Efficient and Scalable Graph Pattern Mining on GPUs

Xuhao Chen and Arvind, *MIT CSAIL*

https://www.usenix.org/conference/osdi22/presentation/chen

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

NetApp®

Efficient and Scalable Graph Pattern Mining on GPUs

Xuhao Chen
MIT CSAIL

Arvind
MIT CSAIL

Abstract

Graph Pattern Mining (GPM) extracts higher-order information in a large graph by searching for small patterns of interest. GPM applications are computationally expensive, and thus attractive for GPU acceleration. Unfortunately, due to the complexity of GPM algorithms and parallel hardware, hand optimizing GPM applications suffers programming complexity, while existing GPM frameworks sacrifice efficiency for programmability. Moreover, little work has been done on GPU to scale GPM computation to large problem sizes.

We describe G^{2}Miner, the first GPM framework that runs efficiently on multiple GPUs. G^{2}Miner uses *pattern-aware*, *input-aware* and *architecture-aware* search strategies to achieve high efficiency on GPUs. To simplify programming, it provides a code generator that automatically generates pattern-aware CUDA code. G^{2}Miner flexibly supports both breadth-first search (BFS) and depth-first search (DFS) to maximize memory utilization and generate sufficient parallelism for GPUs. For the scalability of G^{2}Miner, we propose a customized scheduling policy to balance workload among multiple GPUs. Experiments on a V100 GPU show that G^{2}Miner is 5.4× and 7.2× faster than the two state-of-the-art single-GPU systems, Pangolin and PBE, respectively. In the multi-GPU setting, G^{2}Miner achieves linear speedups from 1 to 8 GPUs, for various patterns and data graphs. We also show that G^{2}Miner on a V100 GPU is 48.3× and 15.2× faster than the state-of-the-art CPU-based systems, Peregrine and GraphZero, on a 56-core CPU machine.

1 Introduction

Graph Pattern Mining (GPM) finds subgraphs in a given data graph which match the given pattern(s) (Fig. 1). GPM is a key building block in many domains, e.g., protein function prediction [6, 29, 83], network alignment [62, 76], spam detection [9, 34, 37], chemoinformatics [31, 57, 84], sociometric studies [36, 48], image segmentation [119]. Graph machine learning tasks can also benefit from GPM, including anomaly detection [4, 78], entity resolution [12], community detection [90], role discovery [88] and relational classification [61].

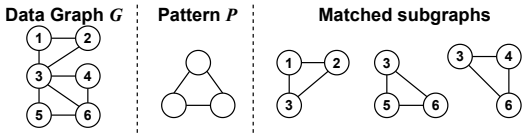


Figure 1: Graph Pattern Mining example. The pattern $\mathcal{P}$ is a triangle, and 3 triangles are found in the data graph $\mathcal{G}$.

GPM is extremely compute intensive, since it searches a space that is exponential in the pattern size. For example, Peregrine [53], a state-of-the-art GPM system on CPU, takes 9 hours to mine the `4-cycle` pattern (see Fig. 3) in the `Friendster` graph on a 56-core CPU machine. GPUs provide much higher compute throughput and memory bandwidth than CPUs, and thus are attractive for GPM acceleration.

However, implementing GPM on GPU efficiently is challenging. This is because it requires sophisticated optimizations by leveraging information in the GPU hardware architecture, the pattern(s) of interest, and the input data graph.

- *Architecture Awareness*: A GPU usually has smaller memory capacity than a CPU and requires more fine-grain data parallelism to be fully utilized. More threads, however, require more memory to accommodate intermediate data! The search order, BFS or DFS, offers a similar tradeoff between memory and parallelism and therefore, GPM on GPU requires careful orchestration of parallelism and memory usage to maximize efficiency. GPUs are also much more sensitive to thread divergence and workload imbalance [18] than CPUs. This necessitates a more sophisticated task-to-hardware mapping for GPU than that for CPU.
- *Pattern Awareness*: State-of-the-art GPM systems on CPU use *pattern aware* search plans that prune the search space using pattern information. This has been shown to be orders-of-magnitude faster than the pattern-oblivious search [53]. This pattern-aware approach has worked well for CPU, but it has not been well explored on GPU. For example, many

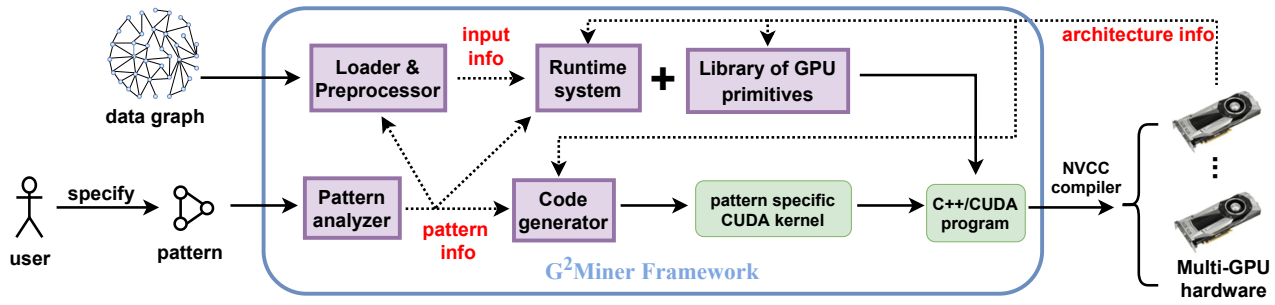


Figure 2: G^{2}Miner system overview. It contains a graph loader, a pattern analyzer, a runtime, a library of GPU primitives and a code generator.

pattern-aware pruning schemes are only effective under DFS exploration, but existing GPU-based systems use BFS, and thus loss these opportunities for pruning.

- *Input Awareness*: Dynamic memory allocation is expensive in GPU [110] and we can avoid it if we can estimate the worst-case memory usage. This can be done by using some meta information such as the maximum degree of the input graph. For labelled graphs, we can use the vertex label distribution of the input graph to get the maximum number of possible patterns, which helps save memory space. In general, input information helps make better tradeoff among work efficiency, parallelism and memory consumption.

Given this complexity, the design goal of our GPM framework on GPUs involves the following considerations:

- **Efficiency:** To achieve high efficiency on GPU, a GPM system must be highly optimized with awareness of the pattern, the input and the hardware architecture. There is no prior system, neither on CPUs nor on GPUs, that considers all three aspects together. This asks for a holistic solution that incorporates sophisticated optimizations systematically.
- **Ease of programming:** Writing efficient GPM code on GPUs is particularly difficult for domain users, who may not be parallel programming experts. Thus, hiding GPU programming complexity is essential for system usability.
- **Scalability:** The skewness in power-law graphs causes load imbalance. This problem is exacerbated for DFS-based GPM algorithms, because accesses to neighbors are multiple hops away. Hence, we need effective task scheduling and distribution policies to scale to multiple GPUs.

	General	CPU	GPU	Multi-GPU	Order	Code Gen
EmptyHeaded [2]		✓			DFS	✓
Graphflow [7,55,75]		✓			DFS	
GraphZero [73,74]		✓			DFS	✓
GraphPi [93]		✓			DFS	✓
Peregrine [53]	✓	✓			DFS	
Pangolin [25,26]	✓	✓	✓		BFS	
PBE [42,43]			✓		BFS	
G^{2}Miner	✓		✓	✓	both	✓

Table 1: Comparison of state-of-the-art GPM systems, in terms of support for generality of the programming model, hardware platforms (CPU/GPU/multi-GPU), search orders, and code generation.

We propose G^{2}Miner to overcome these challenges. Table 1 compares G^{2}Miner to the state-of-the-art systems, including those that solve only the *subgraph matching* problem, which is a subset of the GPM problem. In Table 1, subgraph matching systems include EmptyHeaded, Graphflow, GraphZero, GraphPi and PBE, while Peregrine, Pangolin and G^{2}Miner are general GPM systems. Much of the prior work focuses on CPU, and uses DFS to reduce the memory footprint. GPU-based systems (Pangolin and PBE), on the other hand, use BFS because straightforward DFS implementations on GPU suffer from thread divergence and load imbalance. This, however, limits their efficiency and/or the problem size they can solve. Additionally, G^{2}Miner simplifies GPU programming with automated CUDA code generation, while Pangolin requires users to write CUDA code manually, and PBE is not programmable at all. Last but not least, G^{2}Miner is the only system that scales to multiple GPUs.

Fig. 2 shows the overview of G^{2}Miner. It consists of a graph loader, a pattern analyzer, a runtime system, a library of CUDA primitives and a code generator. The user is only responsible for specifying the pattern(s) of interest using our API (§4). The pattern analyzer does analysis on the pattern and generates a *pattern-specific* search plan, based on which, the code generator (§5) automatically generates pattern-specific CUDA kernels for GPUs. The kernels contain invocations to the device functions defined in the GPU primitive library (§6) which includes efficiently implemented set operations. The generated kernels, the GPU primitive library, and the runtime are compiled together by the NVCC compiler to generate the executable that runs on multi-GPU.

At runtime, the graph loader reads in the data graph, extracts input information (e,g., maximum degree and label distribution) and performs pattern-specific preprocessing on the data graph. The pattern, input and architecture information is fed to the runtime (§7) which heuristically handles GPU memory allocation, data transfer, and multi-GPU scheduling.

This paper makes the following contributions:

- G^{2}Miner is the first pattern-aware, input data-graph-aware and architecture-aware framework for GPM, and it is the first GPM system that automates CUDA code generation for arbitrary patterns to simplify programming.

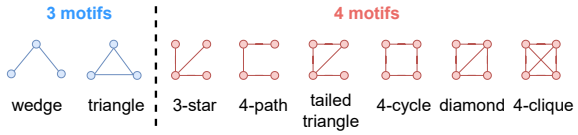


Figure 3: 3-vertex (left) and 4-vertex (right) motifs [26].

- G^{2}Miner is the first multi-GPU framework for GPM and the first GPU-based GPM framework that flexibly supports both BFS and DFS. It uses a novel task scheduling policy to balance workload among GPUs and we show G^{2}Miner performance increases linearly from 1 to 8 V100 GPUs.
- On a V100 GPU, G^{2}Miner is 5.4× faster than *Pangolin*, the only existing GPM system on GPU, and 7.2× faster than *PBE*, the state-of-the-art subgraph matching solver on GPU, thanks to the optimizations enabled in G^{2}Miner (Table 2).
- G^{2}Miner on a V100 GPU is 48.3× and 15.2× faster than state-of-the-art CPU-based GPM system *Peregrine* and subgraph matching system *GraphZero* on a 56-core CPU.

2 Background and Related Work

2.1 Graph Pattern Mining Problems

Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be an undirected graph with $\mathcal{V}$ as the vertex and $\mathcal{E}$ as the edge set. Given a vertex $v \in \mathcal{V}$, the neighbor set of v is $\mathcal{N}(v)$, the degree d_v of v is $|\mathcal{N}(v)|$ and Δ is the maximum degree in $\mathcal{G}$. A graph $G'(W,F)$ is said to be a subgraph of $\mathcal{G}$ if $W \subseteq \mathcal{V}$ and $F \subseteq \mathcal{E}$. G' is a *vertex-induced subgraph* of $\mathcal{G}$ if F contains all the edges in E whose endpoints are in W. G' is an *edge-induced subgraph* of $\mathcal{G}$ if W contains all the vertices in $\mathcal{V}$ which are the endpoints of edges in F.

