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

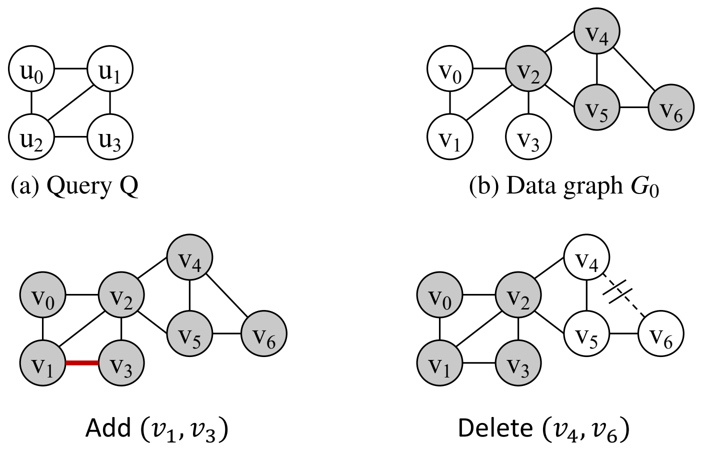
LIST OF FIGURES

1 GCSM: GPU-ACCELERATED CONTINUOUS SUBGRAPH MATCHING FOR LARGE GRAPHS

1.1 Introduction

Subgraph matching, which involves finding all matches of a query pattern Q in a data graph G, is a fundamental task in graph analytics. It is widely used for retrieving information from graph-structured data in various domains, including bioinformatics [13], social network analysis [15], and cybersecurity [15].

While subgraph matching on static graphs has been extensively studied in the past decade [3, 5, 8, 19, 23, 26, 27], there is a growing interest in supporting subgraph matching on dynamic graphs. The task is often referred to as *continuous subgraph matching* (CSM). Fig. 1 shows an example of CSM. Given a data graph G_0 and a query pattern Q. G_0 has one subgraph (v_2, v_4, v_5, v_6) that matches Q. If an edge (v_1, v_3) is added to G_0, the updated graph G_1 contains a new subgraph (v_0, v_1, v_2, v_3) that matches Q. If the edge (v_4, v_6) is removed from G_1 at the next step, the subgraph (v_2, v_4, v_5, v_6) will no longer be a match of Q and should be removed from the matching results.



(a) Query Q

(b) Data graph G_0

Add (v_1, v_3)

Delete (v_4, v_6)

(c) Graph update: $G_0 + \Delta E_0$

Figure 1. An example of continuous subgraph matching.

CSM can be useful in many scenarios. For example, the message transmission on a social network can be modeled as a dynamic graph, and CSM can be used to detect the spread of rumors [25]. The financial transactions among bank accounts are also a dynamic graph, and CSM can be used to monitor suspected transaction patterns such as money laundering [17].

Different algorithms and optimizations have been proposed for CSM [4, 6, 10, 21]. An early work [6] simply re-matches the query pattern on the data graph every time the data graph is updated. More recent work [4, 10, 11, 21] adopts an incremental matching approach, where the query pattern is only matched on the updated edges of the data graph. Their basic idea is to treat the dynamic data graph as multiple constantly updating tables and model CSM as an *Incremental View Maintenance* (IVM) problem [7, 18] (explained in more detail in Sec. 1.2).

Despite their different optimizations, the existing CSM systems are all CPU-based. As subgraph matching is NP-hard, it is always desirable to use GPU to accelerate the computation. Although many GPU-based systems have been proposed for static subgraph matching [26–28], utilizing GPUs to accelerate continuous subgraph matching faces a major challenge. Since the data graph is constantly updated and can easily exceed the GPU memory capacity, we must store the graph on CPU. A matching procedure on GPU involves a lot of data access from the CPU, which can significantly slow down the system.

Previous work [9] has proposed data caching for static subgraph matching on large graphs that cannot fit on a GPU. They divide the graph vertices into small bins and copy the k-hop neighbors of vertices in each bin to the GPU for processing. Here, k is the diameter of the query pattern, which ensures that the vertices copied onto the GPU include all vertices accessed in the matching process. Since many vertices are accessed multiple times during matching, caching them on the GPU reduces redundant accesses from the CPU and achieves better performance than a naive implementation. However, this method does not work for CSM. First, their "bin packing" algorithm for dividing the vertices involves an expensive clustering procedure. While the overhead can be justified for static graphs because the k-hop neighbors can be reused by different query patterns, it cannot be amortized in a dynamic setting. Second, the k-hop neighbors of a small batch of updated edges can still be very large and exceed the GPU memory.

We propose a fine-grained data caching method to achieve efficient continuous subgraph matching on a GPU for large graphs. Our method is based on an important observation that the vertices are accessed in a highly skewed frequency distribution, and most of the k-hop vertices are actually not accessed during matching. Even within the accessed vertices, only a tiny fraction is frequently accessed. If we select the vertices that are frequently accessed and cache them on the GPU, it will significantly reduce CPU-GPU data communication.

The remaining question is how to obtain the most frequently accessed vertices for matching a batch of incoming edges.

To obtain the frequent vertices, we perform multiple random walks from the updated edges on the data graph and estimate the access frequency of different vertices on the CPU. Our random walk strategy ensures that the estimated access frequency is an unbiased estimation of the true access frequency. We then copy the neighbor lists of the most frequently accessed vertices to the GPU and run an exact incremental matching procedure on GPU. Since the distribution of data access frequency is highly skewed for real-world graphs, the matching process only accesses data on GPU most of the time. If it requires vertices that are not on the GPU, data can be accessed from the CPU main memory through the GPU zero-copy mechanism.

In summary, we make the following contributions:

- Our work is the first to support continuous subgraph matching on a CPU-GPU system and achieve state-of-the-art performance by overcoming the data transfer bottleneck between the CPU and GPU.
- Our system provides an end-to-end design that supports both efficient dynamic graph maintenance on the CPU and efficient subgraph matching on the GPU.
- We proposed an efficient random walk technique to identify the most frequently accessed vertices for data caching on GPU.
- We developed an efficient CUDA kernel for incremental subgraph matching on GPU.

We evaluated our system using a variety of query patterns and input graphs and compared it with two CPU baselines and four naive GPU implementations. The experiments show that our system addresses the data movement bottleneck in naive GPU implementations, and achieves 1.4x to 2.9x speedups (with an average of 1.8x) against the best naive GPU version. Our system also significantly outperforms the CPU baselines (including a state-of-the-art CPU system [21]), achieving 1.4x to 11.4x speedups (with an average of 4.1x) against the best CPU implementation.

1.2 Background

This section gives a formal definition of the continuous subgraph graph matching problem and describes a join-based algorithm for solving the problem. We also provide a background on CPU-GPU data communication to facilitate our discussion.

1.2.1 Problem Definition

A *graph* G is defined as $G = (V, E, L)$, consisting of a set of vertices V, a set of edges E, and a labeling function L that assigns labels to vertices. 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 1 (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 $L_a(v_i) = L_b(f(v_i)), L_a(v_j) = L_b(f(v_j))$.

The subgraph matching problem is defined as finding all the subgraphs in G that are isomorphic to a given query graph Q. A *continuous subgraph matching* (CSM) procedure aims to find the matching subgraphs of a given query in a *dynamic* graph.

Conventionally, a dynamic graph is modeled as a sequence of edge updates $[(e_0, \oplus), (e_1, \oplus), \ldots]$ applied to an initial graph G_0. Here, e_i represents an edge, and the symbol $\oplus$ can be either $+$ or $-$, meaning an edge insertion or deletion. A newly inserted edge may consist of new vertices, while the deleted edges only involve existing vertices. The edge updates generate a sequence of graph snapshots $[G_0, G_1, G_2, \ldots]$ where $G_{k+1} = G_k \oplus e_k$.

```
// iterate over edges in $G_0$ that
// match $(u_0,u_1)$
for ($(x_0,x_1) \in E$) {
  //$x_2$ is connected to $x_0$ and $x_1$
  for ($x_2 \in N(x_0) \cap N(x_1)$) {
  //$x_3$ is connected to $x_1$ and $x_2$
    for ($x_3 \in N(x_1) \cap N(x_2)$) {
      output($x_0,x_1,x_2,x_3$); } } }
```

(a) Matching Q on G_0

```
//iterate over edges in $\Delta E$ that
//match $(u_0,u_1)$
for ($(x_0,x_1) \in \Delta E$) {
  //$\Delta R_1$ joins with updated $R'_2,R'_3$
  for ($x_2 \in N'(x_0) \cap N'(x_1)$) {
  //and with updated $R'_4,R'_5$
    for ($x_3 \in N'(x_1) \cap N'(x_2)$) {
      output($x_0,x_1,x_2,x_3$); } } }
```

