DCSM: Enabling Inter-Batch Parallelism for Continuous Subgraph Matching on GPU

anonymous authors

Abstract

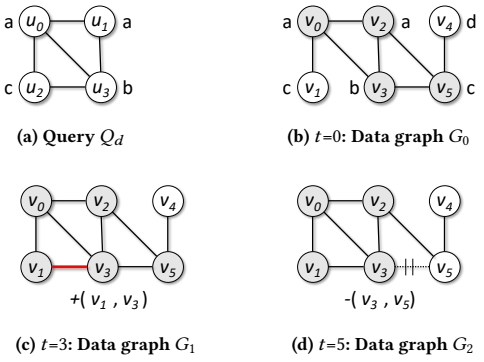
Continuous subgraph matching (CSM) is a fundamental building block in many real-world applications. While prior studies have explored executing CSM on heterogeneous systems with GPUs, they only exploit intra-batch parallelism and cannot process multiple batches concurrently—a capability essential for handling real-time requests. In this work, we propose a GPU-based system to accelerate CSM in practical, real-world settings. We adopt an algorithm-system co-design approach to unlock inter-batch parallelism. We introduce several key components, including a warp-specialized execution model and a multi-version graph data structure, along with version control logic for CSM tasks. Additionally, we propose optimizations such as warp-level parallel execution for data copying and incremental matching. Experimental results show that our system demonstrates optimal throughput and response time on GPU platforms across various update arrival rates.

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 [21]. It can also be applied to monitor money laundering in transaction networks [23], trace rumor propagation paths in social networks [30], and identify system anomalies in computer communication networks [16].

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).

Many efforts have been made to improve the performance of CSM on CPUs, primarily by reducing the search space during the matching process [12, 15, 20, 26]. However, due to the exponential time complexity of subgraph matching [28], CSM remains a computationally expensive procedure. To address this limitation, recent systems [22, 32] have begun exploiting the massive parallelism of GPUs to accelerate CSM.



(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.

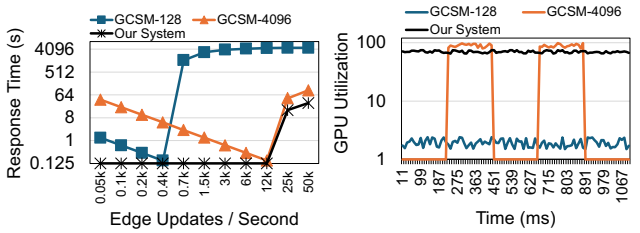
The existing GPU-based CSM systems process updates using fixed-size batches. Specifically, they wait until a predefined number of edge updates accumulate before grouping them into a batch and launching it on the GPU. To fully utilize GPU cores, the batch size is typically set to a large value (e.g., 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 [32]) 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.

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

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX



(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 10). Graph: Unlabeled Netflow [1].

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 [32], 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 [26].

2 Background

2.1 A Formal Definition of CSM

Definition 2.1 (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 2.2 (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 2.3 (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 2.4 (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 2.5 (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 2.6 (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 $\Delta e_k \in \Delta G$.

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 [22, 32] 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 [32] stores G_k in the CPU's main memory, whereas GAMMA [22] stores G_k in the GPU's global memory.

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

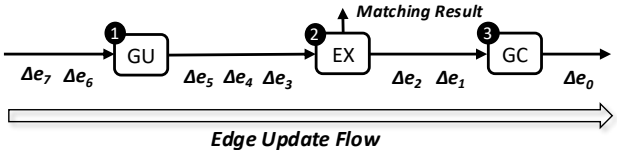


Figure 3: Overview of DCSM. DCSM consists of three functional modules: Graph Updater (GU), Executor (EX), and Garbage Collector (GC). The edge updates Δe_k flows through these modules sequentially.

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 [32] with a 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 4, 5, and 6 describe the three functional modules—GU, EX, and GC, respectively. Section 7 presents the evaluation results.

4 Graph Updater and Multi-Version Graph

All existing CSM systems [22, 32] on GPUs are derived from static subgraph matching frameworks [4, 31], 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 $(B_1, \ldots, B_n)$ in a strictly sequential manner: $P(B_1) \rightarrow M(B_1) \rightarrow U(B_1) \rightarrow \ldots \rightarrow P(B_n) \rightarrow M(B_n) \rightarrow U(B_n)$, where P, M, and U represent the Prepare, Match, and Update phases in Section 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.

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

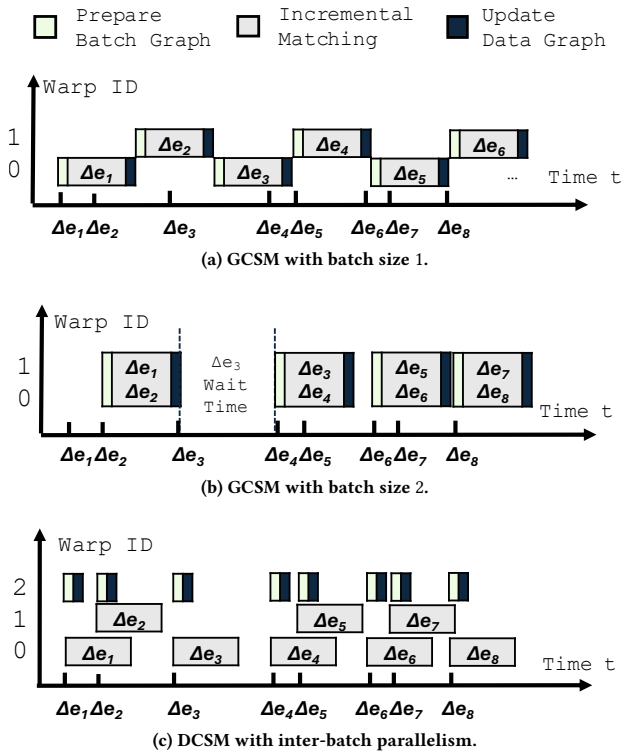


Figure 4: Comparison between GCSM and DCSM. Eight updates Δe_1 - Δe_8 arrive sequentially. The bar in the figure denotes warp activity along the time axis. Each bar is associated with an update Δe_k, labeled inside the bar. In practice, DCSM uses 5,000-10,000 warps for incremental matching, while only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

