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 STMATCH: ACCELERATING GRAPH PATTERN MATCHING ON GPU WITH STACK-BASED LOOP OPTIMIZATIONS

1.1 Introduction

Graph pattern matching is widely used for retrieving information from graph-structured data in many application domains, including bioinformatics [58], social network analysis [81], and cybersecurity [62]. The problem stems from the well-known subgraph isomorphism problem, which aims to find all subgraphs that are isomorphic to a given query pattern, and it is the fundamental task for many related problems, such as motif counting and clique listing [54].

Due to its importance in real applications, graph pattern matching has been extensively studied in the past decades. Numerous algorithms and implementations have been proposed [9, 22, 28, 29, 45, 56, 73, 82, 97]. However, as the problem is NP-hard [31], it is still a performance bottleneck in many applications, and it is always desirable to scale the computation to large graphs. Therefore, there is a growing interest in exploiting the massive parallelism on GPU to accelerate the computation [80, 86, 92, 96].

Despite their different optimizations, the existing GPU graph pattern matching systems all take a subgraph-centric approach. They maintain a list of valid partial subgraphs and extend them by one vertex/edge in each step until the desired pattern size is reached. To extend a partial subgraph, they either add a new edge to it by performing a binary join operation [80, 86], or they find a match for the next pattern node by performing set operations on the neighbor lists of the previous nodes [92, 96]. A common feature of these systems is that they need to store the partially matched subgraphs explicitly. For example, GSI [96] stores the partial subgraphs in a table, and it assigns each partial subgraph to a warp on GPU for extension. A more recent work, cuTS [92], reduces the memory consumption by using a trie-based data structure to store the partial subgraphs. It also improves GPU thread utilization by assigning each partial subgraph to a virtual warp.

The subgraph-centric implementation facilitates parallelization; however, it has several inherent issues for GPU execution. First, it requires synchronization at the end of each extension step.

The current systems maintain a list of partial subgraphs and launch a GPU kernel to process them at every step. The kernel launch and the synchronization incur an overhead. Second, the partial subgraphs take a lot of memory space. Although a hybrid DFS and BFS extension order can alleviate the issue [92], it requires much more kernel launches and synchronizations. Third, the subgraph-centric implementation loses the implicit hierarchy of partial subgraphs and thus disables some optimizations that can be applied to the backtracking procedure (e.g., loop-invariant code motion and pattern merging [53]). As a result, the state-of-the-art GPU graph pattern matching system (cuTS [92]) can be even slower than a highly optimized CPU implementation (Dryadic [53]) in many cases (See Table 2).

To overcome the limitations of the subgraph-centric systems, we study the parallelization of backtracking on GPU in this work. We first show that the subgraph-centric approach taken by the existing systems corresponds to the *inner-loop* parallelization of the backtracking algorithm. Unlike the previous systems, we choose to parallelize the backtracking procedure from the outermost loop. This eliminates the synchronization on GPU and enables our system to finish the computation with one kernel launch. The main issue with this *outer-loop* parallelization is that it suffers from severe load imbalance. We address the issue by adopting a stack-based implementation of the backtracking algorithm and proposing a two-level work-stealing technique. We also observe that the intra-warp thread utilization is low when the data graph is sparse. This is because most nodes in these graphs have only a few neighbors and the set operations cannot occupy all the threads in a warp. To improve the thread utilization, we propose a loop-unrolling technique that combines the set operations for multiple sets and assigns them to a single warp. Finally, as pointed out in [53], the backtracking procedure involves a lot of redundant set operations, which can be eliminated by loop-invariant code motion. We adapt the code motion technique in [53] and show that our system can be easily and efficiently extended with this optimization.

In summary, we make the following contributions:

1. We propose the first stack-based graph pattern matching system on GPU, which avoids the synchronization and the memory consumption issue of previous subgraph-centric systems.

2. We propose a two-level work-stealing and a loop unrolling technique to improve the inter-warp and intra-warp GPU resource utilization for our system.

3. We implement a code-motion technique to reduce redundant computation in our system and showcase that our system is compatible with the existing optimizations for backtracking-based graph pattern matching.

We perform an extensive evaluation of our system using various query patterns and input graphs, and compare with three state-of-the-art graph pattern matching systems: cuTS [92], GSI [96] and Dryadic [53]. The experiments show that our system achieves 24x to 3385x speedups against cuTS and GSI on an Nvidia GeForce RTX 3090 GPU and up to 898x speedups against Dryadic.

1.2 Preliminaries

This section gives a formal definition of the graph pattern matching problem and describes the backtracking algorithm for solving the problem. We also provide a brief background on GPU architecture 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 the vertices and edges. A graph $G' = (V', E', L')$ is a *subgraph* of graph $G = (V, E, L)$ if $V' \subseteq V$, $E' \subseteq E$ and $L'(v) = L(v), \forall v \in V'$. A subgraph $G' = (V', E', L')$ is *vertex-induced* if all the edges in E that connect the vertices in V' are included E'. A subgraph is *edge-induced* if it is connected and is not vertex-induced.

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 $(f(v_i), f(v_j)) \in E_b$.

The graph pattern matching problem is defined as finding all the subgraphs in G that are isomorphic to a given query graph Q. The subgraphs to be found can be either vertex-induced

Algorithm 1: Backtracking for graph pattern matching

```
   Input: a data graph G and a query graph Q
   Output: all subgraphs in G that are isomorphic to Q
1  π ← generate a matching order;
2  Enumerate(G, Q, π, {}, 0);
3  Procedure Enumerate(G, Q, π, m, l):
4      if l = Q.size then Output m, return;
5      u ← π[l];
6      C_m(u) ← getCandidates(G, Q, π, m, l);
7      foreach v ∈ C_m(u) do
8          Add v to m;
9          Enumerate(G, Q, π, m, l + 1);
10         Remove v from m;
```

or edge-induced. If the subgraphs are edge-induced, the problem is equivalent to the subgraph isomorphism problem [82].

1.2.2 Backtracking for Graph Pattern Matching

Algorithm 1 shows the backtracking algorithm that is commonly used for graph pattern matching. The algorithm first generates a matching order π for the nodes in the query graph (line 1). The matching order ensures that the node to be matched in the next step (i.e., $\pi[l+1]$) is connected to at least one of the nodes matched in previous steps (i.e., $\pi[0], \ldots, \pi[l]$). Prior work has shown that a carefully selected matching order can effectively prune the exploration space and reduce the computation [22, 56, 73, 97]. After obtaining the matching order, the algorithm invokes a recursive procedure to enumerate the subgraph isomorphisms (line 2). Starting from an empty subgraph, the `Enumerate` procedure gradually grows the subgraph until it reaches the size of the query graph (line 4). At each step l, it computes a set of nodes in G that match the lth node of the query graph (line 6). Then, for each node v in the candidate set, it adds the node to the partially matched subgraph m (line 8) and call the `Enumerate` procedure to match the next node in the pattern (line 9). Once it returns from the recursive call, which means all the subgraphs extended from m have been explored, the procedure remove v from m and backtracks (line 10).

Although the above backtracking algorithm can be directly translated into a recursive function, many graph pattern matching/mining systems implement it as a nested loop because it is more

```
    // u₀ can be mapped to any node in data graph
1:  C₀ = V;
2:  for (i₀ = 0; i₀ < C₀.size; i₀++) {
3:    v₀ = C₀[i₀];
      // u₁ is a neighbor of u₀
4:    C₁ = N(v₀);
5:    for (i₁ = 0; i₁ < C₁.size; i₁++) {
6:      v₁ = C₁[i₁];
        // u₂ is a neighbor of u₀ but not a neighbor of u₁
7:      C₂ = N(v₀) − N(v₁);
8:      for (i₂ = 0; i₂ < C₂.size; i₂++) {
9:        v₂ = C₂[i₂];
          // u₃ is a neighbor of u₀, u₁ and u₂
10:       C₃ = N(v₀) ∩ N(v₁) ∩ N(v₂);
11:       for (i₃ = 0; i₃ < C₃.size; i₃++) {
12:         v₃ = C₃[i₃];
13:         Output({ v₀, v₁, v₂, v₃; } } } }
```

Figure 1. Graph pattern matching implemented as a nested loop. $N(v)$ means the neighbors of node v in the data graph.

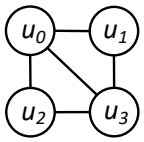


Figure 2. A query graph.

convenient for parallelization and optimization [6, 26, 53, 54, 74, 76]. As an example, Fig. 1 shows the nested loop for matching the query of Fig. 2.

The candidate nodes for the first loop are the nodes in the data graph that can be mapped to u_0. The first loop iterates over all these nodes and tries to extend each node with its neighbors (line 4). The second loop iterates over the candidate nodes for u_1 and tries to further extend the subgraph. Because u_2 is a neighbor of u_0 but not a neighbor of u_1 in the query graph, the candidate nodes for u_2 must be in the neighbor list of v_0 but not in the neighbor list of v_1. Thus, we compute the candidate nodes for u_2 as $N(v_0) - N(v_1)$. Similarly, the third loop computes the candidate nodes for u_3 as $N(v_0) \cap N(v_1) \cap N(v_2)$ since u_3 is neighboring to u_0, u_1 and u_2.

