RUNTIME AND COMPILER OPTIMIZATIONS FOR SUBGRAPH MATCHING ALGORITHMS ON GPUS

by

Yihua Wei

A thesis submitted in partial fulfillment
of the requirements for the Doctor of Philosophy
degree in Computer Science in the
Graduate College of
The University of Iowa

May 2026

Thesis Committee: Peng Jiang, Thesis Supervisor
Bijaya Adhikari
Steve Goddard
Kasturi Varadarajan
Kasturi Varadarajan

TABLE OF CONTENTS

LIST OF TABLES

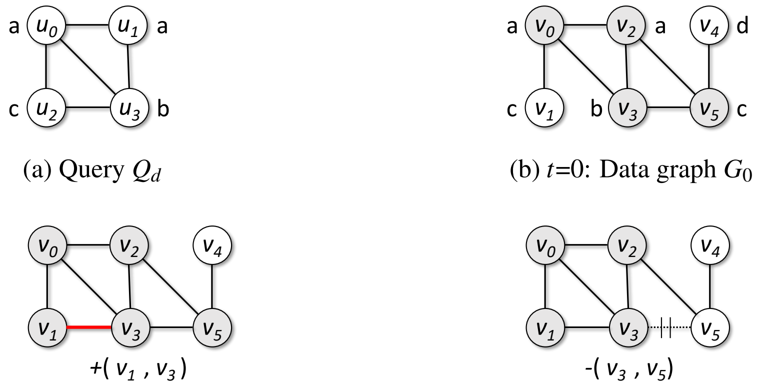
LIST OF FIGURES

1 DCSM: ENABLING INTER-BATCH PARALLELISM FOR CONTINUOUS SUBGRAPH MATCHING ON GPU

1.1 Introduction

Continuous subgraph matching (CSM) refers to the process of incrementally matching a query pattern against a dynamic data graph. CSM plays a critical role in many real-world graph analytics applications. For example, transactions in e-commerce platforms can be modeled as a dynamic graph, and CSM can be employed to detect fraudulent merchants [8]. It can also be applied to monitor money laundering in transaction networks [10], trace rumor propagation paths in social networks [14], and identify system anomalies in computer communication networks [6].

Figure 1 illustrates an example of CSM. Given an initial data graph G_0 at time $t = 0$ and a query pattern Q_d, G_0 contains one matching subgraph (v_0, v_2, v_3, v_5). At $t = 3$, an edge (v_1, v_3) is added to G_0, the updated graph G_1 produces an additional match (v_0, v_1, v_2, v_3) for the query. At $t = 5$, the edge (v_3, v_5) is deleted from the graph, which invalidates the previous match (v_0, v_2, v_3, v_5).



(a) Query Q_d (b) t=0: Data graph G_0

(c) t=3: Data graph G_1 (d) t=5: Data graph G_2

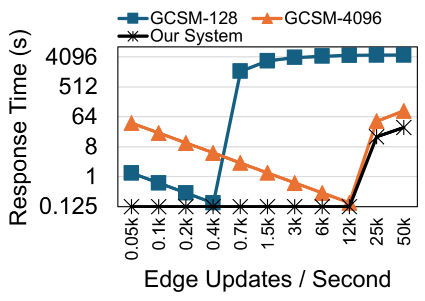
Figure 1. An example of continuous subgraph matching. G_{k+1} is the data graph after applyting update on G_k. Q_d is the query of ”diamond” structure. *abcd* next to the vertices are the vertex labels.

Many efforts have been made to improve the performance of CSM on CPUs, primarily by reducing the search space during the matching process [4,5,7,12]. However, due to the exponential time complexity of subgraph matching [13], CSM remains a computationally expensive procedure.

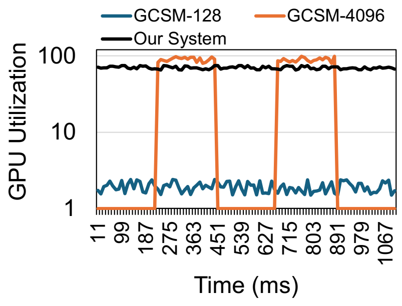
To address this limitation, recent systems [9, 16] have begun exploiting the massive parallelism of GPUs to accelerate CSM.

The existing GPU-based CSM systems process updates using fixed-size batches. Specifically, they wait until a predefined number of edge updates accumulate before grouping them into a batch and launching it on the GPU. To fully utilize GPU cores, the batch size is typically set to a large value (e.g., 4096). While a large batch size improves the GPU occupancy during the matching procedure, it increases the response time of individual updates and may even reduce overall GPU utilization, since edges must wait for the entire batch to be formed before execution.

To validate this point, we tested the state-of-the-art GPU-based CSM system (GCSM [16]) under different batch sizes. The performance results are presented in Figure 2a. We define the response time of a graph update as the interval between its arrival and the start of its processing. We observe that GCSM exhibits poor response time under different graph update rates. With a batch size of 4096, GCSM-4096 suffers from long response times at low update rates, as it must wait to accumulate a sufficient number of updates to form a large batch. Conversely, with a batch size of 128, GCSM-128 experiences longer response times at high update rates due to its limited parallelism and thus low GPU utilization, as shown in Figure 2b.