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

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

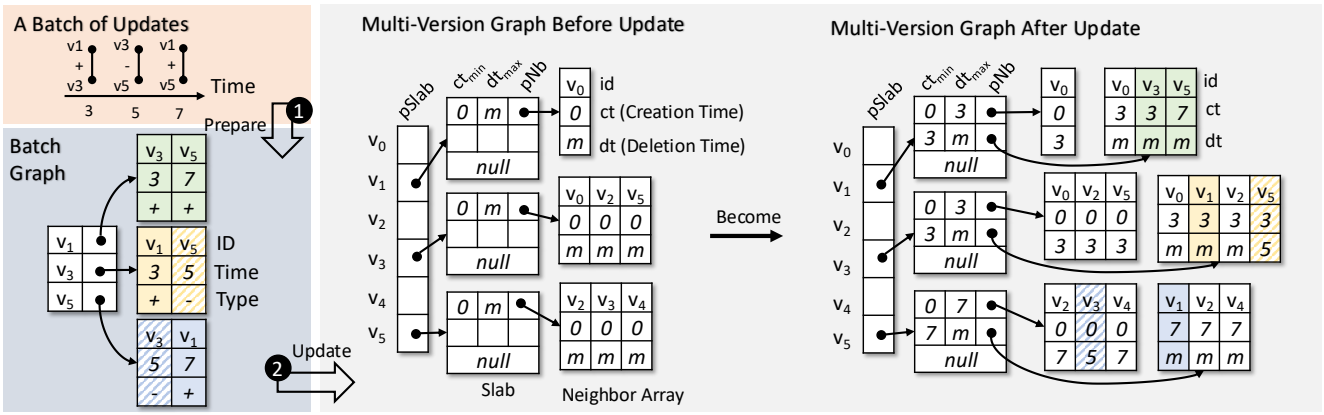


Figure 5: The multi-version graph (MVG) data structure and the procedure for updating it with a batch. The MVG before update represents G_0 in Figure 1. The vertex ID indexes the *pSlab*, pointing to a *Slab*. Each row of the *Slab* corresponds to a neighbor array version, storing its creation time (ct_{min}), 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: $(ct, dt]$ 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.

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.

4.3 Parallel Graph Updater

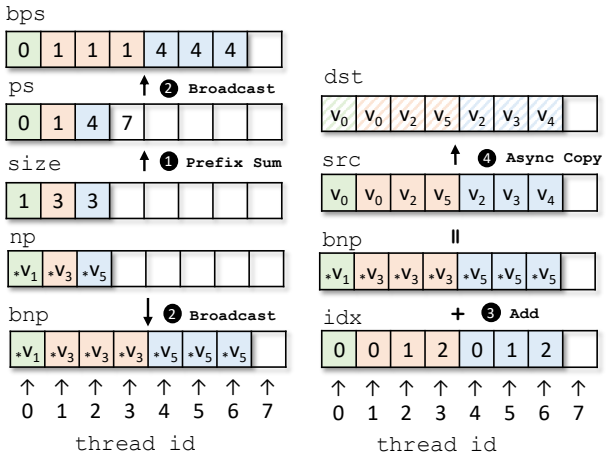


Figure 6: The process of copying three different-sized arrays in parallel using 8 threads of a warp. $*v_i$ represents the address of v_i's latest neighbor array version in the MVG.

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.

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 [28], 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 [4, 31] for EX to ensure both correctness and efficiency in CSM.

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 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.

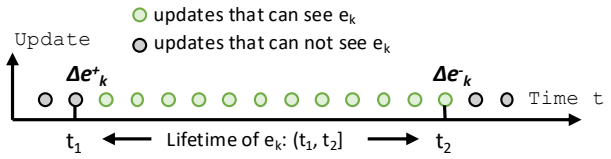


Figure 7: Visualization of edge e_k's lifetime and visibility across other updates.

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 5.1. 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.

5.2 Continuous Subgraph Matching-Specific Optimizations

5.2.1 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.

5.2.2 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.

5.2.3 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.

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.

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.

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.

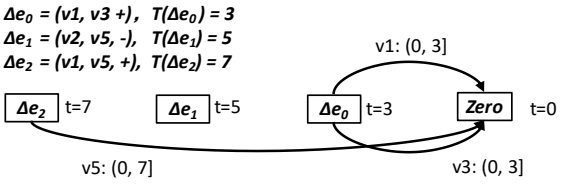
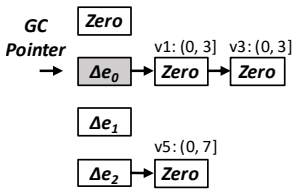


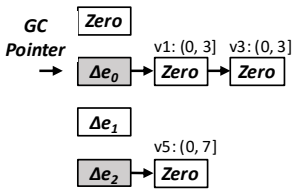
Figure 8: A garbage collection graph (GCG) built from the 3 updates Δe_0, Δe_1, Δe_2 in Figure 5.

