Efficient GPU-Based Continuous Subgraph Matching on Batch Updates

Guanghua Li[1⋆], Xibo Sun[2⋆], Qiong Luo[2,1], and Lijun Chang[3]

[1] The Hong Kong University of Science and Technology (Guangzhou), China
gli945@connect.hkust-gz.edu.cn
[2] The Hong Kong University of Science and Technology, Hong Kong SAR
xsunax@connect.ust.hk, luo@cse.ust.hk
[3] The University of Sydney, Sydney, Australia
lijun.chang@sydney.edu.au

Abstract. Continuous subgraph matching (CSM) algorithms process edge updates (insertions and deletions) on the data graph and find new matches of the query graph that either appear or disappear as a result of each update. Existing CSM algorithms organize the matching process by the edge update to the data graph. We refer to this approach as update-oriented CSM (UO-CSM). While UO-CSM effectively handles single updates, directly applying it to a batch of updates can lead to redundant computations and duplicate results. To address this problem, existing work uses a deduplication technique based on update-edge ordering, which entails additional data structures to record whether an edge is an update-edge and an explicit duplication-removal step during the matching result enumeration. We propose a novel query-oriented CSM algorithm (QO-CSM), which avoids duplicate matches and ensures consistent results on dynamic graphs with batch updates, without additional data structures and the explicit de-duplication step. Our main idea is to put each update (insertion or deletion) edge to the data graph into its corresponding query edge delta relation ΔR, if any, and then process these ΔRs in order. Building on this paradigm, we develop a GPU-based CSM algorithm, GCSM-BU, which employs a two-level indexing mechanism for each ΔR and utilizes parallel breadth-first or depth-first search, based on runtime workload, for result enumeration. With the QO-CSM paradigm and effective GPU parallelism, GCSM-BU achieves correct query results on batch updates, while outperforming state-of-the-art CPU-based CSM methods by up to two orders of magnitude. It also outperforms existing GPU-based UO-CSM methods as well as their QO-CSM adaptations.

Keywords: Continuous Subgraph Matching · GPU Acceleration

1 Introduction

Subgraph matching (SM), which extracts all matches in a data graph G that are isomorphic to a query graph Q, is a common operation in various graph an-

⋆ Equal Contribution.

Table 1: Categorization of CSM algorithms. **UO** is for **U**pdate-**O**riented. **QO** is for **Q**uery-**O**riented.

Handle batches?	**UO-CSM**	**QO-CSM**
Yes	GAMMA [24]	GCSM-BU (ours)
No	SymBi [22], CaLiG [39], RapidFlow [32]	

alytic tasks [9, 25]. Since many real-life graphs are dynamic, with edges inserted and deleted over time, researchers have investigated the problem of continuous subgraph matching (CSM), which finds *incremental matches*, i.e., matches of the query graph Q in the data graph G that appear or disappear due to each edge update. Existing CSM algorithms [6, 8, 10, 15, 16, 18, 22] organize the matching process by the edge update to the data graph. We refer to this approach as update-oriented CSM (UO-CSM). While UO-CSM effectively handles single updates, directly applying it to a batch of updates can lead to redundant computations and duplicate results, resulting in efficiency and consistency issues. In this paper, we propose a consistent and efficient approach to CSM with batch updates, without requiring additional data structures or explicit steps for de-duplication.

Most of the existing CSM algorithms process a single updated edge at a time and are designed for running with a single thread on the CPU. However, some real-world dynamic graphs evolve at an extremely high speed [2,28], e.g., 143,000 new tweets appear in the Twitter graph within a second [26]. In addition, batch updates are common in real-world graph analytics where updates are grouped and periodically processed [8, 11, 20]. Such high-rate and batch-based graph updates are challenging for single-update based algorithms to catch up with. To process batch updates efficiently, a recent GAMMA algorithm [24] parallelized the matching process on the GPU. GAMMA's matching speed was much faster than that of CPU-based single-update algorithms, but it must explicitly remove duplicates due to its direct application of UO-CSM to a batch of updates. This explicit de-duplication technique relies on additional data structures and a duplicate removal step during matching result enumeration, occupying additional GPU memory and degrading the overall performance. We will show that the UO-CSM framework is efficient only in the special case of single-update and subgraph isomorphism.

Specifically, we first study the existing UO-CSM framework and identify its problem. We formulate the execution of algorithms based on the UO-CSM framework as the evaluation of relational algebra (RA) expressions, and explain why the existing framework is at risk of redundant computation and requires explicit duplicate removal. Based on our findings, we propose a new query-oriented CSM framework (QO-CSM) that can handle both single updates and batch updates correctly and efficiently.

Following our proposed QO-CSM framework, we implement GCSM-BU, a GPU-accelerated continuous subgraph matching algorithm for batch updates. Table 1 shows our GCSM-BU in comparison with existing work on the processing scheme and update size. One challenge of implementing such a QO-CSM algorithm on the GPU is the balance between parallelism and memory footprint. We adopt a dynamic strategy for searching matching results: we start with a parallel breadth-first-search and then switch to depth-first-search when the partial matches become sufficiently large. Another challenge is the load balance among GPU thread warps, which is caused by the skewness of vertex degree distribution in data graphs. The performance degradation due to load imbalance is prominent when we match the leaf vertices in the query graph. We introduce the backtracking flattening technique for leaf vertex matching to achieve a balanced workload distribution on the GPU. In summary, we make the following contributions.

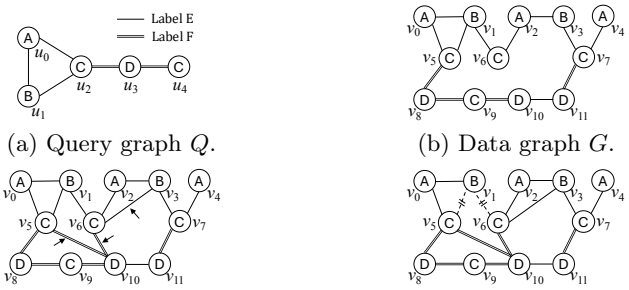
- We study the existing UO-CSM framework and point out the problem in applying it directly to batch updates. To address this issue, we propose the QO-CSM framework that supports parallel CSM processing for batch updates.
- Based on the proposed framework, we implement our own CSM algorithm GCSM-BU, which runs on the GPU and processes batch updates efficiently.
- We adopt a dynamic search strategy for matching results, balancing the parallelism of the algorithm and the GPU memory footprint. We also propose the backtracking flattening technique for leaf vertex matching to avoid load imbalance.
- We conduct extensive experiments to demonstrate the overall efficiency of our algorithm, the efficiency of the QO-CSM framework, and the performance benefit of each individual technique.