Definition of GPM. Given an undirected graph $\mathcal{G}$ and a set of patterns S_p={P_1, P_2, ...} by the user, GPM finds vertex-induced or edge-induced subgraphs in $\mathcal{G}$ that are isomorphic to any $\mathcal{P}$ in S_p. If the cardinality of S_p is 1, we call it a single-pattern problem. Otherwise, it is a multi-pattern problem. The output of GPM varies in different problems, e.g., the pattern frequency (a.k.a, *support*) or listing all matched subgraphs. The definition of support also varies, e.g., the count of matches or the *domain support* [26] used in FSM. Note that *listing* requires enumerating every subgraph, but *counting* does not. Thus, counting allows more aggressive search-space pruning.

A pattern $\mathcal{P}$ is a small graph that can be defined explicitly or implicitly. An explicit definition specifies the vertices and edges of $\mathcal{P}$, whereas an implicit definition specifies the desired properties of $\mathcal{P}$. For explicit-pattern problems, the solver finds matches of $\mathcal{P}$ in S_p. For implicit-pattern problems, S_p is not known in advance. Therefore, the solver must find the patterns as well as their matches during the search.

GPM requires guarantee for *completeness*, i.e., every match of $\mathcal{P}$ in $\mathcal{G}$ should be found, and often *uniqueness*, i.e., every distinct match should be reported only once [101]. To avoid confusion, we call a vertex in the pattern $\mathcal{P}$ as a *pattern vertex* and denote it as u_i, and a vertex in the data graph $\mathcal{G}$ as a *data vertex* and denote it as v_i. Our work covers the following GPM problems from the literature [26, 33, 101]:

- *Triangle counting* (TC): It counts the number of triangles (Fig. 1), i.e., 3-cliques, in $\mathcal{G}$.
- *k-clique listing* (k-CL): It lists all the k-cliques in $\mathcal{G}$ ($k \geq 3$). A k-clique is a k-vertex graph whose every pair of vertices are connected by an edge.
- *Subgraph listing* (SL). It lists all edge-induced subgraphs of $\mathcal{G}$ that are isomorphic to a pattern $\mathcal{P}$.
- *k-motif counting* (k-MC): It counts the number of occurrences of all possible k-vertex patterns. Each pattern is called a *motif* [11, 77]. Fig. 3 shows all 3-motifs and 4-motifs. This is also an example of a multi-pattern problem because we have to find all the subgraphs that are isomorphic to *any* pattern in a given set of patterns.
- *k-frequent subgraph mining* (k-FSM): Given k and a threshold σ_{min}, this problem considers all patterns with fewer than k edges and lists a pattern $\mathcal{P}$ if the support σ of $\mathcal{P}$ is greater than σ_{min}. This is called a *frequent* pattern. If k is not specified, it is set to ∞, meaning that it is necessary to consider all possible values of k. In k-FSM, vertices in $\mathcal{G}$ have application-specific labels.

For TC and k-CL, vertex-induced and edge-induced subgraphs are the same. SL and FSM find edge-induced subgraphs, while k-MC looks for vertex-induced subgraphs. All problems seek to find explicit pattern(s) except FSM which finds implicit patterns. k-MC and FSM are multi-pattern problems, while the others are single-pattern problems.

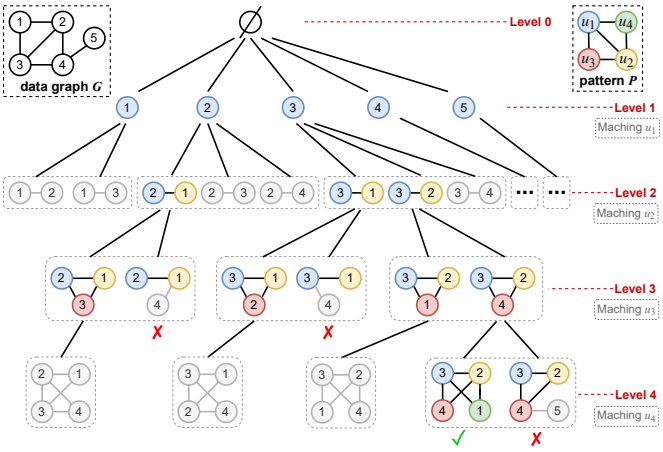


Figure 4: A search tree using vertex extension. Vertex colors (not vertex labels) show the matching between data vertices and pattern vertices. The matching order is $\{u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4\}$. The symmetry order is $\{v_a > v_b, v_c > v_d\}$. Subgraphs in grey are ruled out by symmetry breaking. × shows the unnecessary extensions that are pruned by the matching order. ✓ shows the matched subgraph.

Algorithm 1 Pseudo code for finding `diamond` in DFS order

```
1: for each vertex v1 ∈ V in parallel do                    ▷ match v1 to u1
2:     for each vertex v2 ∈ N(v1) do                        ▷ match v2 to u2
3:         if v2 ≥ v1 then break;                           ▷ symmetry breaking
4:         W ← N(v1) ∩ N(v2);                               ▷ set intersection: buffered in W
5:         for each vertex v3 ∈ W do                        ▷ match v3 to u3
6:             for each vertex v4 ∈ W do                    ▷ match v4 to u4; W is reused
7:                 if v4 ≥ v3 then break;                   ▷ symmetry breaking
8:                 else count ++;                           ▷ do the counting
```

Algorithm 2 Pseudo code for finding Pattern $\mathcal{P}$ in BFS order

```
1: for each level i ∈ [1, P.size] do          ▷ level i from 1 to the pattern size
2:     for each subgraph sg ∈ SL_i in parallel do          ▷ SL_i: subgraph list
3:         for each vertex u ∈ sg do
4:             for each vertex v ∈ N(u) do
5:                 sg' ← sg ∪ v                ▷ vertex extension: add vertex v
6:                 if sg' satisfy P.constraints(i) then
7:                     if i = P.size then count ++;      ▷ leaf: a match found
8:                     else SL_{i+1}.insert(sg')         ▷ go to the next level
```

2.2 Pattern-Aware GPM Algorithms

A GPM problem is a search problem, whose search space is a *subgraph tree* [25, 27] (Fig. 4). Each node in the tree is a subgraph of the data graph $\mathcal{G}$. Subgraphs in level l of the tree have l vertices. The root of the tree (level 0) is an empty subgraph, while the leaves of the tree are potential candidates of matches. A GPM problem can be solved by building this search tree, and checking each leaf if it is isomorphic to the pattern $\mathcal{P}$ using the typical *graph isomorphism test*.

The search tree is built by *vertex extension*: subgraph S_1=(W_1, E_1) can be extended by a single vertex $v \notin W_1$ to obtain subgraph S_2=(W_2, E_2), if v is connected to some vertex in W_1 (i.e., v is in the *neighborhood* of subgraph S_1). When two subgraphs are related in this way, we say that S_2 is a child of S_1. Formally, this can be expressed as W_2=$W_1 \cup \{v\}$ where $v \notin W_1$ and there is an edge $(v,u) \in \mathcal{E}$ for some $u \in W_1$. Similarly, *edge extension* extends a subgraph S_1 with a single edge (u,v), with at least one of the endpoints of the edge is in S_1.

The efficiency of a GPM algorithm depends heavily on how much we can prue the search tree. State-of-the-art GPM frameworks [53, 74] use *pattern-aware* search plans that leverage the properties of the pattern to prune the tree. A pattern-aware search plan consists of a *matching order* and *symmetry order*.

Matching order is a total order that defines how the data vertices are matched to pattern vertices. This order is used to eliminate irrelevant subgraphs on-the-fly. As shown in Fig. 4, to find the `diamond` pattern, we use a matching order among pattern vertices: $\{u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4\}$, meaning that each vertex v_1 added at level 1 is matched to u_1; each vertex v_2 added at level 2 are matched to u_2, and so on. To search for matching candidates, there are connectivity constraints for the data vertices. For example, in `diamond`, since u_3 is connected to both u_1 and u_2, candidate vertices of v_3 must be found in the intersection of v_1 and v_2's neighborhoods, i.e., $v_3 \in \mathcal{N}(v_1) \cap \mathcal{N}(v_2)$. The same constraint should also be applied to v_4. For a given pattern $\mathcal{P}$, there exist multiple valid matching orders. To choose the best performing matching order, prior works [7, 21, 22, 53, 59, 73, 74, 93] have proposed various cost models to predict the performance of matching orders, and choose the one with the highest expected performance.

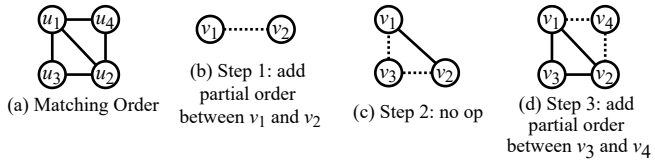
Symmetry order is a partial order enforced among data vertices for *symmetry breaking*, which removes redundant subgraph enumerations (a.k.a *automorphism* [26]), and thus guarantees that any match of $\mathcal{P}$ in $\mathcal{G}$ is found only once. For example, for `diamond`, we enforce that vertices added at level 1 must have larger ids than vertices added at level 2, i.e., $v_1 > v_2$. Thus, in level 2 of the tree in Fig. 4, the subgraph $\{2, 1\}$ is selected to be extended further, but subgraph $\{1, 2\}$ is pruned. Similarly we add a constraint that $v_3 > v_4$. So the symmetry order for `diamond` is $\{v_1 > v_2, v_3 > v_4\}$.

2.3 DFS vs. BFS

Any search order (e.g., BFS, DFS) can be used to explore the search tree, but different search orders come with different work efficiency, parallelism and memory consumption.

Algorithm 1 shows a DFS algorithm to mine the pattern `diamond`. It contains 4 nested `for` loops (Line 1, 2, 5, 6). Each loop corresponds to a data vertex (v_1, v_2, v_3, v_4) that is mapped to a pattern vertex (u_1, u_2, u_3, u_4) in Fig. 5 (a). A buffer W in Line 4 holds intermediate data that is reused multiple times, which avoids redundant computation and thus improves *work efficiency*. The memory footprint contains only four vertices $(v_i, i = 1, 2, 3, 4)$ and W in Line 4 whose size is bounded by Δ. In DFS, every parallel *task* does a DFS walk on the entire subtree rooted at v_1 (Line 1). This is known as *vertex parallelism*. The amount of parallelism is $|\mathcal{V}|$. Another way to parallelize it is *edge parallelism*, in which every task contains the subtree rooted at each edge (say, if we make Line 2 in parallel). The amount of parallelism then is $|\mathcal{E}|$.

The BFS algorithm in Algorithm 2 explores the tree level by level. In each level, it maintains a *subgraph list* that is shared globally among all threads. Each thread takes a subgraph from the subgraph list (Line 2), and extends it to generate its child subgraphs (Line 5). The child subgraphs are inserted into the next-level subgraph list (Line 8). In BFS, each parallel task is a subgraph in the subgraph list of the current level. Since the size of the subgraph list increases exponentially level by level, the amount of parallelism increases rapidly.



(a) Matching Order (b) Step 1: add partial order between v_1 and v_2 (c) Step 2: no op (d) Step 3: add partial order between v_3 and v_4

Figure 5: Generating symmetry order for `diamond` [27].

Although it provides more parallelism than DFS, BFS needs much more memory to accommodate the subgraph list. For example, the BFS-based GPM system Pangolin [26] needs more than 40GB memory to mine the 5-`clique` pattern in a moderate size graph `livejournal`, making it impossible to run in most of off-the-shelf GPUs.

