GPU-Accelerated Batch-Dynamic Subgraph Matching

Linshan Qiu[†], Lu Chen[†], Hailiang Jie[†], Xiangyu Ke[†], Yunjun Gao[†], Yang Liu[§], Zetao Zhang[§]

[†]*Zhejiang University, Hangzhou, China*　　　[§]*Huawei, Chengdu, China*

[†]*{lsqiu, luchen, hljie, xiangyu.ke, gaoyj}@zju.edu.cn*　　　[§]*{liuyang169, zhangzetao3}@huawei.com*

Abstract—**Subgraph matching has garnered increasing attention for its diverse real-world applications. Given the dynamic nature of real-world graphs, addressing evolving scenarios without incurring prohibitive overheads has been a focus of research. However, existing approaches for dynamic subgraph matching often proceed serially, retrieving incremental matches for each updated edge individually. This approach falls short when handling batch data updates, leading to a decrease in system throughput. Leveraging the parallel processing power of GPUs, which can execute a massive number of cores simultaneously, has been widely recognized for performance acceleration in various domains. Surprisingly, systematic exploration of subgraph matching in the context of batch-dynamic graphs, particularly on a GPU platform, remains untouched.**

In this paper, we bridge this gap by introducing an efficient framework, <u>GAMMA</u> (<u>G</u>PU-<u>A</u>ccelerated Batch-Dyna<u>m</u>ic Subgraph <u>Ma</u>tching). Our approach features a DFS-based warp-centric batch-dynamic subgraph matching algorithm. To ensure load balance in the DFS-based search, we propose warp-level work stealing via shared memory. Additionally, we introduce coalesced search to reduce redundant computations. Comprehensive experiments demonstrate the superior performance of GAMMA. Compared to state-of-the-art algorithms, GAMMA showcases a performance improvement up to hundreds of times.

Index Terms—**Subgraph Matching, Batch-dynamic, GPU**

I. INTRODUCTION

Subgraph matching involves identifying subgraphs in a given data graph that are isomorphic to a targeted query graph. As an example, in Figure 1, $\{(u_0, v_1), (u_1, v_5), (u_2, v_6), (u_3, v_9)\}$ is a match of query graph Q in data graph G. This process plays a crucial role in various domains [3], [21], [39], including network alignment, graph learning, and VLSI placement, among others. For instance, in VLSI placement, engineers leverage subgraph matching to pinpoint and replace areas that can be optimized, thereby mitigating costs.

Considerable efforts have been directed towards the development of algorithms tailored for static scenarios [4], [5], [12], [21], [23], [25], [31], [36], [42], [50], [51]. However, real-life applications often involve evolving graph structures. Simply applying the aforementioned static approaches to dynamic scenarios necessitates *re-finding matches from scratch*, leading to prohibitively high computational overhead. More specifically, they are required to compute matches resulting from the consecutive updates of two edges and identify the incremental matches. To address these challenges, researchers have explored the continuous subgraph matching (CSM) [11], [14], [26], [27], [33], [34], [40], which searches for incremental matches for each individual update in a *sequential* manner.
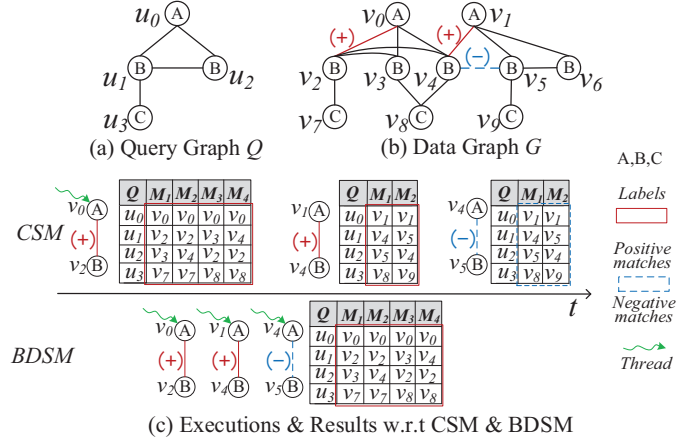

Fig. 1. A running example of batch-dynamic subgraph matching

In the current landscape of data processing, characterized by immense dataset sizes and rapid transformations, updates are often grouped together and applied as *batches* [13]. This shift has led to a recent surge of interest in the development of efficient *parallel batch-dynamic* algorithms [1], [13], [15], [29], [48]. Building on this trend, our paper delves into the intricate problem of subgraph matching under a batch-dynamic setting (BDSM). In BDSM, incremental matches are computed for a batch of updates rather than handling them individually. This problem holds practical significance in various domains, including cellular networks [24], social networks [8], and e-commerce platforms [48]. In these scenarios, graph databases are collected and updated in batches, leveraging subgraph matching for tasks such as identifying patterns of malicious activity [16]. Despite the extensive literature on dynamic graph subgraph matching, i.e., the aforementioned CSM, scant attention has been given to batch-dynamic subgraph matching algorithms that enable parallel processing of update batches. To illustrate this distinction, a comparative example between CSM and BDSM is presented below.

Example 1. *In Figure 1(b), three updates (v_0, v_2), (v_1, v_4), (v_4, v_5) unfold sequentially, with insertions denoted by $(+)$ and deletions by $(-)$. The detailed execution is illustrated in Figure 1(c), comparing CSM and DBSM. CSM processes updates individually, starting with (v_0, v_2), generating four affirmative matches. Subsequent updates (v_1, v_4) and (v_4, v_5) result in two positive and two negative matches, respectively. Conversely, BDSM tackles these updates collectively as a parallel batch, yielding four positive matches. Note that BDSM disregards the order of updates, focusing solely on updated*

matches post-batch update [48], hence eliminating the redundant matches concerning (v_1, v_4) and (v_4, v_5) by CSM. In conclusion, BDSM (i) mitigates the computational load by consolidating updates into batches, and (ii) lends itself to facile parallelization as each update is independent.

With the development of modern hardware boasting high computational capacities [32], [38], [46], there has been a dedicated research effort to harness these advancements in addressing traditional computation-heavy tasks [6], [9], [10], [18], [37]. Among these hardware options, GPUs, with their numerous computing cores [38], show great promise. Surprisingly, systematic exploration of subgraph matching in the context of batch-dynamic graphs, particularly on a GPU platform, remains largely untouched. Enabling each update to be independently processed, yet retaining a comprehensive view of all updates within a batch, harnesses the parallel processing capabilities of GPUs to effectively address BDSM. The intricacies of the subgraph matching process involve frequent yet straightforward operations, like set intersections, contributing significantly to its exponential algorithmic complexity[1]. Leveraging the multitude of simpler cores in GPUs proves highly conducive to efficiently managing these computations, aligning well with the demands of the problem at hand. The primary challenges stem from the disparate architectures of CPUs and GPUs, characterized by variations in core placement and memory hierarchy. These differences result in *distinct execution modes and memory access patterns*. To ensure the optimal execution of algorithms on GPUs, it is imperative to meticulously devise algorithms tailored to exploit the unique strengths of GPU architectures. Given the nature of our problem and the underlying hardware architecture, three key challenges come to the forefront.

Challenge I: *How to maximize the GPU resource utilization for enhanced parallelism?* GPUs, with their hierarchical composition of threads and memory, feature a relatively smaller memory capacity but a significantly larger number of cores compared to CPUs [38]. Existing graph processing algorithms [14], [26], [50] often resort to exploration based on Breadth-First Search (BFS) to exploit parallelization capabilities, albeit incurring substantial *memory overhead*. In contrast, exploration based on Depth-First Search (DFS) significantly reduces memory consumption by exploring only a small portion of vertices at a time [9]. However, DFS introduces challenges related to *load imbalance* due to the unpredictable workload associated with each task. The need to retrieve multi-hop neighbors during DFS search, coupled with significant variations in adjacency lists, exacerbates the issue. Given the constraints of GPU memory, our system adopts DFS search for match retrieval. To address the associated performance challenges, we establish a baseline with a warp-centric granularity (§IV-C). This approach effectively mitigates load imbalance by leveraging the memory hierarchy of GPUs. Specifically, we assign a warp to cooperatively handle an updated edge, thereby reducing thread divergence. Furthermore, the threads

within a warp remain together to facilitate memory transitions in a coalesced manner, leading to a substantial reduction in memory divergence. To tackle load imbalance, we propose a work-stealing strategy, distributing workload information for each warp in shared memory. This enables idle warps to seize work from active ones, significantly enhancing thread utilization and overall system efficiency (§V-A).

Challenge II: *How to avoid unnecessary graph traversing to improve the matching efficiency?* In BDSM, inherent computational redundancies during the search primarily arise from *automorphisms* present in query subgraphs. A vivid example is illustrated with the insertion of $e(v_0, v_3)$ in G (Figure 1). When mapping $e(v_0, v_2)$ to $e(u_0, u_1)$, the incremental matches are M_1 and M_2. Similarly, mapping this update to (u_0, u_2) leads to additional matches M_3 and M_4. Remarkably, the subgraph induced by vertices v_0, v_2, v_3 is visited twice during this process. This redundancy emanates from the automorphisms present in the subgraph of the query graph, specifically the induced subgraph composed of u_0, u_1, u_2. To mitigate this redundancy, we introduce the coalesced search technique (§V-B), anchored in the concept of k-degenerated automorphic subgraph. This subgraph retains its automorphisms even after removing k vertices from the original graph. Using the k-degenerated automorphic subgraph, we aggregate equivalent edges into a group. Leveraging the k-degenerated automorphic subgraph, we aggregate equivalent edges into a group. Hence, we need only consider one edge among the equivalent edges during mapping and generate other partial matches through permutation operations. This innovative approach significantly reduces the redundant traversals, thereby streamlining the matching process and improving overall efficiency.