2 Preliminaries

2.1 Problem Statement

This paper focuses on edge- and vertex-labeled undirected graphs denoted by $g = (V, E)$, where V is the set of vertices and E is the set of edges. We use $V(g)$ and $E(g)$ to denote the vertex set and edge set of the graph g, respectively. The notation $e(u_x, u_y)$ represents the edge connecting vertices u_x and u_y. For a given vertex $u \in V(g)$, $N(u)$ denotes the set of neighbors, i.e., the vertices adjacent to u, and $d(u) = |N(u)|$ is the degree of vertex u. Q represents the *query graph* and G is the *data graph*.

Definition 1. *Given graphs g and g', a subgraph isomorphism is a* ***injective mapping*** *M: $V(g) \to V(g')$ such that 1) $\forall u \in V(g), L(u) = L(M(u))$; 2) $\forall e(u, u') \in E(g), e(M(u), M(u')) \in E(g')$; and 3) $\forall e(u, u') \in E(g), L(e(u, u')) = L(e(M(u), M(u')))$.*



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

(c) G': Insert edges into G. (d) G'': Delete edges from G'.

Fig. 1: Example graphs.

Subgraph matching (SM) finds all isomorphisms of Q in G. The set of all isomorphisms (matches) is denoted by $\mathcal{M}$.

In this paper, we consider a dynamic data graph G, which changes over time. $\Delta\mathcal{G}$ is a sequence of graph updates $\{(\oplus_1, \Delta G_1), (\oplus_2, \Delta G_2), \cdots)\}$ on G. Each $\oplus$ is a graph update operation that can be either insertion ($+$) or deletion ($-$). Each ΔG contains a set of update edges $\{e\}$. The data graph edges to be deleted must exist in G, whereas the edges to be inserted must not already exist in G. $|\Delta G|$ denotes the *batch size*. Usually, $|\Delta G| > 1$. When the update ΔG is applied to G, resulting in a modified graph G', we can identify the *incremental matches* $\Delta\mathcal{M}$, i.e., the differences between the matches $\mathcal{M}$ of Q in G and $\mathcal{M}'$ of Q in G'. We define the *continuous subgraph matching* problem in Definition 2.

Definition 2 (Problem Statement). *Given Q, G and $\Delta\mathcal{G}$, continuous subgraph matching (CSM) finds all incremental matches $\Delta\mathcal{M}$ for each $(\oplus, \Delta G) \in \Delta\mathcal{G}$.*

We follow the practice in graph databases that data graph edges are grouped by their edge types. The *edge type* is defined as a 3-tuple (source-vertex label, edge label, destination-vertex label). Each edge group is considered a relation.

2.2 The Search Procedure in Subgraph Matching

Existing subgraph matching algorithms enumerate matching results with a backtracking search procedure, which maps each query vertex to a data graph vertex along a *matching order* φ. An order φ is a permutation of query vertices. We denote $u_x \prec_\varphi u_y$ if u_x appears before u_y in φ. $N^\varphi_+(u)$ (resp., $N^\varphi_-(u)$) denotes the neighbors of u positioned before (resp., after) u, namely the backward (resp., forward) neighbors. We use $\varphi[i]$ to represent the i-th query vertex (counting from 1) and $\varphi[i, j]$ is the subsequence of φ from position i to j (both ends included).

Algorithm 1 shows the matching result search procedure, commonly used in both SM and CSM [32]. A is an index of query edges to data edges. $A^u_{u'}$ contains the candidate edges in the data graph for the query edge $e(u', u)$, $A^u_{u'}[v']$ is the set of v''s neighbors found in $A^u_{u'}$, with v' matching u' and neighbors matching

Algorithm 1: The Backtracking Search Procedure

```
1 Procedure ENUMERATE(φ, A, M, i)
2     if i = |φ| + 1 then Output M, return;
3     else if i = 1 then u ← φ[1], C_M(u) ← π_{φ[1]} A^{φ[2]}_{φ[1]};
4     else u ← φ[i], C_M(u) ← ⋂_{u'∈N^φ_+(u)} A^u_{u'}(M(u'));
5     foreach v ∈ C_M(u) do
6         if v is not visited then //remove "if" to find homomorphism
7             Add (u, v) to M;
8             ENUMERATE(φ, A, M, i + 1);
9             Remove (u, v) from M;
```

u. i is the recursion depth (counting from 1) and M is a partial matching result that contains the mappings for $\varphi[1, i-1]$. $C_M(u)$ is the *enumeration candidate set* for query vertex u. At Lines 5-9 of Algorithm 1, we loop over vertices in $C_M(u)$ and extend M with one mapping for u inside each loop. Line 6 performs the isomorphism check. If all query vertices have been mapped, Line 2 outputs one matching result M. If the current recursion depth is 1, we generate the enumeration candidate set $C_M(u)$ by taking the candidates for u in the index $A^{\varphi[2]}_{\varphi[1]}$; otherwise, $C_M(u)$ is set to the intersection of the neighbor sets for all candidates that match the backward neighbors of u. In practice, the set intersection operation is usually implemented as a traversal of the neighbor set of one backward neighbor's match, followed by a check on whether each neighbor also belongs to the neighbor sets of other backward neighbors' matches. As previous work [31] has shown, if we view each $A^u_{u'}$ as a relation and remove the isomorphism check at Line 6, the ENUMERATE procedure in Algorithm 1 is equivalent to a multi-way join for relations $\{A^u_{u'} \in A \mid e(u', u) \in E(Q) \wedge u' \prec_\varphi u\}$.

3 Related Work

Subgraph Matching. Ullmann et al. [35] first proposed to solve subgraph matching with backtracking. Since then, extensive techniques have been proposed to improve the matching efficiency [4, 7, 14, 17, 27, 29, 30]. Recent work has started to utilize modern hardware, including GPU-based systems such as GunRockSM [36], GSI [40], and CuTS [38]. These methods enumerate results in BFS order. STMatch [37] adopted DFS, and EGSM [33] developed a DFS method optimized for GPU warps.