2.4 GPM Systems and Applications

Many existing GPM systems [20, 26, 101, 107, 120] use the BFS order. As they do level-by-level subgraph extension, they generate massive intermediate data and thus are limited to small graphs and patterns. Recently, a few DFS-based GPM systems [25, 33, 53, 74] have been proposed to support larger datasets, but they are all CPU-based. Among all, Pangolin is *the only existing GPM system that supports GPU*. However, limited by the BFS order, Pangolin can only handle small graphs, and it lacks pattern and input awareness.

There also exist subgraph matching systems on CPU [2, 55, 73, 93, 104] and GPU [42, 43, 117]. But they only support a subset of GPM problems and are usually not programmable.

Numerous hand-optimized GPM applications have been developed, including triangle counting [32, 39, 47, 50, 80, 81, 94, 98, 109, 111, 116] , k-clique listing [30] and counting [5, 19, 52, 92], motif counting [3, 67, 70, 82, 89, 95], subgraph listing/matching [13, 14, 17, 46, 54, 59, 60, 65, 68, 72, 85, 86, 91, 96, 97, 103, 105, 108], and FSM [1, 35, 58, 99, 100, 102, 106, 114]. All of them are manually optimized with significant programming effort to achieve high efficiency, which is quite a lot of burden for the domain programmers.

3 Challenges of Efficient GPM on GPU

3.1 GPM vs. Graph Analytics

Similar to graph analytics, GPM algorithms are *irregular* [18] because the control flow and memory accesses are input-data dependent and thus, cannot be predicted statically. This irregularity causes random memory accesses and load imbalance, making it difficult to be efficiently parallelized. Unlike graph analytics that only accesses 1-hop neighbors, GPM requires accessing multi-hop neighbors, which exacerbates the irregularity problem. For example, load imbalance is much worse for DFS-based GPM than graph analytics because each parallel *task* (a DFS walk on the entire sub-tree) is more coarse-grain in GPM. In addition, GPM generates intermediate data during the search (buffer W in DFS or subgraph list in BFS), which consumes extra memory than graph analytics.

3.2 GPM on GPU vs. GPM on CPU

Since the tasks are independent of each other, they are fairly easy to parallelize on CPU, as shown in Algorithm 1 Line 1. But it is not as straightforward on GPU due to GPU's massively parallel model and limited memory capacity.

A GPU often consists of multiple *streaming multiprocessors* (SM). Each SM accommodates multiple vector units. This hardware organization results in a hierarchical parallel model: each CUDA *kernel* includes groups of threads called *cooperative thread arrays* (CTAs) or *thread blocks*. Within each CTA, subgroups of threads called *warps* are executed simultaneously. Thus GPUs, to be fully utilized, require much more hierarchical parallelism than CPUs.

GPUs generally have less memory than CPUs, while BFS-based GPM algorithms consume memory exponential in the pattern size. Using DFS can reduce memory consumption, and also improve work efficiency. Hence, state-of-the-art CPU-targeted GPM frameworks [25, 53, 73, 74, 93] all adopt DFS. However, naively porting the DFS-based CPU algorithms to GPU is not efficient because of the following reasons:

(1) Branch Divergence. In Algorithm 1, each thread takes a vertex v_1 from $\mathcal{V}$ and starts DFS walk rooted by v_1. Since different vertices have different neighborhoods, the threads in a warp may take different paths at the branches, leading to inefficiency on GPU [87]. Branch divergence is much more severe for DFS than BFS due to the multiple nested loops for DFS backtracking that access multi-hop neighborhoods.

(2) Memory Divergence. DFS walk also makes memory accesses more irregular. This causes memory divergence in GPU, i.e., threads in a warp access non-consecutive memory locations. In this case, each load instruction generates multiple (up to the warp size, i.e., 32) memory requests to the memory subsystem, which wastes memory bandwidth, congests on-chip data path [24], and thus results in poor GPU performance.

(3) Load Imbalance. Variance of neighborhood sizes in power-law graphs causes load imbalance. In CPU it is less significant because there are limited number of cores/threads and each core is very powerful. However, GPUs have thousands of lightweight cores and more than ten times the number of active threads. If unbalanced, it would be much more costly since the slowest thread is running on a low-frequency core and thousands of cores are waiting. Load imbalance is also less concerned for BFS, since it does level-by-level extension and at each level the tasks are lightweight, i.e., fine-grained.

Therefore, existing GPU-based GPM systems [26] and subgraph matching systems [42, 43] all use BFS order. This severely limits the graph sizes that they can handle. PBE [42] partitions the data graph to support large graphs, but partitioning introduces cross-partition communication. Note that using *beam search* [71] or bounded DFS does not fully resolve these issues, but loses the benefit of work efficiency of using DFS.

4 G^{2}Miner System Overview and Interface

We propose G^{2}Miner (Fig. 2) to address the challenges in §3. It hides away GPU programming complexity, and takes into

Listing 1: k-Clique Listing (k-CL) user code in G^{2}Miner

```
1 Graph G = loadDataGraph("graph.csr");
2 Pattern p = generateClique(k);
3 list(G, p); // count(G, p) for counting
```

Listing 2: Subgraph Listing (SL) user code in G^{2}Miner

```
4 Pattern p("pattern.el", EdgeInduced);
5 list(G, p);
```

Listing 3: k-Motif Counting (k-MC) user code in G^{2}Miner

```
6 Set<Pattern> patterns = generateAll(k);
7 Map<Pattern,int> result = count(G, patterns);
```

Listing 4: Frequent Subgraph Mining (k-FSM) user code in G^{2}Miner

```
8  Void updateSupport(Subgraph s) {
9    map(s.getPattern(), s.getDomain());
10 }
11 bool patternFilter(Pattern p) {
12   return p.getDomainSupport() >= threshold;
13 }
14 Set<Pattern> patterns = generateAll(k,
15                  EdgeInduced, patternFilter);
16 list(G, patterns, PATTERN_ONLY);
```

account the properties of the pattern, input data graph and hardware architecture to achieve high efficiency on GPU. We first describe how to program in G^{2}Miner in §4.1, and then introduce the system interface for extracting information out of the input, pattern and architecture (§4.2). Lastly we give an overview of the optimizations in §4.3.

4.1 Making Programming Easy

G^{2}Miner provides the same API as state-of-the-art CPU-based systems, e.g., Peregrine and Sandslash, making it friendly to users of CPU frameworks. As shown in Listing 1, to program a k-CL solver in G^{2}Miner, the user specifies the pattern using an utility function `generateClique()` (Line 2), and then call `list()` to do listing or `count()` to do counting. If `count()` is used, it allows the system enable counting-only optimizations (details in §5). To list an arbitrary pattern $\mathcal{P}$ (Listing 2), the user can specify $\mathcal{P}$ using its edgelist (`pattern.el` at Line 4). By default G^{2}Miner finds vertex-induced subgraphs. Since SL requires listing edge-induced subgraphs by definition, the user needs to specify it (`EdgeInduced` at Line 4).

For multi-pattern problems, the user is interested in a set of patterns instead of just one. For k-MC in Listing 3, the patterns can be generated by calling an utility function `generateAll()` (Line 6) or parsing the patterns' edgelists.

Programmability is particularly important for implicit-pattern problems. The user must implement API functions to specify the patterns. For example, for k-FSM in Listing 4, the user chooses to use `domain support` by implementing `updateSupport` (Line 8). To specify the properties that differentiate the interesting patterns with irrelevant patterns, the user must define `patternFilter` (Line 11). As FSM asks for only listing the patterns, we can specify a `PATTERN_ONLY` keyword in `list` to avoid listing the subgraphs (Line 16). If the user wants to customize the output, one can define a `output()` function and pass it to `list`, instead of using `PATTERN_ONLY`. This function defines custom operations on each subgraph of interest, which can also be used to do *early termination* [53] by checking a user-defined condition.

4.2 System Interface

The pattern specified by user API is fed to a *pattern analyzer* to extract useful pattern information. Meanwhile, the GPU hardware information is taken by G^{2}Miner to enable optimizations in the runtime, code generator and GPU primitives. At runtime, the data graph is loaded by a *graph loader* which collects input information and also performs preprocessing.

Pattern Analyzer. The *pattern analyzer* generates: (1) a search plan with a matching order and a symmetry order, which is used by the code generator; (2) reuse opportunities using buffers (e.g., W in Algorithm 1), used by the code generator and the runtime; (3) other important properties of the pattern, e.g., whether the pattern is a clique or `hub-pattern` (§5.4 (2)), used by the runtime and code generator.

The pattern analyzer enumerates all the possible matching orders of $\mathcal{P}$, and uses a cost model to pick the best one. We use the same cost model as GraphZero [73] for fair comparison, but any cost model can be employed by G^{2}Miner. We also use the algorithm in GraphZero to generate a symmetry order: it takes the generated matching order $\mathcal{MO}$ and builds a subgraph incrementally in the order specified by $\mathcal{MO}$. At each step it detects symmetric vertex pairs and adds orders accordingly. For example, for `diamond`, the matching order in Fig. 5 (a) results in the three steps shown in (b), (c) and (d), during which we add partial order $v_2 < v_1$ and $v_4 < v_3$.

Graph Loader and Preprocessor. The data graph $\mathcal{G}$ is loaded by the *graph loader* into the memory in the compressed sparse row (CSR) format. As $\mathcal{G}$ is being loaded, useful input information of the data graph is extracted, e.g., $|\mathcal{V}|$, $|\mathcal{E}|$ and Δ of $\mathcal{G}$. In addition, if the graph is labelled, the vertex frequency of each label is computed (see usage for FSM in §7.2). After $\mathcal{G}$ is loaded into memory, some preprocessing is performed on $\mathcal{G}$. First, the neighbor list of each vertex is sorted by ascending order of vertex IDs, so that we can apply early exit when we search the list with an upper bound (i.e., symmetry breaking). Second, if a pattern of `clique` is detected, G^{2}Miner enables a typical optimization called *orien-*

Optimizations		Effect					Used in Pangolin?	Used in hand written apps?	Conditions to apply
		mitigate divergence	load balance	mem. saving	algorithm pruning	extra GPU efficiency			
Category-(1): Known	A: Data graph preprocessing (edge orientation) §4.2			✓	✓		✓	✓	cliques
Category-(2): Known, but not enabled in prior GPM systems	B: Data graph partitioning §7.2 (1)			✓			×	TC only	hub patterns, graph size, GPU memory size
	C: Two-level parallelism §5.1	✓	✓				×	TC only	always enabled on GPU
	D: Counting-only pruning §5.4 (1)				✓		×	CPU only	automatic pattern decomposition [82]
	E: Local graph search §5.4 (2)				✓		×	CL only	hub patterns & Δ <1024
	F: Flexible data format §6.2	✓					×	CC only	
	G: Multi-gpu scheduling §7.1		✓				×	MC only	always used on multi-GPU
Category-(3): Novel for GPM	H: SIMD-aware primitives §6.1					✓	×	×	hardware support for warp level primitives
	I: Multi-pattern fission §5.3					✓	×	×	explicit multi-pattern & kernel occupancy by NVCC
	J: Edgelist reduction §7.2 (2)			✓			×	×	if $v_0 > v_1$ in symmetry order
	K: Adaptive buffering §7.2 (3)			✓			×	×	buffer W usage in matching order & GPU memory size
	M: Hybrid order on GPU §5.2			✓			×	×	implicit, intermediate data unbounded, user-specified
	N: memory reduction using label frequency §7.2 (4)			✓			×	×	implicit, vertex label frequency, user-specified

Table 2: Optimizations in G2Miner. Among them, optimizations A, B, D, E, F, I, J, K, M, N are pattern-aware; optimizations B, C, G, H, I, K, M are architecture-aware; and optimizations B, E, F, K, N are input-aware. Pattern-aware optimizations are applied based on the pattern analysis, while input-aware and architecture-aware optimizations are enabled according to the input and architecture information, respectively. TC: triangle counting. CL/CC: clique listing/counting. MC: motif counting.

tation [26] . It gives every edge a direction in the undirected data graph $\mathcal{G}$, which in turn converts $\mathcal{G}$ into a directed graph. This halves the edge count in $\mathcal{G}$, significantly reduces Δ, and completely eliminates on-the-fly checking. Third, our preprocessor also supports sorting (e.g., by degree) and renaming the vertices in $\mathcal{G}$ to improve load balance [53, 73]. Note that all these preprocessing operations need to be done only once.