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}.

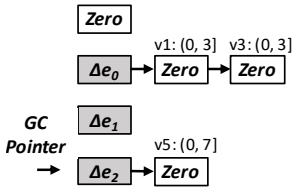
Figure 8 shows a GCG example; it has four vertices, three of which are the edge updates in Figure 5. Since GU has created v_1:$(3, m]$, v_3:$(3, m]$, and v_5:$(7, m]$, the arrays v_1:$(0, 3]$, v_3:$(0, 3]$, and v_5:$(0, 7]$ become deletable. Therefore, three edges are added to the GCG and are marked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



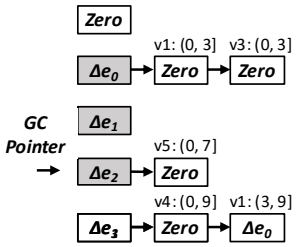
(a) The incremental matching of Δe_0 is finished, v_1:$(0, 3]$ and v_3:$(0, 3]$ can be released.



(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.

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 6.1, once all edge updates 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.

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 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.

7 Experimental Results

In this section, we first introduce the experimental setup, then compare the performance of DCSM with previous systems in terms of throughput and response time, and finally evaluate the effectiveness of our proposed optimizations through ablation studies.

7.1 Experimental Setup

Table 1: Graph datasets.

Graphs	Abbr.	# nodes	# edges	Max deg.
Enron	en	37K	183K	1383
Amazon	az	334K	0.9M	549
DBLP	db	317K	1.0M	343
Netflow	nf	3.1M	2.9M	0.2M
LSBench	lb	5.2M	20.3M	2.3M
LiveJournal	lj	4.0M	34.7M	14815

Platform: All experiments are finished on a host with two Intel Xeon Gold 6226R 2.9GHz CPUs (32 cores in total) and Nvidia RTX3090 GPU. The CPUs have 512GB RAM. Each GPU has 24GB global memory. All experiments run on Ubuntu-20.04. DCSM was developed in C++ and CUDA. The CPU code was compiled using GCC 9.4.0 with O3 optimization, and the GPU code was compiled using NVCC 11.6.

Datasets and query patterns: Table 1 lists the data graphs used in our experiments. All the data graphs are real-world graphs. Enron, Amazon, DBLP, and LiveJournal are networks of ground-truth communities from the SNAP dataset [14]. Netflow [2] is a graph of passive traffic traces. LSBench [13] is a synthesized graph social network from multiple sources. All graphs are simplified to simple undirected graphs. The dynamic graphs are generated by selecting 45% of the edges as insertion edges and 5% as deletion edges. The nine query patterns from size-3 to size-5 used in experiments are shown in Figure 10. The experiments include labeled and unlabeled matching. For labeled matching, we randomly assign five labels to the data graph and query pattern.

Baselines: We select three recent work, RapidFlow [26], CaLiG [34], and NewSP [15] as our CPU baselines. They are three state-of-the-art open-sourced continuous subgraph matching systems

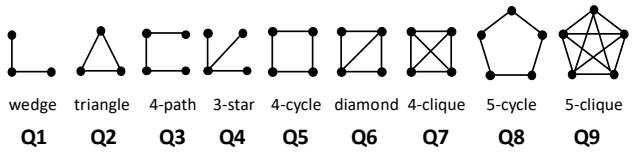


Figure 10: Query patterns for evaluation.

on the CPU, and have reported better performance than other CPU systems, including Graphflow [11], TurboFlux [12], SJ-Tree [6], IEDyn [9], and Symbi [20]. Each of them features different optimizations.

There are two previous GPU systems: GCSM [32] and GAMMA [22]. We summarize their performance and differences in Table 2. Since GAMMA is not totally open-sourced, we reimplement GAMMA as GAMMAr. We evaluate them following the experimental settings in the GAMMA paper. GCSM outperforms GAMMA due to two additional optimizations absent in GAMMA: code generation, which simplifies the program, and dynamic scheduling, which reduces load imbalance among thread blocks. GCSM consistently outperforms others in all test cases, so we use GCSM as our GPU baseline.

We also evaluate two straightforward but suboptimal strategies that can enable inter-batch parallelism: Edge-Extension and Hash-Indexing. Edge-Extension extends subgraphs by one edge at a time, so it does not require the neighbor arrays to be sorted, but it generates significantly more intermediate results than our algorithm, slowing down the matching step. Hash-Indexing stores the neighbor arrays as hash sets, but previous studies [4] have shown that it cannot achieve optimal performance on GPU platforms. We evaluate both approaches as baselines in the experimental section.

Another potential baseline approach is to exploit high parallelism within a small batch, such as traversing the search space in BFS order. However, previous studies [31, 33] have already extensively evaluated BFS and shown that it incurs high overhead due to intensive memory writes, large memory consumption, and thread synchronization, leading to significant performance degradation. Therefore, we did not include BFS in our baseline.

Table 2: Comparison of baselines. *Avg Time* is the average execution time on four data graphs and query patterns used in GAMMA. The *Avg Time* for GAMMA is taken from the original paper, while the *Avg Time* of others is obtained from our own experiments.