Continuous subgraph matching has been explored extensively on CPUs [6, 8, 10, 15, 18]. Various CPU algorithms have been proposed, including SymBi [22], CaLiG [39], RapidFlow [32], and NewSP [19]. Most recently, Qiu et al. designed the GPU-based CSM method, GAMMA [24], to process batch updates, and it significantly outperforms state-of-the-art CPU-based CSM methods that handle single updates. However, since GAMMA follows the existing UO-CSM al-

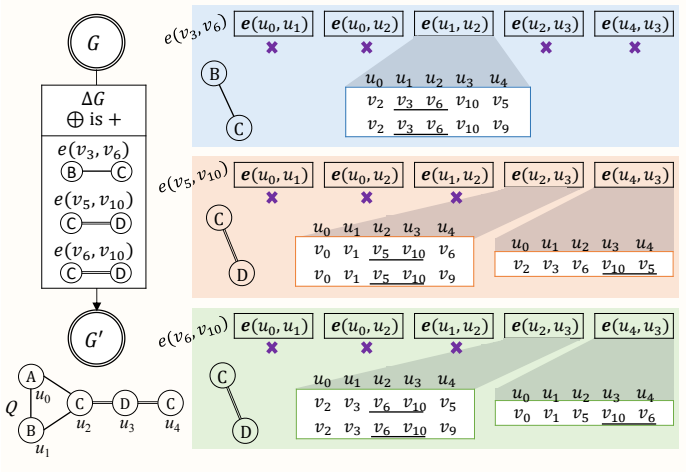


Fig. 2: Overview of the existing UO-CSM framework (w/o de-duplication).

gorithmic framework, it suffers from the memory and time cost of explicit de-duplication.

4 Framework Design

4.1 Foundation of an Efficient CSM Framework

Given the candidate relation for each query edge, subgraph isomorphism query processing is equivalent to performing a multi-way join as shown in Equation 1 [1, 8, 15, 16, 18, 21, 22].

$$\mathcal{M} = \sigma_{iso}(\bowtie_{e(u_x,u_y)\in E(Q)} R(u_x, u_y)) \tag{1}$$

Equation 1 can be decomposed as shown in Equation 2 [3, 12, 13, 34].

$$\begin{aligned} &\Delta\mathcal{M} = \bigcup_i \Delta\mathcal{M}_i. \\ &\Delta\mathcal{M}_1 = \sigma_{iso}(\Delta R_1 \bowtie R_2 \bowtie \cdots \bowtie R_m), \\ &\Delta\mathcal{M}_i = \sigma_{iso}(R'_1 \bowtie \cdots \bowtie R'_{i-1} \bowtie \Delta R_i \bowtie R_{i+1} \bowtie \cdots \bowtie R_m), \\ &\Delta\mathcal{M}_m = \sigma_{iso}(R'_1 \bowtie \cdots \bowtie R'_{m-1} \bowtie \Delta R_m), \end{aligned} \tag{2}$$

We have the following proposition.

Proposition 1. *In Equation 2,* $\forall i, j \in [1, m]$ *and* $i \neq j, \Delta\mathcal{M}_i \cap \Delta\mathcal{M}_j = \emptyset$

If we decompose the CSM problem into subtasks corresponding to $\Delta\mathcal{M}_i$, there will be no duplicate results across subtasks.

Algorithm 2: Existing UO-CSM for Batch Updates

Input: query graph Q, initial data graph G, update stream $\Delta\mathcal{G}$
Output: incremental matches $\Delta\mathcal{M}$ for each $(\oplus, \Delta G) \in \Delta\mathcal{G}$

```
1  I ← build an index based on Q and G;
2  foreach (⊕, ΔG) ∈ ΔG do
3      if ⊕ is + then
4          G ← G ⊕ ΔG and update I;
5          foreach e(v_a, v_b) ∈ ΔG do // parallel or sequential
6              FindIncrementalMatchWithDuplicateRemoval(Q, I, e(v_a, v_b));
7      else // ⊕ is −
8          // Omitted.
9  Procedure
       FindIncrementalMatchWithDuplicateRemoval(Q, I, e(v_a, v_b))
10     ΔM ← {};
11     foreach e(u_a, u_b) ∈ E(Q) do // parallel or sequential
12         if e(v_a, v_b) ≡ e(u_a, u_b) then
13             φ ← generate a matching order;
14             A ← build a local index based on I (or just use I);
15             M ← {(u_a, v_a), (u_b, v_b)};
16             ΔM_{e(u_a,u_b)} ← EnumerateWithDuplicateRemoval(φ, A, M, 3);
17             Add results in ΔM_{e(u_a,u_b)} into ΔM;
18     Output ΔM;
```

4.2 Why is Explicit De-duplication Required in UO-CSM?

Algorithm 2 [34] presents the existing CSM framework. Figure 2 shows an example of using the existing framework to find incremental matches for the data graph in Figure 1. In Algorithm 2, each iteration in the for-loop in the function FindIncrementalMatch corresponds to a select-join expression evaluation. Specifically, if we number the query edges in the order they appear in the for-loop at Line 13 in Algorithm 2, then collecting the results of the i-th loop (counting from 1) from each update edge in ΔG is equivalent to performing $\sigma_{iso}(R'_1 \bowtie \cdots \bowtie R'_{i-1} \bowtie \Delta R_i \bowtie R'_{i+1} \bowtie \cdots \bowtie R'_m)$. Note that all relations (except the i-th one) are updated versions, because before executing FindIncrementalMatch, we have updated both the data graph G (candidate relations) and the index I with the newly inserted edges in the current batch. ΔR_i contains the update edges that match the query edge e_i. In short, Algorithm 2

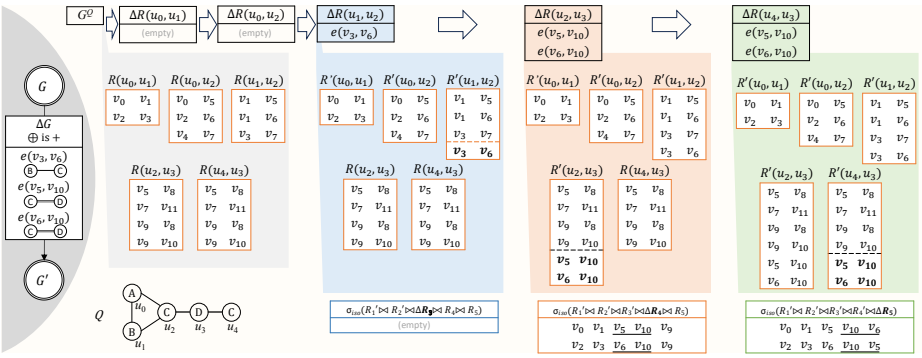


Fig. 3: Overview of the proposed QO-CSM framework.