4.3 Overview of Optimizations

Table 2 lists all the optimizations enabled in G^{2}Miner. We classify them into three categories. Optimizations in Category-(1) are those exist in prior GPM systems. Optimizations in Category-(2) do not exist in prior GPM systems (e.g., Pangolin) but have been used in some hand-written GPM applications. For example, optimization `D: data graph partitioning` has only been used for triangle counting, while in G^{2}Miner we generalize it for all the clique patterns. These optimizations are missing in prior GPM systems because prior systems are oblivious to the required pattern, input or architecture information. Optimizations in Category-(3) are novel as they have never been used for GPM, though some of them are known for GPU computing in general.

As shown in column 3 to 7 of Table 2, these optimizations have different kinds of effect on GPM applications: (1) mitigating thread divergence; (2) improving load balancing; (3) reducing memory consumption; (4) pruning search space; and (5) improving efficiency based on GPU hardware features.

The last column of Table 2 shows the conditions for each optimization to be applied. All the optimizations in Table 2 are automated in G^{2}Miner based on detecting the conditions, except for M and N (the last two rows). M and N are particularly used for implicit-pattern problems like FSM, for which the system cannot infer the conditions automatically. Thus, M and N are user-activated by specifying a flag.

Next, we describe these optimizations in detail, in the three major components of G^{2}Miner: the code generator (§5), the device function library (§6) and the runtime scheduler (§7).

5 Pattern-specific GPU Code Generation

G^{2}Miner includes a *pattern-aware* code generator that automatically generate CUDA code specific to the pattern. Prior work [73, 74] has explored how to generate pattern-specific CPU code based on the matching order and symmetry order, but code generation is more challenging for GPU.

Generating pattern-specific CPU code is relatively straightforward. For example, to generate Algorithm 1 for `diamond`, the matching order in Fig. 5 (a) is used to generate the 4 nested `for` loops, and the symmetry order is then used to insert breaks at Line 3 and 7. Whenever a set operation is needed, a function call to the set operation primitive (imple-

mented in a library) is inserted (Line 4). Since v_3 and v_4 are both from $\mathcal{N}(v_1) \cap \mathcal{N}(v_2)$, a buffer W is created for data reuse. Finally, *task parallelism* is used to parallelize the program, i.e., each thread processes one task at a time (Line 1).

However, generating efficient GPU code is more challenging, because (1) DFS-based GPM suffers from the thread divergence and load imbalance issues (§5.1); (2) hybrid search orders are needed in some cases (§5.2); and (3) extra support is needed for multi-pattern problems (§5.3) and advanced pruning schemes (§5.4).

5.1 Parallel Strategies for DFS on GPU

To maximize GPU efficiency for the DFS algorithm, we employ a *two-level parallelism* strategy in G^{2}Miner to exploit both inter-warp task parallelism and intra-warp *data parallelism*. This is motivated by our key observation that in GPM algorithms *most of execution time is spent on set operations*. For example, when we executed Peregrine on a multicore CPU, set operations for each benchmark took 75% to 92% of the total execution time. This motivated us to parallelize set operations by exploiting the data parallelism within each warp. It alleviates divergence and also provides more parallelism to fully utilize GPUs. To reduce load imbalance and further increase parallelism, we use edge parallelism for GPU instead of the vertex parallelism used for CPU.

(1) Reduce divergence with warp-centric parallelism. We could map each task to a thread, a warp or a CTA in a GPU. As DFS has much more coarse-grained tasks than BFS, mapping a task to a thread would be highly divergent and unbalanced for GPUs. However, if we map a task to a CTA, all (e.g., 256) threads in the CTA will be used to process the same set operations. If the two input neighbor-lists of a set operation are small, many threads in the CTA will be idle, leading to low utilization. Moreover, all threads in the CTA will do the same DFS walk, which is a lot of redundant computation.

In G^{2}Miner we use *warp-centric data parallelism*. Each task is assigned to a warp. All threads in a warp synchronously perform the same DFS walk of the task. During the DFS walk, whenever a set operation is encountered, all threads in the warp work cooperatively to compute the set operation in parallel. It has several benefits. First it achieves higher throughput than CPU since set operations are parallelized. Second, it alleviates thread divergence within each warp as all threads in a warp are progressing synchronously. Third, it causes less redundancy than using CTA. Our evaluation shows it is on average 2× faster than CTA-centric parallelism.

(2) Reduce task granularity for load balance. GPM systems on CPU use vertex parallelism [25, 53, 73, 74], i.e., each task is a DFS walk rootedin a vertex, as shown in Fig. 6 (a). This can already provide enough parallelism for CPU, needs no auxiliary data, and potentially enjoys data reuse within the sub-tree. But the coarse-grain tasks lead to load imbalance which can not be well tolerated by GPUs. To reduce task granularity, we use edge-parallelism, i.e., each task explores the subtree rooted by an edge. As shown in Fig. 6 (b), apparently more work is required to search the subtree below a vertex on average compared to searching the subtree below an edge. In addition to better load balance, edge parallelism can provide more parallelism ($|\mathcal{E}| > |\mathcal{V}|$) for GPU than vertex parallelism.

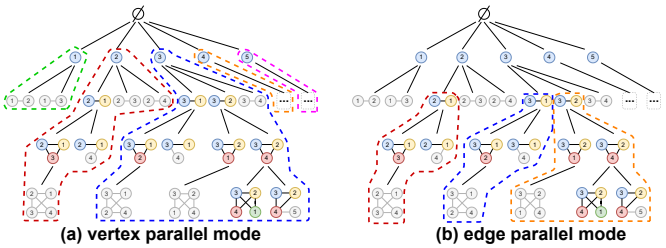


Figure 6: (a) vertex-parallel vs. (b) edge-parallel execution. Each dashed circle is a parallel task. A task is mapped to one thread on CPU, but in G^{2}Miner it is mapped to one warp on GPU.

- By default, our code generator generates edge-parallel kernels. Our evaluation shows they are mostly (1.5× on average) faster than vertex parallel ones. But some GPM algorithms must use vertex parallelism. For example, the 3-MC algorithm in [25] can only be done in vertex parallelism. G^{2}Miner supports both vertex and edge parallelism. The user can set a compiler flag to use vertex parallelism, in which case Ω is not generated to save memory.

Discussion. *Two-level parallelism* has been only used for triangle counting [50], and it is challenging to extend it for all GPM problems. First, triangle counting does subgraph extension only once, which needs no DFS traversal. Thus, G^{2}Miner is the first to support DFS for GPM on GPU. Second, naive GPU implementations for complex patterns can easily run out of memory for intermediate data. This is not a concern for triangle counting. Third, during the DFS traversal, it requires extension to support high-performance generic set operations and multi-pattern, which triangle counting does not require. In the following, we show that these challenges can be resolved by applying optimizations H, I, J, K, M, N in Table 2.

5.2 Support for Hybrid Search Orders

With the two-level parallelism in §5.1, for many GPM problems, DFS is faster than BFS in G^{2}Miner. However, this is not the case for problems like FSM. FSM computes the `domain support` and thus requires aggregating all the subgraphs for each pattern to compute its support. In the DFS-based FSM algorithm [99, 114], each task is a single-edge pattern (instead of subgraph) and the entire subtree of that pattern. This is *pattern-parallel*, instead of vertex-parallel or edge-parallel. Since the number of patterns is much smaller than the number of vertices or edges, the parallelism in FSM is not sufficient for GPU. Moreover, the task granularity in pattern-parallelism is much larger than that in vertex- or edge-parallelism, making

Algorithm 3 Pseudo code for counting `diamond`

```
1: for each vertex v1 ∈ V in parallel do                ▷ match v1 to u1
2:     for each vertex v2 ∈ N(v1) do                    ▷ match v2 to u2
3:         if v2 ≥ v1 then break;                       ▷ symmetry breaking
4:         n = |N(v1) ∩ N(v2)|;                         ▷ # triangles incident to (v1, v2)
5:         count += n*(n-1)/2                           ▷ choose 2 from n to form a diamond
```

the problem even more unbalanced.

In G^{2}Miner we use a hybrid of BFS and DFS, or *bounded BFS* search for problems that use `domain support` (e.g., FSM). At the single-edge level (i.e., level-2), we start with BFS search to aggregate edges by their patterns in parallel, which provides abundant parallelism. As the search goes deeper, the number of subgraphs increases exponentially. To fit the intermediate data in memory, we divide the subgraphs into blocks. Each block has a size that can resides in GPU memory, but also contains enough amount of subgraphs that can fully utilize the GPU. Once the current block is processed, it moves to the next block. Using this bounded BFS search, G^{2}Miner can support larger graphs than Pangolin.

5.3 Support for Multi-pattern Problems

Multiple patterns may have a common sub-pattern, which can be shared if they are searched in the same CUDA kernel. On the other hand, mining multiple patterns simultaneously would need a significant amount of intermediate resources, e.g., registers, which results in low hardware utilization (occupancy) on GPU.

Instead of generating a single gigantic kernel for all patterns, we employ *kernel fission* to reduce register pressure. Given multiple patterns, we leverage pattern analysis to find which patterns share the same sub-pattern, so that they should be merged into the same kernel to enjoy sharing. For those patterns do not share the same sub-patterns, we generate different kernels for them, so that each kernel is lightweight enough to avoid high register pressure. For example, in 4-motifs (Fig. 3), `tailed-triangle`, `diamond` and `4-clique` share the same sub-pattern `triangle`. So we generate a single CUDA kernel for the three patterns, in which they share the same workflow that enumerates triangles. However, for the other patterns, since there is no sharing opportunity, we generate one kernel for each. These separated kernels use fewer registers than a combined kernel, so that each SM in GPU can accommodate more co-running warps to maximize utilization. This improves performance by 15% for mining 4-motifs.

5.4 Support for Advanced Pruning Schemes

(1) Counting-only Pruning. If the user is interested in *counting* instead of *listing* subgraphs, there may exist an advanced pruning opportunity to further reduce the search space. For example, to count edge-induced `diamond` (Algorithm 3), because a `diamond` consists of two triangles, we first compute the triangle count n for each edge (v_1, v_2) using set intersection (Line 4), and then use the formula $\binom{n}{2} = n \times (n-1)/2$ to get the `diamond` count (Line 5). Note that this pruning opportunity is pattern specific and is not always available. For example, there is no such opportunity for `4-cycle`. Our pattern analyzer detects the opportunities by using automatic pattern decomposition [21,82], and based on the detection, our code generator can accordingly generate the CUDA kernel.

(2) Local Graph Search (LGS). This is a pruning scheme used for `hub-patterns`. A hub-pattern contains at least one `hub` vertex that is connected to all other vertices. For example, any vertex in a `clique` is a hub vertex. The key idea of LGS is, instead of searching a massive data graph $\mathcal{G}$, we can construct a small local graph for each vertex in $\mathcal{G}$ and search in the local graphs. For a hub pattern with a hub vertex u_1, we match the first data vertex v_1 to u_1, and the entire sub-tree rooted by v_1 is confined within v_1's 1-hop neighborhood. Fig. 7 shows an example of constructing a local graph. Search in the local graph is faster because the vertex degrees in the local graph are smaller than those in the global data graph. When the pattern analyzer detects a hub-pattern, the code generator inserts a call to construct local graphs, and generates code to search in the local graphs, instead of the original data graph.

