gMatch: Fine-Grained and Hardware-Efficient Subgraph Matching on GPUs

Weitian Chen
Shanghai Jiao Tong University
chenweitian@sjtu.edu.cn

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

Cheng Chen
ByteDance Inc
chencheng.sg@bytedance.com

Yongmin Hu
ByteDance Inc
huyongmin@bytedance.com

Yingqian Hu
ByteDance Inc
huyingqian@bytedance.com

Minyi Guo
Guizhou University
myguo@gzu.edu.cn

ABSTRACT

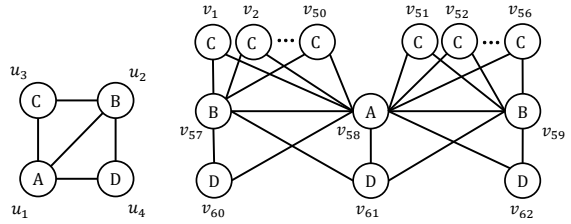
Subgraph matching is a core operation in graph analytics, supporting a broad spectrum of applications from social network analysis to bioinformatics. Recent GPU-based approaches accelerate subgraph matching by leveraging parallelism but rely on a coarse-grained execution model that suffers from scalability and efficiency issues due to high memory overhead and thread underutilization. In this paper, we propose gMatch, a hardware-efficient subgraph matching approach on GPUs. gMatch introduces a fine-grained execution model that reduces memory consumption and enables flexible task scheduling among threads. We further design warp-level batch exploration and lightweight load balancing to improve execution efficiency and scalability. Experiments on diverse workloads and real-world datasets show that gMatch outperforms state-of-the-art subgraph matching methods, including STMatch, T-DFS, and EGSM, in both performance and scalability. We also compare against state-of-the-art systems for mining small patterns, such as BEEP and G^{2}Miner. While these systems achieve better performance on small datasets, gMatch scales to substantially larger queries and datasets, where existing approaches degrade or fail to complete.

PVLDB Reference Format:
Weitian Chen, Shixuan Sun, Cheng Chen, Yongmin Hu, Yingqian Hu, and Minyi Guo. gMatch: Fine-Grained and Hardware-Efficient Subgraph Matching on GPUs. PVLDB, 19(8): 1740 - 1753, 2026.
doi:10.14778/3811243.3811248

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SJTU-Liquid/gMatch.

1 INTRODUCTION

Given a query graph Q and a data graph G, *subgraph matching* aims to find all embeddings of Q in G. For example, in Figure 1, the mapping $M = \{(u_1, v_{58}), (u_2, v_{57}), (u_3, v_1), (u_4, v_{60})\}$ is a valid embedding, or *match*, of Q in G. As a fundamental operation in graph analysis, subgraph matching underpins a wide range of real-world applications, including social network recommendation [16], fraud detection [22], bioinformatics [6, 47], and cheminformatics [40].



(a) Query graph Q. **(b) Data graph G.**

Figure 1: Example query graph and data graph.

Given the importance of subgraph matching, numerous algorithms have been proposed [5, 17, 34]. Although they adopt different optimization techniques, most follow the same core idea: iteratively extend partial matches by mapping query vertices in Q to data vertices in G according to a predefined order. However, since subgraph isomorphism is NP-complete, the search space can be prohibitively large. To accelerate the computation, recent studies leverage GPUs to parallelize the matching process [35, 37, 39, 41, 44, 46].

All these methods adopt a *coarse-grained parallel execution model* for subgraph matching, where each partial match M is treated as a parallel task and a warp, the basic scheduling unit on GPUs, serves as the worker. The task involves extending M by mapping the next query vertex to valid data vertices. Based on their search strategies, these methods can be classified into two categories: breadth-first search (BFS)-based and depth-first search (DFS)-based.

Early approaches such as GpSM [37], GSI [46], and cuTS [41] adopt a BFS-based search strategy, which explores the search space level by level. At each iteration, all partial matches are extended by mapping the next query vertex to candidate data vertices. This strategy enables a large number of parallel tasks, allowing efficient GPU utilization and balanced workload distribution across warps. However, it requires storing all partial matches at every level, leading to high memory consumption due to the large search space. As a result, the scalability of BFS-based methods is severely limited by GPU memory capacity.

To reduce memory overhead, recent methods such as T-DFS [44], STMatch [39], and EGSM [35] adopt a DFS-based search strategy. Instead of expanding all partial matches level by level, each warp recursively explores the search space. Specifically, after generating new partial matches from a given match M, the warp selects one

Proceedings of the VLDB Endowment, Vol. 19, No. 8 ISSN 2150-8097.
doi:10.14778/3811243.3811248

to continue the search. By avoiding the need to store all partial matches at each level, this strategy significantly improves scalability compared to BFS-based approaches.

Key Insights. Despite its widespread adoption, *the coarse-grained parallel execution model fundamentally conflicts with real-world power-law graphs.* Because each partial match has highly dynamic and skewed workloads, while GPUs adopt the SIMT execution model and a warp provides fixed and inflexible computing resources, this design introduces fundamental limitations in both scalability and computational efficiency for GPU-based subgraph matching.

First, *the execution stack incurs high memory overhead.* To handle dynamic workloads, the coarse-grained method must pre-allocate a large execution stack of size $O(|V(Q)| \times d_{\max})$ for each warp, where $d_{\max}$ is the maximum degree of G. The stack stores intermediate results during the search, e.g., feasible candidates and derived partial matches. Since $d_{\max}$ can reach millions in power-law graphs and GPUs require hundreds of active warps for full utilization, this high memory cost limits scalability and reduces concurrency.

Second, *the mismatch between the fixed warp size (32 threads) and the dynamic workloads leads to severe thread underutilization.* To extend a partial match, the coarse-grained method performs set intersections over the neighbors of previously matched data vertices. Most candidate vertices have very small degrees after filtering. A warp must still execute 32 threads in lock-step, causing pervasive idle threads and wasted GPU parallelism.

These limitations are not caused by specific implementations; they stem from the inherent structure of the coarse-grained paradigm itself. Overcoming them requires fundamentally rethinking how subgraph matching should be parallelized on GPUs.

Our Work. In this paper, we propose gMatch, a hardware-efficient subgraph matching approach on GPUs. *To resolve the structural mismatch between the GPU execution model and the coarse-grained parallel execution, we first introduce a fine-grained parallel execution model that redefines the basic task and the basic worker.* Instead of letting a warp process one partial match, we let a single thread process the lightweight task of checking whether a query vertex can be mapped to a data vertex. This design provides two key advantages. First, it reduces the per-worker state from $O(|V(Q)| \times d_{\max})$ to $O(|V(Q)|)$, eliminating stack-induced memory bottlenecks and enabling the system to process much larger graphs while supporting more concurrent warps. Second, it decomposes partial-match processing into naturally divisible units, allowing the workload of a single partial match to be distributed across multiple warps or tasks from different partial matches to be combined within a warp.

However, scheduling such lightweight tasks at scale is challenging, because tens of thousands of threads execute concurrently and GPU threads are organized hierarchically into warps and thread blocks. *To address this, we propose a warp-level batch exploration mechanism.* We merge task groups (i.e., the tasks involved in processing a partial match) into a virtual task pool and let threads within a warp pull tasks using two shared pointers that track task boundaries. This mechanism provides minimal memory overhead, high warp utilization, and efficient composability across tasks. Such batch processing is impossible under prior coarse-grained models because each partial match is an indivisible unit; processing multiple partial matches together would require storing large intermediate states, leading to prohibitive memory costs (e.g., STMatch).

Load balancing is essential for efficient subgraph matching on GPUs. Our key observation is that although the root of the search tree may offer limited parallelism, the number of partial matches expands exponentially as the search proceeds, making the workload inherently embarrassingly parallel. When the number of tasks far exceeds the number of workers, load balancing becomes naturally easy because task variability is automatically smoothed out across a large shared task pool. *Building on this observation, we introduce a lightweight load balancing strategy that includes an initialization phase to generate a large pool of partial matches as initial tasks.* This creates abundant initial parallelism and makes load balancing largely self-sustaining, minimizing the need for work stealing and reducing scheduling overhead. Consequently, work stealing becomes a supplement role in gMatch rather than the primary mechanism, as in prior methods.

We evaluate gMatch on two representative workloads: (1) large queries on medium-sized graphs with multiple labels, and (2) small queries on large graphs with few labels. Experimental results show that gMatch significantly outperforms state-of-the-art methods, including STMatch [39], T-DFS [44], and EGSM [35], while offering better scalability. We also compare our approach with systems designed for mining small patterns, including BEEP [24] and G^2Miner [9]. Although these systems outperform our approach on small datasets, our method scales to substantially larger queries and datasets. Due to space limitations, we refer readers to the full version [8] of this paper for complete experimental results.

2 BACKGROUND

In this section, we introduce the background related to this paper.

2.1 Preliminaries

We focus on undirected, labeled graphs $g = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. For a vertex $u \in V$, let $N(u)$ denote its neighbor set and $d(u) = |N(u)|$ its degree. Each vertex is associated with a label via a labeling function $L : V \to \Sigma$, where Σ is the label set. We denote the query graph by Q and the data graph by G. Vertices and edges in Q are referred to as *query vertices* and *query edges*, and those in G as *data vertices* and *data edges*. Frequently used notations are summarized in Table 1. Definition 2.1 introduces *subgraph isomorphism*, which we refer to as a *match*. The goal of *subgraph matching* is to find all matches of Q in G.

Definition 2.1. Given graphs g and g', a subgraph isomorphism is an injective function $M : V(g) \to V(g')$ s.t. 1) $\forall v \in V(g), L(v) = L(M[v])$; 2) $\forall e(v, v') \in E(g), e(M[v], M[v']) \in E(g')$.