	Avg Time	Work Stealing	Dynamic Scheduling	Warp Centric	Redundancy Elimination	Code Generation
GAMMA	0.82	✓	x	✓	✓	x
GAMMAr	0.97	✓	x	✓	✓	x
GCSM	0.53	✓	✓	✓	✓	✓

Settings: Each GPU launches 82 thread blocks, each containing 2048 threads, which fully utilize the available active threads. RapidFlow, CaLiG, and NewSP run on a single thread because their algorithm can only process edge updates one by one, and their implementation is based on a single-threaded design. All systems adopt the same matching order of the query pattern. Unless otherwise

specified, we set the batch size each time consumed by the DCSM Graph Updater (GU) to 64, and set its timeout threshold x to 0.125ms. We allocate 4 warps to the Graph Updater, 1 warp to the Garbage Collector, and all remaining warps to the Executor. GCSM uses the same 82 blocks as DCSM, each containing 2048 threads, but all warps are used for matching.

7.2 Throughput

In this section, we compare the throughput of DCSM with that of other systems. We set the update rate to the maximum (i.e., all updates arrive at time 0) to measure how long each system takes to process all updates. A shorter processing time indicates that the system has higher throughput. The graph updater (GU) of DCSM consumes the updates with a batch size of 64; this setting can ensure the best performance.

7.2.1 Comparison with the CPU Systems. Table 3 shows the processing time of RapidFlow, CaliG, NewSP, and DCSM for matching query patterns Q1-Q9 on different data graphs. DCSM runs on a single GPU. In most test cases, RapidFlow outperforms the other two CPU-based systems, achieving the best performance on CPU. DCSM further outperforms RapidFlow, achieving speedups ranging from 1.7x to 42x, with an average of 10.5x. The experiment result indicates the effectiveness of our system. The speedup mainly comes from the massive parallelism of the GPU and the various optimizations we proposed.

For unlabeled matching, on the four graphs—Enron, Amazon, DBLP, and Netflow—DCSM achieves average speedups of 6.2x, 22x, 16x, and 8.6x over RapidFlow. For labeled matching on the four small graphs—Enron, Amazon, DBLP, and Netflow—DCSM achieves average speedups of 6.7x, 4.2x, 5.6x, and 5.1x over RapidFlow; on the two large graphs, LSBench and LiveJournal, the average speedup increases significantly to 14.5x and 13x. This is because the workload on small graphs is already low, even without labels, and adding labels to both the graphs and query patterns further reduces it. As a result, the search space may be limited to only a few thousand elements—or even just a few hundred. In such cases, warps experience higher overhead when competing for updates, as they may spend more time waiting to acquire the queue lock than performing actual matching.

In some cases, such as az-Q7, nf-Q3, and en-Q2, NewSP and CaliG can slightly outperform RapidFlow; this is because they include optimizations for specific cases, such as the reordering of extensions and operations in NewSP. In most cases, RapidFlow achieves better performance, since the local index used in RapidFlow greatly reduces the search space. Our system does not incorporate RapidFlow's local index method, resulting in a larger search space. Therefore, DCSM does not achieve the order-of-magnitude speedup over RapidFlow comparable to GPU's computational advantage over CPU. Nevertheless, we still outperform RapidFlow through massive parallelism.

7.2.2 Comparison with the GPU Naive Methods. As introduced in Section 7.1, we have two intuitive methods to enable inter-batch parallelism, but both have limitations. In this section, we compare the throughput of DCSM with these two naive methods: Hash-Indexing and Edge-Extension.

Table 3: Execution time of different systems for matching unlabeled and labeled query patterns. The '-' indicates a timeout after 8 hours. The default time unit for unlabeled matching is seconds, while for labeled matching it is milliseconds. The 's' after a number denotes seconds