1.2.3 GPU Architecture

There are two types of parallelism on GPU: SIMT (Single Instruction Multiple Threads) and MIMD (Multiple Instruction Multiple Data). The GPU threads are organized into *warps*, and the threads within a warp execute the same instruction simultaneously. For branch statements, if different threads in a warp need to execute different branches, they have to execute one by one. The situation is called *thread divergence*, and it hurts the performance because only a portion of the threads in a warp can be active at a time. The warp is the smallest scheduling unit on GPU. Different warps can execute different instructions on different data. The warps are further organized into *threadblocks* (also called cooperative thread arrays) and are launched together onto the streaming multiprocessors.

A GPU typically has tens of streaming multiprocessors. Each streaming multiprocessor has a number of registers and a programmable cache called *shared memory*. Although it is possible to declare more register variables than the physical registers in a CUDA program, the variables will be spilled to constant memory, which may significantly slow down the program. All the threads in a threadblock can access the shared memory. The shared memory is much faster than the GPU global memory but also much smaller. The typical size of shared memory on a streaming multiprocessor is tens of KB. When a threadblock uses more shared memory than what is available on a streaming multiprocessor, it cannot be launched. A modern GPU can run more than 1K threads simultaneously on a streaming processing, but the shared memory usually puts a limit on the number of threads that can be active.

1.3 Challenges for Parallelizing Backtracking

We now consider the parallelization of the nested loop in Fig. 1 for GPU execution.

Challenge 1: Load imbalance among warps. As real-world graphs are irregular, the workload associated with each node in the matching procedure varies significantly. If we parallelize the loop at the outermost level, the program will suffer from severe load imbalance. Previous CPU systems have proposed to combine the first two loop levels and distribute the computation based

on edges [53, 54]. While they work well for queries of up to four nodes, we find the load balance degrades dramatically for queries of more than five nodes. Previous work has also adopted work-stealing to balance the workload in distributed systems [8, 53, 74]. However, their work-stealing strategy cannot be directly applied to GPU due to the memory hierarchy and the lack of signaling mechanism on GPU. Another solution is to parallelize the inner loops. The subgraph-centric approach taken by the existing GPU graph pattern matching systems actually falls into this category. They materialize the intermediate results of each loop level (i.e., the partial subgraphs) and distribute the partial subgraphs to different warps. The downside of this approach is that it requires a synchronization at the end of each extension step and the materialization of intermediate results consumes a lot of memory.

Challenge 2: Thread underutilization within a warp. Since the threads within a warp execute in a SIMD manner, it is natural to assign each subgraph to a warp and use 32 threads to perform the set operation (at line 7 and 10 in Fig. 1). The problem is that, because the number of elements in each set is upper bounded by the degree of nodes in the data graph, the sets usually have less than 32 nodes. As shown in Table 17, the median degrees of most real-world graphs are much smaller than 32. This leads to idle threads during execution. While the problem can be easily solved by assigning multiple subgraphs to a warp in a subgraph-centric system [92], it is nontrivial if we want to parallel the loop from the outermost level and avoid the explicit storage of partial subgraphs.

Challenge 3: Redundant computation. As pointed out in [53], the nested loop conducts a lot of redundant set operations in its original form. For example, the result of $N(v_0) \cap N(v_1)$ at line 10 of Fig. 1 is the same for all iterations of the loop at line 8 because the computation is independent of i_2. We can lift this set operation outside the loop at line 8 to eliminate the redundant computation. While this code motion technique is easy to implement on CPU, it require a more careful design of data structures for storing the code motion information on GPU. In particular, the code-motion technique in [53] needs to store multiple intermediate sets for different labels in each loop level. If naively applied to our system, it may cause shared memory overflow for large labeled queries.

```
   // C[i] stores candidate nodes in loop level i
1: C = array(PAT_SIZE, MAX_DEGREE);
   // Csize[i] is number of nodes in C[i]
2: Csize = array(PAT_SIZE);
   // iter[i] is loop iterate in level i
3: iter = array(PAT_SIZE);
4: l = 0;   // start from loop level 0
5: while (true) {
6:   if (l < Q.size) {
       // if this subgraph has not been extended
7:     if (Csize[l] == 0) {
         // extend it
8:       getCandidates(G, Q, l, C, Csize);
         // algorithm stops if no subgraph can be extended
9:       if (l == 0 && Csize[l] == 0) break;
10:      iter[l] = 0; }
       // if there are unexplored nodes
11:    if (iter[l] < Csize[l]) { l++; } // go to next level
12:    else { // if all candidates are explored
         // empty the candidate set
13:      Csize[l] = 0;
         // and backtrack to previous level
14:      if (l > 0) { l--; iter[l]++; } }
15:  } else { // output subgraph at last level
16:    Output(C, iter); l--; } }
```

Figure 3. Graph pattern matching implemented as a stack-based while loop.

1.4 Overview of Stmatch

Our system is built around a stack-based implementation of Algorithm 1. The idea is to simulate the recursive procedure by explicitly maintaining a function call stack. As shown in Fig. 3, the call stack is composed of three arrays *C*, *Csize* and *iter*, which store the candidate nodes, the number of candidate nodes, and the loop iterate for each recursion level. A variable *l* is used to indicate the current recursion level and is initialized to 0. Every time the execution enters the next level, we `getCandidates` based on the data graph *G*, the query graph *Q*, and the current level *l* (line 8 in Fig. 3 corresponding to line 6 in Algorithm 1). Then, we iterate over the candidates, match each candidate node to the query node, and go to the next level (line 11 in Fig. 3 corresponding to line 9 in Algorithm 1). The matching subgraphs are output at the last

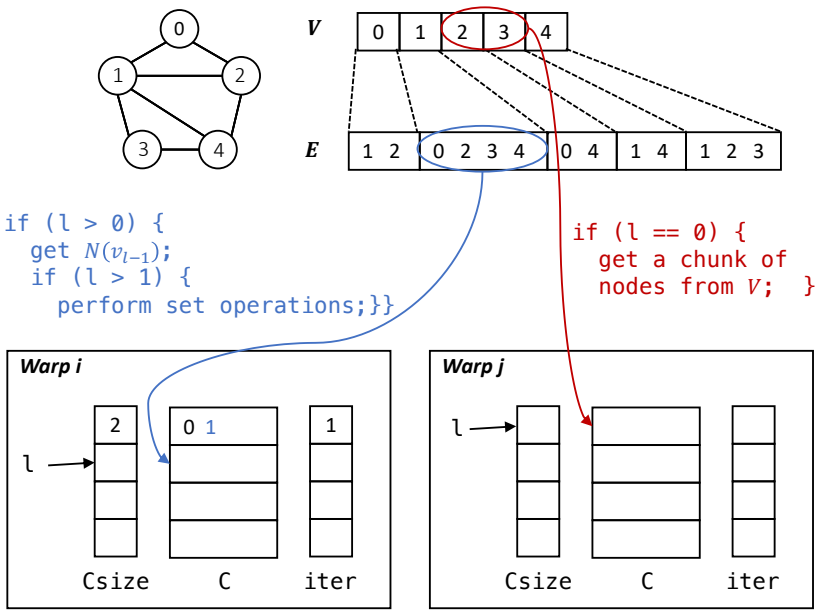


Figure 4. An example of *getCandidates* in two different warps.

level (line 16 in Fig. 3 corresponds to line 4 in Algorithm 1). If all candidate nodes at level l have been processed, we backtrack to the previous level (line 14 in Fig. 3 corresponding to return from *Enumerate* function at line 9 in Algorithm 1). The algorithm stops when all candidate nodes at level zero have been processed (line 9 in Fig. 3).

For GPU execution, we run the while-loop of Fig. 3 independently on different warps. A call stack is allocated for each warp. Since *Csize*, *iter* and l are small, we allocate them in shared memory. C is allocated in global memory. Different warps execute the same piece of code, but they obtain different nodes when $l = 0$ with the *getCandidates* function. Fig. 4 illustrates the procedure of *getCandidates* on two different warps. Suppose warp-i has two nodes (node-0,1) at the first level and is current processing node-1. Warp-j has just started its execution, and its stack is empty. The *getCandidates* function obtains the next chunk of nodes from V (node-2,3) and copies them to $C[0]$ on warp-j's stack. This corresponds to dividing the outermost loop of Fig. 1 and assigning different chunks of iterations to different warps for parallel processing. On warp-i, when the execution enters the second level, the *getCandidates* function obtains the neighbor list of node-1 and copies it $C[1]$. This procedure is conducted in parallel with different threads in the warp copying different elements in the neighbor list. When $l > 1$, the *getCandidates* function performs

Algorithm 2: Selecting a target warp within a threadblock

Input: stacks of all warps in the threadblock: $stks$; index of the calling warp: cur_idx; number of warps in the threadblock: NW; query graph: Q

Output: index of the target warp: $target\_idx$; the first level in the target that can be split: $target\_level$

```
1  target_idx ← −1;
2  target_level ← −1;
3  target_left_task ← 0;
4  for l ← 0 to StopLevel do
5      for idx ← 0 to NW − 1 do
6          if idx = cur_idx then continue;
7          left_task ← stks[idx].Csize[l] − stks[idx].iter[l] − 1;
8          if left_task > target_left_task then
9              target_idx ← idx;
10             target_level ← l;
11             target_left_task ← left_task;
12     if target_idx ≠ −1 then
13         return target_idx, target_level
14 return −1, −1;
```

set operations according to the query pattern. Different threads in the warp are assigned different nodes in the neighbor list and perform binary search simultaneously for set intersection/difference.