Challenge III: *How to harmonize modules to achieve efficient computation pipelining?* This challenge is crucial from a system development perspective, requiring meticulous groundwork in preprocessing and graph updating to facilitate incremental subgraph matching at higher levels and achieve optimal performance. Our objective is to pave the path for incremental subgraph matching while minimizing the time overhead of preceding steps prior to result computation. To achieve this, we adopt an asynchronous approach (§IV-A), conducting preprocessing on the CPU concurrently with GPU computations to alleviate waiting times. Facing the continuous arrival of update batches, efficient application of updates to the data graph becomes paramount. Inspired by this, we embrace the widely used GPMA [35] as the underlying dynamic graph structure. However, certain inefficiencies arise when dealing with small-sized segments and locating segments with GPMA. Consequently, we introduce practical optimizations by leveraging the Cooperative Group and shared memory, respectively, making the entire computational pipeline more seamless and responsive to the continuous influx of update batches(§V-C).

To sum up, the key contributions are summarized as follows:

- We introduce GAMMA, the first GPU-based approach tailored for efficient batch-dynamic subgraph matching. This groundbreaking proposal leverages the parallel pro-

---

[1]Set intersections take 58.2% of total runtime in subgraph matching [20].

cessing power of GPUs, marking a significant advancement in this domain.

- We design a warp-centric batch-dynamic subgraph matching algorithm that capitalizes on GPU parallelism, addressing challenges like thread and memory divergence. Additionally, we introduce a warp-centric work-stealing strategy, utilizing shared memory to balance workloads among warps within the same block.
- We devise a coalesced search technique to tackle computational redundancies arising from subgraph automorphisms. Unnecessary computations are minimized during matching by only finding matches for one automorphism and deriving other partial ones through permutations.
- We conduct extensive experiments on 6 public datasets, showcasing that our framework significantly outperforms popular advanced methods by up to hundreds of times.

The rest of the paper is organized as follows. We formulate our problem and introduce GPUs in Section II. Section III reviews the related work. Sections IV and V illustrate our designs. The experimental results and our findings are reported in Section VI. Finally, Section VII concludes the paper.

II. Preliminaries

In this section, we formally define the problem and introduce GPUs. The frequently used notations are listed in Table I.

A. Problem Definition

We first introduce some basic concepts, and then we formulate the studied problem.

Let $g = (V, E, L)$ be an undirected labeled graph, where V denotes the vertex set, $E \subseteq V \times V$ is the edge set, and L is a mapping function that maps a vertex $v \in V$ or an edge $e \in E$ to a label l in a label set Σ. Take the query graph Q in Figure 1(a) for example. The label set of Q is $\Sigma = \{A, B, C\}$, and the label of u_0 is A, i.e., $L(u_0) = A$. Specifically, in biological networks, vertices are labeled with protein types, and then, biologists can count the occurrences of a particular pattern to determine the property. In recommendation systems, users are labeled with different attributes, and then, a company can find its target customers by detecting community patterns. Here, we use $e(u, u')$ to represent the edge between vertices u and u'. Given a vertex $v \in V$, we denote its neighbor set as $N(v)$ and $deg(v) = |N(v)|$ as its degree. Besides, $N^l(v)$ are the neighbors of v that have the label l. In our problem, two types of graphs are involved, i.e., query graph Q and data graph G. We call vertices and edges in the query graph Q (resp. data graph G) query vertices and query edges (resp. data vertices and data edges).

Definition 1. *A graph update stream $\Delta\mathcal{B} = (\Delta B_1, \Delta B_2, \dots)$ is a sequence of update operations, where ΔB is a set of edge insertions/deletions $\{\Delta e_i\}$, as $\Delta e_i = (\oplus, e_i)$ is the insertion/deletion of an edge e_i, where $\oplus$ denotes insertion $(+)$ or deletion $(-)$.*

A graph is batch-dynamic when each update stream contains multiple insertion/deletion operations, that is $|\Delta B| > 1$.

TABLE I
SYMBOLS AND DESCRIPTIONS

Symbos	Descriptions
G, Q	the data graph and the query graph
$V(Q), E(Q)$	the vertex set and the edge set of Q
$e(u, u')$	an edge between vertex u and u'
$L(u)$	the label of vertex u
$N(u), N^l(u)$	neighbors and neighbors with label l of vertex u
$deg(u)$	the degree of vertex u
$\Delta\mathcal{G}, \Delta G, \Delta e$	graph stream, graph update, single update
$\oplus = +/-$	an edge insertion/deletion
$C(u)$	the candidate set of data vertex u
$\Delta\mathcal{M}$	incremental matches
π	matching order

Note that, vertex insertions/deletions can be treated as edge insertions/deletions. For vertex insertions, we first insert the vertices into the data graph and regard the incident edges as a collection of edge insertions. For vertex deletions, we first remove the corresponding vertices and treat the incident edges as a collection of edge deletions.

Definition 2. *Given a query graph Q and a data graph G, a subgraph isomorphism of Q in G is a bijective function M between $V(Q)$ and $V(G_s)$, where G_s is a subgraph of G such that (1) $\forall u \in V(Q), L(u) = L(M(u))$; and (2) $\forall e(u, u') \in E(Q), e(M(u), M(u')) \in E(G_s)$.*

According to Definition 2, we call a subgraph isomorphism an embedding or a match. As the example shown in Figure 1, a subgraph isomorphism of Q in G is $M = \{M(u_0) = v_1, M(u_1) = v_5, M(u_2) = v_6, M(u_3) = v_9\}$. Given a graph update $\Delta B \in \Delta\mathcal{B}$, we denote G' the graph resulted from applying ΔB to G. The incremental matches $\Delta\mathcal{M}$ with respect to the graph update ΔB is the difference between $\Delta\mathcal{M}$ and $\Delta\mathcal{M}'$, where $\Delta\mathcal{M}$ and $\Delta\mathcal{M}'$ represent the matches in G and G', respectively.

Existing continuous subgraph matching (CSM) algorithms [11], [14], [26], [27], [33], [34], [40] assume that the graph update contains only a single edge update, i.e., $|\Delta B| = 1$, and aim to find the incremental matches $\Delta\mathcal{M}$ for each single updated edge separately. To fully utilize the power of parallelism a GPU provides, we suppose $|\Delta B| > 1$ and perform batch-dynamic subgraph matching on ΔB.

Problem Statement. Given update batches $\Delta\mathcal{B}$, batch-dynamic subgraph matching (**BDSM** for short) finds all incremental matches $\Delta\mathcal{M}$ for each $\Delta B \in \Delta\mathcal{B}$, where $|\Delta B > 1|$.

B. Graphics Processing Units (GPUs)

In contrast to CPUs, which have only a few cores, GPUs offer thousands of lightweight cores. CUDA (Compute Unified Device Architecture) provides a programming abstraction that connects applications with GPU hardware. Refer to Figure 2 for a simplified GPU architecture.

Thread Hierarchy. A modern GPU consists of streaming multiprocessors (SMX), each containing streaming processors (SP). When a program (known as a kernel in CUDA), multiple threads collaborate to perform computations. Threads use SPs, with 32 threads forming a warp. Warps are the SM's
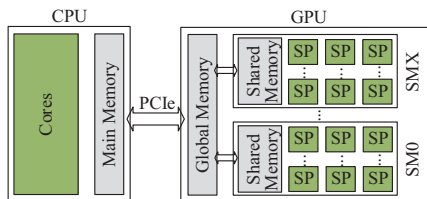
Fig. 2. The simplified GPU architecture

scheduling units, operating in a single-instruction multiple-thread (SIMT) manner. Threads that meet a branch condition execute concurrently, while others idle, a situation called warp divergence to avoid. Several warps create a Cooperative Thread Array (CTA) or block, assigned to an SM and unchangeable during runtime. Blocks form a grid, encompassing all GPU threads.

Memory Hierarchy. GPUs are memory-efficient due to their high-bandwidth and parallel access design. They feature multiple memory levels: global memory (shared by all threads, with slower access), shared memory (small but with high bandwidth, allocated to thread blocks), and registers (smallest, for frequently accessed data). To improve efficiency, threads within the same warp should aim for consecutive memory accesses, called memory coalescing. Non-consecutive access results in lower bandwidth use, underscoring the importance of data locality.

III. RELATED WORK

We briefly review related work about subgraph matching on static graphs and dynamic counterparts, followed by GPU-accelerated graph processing.

A. Subgraph Matching

Subgraph matching, the task of identifying query graph Q occurrences within data graph G, has seen extensive research. Ullmann [42] introduces a backtracking algorithm that expands a partial embedding by sequentially mapping query vertices to data vertices. Subsequent research falls into three categories: the first [12], [36] directly searches the data graph to find matches, suffering from prohibitive space and computational overheads. The second [50], [51] constructs an index for the data graph before searching. The third category [4], [5], [21], [23] creates candidate sets for query vertices, maintains edges between them using an auxiliary data structure. Subsequently, a matching order is generated, and the matched results are enumerated.

However, the aforementioned approaches remain largely oblivious to the query graph's structure, potentially exploring unnecessary subgraphs. Recently, there have been advancements in pattern-aware subgraph matching algorithms [25], [31]. Yet, applying such static subgraph matching algorithms to dynamic scenarios necessitates rebuilding the index and searching the differential matched subgraphs between snapshots, incurring excessive space and computational costs.

B. Continuous Subgraph Matching

Numerous studies have addressed efficient subgraph matching in dynamic graphs(CSM) [11], [14], [26], [27], [33], [34],

[40], [47]. IncIsoMat [14] extracts relevant subgraphs from the data graph and performs subgraph matching before and after updates. However, it enumerates unnecessary matches, leading to substantial computational overhead. Recent approaches use an incremental style for CSM. For example, Graphflow [26] maps updated edges to the query graph and extends partial results by repeatedly joining the remaining vertex of the query graph. SJ-Tree [11] employs index-based binary joins but requires significant memory storage. TurboFlux [27] efficiently solves CSM using a data-centric graph (DCG). SymBi [34] maintains a directed acyclic graph and embeds weak embeddings of directed acyclic graphs to quickly retrieve matches and support efficient updates. RapidFlow [40] reduces CSM to batch subgraph matching (BSM), upon which an effective matching order can be generated. Calig [47] minimizes incremental match generation time by reducing backtracking.