corresponds to Equation 3:

$$
\begin{aligned}
&\Delta\mathcal{M} = \bigcup_i \Delta\mathcal{M}'_i. \\
&\Delta\mathcal{M}'_1 = \sigma_{iso}(\Delta R_1 \ltimes R'_2 \ltimes \cdots \ltimes R'_m), \\
&\Delta\mathcal{M}'_i = \sigma_{iso}(R'_1 \ltimes \cdots \ltimes R'_{i-1} \ltimes \Delta R_i \ltimes R'_{i+1} \ltimes \cdots \ltimes R'_m), \\
&\Delta\mathcal{M}'_m = \sigma_{iso}(R'_1 \ltimes \cdots \ltimes R'_{m-1} \ltimes \Delta R_m),
\end{aligned}
\tag{3}
$$

Note that all relations R in Equation 3 are updated versions. Proposition 1 does not hold for Equation 3. As a result, explicit duplicate removal is required for Algorithm 2.

Example 1. In Figure 2, the matches $(v_2, v_3, v_6, v_{10}, v_5)$, $(v_2, v_3, v_6, v_{10}, v_9)$, and $(v_0, v_1, v_5, v_{10}, v_6)$ are duplicated and must be explicitly removed.

4.3 Our Proposed QO-CSM Framework

Algorithm 3 shows our proposed QO-CSM framework. Figure 3 gives an example of using QO-CSM to find incremental matches for the data graph G in Figure 1.

In Algorithm 3, we use G^Q to denote a collection of raw candidate relations for query Q. The difference between Algorithm 3 and Algorithm 2 is that, in Algorithm 3, we put each update edge into a ΔR according to its edge type. A ΔR corresponds to a query edge. A single update edge can be added to multiple ΔRs, because multiple query edges may share the same edge type. We update G^Q one ΔR at a time. FindIncremental is interleaved with the ΔR updates. Inside FindIncremental, we just use the Enumerate procedure as in the static subgraph matching, without the need to perform duplicate removal. For the iteration on ΔR_i at Line 6, the raw candidate relations for query edges of previous iterations ΔR_1, ..., ΔR_{i-1}, have been updated, and the updates ΔR_{i+1}, ..., ΔR_m have not been applied. Hence, iteration i can be expressed as

Algorithm 3: Proposed QO-CSM Framework

Input: query graph Q, a set of raw candidate relations $G^Q = \{R(u_a, u_b) | e(u_a, u_b) \in E(Q)\}$, update stream $\Delta\mathcal{G}$
Output: incremental matches $\Delta\mathcal{M}$ for each $(\oplus, \Delta G) \in \Delta\mathcal{G}$

$I \leftarrow$ build an index based on Q and G^Q;
foreach $(\oplus, \Delta G) \in \Delta\mathcal{G}$ **do**
  foreach $e(u_a, u_b) \in E(Q)$ **do**
    $\Delta R(u_a, u_b) \leftarrow \{e | e \in \Delta G \wedge e \equiv e(u_a, u_b)\}$;
  if $\oplus$ *is* $+$ **then**
    foreach $e(u_a, u_b) \in E(Q)$ **do** // `sequential`
      if $\Delta R(u_a, u_b) = \emptyset$ **then continue**;
      $R(u_a, u_b) \leftarrow R(u_a, u_b) \oplus \Delta R(u_a, u_b)$, update I;
      FindIncremental$(Q, I, e(u_a, u_b), \Delta R(u_a, u_b))$;
  else // $\oplus$ `is` $-$
    foreach $e(u_a, u_b) \in E(Q)$ **do** // `sequential`
      if $\Delta R(u_a, u_b) = \emptyset$ **then continue**;
      FindIncremental$(Q, I, e(u_a, u_b), \Delta R(u_a, u_b))$;
      $R(u_a, u_b) \leftarrow R(u_a, u_b) \oplus \Delta R(u_a, u_b)$, update I;
Procedure *FindIncremental*$(Q, I, e(u_a, u_b), \Delta R(u_a, u_b))$
  // Omitted.

$\sigma_{iso}(R'_1 \bowtie \cdots \bowtie R'_{i-1} \bowtie \Delta R_i \bowtie R_{i+1} \bowtie \cdots \bowtie R_m)$, which is exactly $\Delta\mathcal{M}_i$ in Equation 2. According to Proposition 1, no de-duplication is needed.

The QO-CSM framework also applies to edge deletions. The main difference lies in the order between enumeration and relation/index update. For an insertion batch, GCSM-BU first inserts the corresponding query-edge delta relation ΔR_i into R_i and updates the index, and then enumerates the newly generated matches anchored at ΔR_i. For a deletion batch, GCSM-BU performs these two steps in the reverse order: it first enumerates the matches that contain edges in ΔR_i, which are the matches that will disappear after the deletion, and then removes ΔR_i from R_i and updates the index. Therefore, deletion can be handled by the same query-oriented delta-relation processing principle, with the update operation placed after enumeration. Our experimental evaluation will focus on insertion streams, following the experimental settings of prior CSM studies [32].

5 Implementation

In this section, we describe the implementation of our GPU-based CSM algorithm GCSM-BU, following the proposed CSM framework in Algorithm 3. Figure 4 gives an overview.

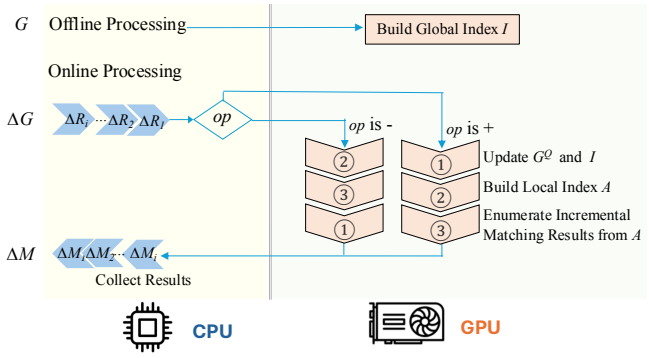


Fig. 4: Overview of GCSM-BU.