	G	System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Unlabeled	*en*	Rapidflow	0.16	0.28	4.6	9.1	3.2	2.7	1.7	82.3	3.8
		CaliG	2.4	3.1	146	7.2	162	394	53	24332	1116
		NewSP	5.2	1.1	864	13	110	41	27	-	471
		HashIdx	0.17	0.2	3.3	3.8	1.8	3.4	0.5.	9.9	2.7
		Edge-Ext	0.1	0.1	1.5	2.1	1.6	8.4	8.8	92	12
		DCSM	0.1	0.1	1.5	2.1	0.7	1.2	0.2	7.1	1.2
	az	Rapidflow	1.1	2.4	3.7	1.9	4.6	5.2	4.2	7.8	5.1
		CaliG	4.1	5.7	15	7.8	32	39	33	230	379
		NewSP	2.2	1.9	28	5.2	13	5.1	4.6	92	8.4
		HashIdx	0.25	0.27	0.36	0.23	0.31	0.58	0.22	0.39	0.71
		Edge-Ext	0.09	0.1	0.1	0.1	0.1	0.5	1.2	5.5	3.2
		DCSM	0.09	0.1	0.2	0.1	0.2	0.2	0.1	0.3	0.4
	db	Rapidflow	1.1	2.6	5.2	3.2	5.5	6.7	6.6	29	16
		CaliG	5.4	7.8	54	12	102	152	101	5202	7638
		NewSP	4.8	3.1	181	9.1	64	45	76	3861	7363
		HashIdx	0.16	0.27	1.6	0.26	0.96	1.4	1.4	4.8	1.6
		Edge-Ext	0.05	0.1	1.2	0.2	1.2	2.8	3.4	116	11
		DCSM	0.05	0.1	1.2	0.2	0.5	0.6	0.6	1.8	0.6
	nf	Rapidflow	10.1	4.4	7.2	1278	5.8	5.5	5.1	6.2	5.9
		CaliG	247	58	238	1508	88	1678	1525	244	206
		NewSP	1654	2.7	1384	4027	480	5.8	1.8	803	2.1
		HashIdx	2.7	0.93	2.5	464	1.7	2.3	1.4	2.5	3.5
		Edge-Ext	1.2	0.6	1.2	232	1.2	1.1	1.7	21	4.8
		DCSM	1.2	0.6	1.2	232	0.9	0.9	0.8	1.7	1.2
Labeled	*en*	Rapidflow	27	78	37	38	133	102	127	996	42
		CaliG	32	44	188	72	299	408	164	7190	555
		NewSP	95	61	1283	135	306	125	105	6633	210
		HashIdx	20	9.4	19	23	28	49	55	221	80
		Edge-Ext	7	6	10	11	10	50	62	98	168
		DCSM	7	6	10	11	10	21	21	91	27
	az	Rapidflow	71	110	171	98	174	166	104	204	107
		CaliG	91	174	141	114	241	334	91	378	387
		NewSP	283	263	359	309	324	301	274	433	267
		HashIdx	91	43	48	71	85	53	88	50	66
		Edge-Ext	31	32	31	33	33	49	52	178	395
		DCSM	31	32	31	33	33	32	32	32	45
	db	Rapidflow	97	123	172	111	184	238	122	372	179
		CaliG	114	177	252	172	433	647	997	3945	13s
		NewSP	299	282	704	352	498	433	494	3885	4.8s
		HashIdx	53	56	82	35	60	106	53	85	69
		Edge-Ext	31	30	31	27	30	82	48	106	917
		DCSM	31	30	31	27	30	42	27	32	35
	nf	Rapidflow	1.1s	495	0.7s	154s	608	719	609	715	1007
		CaliG	1.7s	639	1.2s	611s	758	1247	1544	969	3053
		NewSP	18s	1183	11s	48s	4387	1164	1190	5083	1195
		HashIdx	0.67	749	1.1	40	427	215	285	260	185
		Edge-Ext	0.4s	116	0.4s	18s	149	373	501	837	1834
		DCSM	0.4s	116	0.4s	18s	149	110	101	96	128
	lb	Rapidflow	2.3s	2.5s	12s	455s	3.1s	2.9s	2.5s	173s	3.1s
		CaliG	9.5s	11s	449s	575s	80s	713s	60s	16973s	961s
		NewSP	22.8s	5.1s	2625s	497s	77s	6.3s	5.8s	9953s	6.1s
		HashIdx	2.9s	3.2s	1.4s	15s	2.3s	1.8s	0.43s	26s	1.5s
		Edge-Ext	1.3s	1.3s	1.1s	6.2s	1.3s	2.2s	1.1s	80s	4.2s
		DCSM	1.3s	1.3s	1.1s	6.2s	1.3s	1.1s	0.2s	16s	0.9s
	lj	Rapidflow	4.8s	7.1s	17s	17s	12s	92s	9.7s	101s	25s
		CaliG	30s	45s	265s	62s	264s	1124s	223s	22594s	5550s
		NewSP	26s	17s	1284s	39s	333s	111s	125s	-	5129s
		HashIdx	1.9s	1.8s	2.8s	3.7s	2.6s	25s	1.5s	29s	8.9s
		Edge-Ext	1.8s	2.9s	3.3s	3.5s	2.2s	76s	1.9s	282s	76s
		DCSM	0.8s	1.0s	1.3s	1.5s	1.2s	10s	1.4s	13s	3.2s

From Table 3, we can see that DCSM consistently outperforms Hash-Indexing. The Hash-Indexing method does not work well for GPU-based subgraph matching tasks, as it causes more thread divergence within a warp during set operations.

Edge-Extension performs especially well on sparse query patterns but is slower on dense query patterns (Q7, Q8, and Q9). This

is because the time complexity of Edge-Extension is $O(n^{|E(Q)|})$, which is higher than DCSM's $O(n^{|V(Q)|})$ on dense query patterns. As a result, Edge-Extension explores a larger search space than DCSM on dense query patterns. However, Edge-Extension still outperforms most CPU-based systems, as the high parallelism of GPUs offsets the drawbacks introduced by its algorithmic complexity.

Table 4: Performance comparison between GCSM and DCSM for matching unlabeled and labeled query patterns. The table shows the batch size b used by GCSM. When the batch size is set to b, GCSM and DCSM exhibit nearly the same processing time. Both GCSM and DCSM are executed on a single GPU.

	G	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Unlabeled	*en*	7456	5984	7264	5216	6784	5600	6304	7136	5088
	az	5600	6176	7136	6144	6880	7232	6848	6976	5632
	db	6400	5824	7520	6208	7200	5120	7648	6496	5152
	nf	6528	6176	6144	4864	6400	5824	4672	6272	5120
Labeled	*en*	6976	10848	6464	7712	10720	7584	8384	11008	8640
	az	8736	7328	6816	7968	8256	9888	6592	9504	7232
	db	9184	6752	7968	9024	6688	10304	9824	7424	8096
	nf	8448	8832	10976	9792	9632	8160	8288	10080	9312
	lb	6880	6144	6560	5536	5088	6112	5376	5056	5824
	lj	4768	7520	6592	5504	7072	5504	6560	6816	5696

7.2.3 Comparison with GCSM. Table 4 shows the performance comparison between GCSM and DCSM for unlabeled and labeled continuous subgraph matching. As introduced in Section 1, GCSM processes updates in fixed-size batches, which leads to low GPU utilization when the batch size is small. The graph updater (GU) of DCSM also consumes updates in fixed-size batches. We identify a batch size b at which GCSM and DCSM achieve the same processing time. While GCSM's processing time decreases as b increases, DCSM's execution time remains stable regardless of the batch size, because DCSM can process multiple batches in parallel.