- Previously, LGS has only been used for clique patterns [30], while G^{2}Miner generalizes and automates it for all hub patterns. Moreover, unlike CPUs, naive implementation on GPUs is not beneficial. We combine LGS with the `bitmap` format (see §6.2) to achieve significant speedups.
- *Input Awareness*. LGS is not always beneficial [25]. The key indicator is the maximum degree Δ of the data graph. For example, if Δ is too large, it is not beneficial due to high overhead of local graph construction. Therefore, we generate CUDA kernels for both cases: LGS enabled and disabled. The runtime system checks if Δ is above a threshold and decides accordingly which kernel to use. LGS brings us 1.2 $\sim$ 3.7$\times$ speedup on GPU for various data graphs.

6 Device Primitives for Set Operations

As G^{2}Miner assigns each task to a warp, whenever there is a set operation, all the thread in a warp work cooperatively to compute it. For example, in Algorithm 1, there is a set intersection at Line 4. In G^{2}Miner, set operations are done by invoking the corresponding device functions predefined in the GPU primitive library. We leverage GPU hardware SIMD support to implement efficient set operations (§6.1) and flexibly support various data formats for vertex sets (§6.2).

6.1 SIMD-aware Primitives

Given two sets A and B, we need two major set operations in GPM: (1) set intersection: $C = A \cap B$; (2) set difference: $C = A - B$, where C is the output set. Besides, another operation

set bounding is also often needed: given a set A and an upper bound y, set bounding computes $\{x|x < y \& x \in A\}$. We discuss set intersection in detail, and the other operations are similar.

In Algorithm 1 Line 4, the result of set intersection is stored in a buffer W for reuse. Buffering is widely used in GPM algorithms to avoid repetitive computation [74]. To support buffering in G^{2}Miner, each warp is allocated a private buffer in the GPU memory. In the primitive functions, threads in a warp write outputs to the buffer in parallel. To do this efficiently, we use CUDA warp-level primitives [69] which are supported by the GPU hardware (special instructions). For each vertex v in set A, we use a boolean flag to indicate whether it exists in set B. Using the flag, we compute a mask using `__ballot_sync` primitive. The mask is then used to compute the index and the total size of the buffer using `__popc` primitive.

Implementation details. Previous work has explored set intersection for SIMD [10, 15, 45, 51, 118] or GPU [8, 38, 40, 41, 49, 50, 79, 80, 112, 113]. We classify their algorithms into 3 categories: *Merge-path* [40, 41], *Binary-search* [38, 50] and *Hash-indexing* [80]. We have extensively evaluated these methods on GPU, and we find that binary search works the best since it is less divergent. In our library, we implement a high-performance binary search [50]: to exploit temporal locality, we leverage the scratchpad in GPU to pre-load the first five layers of the binary search tree, which further mitigates memory divergence. We extend this method to also support set difference, set bounding, and local graph construction.

6.2 Flexible Data Representation

Vertex set is a key data structure in GPM, which is used for the neighbor list in $\mathcal{G}$ and the buffer W in Algorithm 1. Its representation on GPU has a major impact on performance. For the set operations particularly, using a dense representation makes set operations easy to compute, but it requires more storage space. If using a sparse representation, it saves space but complicates the computation of set operations.

We support two types of formats for vertex set on GPU: `sorted-list` (sparse) and `bitmap` (dense). `sorted-list` is a list (i.e. array) of vertices sorted in ascending order. `bitmap` is a sequence of bits (length=$|\mathcal{V}|$), each of which indicates the connectivity to a vertex in $\mathcal{V}$. Set operations on `bitmap` are very simple and efficient, but `bitmap` consumes more space when $\mathcal{V}$ is large. Thus, by default we use `sorted-list`, and we only enable `bitmap` for hub-patterns since the `bitmap` size can be reduced significantly (Δ instead of $|\mathcal{V}|$).

7 Runtime Scheduling and Management

Our runtime system is aware of the pattern, input data graph and GPU architecture to balance workload among multiple GPUs (§7.1) and make full use of the GPU memory (§7.2).

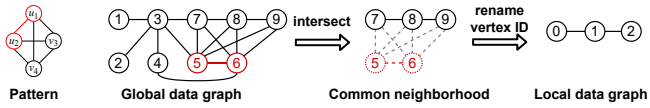


Figure 7: Local graph constructed for v_1=5 and v_2=6 which are matched to hub vertices u_1 and u_2 in the pattern respectively. We first compute set intersection of vertex 5 and 6, to get their common neighbors (vertex 7, 8, 9). The common neighbors are renamed to form a local graph. Renaming can reduce `bitmap` storage.

7.1 Task Scheduling for Multi-GPU

Given n as the number of GPUs[1] available in the system and a data graph $\mathcal{G}$ with an edgelist $\Omega = \{e_1, e_2, ..., e_m\}$ where $m = |\mathcal{E}|$ (in the case of symmetry breaking at level 2, $m = |\mathcal{E}|/2$), the task scheduler aims to divide GPM computation onto the n GPUs, by dividing Ω into n segments, each of which has the same amount of work, such that the execution time of the last completed GPU is minimized.

BFS-based GPM systems, e.g., Arabesque, RStream, and Pangolin, balance workload by reassigning tasks at every level. But this does not work for the DFS algorithm because DFS does not work in the level-by-level way as BFS. Existing DFS-based GPM systems target only CPUs, and thus can use sophisticated work stealing techniques [33]. But this will incur non-trivial runtime overhead on multi-GPU ($\sim$20%) [23, 50].

Policy 1: Even-split Scheduling. Ω is to evenly split into n consecutive ranges, each of which contains m/n tasks. This is used in existing triangle counting solvers on multi-GPU [80]. This policy is simple and has no scheduling overhead, but it results in severe load imbalance for skewed graphs. Fig. 8 shows the time spent on each GPU to finish its work under the even-split scheme. Due to the skewness of the workload assigned to each GPU, under the 2-GPU setting we observe that GPU_0 takes much more time to finish its work than GPU_1. The same time variance is observed for the 3-GPU and 4-GPU setting. Worse still, in the 4-GPU setting, since most of the heavy tasks are assigned to GPU_1, it makes the 4-GPU setting even slower than the 3-GPU setting. This means the even-split scheme does not scale beyond 3-GPU for this benchmark. The reason of poor scalability is two-folds: (1) the granularity of splitting workload is too coarse-grain; (2) it is unaware to the skewness of task workload by assuming every task has the same amount of work.

Policy 2: Round-robin Scheduling. Each GPU has a task queue, denoted as Q_i for the i-th GPU, $i \in [0, n)$. The tasks in Ω are assigned to each queue in a round-robin fashion, i.e., e_j is assigned to Q_i, where $i = j \bmod n$, $j \in [0, m)$. This is a fine-grained scheduling policy that has been used in existing motif counting solvers on multi-GPU [89]. The policy comes with some overhead, i.e., copying tasks into task queues. This copy is needed only once for a specific data graph and n, i.e., once

[1]We assume that every GPU has the same compute power for simplicity, otherwise it is not difficult to scale the workload by a factor accordingly.

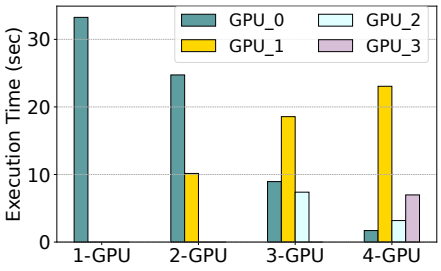


Figure 8: Running time of each GPU using even-split: 3-MC on Tw2.

done, the queues can be reused for mining different patterns.

Policy 3: Chunked Round-robin Scheduling. Ω is first split into lots of small chunks, and then we assign chunks to the task queues in a round-robin way. This is a generalization of the previous two policies. When the chunk size $c = m/n$, it becomes the same as policy 1. When $c = 1$, it becomes the same as policy 2. Thus if c is too small the data copying overhead will be high, but if c is too large, we see load imbalance as in policy 1. We use $c = \alpha \times y$, where y is the total number of warps and α is a constant (set to 2 empirically). Our chunking is also pattern aware, as described in §7.2.

Implementation details. To further reduce data copy overhead, we parallelize it as the location to copy to is fixed for each queue if the chunk size is fixed. Note that this overhead is constant to the pattern size k, which is trivial (< 1%) when $k > 3$, since the GPM computation is exponential to k. For small pattern like triangle, we overlap the scheduling overhead with the GPU computation, by first assigning a few chunks to each GPU and launch the kernel. During the GPU computation, we continue sending the remaining chunks from the CPU to feed the GPUs. Orthogonal work on ordering tasks in Ω [89] or grouping tasks by community may help further improve load balance and locality.

7.2 GPU Memory Management

GPU memory is a scarce resource. In GPM algorithms, the major memory usage involves the data graph $\mathcal{G}$, the edgelist Ω and the buffers (e.g., W in Algorithm 1)). For FSM, the subgraph list of each pattern requires additional space.

(1) Preprocessing the data graph. We have discussed *orientation* in §4.2. In the multi-GPU setting, for any hub-pattern, since the search is confined in the root vertex v_1's neighborhood, we partition $\mathcal{V}$ into n subsets (n is the number of GPUs). For each i-th subset we generate its vertex induced subgraph of $\mathcal{G}$, and copy it to the i-th GPU. This partitioning reduces memory usage and guarantees that there is no communication needed between GPUs. This technique has been used in [47] only for triangle counting. We generalize it for all hub-pattern problems. The scheduling policy is then adjusted by chunking vertices and assigning incident edges in Ω to the corresponding GPUs. For non-hub patterns, we do not partition $\mathcal{G}$ if it can fit in the single-GPU memory. This is because GPM algorithms access multi-hop neighbors, which leads to non-trivial communication overhead [99], especially for small-diameter graphs. When $\mathcal{G}$ is too large to fit in memory, we leverage community-aware partition [56] to minimize communication.

(2) Reducing the size of edgelist. For Ω, we apply an important optimization by considering symmetry at the edge level (level-2). Since $\mathcal{G}$ is an undirected graph, for each undirected edge in $\mathcal{G}$, the edgelist contains two instances, each for one of the two directions of the edge. However, when there is a partial order between v_1 and v_2 for symmetry breaking, we generate the edgelist that contains only one instance. More specifically, if $v_1 > v_2$ is included in the symmetry order (e.g., in Fig. 5 (b)), the edgelist includes only the edges whose source vertex id is larger than its destination vertex id. In this way, we can reduce half of the edges before execution. It not only saves memory but also reduces checking on-the-fly. Note that there is a similar optimization [96] to split the neighbor list of each vertex v into two sets, with one holding all neighbors whose IDs are larger than v, and the other holding the rest which have smaller IDs than v. This reduces on-the-fly checking, but it is not used to reduce memory usage.