5.1 Data Structures

Given a data graph G and a query graph Q, our QO-CSM framework operates on a collection of raw candidate relations G^Q. We store a raw candidate relation $R(u_x, u_y)$ using two storage units, considering u_x and u_y as source vertices, respectively, to facilitate neighbor traversal in both directions. A storage unit is a data structure called a consecutive adjacency array, which consists of three arrays, namely *ptrs*, *capacities*, and *degrees*, indexed by source vertex IDs. These arrays maintain a dynamic array for each neighbor list.

5.2 Two-Level Indexing Mechanism

We utilize the two-level indexing mechanism following the CPU-based algorithm RapidFlow [32], and apply it to the batch-update case, using a GPU implementation. When the query graph is dense, the time cost of local index building for a batch of update edges becomes significant. Hence, we decide to build the local index only when the query graph is acyclic or the average degree of the query graph is smaller than a threshold.

5.3 Search Strategy

We call the search strategy used by GCSM-BU for result enumeration *dynamic semi-BFS*, which takes advantage of both BFS and DFS search on the GPU. The execution of the ENUMERATE function as described in Algorithm 1 forms a search tree. Each pair of (an updated edge e, a query edge e_i that can be mapped to e) corresponds to one such search tree. In the case of depth-first-search (DFS), a search tree is assigned to a GPU warp, and the warp performs DFS on that tree. In contrast, breadth-first-search (BFS) performs the search layer by layer. At each layer, we assign a tree node to a GPU warp, which is responsible for enumerating the children of that tree node. BFS has a high

GPU memory consumption and DFS may fail to fully utilize the parallelism of a GPU. In dynamic semi-BFS, we first perform the BFS until the size of the intermediate results reaches a pre-defined threshold. Then, we assign each intermediate matching result to a GPU warp and let that warp perform DFS.

We also adopt a backtracking flattening strategy. Given a query graph Q and a matching order $\varphi = u_0u_1u_2...u_lu_{l+1}...u_n$. If u_l has more than one neighbor in Q and each query vertex positioned after u_l in φ has only a single neighbor, we call the latter vertices *tail leaf vertices*. For any tail leaf vertex u_p, its neighbor u_q cannot also be a tail leaf vertex, because if so, u_p and u_q are not connected with the other part of the query graph Q, which contradicts our basic assumption that Q is connected. Hence, there are no edges between any two tail leaf vertices, and each tail leaf vertex must be connected to a vertex that belongs to the non-leaf part of Q. We propose to use *backtracking flattening* to enumerate the matching results for the tail leaf vertices. We illustrate this technique in Example 2.

Example 2. Suppose u_p and u_q are the only two tail leaf vertices, $e(u_1, u_p), e(u_2, u_q) \in E(Q)$ and $u_p \prec_\varphi u_q$ (i.e. $p = l + 1, q = n$). Given an intermediate result $(v_0, v_1, v_2, ..., v_l)$, suppose v_1 and v_2 have d_1 and d_2 neighbors that can map u_p and u_q, respectively. Then, this intermediate result can be extended to at most $d_1 * d_2$ final results. Given integers i, j, k, if $i \in [0, d_1 * d_2)$, $j = \lfloor i/d_1 \rfloor$ and $k = i \bmod d_2$, then in the i-th of the $d_1 \times d_2$ potential final results, v_p is mapped to j-th neighbor of v_1 and u_q is mapped to the k-th neighbor of v_2 (all counting from 0). Hence, we can use $d_1 * d_2$ GPU threads to extend the intermediate result $(v_0, v_1, v_2, ..., v_l)$. Inside each thread, we locate the corresponding potential final result using thread ID and perform the isomorphism check. If that potential result passes the isomorphism check, we count it as a match.

6 Experiments

6.1 Experimental Setup

We compare GCSM-BU with the three latest CPU-based single-update CSM algorithms, SymBi [22], RapidFlow [32] and NewSP [19], using their publicly available GitHub implementations[4,5,6]. We set the query-graph-average-degree threshold for GCSM-BU to adopt the two-level indexing mechanism to 3. We have implemented the GPU-based batch-update CSM algorithm GAMMA [24] because the original code is unavailable, denoted as GAMMA*. In our implementation, QO-GAMMA* and GAMMA* use the same graph data structure as GCSM-BU for a fair comparison. Experiments show that our GAMMA* implementation achieves better overall performance than that originally reported in the paper [24] on the same GPU hardware model (5–30 times faster). All methods under comparison are compiled with g++ 11.4.0 and cuda-12.8. We conduct

[4] https://github.com/RapidsAtHKUST/ContinuousSubgraphMatching
[5] https://github.com/shixuansun/RapidFlow
[6] https://github.com/hnuGraph/NewSP

Table 2: Experimental Statistics.

(a) Dataset statistics[a].

Datasets	$\|V\|$	$\|E\|$	$\|\Sigma_V\|$	$\|\Sigma_E\|$	d_{avg}	d_{max}	raw_size
LSBench (lb)	5.2M	20.3M	1	44	8.2	2.3M	443.74MB
Netflow (nf)	3.1M	2.9M	1	7	2.0	0.2M	83.64MB
LiveJournal (lj)	4.9M	42.9M	30	1	18.1	4.3M	819.57MB
Amazon (az)	0.4M	2.4M	6	1	12.2	0.2M	43.92MB

[a] d_{avg}: average vertex degree; d_{max}: maximum vertex degree; raw_size: total size of dataset files.

(b) Number of solved queries.