Given Q and G in Figure 1, $\{(u_1, v_{58}), (u_2, v_{57}), (u_3, v_1), (u_4, v_{60})\}$ is a match. We can observe that there exist 112 distinct matches of Q in G. The subgraph matching problem requires finding all such mappings from Q to G.

2.2 Coarse-Grained Parallel Execution

Although existing methods apply various optimizations, they share the same core principle: given a query graph Q and a data graph G, they iteratively extend intermediate results along a *matching order* ϕ to find all matches. A matching order ϕ is a permutation of the

Table 1: Frequently used symbols and notations.

Notations	Descriptions
g, Q and G	graph, query graph and data graph
$V(g)$, $E(g)$ and $\Sigma(g)$	vertex set, edge set and label set
$d(u)$, $L(u)$ and $N(u)$	degree, label and neighbor set of u
$e(u, u')$	undirected edge connecting u and u'
$d_{\max}$, d_{avg}	maximum and average degree of the graph
$C_M^F(u)$	feasible candidate set of u given partial match M
$C_M^L(u)$	local candidate set of u given partial match M
ϕ	matching order
$N_+^\phi(u)$	backward neighbors of u given ϕ

query vertices, where $\phi[i]$ denotes the i-th vertex in the sequence ($1 \le i \le |V(Q)|$). For each query vertex u, let $N_+^\phi(u)$ denote the set of neighbors that appear earlier than u in ϕ. We call $N_+^\phi(u)$ the backward neighbors of u. The matching order is required to be *connected*, i.e., $\forall u \in V(Q) \bigwedge u \neq \phi[1], N_+^\phi(u) \neq \emptyset$. Algorithm 1 illustrates the parallel subgraph matching process. Based on their traversal strategies, existing methods can be categorized into BFS-based and DFS-based.

BFS-Search. Lines 1–7 describe a BFS-based approach. It begins by generating an initial set $\mathcal{M}$ of partial matches by mapping the first query edge (based on the matching order ϕ) to data edges that satisfy vertex label constraints. A partial match with i query vertices corresponds to a subgraph isomorphism from the induced subgraph of the first i vertices in ϕ to G. Each $M \in \mathcal{M}$ is then iteratively extended by mapping the next query vertex u to candidate data vertices. At each iteration, $\mathcal{M}$ is processed in parallel, with each warp handling one partial match. Lines 19–26 perform the extension by mapping u to data vertices v that satisfy label and connectivity constraints, i.e., v is adjacent to the data vertices mapped to $N_+^\phi(u)$. The set of such candidates is denoted by $C_M^F(u)$ (i.e., the feasible candidate sets), and the extension is executed in parallel across threads within a warp, leveraging intra-warp parallelism. Once all query vertices are matched, results are reported (Line 7). GpSM [37] and GSI [46] adopt this BFS-based strategy.

DFS-Search. Storing all partial matches at each iteration incurs high memory overhead and limits scalability. To address this, recent methods such as STMatch [39], T-DFS [44], and EGSM [35] employ a depth-first strategy (Lines 8–11), where each warp recursively extends a single partial match by mapping one query vertex at a time (Lines 12–16). This reduces memory usage, as only the current path in the search tree is maintained.

Abstraction. The exploration process builds a search tree where each node represents a partial match M, and its children $\mathcal{M}_M$ are the new partial matches obtained by mapping the next query vertex $u \in \phi$ to feasible candidates in $C_M^F(u)$. The difference lies in the traversal strategy: BFS-based methods explore the tree in a breadth-first manner, while DFS-based methods use depth-first traversal. Despite this difference, both approaches adopt a *coarse-grained parallel model*, where each warp is assigned a partial match as a task and extends it independently. Intra-warp parallelism is employed to compute feasible candidates for extending each partial match.

Take Q and G in Figure 1 as an example. The subgraph matching algorithm first generates a matching order $\phi = (u_1, u_2, u_3, u_4)$ and

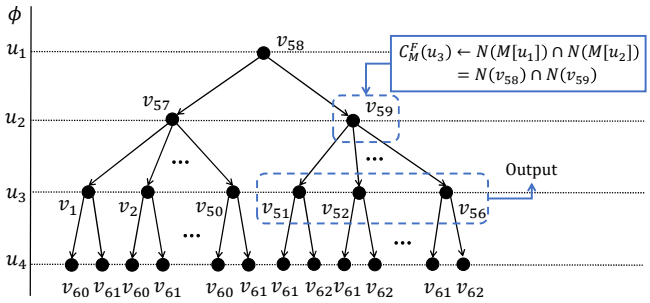


Figure 2: The search tree of coarse-grained parallel execution on the example graphs in Figure 1.

then performs a BFS or DFS search following ϕ, as shown in the search tree (Figure 2). At level i of the tree, the algorithm computes feasible candidates for query vertex $\phi[i]$ based on the current partial match (i.e., the path from the root). The path (v_{58}, v_{59}) corresponds to a match of the subgraph induced by $\{u_1, u_2\}$. To extend this match to u_3, the algorithm computes the intersection $N(v_{58}) \cap N(v_{59})$ and retains only vertices with the same label as u_3. This yields the feasible candidate set $C_M^F(u_3) = \{v_{51}, v_{52}, ..., v_{56}\}$. Note that for GPU-based subgraph matching, the set intersection operation is computed by a warp, where each thread is assigned a vertex in $N(v_{59})$ and checks the existence of this vertex in $N(v_{58})$.

Remark. Our paper, together with prior work [33, 34, 48], identifies two representative workload patterns: (1) large queries over medium-sized graphs, and (2) small queries over large-scale graphs. A detailed discussion on these workloads, including why existing coarse-grained methods struggle to generalize across both efficiently, is deferred to Appendix C of the complete version [8] due to space limitations.

2.3 Issues in Coarse-Grained Parallel Execution

Although DFS-based search significantly reduces memory consumption compared to BFS-based search, we observe that its coarse-grained parallel execution leads to severe performance bottlenecks.

Issue 1: The execution stack incurs high memory overhead, limiting both scalability and computational efficiency. The *execution stack* stores intermediate results during the search process. As shown in Algorithm 1, when processing a partial match M, a warp must store both the feasible candidates $C_M^F(u)$ and the set of derived partial matches $\mathcal{M}_M$. Since the maximum size of $C_M^F(u)$ and $\mathcal{M}_M$ is bounded by the maximum degree $d_{\max}$ of G, the coarse-grained strategy requires a buffer of size $O(d_{\max})$. Dynamic memory allocation based on the actual size of $C_M^F(u)$ and $\mathcal{M}_M$ is impractical on GPUs due to high allocation overhead. Given that the DFS stack depth is $|V(Q)|$, the total space complexity of the execution stack per warp is $O(|V(Q)| \times d_{\max})$.

Real-world graphs typically follow a power-law degree distribution, where a small fraction of vertices have a large number of neighbors. This leads to high memory overhead for the execution stack, limiting scalability and efficiency on GPUs, which offer abundant compute resources but limited memory.

For example, in LDBC, a social network benchmark, $d_{\max}$ reaches 4 million, requiring 16 MB to store feasible candidates per partial

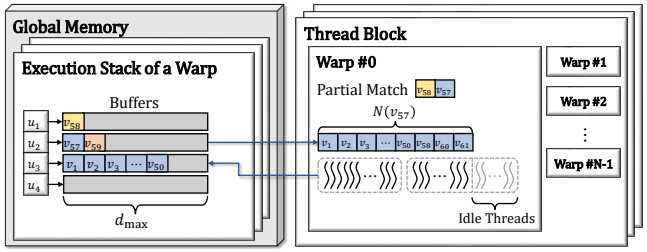
Algorithm 1: Coarse-Grained Parallel Subgraph Matching

```
1  Procedure BFS-Search(Q, G, φ)
2      M ← {{(φ[1], v), (φ[2], v')} | e(v, v') ∈ E(G) ∧ L(φ[1]) =
         L(v) ∧ L(φ[2]) = L(v')};
3      for i ← 3 to |φ| do
4          M' ← M, M ← ∅;
           /* Inter-warp parallelism.                          */
5          parallel foreach M ∈ M' do
6              M ← M ⋃ Process(G, M, φ[i]);
7      Output M;
8  Procedure DFS-Search(Q, G, φ)
9      M ← {{(φ[1], v), (φ[2], v')} | e(v, v') ∈ E(G) ∧ L(φ[1]) =
         L(v) ∧ L(φ[2]) = L(v')};
       /* Inter-warp parallelism.                              */
10     parallel foreach M ∈ M do
11         Search(Q, G, M, φ, 3);
12 Procedure Search(Q, G, M, φ, i)
13     if i = |φ| + 1 then Output M, return;
14     M_M ← Process(G, M, φ[i]);
15     foreach M' ∈ M_M do
16         Search(Q, G, M', φ, i + 1);
17 Procedure Process(G, M, u)
18     C^F_M(u) ← {};
       /* Intra-warp parallelism.                              */
19     C^L_M(u) ← N(M[u']) where u' ∈ N^φ_+(u);
20     parallel foreach v ∈ C^L_M(u) do
21         if L(u) = L(v) ∧ v is not mapped in M then
22             F ← true;
23             foreach u'' ∈ N^φ_+(u) where u'' ≠ u' do
24                 if v ∉ N(M[u'']) then F ← false, break;
25             if F = true then C^F_M(u) ← C^F_M(u) ⋃ {v};
26     return {M ⋃ {(u, v)} | v ∈ C^F_M(u)};
```

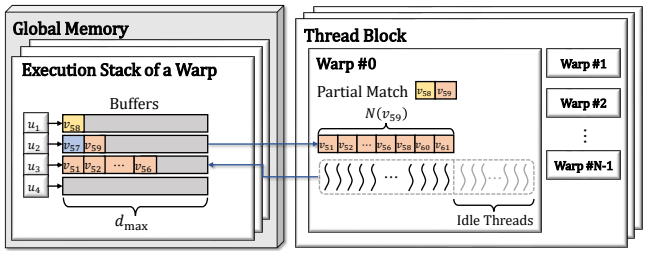
match (assuming 4-byte vertex IDs). On an NVIDIA 4090 GPU with 128 streaming multiprocessors (SMs), each capable of 64 active warps and 24 GB memory, the execution stack quickly becomes a bottleneck. For a query graph with just five vertices and eight active warps per SM, the stack alone would exceed 80 GB of memory. Reducing the number of active warps to save memory significantly underutilizes computational resources. Additionally, the stack must reside in global memory (as it is too large to fit in shared memory), further degrading data access speed.