1.5 Load Balancing With Two-level Work Stealing

To balance the workload among warps, we propose a two-level work stealing technique. The idea is to let an idle warp steal work from other warps in the same threadblock first, and only when there is no warp to steal within the threadblock, we let it steal from other threadblocks. We use this two-level stealing mechanism for two reasons. First, as there are thousands of warps on GPU, selecting the best target to steal from all warps is expensive, whereas finding a good target within the threadblock is much easier. Second, migrating work within a threadblock is cheaper than across threadblocks. Since the stack of each warp is stored in the shared memory, work stealing within a threadblock can be done efficient in shared memory, whereas stealing across threadblocks has to go through global memory. This section gives a detailed description of our two-level work stealing technique.

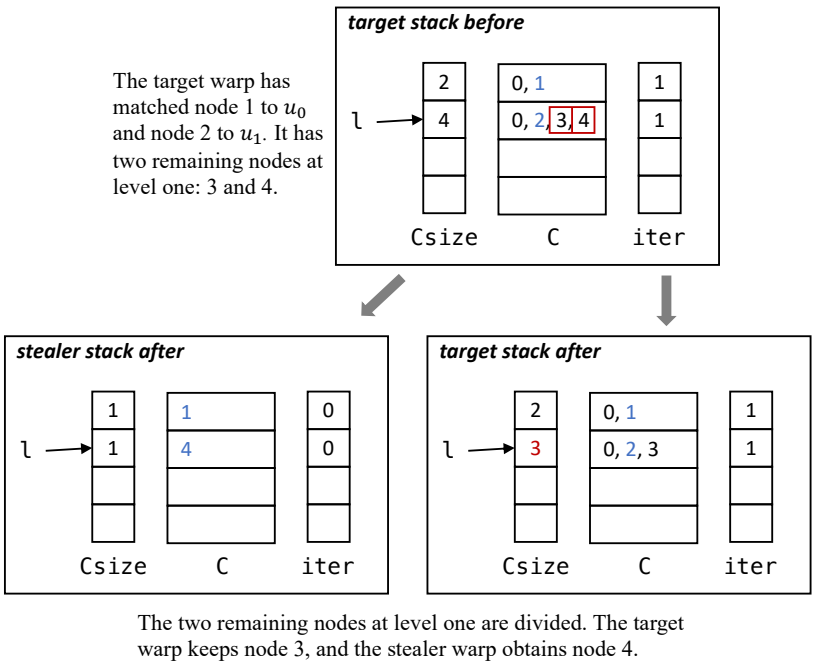


Figure 5. An example of dividing and copying tasks from a target warp.

1.5.1 Stealing Within a Threadblock

The work stealing procedure is inserted at line 9 of Fig. 3. Before a warp breaks out of the loop, it checks the stacks of other warps in the same threadblock and selects the one with the most remaining work. The selection procedure is shown in Algorithm 2. Starting from level zero, we check the stack level-by-level (line 4). Since the actual remaining work on a warp is unknown, we estimate it as the number of unexplored nodes at each level on the stack (line 7), and we assume that a warp with more unexplored nodes at a smaller level has more remaining work. Once we find a warp with at least one unexplored node at level l (line 8), we store its index to $target\_idx$ and the number of unexplored nodes to $target\_left\_task$. The procedure scans all the warps in the threadblock (line 5). If it later finds another warp that has more unexplored nodes at the same level, it updates the target warp (line 8-11). The selection procedure returns as soon as it finds a target at a certain level (line 12-13). If it cannot find a target after checking all levels, the procedure returns -1 (line 14) .

After the idle warp finds a target warp, it divides the remaining tasks in the target warp and

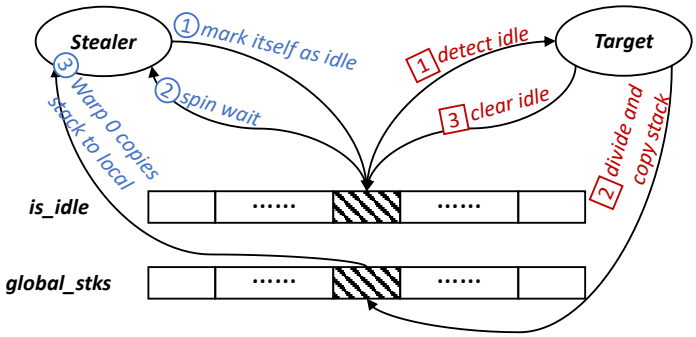


Figure 6. Work stealing across threadblocks.

copies half of them to its own stack. The procedure is illustrated in Fig. 5. Suppose we want to match the query graph of Fig. 2 to the data graph in Fig. 4. As shown in Fig. 5, the target warp has node 0 and 1 at level zero and has matched node 1 to u_0; it has no work left at level zero. The four neighbors of node 1 (i.e., node 0,2,3,4) are stored in the second level of the stack, and they are the candidates for u_1. Suppose the warp has processed node 0 at level one and is processing node 2; it has two remaining nodes: 3 and 4. We split the two nodes. Node 3 is kept in the target warp, and node 4 is migrated to the stealer warp. $Csize[1]$ of the target warp is changed to 3 since one node in $C[1]$ has been migrated to the stealer. The stealer copies the matching nodes from the target to its own stack from level zero to $target\_level - 1$. The $Csize$ are all set 1 and the $iter$ are all set to zero for these levels. In this example, $target\_level$ is one, so we copy the matching node at level zero of the target (which is node 1) and set $Csize[0]$ of the stealer to 1. For $target\_level$, we copy the stolen nodes (node 4 in this case) from the target to the stealer and set $C[target\_level]$ to be the number of stolen nodes. This completes the setup of both the target and the stealer stack.

The most expensive part of the stealing procedure is the copying of candidate nodes stored in global memory. Because of the overhead, we want to avoid stealing when there is not enough work left in the target warp. This can be achieved by adjusting the $StopLevel$ at line 4 of Algorithm 2.

1.5.2 Stealing Across Threadblocks

If a warp cannot find a target to steal in the same threadblock, it goes to other threadblocks. However, this cannot be done in the same way as stealing within a threadblock. Since the stack

of each warp is allocated in shared memory, a warp does not have direct access to warps in a different threadblock. We cannot let a warp check the stacks of warps in a different threadblock and pull tasks from a target. Instead, we have to let the target warp detect an idle warp and push tasks to it. The procedure is illustrated in Fig. 6. We maintain two arrays of *NB* elements in global memory where *NB* is the total number of threadblocks. Each element in the is_idle array is a bitmap indicating idle status of warps in the threadblock. Each slot of the $global\_stks$ array stores the tasks that the stealer receives from a target. When a warp fails to steal from warps in the same threadblock, ① it marks itself as idle in the is_idle array and ② spins wait on the idle status. During the matching process, a warp checks the status of other warps periodically. This is achieved by adding a `steal_across_block` function between line 6 and line 7 in Fig. 3. Every time a warp enters a level, it checks if it has unexplored nodes in the previous levels. To make sure the workload is large enough to justify the stealing overhead, we call this function only when the level is smaller than *DetectLevel* (which is a configurable parameter in our system). If the warp has unexplored nodes in the previous levels, 1 it scans the is_idle array to see if there is a threadblock where all the warps are marked idle. If it finds an idle threadblock, 2 it divides and copies its tasks to the $global\_stk$ of that threadblock. The divide-and-copy procedure is the same as shown in Fig. 5. After it finishes copying the stack to global memory, 3 the target warp clears the idle mask for all warps in the stealer threadblock. ③ Then, all warps in the stealer threadblock are activated, and warp zero will get the tasks and copy them to its local stack. The other warps will go back to the beginning of the while-loop. Since these warps do not have any remaining tasks, they will enter the stealing procedure again quickly, and they will try to steal from warp zero. With this, we prioritize stealing within a threadblock and avoid global stealing as much as possible. The program will eventually stop when all the threadblocks are idle.

1.6 Improving Thread Utilization With Loop Unrolling

With the original loop of Fig. 3, a warp performs one set operation at a time. If the sets have only a few elements, most of the threads will be idle. To improve thread utilization, we can unroll

```
   // store candidate nodes for every unrolled iteration
1: C = array(PAT_SIZE, UNROLL, MAX_DEGREE);
2: Csize = array(PAT_SIZE, UNROLL);
3: iter = array(PAT_SIZE);
   // iterate for unrolled iterations
4: uiter = array(PAT_SIZE);
5: l = 0;
6: while (true) {
7:   if (l < Q.size) {
       // if in the first of the unrolled iterations
       // and if candidate set is empty
8:     if (uiter[l] == 0 && Csize[l][0] == 0) {
         // extend subgraphs for all unrolled iterations
9:       getCandidates(G, Q, l, C, Csize, UNROLL);
         // if no subgraph can be extended
10:      if (l == 0 && Csize[0][0] == 0) {
           // try to steal from other warps
11:        if (!local_steal()) {
12:          if (!global_steal()) { break; }}}
13:      iter[l] = 0; uiter[l] = 0; }
       // if there are more unrolled iterations
14:    if (uiter[l] < UNROLL) {
         // and if there are unexplored nodes in current
         // unrolled iteration, go to next level
15:      if (iter[l] < Csize[l][uiter[l]]) { l++; }
16:      else {
           // if all candidates are explored in current
           // unrolled iteration, go to next unrolled iteration
17:        Csize[l][uiter[l]] = 0;
18:        iter[l] = 0;
19:        uiter[l]++; } }
20:    else {
         // if all unrolled iterations have been executed
         // reset unroll iterate
21:      uiter[l] = 0;
         // and backtrack to previous level
22:      if (l > 0) { l--; iter[l] += UNROLL; } } }
23:  else {
24:    for (i = 0; i < UNROLL; i++) Output(C, i, iter);
25:    l--; } }
```

Figure 7. An unrolled version of the loop in Fig. 3.