(a) Response Time



(b) GPU Utilization

Figure 2. Average response time and GPU utilization of GCSM. GCSM-x denotes GCSM processing with batch size x. **Query:** Q4 (Figure **??**). **Graph:** Unlabeled Netflow [1].

To address the limitation of existing GPU-based CSM systems, we propose DCSM in this work. We find that the problem with existing systems stems from the fact that they cannot process different batches in parallel. A batch must wait for the previous batch to finish processing before it can be launched. DCSM addresses the response time and utilization issue by enabling parallel execution across batches. Intuitively, such inter-batch parallelism allows our system to process graph updates in small batches while keeping a high GPU occupancy. As shown in Figure 2, our system maintains short response times across a wide range of update rates and achieves higher average GPU utilization than GCSM.

The key technique that enables inter-batch parallelism in our system is resolving data races on the data graph while processing multiple batches concurrently. We propose the first **multi-version graph data structure** for CSM. It records a distinct graph version with minimum overhead for each batch update, allowing safe and efficient inter-batch parallelism. To manage memory consumption, we develop a **garbage collection mechanism** that releases graph versions no longer in use. We further introduce several system-level optimizations, including **pipelined and parallel updates** to the multi-version graph using GPU warps, speeding up the creation of new graph versions, and a **dynamic scheduling strategy** that assigns pending batches to idle warps to achieve better load balance among warps.

We evaluate our system against state-of-the-art CPU and GPU systems. The experiments demonstrate that DCSM ensures optimal response time and throughput across different update rates. Compared to GCSM [16], DCSM achieves up to 87.9x speedup under high update rates and up to 312x speedup under low update rates. Furthermore, DCSM achieves 10.5x higher throughput compared to the state-of-the-art CPU-based CSM system RapidFlow [12].

1.2 Background

1.2.1 A Formal Definition of Csm

Definition 1 (Graph)**.** A graph $G = (V,E,L)$, consists of a set of vertices V, a set of edges E, and a labeling function L that assigns labels to vertices.

Definition 2 (Subgraph)**.** A graph $G' = (V',E',L')$ is a subgraph of graph $G = (V,E,L)$ if V' is a subset of V, E' is a subset of E, and $L'(v) = L(v)$, for all v in V'.

Definition 3 (Isomorphism)**.** Two graphs $G_a = (V_a, E_a, L_a)$ and $G_b = (V_b, E_b, L_b)$ are isomorphic if there is a bijective function $f : V_a \Rightarrow V_b$ such that $(v_i, v_j) \in E_a$ if and only if $(f(v_i), f(v_j)) \in E_b$ and $L_a(v_i) = L_b(f(v_i)), L_a(v_j) = L_b(f(v_j))$.

Definition 4 (Static Subgraph Matching)**.** The static subgraph matching problem is finding all the subgraphs in a data graph G that are isomorphic to a query pattern Q.

Definition 5 (Dynamic Graph)**.** A dynamic graph $G = (G_0, \Delta G)$ is a sequence of edge updates $\Delta G = [\Delta e_0, \ldots, \Delta e_k, \ldots]$ applied to an initial graph G_0. Each update $\Delta e_k = (v_i, v_j, \oplus)$ inserts or deletes edge (v_i, v_j) from the graph G_k. The symbol $\oplus$ can be $+$ or $-$, meaning an edge insertion or deletion. Each update Δe_k arrives at a random time $T(\Delta e_k) > 0$, and $T(\Delta e_{k+1}) > T(\Delta e_k)$. The edge updates generate a sequence of graph snapshots $[G_0, G_1, G_2, G_3, \ldots]$ where $G_{k+1} = G_k \oplus \Delta e_k$.

Definition 6 (Continuous Subgraph Matching)**.** Continuous subgraph matching (CSM) aims to find the incremental subgraphs Δm_k that are isomorphic to a query pattern Q in a dynamic graph $(G_0, \Delta G)$ for each update $\Delta e_k \in \Delta G$.

1.2.2 Existing Gpu-based Csm Systems

A GPU contains tens of streaming multiprocessors (SMs), each with 32-192 CUDA cores depending on the architecture. GPU kernels are organized as grids of thread blocks, where each block contains multiple 32-thread warps. Thread blocks are assigned to SMs for execution, with each SM capable of hosting multiple blocks. Threads within a warp execute in SIMT fashion, and all threads can access the GPU's global memory.

Previous GPU-based CSM systems [9, 16] are designed to operate with a fixed batch size: they wait until a predefined number of edge updates are accumulated to form a batch, process that batch on the GPU, and then move on to the next one. Each batch is processed in three steps:

- **Prepare:** Construct a batch graph G_b from the edges in the batch on the CPU and transfer G_b to the GPU's global memory. The system now maintains two graphs: the data graph G_k (to be updated) and the batch graph G_b.

- **Match:** Perform incremental matching for each edge in the batch. Each GPU warp processes one edge and explores its k-hop neighborhood in both G_b and G_k. Larger batch sizes provide higher parallelism.

- **Update:** Merge the edges from G_b into G_k, and then sort each updated neighbor list in G_k by vertex ID. Prior GPU systems differ in their data graph placement: GCSM [16] stores G_k in the CPU's main memory, whereas GAMMA [9] stores G_k in the GPU's global memory.