Figure 3a illustrates the issue of high memory consumption of the execution stack, using the example graph of Figure 1. To extend the partial match $M = \{(u_1, v_{58}), (u_2, v_{57})\}$, the coarse-grained parallel approach computes the intersection $N(v_{58}) \cap N(v_{57})$ and filters out vertices whose label is not $L(u_3)$. The result $C^F_M(u_3) = \{v_1, v_2, ..., v_{50}\}$ is completely computed and materialized in the third level of the stack. The result has 50 vertices and is stored in the preallocated buffer of size $O(d_{\max})$. This preallocated buffer incurs high memory overhead.

Issue 2: The mismatch between the fixed warp size and the dynamic workload causes severe thread underutilization. The



(a) Extending partial match $\{(u_1, v_{58}), (u_2, v_{57})\}$.



(b) Extending partial match $\{(u_1, v_{58}), (u_2, v_{59})\}$.

Figure 3: Coarse-grained parallel execution on the example graphs in Figure 1.

coarse-grained parallel execution assigns one partial match to each warp, where 32 threads execute in lockstep. However, as shown in Lines 17–26 of Algorithm 1, the size of the local candidate set $C^L_M(u)$ (i.e., $N(M[u'])$) varies dynamically during the search. Real-world graphs typically follow a power-law degree distribution, where most vertices have few neighbors. As a result, $|C^L_M(u)|$ is often smaller than the warp size, leading to poor intra-warp parallelism and underutilized threads.

Figure 3b further illustrates this issue, using the example graphs of Figure 1. When extending the partial match $M = \{(u_1, v_{58}), (u_2, v_{59})\}$, the coarse-grained parallel approach assigns each thread within the warp a vertex from $C^L_M(u_3)$, namely $N(v_{59})$. Each lane then check the validity of the vertex and thus construct the feasible candidates for the next level. However, there are only 9 vertices within $N(v_{59})$, and 23 threads within the warp are idle. Moreover, this underutilization problem worsens at the next level. This is because each feasible candidate identified at the current level must then verify its own local candidate set (which is also $N(v_{59})$), further amplifying the inefficiency.

STMatch [39] addresses this issue using *loop unrolling*, which sets a fixed unrolling factor σ (e.g., 2, 4, or 8) to process multiple partial matches per warp by unrolling the loop at Line 15 in Algorithm 1. While loop unrolling reduces the idle rate, it does not fully resolve thread underutilization since σ is fixed while $|C^L_M(u)|$ varies dynamically. More critically, it increases memory consumption by expanding the execution stack to $O(|V(Q)| \times d_{\max} \times \sigma)$, further limiting scalability.

To quantify this thread underutilization issue, we define the *idle rate* as $\frac{1}{|\mathcal{M}|} \sum_{M \in \mathcal{M}} \frac{32 - |C^L_M(u)|}{32}$, where $\mathcal{M}$ is the set of partial matches generated during the search. We generate 100 random 12-vertex query graphs per dataset and obtain the average idle rate.

Table 2: Idle rates with varying loop unrolling sizes.

Unrolling Size	*db*	*en*	*gw*	*gh*	*wt*
1	70.74%	45.14%	40.34%	48.19%	58.80%
2	50.22%	31.38%	38.62%	59.47%	29.82%
4	30.16%	20.75%	22.94%	21.62%	12.49%
8	19.32%	10.84%	15.54%	11.41%	9.80%

Table 2 shows the idle rate across five datasets, revealing severe hardware underutilization (see Table 3 for details of these datasets).
Summary. Existing methods suffer from significant performance limitations due to their coarse-grained parallel execution model on GPUs: (1) high memory overhead and poor scalability, as the total space complexity required for execution stacks grows as $O(|V(Q)|\times d_{\max})$; and (2) severe thread underutilization, as the neighbor sets explored by a warp are often much smaller than the warp size, leaving many threads idle.

3 AN OVERVIEW OF GMATCH

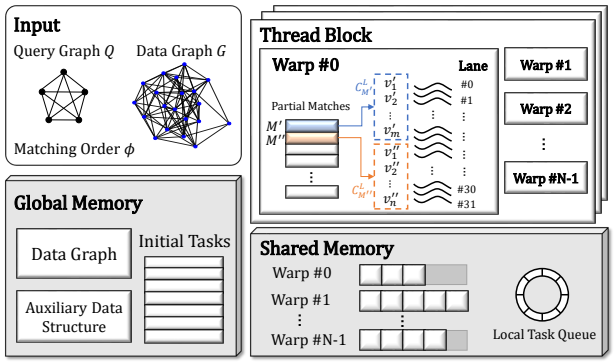


Figure 4: An overview of gMatch.

Figure 4 presents an overview of gMatch, a hardware-efficient subgraph matching approach. Given Q and G, we first generate a matching order ϕ. As matching order selection has been extensively studied and its overhead is negligible, we generate ϕ on the CPU using the RI method [6], which has demonstrated strong performance in prior work [33, 48]. gMatch is compatible with other matching order generation strategies as well.

gMatch focuses on efficient parallel execution on GPUs. Unlike prior coarse-grained parallel model that treat a warp as the basic unit of work, we propose a *fine-grained parallel execution model* that treats each thread as a worker and each candidate validation as a task. This model enables fine-grained processing of partial matches and performs the search on-the-fly without materializing intermediate results, significantly reducing execution stack space and eliminating scalability constraints. Building on this model, we introduce a *warp-level batch exploration* technique that groups multiple partial matches into a single warp. This aligns with the GPU architecture, where a warp is the basic scheduling unit. The technique eliminates thread underutilization without incurring additional memory overhead. To further improve scalability, we design a *lightweight load balancing strategy* based on the observation that subgraph matching is an embarrassingly parallel problem. At the start of execution, we perform a BFS-based exploration to generate partial matches until a specified threshold is reached, forming an initial task pool. Warps fetch tasks independently from this pool. Once the pool is exhausted, idle warps perform work stealing to maintain balance. Thanks to the large initial task pool, stealing is rarely triggered, keeping scheduling overhead low. We detail each of these components in the following sections.

4 EFFICIENT PARALLEL EXECUTION

In this section, we present the efficient parallel execution strategy employed in gMatch.

4.1 Fine-Grained Parallel Execution

Given Q, G, and ϕ, existing methods [35, 39, 41, 44] treat each partial match M as a parallel task and assign it to a warp. Suppose that M contains i matched vertices and u is the $(i+1)$-th query vertex in ϕ. The warp extends M by mapping u to candidates v in the feasible set $C^F_M(u)$. To store $C^F_M(u)$ and the resulting partial matches, each warp allocates a buffer of size $O(d_{\max})$. As a result, a worker must maintain an execution stack of size $O(d_{\max}\times|V(Q)|)$, leading to significant memory overhead as discussed in Section 2.3.

To address these limitations, we design a *fine-grained execution model*. Instead of assigning a partial match M to a warp as a monolithic task, we decompose it into smaller and independent tasks of the form $T_M(u,v)$, representing an attempt to extend M by mapping u to a candidate vertex v. A task is valid if $v \in N(M[u'])$ for all $u' \in N^\phi_+(u)$. To reduce candidate enumeration, we further restrict attention to a local candidate set $C^L_M(u) = N(M[u'])$, where $u' = \arg\min_{u'\in N^\phi_+(u)} |N(M[u'])|$, i.e., the neighbor set of the mapped backward query vertex with the fewest neighbors. Thus, we define the corresponding set of tasks as the task group for M, denoted by $T^G_M(u) = \{T_M(u,v) \mid v \in C^L_M(u)\}$.

In GPUs, threads are the fundamental execution units, while warps serve as the basic scheduling units. Our execution model assigns each task group $T^G_M(u)$ to a warp, and distributes its tasks $T_M(u,v)$ to individual threads within the warp. Each thread independently processes its assigned task by verifying the feasibility of mapping u to v. After task execution, the warp gathers the valid candidates and generates the corresponding extended partial matches. Figure 5 illustrates the search tree in fine-grained parallel execution, where each edge represents a task processed by a thread.

Because threads in a warp execute in lockstep, distributing tasks and gathering results incurs minimal synchronization overhead. Unlike traditional approaches storing all child matches, this fine-grained execution extends M on the fly (expanding at most 32, the warp size) and avoids materializing all child matches at once and significantly reduces the space required for the execution stack.