We observe that the numbers in Table 4 roughly correspond to the number of active warps on the GPU. GCSM achieves the same performance as DCSM when processing with a batch size large enough to saturate GPU resources. For some small labeled data graphs, such as en, az, and db, GCSM requires a larger batch size, because a smaller batch is processed too quickly to effectively hide the kernel launch time for the next batch.

Figure 11 shows the trend of GCSM and DCSM performance with the batch size. GCSM's processing time becomes longer on small batch sizes, while DCSM remains stable regardless of the batch size. If all edge updates in the test data are grouped into an extremely large batch, GCSM has almost the same performance as DCSM. We also tested GPU utilization under different batch sizes. We can see that as the batch size decreases, DCSM's GPU utilization remains constant while GCSM's GPU utilization decreases roughly by half successively.

7.3 Response Time

In this section, we compared the response times of DCSM, GCSM, and RapidFlow over different update rates (updates / second). Figure 12 shows the experimental results.

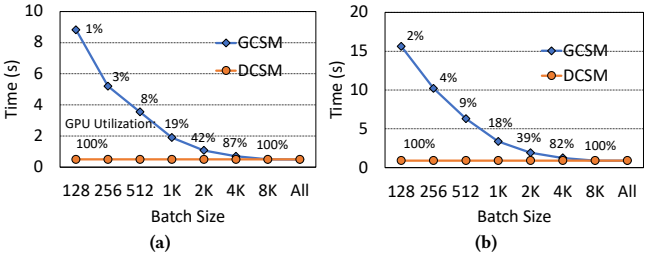


Figure 11: (a) and (b) show the trend of GCSM performance with batch size b on unlabeled db-Q5 and labeled lj-Q5. "All" refers to the batch size being equal to the total number of edge updates in the test data. The numbers marked on the line stand for GPU utilization.

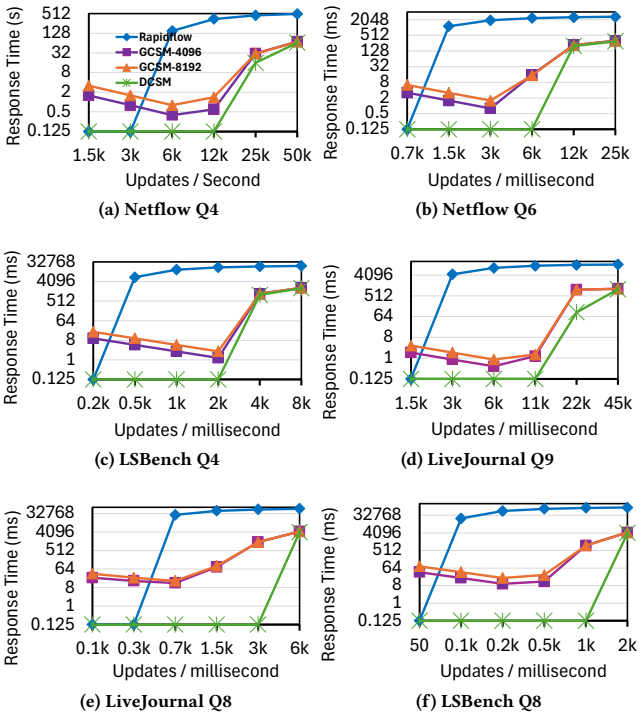
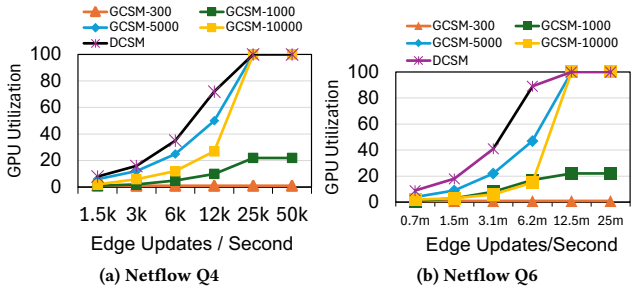


Figure 12: Response time over different update rates. The time unit is seconds for Netflow Q4 and milliseconds for all others. GCSM-4096 refers to GCSM configured with a batch size of 4096.

By observing real-world data, such as Twitter traffic and real-time updates in social networks, we found that in reality, the time intervals between two consecutive arriving updates in dynamic graphs follow an exponential distribution. We generated timestamps for the updates in the edge stream for each data graph according to this exponential distribution. By setting the parameters of the exponential distribution, we can control the time span between the last update and the first update. A longer time span means



(a) Netflow Q4 (b) Netflow Q6

Figure 13: GPU utilization vs. update rate (k: thousand, m: million) for different systems

fewer updates arrive per unit time, which has a lower Δe_k update rate. Conversely, a shorter time span has a higher Δe_k update rate. The horizontal axis in Figure 12 represents update rate.

The experimental results confirm the effectiveness of the DCSM. As shown in Figure 12, RapidFlow exhibits higher response time at high update rates because the processing capability of a single CPU is limited and cannot guarantee RapidFlow's high throughput. At low update rates, GCSM takes longer to form a batch, which also leads to increased response time. DCSM can effectively handle various update rates. The parallelism between batches enables DCSM to achieve optimal performance regardless of the update rate. Therefore, DCSM can flexibly adapt to traffic fluctuations over time in real-world scenarios.