1.3 Overview of Dcsm

Figure 3 provides an overview of our DCSM system, which consists of three modules: a graph updater (GU), an executor (EX), and a garbage collector (GC).

Each edge update $\Delta e_k = (v_i, v_j, \oplus)$ flows through these three modules, which process multiple updates in a pipelined manner. The graph updater (GU) accepts batches of edge updates Δe_k from the CPU and applies them to the data graph on the GPU using 1-4 warps, where all warps work together to process each batch in parallel. The executor (EX) performs incremental matching for each Δe_k and produces the matching results; it runs on thousands of warps and dynamically schedules new updates to idle warps. The garbage collector (GC) runs on the GPU and reclaims the memory allocated by GU back to the memory pool. To ensure correctness, edge updates must preserve their order of arrival—i.e., Δe_{k+1} cannot pass through GU, EX, or GC before Δe_k.

Each functional module serves as both a consumer and a producer, and its workload influences the flow speed. For instance, if GU is a lightweight module and EX is a heavy module, it will lead to multiple Δe_k accumulating between GU and EX. Conversely, if GU is heavier than EX, it will cause EX to become idle most time. DCSM adopts a warp-specialized design, in which each module is executed by one or more warps on a GPU. Each module can also adjust itself based on the accumulation of Δe_k on its left and right sides. For example, if too many Δe_k accumulate between GU and EX, then GU will pause and wait for the accumulated Δe_k to be consumed by EX.

Figure 4 illustrates how DCSM processes a sequence of contiguous graph updates. In this example, the GPU has two parallel computing units: warp 0 and warp 1. In GCSM [16] with a

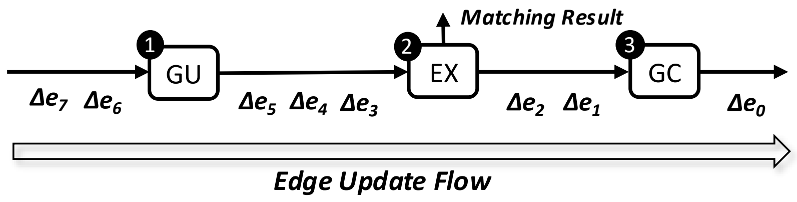


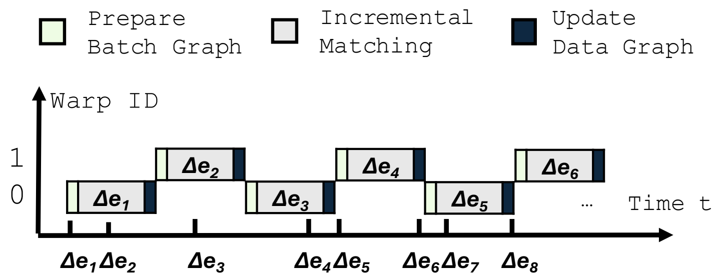
Figure 3. Overview of DCSM. DCSM consists of three functional modules: Graph Updater (GU), Executor (EX), and Garbage Collector (GC). The edge updates Δe_k flows through these modules sequentially.

batch size of 2, Δe_3 must wait until Δe_4 arrives to form a batch, which delays the processing of Δe_3. In contrast, GCSM with a smaller batch size of 1 processes all Δe_k serially, resulting in lower throughput and longer average response time. DCSM, by enabling inter-batch parallelism, can flexibly schedule any newly formed batch to an idle warp, thereby improving both response time and throughput.

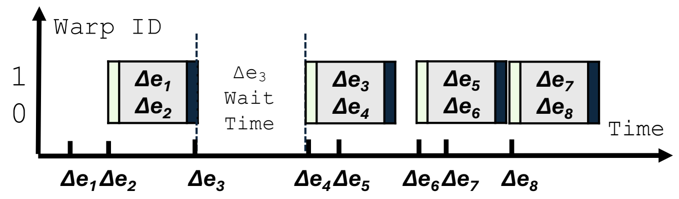
The rest of the paper is organized as follows. Sections 1.4, 1.5, and 1.6 describe the three functional modules—GU, EX, and GC, respectively. Section **??** presents the evaluation results.

1.4 Graph Updater And Multi-version Graph

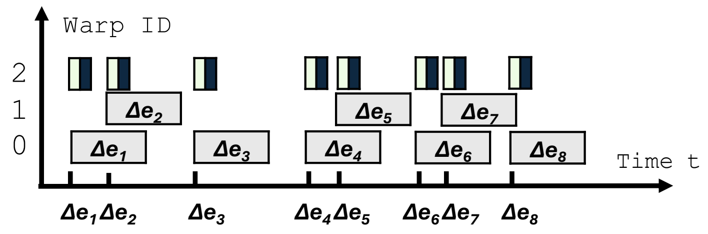
All existing CSM systems [9,16] on GPUs are derived from static subgraph matching frameworks [3, 15], and they fail to address data race problems that arise when processing multiple batches concurrently. As a result, these systems are forced to handle n batches (B_1, ..., B_n) in a strictly sequential manner: $P(B_1) \to M(B_1) \to U(B_1) \to \ldots \to P(B_n) \to M(B_n) \to U(B_n)$, where P, M, and U represent the Prepare, Match, and Update phases in Section 1.2.2. This serialization stems from the sort operation in the Update phase: executing $U(B_k)$ concurrently with $M(B_k)$ would introduce data races, and $M(B_{k+1})$ must be deferred until $U(B_k)$ finishes, because $M(B_{k+1})$ depends on the data graph state produced by $U(B_k)$. To overcome this limitation, we introduce a multi-version graph data structure for CSM.