Figure 6 illustrates the fine-grained parallel execution model of gMatch. We use blue to denote the subtree rooted at v_{57} and red for the subtree rooted at v_{59}. While a warp attempts to extend the partial match $M = \{(u_1, v_{58}), (u_2, v_{57})\}$, the intersection $N(v_{57}) \cap N(v_{58})$ will be computed. As $|N(v_{57})|$ is smaller than $|N(v_{58})|$, we say $C^L_M(u_3) = N(v_{57})$ and check the validity of each vertex within

$C_M^L(u_3)$. Instead of computing and materializing all feasible candidates, the warp only check the validity of the first 32 vertices in $C_M^L(u_3)$. All valid candidates among them (i.e., $v_1, v_2, ..., v_{32}$) are stored in the buffer of the next level. The unexplored candidates in $C_M^L(u_3)$ will be checked during backtracking.

In summary, this design aligns naturally with the GPU's architecture and offers two key benefits: (1) it reduces space overhead, as each thread handles only one candidate and does not need to store the feasible set or generate multiple partial matches; and (2) it enables fine-grained task divisibility and flexible scheduling across threads, the fundamental execution units on GPUs.

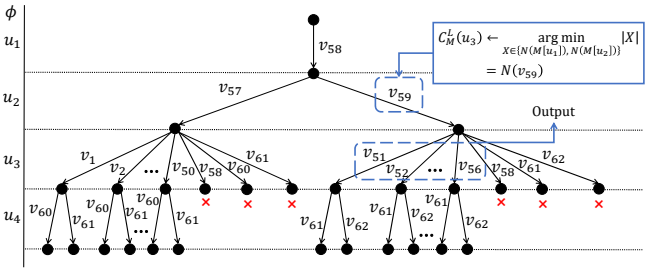


Figure 5: The search tree of fine-grained parallel execution on the example graph in Figure 1.

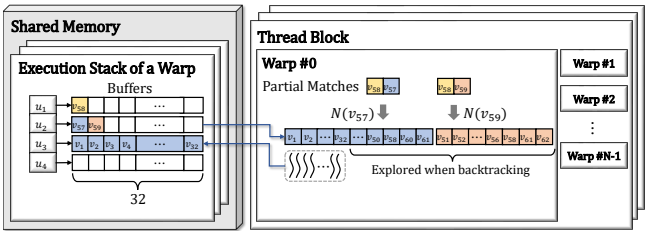


Figure 6: Fine-grained parallel execution on the example graphs in Figure 1.

Remark. Existing GPU-based methods, such as STMatch, T-DFS, and G^{2}Miner, parallelize set intersection at the warp level using SIMT execution. Specifically, each thread processes one element from a set and checks its presence in the other set via binary search. This constitutes "fine-grained" parallelism within a single intersection. However, this strategy follows a coarse-grained execution model: a set intersection is treated as an indivisible unit, and all resulting candidates must be materialized in an array. As a result, each warp is assigned to extend a single partial match and enumerate all its feasible candidates, which leads to the issues described in Section 2.3. In contrast, our fine-grained computation model decomposes the extension of a partial match into independent per-candidate validation tasks and assigns each task to a single thread. This allows a warp to simultaneously process tasks from multiple partial matches, or to distribute the processing of a single partial match across multiple warps, while significantly reducing memory overhead and improving the warp utilization.

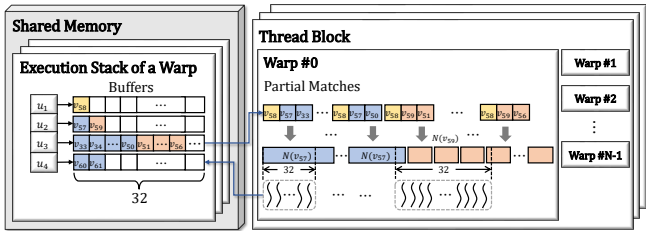


Figure 7: Warp-level batch exploration.

4.2 Warp-Level Batch Exploration

Real-world graphs typically follow a power-law degree distribution, where most vertices have low degree. Moreover, for labeled graphs and complex queries, advanced filtering techniques [35, 46] further reduce the number of candidates. As a result, the local candidate set of a partial match is often small, and its task group cannot fully utilize all threads in a warp, leading to underutilized GPU resources. Experimental results in Section 2.3 confirm the severity of this issue.

To address this, we propose *warp-level batch exploration* on top of the fine-grained execution model. This model enables task divisibility and flexible scheduling across threads, making it natural to process multiple task groups in a batch. Instead of assigning a single task group to each warp, we unify multiple task groups into a shared virtual task pool. Each task group $T_M^G(u)$ is defined by a partial match M and its local candidate set $C_M^L(u)$. During exploration, a warp maintains up to 32 partial matches at each level, with each thread responsible for one candidate mapping. These task groups collectively form a unified task pool accessible to all threads in the warp.

Threads retrieve tasks from the pool using two lightweight pointers that track the current partial match and candidate position. This design incurs negligible memory overhead while significantly improving warp utilization without sacrificing memory efficiency. Materializing partial matches at each level can cause significant memory pressure on shared memory, with space complexity up to $O(|V(Q)|^2)$. To address this, we construct partial matches on-the-fly by traversing the execution stack from bottom to top during the computation. Since each mapping from a query vertex to a feasible candidate defines a partial match, we store, for each candidate at level $i+1$, a pointer to its corresponding candidate at level i. This compact representation enables efficient reconstruction of partial matches with minimal memory overhead.

Figure 7 illustrates the concept of warp-level batch exploration. This occurs during the backtracking phase of what is shown in Figure 6. To complete the tasks at level 2, the warp simultaneously processes the remaining part of $N(v_{57})$ and the entire $N(v_{59})$, treating each as a distinct batch. Since the number of unexplored candidates is fewer than 32, they can be processed within a single warp without requiring further backtracking. The feasible candidates for these two paths are stored at level 3 of the stack, denoted in blue and red, respectively. To extend these partial matches, the warp iterates through their local candidate sets using a fixed width of 32. For example, when processing the first $N(v_{57})$, the warp verifies the first 32 candidates and reports two final matches. The benefit of batch processing is more pronounced when processing

$N(v_{59})$. To minimize thread idling, the warp batches candidates across multiple instances of $N(v_{59})$ together. This design inherently mitigates thread idling, as processing $N(v_{59})$ in a coarse-grained manner would lead to severe underutilization. Importantly, such batch processing is not supported by prior coarse-grained models, where each partial match is treated as an indivisible task. Processing multiple partial matches together in those models would require significant memory for storing all candidate sets and intermediate states, e.g., the loop unrolling technique in STMatch [39].

4.3 Lightweight Load Balancing

The search space contains an exponential number of partial matches, each forming a task group of fine-grained tasks. This leads to an embarrassingly parallel workload. In our fine-grained execution model, intra-warp load balancing is naturally achieved through a unified task pool shared by threads within a warp. We therefore focus on load balancing across warps within a thread block and across thread blocks. Warps in the same thread block can coordinate via shared memory, while communication across blocks requires global memory. To fully utilize GPU resources, we design a lightweight two-phase load balancing mechanism: (1) an initialization phase that prepopulates an initial task pool to avoid early-stage underutilization, and (2) a dynamic balancing phase that redistributes work at runtime when imbalances occur.

Initialization Phase. Although the total search space is large, the number of partial matches at the root level (i.e., first-level nodes in the search tree) may be small, leading to insufficient initial parallelism. To mitigate this, we first perform a breadth-first traversal of the search tree and accumulate partial matches into an initial task pool until the number of partial matches reaches a threshold τ. We then launch the fine-grained parallel execution, where each warp fetches a partial match from the pool using an atomic counter when it runs out of work.

The parameter τ is critical because it regulates the need for dynamic load balancing. When the number of initial partial matches significantly exceeds the number of available warps (typically hundreds to thousands on modern GPUs), load balancing becomes naturally easy because task variability is automatically smoothed out across a large shared task pool, reducing the necessity for explicit load-balancing mechanisms. This principle makes it straightforward to choose τ empirically: a larger value leads to better workload balance but comes with higher memory consumption. In our experiments, we set τ to 10^6, achieving a good balance between memory consumption and efficiency. The experiments on the choice of τ are presented in Appendix B of the complete version [8].

Dynamic Balancing Phase. Once the initial task pool is exhausted, we shift to dynamic load redistribution. We follow the same high-level idea as STMatch [39], idle warps receive half of the work from busy warps by splitting the execution stack of the selected busy warp, but our implementation is simpler and better aligned with our two-level load balancing.

Specifically, when a warp runs out of tasks, it marks itself as idle in shared memory and enqueues its warp ID into a concurrent queue maintained per thread block. Each block also maintains a status array to track warp activity. Busy warps periodically check the queue before each recursive call. If a busy warp dequeues an idle warp ID, it splits its current task stack in half and assigns the split portion to the idle warp, updating the status accordingly. Concurrently, idle warps continuously poll their status in the array. Once an idle warp detects its status change to active, it resumes execution with the newly assigned tasks. This avoids the costly stack scanning in STMatch and reduces contention, even though the busy warp selected may not always have the largest remaining workload. To support inter-block load balancing, the same mechanism is replicated at the block level using global memory for the queue and status array. In summary, by exploiting the intrinsic parallel structure of subgraph matching, our design makes work stealing a supplementary rather than primary mechanism for achieving load balancing.

5 IMPLEMENTATION AND COST ANALYSIS

In this section, we first introduce the implementation and then have a discussion on the cost of gMatch.

5.1 Implementation Details

Algorithm 2 presents the details of gMatch. We begin by performing a BFS to generate partial matches and populate the initial task pool, up to a threshold τ. Each warp executes independently using an execution stack S, which is a 2D array of size $|V(Q)| \times 32$. Each element $S[i][j]$ represents the j-th thread's state at depth i and contains four fields: (1) v: a candidate data vertex, (2) F: a flag indicating whether mapping $\phi[i]$ to v is valid, (3) pid: the index of the parent vertex in $S[i-1]$ corresponding to $\phi[i-1]$, (4) C: a pointer to the local candidate set for the current partial match.