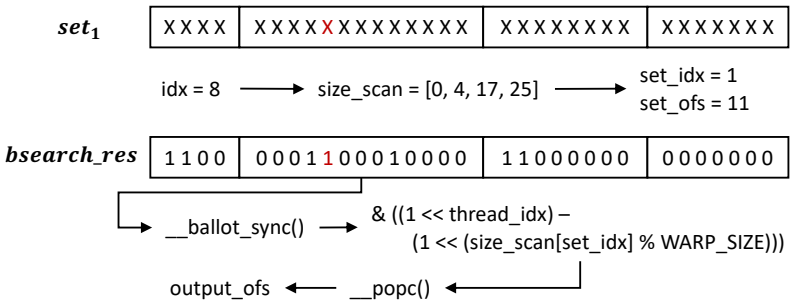


Figure 8. Perform multiple set operations in one warp.

the loop iteration at each recursion level and perform the set operations of the unrolled iterations together.

Fig. 7 shows the unrolled while-loop. The idea is to add an unroll dimension to C and $Csize$ so that the candidate nodes of multiple iterations can be stored at the same time. In addition to $iter$ that stores the iterate number of the original loop, we use an $uiter$ to store the index of unrolled iterations. The program iterates over the unrolled iterations and the original loop iterations alternatively. Every time the execution enters the next level, we compute the candidate nodes for all the unrolled iterations together (line 9). Then, we iterate over the candidates in each of the unrolled iterations, match each candidate node to the query node, and go to the next level (line 15). If all the unrolled iterations at level l have been processed, we backtrack to the previous level and increment the iterate of the previous level by the unroll size (line 22).

With the unrolled loop, we can combine multiple set operations and process them using one warp. Fig. 8 shows the implementation of combined set operation. We consider the general case where M set_1's need to intersect (or difference) with M set_2's. Each thread in the warp gets one element from set_1's at a time. We first compute a prefix sum of the set sizes ($size\_scan$) and use it to get the set index (set_idx) and the offset in the set (set_ofs) for that element. Then, we obtain the value of that element from $C[l][set\_idx][set\_ofs]$. The value and the corresponding $set2[set\_idx]$ are given to a binary search procedure, which produces a result of 0 or 1. For intersection operation, 1 means the value is found in $set2[set\_idx]$; for difference operation, 1 means the value is not found in $set2[set\_idx]$. Next, with the $bsearch\_res$, we compute the output offset for each element that needs to be written to the output. This corresponds to counting the number of 1's prior to that element

in the same set, and it can be efficiently implemented with the `__ballot_sync()` and `__popc()` primitive provided by CUDA. Finally, these elements are written to the result sets consecutively based on set_idx and $output\_ofs$. It is obvious that this combined set operation has a higher thread utilization than computing them one-by-one.

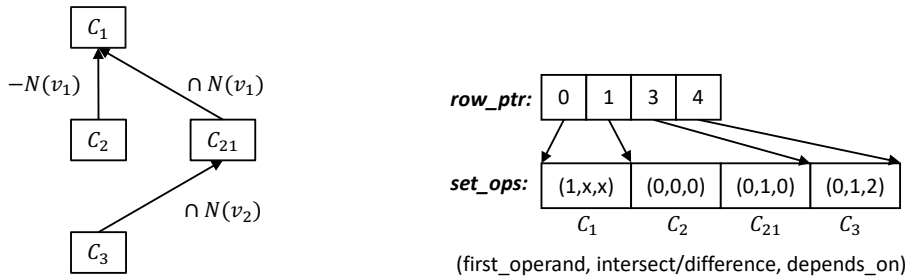
The divide-and-copy procedure in Fig. 5 needs to be slightly modified to enable working stealing for the unrolled loop. We use the same procedure for dividing and copying the tasks in the current unrolled iteration. But for the remaining unrolled iterations, we need to set the *Csize* to zero in the stealer stack since the tasks in these iterations are not stolen from the target. We also need to copy $uiter$ from level zero to $target\_level$.

1.7 Reducing Redundancy With Loop-invariant Code Motion

To show that our system is compatible with the existing optimizations for backtracking-based graph pattern matching, we implement the code motion technique proposed in [53] in our system. The idea is to lift the loop-invariant part of the set operations to upper levels so that they will not be computed repeatedly. For example, the $N(v_0) \cap N(v_1)$ operation at line 10 of Fig. 1 can be moved outside of the loop at line 8. We can store the result of $N(v_0) \cap N(v_1)$ and use the cached result for every iteration of the inner loop.

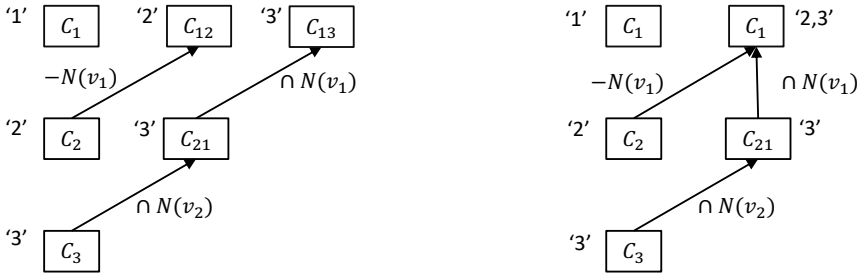
While it is straightforward to apply code motion to the nested loop in Fig. 1, it is nontrivial to incorporate this optimization into the existing subgraph-centric systems on GPU. In these systems, because the computation is driven by the subgraphs, the set operation is associated with each individual subgraph and the hierarchy of the set operations is lost. It is not obvious how to identify the loop-invariant operations and lift them for a batch of subgraphs. Although the hierarchy of set operations can be recovered from the subgraphs, maintaining a data structure to store the information is expensive.

Since our stack-based implementation is a direct simulation of the original nested loop, our system can be easily extended to support code motion. To perform the lifted set operations, we need to maintain more than one sets for each level in the stack. Therefore, we change the first



(a) Set dependence graph (b) A compact storage

Figure 9. The set dependence graph for unlabeled query of Fig. 2.



(a) Separate label sets (b) Merged label sets

Figure 10. The set dependence graph for labeled query of Fig. 2. '' denotes the label(s) of nodes in a set.

dimension of *C* and *Csize* from *PAT_SIZE* to the total number of sets of all levels. We also need to change the set operations in the `getCandidates` function to compute and use the results of the lifted operations. As an example, Fig. 9a shows the sets in the loop of Fig. 1 after code motion. In addition to the candidate sets ($C_0 \sim C_4$), we store an *intermediate set* C_{21} for $N(v_0) \cap N(v_1)$. The arrows indicate the dependence among the sets: C_1 is used for computing C_2 and C_{21}, and C_{21} is used for computing C_3. For any query graph, we can obtain such a dependence graph with a simple code motion analysis [53].

To pass set dependence information to `getCandidates` function efficiently, we propose a compact storage of the dependence graph as shown in Fig. 9b. The *row_ptr* array indicates the starting position of the sets in each level, and the *set_ops* array stores the set operations for each set. The set operation is represented with three numbers. The first number indicates whether the set operation has $N(v_{l-1})$ as the first operand at level l. In this example, since $C_1 = N(v_0)$, the first number in the first element of *set_ops* is 1. If the first number is 0, the set operation has

$N(v_{l-1})$ as the second operand. The second number in *set_op* indicates whether the set operation is intersection or difference. For C_2, we need to compute $C_1 - N(v_1)$, so we have 0 as the second number. For C_{21}, we need to compute $C_1 \cap N(v_1)$, so we have 1 as the second number. The third number is the index of the set that the current set depends on. The two arrays take only tens of bytes and are stored in shared memory. The `getCandidates` function reads in the two arrays and performs set operations accordingly at each level.

One limitation of the original code motion technique in [53] is that it needs to store multiple intermediate sets for different labels. To see this point, let us consider again the matching of query graph of Fig. 2. Suppose we restrict u_1 to label 'a', u_2 to label 'b', and u_3 to label 'c'. The dependence graph in Fig. 9a will not work – if C_1 only stores nodes of label 'a', we cannot have nodes of label 'b' in C_2. To fix this problem, [53] separates the candidate sets from the intermediate sets and stores nodes of different labels in different sets, as shown in Fig. 10a. If a set of label 'x' is dependent on a set in the previous level, they add an intermediate set of label 'x'. The labels are propagated from the bottom to the top level. The total number of sets is at least $n(n-1)/2$ where n is the number of nodes in the query graph. While this is not a problem on CPU, it may cause shared memory overflow in our system as we store *Csize* for all the sets of all unrolled iterations in shared memory. To reduce the usage of shared memory, we merge the intermediate sets of different labels that are split from the same unlabeled set into one set with multiple labels. For example, C_{12} and C_{13} in Fig. 10a can be merged into one set with label '2,3', as shown in Fig. 10b. The larger the query graph, the more sets we save with this method. This enables us to support larger query graphs without affecting the efficiency.

For work stealing, the candidate sets are divided and copied in the same way as in Fig. 5. We also copy all the intermediate sets that are used by sets after *target_level* so that they need not be computed again by the stealer.