(a) GCSM with batch size 1.



(b) GCSM with batch size 2.



(c) DCSM with inter-batch parallelism.

Figure 4. Comparison between GCSM and DCSM. Eight updates Δe_1 - Δe_8 arrive sequentially. The bar in the figure denotes warp activity along the time axis. Each bar is associated with an update Δe_k, labeled inside the bar. In practice, DCSM uses 5,000-10,000 warps for incremental matching, while only 1-4 warps are assigned for graph updates.

1.4.1 Multi-version Graph (Mvg) Data Structure

Our MVG data structure adopts the classic adjacency list format, but each vertex maintains multiple neighbor arrays of different versions. Figure 5 shows an example; its caption provides an explanation. This design fully considers the efficiency of the matching algorithm. To satisfy the matching efficiency, each neighbor array is sorted by vertex ID. To enable dynamic extension and shrinkage, slab objects of each vertex are linked together as a list. To avoid the data race problem mentioned above, we maintain multiple neighbor array versions for each vertex. Therefore, inserting a batch of edges in the data graph will not cause data races with the ongoing matching procedures since the newly inserted edges reside in newly created versions of the neighbor arrays.

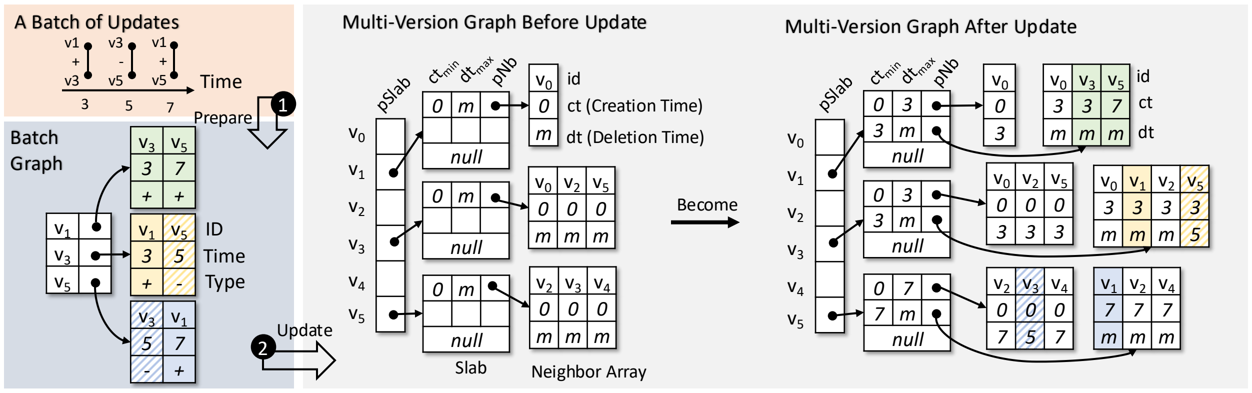


Figure 5. The multi-version graph (MVG) data structure and the procedure for updating it with a batch. The MVG before update represents G_0 in Figure 1. The vertex ID indexes the *pSlab*, pointing to a *Slab*. Each row of the *Slab* corresponds to a neighbor array version, storing its creation time (ct_{min}), deletion time (dt_{max}), and address (*pNb*). Each column of the neighbor array stores the ID, creation time (ct), and deletion time (dt) of an edge. We use v_x**:** $\mathbf{(}ct\mathbf{,}\ dt\mathbf{]}$ to denote a neighbor array version of v_x. For example, v_1 has two neighbor array versions: v_1: (0, 3] and v_1: (3, m] where m means infinity.

The ct_{min}, dt_{max}, ct, and dt in MVG guide the GPU kernel in determining which version to visit during incremental matching.

1.4.2 Update Procedure

Figure 5 visualizes the graph update process. For three edge updates arriving at times 3, 5, and 7, we group them into a batch. The graph update consists of two steps. (1) Form a batch graph of adjacency list format using the edges in the batch, its neighbor arrays are sorted by edge arrival time, and record the edge update type (insertion or deletion). (2) Update the MVG. Each neighbor array in the batch graph is partitioned into two parts at the first insertion edge. The left part includes the early-arrived edges to be deleted in-place in MVG, while the right part includes the later-arrived edges to be inserted or deleted, but needs to create a new neighbor array version for the MVG. Take the example in Figure 5. To update v_5's neighbor array in the MVG, we first update the deletion time of v_3 to 5, then create a new neighbor array version and insert v_1 into it.

The Graph Updater (GU) can be configured with two parameters: a waiting time t and a batch size b. Within the time window t, GU waits for b edge updates Δe_k to arrive to form a batch.