We also compared against GCSM-128. However, due to GCSM-128's extremely low throughput, it cannot handle the update rates within the range shown in Figure 12; the GCSM-128 curve appears as a nearly flat line above RapidMatch. In Figures 12a to 12f, DCSM achieves 70.7x, 53.9x, 60.4x, 80.3x, 87.9x, and 60.4x higher throughput compared to GCSM-128, respectively. GCSM-128's response time drops to near-zero at an extremely low update rate, but GCSM with small batch size performs far worse than CPU systems under typical update rates.

Figure 13 illustrates GPU utilization as a function of the update rate. DCSM consistently achieves higher GPU utilization than GCSM across all update rates. With large batch sizes, GCSM exhibits low utilization due to its batching strategy, which delays the response time, whereas with small batch sizes, insufficient parallelism limits utilization. By contrast, DCSM dynamically schedules incoming updates to available warps, enabling high GPU utilization across update rates. At high update rates, the GPU utilization of DCSM, GCSM-500, and GCSM-10000 converges to 100%.

7.4 Overhead Analysis

The MVG data structure causes DCSM to perform more data copying than other systems, which increases memory access overhead.

Table 5 reports the time required for the graph updater (GU) to process all updates and apply them to the MVG. These data were collected while GU, EX, and GC were running concurrently, rather than with GU running in isolation.

By comparing Table 5 and Table 3, it can be observed that, in most cases, the time required by GU is significantly lower than

Table 5: Time (ms) spent by the graph updater (GU) to synchronously consume all updates.

	Batch Size	en	az	db	nf	lb	lj
Unlabeled	1	59	68	114	1050	x	x
	32	7	27	11	122	x	x
	128	5	16	9	70	x	x
	512	3	3	9	67	x	x
	2048	3	3	9	57	x	x
Labeled	1	11	7	12	210	513	1155
	32	7	2	2	24	52	104
	128	5	2	2	14	41	81
	512	3	2	2	13	39	76
	2048	3	2	2	11	39	77

that for incremental matching. This indicates that GU can produce updates faster than the executor (EX) can consume them, even when updating the MVG with a batch size of 1. Thus, EX is the performance bottleneck in most cases.

GU is slower than EX only when DCSM matches size-3 query patterns and GU uses a batch size of 1. For example, in Figure 3, lj-Q1 requires 800 ms for matching, while GU takes 1155 ms; nf-Q5 requires 800 ms, while GU takes 1050 ms. However, in DCSM, the batch size of GU is typically set to 64 or larger. Therefore, at high update rates, GU does not become a performance bottleneck. At low update rates, timeouts may cause GU to update the MVG with a batch size of 1, but this millisecond-level slowdown does not noticeably increase response time or affect user experience.

We also profiled DCSM's memory footprint. At high update rates, GPU memory consumption rises sharply within a few milliseconds and then decreases gradually at a rate comparable to that of EX, as GC operates at a speed determined by EX, which is much faster than GU. At update rates equal to or lower than the processing speed of EX, GPU memory consumption remains stable over time.

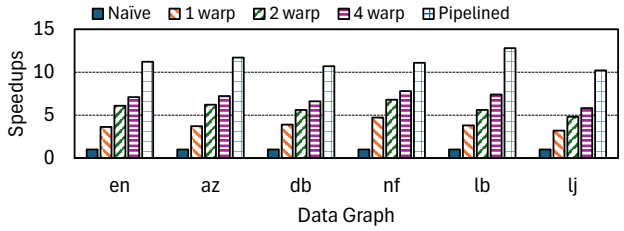
7.5 Ablation Study

Figure 14 shows the speedup achieved by the parallel graph updater (GU) across different data graphs.

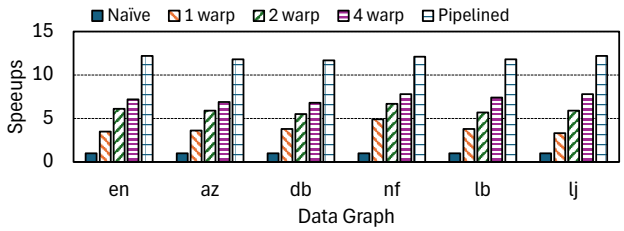
We observe that assigning more warps to the GU achieves better speedups, as more warps can issue more in-flight memory request instructions in parallel within the same cycle, thereby saturating the GPU global memory bandwidth. Higher parallelism ensures that memory throughput is not bounded by instruction dispatch.

However, when the number of warps continues to increase such that the total number of threads exceeds the batch size, further increasing warps no longer brings significant speedup. For example, in Figure 14a, using 8 warps produces almost the same speedup as using 4 warps. This is because different batches cannot be updated to the MVG in parallel, and the average neighbor size in the data graph is very small, resulting in the total length of all arrays to be copied being smaller than the number of available threads.

We also observe that a single warp achieves the most significant speedup compared to Naive, where one warp iterates 32 times. Since most neighbor arrays have a size of only 1-2, a single warp in parallel only needs to iterate 2 to 3 times. Single warp parallelization greatly reduces the number of iterations and increases memory request parallelism. Meanwhile, 2 warps only halve the number of iterations compared to 1 warp.



(a) Batch Size = 64



(b) Batch Size = 128

Figure 14: Speedups brought by parallel graph updater on different data graphs. Naive refers to a single warp sequentially copying different arrays, rather than copying multiple arrays in parallel.