Graph	# nodes	# edges	Max deg.	Med deg.	Deg. $>$ 4096
WikiVote	7K	100K	1065	3	0
Enron	37K	183K	1383	3	0
MiCo	96K	1.1M	1359	18	0
YouTube	1.1M	3.0M	28754	1	0.06%
LiveJournal	4M	34.7M	14815	6	0.12%
Orkut	3.1M	117.2M	33313	45	1.13%
Friendster	65.6M	1.8B	5214	9	9.1e-8%

Table 1. Graph datasets.

1.8 Experimental Results

In this section, we compare our system with two state-of-the-art GPU graph pattern matching systems: cuTS [92] and GSI [96], and a state-of-the-art CPU system: Dryadic [53]. GSI achieves consistently better (or equal) performance than many previous CPU and GPU implementations such as CFL-Match [9], CBWJ [56], VF3 [15], GPSM [80] and Gunrock [86]. CuTS has shown even better performance than GSI. However, because cuTS does not support labeled queries, we also compare with GSI for labeled matching tasks.

1.8.1 Experimental Setup

Platform: Our experiments are conducted on a dual socket machine with four Nvidia RTX3090 GPUs, two Intel Xeon Gold 6226R 2.9GHz CPUs (32 cores in total), and 512GB RAM. We use GCC9.4.0 and NVCC11.2 with O2 level optimization to compile the code.

Datasets and query graphs: Table 17 lists the data graphs used in our experiments. These graph are obtained from the SNAP [46] repository and are commonly used for evaluating graph pattern matching systems. For query graphs, we first adopt the queries from cuTS. CuTS uses directed query graphs. The 33 directed queries used in their experiments actually have only six undirected patterns. While our system supports both directed and undirected graphs, we use undirected graph in our evaluation because it is a more common setup in the graph pattern matching literature [9, 53, 80, 96, 97]. The 33 queries in cuTS experiments are covered by q_7, q_8, q_{15}, q_{16}, q_{23}, q_{24} in our experiments, among which q_8, q_{16} and q_{24} are cliques (i.e., fully connected graphs). Then,

we randomly select six query graphs from size-5, size-6, size-7 motifs, respectively. In total, we test with 24 distinct query graphs: $q_1 \sim q_8$ of size-5, $q_9 \sim q_{16}$ of size-6, and $q_{17} \sim q_{24}$ of size-7. Our system supports both labeled and unlabeled graphs. To evaluate the performance of labeled matching, we follow the setup in Dryadic [53] and randomly assign ten labels to the data and query graphs. For fairness, we use the same matching order of query nodes (adopted from Dryadic) for all systems.

Settings: In all experiments, we set the *StopLevel* in Algorithm 2 to 2, and the *DetectLevel* for global work-stealing in Section 1.5.2 to 1. The unrolling size is set to 8. We set the *MAX_DEGREE* of array *C* to 4096. The size of array *C* is $NUM\_SETS \times UNROLL \times MAX\_DEGREE \times NUM\_WARPS$. For queries of no more than seven nodes, we have $NUM\_SETS \leq 15$ (this number is determined by the code motion analysis). The maximum number of active warps on our GPU is $82 \times 32 = 2624$. Thus, besides the storage of data graph, our system consumes a fixed 4.8GB GPU memory for queries of up to seven nodes. For graphs whose max degrees are larger than 4096, we store the extra nodes in the sets with more than 4096 nodes on CPU. Because real-world graphs are skewed and few nodes have degrees greater than 4096 (last column in Table 17), it is very rare for the program to access CPU memory. We run cuTS and GSI with the same number of total warps on GPU, and run Dryadic with 64 threads on CPU.

1.8.2 Overall Performance

GSI and cuTS are subgraph isomorphism systems: they obtain edge-induced subgraphs that are isomorphic to the query graph. Thus, we configure our system and Dryadic to run edge-induced matching (by removing the set difference operations) and compare with the two systems. Table 2(a) lists the execution time of unlabeled edge-induced queries on a single GPU. We do not list the execution time of GSI in this table because it either aborts or is dominated by cuTS. We can see that Dryadic consistently outperforms cuTS, while our system outperforms both Dryadic and cuTS for all testcases. Compared with cuTS, our system achieves up to 3385x speedups with an average of 694x. Compared with Dryadic, our system achieves up to 52x speedups with an average

	WikiVote			Enron			MiCo	
	Our	**Dryadic**	**cuTS**	**Our**	**Dryadic**	**cuTS**	**Our**	**Dryadic**
q_1	492	4534	168512	349	3002	155051	22367	128527
q_2	1689	9543	110529	2033	7814	89721	96785	105408
q_3	131	3045	41077	137	2866	36946	20327	46350
q_4	295	3451	31679	355	4151	25935	32288	70327
q_5	246	1647	15612	185	2411	16357	26398	142830
q_6	109	713	29449	73	593	24455	14330	23623
q_7	23	679	11845	32	808	12502	4153	171986
q_8	36	120	9602	62	146	9990	1597	23267
q_9	2932	5271	×	2095	6860	×	1408189	4028865
q_{10}	27904	215643	×	33514	362646	×	4752770	–
q_{11}	1858	41214	×	1641	61693	×	1358694	–
q_{12}	8993	315366	×	9807	510956	×	5636367	–
q_{13}	7485	54704	×	5995	61520	×	4368254	–
q_{14}	651	1423	165745	981	2071	204686	1827655	9364411
q_{15}	88	598	137895	107	886	160304	196210	1237420
q_{16}	48	288	131286	47	305	160022	56089	92557
q_{17}	5599	5963	×	5228	9167	×	–	–
q_{18}	39178	235003	×	31010	344761	×	–	–
q_{19}	9634	131236	×	13417	162903	×	–	–
q_{20}	3608	31267	×	3995	38992	×	–	–
q_{21}	2000	5361	×	4150	7917	×	–	–
q_{22}	2075	33991	×	2194	41391	×	–	–
q_{23}	265	1645	×	338	2607	×	8926635	–
q_{24}	139	531	×	129	608	×	1982441	2530072

(a) Edge-induced

WikiVote		Enron		MiCo	
Our	**Dryadic**	**Our**	**Dryadic**	**Our**	**Dryadic**
296	3169	229	2216	5365	36026
850	6949	1075	4863	20383	69869
106	1429	101	731	1946	14486
186	1471	206	674	3339	12361
194	1767	133	640	3439	13105
65	468	41	200	882	2397
28	178	34	96	6255	23963
36	120	62	146	1597	23267
2088	14515	1455	4947	82274	487075
6380	63026	8871	40426	109851	801632
707	8921	388	2355	4837	34842
21384	294242	22693	144714	1173245	9161840
2668	80389	1659	22919	43399	762982
497	2279	814	1480	275703	923136
86	579	110	307	315526	1298950
48	288	47	305	56089	92557
2216	7667	1399	2639	2159917	4488633
6729	16586	4469	4681	750974	896309
2301	3968	1091	1437	142918	164562
2037	13433	1645	5007	1889880	2854831
1567	4236	2732	2860	–	–
996	3940	902	1770	3235240	6674460
285	1089	356	535	17854918	–
139	531	129	608	1982441	2530072

(b) Vertex-induced

Table 2. Execution time (in milliseconds) of different systems for unlabeled matching. '×' indicates program failure due to out-of-memory. '–' indicates timeout after 8 hours. CuTS only supports edge-induced matching. It fails for all queries on MiCo.

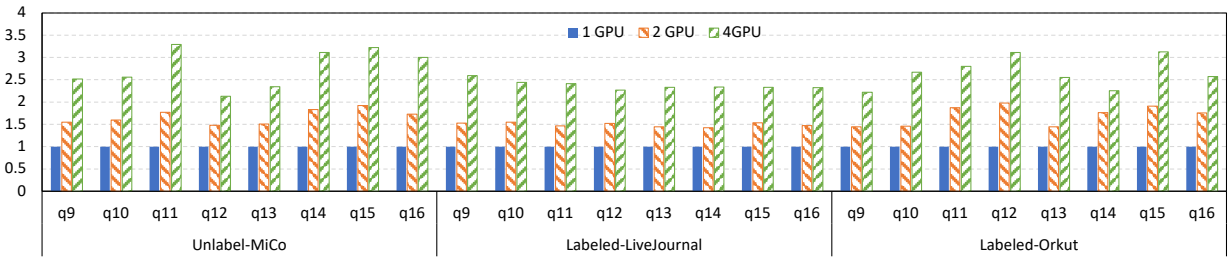


Figure 11. Speedups of labeled and unlabeled size-6 queries across multiple GPUs.

of 11x.

Table 3 lists the execution time of labeled edge-induced queries on a single GPU. Our system shows consistently better performance than the other two systems. It achieves 24x to 991x speedups against GSI, and 1.4x to to 898x speedups against Dryadic. The speedups are more significant on larger data graphs. The average speedup against GSI is 67x, 89x and 306x for WikiVote, Enron and YouTube, respectively. The average speedup against Dryadic grows from 6x to 56x for different graphs. The results indicate that our system scales better on large graphs compared to GSI and Dryadic.