(b) ΔM_1

```
//iterate over edges in $\Delta E$ that
//match $(u_0,u_2)$
for ($(x_0,x_2) \in \Delta E$) {
  //$\Delta R_2$ joins with $R_1$ and $R'_3$
  for ($x_1 \in N(x_0) \cap N'(x_2)$) {
  //and with updated $R'_4$ and $R'_5$
    for ($x_3 \in N'(x_1) \cap N'(x_2)$) {
      output($x_0,x_1,x_2,x_3$); } } }
```

(c) ΔM_2

```
//iterate over edges in $\Delta E$ that
//match $(u_1,u_2)$
for ($(x_1,x_2) \in \Delta E$) {
  //$\Delta R_3$ joins with $R_1$ and $R_3$
  for ($x_0 \in N(x_1) \cap N(x_2)$) {
  //and with updated $R'_4$ and $R'_5$
    for ($x_3 \in N'(x_1) \cap N'(x_2)$) {
      output($x_0,x_1,x_2,x_3$); } } }
```

(d) ΔM_3

```
//iterate over edges in $\Delta E$ that
//match $(u_1,u_3)$
for ($(x_1,x_3) \in \Delta E$) {
  //$\Delta R_4$ joins with $R_1$
  for ($x_0 \in N(x_1)$) {
  //and with $R_2,R_3$, and updated $R'_5$
    for ($x_2 \in N(x_0) \cap N(x_1) \cap N'(x_3)$) {
      output($x_0,x_1,x_2,x_3$); } } }
```

(e) ΔM_4

```
//iterate over edges in $\Delta E$ that
//match $(u_2,u_3)$
for ($(x_2,x_3) \in \Delta E$) {
  //$\Delta R_5$ joins with $R_2$
  for ($x_0 \in N(x_2)$) {
  //and with $R_1,R_3,R_4$
    for ($x_1 \in N(x_0) \cap N(x_2) \cap N(x_3)$) {
      output($x_0,x_1,x_2,x_3$); } } }
```

(f) ΔM_5

Figure 2. Continuous subgraph matching implemented as worst-case optimal join. E represents the edges in the initial graph. ΔE represents a batch of edge updates. N and N' represent the original and the updated neighbor lists respectively.

CSM can be conducted on a dynamic graph with either single-edge updates or batch updates. In the single-edge setting, a matching procedure is invoked at each edge update $(e_k, \oplus)$ to find all the subgraphs that contain e_k. Depending on whether e_k is inserted or deleted, the subgraphs are either added or deleted from the previous matching result. In the batch setting, CSM computes the incremental subgraphs for a batch of edges simultaneously. A more rigorous description of the CSM algorithm is given below.

1.2.2 Worst-case Optimal Join for Csm

It is well-known that subgraph matching on a static graph can be considered as a multi-way join on graph edges [12, 14, 22]. For example, matching Q in G_0 in Fig. 1 is equivalent to

$R(u_0,u_1) \bowtie R(u_0,u_2) \bowtie R(u_1,u_2) \bowtie R(u_1,u_3) \bowtie R(u_2,u_3)$ where $R(u_i,u_j)$ is a relation that contains the edges in the data graph that can be mapped to the edge (u_i,u_j) in the query graph. $R(u_0,u_1) \bowtie R(u_0,u_2)$ returns a list of subgraphs with two edges connected on a common vertex mapped to u_0. Suppose the result of the first join is $R(u_0,u_1,u_2)$. We can see that $R(u_0,u_1,u_2) \bowtie R(u_1,u_2)$ returns the subgraphs in $R(u_0,u_1,u_2)$ with the two vertices mapped to u_1 and u_2 connected. Each join operation extends an edge on some partially matched subgraphs until all the edges in the query pattern are matched. More formally, for any query pattern Q, its matching subgraphs in a data graph G can be computed as $\bowtie_{e(u_x,u_y)\subseteq E(Q)} R(u_x,u_y)$ where $E(Q)$ is the edge set of the query graph and $R(u_x,u_y) = \{e(v_x,v_y) \subseteq E(G) | L(v_x) = L(u_x) \wedge L(v_y) = L(u_y)\}$.

Based on the connection between subgraph matching and multi-way join, CSM can be modeled as an *Incremental View Maintenance* (IVM) problem [7,18], which joins multiple constantly updating tables $R_1,\ldots,R_m$ where m is number of edges in the query pattern. Instead of joining from scratch after each update, it only computes an incremental join result for the update. Suppose the update to relation R_i is ΔR_i and $R'_i = R_i + \Delta R_i$ is the table after the update. The incremental join result ΔM can be computed as

$$\begin{aligned}
\Delta M_1 &= \Delta R_1 \bowtie R'_2 \bowtie \ldots \bowtie R'_m \\
\Delta M_i &= R_1 \bowtie \ldots \bowtie \Delta R_i \bowtie R'_{i+1} \bowtie \ldots \bowtie R'_m \\
\Delta M_m &= R_1 \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie \Delta R_m \\
\Delta M &= \bigcup_{i=1}^{n} \Delta M_i.
\end{aligned} \tag{1}$$

The formula can be verified by subtracting the join result before the update (i.e., $R_1 \bowtie \ldots \bowtie R_n$) from the join result after the update (i.e., $(R_1 + \Delta R_1) \bowtie \ldots \bowtie (R_n + \Delta R_n)$). Readers are referred to [10] for a more detailed explanation. In continuous subgraph matching, $R_i(u_x,u_y)$ is a relation corresponding to an edge (u_x,u_y) in the query graph. ΔR_i includes the updated edges (either added or deleted) that can be mapped to (u_x,u_y).

Since CSM can be computed by multi-way joins, previous work [1,10] has employed worst-case optimal join (WCOJ) algorithms (e.g., Leapfrog Triejoin [24]) for CSM. When applied to subgraph matching, it joins multiple edge lists on a common vertex at each step, instead of binary joining an edge at each step. The algorithm can be expressed as nested loops, as shown in Fig. 2. The loop in Fig. 2a matches Q in Fig. 1a on the initial graph G_0. The algorithm first iterates over all graph edges that match (u_0, u_1). Since u_2 is adjacent to both u_0 and u_1, the nodes that match u_2 can be computed by performing a set intersection on $N(x_0)$ and $N(x_1)$. Next, since u_3 is connected to both u_1 and u_2, the nodes that match u_3 can be computed as the set intersection of $N(x_1)$ and $N(x_2)$. Once all the pattern vertices are matched, the matching subgraph is outputted in the innermost loop.

Since Q has five edges, we need five nested loops to compute ΔM_1 to ΔM_5. Fig. 2b shows the nested loop for computing ΔM_1. It iterates over all edges in ΔE that match (u_0, u_1), which correspond to ΔR_1 in (1). Because $\Delta R_1(u_0, u_1)$ joins with the updated $R'_2(u_0, u_2)$, $R'_3(u_1, u_2)$, $R'_4(u_1, u_3)$, $R'_5(u_2, u_3)$, the program computes matching nodes for u_2 by performing set intersections on updated neighbor lists $N'(x_0)$ and $N'(x_1)$. The matching nodes for u_3 are also computed with the updated neighbor lists. The remaining loops are generated based on a similar argument.

It is known that the WCOJ algorithm has a time complexity bounded by the worst-case output size of the join result [14]. Therefore, the join operations in Formula (1) has a time complexity bounded by the size of ΔM_i, which follows the AGM inequality [2]:

$$|\Delta M_i| \leq \prod_{j=1}^{i-1} |R_j|^{\mu_j} \cdot |\Delta R_i|^{\mu_i} \cdot \prod_{j=i+1}^{m} |R'_j|^{\mu_j} \tag{2}$$

where $\mu = (\mu_1, \ldots, \mu_m)$ is a fractional edge cover of the query pattern Q, which follows the constraints $\mu_j > 0, \forall j \in \{1, \ldots, m\}$ and $\forall v \in V(Q), \sum_{e \in E(Q), v \in e} \mu_e \geq 1$. The time complexity of continuous subgraph matching is $O(\sum_{i=1}^{m} |\Delta M_i|)$.

1.2.3 Cpu-gpu Communication

In a CPU-GPU system, the GPU is connected to CPU through PCIe or NVLink. A program running on the GPU can directly access its global memory. However, because the size of the global memory is limited (typically 10$\sim$30GB on a modern GPU), the data of a program may exceed the GPU memory limit. In such cases, we must store data in the CPU main memory and synchronize data between the CPU and GPU. This is often referred to as "out-of-core" GPU computation in the literature [20]. CUDA provides three ways for CPU-GPU data communication [16]: 1) *DMA*, 2) *unified memory*, and 3) *zero-copy access*.