Despite reducing recomputation, these methods conduct CSM sequentially. Motivated by this, we investigate batch-dynamic subgraph matching to enhance efficiency. Batch processing excels at handling substantial volumes of evolving data. Several algorithms have been developed for batched-updates, including computing clustering coefficients [2], single-source shortest-path [50], dynamic connectivity problems [5], and triangle counting [5]. However, subgraph matching in batch-dynamic graphs, a fundamental problem in graph processing, remains unexplored to the best of our knowledge.

C. GPU-accelerated Graph Processing

The emergence of novel hardware, particularly GPUs, presents opportunities to expedite a range of computational tasks [38], including graph processing tasks like clique counting [38], subgraph enumeration [19], [28], pattern mining [9], [10], and PageRank computation [37]. Systems like Gunrock [43], Pangolin [10], and GraphPEG [30] simplify graph analysis on GPUs. However, these tools primarily target static graphs. To hasten dynamic graph processing, GPMA [35] employs the Packed Memory Array (PMA) for quick updates. Subsequently, cuSTINGER [17], aimGraph [45], FaimGraph [44], Hornet [7], and others have proposed various data structures for dynamic graphs. These data structures are fundamentally intended to curtail dynamic graph updates rather than diverse graph processing tasks. This paper investigates GPUs-accelerated batch-dynamic subgraph matching.

IV. METHOD

In this section, we will initially furnish an overview of our method, followed by a detailed exposition of each component.

A. Overview

Figure 3 provides an insightful overview of our innovative CPU-GPU heterogeneous framework, GAMMA, for batch-dynamic subgraph matching. The framework comprises four key components: Preprocess, Update, Computational Kernel (BDSM), and Postprocess. For each batch, the preprocess component conducts fundamental analyses, such as maintaining neighborhood information and generating candidate
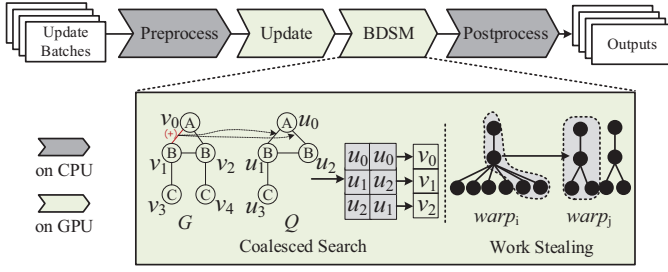
Fig. 3. Overview of GAMMA

sets. These outcomes, along with the updated edges, are then dispatched to the GPU. Subsequently, GAMMA executes the update on the data graph, followed by the computational kernel identifying incremental matches. Finally, the postprocess component utilizes the matching results for application-specific tasks.

The computational kernel integrates two practical optimizations: work stealing (§ V-A) and coalesced search (§ V-B). These optimizations strategically balance the workload and eliminate redundant computations. Importantly, all four components operate asynchronously. The computational kernel is intricately designed to overlap the preprocessing step and the host-to-device data transfer for the next batch. Likewise, once the matching results are generated, they seamlessly overlap with the next update and computation step. This asynchronous process continuously repeats, enhancing the overall efficiency.

B. Preprocessing

The primary function of preprocessing is to expeditiously and effectively generate candidate sets for each query vertex in the query graph. Existing studies have put forth a plethora of filtering, among which neighborhood-label-frequency-based [21] are widely employed. The idea of this filtering strategy is that the data vertices in the candidate set for a query vertex should exhibit *a similar neighborhood structure*. To be more specific, for each candidate vertex v in the candidate set $C(u)$ of a given query vertex u, in addition to the label constraint, i.e., $L(v) = L(u)$, each candidate data vertex $v \in C(u)$ should contain neighbors with same labels from the query vertex's neighborhoods, and the number of such neighbors should not be fewer than the count of corresponding neighbors of the query vertex, i.e., $|N^l(v)| \geq |N^l(u)|$ where $l \in \{L(u')|u' \in N(u)\}$. Leveraging this concept, GSI [49] applies binary encoding on vertices, where each vertex is represented by a K-bit binary code. Hence, the K-bit binary code is divided into two parts: the first N bits for the vertex label, and the remaining $(K - N)$ bits for the neighborhood structure. The second part is further divided into groups of M bits to record the neighbor counts with different labels. Inspired by GSI, we refine this encoding strategy to avoid encoding labels absent in the query graph. Figure 4 illustrates the encoding results of G before and after applying the updates (denoted as G_1 and G_2, respectively) and Q in Figure 1. Regarding the encoding table of G_2, there are $K = 9$ bits for each vertex and the first $N = 3$ bits denote the label, e.g., "001" for label "A" on v_0 and v_1. The
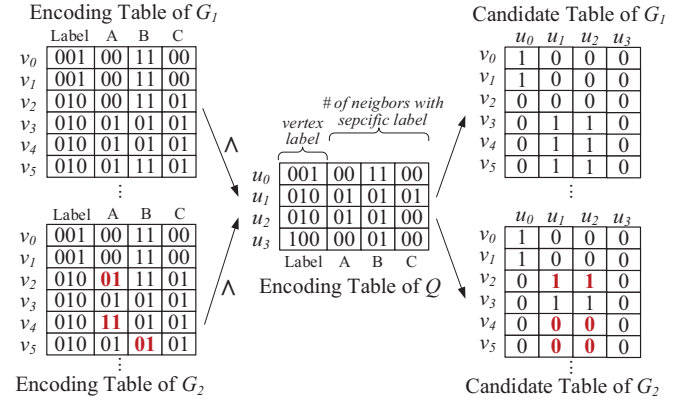
*G_1, G_2: G before and after update.   ∧: bitwise AND.

Fig. 4. Preprocessing for candidate table generation. $v_6 \sim v_9$ are omitted for brevity. In the example, we use the first 3 bits for vertex label encoding and the remaining 6 bits for counting neighbors with specific labels.

remaining $(K - N) = 6$ bits, grouped in sets of $M = 2$, depict the count of neighbors with specific labels. For instance, v_0 has three neighbors with label B, hence in the encoding table corresponding to the columns for label B, we set it as "11". The encoding is conducted on the CPU. To generate the candidate sets for each query vertex, we can simply perform a bitwise AND operation. For example, to generate the candidate set for query vertex u, we perform "$ENC(u) \bigwedge ENC(v)$" for each data vertex v, where $ENC(v)$ represents the encoding of v. If the result is $ENC(u)$, then v is a candidate of u. With high parallelism provided by GPU, the bitwise AND operation can be efficiently performed.

Encoding of dynamic graphs. Given the dynamic nature of our problem setting, re-encoding data vertices for each batch is not only time-consuming but also incurs a substantial data transfer cost, potentially becoming a bottleneck. Henceforth, during the initialization phase, we compute the encodings for all data vertices. For subsequent batches, we load only the vertices with modified encodings into the GPU and compute corresponding candidate sets. Take G in Figure 1(b) as an example. After applying the updates, the encodings of the corresponding vertices update accordingly, as depicted by the red bold font in the bottom left part of Figure 4. Specifically, following the insertion of $e(v_0, v_2)$, v_2 gains a neighbor with label A, prompting an increment from "00" to "01" in the respective column of the encoding table for G_2, marked in red. Notably, despite $e(v_0, v_2)$ involves vertex v_0, its encoding remains unchanged due to our use of a 2-bit binary code, a trade-off between space and filtering capabilities. Subsequently, we compute candidate sets for query vertices, updating the candidate table (to be explained later) for G_2. For example, due to the insertion of $e(v_0, v_2)$, v_2 will match u_1 and u_2.

Candidate Table. Directly allocating an array for each query vertex to store its candidate data vertices incurs a substantial memory storage cost. Given the limited GPU memory, we introduce a more space-efficient representation known as the candidate table. The structure of the candidate table is depicted in the right part of Figure 4, with each row corresponding to a data vertex and each column corresponding

to a query vertex. Binary markers are employed to signify whether a data vertex belongs to the candidate set of a specific query vertex. For example, v_0 is a candidate vertex of u_0, and thus, we mark the entry in the first row and first column as 1 in the candidate table of G_2.

C. Search Strategies *in BDSM Computational Kernel*

The update component can be seamlessly accommodated in our data structure proposed in §V-C. In this subsection, we proceed to discuss the search strategy employed in GAMMA.

BFS vs DFS. Current GPU-centric pattern mining algorithms predominantly optimize data locality and enhance parallelism by leveraging BFS [9]. However, this approach often leads to an exponential increase in intermediate results, consuming a significant amount of memory [9]. When device memory reaches its limit, the system resorts to moving data between system memory and global memory, causing further performance degradation [19]. On the other hand, traditional CPU-based subgraph matching algorithms commonly adopt the DFS-based backtracking framework introduced by Ullmann [42]. DFS-based searches significantly reduce memory overhead by materializing only the final results and avoiding the storage of numerous (invalid) partial matches.

In Figure 5(a), we illustrate the trends in device memory usage for BFS and DFS using the *LS* dataset. It's apparent that BFS leads to exponential memory growth, quickly depleting the available memory. In contrast, DFS maintains a more gradual memory consumption profile, significantly lower than that of BFS. Due to memory exhaustion, a substantial volume of data transfers (Comm. in Figure 5(b)) between main memory and device memory becomes necessary, exerting a dominant influence on the total time, even surpassing the computation time (Comp. in Figure 5(b)) by several times. Additionally, the computational time of BFS also exceeds that of DFS, as BFS requires synchronization after each expansion.

