.

[32] I. Giechaskiel, G. Panagopoulos, and E. Yoneki, "PDTL: Parallel and distributed triangle listing for massive graphs," in *2015 44th International Conference on Parallel Processing*, Sep. 2015, pp. 370–379.

[33] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 17–30.

[34] M. Granovetter, "The strength of weak ties: A network theory revisited," *Sociological theory*, pp. 201–233, 1983.

[35] V. Guralnik and G. Karypis, "A scalable algorithm for clustering sequential data," in *Proceedings 2001 IEEE International Conference on Data Mining*. IEEE, 2001, pp. 179–186.

[36] D. Hales and S. Arteconi, "Motifs in evolving cooperative networks look like protein structure networks," *Networks & Heterogeneous Media*, vol. 3, no. 2, p. 239, 2008.

[37] B. H. Hall, J. A. B., and T. M., "The NBER patent citation data file: Lessons, insights and methodological tools," http://www.nber.org/patents/, 2001.

[38] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[39] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "ExTensor: An Accelerator for Sparse Tensor Algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 319–333.

[40] L. Hoang, V. Jatala, X. Chen, U. Agarwal, R. Dathathri, G. Gill, and K. Pingali, "DistTC: High performance distributed triangle counting," in *HPEC 2019 23rd IEEE High Performance Extreme Computing, Graph Challenge*, September 2019.

[41] P. W. Holland and S. Leinhardt, "Local structure in social networks," *Sociological methodology*, vol. 7, pp. 1–45, 1976.

[42] Y. Hu, H. Liu, and H. H. Huang, "Tricore: Parallel triangle counting on gpus," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2018, pp. 171–182.

[43] J. Huan, W. Wang, and J. Prins, "Efficient mining of frequent subgraphs in the presence of isomorphism," in *Third IEEE International Conference on Data Mining*, Nov 2003, pp. 549–552.

[44] K. Jamshidi, R. Mahadasa, and K. Vora, "Peregrine: A pattern-aware graph mining system," in *Proceedings of the Fifteenth EuroSys Conference*, ser. EuroSys '20, 2020.

[45] M. Jha, C. Seshadhri, and A. Pinar, "Path sampling: A fast and provable method for estimating 4-vertex subgraph counts," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2015, pp. 495–505.

[46] O. Kalinsky, B. Kimelfeld, and Y. Etsion, "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*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1217–1231.

[47] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1695–1698.

[48] H. Kashima, H. Saigo, M. Hattori, and K. Tsuda, "Graph kernels for chemoinformatics," in *Chemoinformatics and advanced machine learning perspectives: complex computational methods and collaborative techniques*. IGI Global, 2011, pp. 1–15.

[49] H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrah, "DUALSIM: Parallel subgraph enumeration in a massive graph on a single machine," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 1231–1245.

[50] K. Kim, I. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong, "Turboflux: A fast continuous subgraph matching system for streaming graph data," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 411–426.

[51] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in mapreduce," *Proc. VLDB Endow.*, vol. 8, no. 10, pp. 974–985, Jun. 2015.

[52] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[53] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: a cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, 2020.

[54] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A New Parallel Framework for Machine Learning," in *Proceedings Conf. Uncertainty in Artificial Intelligence*, ser. UAI '10, July 2010.

[55] S. Ma, Y. Cao, J. Huai, and T. Wo, "Distributed graph pattern matching," in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12. New York, NY, USA: ACM, 2012, pp. 949–958.

[56] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146.

[57] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu, "Graphzero: Breaking symmetry for efficient graph mining," 2019.

[58] D. Mawhirter and B. Wu, "Automine: Harmonizing high-level abstraction and high performance for graph mining," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: ACM, 2019, pp. 509–523.

[59] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *Proc. VLDB Endow.*, vol. 12, no. 11, p. 1692–1704, Jul. 2019.

[60] 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.

[61] T. Milenković and N. Pržulj, "Uncovering biological network function via graphlet degree signatures," *Cancer informatics*, vol. 6, pp. CIN–S680, 2008.

[62] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 1–14.

[63] Nan Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally, "A detailed and flexible cycle-accurate network-on-chip simulator," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 86–96.