The DMA API (`cudaMemcpy`) is efficient for transferring large blocks of data between CPU and GPU. It uses a copy engine to asynchronously move the data over PCIe rather than loads and stores. It offloads the computing units on GPU, leaving them free for other work. However, DMA is not efficient for transferring small data because every DMA request incurs an overhead for packing the data and setting up the communication. Unified memory enables a GPU kernel to access data on the CPU directly, without explicit data transfer operations. This is accomplished by allocating a chunk of managed memory on the CPU (through `cudaMallocManaged`) and mapping it into the GPU address space. When the GPU kernel attempts to access the mapped data, the GPU automatically transfers the pages that contain the data between the CPU and GPU. This simplifies programming and supports caching for the accessed pages. However, unified memory is not efficient for fine-grained data access, as the data are always transferred at page granularity (4KB), which wastes PCIe bandwidth. In contrast, zero-copy is most suitable for fine-grained data access between the CPU and GPU. It directly loads (or stores) data on the CPU in cache lines (128B). Compared to DMA and unified memory, it does not have the communication setup overhead and only copies data that are needed. However, zero-copy access stalls the GPU kernel. The GPU computing units must wait for the data access to finish before it can continue execution. Our system uses a combination of DMA and zero-copy access for synchronizing graph data between CPU and GPU and achieves high performance by caching the most frequently accessed data on the GPU.

1.3 Overview of Gcsm

The goal of our system is to use GPU to accelerate the computation in Fig. 2. Since previous work has mapped the static matching procedure in Fig. 2a onto GPU [9, 26–28], the focus of this work is to efficiently run the incremental matching procedures (Fig. 2b-f) on GPU. The main challenge is how to support real-world graphs that do not fit on GPU.

Our system is designed based on an important observation: Although a batch of edges may have many vertices in its k-hop neighborhood, only a small fraction of them are frequently accessed during the matching procedure. Our experiments with different input graphs and query patterns show that the top 5% of most frequently accessed vertices account for over 80% of the memory access (see Fig. 15a). This strong data locality indicates that the program could greatly benefit from data caching on the GPU if we can identify the most frequently accessed vertices.

To obtain the most frequently accessed vertices, we propose to run multiple random walks from the updated edges on the CPU. Our random walk strategy ensures that frequent vertices are more likely to be accessed during sampling, and thus, the sampled vertices have a large overlap with the most frequently accessed vertices in the actual matching. In fact, we can obtain an unbiased estimation of the access frequency of vertices based on the sampling results. Details of our random walk technique are described in Sec. 1.4.

Once the frequent vertices are obtained, their neighbor lists are packed and sent to the GPU as cached data for exact matching on GPU. During the matching procedure, GPU accesses the neighbor lists from the cache whenever possible. If a neighbor list is not cached on GPU, the GPU reads data directly from CPU memory. The main challenge is how to support efficient access to both the original and new neighbor lists (N and N' in Fig. 2) and how to achieve efficient data transfer for the dynamically updated neighbor lists from CPU to GPU. We explain our data structure design and GPU implementation in Sec. 1.5.

Fig. 3 shows the workflow of our system. For every batch of edge updates ΔE_k, our system maintains the dynamic graph and performs incremental matching in five steps:

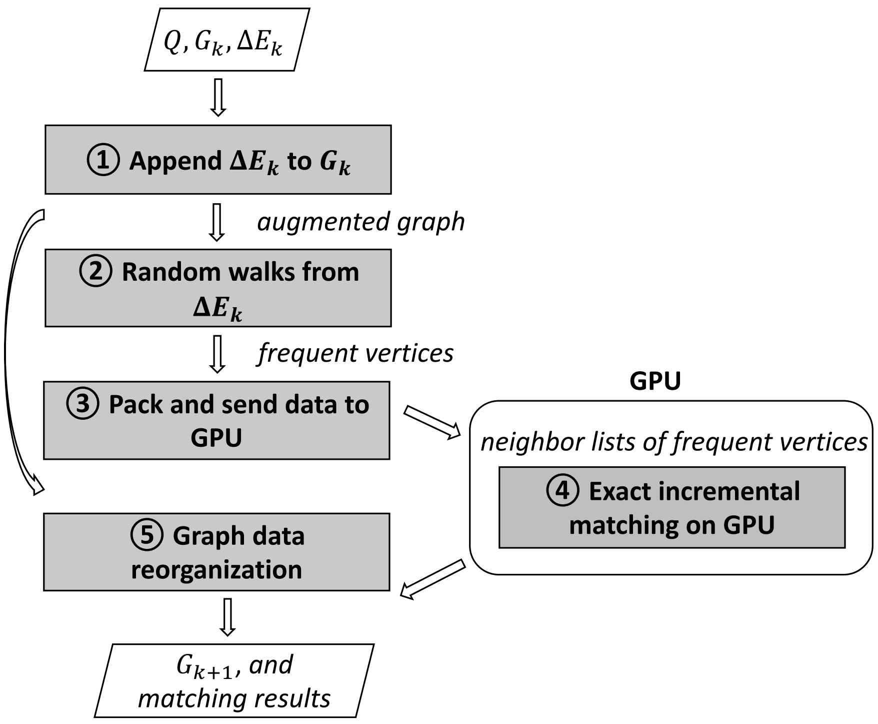

$Q, G_k, \Delta E_k$
① Append ΔE_k to G_k
augmented graph
② Random walks from ΔE_k
frequent vertices
③ Pack and send data to GPU
GPU
neighbor lists of frequent vertices
④ Exact incremental matching on GPU
⑤ Graph data reorganization
G_{k+1}, and matching results

Figure 3. Workflow of GCSM.

① The edge updates ΔE_k are appended to the neighbor lists of G_k on the CPU.

② Multiple random walks from the updated edges are generated on the CPU to obtain a set of frequent vertices;

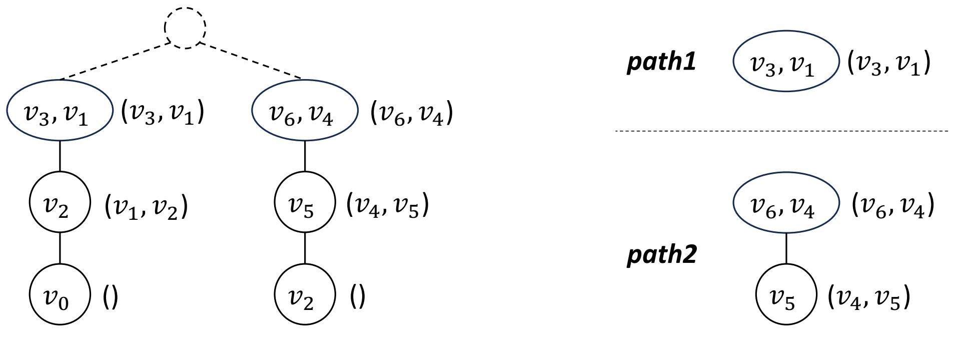
③ The neighbor lists of the frequent vertices are packed and copied to the GPU;

④ An exact incremental matching procedure is executed on the GPU;

⑤ The graph data are reorganized on the CPU to ensure the neighbor lists of G_{k+1} are sorted.

1.4 Obtaining Frequent Vertices

An important step of our system is to identify the vertices that can serve as the optimal cache data for the matching task on GPU. The procedure should meet two requirements: 1) Its overhead must be small compared to the exact matching on the GPU; 2) To ensure good cache efficiency, the identified vertices should have substantial overlap with the most frequently accessed vertices



(a) An example execution tree

(b) Two sampled paths

Figure 4. Sampling multiple paths from an execution tree.

during exact matching. To achieve these two goals, we propose a frequency estimation technique based on random walks on the graph.

1.4.1 Estimating Access Frequency With Random Walk

Consider the exact matching procedure in Fig. 2. The execution of each nested loop can be depicted as a tree structure where different tree levels represent different loop levels and each tree node represents one iteration of the loop. For example, Fig. 4a shows the execution tree of the nested loop in Fig. 2b with G_0 and ΔE from Fig. 1. The first tree level contains two nodes representing the two edges (v_3, v_1) and (v_6, v_4) in ΔE. (For simplicity of illustration, we did not include the reverse edges (v_1, v_3) and (v_4, v_6) in the drawing.) The node (v_3, v_1) has one child node v_2 which represents the intersection of $N'(v_3)$ and $N'(v_1)$. Similarly, the node (v_6, v_4) has one child node v_5 representing the intersection of $N'(v_6)$ and $N'(v_4)$. The nodes in the third tree level represent the iterations of the innermost loop in Fig. 2b. We differentiate a 'node' in the execution tree from a vertex in the graph. The computation in each node of the tree requires accessing the neighbor lists of multiple vertices, as shown in the parenthesis in Fig. 4a.