	WikiVote			Enron			YouTube			MiCo		LiveJournal		Orkut		Friendster	
	Our	**Dryadic**	**GSI**	**Our**	**Dryadic**	**GSI**	**Our**	**Dryadic**	**GSI**	**Our**	**Dryadic**	**Our**	**Dryadic**	**Our**	**Dryadic**	**Our**	**Dryadic**
q_1	6	52	307	8	41	527	28	188	×	49	231	558	7524	6701	459615	59850	290752
q_2	6	43	276	9	44	514	51	204	11656	44	335	534	6402	7874	91568	59998	420521
q_3	4	37	193	8	26	415	22	66	9856	12	168	224	3101	359	7595	5681	35879
q_4	5	37	194	7	34	445	24	876	9596	31	243	403	4518	2014	17267	6563	199131
q_5	4	29	179	7	31	440	25	79	9425	30	132	388	3082	2456	3723	5788	35462
q_6	4	32	193	8	28	399	23	124	9694	30	163	385	4360	987	5326	8289	38578
q_7	4	31	170	6	31	426	22	92	9498	11	155	232	3682	332	3221	4829	39381
q_8	4	36	170	7	34	433	22	97	10150	29	205	361	4667	550	2602	5400	39469
q_9	4	28	336	6	139	631	23	84	11531	59	1358	1188	58189	7778	12585	6331	55096
q_{10}	6	77	3453	10	104	6371	54	605	×	108	4937	2083	133943	16867	800817	21219	631370
q_{11}	5	50	454	8	83	1019	27	406	26830	92	2468	1479	130529	1585	1423820	9975	273557
q_{12}	7	50	532	10	65	1043	320	461	34503	362	2693	5943	127701	16316	415670	10472	239624
q_{13}	6	60	434	9	48	754	42	173	11659	86	2048	1487	121818	3788	503492	8078	109472
q_{14}	5	28	183	7	40	450	39	120	9898	436	1550	6927	110435	1288	6013	7726	66602
q_{15}	4	35	211	7	40	424	23	128	9640	88	2230	1212	156588	814	5933	5816	71651
q_{16}	5	49	219	8	39	409	35	152	9759	456	2592	6695	172881	1115	5438	6800	73175
q_{17}	7	43	209	10	39	468	86	155	10053	7313	44682	241432	4163993	7064	18595	12417	115459
q_{18}	8	48	236	11	53	469	65	277	9851	7337	59603	247403	7811903	11116	39459	20339	139946
q_{19}	10	75	251	11	55	513	70	319	9718	7176	55240	228997	7362388	7777	53865	14304	155747
q_{20}	6	55	224	13	53	492	179	253	10582	1129	61624	30353	6278270	6859	27509	11451	136761
q_{21}	6	61	224	10	63	439	54	219	10429	7273	46943	201273	4233415	4286	20143	12280	134500
q_{22}	6	55	202	9	48	465	62	244	10033	1518	56191	35000	7102923	2685	23024	8449	138652
q_{23}	6	48	233	9	44	454	64	324	10149	1522	48920	32808	5669286	1645	20310	7799	140028
q_{24}	6	42	203	9	52	442	58	289	10133	7312	65286	196941	7166516	3395	19515	10332	146617

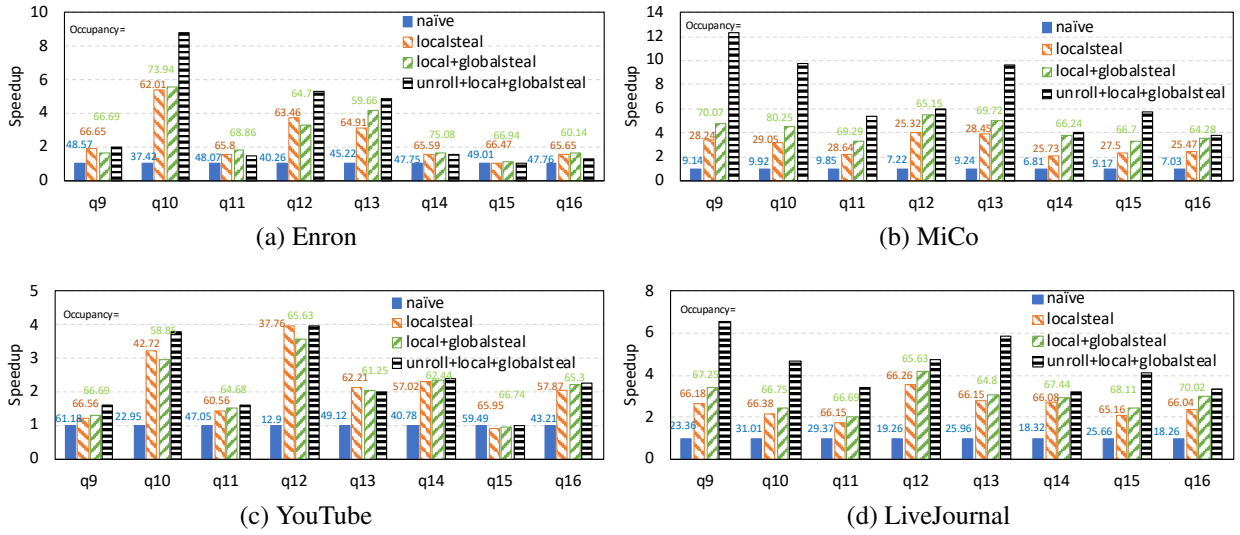
Table 3. Execution time (in milliseconds) of different systems for labeled edge-induced matching. GSI fails for all queries on MiCo, LiveJournal, Orkut and Friendster.

We also compare with Dryadic for vertex-induced matching. For q_8, q_{16} and q_{24} which are cliques, vertex-induced matching is the same as edge-induced. The execution time of unlabeled queries is shown in Table 2(b). Our system outperforms Dryadic for all testcases with a maximum speedup of 30x and an average speedup of 6x.

Our system can be easily run on multiple GPUs by duplicating the input graph and dividing the outermost loop iterations (i.e., V in Fig. 4) across GPUs. Fig. 11 shows the speedups of running labeled and unlabeled $q_9 \sim q_{16}$ on LiveJournal, Orkut and MiCo with two and four GPUs. Our system can also be extended to run on distributed GPU clusters with slight changes in the work-stealing procedure to take the communication cost across machines into consideration.

1.8.3 Benefits of Proposed Techniques

To show the effectiveness of our work-stealing and loop unrolling techniques, we perform an ablation study on our system. We first run a `naive` version without work-stealing and loop unrolling. Then, we allow the warps to steal workloads from other warps within the threadblock (`localsteal`) and across threadblocks (`local+globalsteal`). Last, we unroll the loops and make each warp perform multiple set operations simultaneously (`unroll+local+globalsteal`).



(a) Enron (b) MiCo

(c) YouTube (d) LiveJournal

Figure 12. Speedups of labeled size-6 queries with and without work-stealing and loop unrolling.

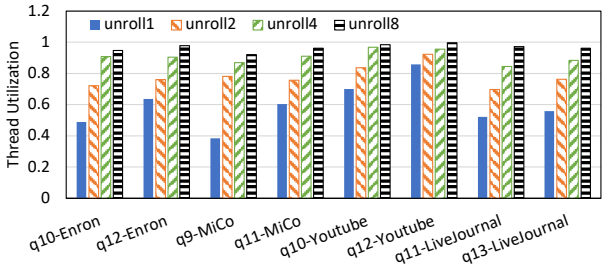


Figure 13. Thread utilization with different unrolling sizes.

Fig. 12 shows the execution time of different versions for labeled size-6 queries on different graphs. We can see that local work-stealing brings the most benefit to our system, achieving more than 2x speedups for almost all testcases. Global work-stealing further improves the performance on MiCo and LiveJournal where the workload is large enough to justify the overhead of copying stacks among threadblocks. It achieves 1.3x to 2.0x speedups on top of local work-stealing on MiCo graph and 1.1x to 1.3x speedups on LiveJournal. Global stealing is less effective on Enron and YouTube as the workload in each warp is already small after applying local work-stealing. The execution time with global stealing is almost the same as without global stealing on these two graphs, indicating that the overhead of our global work-stealing technique is small. To show direct evidence of improvement brought by work-stealing, we profile our system with Nvidia Nsight and obtain warp occupancy with and without work-stealing. The occupancy numbers are labeled in the

figures, and they are consistent with the speedups.

After applying loop unrolling, the performance is further improved. Fig. 13 shows the thread utilization of various queries with different unrolling sizes. As expected, a larger unrolling size leads to higher utilization. Due to increased thread utilization, loop unrolling achieves 1.1x to 2.6x speedups on top of `local+globalsteal` on MiCo, and 1.1 to 1.9x speedups on LiveJournal. Compared to naive version, work-stealing and loop unrolling together achieve up to 12x speedups. All the versions above use code motion. If we disable code motion, the naive baseline will be about 3x slower.

1.9 Related Work

Graph pattern matching and its related problems have been extensively studied in the past decades. Numerous systems with different algorithms have been proposed. As we focus on parallelizing backtracking in this work, we give a summary of backtracking-based graph pattern matching systems.

CPU-based systems: The study of subgraph isomorphism problem dates back to 1970s. Ullmann [82] proposes the first backtracking algorithm that iteratively matches query nodes on data graph based on a certain order. Many studies follow this seminal work and propose different strategies to optimize the matching order [22, 56, 73, 97]. They show that a good matching order can significantly reduce the exploration space and accelerate the matching process. Some recent work show that a dynamic matching order based on the local topology and label distribution of the data graph can further reduce the exploration space [9, 28, 29, 45]. A more recent work, Dryadic [53], proposes to search for an optimal static matching order and optimize the computation tree instead of input adaptation. It achieves state-of-the-art performance on CPU compared with the earlier systems. Since matching order is not the focus of this work, we simply adopt the matching order of Dryadic in our system. However, our system can be extended with any of the previous matching order strategies.