Consequently, we opt for DFS to retrieve the incremental matches, prioritizing memory resources due to their significance and limited quantities. When performing DFS on CPUs, the number of simultaneously running threads is constrained by core limitations. In the context of GPU execution, a massive number of threads operating concurrently exacerbate the load imbalance issue. Moreover, the task of estimating workloads and consumed memory in the DFS process proves to be a formidable challenge. Finally, the random access nature of DFS conflicts with the coalesced memory access characteristics of GPUs. Therefore, the organization of threads becomes crucial.

Choice of Thread Granularity. In our problem setting, a task is responsible for searching the matches of an updated edge. Adopting various thread granularities may lead to notable performance disparities. The following discussion delves into the thread granularity choices for our specific problem.

- **Thread-centric Assignment**. Assigning a thread per update seems intuitive, offering high parallelism, but it introduces performance issues. A surplus of threads can lead to reduced resources per thread, and divergent branch
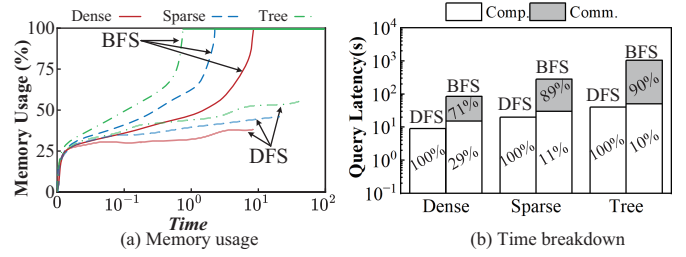

Fig. 5. A comparison of BFS and DFS in a GPU environment

conditions for different vertices result in thread divergence. Additionally, non-consecutive memory transactions for each thread cause memory divergence, incurring unnecessary costs for memory-fetching operations.

- **Warp-centric Assignment**. The warp-centric task assignment designates a warp to handle an update, enabling cooperative operations among its threads with low-cost primitives. This helps address the memory divergence problem, as the 32 threads within a warp stay together during memory reading for adjacency lists. Additionally, they collaborate on set intersections for extracting candidate sets, contributing to improved performance and reduced thread divergence. Despite these benefits, workload imbalance remains inherent due to the unpredictable execution paths of DFS, and we will soon introduce a solution to address this issue among warps. (§V-A).

- **Block-centric Assignment**. The block-centric strategy allocates more resources for each update. However, handling load imbalances between blocks requires high-latency device memory transactions. The limitation on the number of blocks that can execute concurrently in streaming multiprocessors presents a constraint that we acknowledge and plan to explore in future work.

Algorithm 1 depicts our warp-centric batch-dynamic subgraph matching algorithm (termed as WBM), i.e., assigning a warp to process each updated edge concurrently. WBM takes a data graph G, a query graph Q, update stream $\Delta\mathcal{B}$, and a matching order π as inputs, and outputs the incremental matches $\Delta\mathcal{M}$. The matching order guides the order in which query vertices are matched, and we generate it for each query edge offline. The matching order tends to prioritize the more selective query vertices, such as those with higher degrees and fewer candidates, which can provide more strict constraints for candidate generation and minimize the candidate size, hence pruning the search space. Under the update stream $\Delta\mathcal{B}$, WBM conducts a while-loop to enumerate each update batch ΔB (lines 1–22). Hence, a for-loop processes each record of ΔB in parallel (lines 2–22). The algorithm first maps each update edge Δe to a query edge and initializes the partial matches M (lines 4–5). Subsequently, the DFS search starts from level $l = 2$ (line 6), as the first two query vertices are already determined during mapping. The counter p counts the candidate vertices being handled at layer l. Built on the current partial matches, WBM computes the candidate set for the next query vertex using GenCandidates(line 7). GenCandidates

Algorithm 1: Warp-centric batch-dynamic subgraph matching algorithm (WBM)

Input: data graph G, query graph Q, update stream $\Delta\mathcal{B}$, matching order π
Output: incremental matches $\Delta\mathcal{M}$

```
1  while  ΔB ⊆ ΔB ≠ null do
2      foreach  Δe ∈ ΔB do
           // do in parallel
3          foreach  e(u₁, u₂) ∈ E(Q) do
4              if L(v₁) = L(u₁) && L(v₂) = L(u₂) then
5                  M ← {(u₁, v₁), (u₂, v₂)};
6                  l = 2; p[l] = 0;
7                  C[l] = GenCandidates(G, Q, M, π, l);
8                  while l ≥ 2 && p[l] < C[l].size do
9                      if l = (|V(Q)| − 1) then
10                         foreach c ∈ C[l] do
11                             ΔM+ = {M, (π[l], c)};
12                         while (− − l ≥ 2) &&
                               (+ + p[l] ≥ C[l].size) do
13                             pop an entry from M;
14                     else
15                         while p[l] < C[l].size do
16                             C[l + 1] =
                                 GenCandidates(G,Q,M ∪
                                 (π[l], C[l][p[l]]),π,l + 1);
17                             if C[l + 1] ≠ ∅ then
18                                 M+ = (π[l], C[l][p[l]]);
19                                 l + +,p[l] = 0, break;
20                             else p[l] + +;
21                         if p[l] ≥ C[l].size then
22                             execute as lines 12–13;


23 Procedure GenCandidates(G,Q,M,π,l)
24 res = CTable[π[l]];// CTable: candidate table
25 for i = 0 to (l − 1) do
26     if π[i] ∈ N(π[l]) then
27         v = M[π[i]];
           // intersect res with N(v)
28         res = Intersection(res, N(v));

29 return res
```

initializes the candidates res at layer l to $CTable[\pi[l]]$ and filters it by intersecting with the neighbors of the data vertices matched. The algorithm then proceeds to explore level by level (lines 8–22). When the search reaches the final level ($l = (|V(Q)| − 1)$, i.e., all the vertices have been visited), the algorithm joins M with each candidate vertex c and adds to ΔM (lines 10–11). Then, WBM backtracks to the nearest preceding level with unexplored candidates (lines 12–13). Otherwise, it attempts to find a candidate vertex at the current level that can produce the candidate set for the next level (lines 15–20). If such a qualified candidate vertex is found (i.e., $C[l + 1] \neq \emptyset$), it appends $(\pi, C[l][p[l]])$ to M and advances to the next level (lines 17–19). If there is no such candidate vertex, WBM backtracks to the previous level (lines 21–22). As each updated edge is handled by a warp, the threads responsible for the edge cooperatively compute the candidate vertices via GenCandidates, i.e., executing the intersection and
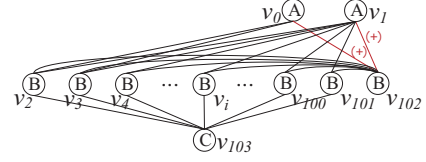
Fig. 6. Updates with skewed workloads

reading the global memory together. Moreover, the intersection is implemented by parallel binary search and is efficient. It is worth noting that incremental matching can involve multiple updates, and due to the independent handling of each update, duplicates may arise. To eliminate duplicates, we assign a total order to each update and establish a rule during the matching process that dictates matches can only form from lower-order update edges to higher-order update edges [19].

Complexity. For each update $\Delta e \in \Delta B$, procedure GenCandidates governs the execution time, which is implemented by binary search, leading to a time complexity $O(d_{max}(G) \log C_{max})$, where $d_{max}(G)$ and C_{max} denotes the maximum degree of data vertex and the maximum count of candidate vertices, respectively. The intersection executes at most $d_{max}(Q)$ times, resulting in a worst case complexity $O_{gc} = O(d_{max}(Q)d_{max}(G) \log C_{max})$. Consequently, for each update batch ΔB, the overall time complexity of Algorithm 1 is $O(|\Delta B||E_Q|d_{max}(G)^{|V_Q - 1|}d_{max}(Q) \log C_{max})$.

V. PRACTICAL OPTIMIZATIONS

In this section, we delve into the intricacies of Algorithm 1, shedding light on its bottlenecks and presenting comprehensive insights into our strategic enhancements.

A. Handling Load Imbalance

While the warp-centric assignment proves effective in alleviating thread and memory divergence, the persistent challenge lies in the *load imbalance* inherent to the DFS process. This stems from substantial *variations in the adjacency lists* of different vertices, making it impractical to precisely estimate the total number of neighbors for exploration. High-latency memory transactions are necessary to fetch multi-hop neighbors, adding to the complexity. Therefore, statically distributing tasks before execution becomes a formidable challenge. To tackle thist, we propose a warp-centric work-stealing strategy aimed at resolving load imbalances among warps. Let's illustrate this issue further with an example.

Example 2. *The data graph G in Figure 6 contains two insertions, namely bluee(v_0, v_{102}) and bluee(v_1, v_{102}), both matching the query edge $e(u_0, u_1)$ shown in blueFigure 6(a). blueFigure 1(a) illustrates the partial search tree corresponding to these insertions. The query vertices on the left side denote the matching order, where the first two vertices correspond to the query edge that matches the update edge. In this case, we initially map the insertions bluee(v_0, v_{102}) and bluee(v_1, v_{102}) to $e(u_0, u_1)$. In blueFigure 7(a), we assign warp i and warp j to handle the updates bluee(v_0, v_{102}) and bluee(v_1, v_{102}), respectively. As depicted in blueFigure 7, the number of qualified candidate vertices of u_2 and u_3 with*

(a) Workloads before work stealing (b) Workloads after work stealing

Fig. 7. Workloads of different warps and work stealing

respect to insertion $blue(v_1, v_{102})$ *is much higher than that of insertion* $blue(v_0, v_{102})$. *Consequently, bluewarp j bears a heavier workload compared to bluewarp i, resulting in a notable workload imbalance. It's essential to note that, since the search is conducted in a DFS style rather than BFS, it's only possible to calculate the workloads after the search exhausts. Once bluewarp i completes its assigned workloads, bluewarp j still has unfinished workloads, indicating a significant imbalance.*

As mentioned earlier, shared memory in a GPU is shared among threads within a block, offering fast accessibility and providing an opportunity to balance workloads among warps. We leverage the shared memory to implement our load balance optimization. There are two work stealing strategies to address the load imbalance issue.