If the waiting time t expires before b edge updates arrive, all arrived Δe_k form a batch smaller than b. A smaller b can have better response time, but if b is too small it will slow down the GU module. Typically, we set b to 64 and t to a small value (e.g., 0.125ms) that does not affect user experience.

1.4.3 Parallel Graph Updater

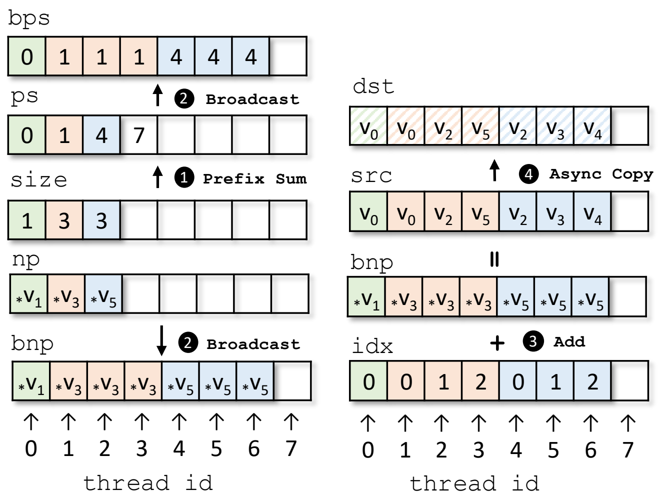


Figure 6. The process of copying three different-sized arrays in parallel using 8 threads of a warp. $*v_i$ represents the address of v_i's latest neighbor array version in the MVG.

To prevent the graph updater (GU) from becoming a performance bottleneck for lightweight query workloads, we need to accelerate it on GPU. However, because GPU warps follow the SIMT execution model and the average neighbor array is small, it is difficult to keep all 32 threads in a warp busy. More specifically, to merge a batch graph G_b into the MVG G_k, for each vertex v in G_b, we first copy both v's latest neighbor array from G_k and v's neighbors from G_b into a newly allocated array, which becomes v's updated neighbor version, and then sort this array while excluding deleted vertices. A naive approach assigns one warp for the task, with each thread copying one array element. However, since the average neighbor size in most real-world graphs is only 2 to 3, only a few threads are active during the execution. To improve thread utilization, we would like the 32 threads in a warp to copy multiple neighbor arrays in parallel.

To copy multiple arrays in parallel with one warp, we use the following strategy. Figure 6 shows how a warp copies three arrays in parallel; specifically, it copies v_1: (0, 3], v_3: (0, 3] and

v_5: (0, 7] for the update procedure shown in Figure 5. In this example, `np`, `size`, etc., are register variables, and cross-thread operations on them use CUDA warp-level primitives. Initially, each thread stores the address (`np`) and size (`size`) of a distinct array. We first compute the prefix sum (`ps`) over the `size` values. Then, we broadcast `ps` and `np` into `bps` and `bnp` based on `size` value. For example, since thread 2 has `size = 3`, $*v_3$ is broadcast to three copies in `bnp`. Next, we compute the `idx` as `idx` = *tid* - `bps`, where *tid* is thread id. Using `idx` and `bnp`, each thread maps to a unique element within its assigned array. For instance, thread 3 accesses the neighbor at index 2 in v_3's array, which is vertex v_5. If the sum of neighbor sizes exceeds the number of threads within a warp, the warp iterates to complete the copy. In our actual system design, we treat n warps as a large "super-warp" with $n \times 32$ threads, copying multiple neighbor arrays in parallel.

We also process two consecutive batches in a pipelined manner. Specifically, data copying is implemented using the `memcpy_async` instruction, allowing it to overlap with the sorting phase of the previous batch.

1.5 Executor

The Executor (EX) consumes updates Δe_k produced by GU and performs incremental matching for each Δe_k. Since subgraph matching is NP-hard [13], the EX is the performance bottleneck of the system and needs to be parallelized across multiple GPU warps. This section explains how we adapt an existing subgraph matching kernel [3,15] for EX to ensure both correctness and efficiency in CSM.

1.5.1 Correctness Guarantee

For any update $\Delta e_k = (v_i, v_j, \oplus)$ that arrives at time $T(\Delta e_k)$, its incremental matching will access the neighbor arrays of v_i and v_j's k-hop neighbor vertices. However, our multi-version graph (MVG) data structure introduces a problem: which neighbor array version should be used when visited? Property 1 to Property 3 provide guidance on how the Executor's GPU kernels can correctly access the MVG.

Property 1. For an edge e_k that is added to the data graph at time t_1 and deleted at time t_2, the incremental matching for updates arriving within $(t_1,t_2]$ should see e_k as existing in the data graph. Conversely, the incremental matching for updates outside $(t_1,t_2]$ should not.