Queries	Tree				Sparse				Dense			
Datasets	lb	nf	lj	az	lb	nf	lj	az	lb	nf	lj	az
SymBi	88	55	96	**100**	96	88	**100**	**100**	**100**	**100**	**100**	**100**
NewSP	89	86	**100**	**100**	81	**100**	**100**	**100**	**100**	**100**	**100**	**100**
RapidFlow	89	86	**100**	**100**	97	**100**	**100**	**100**	**100**	**100**	**100**	**100**
GAMMA*	48	72	**100**	**100**	48	78	**100**	**100**	27	58	**100**	**100**
QO-GAMMA*	89	87	**100**	**100**	97	**100**	**100**	**100**	**100**	**100**	**100**	**100**
GCSM-BU	**92**	**90**	**100**	**100**	**98**	**100**	**100**	**100**	**100**	**100**	**100**	**100**

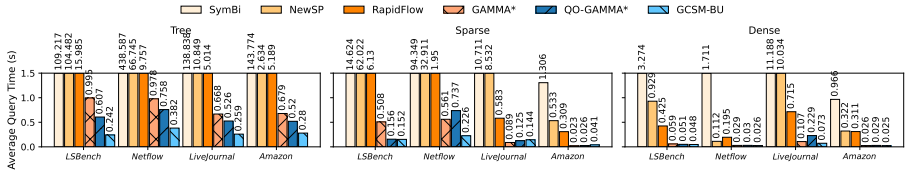


Fig. 5: Average query time of solved queries (in seconds).

experiments on a Linux server with 256 GiB of RAM, two AMD EPYC 7302 CPUs, and one NVIDIA RTX 3090 GPU with 24 GiB of device memory.

Datasets We adopt four datasets frequently used in previous work, including LSBench [23], Netflow [5], LiveJournal, and Amazon. Table 2a lists the statistics of each dataset. Since all competing algorithms perform symmetric operations for edge insertions and deletions, we follow previous studies [19, 24, 32] and only report the results on the insertion stream for brevity. We split the edge set in a dataset into initial data graph edges and inserted edges. The insertion rate is set to 10%, following previous studies [32]. We group all update edges into a single batch by default.

Query Graphs and Metrics We use query graphs from previous studies [32], which were constructed by random walks in the data graph. For all datasets, we use query graphs with 6 vertices by default. We generate a set of 100 queries for each query type and size. We measure the elapsed time of the entire online processing as query time. We set the time limit for processing a query to 0.5 hours to complete our experiments in a reasonable amount of time. We call a query completed within the time limit a *solved query*; otherwise, we call it an *unsolved query*. We report the average query time over the set of queries that are solved by all compared methods.

6.2 Overall Evaluation

Table 2b lists the number of solved queries for all query graph shapes on the four datasets. Unsolved queries mainly occur for tree and sparse query graphs, especially with skewed label distribution, such as lb and nf. Our method, GCSM-BU, can solve more queries than all CPU- and GPU-based baselines in all cases. We measure the average query time on the four datasets. RapidFlow and NewSP are the top performers among existing CPU-based algorithms, and GCSM-BU consistently outperforms them. On tree queries, our method runs 111×, 25×, 20× and 9× faster than the best CPU-based method on lb, nf, lj, and az, respectively, and nearly 3–4× faster than GAMMA*. On sparse queries, this performance trend remains in comparison with CPU-based methods, and GCSM-BU is still faster than or on par with GPU-based GAMMA* and QO-GAMMA*. Dense queries are generally easier to solve because dense edges provide stronger pruning power. GCSM-BU achieves a speedup of up to 9× over the best CPU-based algorithm and is on par with GAMMA* and QO-GAMMA* on the dense queries.

6.3 Efficiency of QO-CSM Framework

We compare the query time of GAMMA* and QO-GAMMA* to evaluate the efficiency of our QO-CSM framework. Figure 6 shows that, in most cases, QO-GAMMA* runs faster than GAMMA*. The speedup ratio ranges from 1.2 to 2.1, which demonstrates the benefit of QO-CSM, which does not require explicit duplicate removal. For sparse and dense queries on lj and az, the query workload is small, so the update cost of QO-CSM outweighs the benefit of avoiding explicit de-duplication. We have seen in Figure 5 that even in the cases where QO-GAMMA* runs slower than GAMMA*, by adopting optimizations such as the dynamic semi-BFS search strategy, our GCSM-BU still performs better than or on par with GAMMA*.

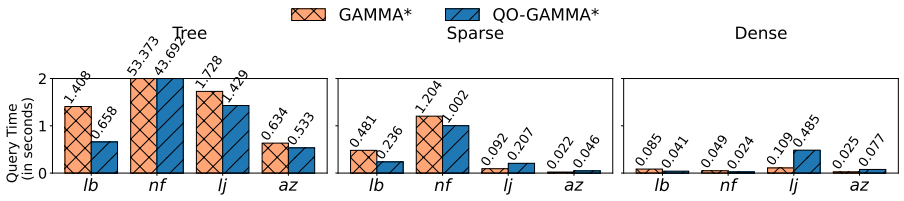


Fig. 6: Query time comparison between UO- & QO-CSM on solved queries. GAMMA* and QO-GAMMA* are different only in the framework they adopt, namely UO-CSM and QO-CSM, respectively.

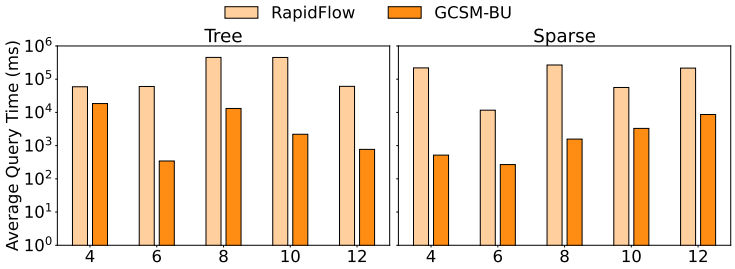


Fig. 7: Varying $|V(Q)|$.

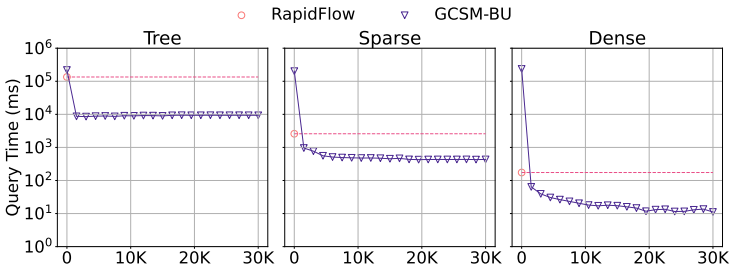


Fig. 8: Query time with various batch sizes.

6.4 Evaluation of Optimizations in Indexing and Enumeration

Table 3: Average speedups of our optimizations.