Our main idea for obtaining frequent vertices with minimal overhead is to sample multiple paths from the execution tree instead of executing the entire tree. The sampling process can be considered as conducting multiple random walks on the data graph guided by the matching procedure. Specifically, we start by randomly selecting one edge from ΔE, with a probability of $1/|\Delta E|$. From the sampled edge, we compute the set of matching vertices V for the next pattern vertex. We then randomly select one vertex from V, with a probability of $1/|V|$. After the matching vertex is selected, we decide whether to continue the walk with a probability of $|V|/D$, where D is the maximum degree of the data graph. Fig. 4b shows two example paths sampled from the execution tree in Fig. 4a.

During the random walk, we record the access to each vertex at each step (shown in the parenthesis in Fig. 4b). Suppose a random walk generates a path of k nodes in the execution tree. We know that the first node is sampled at probability $1/|\Delta E|$, and the remaining nodes are sampled at probability $(1/|V_i|) \cdot (|V_i|/D) = 1/D$. An unbiased estimation of the access frequency of different vertices can be obtained as

$$\tilde{C_v} = \sum_{i=1}^{k} |\Delta E| \cdot D^{i-1} \cdot c_{v,i}. \tag{3}$$

Here, $c_{v,i}$ is equal to 1 if vertex v is accessed in the ith node of the random walk, and 0 otherwise.

Our random walk technique is reminiscent of the neighbor sampling technique for approximate subgraph matching [**?**, **?**]. However, unlike the previous technique, which samples an entire path of the execution tree and estimates the number of subgraphs at the bottom level, our random walk may stop at any level of the tree, and it estimates the access frequency of vertices at all levels. It is clear that a single random walk will not be sufficient for obtaining a good set of frequent vertices. To reduce the estimation variance, we generate multiple random walks and use the average of $\tilde{C_v}$ as our estimate. The effectiveness of our method in identifying frequent vertices is summarized below.

Theorem 1. Suppose the access frequency of two vertices x and y are C_x and C_y in an exact matching procedure and $C_x = (1+\alpha)C_y$ with $\alpha > 0$. The probability that our estimation method incorrectly ranks y before x is bounded as

$$\Pr[\tilde{C_x} < \tilde{C_y}] \leq \frac{(n-1)(2+\alpha)|\Delta E|D^{n-2}}{\alpha^2 M C_y}. \tag{4}$$

Here, $n \geq 2$ is the pattern size, and M is the number of generated random walks.

Formula (4) indicates that the probability of incorrect ranking is smaller for vertices with a larger difference between C_x and C_y. As more random walks are drawn (i.e., a larger M), this probability can be decreased to an arbitrarily small value. To achieve an estimation that correctly ranks the vertices with confidence δ, we set the right-hand side of (4) to be $\leq 1-\delta$ and compute the minimum number of random walks needed to achieve the confidence level:

$$M \geq \frac{(n-1)(2+\alpha)|\Delta E|D^{n-2}}{\alpha^2(1-\delta)C_y}. \tag{5}$$

Assuming the query pattern size n is a constant w.r.t. the input graph size, a random walk takes constant time. Formula (5) represents the time complexity of our estimation algorithm. The formula indicates that fewer random walks are needed for more frequent vertices (i.e., vertices with larger C_y).

Since C_y is unknown before the matching procedure, in practice, we can set M to a small value initially. Once we obtain the estimated access frequency of different vertices based on the M random walks, we can set C_y in (5) to the smallest estimated frequency and check if our initial M is large enough. If not, we calculate a new M based on (5), collect more samples, and re-estimate the access frequency.

Proof. For each node t at level i of the execution tree, we define a variable y_t and let $y_t = 1$ if v is accessed in node t, and $y_t = 0$ if v is not accessed. The number of accesses to v in an exact matching can be written as $C_v = \sum_{i=1}^{n} \sum_{t \in S_i} y_t$, where S_i represents all nodes at level i of the tree. We then define a random variable Y_t and let $Y_t = y_t$ if tree node t is sampled in a single walk, and $Y_t = 0$ if it is not sampled. We can see that $\mathbb{E}[Y_t] = P_t y_t + (1-P_t)\cdot 0 = P_t y_t$ where $P_t = 1/(|\Delta E|\cdot D^{i-1})$, and the $c_{v,i}$ in (3) is equal to $\sum_{t\in S_i} Y_t$. It follows that

$$\begin{aligned}\mathbb{E}[\tilde{C}_v] = \mathbb{E}\left[\sum_{i=1}^{k} |\Delta E|\cdot D^{i-1}\cdot c_{v,i}\right] &= \mathbb{E}\left[\sum_{i=1}^{n-1}\sum_{t\in S_i}\frac{Y_t}{P_t}\right] \\ &= \sum_{i=1}^{n-1}\sum_{t\in S_i}\mathbb{E}\left[\frac{Y_t}{P_t}\right] = \sum_{i=1}^{n-1}\sum_{t\in S_i} y_t = C_v.\end{aligned} \tag{6}$$

This validates that our estimation of access frequency in (3) is unbiased. The variance of Y_t is

$P_t(y_t - P_t y_t)^2 + (1-P_t)(0-P_t y_t)^2 = (1-P_t)P_t y_t^2$. It follows that

$$\begin{aligned}\mathrm{Var}[\tilde{C_v}] = \mathrm{Var}\left[\sum_{i=1}^{n-1}\sum_{t\in S_i}\frac{Y_t}{P_t}\right] \le (n-1)\sum_{i=1}^{n-1}\mathrm{Var}\left[\sum_{t\in S_i}\frac{Y_t}{P_t}\right] \\ \le (n-1)\sum_{i=1}^{n-1}\left(|\Delta E|D^{i-1}\sum_{t\in S_i}y_t^2\right) \le (n-1)|\Delta E|D^{n-2}C_v.\end{aligned} \tag{7}$$

The third step above is due to the independent sampling of $t \in S_i$. According to the law of large numbers, the sample average of M random walks converges to the expected value C_v and its variance is $\mathrm{Var}[\tilde{C_v}]/M$. Next, we define a random variable $e = \tilde{C} - C$. It is easy to see that $\Pr[\tilde{C_x} < \tilde{C_y}] = \Pr[e_x - e_y - \mathbb{E}[e_x - e_y] < -\alpha C_y]$. According to Chebyshev's inequality [**?**], we have

$$\Pr[\tilde{C_x} < \tilde{C_y}] \le \frac{\mathrm{Var}[e_x - e_y]}{\alpha^2 C_y^2} = \frac{\mathrm{Var}[\tilde{C_x}] + \mathrm{Var}[\tilde{C_y}]}{\alpha^2 C_y^2}. \tag{8}$$

Plugging (7) into (8), we obtain (4).

□

1.4.2 Multiple Random Walks With One Single Execution

According to (4), we need to generate many random walks to achieve a confident estimation. If we run the nested loop for each walk, the sampling process will be slow due to redundant set operations and poor data locality between different runs. To accelerate the process, we propose a novel implementation that merges multiple random walks into one single execution of the nested loop. Specifically, at the beginning of each iteration at loop level i, we generate a random number B_i to indicate the number of times the iteration is sampled in the M runs. Since the random walk samples a loop iteration (i.e., a neighboring node) with a fixed probability at each step, the number of times a node is sampled (i.e., B_i) follows a binomial distribution. It is obvious that B_1 follows a binomial distribution with the number of trials M and sampling probability $1/|\Delta E|$. If $B_1 \ge 1$, which means the iteration is sampled at least once, we continue to the second loop level. If $B_1 = 0$, which means the iteration is not sampled in the M runs, we skip the iteration and proceed to the

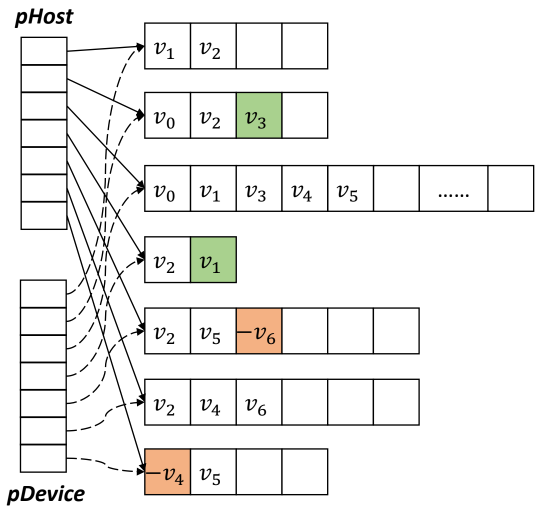


Figure 5. Maintenance of graph data structure on the CPU for the graph update in Fig. 1. New edges (colored in green) are appended to the end of corresponding neighbor lists; Deleted edges (colored in orange) are marked as negative values.

next iteration in the first loop. Similarly, for an iteration at loop level i, the number of times it is executed follows a binomial distribution with the number of trials B_{i-1} and sampling probability $1/D$. This simulated execution is equivalent to M independent random walks but with much better data locality and no redundant set operations.