For a specific edge $e_k = (v_i, v_j)$, there is exactly one edge insertion $\Delta e_k^+ = (v_i, v_j, +)$ and at most one edge deletion $\Delta e_k^- = (v_i, v_j, -)$. We have $T(\Delta e_k^+) = 0$ if e_k exists in the initial data graph G_0, and $T(\Delta e_k^-) = \infty$ (or m) if e_k is never deleted. For both Δe_k^+ and Δe_k^-, the **Update** step starts after **Matching** (see Section 1.2.2), and we regard each edge update as a single batch. Therefore, e_k is invisible to $M(\Delta e_k^+)$, and e_k is visible to $M(\Delta e_k^-)$. In addition, it is obvious that e_k is visible to $M(\Delta e_x)$ for all Δe_x such that $T(\Delta e_k^+) < T(\Delta e_x) < T(\Delta e_k^-)$. Even though our multi-version graph allows **Update** and **Matching** to execute in parallel, the visibility order mentioned above remains unchanged. In summary, the incremental matching for updates arriving within $(T(\Delta e_k^+), T(\Delta e_k^-)]$ should see e_k as existing in the data graph.

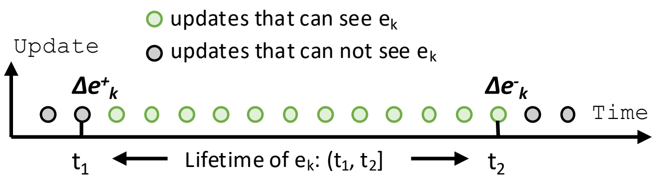


Figure 7. Visualization of edge e_k's lifetime and visibility across other updates.

We therefore define the lifetime of an edge and a neighbor array version. The **lifetime of an edge** e_k is $(t_1, t_2]$, where e_k is added to the data graph at time t_1 and deleted at time t_2. **The lifetime of a neighbor array version** in a multi-version graph is $(ct_{min}, dt_{max}]$, where ct_{min} and dt_{max} are the minimum creation time and maximum deletion time of entries in this neighbor array, respectively. Figure 7 visualizes the lifetime of an edge e_k and which edge updates can see e_k existing in the data graph.

Property 2. To ensure correctness, the incremental matching for an update Δe_k should only visit edges whose lifetime $(t_1,t_2]$ contains $T(\Delta e_k)$ ($T(\Delta e_k) \in (t_1,t_2]$).

According to Property 1, an edge e_x in the data graph should only be visible to the incremental matching procedure for updates arriving within e_x's lifetime $(t_1,t_2]$. In other words, for an update Δe_k arriving at time $T(\Delta e_k)$, its matching procedure should only visit edges whose lifetime $(t_1, t_2]$ contains $T(\Delta e_k)$.

Property 3. To ensure correctness, the incremental matching for an update Δe_k should only visit the neighbor array versions whose lifetime $(ct_{min},\ dt_{max}]$ contains $T(\Delta e_k)$ and filter out the edges whose lifetime does not contain $T(\Delta e_k)$.

In the MVG shown in Figure 5, each $(ct, dt]$ pair represents a segment of an edge's lifetime. For example, the lifetime of edge (v_1, v_0) consists of two segments, $(0,3]$ and $(3,m]$, while edge (v_1, v_5) has a lifetime of $(7,m]$. The lifetime of a neighbor array covers the time ranges of edges in it. Therefore, according to Property 2, the valid neighbor edges must be in neighbor array versions whose lifetime $(ct_{min}, dt_{max}]$ contains $T(\Delta e_k)$.

Example 1. For an update arriving at time 6, if its matching procedure visits v_5's neighbor array, it should visit the neighbor array version with lifetime $(0,7]$ since $6 \in (0,7]$. However, the neighbor vertex v_3 in this version should be ignored because its deletion time is 5, which means v_3 does not exist at time 6.

1.5.2 Continuous Subgraph Matching-specific Optimizations

Efficient Neighbor Array Access: The slab design in the multi-version graph (MVG), together with CUDA warp-level primitives, enables our system to efficiently access neighbor arrays. Each slab in the MVG contains 32 entries, and multiple slabs are connected as a linked list. The matching for an update Δe_k is performed by a single warp. When a warp accesses a vertex's neighbor array, it traverses the slab linked list. For each slab, the warp's 32 thread lanes check in parallel whether $T(\Delta e_k)$ falls within the lifetime of any of the 32 slab entries. We leverage the warp-level primitives `__ballot_sync` and `__ffs` to perform parallel checks. The two functional modules, GU and GC, collaboratively make the number of valid entries in a slab list dynamically grow and shrink. For almost all graphs, the average number of valid entries per slab list remains between 1 and 5, which is much smaller than 32. Therefore, compared with previous systems, our design does not increase the time complexity of neighbor array access.

Dynamic Task Scheduler: The updates Δe_k before EX increase dynamically as the GU produces. Therefore, we need to dynamically schedule newly added Δe_k to idle warps. To solve this problem, we propose a dynamic scheduling strategy. For a Δe_k to be consumed by EX, n idle warps will simultaneously compete for this Δe_k. The warp wins will asynchronously execute this Δe_k, then the remaining $n-1$ idle warps will compete for the next Δe_k. Once a warp finishes the matching of a Δe_k, this warp returns to the competition status to compete for its next Δe_k. Therefore, multiple Δe_k are dynamically scheduled to the idle warps, which can ensure the load balance among warps.