Queries	**Tree**				**Sparse**				**Dense**			
Datasets	*lb*	*nf*	*lj*	*az*	*lb*	*nf*	*lj*	*az*	*lb*	*nf*	*lj*	*az*
-Flatten	1.3	4.7	1.8	4.5	1.0	1.9	1.3	2.6	1.1	0.8	1.0	1.2
-semiBFS	1.2	191.1	1.0	1.0	1.0	2.2	1.3	1.5	1.0	0.8	1.2	1.0

Table 3 shows the speedup achieved by the full-fledged GCSM-BU over two variants of our algorithm with backtracking flattening disabled (-Flatten) and with the dynamic semi-BFS strategy disabled (-semiBFS). The backtracking flattening mechanism improves the matching efficiency in all cases because it is common for a query graph to contain leaf vertices and these leaf vertices are often placed at the end of the matching order. The dynamic semi-BFS strategy shows its benefits mainly on the dataset nf, where the number of updated edges in each sub-batch is small.

6.5 Varying Query Graph Size & Batch Size

Query Size We first perform tree and sparse queries on the dataset ls with the query graph size varied from 4 to 12 in increments of 2. As shown in Figure 7,

GCSM-BU achieves speedups of around 99× and 136× over the best baseline method RapidFlow, on tree and sparse queries, respectively. The results indicate that our proposed techniques consistently show benefits under different query graph sizes.

Batch Size Finally, we run GCSM-BU with the batch size of the update stream varied from 1 to 30k in increments of 1,500. Then, we plot the average query time of GCSM-BU under various batch sizes and that of RapidFlow for single updates in Figure 8. Given a batch size of 1, GCSM-BU underperforms RapidFlow since the matching workload caused by a single edge update is small and cannot fully utilize the parallel computing power of a GPU. The query time drops as the batch size increases. The performance gain becomes negligible when the batch size reaches 1500 in tree queries, 4500 in sparse queries, or 19500 in dense queries because, under these settings, the matching workload of a batch is sufficient to fully utilize the GPU.

7 Conclusion & Future Work

In this paper, we studied the problem of continuous subgraph matching for batch updates and proposed a query-oriented CSM framework, QO-CSM, which avoids duplicate query results automatically. Based on QO-CSM, we implemented a GPU-based CSM algorithm, GCSM-BU, which integrates various techniques and practical optimizations. Experiments show that GCSM-BU significantly outperforms the latest CPU-based CSM algorithms on batch updates and achieves better overall performance than the GAMMA* variants.

Although QO-CSM provides a hardware-independent framework for avoiding duplicate results in batch-update CSM, its performance benefit depends on the workload characteristics. When the matching workload is small, the additional update cost may outweigh the benefit of avoiding explicit duplicate removal. Future work could investigate an adaptive execution framework that chooses between QO-CSM and UO-CSM-style processing based on the estimated update cost, search cost, and duplicate-removal cost. GCSM-BU is designed primarily for GPU-friendly batch workloads. Its performance relies on exposing sufficient parallelism across update edges and search-tree nodes. For very small batches or highly selective queries, the GPU may not be fully utilized, and its advantage over CPU-based or simpler GPU-based methods can become less significant. To address this limitation, future work can explore adaptive CPU-GPU execution, where small or highly selective workloads can be routed to the CPU, while large and highly parallel workloads are executed on the GPU.

Acknowledgments. This work was supported by Grant 16209821 from the Hong Kong Research Grants Council.

References

1. Ammar, K., McSherry, F., Salihoglu, S., Joglekar, M.: Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows. Proc. VLDB Endow. **11**(6), 691–704 (2018). https://doi.org/10.14778/3184470.3184473
2. Armstrong, T.G., Ponnekanti, V., Borthakur, D., Callaghan, M.: Linkbench: a database benchmark based on the facebook social graph. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013. pp. 1185–1196. ACM (2013). https://doi.org/10.1145/2463676.2465296
3. Blakeley, J.A., Larson, P., Tompa, F.W.: Efficiently updating materialized views. In: Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986. pp. 61–71. ACM Press (1986). https://doi.org/10.1145/16894.16861
4. Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D.E., Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. BMC Bioinform. **14**(S-7), S13 (2013). https://doi.org/10.1186/1471-2105-14-S7-S13
5. CAIDA: The caida ucsd anonymized internet traces 2013. `https://catalog.caida.org/details/dataset/passive\_2013\_pcap` pp. (Accessed Date: Mar 07, 2022) (2013)
6. Choudhury, S., Holder, L.B., Jr., G.C., Agarwal, K., Feo, J.: A selectivity based approach to continuous pattern detection in streaming graphs. In: Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015. pp. 157–168. OpenProceedings.org (2015). https://doi.org/10.5441/002/edbt.2015.15
7. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**(10), 1367–1372 (2004). https://doi.org/10.1109/TPAMI.2004.75
8. Dhulipala, L., Durfee, D., Kulkarni, J., Peng, R., Sawlani, S., Sun, X.: Parallel batch-dynamic graphs: Algorithms and lower bounds. In: Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020. pp. 1300–1319. SIAM (2020). https://doi.org/10.1137/1.9781611975994.79
9. Fan, W.: Graph pattern matching revised for social network analysis. In: 15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012. pp. 8–21. ACM (2012). https://doi.org/10.1145/2274576.2274578
10. Fan, W., Li, J., Luo, J., Tan, Z., Wang, X., Wu, Y.: Incremental graph pattern matching. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011. pp. 925–936. ACM (2011). https://doi.org/10.1145/1989323.1989420
11. Farhan, M., Wang, Q., Koehler, H.: Batchhl: Answering distance queries on batch-dynamic networks at scale. In: SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022. pp. 2020–2033. ACM (2022). https://doi.org/10.1145/3514221.3517883
12. Gao, J., Zhou, C., Yu, J.X.: Toward continuous pattern detection over evolving large graph with snapshot isolation. VLDB J. **25**(2), 269–290 (2016). https://doi.org/10.1007/S00778-015-0416-Z
13. Griffin, T., Libkin, L.: Incremental maintenance of views with duplicates. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995. pp. 328–339. ACM Press (1995). https://doi.org/10.1145/223784.223849