- **Passive Stealing.** We reserve an array within shared memory, whose length is equivalent to the count of warps present within a block. Each element in the array signifies whether a warp has completed its assigned workloads, initialized with zeros. As a warp concludes its workloads, it records its status by setting the corresponding element in the array to 1. Periodically, warps with unfinished workloads scan the array to find an idle warp. When an idle warp is discovered, the warp with unfinished workloads undertakes a passive stealing operation, transferring a portion of its workloads to the idle warp. Passive stealing allows the idle warps to acquiescently receive the appropriated workloads. However, this passive stealing can lead to thread underutilization because a warp must interrupt its ongoing workloads to search for an idle warp.

- **Active Stealing.** To eliminate the need for periodic scrutiny of engaged warps, we propose the implementation of an active stealing strategy. After a warp completes its current workloads, it inspects other warps within the same block to determine if any still have unfinished tasks, bypassing the need for a dedicated search for idle warps. If such warps exist, the active stealing warp makes a thoughtful choice based on the remaining workloads and proceeds to actively appropriate half of its tasks. This approach promotes a more efficient balance of workloads among warps within the block.

Example 3. *In Figure 7, the active stealing strategy is visually presented. For simplicity, we assume that there are two warps in a block. Upon completing its workloads, a warp, let's say bluewarp i, examines the variables $csize$ and p stored in the shared memory layer by layer to detect those with unfinished workloads. Within this process, bluewarp i iterates through the $csize$ and p variables and successfully pinpoints unexplored candidates $\{v_5, \cdots, v_{102}\}$. Subsequently, bluewarp i appropriates half of the unexplored candidates, specifically blue$\{v_{54}, \cdots, v_{102}\}$, along with their parents bluev$_1$ and bluev$_{102}$, and adds them to its own candidate set. Figure 7(b) illustrates the resulting workloads of each warp after work stealing.*

Complexity. Work stealing involves an idle warp sequentially scanning $csize$ and p, estimating remaining workloads layer by layer. These procedure takes $O(L|W|)$ time, where L is the maximum number of layers to traverse and $|W|$ is the number of warps in a block.

Assuming workloads in a block are processed sequentially by a single warp with execution time of T_{seq}, work-stealing ideally ensures a fair workload distribution among warps, resulting in an execution time of $\frac{T_{seq}}{|W|}$ for each warp. This is equivalent to the total time due to the fair distribution. Without work stealing, if the maximum execution time among warps in a block is T_{max}, the speedup can be expressed as $\frac{|W|T_{max}}{T_{seq}}$. Notably, work-stealing provides more significant speedup benefits with more skewed workloads (See § VI-C and § VI-D for experimental details).

B. Reducing Redundant Matching

When examining the incremental matches associated with the insertion $e(v_0, v_2)$ in Figure 1, it becomes apparent that the search traverses the same subgraphs induced by v_0, v_2, v_3 and v_0, v_2, v_4, resulting in redundant computations. It is crucial to emphasize that both these induced data subgraphs and the induced query subgraph, comprising vertices $\{u_0, u_1, u_2\}$, exhibit automorphic properties, implying that they are isomorphic to themselves. For instance, a mapping $\{(u_0, u_0), (u_1, u_2), (u_2, u_1)\}$ can be found when considering the query subgraph induced by $\{u_0, u_1, u_2\}$. While previous studies have utilized automorphisms to reduce redundant computation in cases where the original query graph is automorphic, we can further minimize redundant computation by acknowledging the presence of automorphic subgraphs within the given query subgraph. Given a graph g, if $e(v_i, v_j), e(v_x, v_y) \in E(g)$, and $M(v_i) = v_x, M(v_j) = v_y$, then $e(v_i, v_j)$ and $e(v_x, v_y)$ are equivalent. Hence, we define

Definition 3. *An edge set $E'(g) \subseteq E(g)$ is an equivalent edge set, if and only if any two edges $e(v_i, v_j) \in E'(g)$ and $e(v_x, v_y) \in E'(g)$ are equivalent.*

Given a graph g, if the induced subgraph obtained by removing k vertices from $V(g)$ is automorphic, it is referred to as a **k-degenerated automorphic subgraph** of g. The k-degenerated automorphic subgraph is denoted as $g^k = \{V^k, R^k, E^k, M^k\}$, where V^k denotes the remaining vertices that constitute induced subgraph, which is automorphic. R^k represents the removed vertex set, E^k denotes the equivalent

edge set w.r.t. the induced subgraph, and M^k is the mapping of the induced subgraph.

Example 4. *Consider the query graph Q in Figure 1(a). After removing u_3, the induced subgraph comprised of $\{u_0, u_1, u_2\}$ is a 1-degenerated automorphic subgraph of Q and is denoted as Q^1, where $e(u_0, u_1)$ and $e(u_0, u_2)$ are equivalent. Specifically, $V^1 = \{u_0, u_1, u_2\}$, $E^1 = \{e(u_0, u_1), e(u_0, u_2)\}$, $R^1 = \{u_3\}$, and $M^1 = \{(u_0, u_0), (u_1, u_2), (u_2, u_1)\}$.*

Thus, once a partial match M is acquired for a query edge in the equivalent edge set E^k of Q^k, we rearrange the query vertices in M to generate other partial matches without traversing the same data subgraph multiple times. For instance, given the partial match $M = \{(u_0, v_0), (u_1, v_2), (u_2, v_3)\}$ when mapping $e(v_0, v_2)$ to $e(u_0, u_1)$, since $e(u_0, u_1)$, $e(u_0, u_2) \in E^1$ of Q^1, according to M^1, we interchange the positions of u_1 and u_2 in M to obtain another partial match $M' = \{(u_0, v_0), (u_2, v_2), (u_1, v_3)\}$. Consequently, the search for the same data subgraph is conducted only once. This permutation operation is referred to **coalesced search**, as it coalesces the search for the partial matches of multiple permutations of V^k in Q^k.

To generate the set $\{g^k\}$, we begin with $k = 0$ and progressively remove k vertices to generate the induced subgraphs with different sizes. Subsequently, we examine whether each induced subgraph is automorphic. However, as a query edge e can belong to multiple entries in $\{g^k\}$, redundant matching occurs. Therefore, if an edge e is found in both $g_i^{k_i}, g_j^{k_j} \in \{g_k\}$, we heuristically apply the following rules:

1) If $k_i < k_j$, exclude e from $E_j^{k_j}$ of $g_j^{k_j}$;
2) If $k_i = k_j$, select $g_i^{k_i}$ if $|E_i^{k_i}| > |E_j^{k_j}|$.

Rule 1 aims to ensure that a larger data subgraph can be shared, reducing redundant matching. By excluding the common edge e from $E_j^{k_j}$ when $k_i < k_j$, we allow for the inclusion of a larger data subgraph during the matching process. Rule 2, on the other hand, focuses on maximizing the number of edges that can be shared. By selecting $g_i^{k_i}$ with a larger $|E_i^{k_i}|$ size when $k_i = k_j$, we prioritize the inclusion of more edges during the matching, leading to a more efficient exploration. Both rules contribute to reducing redundancy and improving the efficiency of matching by promoting the sharing of larger data subgraphs and maximizing the sharing of edges.

Avoid Invalid Matching. Considering Q^1 of Q in Figure 1, if we first map an updated edge to $e(u_0, u_1)$, we can obtain another valid partial match by interchanging the positions of u_1 and u_2. However, an invalid partial match can be generated if we first map the updated edge to $e(u_0, u_2)$, since a data vertex that matches u_1 should also possess neighbors labeled as C (which is not the case for u_2). In such a situation, we say $e(u_0, u_1)$ dominates $e(u_0, u_2)$ and designate $e(u_0, u_1)$ as a prioritized query edge. For each Q^k, we prioritize the matching of the prioritized query edge to avoid invalid matches.

Remark. Assuming that an update $\Delta e \in \Delta B$ maps to an edge $e^k \in E^k$, discovering an arbitrary partial match for V^k immediately yields $|E^k - 1|$ additional partial matches for the remaining edges in E^k through permutation. This leads to a potential speedup of $|E^k|$. However, query vertices in V^k adjacent to removed vertices may lose specific label constraints, expanding the candidate space for the remaining vertices. To address this, we selectively eliminate isolated query vertices with a degree of 1. Consolidating partial matches involving these vertices through parallel join operations proves significantly more efficient than individual vertex mappings. While the impact is minimal compared to DFS-based exploration, this practical approach ensures a speedup within the range of $(1, |E^k|)$ w.r.t the time complexity of Algorithm 1.

C. Practical Implementation and Optimizations

We further explore the integration of our system with existing dynamic graph data structures and how we address any inherent flaws that may arise.

Graph Container. In our study, we adopt GPMA [35] as the foundational data structure for its simplicity and efficiency. GPMA employs a sorted array, PMA, to manage edges by reserving spaces based on upper and lower thresholds. Each edge in a batch of updates is assigned a thread, locating the leaf segments to which it belongs. GPMA then groups edges within the same segment and materializes updates if the segment has sufficient space within the thresholds. If not, GPMA resorts to its parent, composed of two adjacent segments, in a bottom-up iterative process until all updates are processed.