Output of Executor: EX not only forwards each Δe_k it processes to GC, but also outputs the matching result of each Δe_k. The matching results of an edge insertion mean that we obtain additional valid matches in the data graph, while the matching results of an edge deletion mean that previously valid matches are no longer valid in the data graph. In some scenarios, it is necessary to obtain the matching result for an entire batch. We can compute the union of the matching results of all individual updates in the batch to obtain the overall batch matching result.

1.6 Garbage Collector

The multiple neighbor array versions allocated by the graph updater (GU) consume a large amount of GPU memory. These neighbor arrays should be released once they are no longer in use; otherwise, GPU memory usage will keep growing. This section describes our strategy for releasing these neighbor array versions.

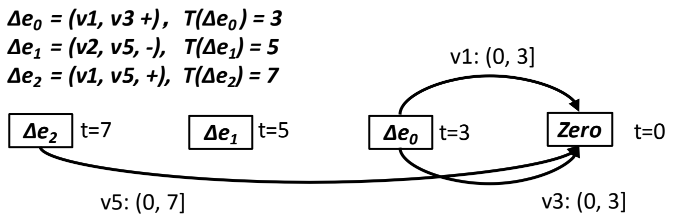
1.6.1 Release Conditions

A neighbor array version with lifetime $(ct_{min}, dt_{max}]$ can be released once all updates Δe_k whose $T(\Delta e_k) \in (ct_{min}, dt_{max}]$ have finished their matching tasks.

We can see this point from another perspective of Property 3. According to Property 3, a neighbor array will be accessed by Δe_k's matching procedure if $T(\Delta e_k)$ lies within the array's lifetime $(ct_{min}, dt_{max}]$. From another viewpoint, once all updates Δe_k with $T(\Delta e_k) \in (ct_{min}, dt_{max}]$ have finished their matching tasks, this neighbor array version will no longer be used and can be deleted.

1.6.2 Garbage Collection Graph (Gcg)

The garbage collector (GC) operates on a garbage collection graph (GCG), which helps GC release neighbor array versions that are no longer in use.



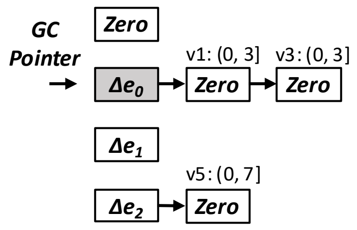
Δe0 = (v1, v3 +), T(Δe0) = 3
Δe1 = (v2, v5, -), T(Δe1) = 5
Δe2 = (v1, v5, +), T(Δe2) = 7
v1: (0, 3]
Δe2
t=7
Δe1
t=5
Δe0
t=3
Zero
t=0
v5: (0, 7]
v3: (0, 3]

Figure 8. A garbage collection graph (GCG) built from the 3 updates Δe_0, Δe_1, Δe_2 in Figure 5.

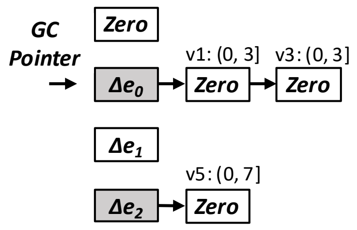
The GCG's vertices include a *Zero* vertex and all Δe_k that have been processed by the graph updater (GU). Each vertex in GCG is an edge update Δe_k timestamped with its arrival time, and the *Zero* vertex is a dummy edge update timestamped with 0. Each edge in GCG corresponds to a "deletable" neighbor array. Once we create a new neighbor version on MVG, the previous neighbor version becomes "deletable". For each deletable neighbor array v_x:$(ct_{min}, dt_{max}]$, we add a directed edge to GCG from the vertex (edge update) whose creation time is ct_{min} to the vertex whose creation time is dt_{max}.

Figure 8 shows a GCG example; it has four vertices, three of which are the edge updates in Figure 5. Since GU has created v_1:$(3,m]$, v_3:$(3,m]$, and v_5:$(7,m]$, the arrays v_1:$(0,3]$, v_3:$(0,3]$, and v_5:$(0,7]$ become deletable. Therefore, three edges are added to the GCG and are marked with their corresponding deletable neighbor arrays.

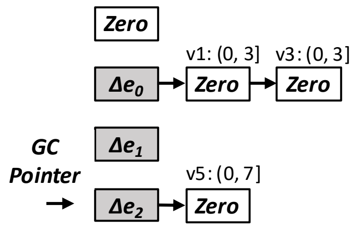
The GCG guides the release of neighbor arrays. An edge v_x:$(ct_{min}, dt_{max}]$ in the GCG corresponds to a neighbor array version, where the vertices (edge updates) below this edge represent updates Δe_k with $T(\Delta e_k) \in (ct_{min}, dt_{max}]$. According to Section 1.6.1, once all edge updates



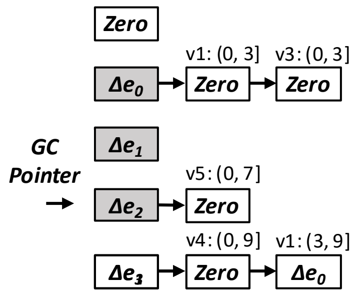
(a) The incremental matching of Δe_0 is finished, v_1:$(0,3]$ and v_3:$(0,3]$ can be released.



(b) The incremental matching of Δe_2 is finished.



(c) The incremental matching of Δe_1 is finished, v_5:$(0,7]$ can be released.