Our proposed pipeline processing method also brings significant speedup to the GU, and becomes particularly effective under high update rates. The copy operation of the subsequent batch can hide the sort operation of the pr

Through these optimizations, the GU can process updates at high speed, thereby avoiding performance bottlenecks that would be slower than EX for most queries.

8 Related Works

Continuous subgraph matching (CSM) originates from static subgraph matching. The design of static subgraph matching systems has a significant impact on CSM. There are many efforts to design high performance systems for subgraph matching. These include CPU systems, such as GraphZero [18], Dryadic [17], and GraphPi[24], Peregrine [10], Automine [19], G2Miner [4], Sandslash [5], DecoMine [3], and RapidMatch [25] These also include GPU systems, such as Gunrock [29], GPSM [27], GSI [35], cuTS [33], PBE [8], and STMatch [31]. These systems introduce various program optimization methods to improve hardware utilization.

Continuous subgraph matching has been studied for decades on CPU. IncIsoMatch [7] is the first continuous subgraph matching work, it retrieves the graph region affected by the edge update of the query and performs the static subgraph matching again in the region to get the incremental results. Graphflow [11] eliminates the repeated matching in IncIsoMatch for each edge update and derives the multi-way join equations for batched continuous graph matching, performing incremental matching based on the equations. SJ-Tree [6] builds an index tree for computing incremental matching results. It stores partial results to eliminate redundant computation that can be precomputed before runtime, but the index tree leads to memory explosion when processing large graphs. TurboFlux [12] proposes a new data structure called data-centric graph structure that can reduce the storage amount of partial results to get a better tradeoff between memory consumption and computation overhead. SymBi [20] is the successive work of TurboFlux and proposes a pruning method for candidate vertices with query edges. Rapidflow [26] builds a local index for breaking the matching order fixation and a dual matching elimination to reduce the redundant computation caused by query automorphisms. CaliG [34] divides the query into two parts, it computes the partial results with the first part and gets the final result with the second part, the extension in the second part can be finished without backtracking. NewSP [15] gets the best performance on the CPU. NewSP avoids premature expansions of the search space by postponing expansion at the operation level.

In recent years, some studies [22, 32] have started using GPUs to accelerate continuous subgraph matching. GCSM [32] designs a sampling algorithm for memory optimization, it only places frequently accessed vertices of graphs on the GPU to support extremely large graphs that exceed the GPU memory size. GAMMA [22] is batch-dynamic subgraph matching with warp-level work stealing and coalesced search for reducing redundant computations. Both GCSM and GAMMA are offline systems, based on the assumption that we can form an extremely large batch size.

9 Conclusion

This paper presents the first continuous subgraph matching system on GPU capable of different batches in parallel. It offers a highly practical solution for various real-world scenarios. However, DCSM also introduces new challenges, such as memory inefficiency due to volatile memory and deeply hidden redundant computations that could be eliminated during preprocessing. These issues are not present in previous systems. Many previous works cache a large number of intermediate results to avoid recomputing incremental matching from scratch. A new challenge arises in how to manage these intermediate results efficiently on GPUs and how to make the indexing structures used to store them compatible with our system. There is still significant room for optimizing DCSM in terms of both performance and efficiency.

References

[1] CAIDA. 2013. The CAIDA UCSD Anonymized Internet Traces 2013. https://catalog.caida.org/dataset/passive_2013_dataset.xml.
[2] JE CAIDA. 2019. The CAIDA UCSD anonymized internet traces.
[3] Jingji Chen and Xuehai Qian. 2022. 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*. 47–61.
[4] Xuhao Chen et al. 2022. Efficient and scalable graph pattern mining on {GPUs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 857–877.
[5] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: a two-level framework for efficient graph pattern mining. In *Proceedings of the ACM International Conference on Supercomputing*. 378–391.
[6] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A selectivity based approach to continuous pattern detection in streaming graphs. *arXiv preprint arXiv:1503.00849* (2015).
[7] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)* 38, 3 (2013), 1–47.
[8] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. Gpu-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1067–1082.

[9] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1259–1274.

[10] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[11] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1695–1698.

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

[13] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. 2012. Linked stream data processing engines: Facts and figures. In *International Semantic Web Conference*. Springer, 300–312.

[14] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[15] Ziming Li, Youhuan Li, Xinhuan Chen, Lei Zou, Yang Li, Xiaofeng Yang, and Hongbo Jiang. 2024. NewSP: A New Search Process for Continuous Subgraph Matching over Dynamic Graphs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3324–3337.

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

[17] Daniel Mawhirter, Samuel Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. 2021. Dryadic: Flexible and fast graph pattern matching at scale. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 289–303.

[18] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2019. Graphzero: Breaking symmetry for efficient graph mining. *arXiv preprint arXiv:1911.12877* (2019).

[19] Daniel Mawhirter and Bo Wu. 2019. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 509–523.

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

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

[22] Linshan Qiu, Lu Chen, Hailiang Jie, Xiangyu Ke, Yunjun Gao, Yang Liu, and Zetao Zhang. 2024. GPU-Accelerated Batch-Dynamic Subgraph Matching. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3204–3216.

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

[24] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

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

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

[27] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*. Springer, 299–315.

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

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

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

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

[32] Yihua Wei and Peng Jiang. 2024. GCSM: GPU-Accelerated Continuous Subgraph Matching for Large Graphs. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1046–1057.

[33] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[34] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast continuous subgraph matching over streaming graphs via backtracking reduction. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

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