Other Practical Optimizations. As the data structure resides in global memory and involves multiple threads executing memory access during the location step, minimizing the associated overhead is crucial. To address this, we optimize by loading the top-k layers into shared memory for efficient reading. Additionally, GPMA employs tailored strategies for insertions based on segment sizes. It utilizes the warp technique for segments up to 32 elements, employs the block method for segments fitting into shared memory, and resorts to the device approach for larger segments exceeding shared memory capacity. While the warp-based optimization may lead to suboptimal parallelism for segments with fewer than 32 elements, we leverage the Cooperative Group (CG) [22] to enhance flexibility in thread grouping within a warp. By partitioning the warp into smaller thread groups based on powers of 2 (e.g., 16, 8, etc.) and allocating them based on segment sizes, we address this issue. For instance, for segments in the 16 to 32 range, we initiate processing with 16-sized thread groups and adaptively allocate smaller groups as entries are processed. This adaptive strategy improves thread utilization and mitigates the stall issue.

VI. EXPERIMENTS

In this section, we evaluate the performance of the proposed system and conduct a comparative evaluation with existing state-of-the-art dynamic subgraph matching methods.

A. Experimental Setup

Datasets. We employ six datasets [40], [47] in our experiments. Detailed dataset descriptions are provided in Table II, where $|V|$ and $|E|$ refer to the numbers of vertices and edges,

TABLE II
SUMMARY OF THE DATASETS

Datasets	$|V|$	$|E|$	$|\Sigma_V|$	$|\Sigma_E|$	d_{avg}
Github (GH)	37.7K	0.3M	5	1	15.3
Skitter (ST)	1.7M	11.1M	25	1	13.1
Amazon (AZ)	0.4M	2.4M	6	1	12.2
LiveJournal (LJ)	4.9M	42.9M	30	1	18.1
Netflow (NF)	3.1M	2.9M	1	7	2.0
LSBench (LS)	5.2M	20.3M	1	44	8.2

TABLE III
OVERALL PERFORMANCE COMPARED WITH BASELINES

QS	DS	Method				
		TF	SYM	RF	CL	GAMMA
Dense	GH	127.746(5)	3.755	0.202	35.082	0.553
	ST	86.813	10.298	0.228	48.933 (1)	0.355
	AZ	7.251	1.385	0.328	7.291	0.469
	LJ	13.678	15.666	0.757	20.372	0.611
	NF	3.557 (1)	2.068	0.185	0.711(16)	0.51
	LS	3.216	3.979	0.443	73.205(20)	0.473
Sparse	GH	662.613(29)	493.747 (12)	140.979	401.436 (12)	8.110
	ST	269.747(23)	257.793(16)	66.325	139.643 (4)	7.218
	AZ	10.548	2.155	0.325	2.431	0.669
	LJ	39.857	14.076	0.680	1.165	0.715
	NF	238.746 (13)	65.609 (6)	1.612	1800(50)	0.99
	LS	63.706 (17)	34.658	4.730	1800(50)	1.469
Tree	GH	745.289(41)	1225.02(45)	366.981	744.025(22)	23.647
	ST	1543.516(46)	1107.324(41)	498.218	382.392(36)	33.465
	AZ	117.9(1)	245.124(2)	7.767	122.538	9.791
	LJ	97.283 (1)	231.416 (1)	8.473	67.651 (3)	2.049
	NF	323.534(140)	452.288 (15)	42.850(8)	1800(50)	5.369 (2)
	LS	119.361 (7)	140.030 (7)	15.434 (6)	1800(50)	5.384 (3)

respectively. Σ_V and Σ_E represent the quantities of vertex labels and edge labels, respectively. d_{avg} is the average. We set the insertion/deletion rate, namely the batch size, in the range of 2% to 10%, with **10%** being the default value. The insertion/deletion rate represents the proportion of the number of inserted/deleted edges to the total number of edges in the data graph.

Queries. Following precedent studies [40], [41], we generate query graphs by randomly extracting subgraphs from the data graph. The query graphs are categorized into Dense ($d_{avg} \geq 3$), Sparse ($d_{avg} < 3$), and Tree ($d_{avg} = |V_Q| - 1$), with d_{avg} representing the average degree of queries. For each type, we create query graphs while varying the number of vertices, $|V_Q|$, from 4 to 12. We produce a query set consisting of 50 query graphs of each size and type. By default, we present the results for query sets composed of query graphs with **6** vertices.

Baseline Methods. We conduct a comparative analysis between our method and state-of-the-art continuous subgraph graph matching algorithms, which encompass TurboFlux (TF) [27], SymBi (SYM) [34], RapidFlow (RF) [40] and CaLig (CL) [47]. Notably, as far as our knowledge extends, there exists no algorithm tailored for batch-dynamic graph subgraph matching specifically designed for the GPU architecture.

Running Platform. The experimental evaluations are carried out on an Ubuntu server, which is equipped with an Intel Core i9-10900X CPU and 128GB of host memory. Additionally, the server features an Nvidia Geforce RTX 3090 GPU, boasting 24GB of device memory and 83 streaming multiprocessors.

Metrics. The performance evaluation of each algorithm is based on the average query latency across all query graphs. To avoid excessively long running times, a time threshold of 30 minutes is set. If the execution of a query surpasses this threshold, it is terminated and classified as an unsolved query. Consequently, the percentage of solved queries is also reported as a performance metric. The average query time excludes queries that exceed the time threshold. In addition, we use GPU utilization to evaluate the performance of our method.
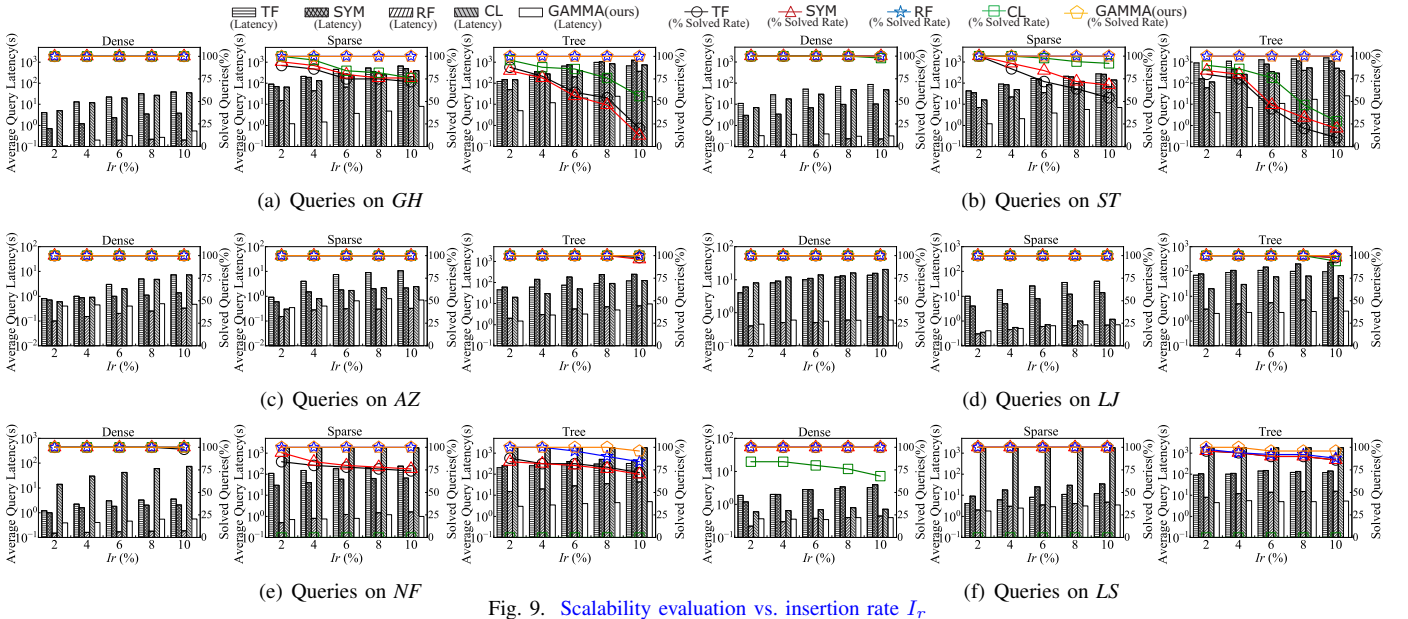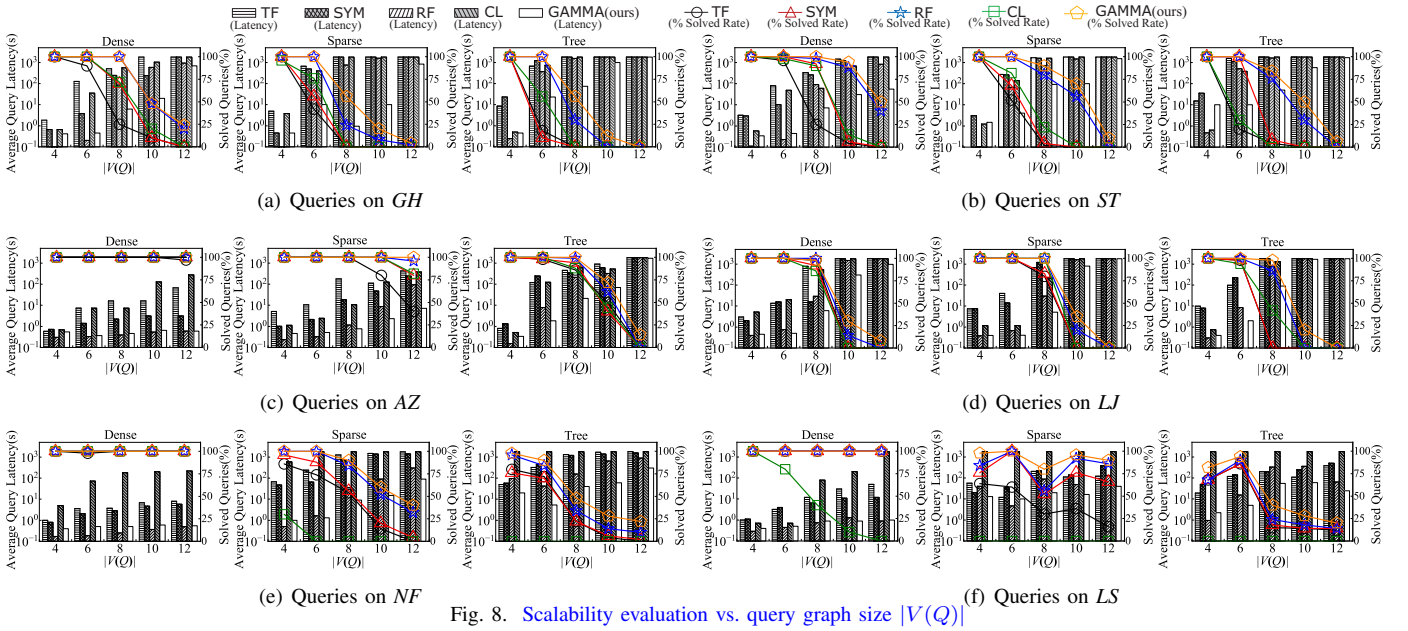
B. Overall Performance

Table III summarizes the performance comparisons. In this context, queries exceeding the predefined time limit have been excluded. The values outside parentheses signify the average query latency in seconds, while the values within parentheses indicate the count of unsolved queries. QS and DS respectively denote the dataset and query structure.