Upon retrieving a partial match M from the task pool, we initialize the execution stack S and begin the search (Lines 2–4). Let l denote the current recursion depth, where $\phi[l]$ is the query vertex to be extended. At Line 7, we generate the *virtual task pool*. As shown in Lines 17–24, each thread traverses the execution stack to reconstruct its corresponding partial match on the fly without materializing it and computes the local candidate set for $\phi[l]$. Each resulting task group forms part of the virtual task pool, enabling fine-grained scheduling at the warp level.

Each thread in the warp then retrieves a task from the virtual task pool (Lines 9–10). To avoid synchronization overhead, a designated leading thread assigns one task to each thread in the warp (Lines 25–32). Since a warp has only 32 threads, this assignment incurs negligible cost. Each thread then processes its assigned task by checking whether the candidate data vertex can extend the partial match. This involves verifying the label constraint, ensuring the vertex has not been used, and checking the edge constraints (Lines 33–41). If the generated partial match reaches the length of the matching order, the result is output (Lines 12–13). Otherwise, we use a warp-level collective primitive to check whether any thread has a feasible candidate; if so, the search continues (Lines 15–16).

5.2 Cost Analysis

We now analyze the space and time complexity of our parallel execution model and highlight its advantages over existing methods.

Space Cost. Since all methods take the same input and produce the same output, we focus on the additional memory used during execution, specifically, the space required for the execution stack

Algorithm 2: gMatch

Input : Query graph Q, data graph G, matching order ϕ, and initial task pool size τ.
Output : All matches from Q to G.

$\mathcal{M} \leftarrow$ generate τ partial matches with BFS-Search;
/* Inter-warp parallelism */
parallel foreach $M \in \mathcal{M}$ **do**
  Initialize execution stack S of a warp with M;
  Search($\phi, S, |M|$);
Procedure Search(ϕ, S, l)
  $i \leftarrow 0, j \leftarrow 0, k \leftarrow 0$;
  /* Each lane processes an element. */
  GenerateTask($\phi, S, l, LANE\_ID$);
  while *true* **do**
    if $LANE\_ID = 0$ **then**
      $(i, j, k) \leftarrow$ScatterTask(S, l, i, j, k);
    /* Each lane processes a task. */
    $S[l][LANE_ID].F \leftarrow$Process($\phi, S, l, k, LANE\_ID$);
    if $l = |\phi| - 1$ **then**
      if $S[l][LANE\_ID].F = true$ **then** Output;
    else
      $F \leftarrow$ ballot_sync(0xFFFF FFFF, $S[l][LANE\_ID].F$);
      if $F \neq 0$ **then** Search($\phi, S, l + 1$);
Procedure GenerateTask(ϕ, S, l, lid)
  $S[l][lid].C \leftarrow \emptyset, C \leftarrow \emptyset, pid \leftarrow lid$;
  if $S[l-1][lid].F = false$ **then return**;
  for $i \leftarrow l - 1$ *to* 0 **do**
    $v \leftarrow S[i][pid].v, pid \leftarrow S[i][pid].pid$;
    if $\phi[i] \in N^{\phi}_{+}(\phi[l])$ *and* $(|N(v)| \leq |C|$ *or* $C \neq \emptyset)$ **then**
      $C \leftarrow N(v)$;
  $S[l][lid].C \leftarrow C$;
Procedure ScatterTask(S, l, i, j, k)
  while $i < 32$ **do**
    while $j < |S[l][i].C|$ **do**
      $S[l][k].v \leftarrow S[l][i].C[j], S[l][k].pid \leftarrow i$;
      $j \leftarrow j + 1, k \leftarrow k + 1$;
      if $k = 32$ **then return** (i, j, k);
    $i \leftarrow i + 1, j \leftarrow 0$;
  return (i, j, k);
Procedure Process(ϕ, S, l, k, lid)
  if $lid < k$ **then return** $false$;
  $u \leftarrow \phi[l], v \leftarrow S[l][lid].v, pid \leftarrow S[l][lid].pid$;
  if $L(v) \neq L(u)$ **then return** $false$;
  for $i \leftarrow l - 1$ *to* 0 **do**
    $u' \leftarrow \phi[i], v' \leftarrow S[i][pid].v, pid \leftarrow S[i][pid].pid$;
    if $v = v'$ *or* $(u' \in N^{\phi}_{+}(u)$ *and* $v \notin N(v'))$ **then**
      return $false$;
  return *true*;

and load balancing structures. As the warp is the basic scheduling unit on GPUs, we analyze space usage at the warp level. Let W be the number of threads per warp, and B the number of warps per thread block. The execution stack depth equals $|V(Q)|$, and each thread maintains one candidate per level. Thus, the stack occupies $O(W \times |V(Q)|)$ space per warp. Given that $|V(Q)|$ is small (e.g., 12), the execution stack requires only a few kilobytes and comfortably fits in shared memory. As a result, shared memory usage is minimal and does not limit warp occupancy.

For load balancing, the local task queue within a thread block stores at most B entries (one per warp), requiring $O(B)$ space—also small enough for shared memory. The global task queue, used for inter-block coordination, is allocated in global memory. Its size is proportional to the number of blocks (typically hundreds), which is negligible compared to the available global memory (usually tens of gigabytes).

Time Cost. The total time cost is determined by the size of the search space and the efficiency of processing each partial match. Given Q, G, and ϕ, the search space is fixed. We thus focus on analyzing the time to process a single partial match M.

Let u be the current query vertex to match. For each candidate $v \in C^L_M(u)$, feasibility checking involves two steps: (1) verifying that v is a neighbor of all data vertices mapped to u's backward neighbors, i.e., for each $u' \in N^{\phi}_{+}(u)$, we check whether $v \in N(M[u']))$ via binary search, costing $O(\log |N(M[u'])|)$; (2) checking whether v has already been used in M, which requires scanning the execution stack in $O(|M|)$ time. We fuse these checks into a single loop to improve efficiency. Thus, the total cost of processing M is: $O(\sum_{v \in C^L_M(u)} (|M| + \sum_{u' \in N^{\phi}_{+}(u)} \log |N(M[u'])|))$.

This cost is comparable to that of existing methods, but our approach achieves significantly better hardware utilization through fine-grained parallelism and warp-level batch exploration. The overhead of the scatter operation is negligible, as a warp processes at most 32 candidates.

Advantages over Existing Methods. Compared with existing methods, our approach does not reduce the size of the search space or the time complexity of processing a partial match. Instead, it focuses on maximizing hardware utilization. First, our method reduces the space complexity of the execution stack per warp from $O(|V(Q)| \times d_{\max})$ to $O(|V(Q)|)$. This removes memory-related constraints on parallel execution, ensuring that the number of concurrent warps is not limited by stack size while also freeing substantial memory to handle larger graphs. Moreover, the stack can reside entirely in shared memory, improving access efficiency. Second, warp-level batch exploration enables full utilization of computing resources within a warp without incurring significant overhead, even when task groups are small. Finally, we employ a lightweight load balancing strategy that uses an initialization phase to prepopulate tasks, reducing the need for work stealing during execution.

6 EXPERIMENTS

In this section, we present the experimental settings and the results.

6.1 Experimental Setup

Methods Under Study. We compare **gMatch** with the latest GPU-based subgraph matching algorithms, including **STMatch** [39], **EGSM** [35], and **T-DFS** [44]. We also include two systems designed for mining small patterns: **BEEP** [24] and **G^{2}Miner** [9].

Their source code is publicly available on GitHub[1,2,3,4,5]. To ensure fair comparison, we unify the matching order in STMatch and T-DFS by replacing their original strategies with the RI [6] ordering, consistent with our method. For EGSM, BEEP, and G^{2}Miner, we use their default ordering, which is specifically optimized for each framework. EGSM leverages a candidate graph for efficient candidate filtering and retrieval, which is particularly effective for workloads with large query graphs and medium-sized data graphs. We adopt the same candidate graph mechanism in gMatch when comparing against EGSM. However, we disable this feature when comparing against STMatch, T-DFS, BEEP, and G^{2}Miner, as they do not use candidate graphs and gain minimal benefit from candidate graphs for their target workloads. Our method's compatibility with candidate graphs highlights its generality.

Testbed. All competing algorithms are compiled with GCC 10.5 and NVCC 12.5. By default, we run our experiments on a Linux machine with an Intel Xeon Gold 6430 CPU and 256 GB main memory. The machine is equipped with an NVIDIA RTX 4090 GPU with 128 SMs and 24 GB global memory space.

Datasets. We use 11 graph datasets in our experiments, as summarized in Table 3. These include eight real-world social networks (*en*, *gh*, *gw*, *wt*, *lj*, *pk*, *fr*, and *ot*), one co-authorship network (*db*), and two synthetic datasets (*ld* and *rm*). The dataset *ld* is from the LDBC social network benchmark [13]. We also use PaRMAT [26] to generate an RMAT graph *rm* because it can imitate the structure of real-world networks and allows us to control the graph scale to meet the memory constraints of a single GPU. The remaining datasets are from SNAP [29]. We choose the datasets *en*, *gh*, *gw*, *wt*, and *db* for comparison with EGSM, as they were also used in its experiments. These datasets are obtained directly from the EGSM authors' release. Additionally, we include the datasets *pk*, *fr*, *ot*, and *lj* for comparison with STMatch and T-DFS, following the same datasets used in their original experiments. We choose the dataset *ld* as it is a popular benchmark for graph query and processing.

We categorize the datasets based on their edge count: datasets with fewer edges belong to the first category, and those with more to the second. We use large query graphs on the first category and small query graphs on the second. All data graphs in the first category are labeled, where each vertex is assigned with a label uniformly at random. All datasets in the second category are unlabeled (i.e., $|\Sigma| = 1$).