(3) Adaptive buffering. In G^{2}Miner's warp-centric DFS walk, each warp is allocated with X buffers. The value of X is pattern specific and the pattern analyzer can decide it when generating the search plan. For a pattern of size k, $X \leq k-3$ because the first two levels and the last level do not need buffers. So the worst case memory consumption for buffering is $O(\Delta \times (k-3))$. This is linear to k for a given specific data graph. In comparison, the intermediate data generated in Pangolin is exponential to k, which can be easily over the GPU memory capacity (see in §8.1). Although Δ is much smaller than $\mathcal{E}$ (see Table 3), given the large number of warps in GPU, the memory space for buffers can still be very large. Therefore, the runtime limits the total number of warps to save memory usage, so that all tasks assigned to the same warp share the buffer usage. In this way, given different data graphs, we can adaptively tune the number of warps to make full use of memory and maximize parallelism. More specifically, we subtract the size of $\mathcal{G}$ and Ω from the total GPU memory size, to get the remaining memory size, denoted as Y. Then we can get the maximum number of warps $Y/(X \times \Delta)$. Finally we launch $\texttt{min}(Y/(X \times \Delta), |\Omega|)$ warps.

(4) Reducing memory allocation using label frequency. This optimization is particularly useful for problems that find *frequent patterns*, such as FSM. The graph loader in G^{2}Miner computes the vertex frequency for each label. This information can be leveraged to find *frequent labels*, i.e., labels with vertex frequency above the user-defined support threshold σ_{min}. Since infrequent labels can not be part of frequent patterns, the total number of possible frequent patterns N can be significantly reduced, if there are many infrequent labels. Note that in FSM we allocate a *subgraph list* for each possible pattern to store subgraphs for aggregation, and the memory consumption of these subgraph lists is proportional to N. With

Graph	Source	|V|	|E|	Label	Max deg. Δ
Mi	Mico [35]	0.1M	2M	29	1,359
Pa	Patents [44]	3M	28M	37	789
Yo	Youtube [28]	7M	114M	28	4,017
Lj	LiveJournal [66]	4.8M	43M	0	20,333
Or	Orkut [66]	3.1M	117M	0	33,313
Tw2	Twitter20 [63]	21M	530M	0	698,112
Tw4	Twitter40 [64]	42M	2,405M	0	2,997,487
Fr	Friendster [115]	66M	3,612M	0	5,214
Uk	Uk2007 [16]	106M	6,603M	0	975,419

Table 3: Data graphs (symmetric, no loops or duplicate edges). Maximum degrees are smaller when orientation is applied for cliques.

Data Graph	Lj	Or	Tw2	Tw4	Fr	Uk
G^{2}Miner (GPU)	0.03	0.14	1.6	5.1	3.2	7.5
Pangolin (GPU)	0.06	0.25	3.0	OoM	5.2	OoM
PBE (GPU)	0.27	1.12	13.4	53.5	23.0	55.3
Peregrine (CPU)	1.63	7.25	112.1	8492.4	100.3	3640.9
GraphZero (CPU)	0.61	2.22	24.4	1399.3	49.0	1041.3

Table 4: TC running time (sec). OoM: out of memory.

this awareness of the input (i.e., label frequency), we can drastically reduce this memory consumption in many cases.

8 Evaluation

We compare G^{2}Miner[2] with state-of-the-art systems: (1) GPM system on GPU, Pangolin [26], (2) subgraph matching solver on GPU, PBE [42, 43], (3) CPU-based GPM system Peregrine [53] and (4) CPU-based subgraph matching system GraphZero [73, 74]. Note that Pangolin also provides a CPU implementation, but it is slower than GraphZero.

Table 3 lists the data graphs. The first 3 graphs (`Mi`, `Pa`, `Yo`) are vertex-labeled graphs which are used for FSM. We use all the GPM applications listed in §2.1 for evaluation, i.e., TC, k-CL, SL, k-MC. For SL, we use two patterns `4-cycle` and `diamond`. Note that GraphZero does not support FSM, Pangolin does not support SL, and PBE does not support k-MC and FSM. For FSM, we include DistGraph [99] in Table 8 as the state-of-the-art hand-written FSM solver.

CPU-based systems and solvers are evaluated on a 4 socket machine with Intel Xeon Gold 5120 2.2GHz CPUs (56 cores in total) and 190GB RAM, while GPU-based solutions are evaluated on NVIDIA V100 GPUs (each with 32GB device memory). We exclude preprocessing (e.g., DAG construction in Pangolin and vertex reordering in Peregrine) time in all systems. We use a time-out of 30 hours for CPU and 8 hours for GPU, and report all results as an average of three runs. We show single-GPU performance in §8.1 and compare with CPU solutions in §8.2. Multi-GPU performance of G^{2}Miner is shown in §8.3. Impact of optimizations is analyzed in §8.4.

8.1 Single-GPU Performance

We compare with Pangolin and PBE on a V100 GPU. Table 4 lists the GPU running time for triangle counting (TC). We

[2]G^{2}Miner source code: https://github.com/chenxuhao/GraphMiner

Pattern	4-CL					5-CL		
Data Graph	Lj	Or	Tw2	Tw4	Fr	Lj	Or	Fr
G^{2}Miner (G)	0.32	0.54	113.3	362.9	7.3	3.2	1.7	13.1
Pangolin (G)	1.48	4.04	OoM	OoM	OoM	OoM	OoM	OoM
PBE (G)	3.90	11.11	3640.1	TO	117.8	246.4	99.2	399.8
Peregrine (C)	15.90	73.70	39921.0	TO	397.3	520.8	782.1	957.6
GraphZero (C)	3.48	12.96	2152.2	20591.1	177.7	60.0	48.3	243.3

Table 5: k-CL running time (sec). TO: timed out.

Pattern	Diamond					4-cycle		
Data Graph	Lj	Or	Tw2	Tw4	Fr	Lj	Or	Fr
G^{2}Miner (G)	0.29	0.75	26.8	183.1	12.8	2.7	33.7	1291.2
PBE (G)	0.48	1.71	26.3	102.0	39.9	17.3	177.8	5211.3
Peregrine (C)	5.38	10.24	553.6	20898.4	178.1	144.4	1867.2	32276.8
GraphZero (C)	1.73	7.27	165.1	7938.6	136.4	34.0	345.5	9251.5

Table 6: SL running time (sec). 'G': GPU; 'C': CPU.

observe that Pangolin runs out of memory for `Tw4`[3] and `Uk`, while G^{2}Miner can run with all the data graphs. We also observe that G^{2}Miner is constantly faster than Pangolin, due to optimized set operations in our library. On average, G^{2}Miner is 1.8× faster than Pangolin on V100 GPU.

The speedups are more significant for k-CL and k-MC. As shown in Table 5, G^{2}Miner outperforms Pangolin by 4.6× and 7.6× for 4-clique listing on `Lj` and `Or` respectively. The speedups mainly come from data reuse enabled in DFS (i.e., buffering W in Algorithm 1) and optimized set operations[4]. Meanwhile, for all the rest of graphs and the larger pattern 5-clique, Pangolin runs out of memory. Similar trend is found in Table 7, where we observe an average of 21.3× speedup over Pangolin on 3-MC, and Pangolin also runs out of memory for most of the cases. G^{2}Miner managed to run all cases, which demonstrates that its DFS order and optimization J and K in Table 2 can effectively reduce memory consumption.

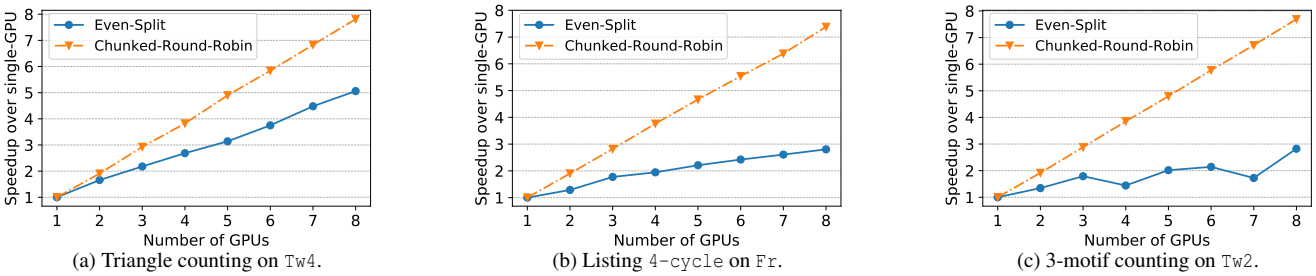
For FSM in Table 8, G^{2}Miner is competitive with Pangolin for the small graphs, since we use bounded BFS (optimization M in Table 2) that provides enough parallelism. For the largest graph `Yo`, Pangolin runs out of memory again, while G^{2}Miner succeeds to run it, thanks to both optimization M and N in Table 2 which help reduce memory consumption.

Overall, G^{2}Miner achieves an average speedup of **5.4**× over Pangolin, and the speedup is more significant for larger patterns. Moreover, G^{2}Miner can run much larger graphs.

We also compare with PBE [42, 43] on the V100 GPU. PBE partitions the data graph when it gets large, which allows it run all the single-pattern workloads. However, its performance is even worse (3.8× slower) than Pangolin, due to the cross-partition communication overhead and lack of data graph orientation. Particularly, for subgraph listing, as `diamond` contains a sub-pattern `triangle` but `4-cycle` does not, searching `diamond` generates much less intermediate data than searching `4-cycle`. Thus in Table 6 we observe that PBE's `4-cycle` performance is much worse than G^{2}Miner as

[3]Since data graphs are oriented in TC, `Fr` takes less memory than `Tw4`

[4]It can not be directly used for Pangolin, as Pangolin maps *connectivity checks* [26] to threads, but G^{2}Miner maps set operations to warps.



(a) Triangle counting on `Tw4`. (b) Listing `4-cycle` on `Fr`. (c) 3-motif counting on `Tw2`.

Figure 9: G^{2}Miner multi-GPU scalability using two task scheduling policies: even-split vs. chunked-split.

Pattern	3-Motif					4-Motif		
Data Graph	Lj	Or	Tw2	Tw4	Fr	Lj	Or	Fr
G^{2}Miner (G)	0.17	0.97	33.3	1703.6	22.0	138.1	2068.4	15475.4
Pangolin (G)	2.05	22.62	1165.5	OoM	OoM	OoM	OoM	OoM
Peregrine (C)	9.36	19.46	418.7	27954.9	367.9	1435.4	20219.1	TO
GraphZero (C)	1.50	7.74	276.5	7439.4	169.6	3039.6	16394.6	TO

Table 7: *k*-MC running time (s). OoM: out of mem.; TO: timed out.

Data Graph	Mico				Patent				Youtube			
σ	300	500	1000	5000	300	500	1000	5000	300	500	1000	5000
G2Miner (G)	0.6	0.4	0.3	0.1	2.6	2.6	2.6	1.7	7.2	6.0	6.0	8.7
Pangolin (G)	0.6	0.5	0.3	0.2	2.7	2.7	2.7	1.7	OoM	OoM	OoM	OoM
Peregrine (C)	4.4	4.4	4.2	4.3	94.2	103.8	118.4	94.3	59.3	52.8	69.9	60.8
DistGraph (C)	56.1	61.0	57.6	57.0	13.2	13.1	13.0	14.1	OoM	OoM	OoM	OoM

Table 8: 3-FSM running time (sec). OoM: out of memory.

it has to do partitioning and suffers from the overhead. Overall, G^{2}Miner achieves a **7.2**× speedup over PBE on average.

8.2 Mining on GPU vs. on CPU

To evaluate how much speedup we can get from GPU over CPU, we compare G^{2}Miner (on V100 GPU) with GraphZero (on 56-core CPU). Note that for each specific GPM application, G^{2}Miner and GraphZero use exactly the same matching order and symmetry order, making it a fair comparison to show the benefit from the difference of hardware architectures. As listed in Table 4, G^{2}Miner is significantly faster than GraphZero on TC, with an average speedup of 38.0×. The same trend is observed for *k*-CL in Table 5, where G^{2}Miner outperforms GraphZero by 18.2×. This tremendous performance improvement is due to three parts: (1) the orientation optimization, (2) higher throughput (i.e. more parallelism) on GPU, and (3) our high-performance set operations on GPU.