The first observation is that both the average query latency and the quantity of unresolved queries typically increase as query density decreases. This is attributable to the prevalence of power-law distributions in real-world graphs, wherein vertices with low degrees predominate, thus culminating in a notably substantial count of query results on sparse queries. Furthermore, as the query graph becomes sparser, it proffers fewer pruning constraints, specifically interconnections among vertices, resulting in a considerably expanded search space. Second, although LJ and LS are of comparable size, the performance on these two datasets diverges markedly. LJ presents a greater challenge when dealing with dense queries, while LS proves more formidable in handling sparse and tree queries. The underlying rationale is that LJ boasts a substantially higher average degree, thereby yielding a profusion of matches on dense queries, whereas LS, conversely, possesses a lower average degree, culminating in an abundance of matches on sparse and tree queries. Third, most of the methods exhibit longer query latency on NF compared to LS, despite NF having a smaller size. One possible reason is the highly skewed edge labels in NF, which in turn enlarges the search space. CL exhibits poor performance on edge-labeled graphs, primarily due to its requirement to transform them into vertex-labeled graphs. The transformation treats labeled edges as labeled vertices connecting two endpoints, thereby altering the graph structure and expanding the search space. RF outperforms both existing methods on all query sets, owing to the proposed query reduction and dual matching techniques. These techniques effectively eliminate invalid partial results and redundant computations caused by automorphisms in query graphs. Fourth, leveraging the abundant parallel potential offered by GPU and incorporating various practical optimization techniques, GAMMA demonstrates competitive or even optimal performance compared to existing algorithms across most query sets. The observed speedup ratio ranges from several times to tens of times when compared to the best baseline. In particular, our proposed method showcases substantial performance improvements of $67\times$, $33\times$, $5\times$ and $712\times$ on average, compared to TF, SYM, RF and CL, respectively. However, for some shorter-running queries, GAMMA demonstrates comparable performance with that of RF. This phenomenon arises from the limitation of such queries to

Fig. 8. Scalability evaluation vs. query graph size $|V(Q)|$

Fig. 9. Scalability evaluation vs. insertion rate I_r

sufficiently saturate the GPU with tasks. Moreover, GAMMA significantly reduces the number of unsolved queries compared to the baselines, with the majority of unsolved queries being tree queries resulting from the abundance of matches. This outcome serves as a clear indication of the scalability of our proposed method.

C. Scalability

We proceed to assess the impact of query graph size and insertion rate on performance. Due to space constraints and similar experimental outcomes, we only report the experimental results on *GH* and *ST*. The version with complete experimental results on all datasets is available on https://github.com/ZJU-DAILY/GAMMA/blob/main/GAMMA.pdf.
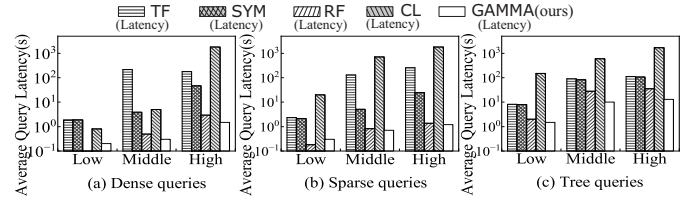
Fig. 10. Scalability evaluation vs. density on *LS*

Varying Query Graph Size. We assess the performance by varying the query graph size $|V(Q)|$ from 4 to 12. Figure 8 illustrates the corresponding results, where GAMMA consistently achieves the best performance. As observed, the average query latency generally increases, and the number
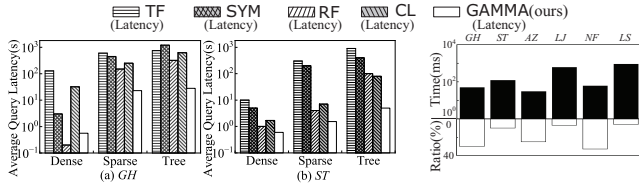
Fig. 11. Performance vs. mixed workloads   Fig. 12. Preprocessing analysis

of unsolved queries rises with the expansion of the query size due to the larger exploration space. With increasing query size, the performance gap between the baselines and our proposed method progressively widens. This phenomenon occurs because GAMMA effectively explores the large search space in parallel, while the baselines conduct the search sequentially. As the exploration space expands with the growth of graph size, the parallel approach becomes more crucial in showcasing the superiority of GAMMA. Moreover, GAMMA significantly outperforms the baselines in terms of the number of solved queries, especially concerning larger queries.

Varying Insertion Rate. We explore the performance under different insertion rates, ranging from 2% to 10%. The experimental results are presented in Figures 9. We have excluded the experimental results for varying deletion rates, as they exhibit a similar pattern. In general, the query time increases as the insertion rate rises. This is particularly evident when updating the indices of the baselines, as they involve the edges of the data graph. Consequently, the query time for a single update grows as the insertion progresses. On the other hand, RF updates the indices by taking into account both the query graph size and the average degree of the data graph. It conducts matching by leveraging the local index with a better matching order. By efficiently utilizing the parallelism offered by the GPU to amortize the query overhead, our proposed method achieves improved scalability.

Varying Density. We evaluate how the density of the update regions impacts the performance. Following the previous study [40], we perform k-core decomposition on LS and sample edges from these cores for insertions. We vary $k \in \{4, 8, 12\}$ to depict the density of regions (i.e., low, middle, and high density). Figure 10 illustrates the results. As observed, the runtime of all methods increases with the growth of density. Notably, our system exhibits a more pronounced acceleration in denser regions, courtesy of heightened parallelism and an optimally distributed workload.

Mixed Workloads. We evaluate the performance under mixed workloads of insertions and deletions. Following [47], we set the insertion-to-deletion ratio to 2 : 1. The experimental results are presented in Figure 11. Similar to the single workload scenario (i.e., Figure 9), the runtime of all the methods increases as the query set density decreases and our method outperforms all the competitors, showcasing the scalability of our approach under mixed workloads.

Efficiency of Preprocessing. We evaluate the performance of preprocessing. Preprocessing consists of CPU-based candidate generation and GPU-based graph update. The candidate generation operates asynchronously with GPU execution and

proves to be efficient. Consequently, the key factor affecting overall preprocessing performance is the graph update, which runs alongside incremental matching. Figure 12 provides insights into the graph update at a 10% update rate, where 'Time' denotes the graph update time and 'Ratio' represents the proportion of graph update time relative to the total running time. Notably, the graph update is primarily impacted by the volume of updates. As depicted, a larger data size, thereby a larger volume of updates, results in more pre-processing time.

D. Ablation Study

Lastly, we conduct an ablation study to assess the efficacy of each individual technique.

Effect of Stealing Strategy on GPU Utilization. We first evaluate the impact of work stealing on GPU utilization. The results are shown in Figure 13, where "ws" represents the work stealing optimization. In general, GAMMA with work stealing consistently achieves higher GPU utilization compared to GAMMA without work stealing (GAMMA w/o ws), with an average 17.5% increase and a peak improvement of 33.8%. Upon comparing the enhancements in utilization across different query sets, it becomes evident that denser queries exhibit a more modest improvement, owing to their comparatively reduced result sets and runtime. Consequently, this leads to a diminished disparity in cumulative execution time across warps. As query size and insertion rate increase, GPU utilization generally declines due to the expanding search space and workload, leading to larger disparities in cumulative execution time among warps. Additionally, as query size and insertion rate increase, the gap in utilization between schemes with and without work stealing progressively widens, demonstrating work stealing's effectiveness in balancing workloads among different warps and improving GPU utilization.