[64] H. Q. Ngo, D. T. Nguyen, C. Re, and A. Rudra, "Beyond worst-case analysis for joins with minesweeper," in *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 234–245.

[65] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 456–471.

[66] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 166–177, 2016.

[67] S. Pandey, X. S. Li, A. Buluc, J. Xu, and H. Liu, "H-index: Hash-indexing for parallel triangle counting on gpus," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2019, pp. 1–7.

[68] R. Pearce, T. Steil, B. W. Priest, and G. Sanders, "One quadrillion triangles queried on one million processors," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–5.

[69] A. Pinar, C. Seshadhri, and V. Vishal, "Escape: Efficiently counting all 5-vertex subgraphs," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 1431–1440.

[70] N. Pržulj, D. G. Corneil, and I. Jurisica, "Modeling interactome: scale-free or geometric?" *Bioinformatics*, vol. 20, no. 18, pp. 3508–3515, 2004.

[71] L. Ralaivola, S. J. Swamidass, H. Saigo, and P. Baldi, "Graph kernels for chemical informatics," *Neural networks*, vol. 18, no. 8, pp. 1093–1110, 2005.

[72] X. Ren, J. Wang, W.-S. Han, and J. X. Yu, "Fast and robust distributed subgraph enumeration," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1344–1356, 2019.

[73] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, "Parallel subgraph listing in a large-scale graph," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 625–636.

[74] N. Shervashidze, S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt, "Efficient graphlet kernels for large graph comparison," in *Artificial Intelligence and Statistics*, 2009, pp. 488–495.

[75] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *2015 IEEE 31st International Conference on Data Engineering*, April 2015, pp. 149–160.

[76] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 531–543.

[77] S. Sun, Y. Che, L. Wang, and Q. Luo, "Efficient parallel subgraph enumeration on a single machine," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 232–243.

[78] S. Sun and Q. Luo, "Scaling up subgraph query processing with efficient subgraph matching," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 220–231.

[79] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 607–614.

[80] N. Talukder and M. J. Zaki, "A distributed approach for graph mining in massive networks," *Data Min. Knowl. Discov.*, vol. 30, no. 5, pp. 1024–1052, Sep. 2016.

[81] N. Talukder and M. J. Zaki, "Parallel graph mining with dynamic load balancing," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 3352–3359.

[82] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: A system for distributed graph mining," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 425–440.

[83] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, p. 31–42, Jan. 1976.

[84] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, "Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. Berkeley, CA, USA: USENIX Association, 2018, pp. 763–782.

[85] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with kokkoskernels," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7.

[86] Xifeng Yan and Jiawei Han, "gspan: graph-based substructure pattern mining," in *Proceedings of the 2002 IEEE International Conference on Data Mining*, Dec 2002, pp. 721–724.

[87] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng *et al.*, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 615–628.

[88] X. Yan, P. S. Yu, and J. Han, "Graph indexing: a frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004, pp. 335–346.

[89] Y. Yang, Z. Li, Y. Deng, Z. Liu, S. Yin, S. Wei, and L. Liu, "Graphabcd: Scaling out graph analytics with asynchronous block coordinate descent," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 419–432.

[90] P. Yao, L. Zheng, Z. Zeng, Y. Huang, C. Gui, X. Liao, H. Jin, and J. Xue, "A locality-aware energy-efficient accelerator for graph mining applications," ser. MICRO '20, 2020, pp. 1–13.

[91] A. Yaşar, S. Rajamanickam, M. Wolf, J. Berry, and Ü. V. Çatalyürek, "Fast triangle counting using cilk," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[92] L. Zhang, Y. Han, Y. Yang, M. Song, S. Yan, and Q. Tian, "Discovering discriminative graphlets for aerial image categories recognition," *IEEE Transactions on Image Processing*, vol. 22, no. 12, pp. 5071–5084, 2013.

[93] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 544–557.

[94] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "GraphIt: A High-performance Graph DSL," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 121:1–121:30, Oct. 2018.

[95] C. Zhao, Z. Zhang, P. Xu, T. Zheng, and J. Guo, "Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine," in *Proceedings of the 2020 IEEE International Conference on Data Engineering (ICDE 2020)*, ser. ICDE '20, 2020.

[96] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 712–725.