For SL, orientation can not be applied. Thus it can be used to evaluate the benefit of the other two parts. As shown in Table 6, G^{2}Miner still achieves overwhelmingly better performance than GraphZero, with an average speedup of 10.5×. The speedup would be marginal if we use the BFS strategy in Pangolin and PBE or implement our DFS scheme naively.

While TC, *k*-CL and SL uses only set intersection, *k*-MC includes both set intersection and set difference. As G^{2}Miner optimizes both operations, we also observe dramatic performance boost for *k*-MC. In Table 7, it constantly outperforms GraphZero for all benchmarks. On average G^{2}Miner is 8.5× faster than GraphZero.

Overall, G^{2}Miner on GPU achieves **15.2**× speedup over GraphZero on CPU, which demonstrates the significant benefit of using GPU to accelerate GPM applications.

As GraphZero does not support FSM, we also compared to Peregrine. G^{2}Miner on GPU is **48.3**× faster than Peregrine on CPU. Note that Peregrine does not mine multiple patterns simultaneously for multi-pattern problems. Instead, for *k*-MC and FSM, it enumerates every pattern one by one, making it impossible to reuse data across similar patterns. Thus it is mostly even slower than GraphZero.

8.3 Multi-GPU Scalability

We evaluate multi-GPU performance by varying the number of GPUs from 1 to 8 in a single machine. Since PBE and Pangolin do not support multi-GPU, we only evaluate G^{2}Miner in this section. We compare two task scheduling policies in Fig. 9. As illustrated, the chunked round-robin scheme constantly works much better than the even-split scheme. More importantly, the chunked scheme scales linearly for all cases, while the even-split scheme fails to scale beyond 3-GPU for 3-MC on `Tw2`. The poor scalability of even-split is dues to the load imbalance. As shown in Fig. 10, in the 4-GPU setting, the execution time of each GPU varies dramatically for the even-split setting. In contrast, for the chunked scheme, each GPU finishes its work roughly at the same time.

8.4 Impact of Optimizations

Different optimizations in Table 2 contribute differently to the performance improvement. First, architecture-aware optimizations are crucial for all workloads on GPU. G^{2}Miner is 5.4× faster then Pangolin, where *two-level parallelism* (C in Table 2) and *SIMD-aware primitives* (H in Table 2) contribute 3.1× and 1.7× respectively. Second, for a pattern-aware optimization, it is beneficial only for the target pattern(s), and the speedups vary a lot depending on how much the search space is pruned. For example, *local-graph search* (E+F in Table 2) brings 1.2× ∼ 3.7× speedup for `hub-patterns` (2.1× on av-

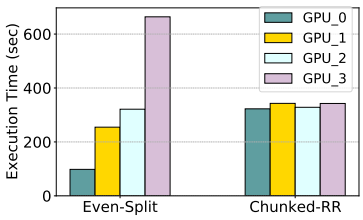


Figure 10: Running time of each GPU in the 4-GPU setting: `4-cycle` on `Fr`.

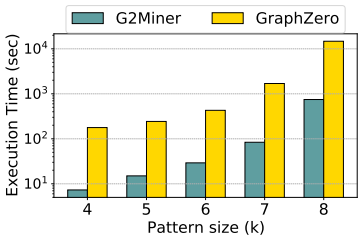


Figure 11: Running time of k-clique listing over `Fr`, $k \in [4,8]$.

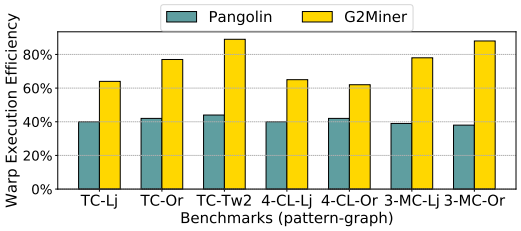


Figure 12: Warp execution efficiency.

Pattern	Diamond					3-Motif					4-Motif		
Time (sec)	Lj	Or	Tw2	Tw4	Fr	Lj	Or	Tw2	Tw4	Fr	Lj	Or	Fr
G^{2}Miner (GPU)	0.09	0.47	9.9	66.9	10.4	0.06	0.27	6.8	21.4	5.2	2.6	34.2	1307.2
Peregrine (CPU)	2.20	8.66	245.8	16312.6	158.8	2.51	4.90	116.0	8447.4	165.3	163.6	1701.4	TO

Table 9: Running time of G^{2}Miner vs. Peregrine, both with counting-only pruning enabled. TO: timed out.

erage), while *counting-only pruning* (D in Table 2) achieves 1.2× (`diamond`, `Fr`) to 79.7× (`3-motif`, `Tw40`), with 6.2× on average. Other optimizations in Table 2 are for memory saving, which is crucial for enabling larger datasets.

Large Pattern and Large Graph. A major advantage of G^{2}Miner over Pangolin is that G^{2}Miner can support much larger graphs and patterns. Fig. 11 shows that G^{2}Miner can run up to 8-clique listing on a billion-edge graph `Fr`. In contrast, Pangolin can not even run 4-clique due to out-of-memory, as shown in Table 5. Fig. 11 also shows that, from 4-clique to 8-clique, G^{2}Miner on GPU consistently achieves an order of magnitude speedup over GraphZero on the CPU, although the GPU has much less memory than the CPU. This trend implies that GPUs can be not only capable but also highly efficient for processing large graphs and patterns, thanks to G^{2}Miner's memory management and optimizations for the GPU architecture.

GPU Efficiency. To evaluate GPU utilization, we measure *warp execution efficiency*, which is the average percentage of active threads in each executed warp. As shown in Fig. 12, the warp execution efficiency in Pangolin is around 40%. This is relatively low since more than half of the compute horse power is wasted. In comparison, G^{2}Miner significantly improves the warp execution efficiency. This is mainly due to the highly efficient implementation of our warp-centric set operations. Besides, we also measure *branch efficiency*, i.e., the ratio of non-divergent branches to total branches. Although G^{2}Miner uses DFS, We find that Pangolin and G^{2}Miner have almost the same branch efficiency, thanks to the two-level parallelism scheme. Since we assign each task to a warp, all threads in a warp does the same DFS walk synchronously, which avoids most of the branch divergence. This creates some redundancy, but since most of execution time is spent on set operations, it is still a good tradeoff.

Counting-only pruning. In §8.1, we do not enable optimization D in Table 2, because GraphZero and Pangolin do not support it. We observe that for those patterns (e.g., `diamond`) enabling this pruning in G^{2}Miner further improve performance by 6.2× on average. Enabling this optimization in Peregrine also improves its performance, as shown in Table 9. However, due to our high efficiency on GPU, G^{2}Miner still outperforms Peregrine by 41.1× when both enable it. This again demonstrates the performance superiority of GPU over CPU, no matter what algorithm optimizations are applied.

Sorting and renaming vertices. For fair comparison, this optimization done by the preprocessor is also not enabled in §8.1. Our evaluation shows that this can futher improve G^{2}Miner performance by 5% (up to 90%). Applying this to GraphZero also helps, but G^{2}Miner is still 12× faster.

9 Conclusion

We present G^{2}Miner, the first multi-GPU GPM framework that supports efficiently mining large graphs and patterns. For high efficiency, G^{2}Miner is aware of the input, pattern and architecture to fully unlock the potential of GPM computing on GPUs, which results in a 5× speedup over the state-of-the-art GPU-based GPM system, Pangolin, on a single GPU. For scalability, G^{2}Miner employs a custom task scheduler that can scale GPM computation to multiple GPUs linearly. For programmability, it automatically enables applicable optimizations and generates CUDA code, which hides away GPU programming complexity, and in turn provides the same easy-to-use programming interface as the state-of-the-art CPU-based GPM frameworks (e.g., Peregrine). We also show that G^{2}Miner on a single V100 GPU is 48× faster than Peregrine on a 56-core Intel CPU, a free lunch for GPM users.

10 Acknowledgements

This research is funded by Samsung Semiconductor (GRO grants) and MIT-IBM Watson AI Lab, and supported by XSEDE allocation TG-CIE-170005 and ASC22045. We thank Tianhao Huang and OSDI reviewers for their feedback.

References

[1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad 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*, SC '16, pages 61:1–61:12, Piscataway, NJ, USA, 2016. IEEE Press.

[2] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4), October 2017.

[3] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *ICDM*, pages 1–10, 2015.

[4] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery*, 29(3):626–688, 2015.

[5] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. K-clique counting on gpus. *arXiv preprint arXiv:2104.13209*, 2021.

[6] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and SC. Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):241–249, 2008.

[7] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow.*, 11(6):691–704, February 2018.

[8] Rasmus Resen Amossen and Rasmus Pagh. A new data layout for set intersection on gpus. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 698–708. IEEE, 2011.

[9] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides 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*, pages 16–24, 2008.

[10] Christos Bellas and Anastasios Gounaris. An evaluation of large set intersection techniques on gpus. In *DOLAP*, pages 111–115, 2021.

[11] Austin R. Benson, David F. Gleich, and Jure Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.

[12] Indrajit Bhattacharya and Lise Getoor. Entity resolution in graphs. *Mining graph data*, 311, 2006.

[13] Bibek Bhattarai, Hang Liu, and H. Howie Huang. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1447–1462, New York, NY, USA, 2019. ACM.

[14] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1199–1214, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Jovan Blanuša, Radu Stoica, Paolo Ienne, and Kubilay Atasu. Manycore clique enumeration with fast set intersections. *Proc. VLDB Endow.*, 13(12):2676–2690, July 2020.

[16] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[17] Vincenzo Bonnici, Rosalba Giugno, and Nicola Bombieri. An efficient implementation of a subgraph isomorphism algorithm for gpus. In *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 2674–2681. IEEE, 2018.

[18] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, Nov 2012.

[19] Amlan Chatterjee, Sridhar Radhakrishnan, and John K. Antonio. Counting problems on graphs: Gpu storage and parallel computing techniques. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 804–812, 2012.

[20] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: An efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Jingji Chen and Xuehai Qian. Dwarvesgraph: A high-performance graph mining system with pattern decomposition, 2021.

[22] Jingji Chen and Xuehai Qian. Kudu: An efficient and scalable distributed graph pattern mining engine, 2021.

[23] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. Dynamic load balancing on single-and multi-gpu systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[24] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive cache management for energy-efficient gpu computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 343–355, Washington, DC, USA, 2014. IEEE Computer Society.

[25] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining. In *Proceedings of the 35th ACM International Conference on Supercomputing*, ICS '21, 2021.

[26] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proc. VLDB Endow.*, 13(8), August 2020.

[27] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind. Flexminer: A pattern-aware accelerator for graph pattern mining. In *Proceedings of the International Symposium on Computer Architecture*, 2021.

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

[29] Young-Rae Cho and Aidong Zhang. Predicting protein function by frequent functional association pattern mining in protein interaction networks. *IEEE Transactions on information technology in biomedicine*, 14(1):30–36, 2009.

[30] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs*. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 589–598, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

[31] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering*, 17(8):1036–1050, Aug 2005.

[32] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 393–404, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1357–1374, New York, NY, USA, 2019. ACM.

[34] Carmel Domshlak, Samir Genaim, and Ronen Brafman. Preference-based configuration of web page content. In *14th European Conference on Artificial Intelligence (ECAI 2000), Configuration Workshop, Berlin, Germany*, pages 19–22, 2000.