We keep updating the estimated frequency of different vertices based on (3) without materializing the random walks, so the space complexity of our estimation procedure is $O(|V|)$.

1.5 Exact Incremental Matching With Cached Data On GPU

The main task of our system is to compute the exact incremental matching result on the GPU. The computation involves set operations on both the original and updated neighbor lists, as shown in the nested loop in Fig. 2. To achieve high performance, we need to 1) efficiently access the neighbor lists with minimal data transfer between the CPU and GPU, 2) perform fast set intersection on the neighbor lists, and 3) efficiently update the graph data on the CPU. This section describes the key designs and implementations of our system to achieve the goals.

1.5.1 Graph Data Structure On CPU

We store the data graph as adjacency lists in CPU memory, as shown in Fig. 5. For each vertex, we allocate a contiguous memory space on the CPU (using `cudaHostAlloc`) to store its neighbors. Then, we map the allocated memory into the GPU address space (using `cudaHostGetDevicePointer`). Two arrays are used to store the addresses of the neighbor lists: one (*pHost*) for the CPU addresses and one (*pDevice*) for the GPU addresses. The CPU addresses are used by the CPU to maintain the graph data structure. The GPU addresses are used by the GPU for accessing the neighbor lists during the matching procedure.

Upon each graph update, the following four steps are executed on the CPU:

1) The new edges are appended to the end of their corresponding neighbor lists. To achieve fast edge insertion, we preallocate the array of each neighbor list to double the size of the initial number of neighbors. If an array is full, a new array with double capacity will be allocated and initialized with the existing data. This ensures an average of $O(1)$ time complexity for edge insertion.

2) If new vertices are added, we allocate an array with an initial size of the average degree of the graph for each new vertex. The CPU and GPU addresses of the new arrays are appended to the end of *pHost* and *pDevice*. Similar to the neighbor lists, we preallocate some extra space for *pHost* and *pDevice* to achieve efficient insertion of new vertices.

3) The edges to be deleted are marked. In our implementation, we simply set the neighbor index v to $-v$ to indicate the edge is removed. Since the neighbor lists are always sorted after an update, each edge deletion can be done with a binary search on the neighbor list.

4) Each updated neighbor list is reorganized by removing the deleted edges and sorting the remaining elements. This step can be done with a merge-sort procedure in linear time for each updated neighbor list.

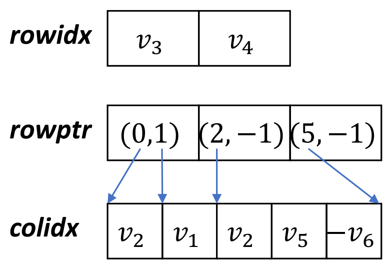


Figure 6. Graph data sent to GPU in DCSR format.

1.5.2 Data Preparation for GPU

During the graph update on the CPU, our system selects the most frequent vertices based on the estimated frequency and packs their neighbor lists into a contiguous chunk of memory. The data is stored in a Doubly Compressed Sparse Row (DCSR) format, which consists of three arrays: *rowidx*, *rowptr*, and *colidx*. The *rowidx* array, which is sorted by vertex indices, records selected vertices. The *colidx* array stores the neighbors of the selected vertices contiguously. For a vertex with an un-updated neighbor list, we simply copy the neighbor list to the array. For an updated neighbor list, we copy the data after Step-3 above. That is, the deleted neighbors are marked, and the new neighbors are appended to the end of the list. The *rowptr* array stores the offsets of the neighbor lists in *colidx*. Each entry in *rowptr* stores two offsets: one for the starting location of the original neighbor list, and one for the starting location of the appended new neighbors. If a selected vertex does not have new neighbors, the second offset is set to -1.

Fig. 6 shows an example of the DCSR data structure with the graph update in Fig. 5. Suppose v_3 and v_4 are selected for caching. The two vertex indices are stored in *rowidx*, and their neighbor lists are stored in *colidx*. The neighbors of v_3 start from location 0, and the new neighbor starts from location 1, so its offset is $rowptr[0] = (0,1)$. The neighbors of v_4 start from location 2, and there is no new neighbor, so its offset is $rowptr[0] = (2,-1)$. The last entry of *rowptr* indicates the length of *colidx*.

Note that the sizes of the three arrays are known before data copying. The lengths of *rowidx* and *rowptr* are determined by the number of selected vertices, and the length of *colidx* is determined by the sum of degrees of selected vertices. This allows us to allocate the three arrays contiguously with a single memory allocation on the CPU and send the packed data to GPU global memory with a single DMA transaction.

1.5.3 Parallel Incremental Matching On GPU

Once the data is cached on GPU, a GPU kernel for incremental matching is invoked to execute the nested loops in Fig. 2. We build the kernel on top of a recent GPU subgraph matching system called STMatch [26]. The system reportedly achieves state-of-the-art performance for static matching tasks on a GPU by managing the intermediate data using a stack data structure and incorporating a series of code optimization techniques.

To adapt STMatch for incremental matching, the first modification we make is to support access to both the original and new neighbor lists in the matching procedure (denoted as N and N' in Fig. 2). Since the new neighbors are appended to the end of each neighbor list, we can treat N' as $N \cup \Delta N$ where ΔN are the appended neighbors, and perform set operations involving N' separately for N and ΔN.

Since N and ΔN are sorted, we can exploit the existing code optimizations in STMatch (including code motion and unrolled set intersection with SIMD parallelism [26]) for each term on the right-hand side. For a neighbor list with deleted edges, since the deleted neighbors are marked by negative values, we simply skip the negative indices when accessing N'.

The second adaptation we need to make is to efficiently utilize the data cached on the GPU. STMatch assumes that the entire graph is stored on GPU, and it only accesses GPU global memory for neighbor lists. In our setting, most of the graph data is maintained on the CPU, and only a small number of selected vertices are cached on the GPU. To access data from the GPU cache whenever possible, we need to look up the vertex in the cache before every access. This can be done efficiently by performing a binary search on the *rowidx* array in Figure 6. If the target vertex is found in *rowidx*, we load its neighbors from the corresponding location in *colidx* on the GPU. If the target vertex is not found in *rowidx*, we obtain the starting address of the vertex's neighbor list on the CPU (from the *pDevice* array in Figure 5), and then load the data from the CPU memory by GPU zero-copy access.

Table 1. Data graphs.

Graph	# Vertices	# Edges	Max deg.	Size (GB)
Amazon (AZ)	0.4M	2.4M	1367	0.019
RoadNetPA (PA)	1.08M	1.5M	9	0.022
RoadNetCA (CA)	1.96M	2.7M	12	0.037
LiveJournal (LJ)	3.1M	77.1M	18311	0.308
Friendster (FR)	65.6M	3612M	5214	28.9
SF3K-fb (SF3K)	33.4M	5824M	4328	46.4
SF10K-fb (SF10K)	100.2M	18809M	4485	151.1

To ensure that the matching procedure accesses consistent data, the graph reorganization on CPU (Step-4 in Sec. 1.5.1) is conducted after the matching is completed on the GPU. Our experiments show that the overhead of graph reorganization is negligible compared to the matching task.

1.6 Experimental Results

This section provides an evaluation of GCSM by comparing it with three naive GPU implementations and two CPU systems.

1.6.1 Experimental Setup

Platform: Our experiments are conducted on a CPU-GPU system with two Intel Xeon Gold 6226R 2.9GHz CPUs (32 cores in total) and an Nvidia RTX3090 GPU. The GPU is connected to the CPUs through PCIe. The CPUs have 512GB RAM, and the GPU has 24GB global memory. The test platform runs a Ubuntu-20.04. Our system was developed in C++ and CUDA (which is the language provided by Nvidia to program its GPUs). All the CPU code was compiled using GCC 9.4.0 with O3 optimization, and the GPU code was compiled using NVCC 11.2.

Datasets and query graphs: Table 1 lists the data graphs used in our experiments. Among the data graphs, AZ, LJ, PA, CA, and FR are from the SNAP dataset [**?**], while SF3K and SF10K are random graphs generated by the LDBC graphalytics [**?**]. Following the existing research on streaming graphs [**?**, 11, 21], we generate dynamic graphs from static graphs. For FR, SF3K, and SF10K, we randomly select 12×8192 edges from each data graph to construct the edge updates.