GPU-based systems: There are a number of GPU systems for subgraph isomorphism. All of

them are subgraph-centric. Some systems [80, 86, 96] adopt a breadth-first extension order that favors GPU architecture. They store all partial subgraphs of a certain size before exploring larger subgraphs. Due to the large intermediate exploration space, the partial subgraphs can easily exceed the GPU memory limit. To reduce the memory consumption, some other works adopt a hybrid DFS and BFS extension order [49, 92]. Given a memory capacity, they pre-allocate a portion of memory for each level. To generate the partial subgraphs for next level, they take a set of partial subgraphs at current level that are estimated to fit into the pre-allocated memory. The procedure is repeated for each level until all matching subgraphs are found. CuTS [92] proposes a compact trie-based data structure to further reduce the size of intermediate subgraphs. It reportedly achieves the state-of-the-art performance on GPU compared with earlier systems. Previous work has also considered exploiting multiple GPUs to accelerate pattern matching on large graphs [27, 92]. PBE [27] proposes a matching algorithm on partitioned graphs so that each GPU only holds a portion of the data graph.

Distributed systems: Graph pattern matching has also been studied on distributed systems [8, 42, 43, 71, 74, 88, 94]. The main challenge is to balance the workload among machines. CECI [8] proposes a compact embedding cluster index to divide the data graph into multiple embedding clusters for parallel processing. They design a proactive workload balancing strategy with a search cardinality based cost function. RADS [71] proposes a region-grouped multi-round expand technique to reduce communication and minimize intermediate result storage. BENU [88] proposes a task splitting technique based on node degree to optimize the load balance among machines. GraphPi [74] uses a communication thread to maintain a task queue on each machine and steal work from other machines when its task is smaller than a threshold.

1.10 Conclusion

In this work, we study the parallelization of backtracking-based graph pattern matching on GPU. We propose a stack-based implementation that avoids the synchronization and memory consumption issues of previous systems. We also show that the performance of our system can

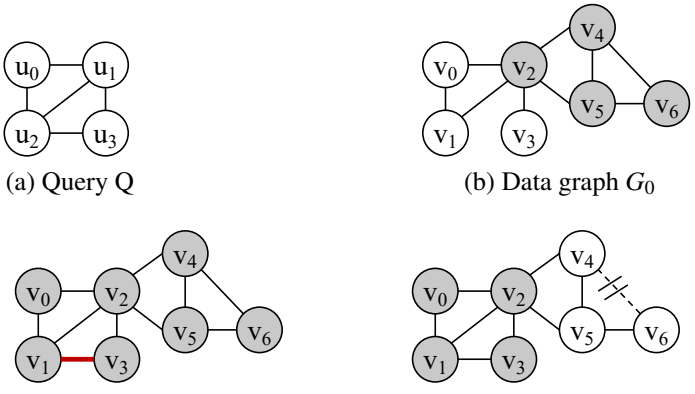
be improved by applying a series of loop optimizations. The experiments show that our system significantly outperforms the state-of-the-art solutions.

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

2.1 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 [58], social network analysis [62], and cybersecurity [62].

While subgraph matching on static graphs has been extensively studied in the past decade [10, 23, 28, 73, 82, 90, 92], there is a growing interest in supporting subgraph matching on dynamic graphs. The task is often referred to as *continuous subgraph matching* (CSM). Fig. 38 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

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

Figure 14. 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 ru-

mors [87]. 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 [69].

Different algorithms and optimizations have been proposed for CSM [21, 24, 40, 78]. An early work [24] simply re-matches the query pattern on the data graph every time the data graph is updated. More recent work [21, 40, 41, 78] 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 [25, 70] (explained in more detail in Sec. 4.2).

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 [90, 92, 96], 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 [38] 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 [78]), achieving 1.4x to 11.4x speedups (with an average of 4.1x) against the best CPU implementation.

2.2 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.

2.2.1 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 2 (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 $G_{k+1} = G_k \oplus e_k$.

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

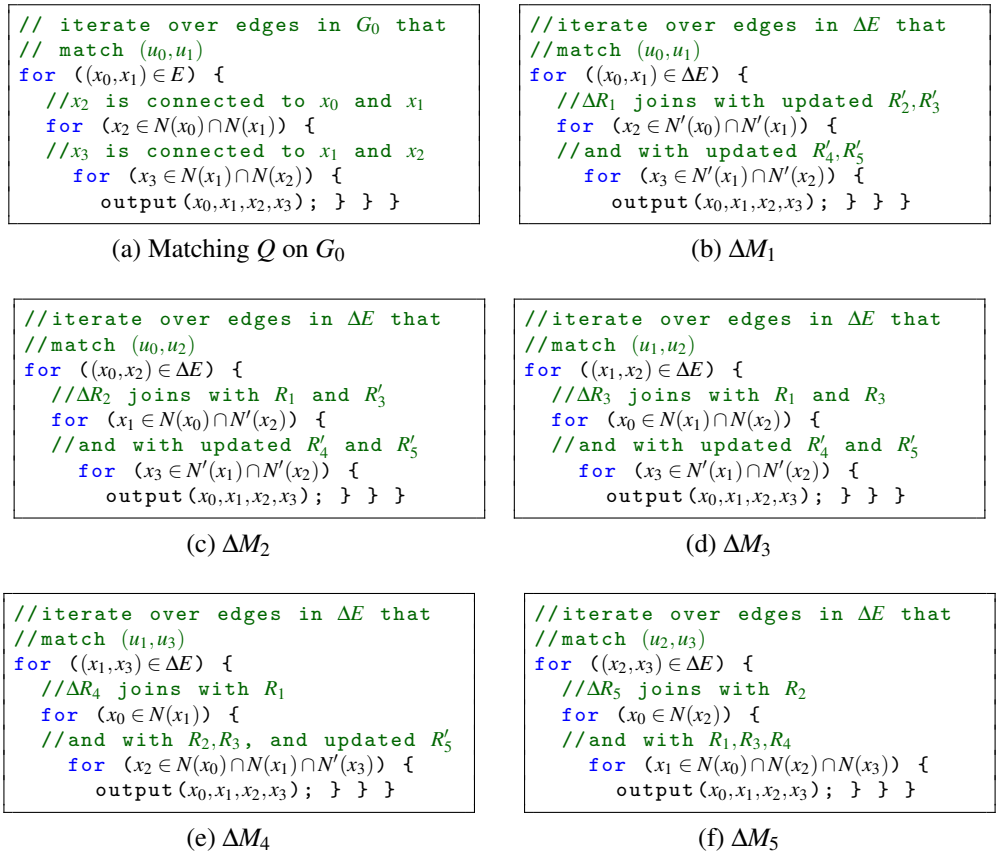


Figure 15. Continuous subgraph matching implemented as worst-case optimal join. E represents the edges in the initial graph. ΔE represents a batch of edge updates. N and N' represent the original and the updated neighbor lists respectively.

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.

2.2.2 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 [57, 60, 80]. For example, matching Q in G_0 in Fig. 38 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 [25, 70], 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

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

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 [40] 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 [3, 40] has employed worst-case optimal join (WCOJ) algorithms (e.g., Leapfrog Triejoin [83]) 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. 15. The loop in Fig. 15a matches Q in Fig. 38a 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. 15b 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 [60]. Therefore, the join operations in Formula (1) has a time complexity bounded by the size of ΔM_i, which follows the AGM inequality [5]:

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

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 $O(\sum_{i=1}^{m} |\Delta M_i|)$.

2.2.3 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∼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 [75]. CUDA provides three ways for CPU-GPU data communication [63]: 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.

2.3 Overview of Gcsm

The goal of our system is to use GPU to accelerate the computation in Fig. 15. Since previous work has mapped the static matching procedure in Fig. 15a onto GPU [38, 90, 92, 96], the focus of this work is to efficiently run the incremental matching procedures (Fig. 15b-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. 28a). 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. 2.4.

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. 15) 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. 2.5.

Fig. 16 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:

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;

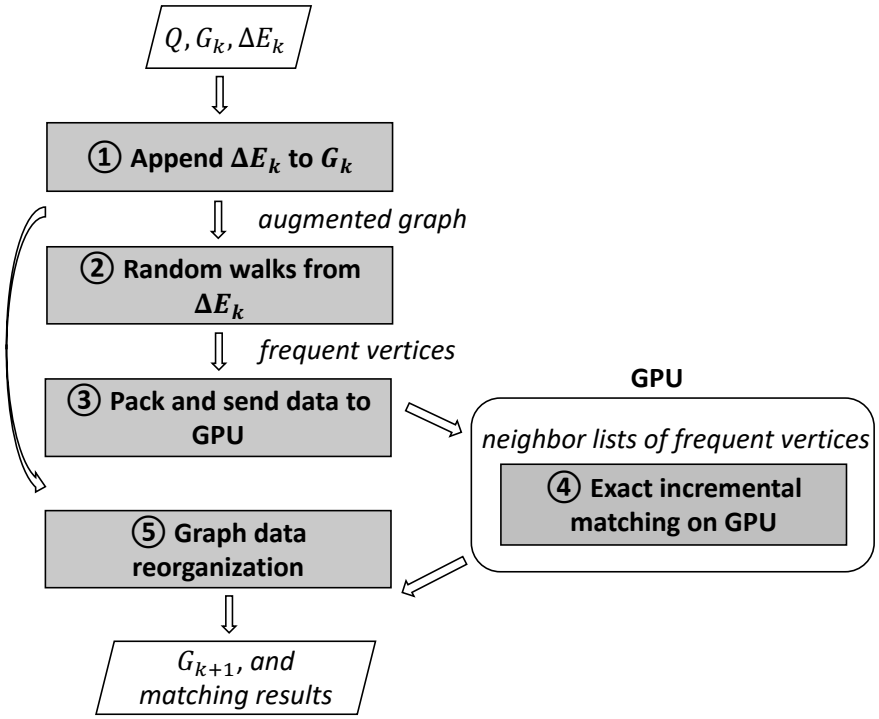


Figure 16. Workflow of GCSM.