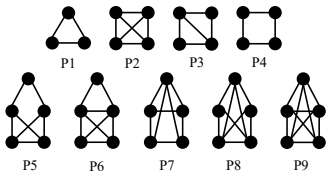


Figure 8: Query graphs.

Query Sets. We evaluate the performance of subgraph matching algorithms across different query graph sizes. For small query graphs ($|V(Q)| \leq 5$), we use the predefined query graphs shown in Figure 8. We consider 9 patterns which are commonly used in existing work [9, 20, 34, 44] for performance evaluation. All of them are unlabeled and will be used for testing on unlabeled data graphs. For larger query graphs, we generate 100 random queries per dataset using the same procedure as in EGSM and other prior work [5, 17, 34]. Specifically, to generate an n-vertex query graph, we start with a random seed vertex from the data graph and iteratively expand it by adding random neighboring vertices, along with all their connecting edges to the existing subgraph, until the target size of n vertices is reached.

[1]https://github.com/HPC-Research-Lab/STMatch (commit: f98462a)
[2]https://github.com/RapidsAtHKUST/EGSM (commit: de854d5)
[3]https://github.com/lyuheng/tdfs (commit: 05ef1b1)
[4]https://github.com/Nagi-Research-Group/BEEP (commit: 3eca170)
[5]https://github.com/chenxuhao/GraphMiner (commit: 2a76e3f)

Table 3: The detailed statistics of datasets.

Datasets	Abbr.	$\|V\|$	$\|E\|$	$\|\Sigma\|$	d_{avg}	d_{max}
Enron	*en*	36,692	183,831	16	10.0	1,383
GitHub	*gh*	37,700	289,003	16	15.3	9,458
Gowalla	*gw*	196,591	950,327	16	9.7	14,730
DBLP	*db*	317,080	1,049,866	16	6.6	343
WikiTalk	*wt*	2,394,385	4,659,565	64	3.9	100,029
Pokec	*pk*	1,632,803	22,301,964	1	27.3	14,854
LiveJournal	*lj*	4,847,571	42,851,237	1	17.7	14,815
Orkut	*ot*	3,072,441	117,185,083	1	76.3	33,313
LDBC	*ld*	29,982,730	175,860,387	1	11.8	4,282,595
Friendster	*fr*	65,608,366	1,806,067,135	1	55.1	5,214
RMAT	*rm*	99,999,983	2,800,000,000	1	56.0	112,497

Metrics. The total query time t_{query} for a query is divided into filtering time $t_{\text{filtering}}$, data transfer time (t_{transfer}), and search time (t_{search}). We define t_{query} as the time from when G and Q are loaded into host memory until all results are found. $t_{\text{filtering}}$ measures the time for candidate filtering and auxiliary structure construction, t_{transfer} captures the time to move G, Q, and auxiliary data to GPU memory, and t_{search} accounts for the time to enumerate all matches. To ensure fair comparison, we report only t_{search}. This is because, for large queries, gMatch and EGSM share the same auxiliary structures, so any difference lies solely in t_{search}. For small queries, STMatch performs degree-based filtering on the CPU, which becomes a bottleneck on large datasets (e.g., *ot*), whereas gMatch and T-DFS implement the same filtering on the GPU, incurring negligible cost. Therefore, t_{search} provides a fair metric for comparison.

To ensure all queries complete in a reasonable time, we set timeouts: 3 hours for small queries and 30 minutes for large queries. Queries exceeding the limit are marked as unsolved, and we report both the number of unsolved queries and the average search time. We also measure speedup using the following formula:

$$\text{Speedup}(\mathcal{A} \text{ over } \mathcal{B}) = \frac{1}{|\mathcal{Q}|} \sum\nolimits_{Q \in \mathcal{Q}} \frac{t_{\mathcal{B}}(Q)}{t_{\mathcal{A}}(Q)},$$

where $\mathcal{Q}$ is the set of queries, and $t_{\mathcal{X}}(Q)$ is the search time of algorithm $\mathcal{X}$ on query Q. We also evaluate GPU memory consumption. Since G occupy the same amount of memory across all methods, we focus on measuring the space consumed by the execution stack.

6.2 Overall Comparison

Small Query. We evaluate gMatch against STMatch, T-DFS, BEEP and G^{2}Miner on the six large datasets. EGSM is excluded since it targets small data graphs and frequently runs out of memory

Table 4: Comparison of search time (millisecond) among T-DFS (TD), STMatch (ST), BEEP (BE), G^{2}Miner (G2), and gMatch (GM). OOT indicates timeouts, OOM indicates out-of-memory, and N/A indicates unsupported queries in the open-source code.

	pk					*lj*					*ot*				
	TD	ST	BE	G2	GM	TD	ST	BE	G2	GM	TD	ST	BE	G2	GM
P1	84	9,030	129	**12**	23	112	12,413	239	87	**31**	463	12,486	661	**92**	122
P2	431	8,566	172	**22**	86	2,083	4,344	350	**116**	422	5,601	9,587	840	**190**	1,077
P3	987	3,667	229	**32**	323	5,665	11,991	562	**118**	2,080	19,244	43,329	1,380	**299**	7,661
P4	5,473	15,134	N/A	**336**	1,626	12,932	29,071	N/A	**885**	4,127	OOT	285,055	N/A	**9,730**	47,113
P5	13,429	12,549	N/A	N/A	**4,390**	926,807	641,476	N/A	N/A	**334,373**	1,234,177	4,153,864	N/A	N/A	**526,114**
P6	29,846	10,260	**773**	N/A	7,935	3,865,506	509,852	**16,875**	N/A	637,667	2,568,786	1,449,627	**9,600**	N/A	807,067
P7	13,225	5,916	6,910	N/A	**3,295**	842,156	259,700	**23,703**	N/A	305,968	976,353	848,283	**134,389**	N/A	322,477
P8	6,223	4,569	**587**	N/A	1,646	520,329	153,237	**12,218**	N/A	221,493	219,505	267,167	**6,268**	N/A	104,190
P9	1,754	8,177	346	**55**	288	106,059	20,799	**3,995**	4,347	25,114	46,943	36,807	2,832	**825**	7,962
	ld					*fr*					*rm*				
	TD	ST	BE	G2	GM	TD	ST	BE	G2	GM	TD	ST	BE	G2	GM
P1	571	OOM	OOM	**45**	172	OOM	OOM	OOM	**1,224**	4,354	OOM	OOM	OOM	**2,551**	5,714
P2	2,149	OOM	OOM	**83**	549	OOM	OOM	OOM	**4,090**	12,132	OOM	OOM	OOM	**3,784**	5,866
P3	12,387	OOM	OOM	OOM	**5,844**	OOM	OOM	OOM	OOT	**53,556**	OOM	OOM	OOM	OOM	**51,243**
P4	377,456	OOM	N/A	205,991	**152,834**	OOM	OOM	N/A	OOT	**2,806,445**	OOM	OOM	N/A	OOT	**3,033,834**
P5	315,378	OOM	N/A	N/A	**149,406**	OOM	OOM	N/A	N/A	**1,068,031**	OOM	OOM	N/A	N/A	**266,772**
P6	439,141	OOM	OOM	N/A	**167,778**	OOM	OOM	OOM	N/A	**1,519,540**	OOM	OOM	OOM	N/A	**225,545**
P7	153,001	OOM	OOM	N/A	**95,427**	OOM	OOM	OOM	N/A	**731,884**	OOM	OOM	OOM	N/A	**138,211**
P8	19,294	OOM	OOM	N/A	**10,191**	OOM	OOM	OOM	N/A	**235,798**	OOM	OOM	OOM	N/A	**42,032**
P9	3,828	OOM	OOM	**114**	910	OOM	OOM	OOM	**4,643**	33,776	OOM	OOM	OOM	**3,801**	6,108

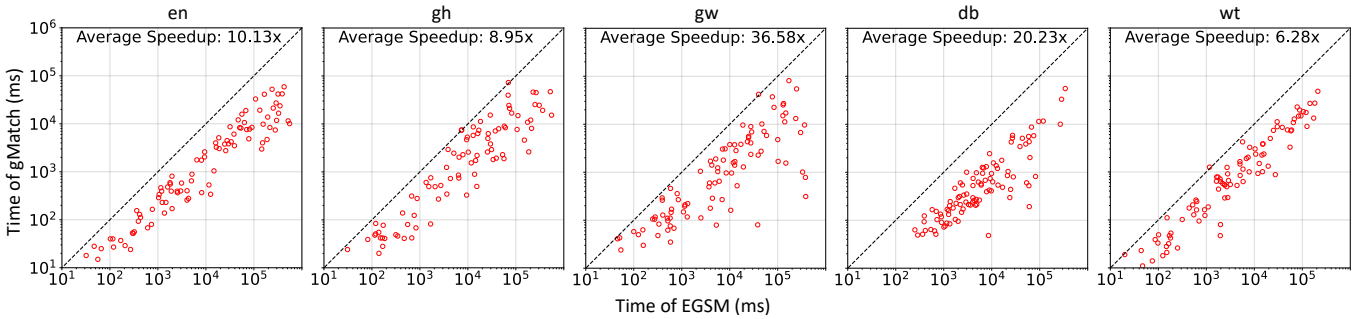


Figure 9: Comparison of search time between EGSM and gMatch (millisecond).

on these datasets. The result is shown in Table 4. For fairness, we disable auxiliary structures and BFS in gMatch, comparing only DFS-based search time with the baselines. gMatch is the only method that completes all queries across all datasets. In contrast, the baselines encounter OOM or OOT failures on large graphs (e.g., *fr* and *rm*) or graphs with high maximum degree (e.g., *ld*), demonstrating the superior scalability and generality of our approach.