Effect of Stealing Strategy and Coalesced Search on Execution Time. We further evaluate the influence of various techniques on performance. The results are reported in Figure 14, where "cs" and "ws" denote the coalesced search and work stealing, respectively. The initial findings reveal that all the other implementations outperform WBM without optimizations, confirming the effectiveness of our proposed techniques. Notably, the load-balanced implementation (WBM+ws) executes faster than that with coalesced search (WBM+cs), underscoring the paramount importance of sophisticated load balancing techniques. When comparing different query sets, we discern significantly higher speedup ratios for sparser queries, mainly due to their larger search space. Coalesced search effectively curtails this search space, resulting in noteworthy improvements in speedup. Consequently, the speedup ratios for sparse and tree queries are substantially greater in comparison to dense queries. Overall, the coalesced search achieves a speedup ranging from $1.1\times$ to $1.9\times$, and the work stealing delivers performance enhancements ranging from $1.2\times$ to $6.4\times$. In conclusion, our well-conceived optimization techniques significantly enhance performance.
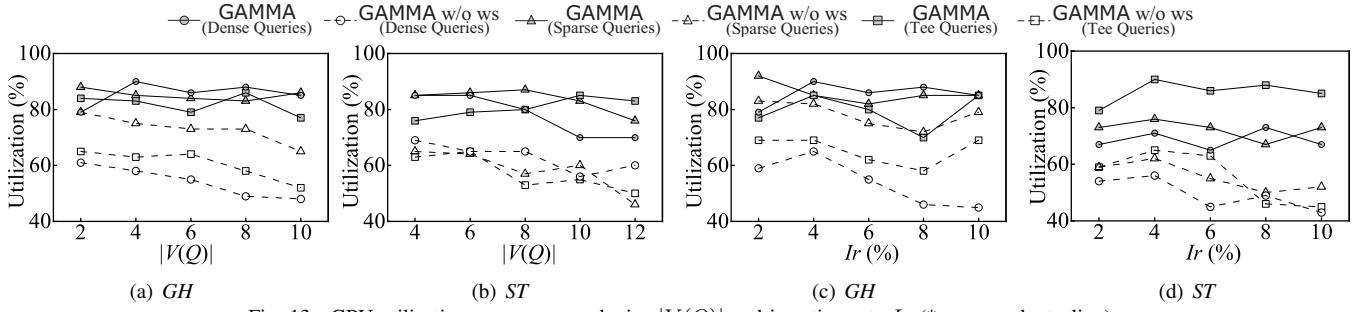
Fig. 13. GPU utilization vs. query graph size $|V(Q)|$ and insertion rate I_r (*ws = work stealing)
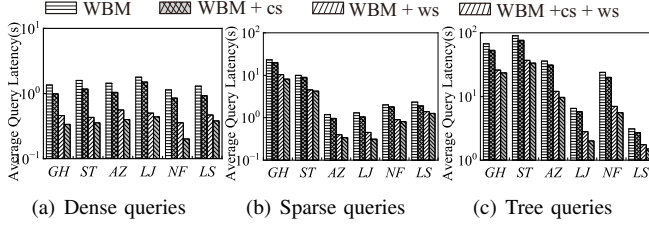

Fig. 14. Ablation study

VII. CONCLUSION

This paper introduces GAMMA, an efficient parallel subgraph matching system tailored for batch-dynamic graphs. Our system harnesses a warp-centric parallel algorithm as its core, adeptly managing each update. To achieve balanced workloads among warps within a block, we implement a work stealing mechanism that effectively utilizes shared memory. Moreover, we integrate a coalesced search technique to mitigate redundant computations arising from automorphisms of subgraphs in the query graph. Lastly, we synergize these techniques with multiple other optimizations, culminating in a comprehensive bottom-up batch-dynamic subgraph matching system. Experiments conducted on four real-world datasets substantiate that our system surpasses state-of-the-art methods by a substantial margin. In the future, we envision expanding GAMMA's capabilities to address more general subgraph matching challenges within batch-dynamic scenarios.

REFERENCES

[1] U. A. Acar, D. Anderson, G. E. Blelloch, and L. Dhulipala. Parallel batch-dynamic graph connectivity. In *SPAA*, page 381–392, 2019.
[2] M. A. Bender and H. Hu. An adaptive packed-memory array. *TODS*, 32(4):26, 2007.
[3] M. Besta, R. Grob, C. Miglioli, N. Bernold, G. Kwasniewski, G. Gjini, R. Kanakagiri, S. Ashkboos, L. Gianinazzi, N. Dryden, et al. Motif prediction with graph neural networks. In *SIGKDD*, pages 35–45, 2022.
[4] B. Bhattarai, H. Liu, and H. H. Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *SIGMOD*, page 1447–1462, 2019.
[5] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, page 1199–1214, 2016.
[6] M. Bisson and M. Fatica. High performance exact triangle counting on gpus. *TPDS*, 28(12):3501–3510, 2017.
[7] F. Busato, O. Green, N. Bombieri, and D. A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *HPEC*, pages 1–7, 2018.
[8] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at facebook. In *SIGMOD*, pages 1087–1098, 2016.
[9] X. Chen and Arvind. Efficient and scalable graph pattern mining on GPUs. In *OSDI*, pages 857–877, 2022.
[10] X. Chen, R. Dathathri, G. Gill, and K. Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *PVLDB*, 13(8):1190–1205, 2020.
[11] S. Choudhury, L. Holder, G. Chin, K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. In *EDBT*, pages 157–168, 2015.
[12] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *TPAMI*, 26(10):1367–1372, 2004.
[13] L. Dhulipala, D. Durfee, J. Kulkarni, R. Peng, S. Sawlani, and X. Sun. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *SODA*, pages 1300–1319, 2020.
[14] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *TODS*, 38(3), 2013.
[15] M. Farhan, Q. Wang, and H. Koehler. Batchhl: Answering distance queries on batch-dynamic networks at scale. In *SIGMOD*, page 2020–2033, 2022.
[16] J. Gao, C. Zhou, and J. X. Yu. Toward continuous pattern detection over evolving large graph with snapshot isolation. *VLDBJ*, 25:269–290, 2016.
[17] O. Green and D. A. Bader. Custinger: Supporting dynamic graph algorithms for gpus. In *HPEC*, pages 1–6, 2016.
[18] O. Green, J. Fox, A. Watkins, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. A. Bader. Logarithmic radix binning and vectorized triangle counting. In *HPEC*, pages 1–7, 2018.
[19] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *SIGMOD*, page 1067–1082, 2020.
[20] S. Han, L. Zou, and J. X. Yu. Speeding up set intersections in graph algorithms using simd instructions. In *SIGMOD*, pages 1587–1602, 2018.
[21] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, page 337–348, 2013.
[22] M. Harris and K. Perelygin. Cooperative groups: Flexible cuda thread programming. https://developer.nvidia.com/blog/cooperative-groups/.
[23] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *SIGMOD*, page 405–418, 2008.
[24] A. Iyer, L. E. Li, and I. Stoica. {CellIQ}:{Real-Time} cellular network analytics at scale. In *NSDI*, pages 309–322, 2015.
[25] K. Jamshidi, R. Mahadasa, and K. Vora. Peregrine: A pattern-aware graph mining system. In *EuroSys*, pages 1–16, 2020.
[26] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An active graph database. In *SIGMOD*, pages 1695–1698, 2017.
[27] K. Kim, I. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *SIGMOD*, page 411–426, 2018.
[28] W. Lin, X. Xiao, X. Xie, and X.-L. Li. Network motif discovery: A gpu approach. *TKDE*, 29(3):513–528, 2017.
[29] Q. C. Liu, J. Shi, S. Yu, L. Dhulipala, and J. Shun. Parallel batch-dynamic algorithms for k-core decomposition and related graph problems. In *SPAA*, pages 191–204, 2022.
[30] Y. Lü, H. Guo, L. Huang, Q. Yu, L. Shen, N. Xiao, and Z. Wang. Graphpeg: Accelerating graph processing on gpus. *TACO*, 18(3), 2021.
[31] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu. Graphzero: A high-performance subgraph matching system. *SIGOPS Oper. Syst. Rev.*, 55(1):21–37, 2021.

[32] N. May and N. Tatbul. International workshop on data management on new hardware (damon). In *SIGMOD*, page 299–300, 2023.

[33] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *PVLDB*, 12(11):1692–1704, 2019.

[34] S. Min, S. G. Park, K. Park, D. Giammarresi, G. F. Italiano, and W.-S. Han. Symmetric continuous subgraph matching with bidirectional dynamic programming. *PVLDB*, 14(8):1298–1310, 2021.

[35] M. Sha, Y. Li, B. He, and K.-L. Tan. Accelerating dynamic graph analytics on gpus. *PVLDB*, 11(1):107–120, 2017.

[36] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[37] J. Shi, R. Yang, T. Jin, X. Xiao, and Y. Yang. Realtime top-k personalized pagerank over large graphs on gpus. *PVLDB*, 13(1):15–28, 2019.

[38] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua. Graph processing on gpus: A survey. *ACM Comput. Surv.*, 50(6), 2018.

[39] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *SIGMOD*, pages 1083–1098, 2020.

[40] S. Sun, X. Sun, B. He, and Q. Luo. Rapidflow: An efficient approach to continuous subgraph matching. *PVLDB*, 15(11):2415–2427, 2022.

[41] X. Sun, S. Sun, Q. Luo, and B. He. An in-depth study of continuous subgraph matching. *PVLDB*, 14(2):176–188, 2020.

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

[43] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens. Gunrock: Gpu graph analytics. *TOPC*, 4(1), 2017.

[44] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger. Faimgraph: High performance management of fully-dynamic graphs under tight memory constraints on the gpu. In *SC*, pages 754–766, 2018.

[45] M. Winter, R. Zayer, and M. Steinberger. Autonomous, independent management of dynamic graphs on gpus. In *HPEC*, pages 1–7, 2017.

[46] M. Xekalaki, J. Fumero, A. Stratikopoulos, K. Doka, C. Katsakioris, C. Bitsakos, N. Koziris, and C. Kotselidis. Enabling transparent acceleration of big data frameworks using heterogeneous hardware. *PVLDB*, 15(13):3869–3882, 2022.

[47] R. Yang, Z. Zhang, W. Zheng, and J. X. Yu. Fast continuous subgraph matching over streaming graphs via backtracking reduction. *PACM-MOD*, 1(1):1–26, 2023.

[48] M. Yu, L. Qin, Y. Zhang, W. Zhang, and X. Lin. Dptl+: Efficient parallel triangle listing on batch-dynamic graphs. In *ICDE*, pages 1332–1343, 2021.

[49] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang. Gsi: Gpu-friendly subgraph isomorphism. In *ICDE*, pages 1249–1260, 2020.

[50] S. Zhang, S. Li, and J. Yang. Gaddi: Distance index based subgraph matching in biological networks. In *EDBT*, page 192–203, 2009.

[51] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1):340–351, 2010.