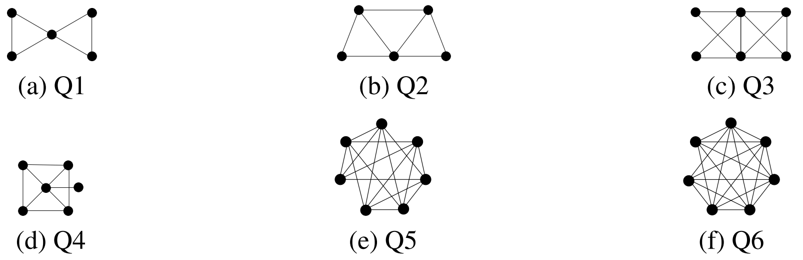


Figure 7. Query graphs.

For AZ, LJ, PA, and CA, we randomly select 10% of edges from the original graph, following the settings in [**?**, 21]. Each selected edge is marked as either insertion or deletion with equal probability. The edges marked for insertion are removed from the data graph. If all the incident edges of a vertex are removed, the vertex is also removed from the graph. We use six query graphs from size-5 to size-7, as shown in Fig. 7.

Baselines: We compare our system with four naive GPU implementations. The first implementation (`UM`) uses GPU unified memory. All the neighbor lists of the data graph are allocated as unified memory on the CPU and are directly accessed by the GPU kernel in pages during the matching procedure. The second GPU implementation (`ZP`) uses GPU zero-copy access. All the neighbor lists are allocated as pinned memory on the CPU and are mapped to the GPU address space. The GPU kernel can directly access the neighbor lists on the CPU in cache lines. The third GPU implementation (`VSGM`) employs the caching technique proposed in [9], which copies the neighbor lists of all *k*-hop neighbors of the updated edges onto the GPU before matching. During the matching, the GPU kernel only accesses GPU memory. The fourth GPU baseline (`Naive`) adopts a similar configuration to our system. It caches the neighbor lists of certain nodes on the GPU while keeping most of the graph data in CPU memory. However, it uses node degree as an estimate of access frequency. For a fair comparison, all the GPU versions use the same GPU kernel adapted from STMatch [26] for the matching task.

We also compare our system with two CPU baselines. The first baseline is RapidFlow (`RF`) [21], which is a state-of-the-art continuous subgraph matching system on the CPU featuring optimized matching order. To achieve the matching order optimization, RapidFlow uses an index data structure to store the candidate vertices for each pattern vertex. The index data structure consumes a lot of memory and causes system crashes for large graphs. Therefore, we compare our system with RapidFlow only using small graphs (AZ and LJ) for which RapidFlow does not crash on our platform. To show the benefit of GPU processing for large graphs, we implement a CPU system based on the nested loops in Fig. 2, which always start the matching process from the updated edges. For a fair comparison, our CPU code uses the same stack-based implementation [26] and the same matching order as our GPU code.

Settings: We execute the GPU code by launching 82 thread blocks, each containing 1024 threads on the GPU. The CPU code is run with 32 CPU threads. The original code of RapidFlow was single-threaded, so we parallelized its outermost loop that iterates over the candidate vertices for the first pattern vertex. Our CPU code is also parallelized at the outermost loop that iterates over the updated edges. For our GPU system, we set the number of random walks M to $|\Delta E| D^{n-2}/32^n$. Since the matching kernel (STMatch) uses about 10GB GPU memory, the maximum GPU buffer size is set to 14GB. Nodes with the highest estimated frequency are cached in the GPU buffer. In all our testcases, the neighbor lists of all nodes sampled by the random walk procedure take less than 2GB and fit in the GPU buffer. Since a node is sampled at least once, this means that nodes with an estimated frequency greater or equal to $|\Delta E|$ are cached on GPU.

1.6.2 Comparison With Naive GPU Implementations

Fig. 8 to Fig. 11 show the average execution time of different implementations for one batch of edge updates. For different data graphs and query patterns, our system is consistently faster than the naive zero-copy implementation (`ZP`), achieving 1.4 to 2.9x speedups, with an average of 1.81x. The speedups are due to the reduced data access to the neighbor lists on the CPU. As labeled in the figures, our caching mechanism reduces the CPU memory access by 1.3x to 6.7x times compared with naive zero-copy.

The overhead of estimating frequent vertices and copying their neighbor lists to GPU (i.e., Step-2 and Step-3 in Fig. 3) are included in the execution time of `GCSM` in Fig. 8 to Fig. 10. Details

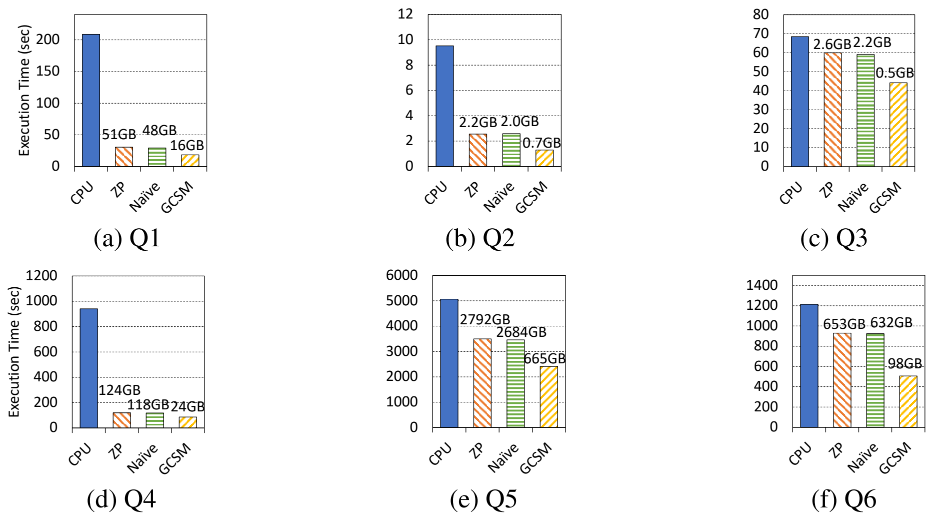


Figure 8. Execution time for matching different query patterns from a batch of 4096 edges on FR graph. Data access sizes from CPU are labeled on each bar.

Table 2. Overhead of frequency estimation (FE) and data copying (DC) in percentage of total execution time.

	FR		SF3K		SF10K	
	FE	DC	FE	DC	FE	DC
Q1	5.1%	1.6%	3.2%	2.1%	2.2%	1.7%
Q2	15.0%	3.6%	14.2%	6.5%	8.1%	4.3%
Q3	0.6%	0.1%	17.3%	5.3%	8.2%	12.7%
Q4	0.3%	0.1%	0.8%	0.4%	0.5%	0.6%
Q5	0.2%	0.1%	1.2%	0.5%	0.9%	0.6%
Q6	0.1%	0.1%	3.7%	0.8%	5.3%	1.7%

of the breakdown overhead are given in Table 2. We can see that the overhead of frequency estimation is small, accounting for less than 10% of the total execution time in most cases. The percentage decreases as the pattern size becomes larger. The overhead of copying the neighbor lists of frequent vertices to GPU is also small, accounting for less than 5% of total execution time in most cases.

The results also indicate that the naive caching policy based on node degree (`Naive`) is ineffective. Its performance is almost the same as zero-copy (which accesses all data from CPU memory). This is because the access of nodes during the matching procedure depends on the query pattern and updated edges. Having a high node degree does not necessarily result in more frequent

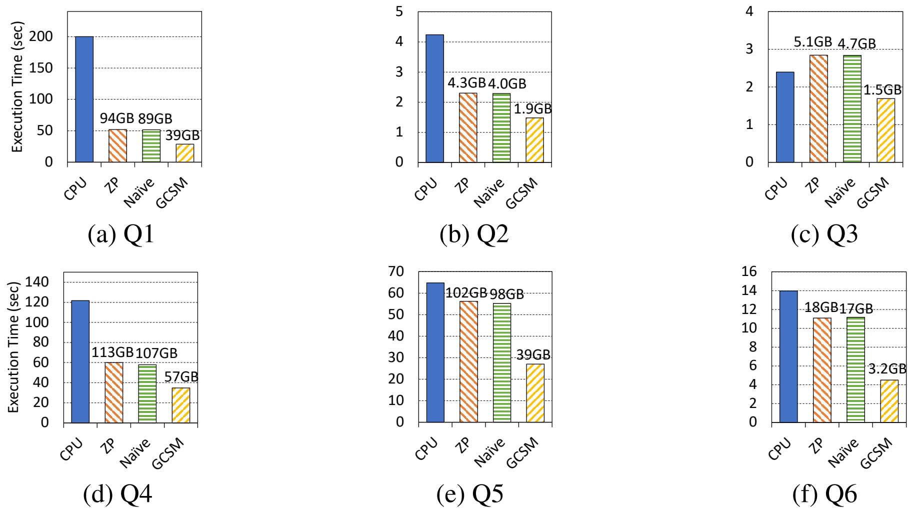


Figure 9. Execution time for matching different query patterns from a batch of 4096 edges on SF3K graph.

access.