[35] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528, March 2014.

[36] Katherine Faust. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Social Networks*, 32(3):221 – 233, 2010.

[37] Dima Feldman and Yuval Shavitt. Automatic large scale generation of internet pop level maps. In *IEEE Global Telecommunications Conference (GLOBECOM)*, pages 1–6. IEEE, 2008.

[38] James Fox, Oded Green, Kasimir Gabert, Xiaojing An, and David A Bader. Fast and adaptive list intersections on the gpu. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

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

[40] Oded Green, James Fox, Alex Watkins, Alok Tripathy, Kasimir Gabert, Euna Kim, Xiaojing An, Kumar Aatish, and David A Bader. Logarithmic radix binning and vectorized triangle counting. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[41] Oded Green, Robert McColl, and David A. Bader. Gpu merge path: A gpu merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 331–340, New York, NY, USA, 2012. ACM.

[42] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page

1067–1082, New York, NY, USA, 2020. Association for Computing Machinery.

[43] Wentian Guo, Yuchen Li, and Kian-Lee Tan. Exploiting reuse for gpu subgraph enumeration. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020.

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

[45] Shuo Han, Lei Zou, and Jeffrey Xu Yu. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1587–1602, 2018.

[46] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turbo_{iso}: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 337–348, New York, NY, USA, 2013. Association for Computing Machinery.

[47] Loc Hoang, Vishwesh Jatala, Xuhao Chen, Udit Agarwal, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. DistTC: High performance distributed triangle counting. In *HPEC 2019 23rd IEEE High Performance Extreme Computing, Graph Challenge*, September 2019.

[48] Paul W Holland and Samuel Leinhardt. Local structure in social networks. *Sociological methodology*, 7:1–45, 1976.

[49] Lin Hu, Lei Zou, and Yu Liu. Accelerating triangle counting on gpu. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD-/PODS '21, page 736–748, New York, NY, USA, 2021. Association for Computing Machinery.

[50] 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*, pages 171–182, Nov 2018.

[51] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proc. VLDB Endow.*, 8(3):293–304, November 2014.

[52] Shweta Jain and C. Seshadhri. A fast and provable method for estimating clique counts using turán's theorem. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 441–449, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[53] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A pattern-aware graph mining system. In *Proceedings of the Fifteenth EuroSys Conference*, EuroSys '20, 2020.

[54] Madhav Jha, C. Seshadhri, and Ali 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*, WWW '15, pages 495–505, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee.

[55] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1695–1698, New York, NY, USA, 2017. Association for Computing Machinery.

[56] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

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

[58] Robest Kessl, Nilothpal Talukder, Pranay Anchuri, and Mohammed J. Zaki. Parallel graph mining with gpus. In *Proceedings of the 3rd International Conference on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications - Volume 36*, BIGMINE'14, pages 1–16. JMLR.org, 2014.

[59] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H.A. Jarrah. DUALSIM: Parallel subgraph enumeration in a massive graph on a single machine. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1231–1245, New York, NY, USA, 2016. ACM.

[60] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 411–426, New York, NY, USA, 2018. ACM.

[61] Daphne Koller, Nir Friedman, Sašo Džeroski, Charles Sutton, Andrew McCallum, Avi Pfeffer, Pieter Abbeel, Ming-Fai Wong, Chris Meek, Jennifer Neville, et al. *Introduction to statistical relational learning*. MIT press, 2007.

[62] Oleksii Kuchaiev, Tijana Milenković, Vesna Memišević, Wayne Hayes, and Nataša Pržulj. Topological network alignment uncovers biological function and phylogeny. *Journal of the Royal Society Interface*, 7(50):1341–1354, 2010.

[63] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[64] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[65] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce. *Proc. VLDB Endow.*, 8(10):974–985, June 2015.

[66] J. Leskovec. Snap: Stanford network analysis platform, 2013.

[67] W. Lin, X. Xiao, X. Xie, and X. Li. Network motif discovery: A gpu approach. In *2015 IEEE 31st International Conference on Data Engineering*, pages 831–842, April 2015.

[68] Xiaojie Lin, Rui Zhang, Zeyi Wen, Hongzhi Wang, and Jianzhong Qi. Efficient subgraph matching using gpus. In Hua Wang and Mohamed A. Sharaf, editors, *Databases Theory and Applications*, pages 74–85, Cham, 2014. Springer International Publishing.

[69] Yuan Lin and Vinod Grover. Using cuda warp-level primitives. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/, 2018.

[70] Yongchao Liu, Bertil Schmidt, Weiguo Liu, and Douglas L. Maskell. Cuda-meme: Accelerating motif discovery in biological sequences using cuda-enabled graphics processing units. *Pattern Recognition Letters*, 31(14):2170 – 2177, 2010.

[71] Bruce T Lowerre. *The harpy speech recognition system.* Carnegie Mellon University, 1976.

[72] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. Distributed graph pattern matching. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 949–958, New York, NY, USA, 2012. ACM.

[73] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. Graphzero: A high-performance subgraph matching system. *SIGOPS Oper. Syst. Rev.*, 55(1):21–37, June 2021.

[74] Daniel Mawhirter and Bo Wu. Automine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 509–523, New York, NY, USA, 2019. ACM.

[75] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, July 2019.

[76] Tijana Milenković, Weng Leong Ng, Wayne Hayes, and Nataša Pržulj. Optimal network alignment with graphlet degree vectors. *Cancer informatics*, 9:CIN–S4744, 2010.

[77] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[78] Caleb C Noble and Diane J Cook. Graph-based anomaly detection. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, 2003.

[79] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. Merge path-parallel merging made simple. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1611–1618. IEEE, 2012.

[80] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. H-index: Hash-indexing for parallel triangle counting on gpus. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, Sep. 2019.

[81] Roger Pearce, Trevor Steil, Benjamin W Priest, and Geoffrey Sanders. One quadrillion triangles queried on one million processors. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5. IEEE, 2019.

[82] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 1431–1440, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[83] Natasa Pržulj, Derek G Corneil, and Igor Jurisica. Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[84] Liva Ralaivola, Sanjay J Swamidass, Hiroto Saigo, and Pierre Baldi. Graph kernels for chemical informatics. *Neural networks*, 18(8):1093–1110, 2005.

[85] Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. Pgx. iso: parallel and efficient in-memory engine for subgraph isomorphism. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6, 2014.

[86] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. Fast and robust distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 12(11):1344–1356, 2019.

[87] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Divergence-aware warp scheduling. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 99–110, 2013.

[88] Ryan A Rossi and Nesreen K Ahmed. Role discovery in networks. *IEEE Transactions on Knowledge and Data Engineering*, 27(4):1112–1131, 2014.

[89] Ryan A. Rossi and Rong Zhou. Leveraging multiple gpus and cpus for graphlet counting in large networks. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, pages 1783–1792, New York, NY, USA, 2016. ACM.

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

[91] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 625–636, New York, NY, USA, 2014. ACM.

[92] Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. *arXiv preprint arXiv:2002.10047*, 2020.

[93] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

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

[95] Shuya Suganami, Toshiyuki Amagasa, and Hiroyuki Kitagawa. Accelerating all 5-vertex subgraphs counting using gpus. In *International Conference on Database and Expert Systems Applications*, pages 55–70. Springer, 2020.

[96] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. Efficient parallel subgraph enumeration on a single machine. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 232–243. IEEE, 2019.

[97] Shixuan Sun and Qiong Luo. Scaling up subgraph query processing with efficient subgraph matching. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 220–231. IEEE, 2019.

[98] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 607–614, New York, NY, USA, 2011. ACM.

[99] N. Talukder and M. J. Zaki. A distributed approach for graph mining in massive networks. *Data Min. Knowl. Discov.*, 30(5):1024–1052, September 2016.

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

[101] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 425–440, New York, NY, USA, 2015. ACM.

[102] Tatsuya Toki and Tomonobu Ozaki. Experimental evaluation of a gpu-based frequent subgraph miner using synthetic databases. In *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, pages 504–507, 2016.

[103] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. Fast subgraph matching on large graphs using graphics processors. In Matthias Renz, Cyrus Shahabi, Xiaofang Zhou, and Muhammad Aamir Cheema, editors, *Database Systems for Advanced Applications*, pages 299–315, Cham, 2015. Springer International Publishing.

[104] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. aDFS: An almost Depth-First-Search distributed Graph-Querying system. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 209–224. USENIX Association, July 2021.

[105] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[106] Fei Wang, Jianqiang Dong, and Bo Yuan. Graph-based substructure pattern mining using cuda dynamic parallelism. In Hujun Yin, Ke Tang, Yang Gao, Frank Klawonn, Minho Lee, Thomas Weise, Bin Li, and Xin Yao, editors, *Intelligent Data Engineering and Automated Learning – IDEAL 2013*, pages 342–349, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[107] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry 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*, OSDI'18, pages 763–782, Berkeley, CA, USA, 2018. USENIX Association.

[108] Leyuan Wang and John D Owens. Fast gunrock subgraph matching (gsm) on gpus. *arXiv preprint arXiv:2003.01527*, 2020.

[109] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D Owens. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 1–8, 2016.

[110] Martin Winter, Mathias Parger, Daniel Mlakar, and Markus Steinberger. Are dynamic memory managers on gpus slow? a survey and benchmarks. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 219–233, New York, NY, USA, 2021. Association for Computing Machinery.

[111] Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[112] Di Wu, Fan Zhang, Naiyong Ao, Fang Wang, Xiaoguang Liu, and Gang Wang. A batched gpu algorithm for set intersection. In *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pages 752–756. IEEE, 2009.

[113] Di Wu, Fan Zhang, Naiyong Ao, Gang Wang, Xiaoguang Liu, and Jing Liu. Efficient lists intersection by cpu-gpu cooperative computing. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

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

[115] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[116] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, Jonathan Berry, and Ümit V Çatalyürek. Fast triangle counting using cilk. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[117] Li Zeng, Lei Zou, M Tamer Özsu, Lin Hu, and Fan Zhang. Gsi: Gpu-friendly subgraph isomorphism. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1249–1260. IEEE, 2020.

[118] Jiyuan Zhang, Yi Lu, Daniele G Spampinato, and Franz Franchetti. Fesia: A fast and simd-efficient set intersection approach on modern cpus. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1465–1476. IEEE, 2020.

[119] Luming Zhang, Mingli Song, Zicheng Liu, Xiao Liu, Jiajun Bu, and Chun Chen. Probabilistic graphlet cut: Exploiting spatial structure cue for weakly supervised image segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1908–1915, 2013.

[120] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng 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)*, ICDE '20, 2020.

A Artifact Appendix

Abstract

This artifact appendix helps the readers reproduce the main evaluation results of the OSDI' 22 paper: Efficient and Scalable Graph Pattern Mining on GPUs.

Scope

The artifact can be used for evaluating and reproducing the main results of the paper, including Table 4, Table 5, Table 6, Table 7, Table 8 and Fig. 9, Fig. 10, Fig. 11, Fig. 12 in §8.

Contents

The artifact evaluation includes all the experiments in the paper. Details of the experiments are listed here: https://github.com/chenxuhao/GraphMiner/blob/master/OSDI-experiments-guide.md

Hosting

The source code of this artifact can be found on GitHub: https://github.com/chenxuhao/GraphMiner, master branch.

Requirements

Hardware dependencies

This artifact depends on an NVIDIA V100 GPU.

Software dependencies

This artifact requires CUDA toolkit 11.1.1 or greater and GCC 8 or greater.

Details of the dependencies are listed here: https://github.com/chenxuhao/GraphMiner/blob/master/OSDI-experiments-guide.md