14. Han, M., Kim, H., Gu, G., Park, K., Han, W.: Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019. pp. 1429–1446. ACM (2019). https://doi.org/10.1145/3299869.3319880
15. Idris, M., Ugarte, M., Vansummeren, S.: The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017. pp. 1259–1274. ACM (2017). https://doi.org/10.1145/3035918.3064027
16. Kankanamge, C., Sahu, S., Mhedhbi, A., Chen, J., Salihoglu, S.: Graphflow: An active graph database. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017. pp. 1695–1698. ACM (2017). https://doi.org/10.1145/3035918.3056445
17. Kim, H., Choi, Y., Park, K., Lin, X., Hong, S., Han, W.: Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In: SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021. pp. 925–937. ACM (2021). https://doi.org/10.1145/3448016.3457265
18. Kim, K., Seo, I., Han, W., Lee, J., Hong, S., Chafi, H., Shin, H., Jeong, G.: Turboflux: A fast continuous subgraph matching system for streaming graph data. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018. pp. 411–426. ACM (2018). https://doi.org/10.1145/3183713.3196917
19. Li, Z., Li, Y., Chen, X., Zou, L., Li, Y., Yang, X., Jiang, H.: Newsp: A new search process for continuous subgraph matching over dynamic graphs. In: 40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024. pp. 3324–3337. IEEE (2024). https://doi.org/10.1109/ICDE60146.2024.00257
20. Mariappan, M., Vora, K.: Graphbolt: Dependency-driven synchronous processing of streaming graphs. In: Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019. pp. 25:1–25:16. ACM (2019). https://doi.org/10.1145/3302424.3303974
21. Mhedhbi, A., Salihoglu, S.: Optimizing subgraph queries by combining binary and worst-case optimal joins. Proc. VLDB Endow. **12**(11), 1692–1704 (2019). https://doi.org/10.14778/3342263.3342643
22. Min, S., Park, S.G., Park, K., Giammarresi, D., Italiano, G.F., Han, W.: Symmetric continuous subgraph matching with bidirectional dynamic programming. Proc. VLDB Endow. **14**(8), 1298–1310 (2021). https://doi.org/10.14778/3457390.3457395
23. Phuoc, D.L., Dao-Tran, M., Pham, M., Boncz, P.A., Eiter, T., Fink, M.: Linked stream data processing engines: Facts and figures. In: The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part II. Lecture Notes in Computer Science, vol. 7650, pp. 300–312. Springer (2012). https://doi.org/10.1007/978-3-642-35173-0_20
24. Qiu, L., Chen, L., Jie, H., Ke, X., Gao, Y., Liu, Y., Zhang, Z.: Gpu-accelerated batch-dynamic subgraph matching. In: 40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024. pp. 3204–3216. IEEE (2024). https://doi.org/10.1109/ICDE60146.2024.00248
25. Qiu, X., Cen, W., Qian, Z., Peng, Y., Zhang, Y., Lin, X., Zhou, J.: Real-time constrained cycle detection in large dynamic graphs. Proc. VLDB Endow. **11**(12), 1876–1888 (2018). https://doi.org/10.14778/3229863.3229874

26. Reichhold, J., Helder, D., Asemanfar, A., Molina, M., Harris, M.: New tweets per second record, and how! `https://blog.twitter.com/engineering/en\_us/a/2013/new-tweets-per-second-record-and-how` pp. (Accessed Date: Jun 30th, 2023) (2013)
27. Rivero, C.R., Jamil, H.M.: Efficient and scalable labeled subgraph matching using sgmatch. Knowl. Inf. Syst. **51**(1), 61–87 (2017). https://doi.org/10.1007/s10115-016-0968-2
28. Sengupta, D., Sundaram, N., Zhu, X., Willke, T.L., Young, J.S., Wolf, M., Schwan, K.: Graphin: An online high performance incremental graph processing framework. In: Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9833, pp. 319–333. Springer (2016). https://doi.org/10.1007/978-3-319-43659-3_24
29. Shang, H., Zhang, Y., Lin, X., Yu, J.X.: Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. Proc. VLDB Endow. **1**(1), 364–375 (2008). https://doi.org/10.14778/1453856.1453899
30. Sun, S., Luo, Q.: In-memory subgraph matching: An in-depth study. In: Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020. pp. 1083–1098. ACM (2020). https://doi.org/10.1145/3318464.3380581
31. Sun, S., Sun, X., Che, Y., Luo, Q., He, B.: Rapidmatch: A holistic approach to subgraph query processing. Proc. VLDB Endow. **14**(2), 176–188 (2020). https://doi.org/10.14778/3425879.3425888
32. Sun, S., Sun, X., He, B., Luo, Q.: Rapidflow: An efficient approach to continuous subgraph matching. Proc. VLDB Endow. **15**(11), 2415–2427 (2022)
33. Sun, X., Luo, Q.: Efficient gpu-accelerated subgraph matching. Proc. ACM Manag. Data **1**(2), 181:1–181:26 (2023). https://doi.org/10.1145/3589326
34. Sun, X., Sun, S., Luo, Q., He, B.: An in-depth study of continuous subgraph matching. Proc. VLDB Endow. **15**(7), 1403–1416 (2022)
35. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. ACM **23**(1), 31–42 (1976). https://doi.org/10.1145/321921.321925
36. Wang, L., Owens, J.D.: Fast gunrock subgraph matching (GSM) on gpus. CoRR **abs/2003.01527** (2020)
37. Wei, Y., Jiang, P.: Stmatch: Accelerating graph pattern matching on GPU with stack-based loop optimizations. In: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022. pp. 53:1–53:13. IEEE (2022). https://doi.org/10.1109/SC41404.2022.00058
38. Xiang, L., Khan, A., Serra, E., Halappanavar, M., Sukumaran-Rajam, A.: cuts: scaling subgraph isomorphism on distributed multi-gpu systems using trie based data structure. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021. p. 69. ACM (2021). https://doi.org/10.1145/3458817.3476214
39. Yang, R., Zhang, Z., Zheng, W., Yu, J.X.: Fast continuous subgraph matching over streaming graphs via backtracking reduction. Proc. ACM Manag. Data **1**(1), 15:1–15:26 (2023). https://doi.org/10.1145/3588695
40. Zeng, L., Zou, L., Özsu, M.T., Hu, L., Zhang, F.: GSI: gpu-friendly subgraph isomorphism. In: 36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020. pp. 1249–1260. IEEE (2020). https://doi.org/10.1109/ICDE48307.2020.00112