Continuous subgraph matching exhibits data locality not only because most real-world graphs have a skewed degree distribution, but also because the matching is performed on small batches of updated edges. The performance results on RoadNetPA and RoadNetCA (where all nodes have a small degree) validate that our system is still efficient when the input graph is less skewed. As shown in Fig. 11, GCSM achieves 1.6x to 2.0x speedups against the zero-copy implementation and 1.6x to 2.1x speedups against the naive caching policy. Note that we tested the performance with all size-3, 4, and 5 motifs instead of specific patterns in this case because the listed patterns in Fig. 7 rarely exist in the road nets.

We also test the performance of naive unified memory implementation (`UM`). The execution time is not plotted in the figures because it is much longer than other versions, making the figures out of scale. For the test cases in Fig. 8 to Fig. 11, `UM` is 69x to 210x slower than the naive zero-copy.

To compare with `VSGM`, which copies all *k*-hop neighbors of the updated edges onto GPU, we have to use smaller batch sizes (128 for SF3K and 64 for SF10K) to fit the data into GPU memory. Figure 13 shows the breakdown of execution time for processing one batch of edge

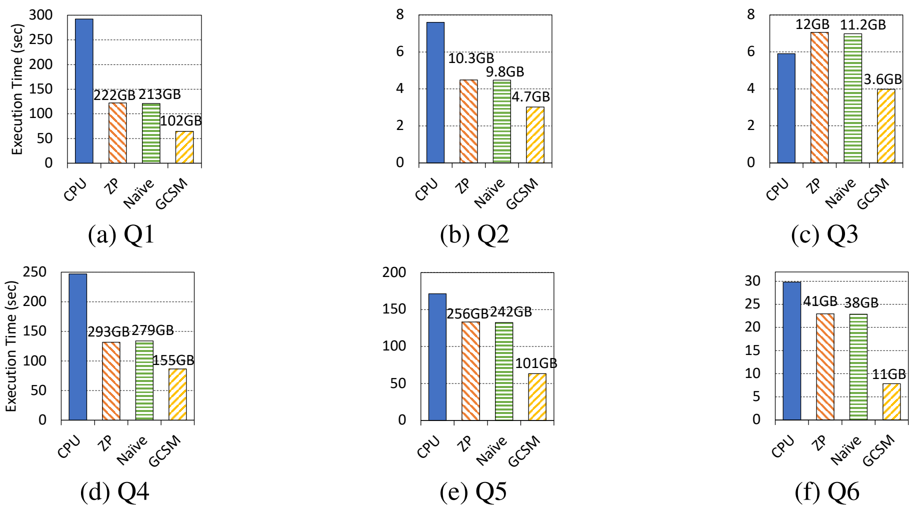


Figure 10. Execution time for matching different query patterns from a batch of 8192 edges on SF10K graph.

updates with both VSGM and GCSM. We can see that the subgraph matching kernel takes almost the same amount of time in GCSM as in VSGM, indicating that our system incurs little overhead of CPU data access. Although VSGM avoids accessing data from the CPU entirely, it requires copying much more data to the GPU before the matching process. This results in a much longer data copy time and thus much worse overall performance.

To show the efficiency of our system on different batch sizes, we test the performance with eight batch sizes from 8192 to 64 on SF3K and SF10K. As shown in Fig. 12, the execution time is almost proportional to the batch size. Our system achieves 1.8x to 2.9x speedups (with an average of 2.1x) against zero-copy, and 1.6x to 2.8x speedups (with an average of 1.9x) against the naive degree-based cache policy.

1.6.3 Comparison With CPU Implementations

The execution time of our CPU implementation is included in Fig. 8 to Fig. 10. The CPU implementation is slower than the naive zero-copy implementation on GPU (for most cases), validating the benefit of GPU processing for subgraph matching. Our system achieves speedups ranging from 1.4x to 11.4x, with an average of 4.1x, compared to the CPU implementation.

The RapidFlow system runs out of CPU memory when storing candidate vertices on the

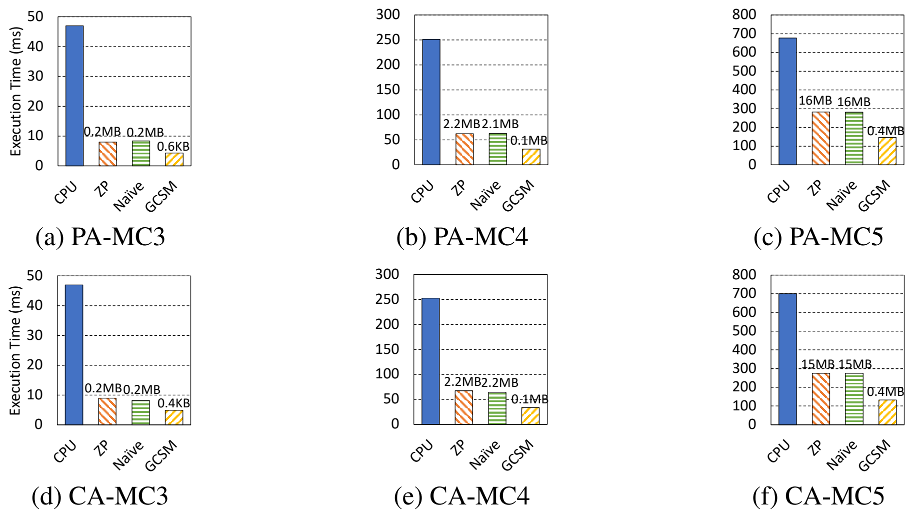


Figure 11. Execution time for counting size-3, 4, and 5 motifs on RoadNetPA and RoadNetCA. The batch size is set to 4096.

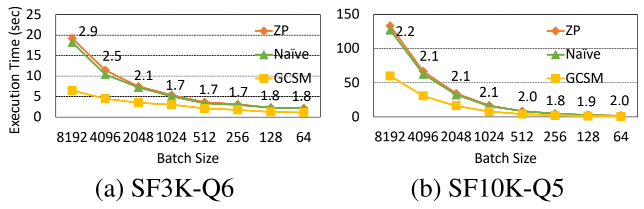


Figure 12. Execution time of different batch size for matching Q6 on SF3K graph and Q5 on SF10K graph. The speedups of GCSM against zero-copy are marked above the line.

three large graphs. Therefore, we compare the performance with RapidFlow using two smaller graphs (AZ and LJ). The results are shown in Fig.14. RapidFlow achieves comparable performance to our CPU implementation in most cases, validating the efficiency of our CPU baseline. However, due to its optimized matching order, RapidFlow can be up to 7.7x faster than our CPU implementation in some cases. This performance advantage of RapidFlow is at the cost of extra memory consumption for storing the candidate vertices for each pattern vertex. Nevertheless, our GPU system outperforms RapidFlow by 1.6x to 4.4x in all cases. It will be interesting to reduce the memory consumption of RapidFlow and incorporate its matching order optimization into our system. We will leave the problem for future work.

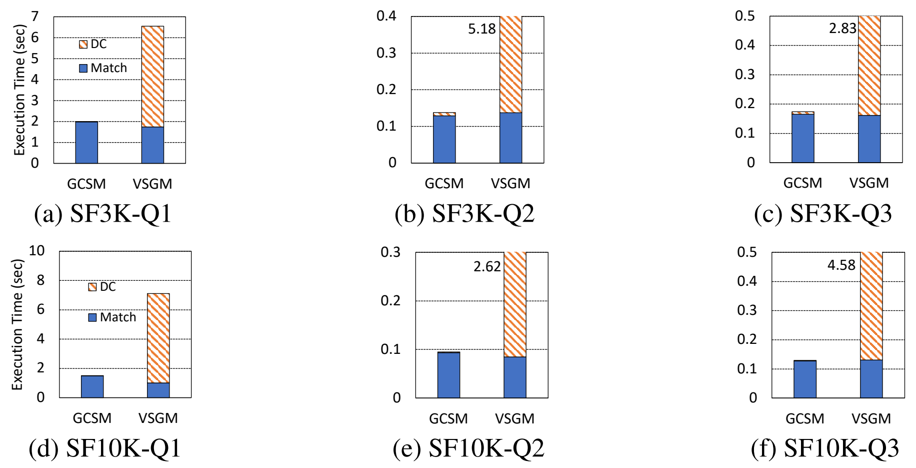


Figure 13. Breakdown execution time of VSGM and GCSM. ‘DC’ includes the time for identifying the caching vertices and copying their neighbor lists to GPU. ‘Match’ is the matching kernel execution time on GPU.