STMatch frequently runs out of memory because it allocates a buffer of size $d_{\max}$ at each stack level. In particular, it fails on *ld*, where $d_{\max}$ exceeds 4 million, and on *fr* and *rm*, where the remaining memory after stack allocation is insufficient. T-DFS is consistently slower than gMatch due to the overhead of its page-table-based stack management. It also fails on *fr* and *rm* because representing data edges as arrays of edge pairs nearly doubles the memory footprint of the data graph, exceeding available memory.

G^{2}Miner is a graph pattern mining system that supports different tasks such as motif discovery and frequent subgraph mining, with subgraph matching as a core operation. Given a query, it employs the AutoMine compiler [32] to optimize execution. However, the open-source release does not include the compiler and supports only a limited set of query patterns. As a result, our evaluation is restricted to a subset of the patterns in Figure 8 (P1–P4 and P9). G^{2}Miner performs well on small graphs (e.g., *pk*, *lj*, and *ot*), but encounters OOM and OOT failures on large graphs.

BEEP supports patterns with a single "central" vertex connected to all others (e.g., P1–P3 and P6–P9 in Figure 8) because it accelerates set intersection by constructing an adjacency matrix of size $O(d_{\max}^2)$ for the neighborhood of the matched central vertex. This matrix-based optimization enables good performance on P6–P9. However, it does not scale to large graphs and fails on *ld*, *fr*, and *rm*. Although the source code provides an option to disable this optimization, doing so leads to incorrect results. In summary, gMatch demonstrates superior scalability and broader applicability than existing methods.

Large Query. To evaluate performance on larger query graphs, we conduct our experiments on *en*, *gh*, *gw*, *db*, and *wt*, as they are used in EGSM's experiments. We compare only against EGSM, since the open-source implementations of the other methods do not support arbitrary queries with tens of vertices. For each dataset, we generate a query set consisting of 100 random 12-vertex query graphs using the method described in Section 6.1. Figure 9 demonstrates the result, where each red dot denotes a test case. Red dot below

the diagonal represents one test case where gMatch is faster than EGSM. Experiments result shows that gMatch outperforms EGSM on all datasets, achieving 36.58× average speedup on *gw* and 20.23× average speedup on *db*. This speedup mainly comes from improved warp utilization enabled by the fine-grained parallel execution and warp-level batch exploration.

Memory Consumption. We compare gMatch's execution-stack memory consumption with that of STMatch and T-DFS. Figure 10 shows the maximum memory consumption of three algorithms on each dataset, where each OOM denotes an out-of-memory error. We can see that STMatch incurs severe memory overhead. While T-DFS effectively reduces memory cost using a page table, it still consumes more memory than gMatch. gMatch minimizes the stack size to fit within GPU shared memory, requiring no additional global memory for the execution stack.

Table 5: Search time (millisecond) for all patterns across datasets. Each value represents the average search time over the *lj*, *ot*, *pk*, and *fr* datasets.

	Naive	Unroll-4	Unroll-8	Fine (naive)	Fine (batch)
P1	1,355	1,277	1,221	1,249	**1,132**
P2	4,334	4,098	3,917	4,058	**3,429**
P3	20,284	19,022	18,443	18,855	**15,905**
P4	935,433	877,175	849,963	869,191	**714,827**
P5	617,268	578,856	560,772	573,618	**483,227**
P6	924,871	866,658	839,577	858,787	**743,052**
P7	418,832	392,459	380,196	388,898	**340,906**
P8	186,048	174,420	168,945	174,327	**140,781**
P9	21,460	20,106	19,487	19,922	**16,785**

6.3 Evaluation of Individual Techniques

We evaluate the effectiveness of fine-grained parallel execution and warp-level batch exploration. We compare the performance of all query graphs from Figure 8. There are five methods under study: (1) Naive coarse-grained model; (2) Coarse-grained model with loop unrolling size of 4; (3) Coarse-grained model with loop unrolling size of 8; (4) Fine-grained model without batch exploration; and (5) Fine-grained model with batch exploration. We evaluate the performance of the five methods across six datasets: *lj*, *ot*, *ld*, *pk*, *fr*, and *rm*. The average search time for each query is reported in Table 5. Note that *ld* and *rm* are excluded from the average time calculation because the Naive, Unroll-4, and Unroll-8 methods fail on these datasets.

The experimental results show that the fine-grained model with batch exploration consistently achieves the best performance. Only our fine-grained methods successfully handle the *ld* and *rm* datasets, which have high maximum degrees. All coarse-grained approaches fail on these challenging datasets due to out-of-memory errors.

To evaluate the effectiveness of fine-grained parallel execution on large queries over medium-sized data graphs, we measure the average speedup on random query sets of 12 vertices (Figure 11). The baseline is the coarse-grained parallel approach without loop unrolling. Our batch exploration strategy achieves significant speedup

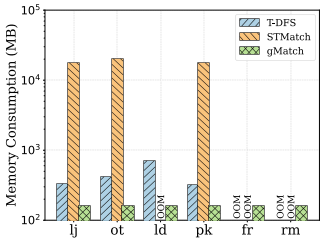


Figure 10: Memory consumption of the execution stacks during DFS.

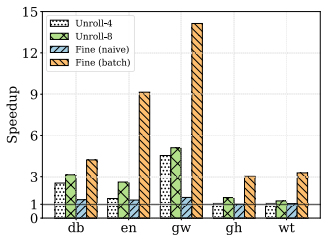


Figure 11: Speedup comparison vs. naive coarse-grained parallel execution.

Table 6: Comparison of idle rates between gMatch and the baseline STMatch under its different loop unrolling configurations (naive, unroll-2, unroll-4, unroll-8).

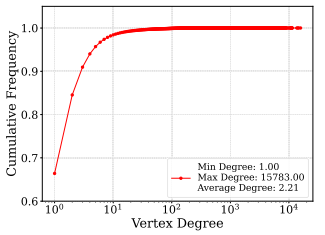
	db	*en*	*gw*	*gh*	*wt*
Naive	70.74%	45.14%	40.34%	48.19%	58.80%
Unroll-2	50.22%	31.38%	38.62%	59.47%	29.82%
Unroll-4	30.16%	20.75%	22.94%	21.62%	12.49%
Unroll-8	19.32%	10.84%	15.54%	11.41%	9.80%
gMatch	4.14%	3.41%	3.53%	4.04%	1.51%

because candidate sets are small after filtering, causing substantial thread idling in coarse-grained models.

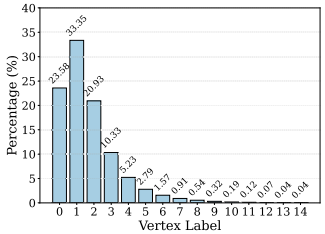
To quantify this effect, we measure the idle rate (defined in Section 2.3) across the datasets, as shown in Table 6. Compared with existing methods, our approach reduces the idle rate to below 5%, demonstrating much higher efficiency.

6.4 Case Study

We conduct a case study on a production graph representing user payment relationships from our industry partner ByteDance, one of the largest social network companies in the world. In this graph, each vertex corresponds to a user, and each undirected edge denotes a payment transaction, reflecting mutual interaction. The graph contains 139,297,601 vertices and 153,853,965 edges. Figure 12a shows its degree distribution. Note that different from our previous experiments, we conduct all our case study on a V100 GPU with 32 GB device memory.



(a) Cumulative distribution of vertex degrees.

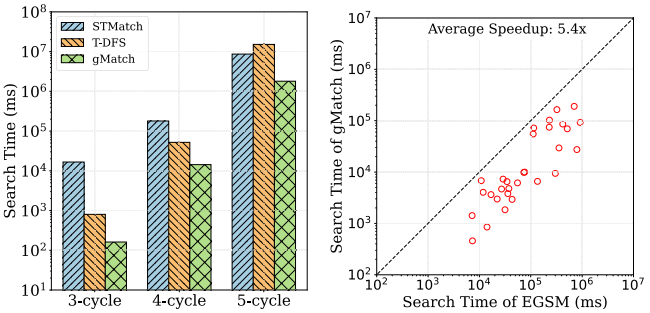


(b) Distribution of vertex risk labels.

Figure 12: Dataset of user payment relationships.

For small queries, we focus on identifying cycles of length 3 to 5, which represent circular transaction flows. These patterns are of particular interest as they often indicate suspicious or closed-loop behaviors. The detected cycles are used as input to downstream tasks such as graph neural network (GNN) analysis. The corresponding results are shown in Figure 13a.

The graph also includes vertex labels indicating user risk levels, ranging from 0 to 14. Figure 12b illustrates the label distribution. For large-query evaluation, we analyze transaction patterns across different risk categories using queries with dozens of vertices. Due to privacy constraints, we cannot publish the actual queries and instead generate 30 synthetic queries with 12 vertices. We generate these queries by adapting the methodology from Section 6.1, with one modification: to ensure the inclusion of risky vertices, the seed vertex is always randomly selected from the set of vertices whose labels fall within the range of 10 to 14. Figure 13b presents the results of our case study. As shown in the figure, our method significantly outperforms the existing approaches.



(a) Small queries. **(b) Large queries.**

Figure 13: Comparison of search times for the methods under study on the user payment network.