(d) A new $\Delta e_3 = (v_1, v_4, +)$ is appended to the end of GCG.

Figure 9. (a)–(d) illustrate how GC releases neighbor arrays on the GCG in Figure 8. This GCG is stored in an adjacency-list format.

below a GCG edge have completed incremental matching, the corresponding neighbor array can be released. For example, in Figure 8, if Δe_0, Δe_1, and Δe_2 under v_5:(0, 7] have finished their incremental matching, the neighbor array corresponding to v_5:(0, 7] can be released.

1.6.3 Implementation

In this section, we introduce how the garbage collector (GC) runs on the garbage collection graph (GCG), and its logic to release neighbor arrays.

GCG is stored in adjacency list format and manipulated collaboratively by GU, EX, and GC. GU appends vertices to GCG for each incoming Δe_k and appends edges for each deletable neighbor array created. EX marks whether each edge update in GCG has finished its incremental matching, and GC releases the corresponding neighbor array versions based on these marks. GC maintains a GC pointer pointing to Δe_k such that all incremental matchings from Δe_0 through Δe_k have been completed. Once the GC pointer reaches a Δe_k, the neighbor array versions associated with the edges starting from Δe_k will be released.

Figure 9 shows how GC releases the neighbor arrays on the GCG in Figure 8. In (a), once the incremental matching of Δe_0 is completed, we can move the GC pointer to Δe_0, and then $v_1{:}(0,3]$ and $v_3{:}(0,3]$ are released by GC. In (b), the incremental matching of Δe_2 is finished, but the GC pointer cannot be moved to it since Δe_1 before Δe_2 is unfinished. In (c), once the incremental matching of Δe_1 is finished, the GC pointer is moved to Δe_2, and thus $v_5 : (0,7]$ is released by GC. In (d), an edge $\Delta e_3 = (v_1, v_4, +)$ at time $T(\Delta e_3) = 9$ is added to the GCG by GU. This edge insertion creates new neighbor versions for both v_1 and v_4. Therefore, Δe_3 has two out-edges, each corresponding to a newly created deletable neighbor array.

As described above, the condition for the GC pointer to move to Δe_k is that the incremental matchings of Δe_0 to Δe_{k-1} are all completed, which means that the neighbor arrays corresponding to out-edges of Δe_k can be released. This logic is also based on the release condition in Section 1.6.1.

Since edge updates are added dynamically, GCG is a dynamic graph. Each vertex in this dynamic graph has at most two out-edges, so the overhead of maintaining this dynamic graph is very light. Unlike GU, which involves data copying, and EX, which performs NP-hard space search, GC is a lightweight module that does not pose any performance issues.

REFERENCES

[1] CAIDA. The caida ucsd anonymized internet traces 2013. `https://catalog.caida.org/dataset/passive_2013_dataset.xml`, 2013.

[2] Jingji Chen and Xuehai Qian. Decomine: A compilation-based graph pattern mining system with pattern decomposition. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 47–61, 2022.

[3] Xuhao Chen et al. Efficient and scalable graph pattern mining on {GPUs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 857–877, 2022.

[4] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 international conference on management of data*, pages 411–426, 2018.

[5] Ziming Li, Youhuan Li, Xinhuan Chen, Lei Zou, Yang Li, Xiaofeng Yang, and Hongbo Jiang. Newsp: A new search process for continuous subgraph matching over dynamic graphs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 3324–3337. IEEE, 2024.

[6] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1035–1044, 2016.

[7] Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F Italiano, and Wook-Shin Han. Symmetric continuous subgraph matching with bidirectional dynamic programming. *arXiv preprint arXiv:2104.00886*, 2021.

[8] L Qin, Y Peng, Y Zhang, X Lin, W Zhang, and Jingren Zhou. Towards bridging theory and practice: hop-constrained st simple path enumeration. In *International Conference on Very Large Data Bases*. VLDB Endowment, 2019.

[9] Linshan Qiu, Lu Chen, Hailiang Jie, Xiangyu Ke, Yunjun Gao, Yang Liu, and Zetao Zhang. Gpu-accelerated batch-dynamic subgraph matching. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 3204–3216. IEEE, 2024.

[10] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 11(12):1876–1888, 2018.

[11] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. Rapidmatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment*, 14(2):176–188, 2020.

[12] Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. Rapidflow: an efficient approach to continuous subgraph matching. *Proceedings of the VLDB Endowment*, 15(11):2415–2427, 2022.

[13] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[14] Shihan Wang and Takao Terano. Detecting rumor patterns in streaming social media. In *2015 IEEE international conference on big data (big data)*, pages 2709–2715. IEEE, 2015.

[15] Yihua Wei and Peng Jiang. Stmatch: accelerating graph pattern matching on gpu with stack-based loop optimizations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2022.

[16] Yihua Wei and Peng Jiang. Gcsm: Gpu-accelerated continuous subgraph matching for large graphs. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1046–1057. IEEE, 2024.

[17] Yichao Yuan, Haojie Ye, Sanketh Vedula Wynn Kaza, and Nishil Talati. Everest: Gpu-accelerated system for mining temporal motifs. *arXiv preprint arXiv:2310.02800*, 2023.