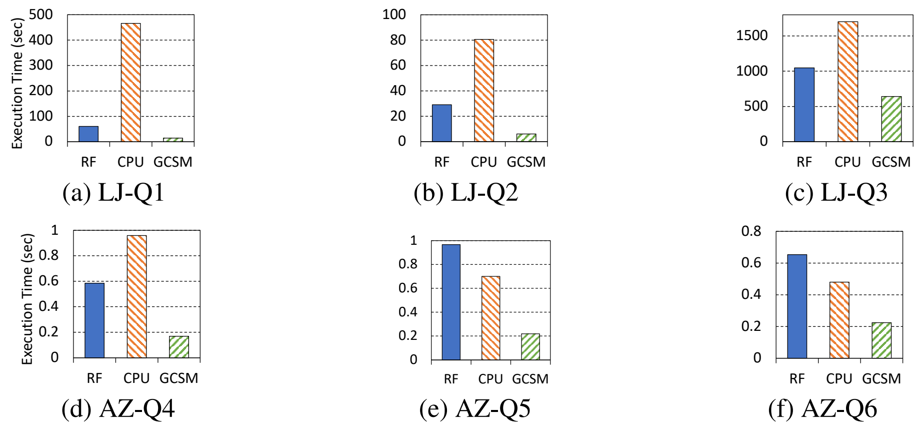
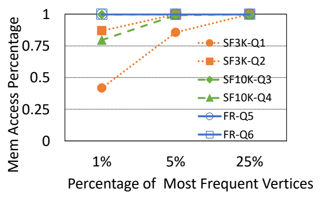


Figure 14. Performance comparison with RapidFlow.

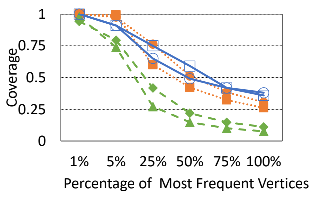
1.6.4 Effectiveness of Frequency Estimation

To show the effectiveness of our random walk technique in identifying frequent vertices, we calculate the coverage of accessed vertices within the GPU cache. Suppose S is the set of most frequently accessed vertices during the exact matching process and T is the set of vertices cached on the GPU. The coverage is calculated as $|S \cap T|/|S|$.

Figure 15b shows the coverage results for different test cases at varying percentages of the most frequent vertices. For SF3K, the coverages for the top 1% to top 5% frequent vertices are



(a) Memory Access Distribution



(b) Cache Coverage

Figure 15. Memory access distribution and cache coverage of incremental matching.

Table 3. Graph reorganization time (in milliseconds).

	$\lvert\Delta E\rvert = 4096$	$\lvert\Delta E\rvert = 8192$
AZ	1.5	3.4
PA	0.8	1.5
CA	0.9	1.5
LJ	3.1	8.4
FR	7.9	8.8
SF3K	5.8	6.4
SF10K	6.7	9.5

nearly 100%. For SF10K, we achieve coverage of more than 90% for the top 1% frequent vertices and coverage of about 75% for the top 5% frequent vertices. For FR graph, our method identifies all of the top 1% frequent vertices and more than 91% of the top 5% frequent vertices. The coverage decreases as percentage of vertices increases. However, since more than 80% memory access is made to the top-5% vertices (as shown in Fig. 15a), the cached data on GPU can save most of the accesses to CPU during the matching process.

1.6.5 Graph Reorganization Overhead

So far we have evaluated Step-1 to Step-4 of our system shown in Fig. 3. The last step of our system is to reorganize the graph data on CPU. It involves removing the deleted edges and sorting all the updated neighbor lists. While our system is not designed to achieve the best performance for edge insertion and deletion, We find the overhead of graph update is almost negligible compared to the matching process. As shown in Table 3, the average time of graph reorganization for a batch of edge updates is no more than a few milliseconds, which is much smaller than the matching time in Fig. 8 to Fig. 11. The graph reorganization overhead reduces as the batch size decreases, and it is always negligible compared to the matching time for batch sizes from 64 to 8192.

1.7 Related Works

Static subgraph matching: The study of static subgraph matching dates back to Ullmann [23], which proposes the first backtracking algorithm for the task. Following Ullmann's work, many studies [**?**, 5, 19] propose optimizations for the matching order to improve performance. They show that a carefully selected matching order can greatly reduce the search space of the algorithm. Automine [**?**] is a compiler system that converts the backtracking algorithm into nested loops and applies various loop optimization techniques to accelerate the matching process. Another line of research treats subgraph matching as multi-way join operations and improves the performance by optimizing the join plans [1,10,12]. With the development of GPUs, many efforts have been made to utilize the massive parallelism of GPU to accelerate static subgraph matching [**?**, 22, 26–28]. CuTS [27] aims to solve the large memory consumption issue by introducing a hybrid BFS-DFS execution strategy. However, it faces a trade-off between memory consumption and load balance. To address these problems, STMatch [26] proposes a stack-based data structure to store intermediate subgraphs and employ work-stealing to balance the workloads among computing units on GPU. PBE [**?**] and VSGM [9] are two works that focus on subgraph matching in distributed GPU systems, allowing for matching on extremely large graphs.

Continuous subgraph matching: IncIsoMatch [6] is the first work that supports continuous subgraph matching on a dynamic graph. Give an edge update, IncIsoMatch finds the region in the data graph affected by the update for a query and performs the matching again in the region. Graphflow [10] avoids the repeated matching for each graph update and computes the incremental matching results by performing multi-way join operations. SJ-Tree [4] uses binary joins with indexes to evaluate incremental matching results. It stores partial results to reduce computation costs, but this approach can lead to a memory explosion when processing large graphs (sj-tree). TurboFlux [11] proposes a data-centric graph structure that optimizes the storage of partial results to achieve a better balance between memory consumption and re-computation overhead. Based on TurboFlux [11], SymBi [**?**] proposes candidate vertices set pruning using query edges. Rapidflow [21] is a state-of-the-art continuous subgraph matching system on CPU, which features an optimal matching order selection and a dual matching technique to eliminate redundant computation caused by automorphisms.

Graph sampling: Sampling techniques have been used for graph pattern matching in previous works. Motivo uses graph coloring and adaptive sampling to accelerate motif counting [**?**]. ScaleMine [**?**] employs sampling to estimate the workload of different branches of the exploration process and uses the information to improve load balance. SampleMine [**?**] proposes a framework for applying random sampling to different graph mining tasks based on loop perforation. C-SAW [**?**] is a library for defining various random walk algorithms on GPU. All these works focus on static graphs. To the best of our knowledge, our work is the first to apply random sampling to continuous subgraph matching, aiming to improve data access efficiency for GPU processing.

1.8 Conclusion

In this work, we propose a system that exploits GPU to accelerate continuous subgraph matching. The main challenge is how to reduce data access from the CPU. We address the problem by identifying the most frequent vertices with a random-walk technique and caching the frequently accessed neighbor lists on GPU. Our experimental results show that our system achieves significant speedups against existing CPU solutions and supports CSM on extremely large graphs.

REFERENCES

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):1–44, 2017.

[2] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 739–748. IEEE, 2008.

[3] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1199–1214, 2016.

[4] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. A selectivity based approach to continuous pattern detection in streaming graphs. *arXiv preprint arXiv:1503.00849*, 2015.

[5] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159. Citeseer, 2001.

[6] Wenfei Fan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)*, 38(3):1–47, 2013.

[7] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 328–339, 1995.

[8] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of*

Data, SIGMOD '19, page 1429–1446, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Guanxian Jiang, Qihui Zhou, Tatiana Jin, Boyang Li, Yunjian Zhao, Yichao Li, and James Cheng. Vsgm: View-based gpu-accelerated subgraph matching on large graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

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

[11] 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.

[12] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *arXiv preprint arXiv:1903.02076*, 2019.

[13] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[14] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

[15] Steven Noel. A review of graph approaches to network security analytics. *From Database to Cyber Security*, pages 300–323, 2018.

[16] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[17] 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.

[18] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.

[19] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, August 2008.

[20] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Out-of-core gpu memory management for mapreduce-based large-scale graph processing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 221–229, 2014.

[21] 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.

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

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

[24] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*, 2014.

[25] 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.

[26] 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.

[27] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 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*, pages 1–14, 2021.

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