7 RELATED WORK

GPU-based subgraph matching. Recent research on subgraph matching has increasingly leveraged GPUs due to their massive parallelism capabilities. Existing GPU-based subgraph matching solutions can be categorized into BFS-based, DFS-based, and hybrid approaches according to their search strategy. Early GPU subgraph matching research primarily adopted BFS-based methods due to their inherent workload balance advantages [37, 41, 46]. However, BFS-based approaches typically generate a large amount of intermediate results that cannot fit in the memory of GPUs. To reduce memory consumption, recent studies have revisited DFS-based strategies [12, 24, 39, 44] or DFS-BFS hybrid strategies [35]. Moreover, there are some researches focusing on processing subgraph matching on very large data graphs that cannot fit in GPU memory. They either divide the data graph into multiple partitions [15] or use a view-based strategy according to the matched data vertex of the source query vertex [21, 43, 45]. These algorithms preprocess the data graph by dividing the problem into multiple chunks and load one chunk into memory at a time. Additionally, there are GPU algorithms specifically designed for k-clique counting [1, 25] and triangle counting [2, 14, 19], which are special cases of subgraph matching. While these works demonstrate high performance, they are not applicable to matching general patterns.

CPU-based subgraph matching. Existing CPU-based subgraph matching algorithms primarily leverage pruning strategies and auxiliary data structures to enhance performance. The solutions of subgraph matching can be roughly divided into join-based and exploration-based, as indicated by [33]. Current exploration-based approaches are fundamentally rooted in Ullmann's algorithms [38], incrementally matching query vertices to candidate vertices. Subsequent researches effectively reduce the search space by using optimized matching orders [6, 18], applying advanced filtering techniques [4, 27, 34, 42], avoid redundant computations [23, 30], and maintaining failure sets [3, 17]. While these exploration-based methods excel at handling complex queries on single machines through their complicated pruning strategies, join-based approaches offer better scalability for large-scale datasets by decomposing the query into sub-structures that can be processed in parallel, making them better suited for distributed environments.

Graph pattern mining. Graph pattern mining (GPM) focuses on finding specific patterns within large data graphs. It includes several applications such as subgraph listing [9, 28], k-motif counting [9, 31], and k-frequent subgraph mining [9, 10]. Early work tackled these problems by creating specialized, custom solutions for each specific task. This strategy is effective but not general-purpose. To make it easier to run different mining tasks efficiently in parallel, general-purpose graph mining systems were developed. These systems used an embedding-centric approach, starting with small pieces and gradually extending them step-by-step into larger subgraphs, checking at each stage if they meet the user's criteria. Systems like [7, 36] used this model. However, similar to subgraph matching algorithms, these GPM systems also face the problem of tracking the huge number of intermediate partial results. To solve this problem, [9, 11, 32] adopt DFS strategies for enumeration.

8 CONCLUSION

We presented gMatch, a fine-grained and hardware-efficient subgraph matching method for GPUs. Our work is driven by the key insight that the coarse-grained parallel execution model fundamentally conflicts with real-world power-law graphs, giving rise to two inherent issues. First, GPU scalability is constrained by memory: large per-task stacks severely limit the number of concurrent warps and underutilize GPU parallelism, making a lightweight stack design essential. Second, mapping highly irregular graph workloads onto fixed-size warps leads to significant thread underutilization. To address these challenges, our fine-grained design incorporates warp-level batch exploration and a lightweight load-balancing mechanism, achieving high efficiency and scalability across a wide range of workloads. Experimental results show that gMatch delivers strong performance and scalability with low memory overhead, making it well suited for practical, large-scale graph applications.

ACKNOWLEDGMENTS

The work is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62572303. Shixuan Sun and Minyi Guo are the corresponding authors.

REFERENCES

[1] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2022. Parallel K-clique counting on GPUs. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 21, 14 pages.

[2] Mohammad Almasri, Neo Vasudeva, Rakesh Nagi, Jinjun Xiong, and Wen-Mei Hwu. 2021. HyKernel: A Hybrid Selection of One/Two-Phase Kernels for Triangle Counting on GPUs. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[3] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. GuP: Fast Subgraph Matching by Guard-based Pruning. *Proc. ACM Manag. Data* 1, 2, Article 167 (June 2023), 26 pages.

[4] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1447–1462.

[5] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1199–1214.

[6] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14 (2013), 1–13.

[7] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 32, 12 pages.

[8] Weitian Chen, Shixuan Sun, Cheng Chen, Yongmin Hu, Yingqian Hu, and Minyi Guo. 2026. gMatch: Fine-Grained and Hardware-Efficient Subgraph Matching on GPUs. *arXiv preprint arXiv:2604.10601* (2026).

[9] Xuhao Chen and Arvind. 2022. Efficient and Scalable Graph Pattern Mining on GPUs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 857–877.

[10] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: an efficient and flexible graph mining system on CPU and GPU. *Proc. VLDB Endow.* 13, 8 (April 2020), 1190–1205.

[11] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1357–1374.

[12] Vibhor Dodeja, Mohammad Almasri, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2022. PARSEC: PARallel Subgraph Enumeration in CUDA. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 168–178.

[13] Orri Erling, Alex Averbuch, Josep-Lluis Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD*. 619–630.

[14] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. 2014. Fast Triangle Counting on the GPU. In *2014 4th Workshop on Irregular Applications: Architectures and Algorithms (IA^3)*. 1–8.

[15] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1067–1082.

[16] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1379–1380.

[17] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *SIGMOD Conference*. 1429–1446.

[18] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 405–418.

[19] Yang Hu, Hang Liu, and H. Howie Huang. 2018. High-Performance Triangle Counting on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–5.

[20] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 13, 16 pages.

[21] Guanxian Jiang, Qihui Zhou, Tatiana Jin, Boyang Li, Yunjian Zhao, Yichao Li, and James Cheng. 2022. VSGM: view-based GPU-accelerated subgraph matching on large graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 52, 15 pages.

[22] Jiawei Jiang, Yusong Hu, Xiaosen Li, Wen Ouyang, Zhitao Wang, Fangcheng Fu, and Bin Cui. 2022. Analyzing Online Transaction Networks with Network Motifs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 3098–3106.

[23] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast Redundancy-Reduced Subgraph Matching. *Proc. ACM Manag. Data* 1, 1, Article 12 (May 2023), 26 pages.

[24] Samiran Kawtikwar, Mohammad Almasri, Wen-Mei Hwu, Rakesh Nagi, and Jinjun Xiong. 2023. Beep: Balanced efficient subgraph enumeration in parallel. In *Proceedings of the 52nd International Conference on Parallel Processing*. 142–152.

[25] Vinayak Kesarwani, Shivangi Gaur, Sudeep Ranjan Sahoo, Kishan Tamboli, and Vishwesh Jatala. 2024. A Partitioning Scheme for Large Scale Clique Counting on Single GPU. In *2024 IEEE 31st International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*. 167–168.

[26] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable SIMD-Efficient Graph Processing on GPUs. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT '15)*. 39–50.

[27] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 925–937.

[28] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H.A. Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1231–1245.

[29] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. Accessed: April 11, 2026.

[30] Yujie Lu, Zhijie Zhang, and Weiguo Zheng. 2025. BⓈX: Subgraph Matching with Batch Backtracking Search. *Proc. ACM Manag. Data* 3, 1, Article 15 (Feb. 2025), 27 pages.

[31] Chenhao Ma, Reynold Cheng, Laks V. S. Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. 2019. LINC: a motif counting algorithm for uncertain graphs. *Proc. VLDB Endow.* 13, 2 (Oct. 2019), 155–168.

[32] Daniel Mawhirter and Bo Wu. 2019. AutoMine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 509–523.

[33] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1083–1098.

[34] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: a holistic approach to subgraph query processing. *Proc. VLDB Endow.* 14, 2 (Oct. 2020), 176–188.

[35] Xibo Sun and Qiong Luo. 2023. Efficient GPU-Accelerated Subgraph Matching. *Proc. ACM Manag. Data* 1, 2, Article 181 (June 2023), 26 pages.

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

[37] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast Subgraph Matching on Large Graphs using Graphics Processors. In *Database Systems for Advanced Applications*, Matthias Renz, Cyrus Shahabi, Xiaofang Zhou, and Muhammad Aamir Cheema (Eds.). Springer International Publishing, Cham, 299–315.

[38] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (Jan. 1976), 31–42.

[39] Yihua Wei and Peng Jiang. 2022. STMatch: accelerating graph pattern matching on GPU with stack-based loop optimizations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 53, 13 pages.

[40] Peter Willett. 1999. *Matching of Chemical and Biological Structures Using Subgraph and Maximal Common Subgraph Isomorphism Algorithms.* Springer New York, New York, NY, 11–38.

[41] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 69, 14 pages.

[42] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction. *Proc. ACM Manag. Data* 1, 1, Article 15 (May 2023), 26 pages.

[43] Lyuheng Yuan, Akhlaque Ahmad, Da Yan, Jiao Han, Saugat Adhikari, Xiaodong Yu, and Yang Zhou. 2024. G2-AIMD: A Memory-Efficient Subgraph-Centric Framework for Efficient Subgraph Finding on GPUs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3164–3177.

[44] Lyuheng Yuan, Da Yan, Jiao Han, Akhlaque Ahmad, Yang Zhou, and Zhe Jiang. 2024. Faster Depth-First Subgraph Matching on GPUs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3151–3163.

[45] Li Zeng, Lei Zou, and M. Tamer Özsu. 2023. SGSI – A Scalable GPU-Friendly Subgraph Isomorphism Algorithm. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2023), 11899–11916.

[46] Li Zeng, Lei Zou, M. Tamer Özsu, Lin Hu, and Fan Zhang. 2020. GSI: GPU-friendly Subgraph Isomorphism. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1249–1260.

[47] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (Saint Petersburg, Russia) *(EDBT '09)*. Association for Computing Machinery, New York, NY, USA, 192–203.

[48] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proc. ACM Manag. Data* 2, 1, Article 60 (March 2024), 29 pages.