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

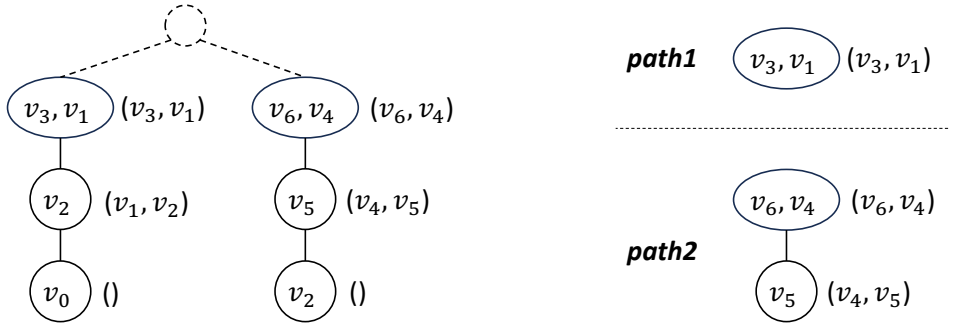
⑤ The graph data are reorganized on the CPU to ensure the neighbor lists of G_{k+1} are sorted.

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

2.4.1 Estimating Access Frequency With Random Walk

Consider the exact matching procedure in Fig. 15. The execution of each nested loop can be depicted as a tree structure where different tree levels represent different loop levels and each



(a) An example execution tree (b) Two sampled paths

Figure 17. Sampling multiple paths from an execution tree.

tree node represents one iteration of the loop. For example, Fig. 17a shows the execution tree of the nested loop in Fig. 15b with G_0 and ΔE from Fig. 38. 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. 15b. 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. 17a.

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. 17b shows two example paths sampled from the execution tree in Fig. 17a.

During the random walk, we record the access to each vertex at each step (shown in the parenthesis in Fig. 17b). 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 [35, 65]. 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

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

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 [89], we have

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

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

□

2.4.2 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 $B_1 \geq 1$, 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

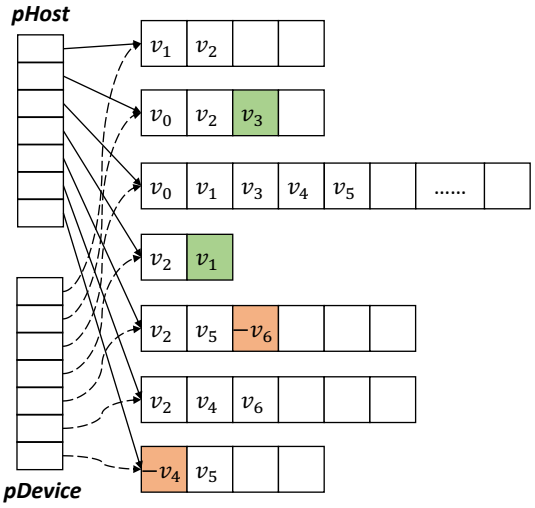


Figure 18. Maintenance of graph data structure on the CPU for the graph update in Fig. 38. New edges (colored in green) are appended to the end of corresponding neighbor lists; Deleted edges (colored in orange) are marked as negative values.

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 $O(|V|)$.

2.5 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. 15. 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.

2.5.1 Graph Data Structure On CPU

We store the data graph as adjacency lists in CPU memory, as shown in Fig. 18. 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.

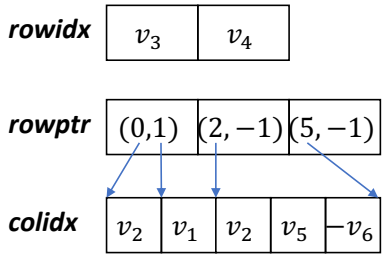


Figure 19. Graph data sent to GPU in DCSR format.

2.5.2 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. 19 shows an example of the DCSR data structure with the graph update in Fig. 18. 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.

2.5.3 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. 15. We build the kernel on top of a recent GPU subgraph matching system called STMatch [90]. 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. 15). 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 [90]) 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 19. 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 18), and then load the data from the CPU memory by GPU zero-copy access.

Table 4. 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. 2.5.1) 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.

2.6 Experimental Results

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

2.6.1 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 17 lists the data graphs used in our experiments. Among the data graphs, AZ, LJ, PA, CA, and FR are from the SNAP dataset [46], while SF3K and SF10K are random graphs generated by the LDBC graphalytics [34]. Following the existing research on streaming graphs [41, 78, 79], 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.

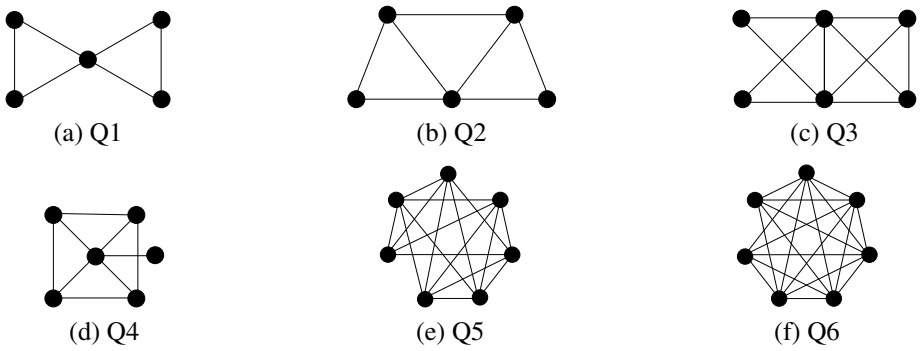


Figure 20. Query graphs.

For AZ, LJ, PA, and CA, we randomly select 10% of edges from the original graph, following the settings in [78, 79]. 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. 47.

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 [38], 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 [90] for the matching task.

We also compare our system with two CPU baselines. The first baseline is RapidFlow (`RF`) [78], 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. 15, which always start the matching process from the updated edges. For a fair comparison, our CPU code uses the same stack-based implementation [90] 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.

2.6.2 Comparison With Naive GPU Implementations

Fig. 21 to Fig. 24 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.,

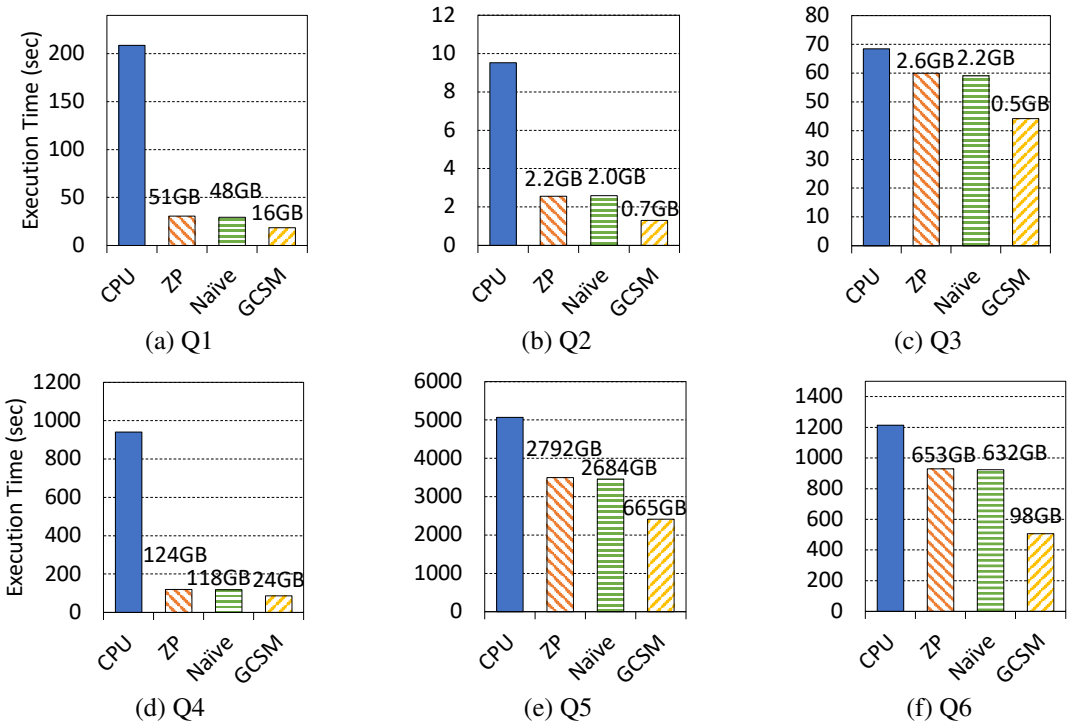


Figure 21. Execution time for matching different query patterns from a batch of 4096 edges on FR graph. Data access sizes from CPU are labeled on each bar.

Step-2 and Step-3 in Fig. 16) are included in the execution time of `GCSM` in Fig. 21 to Fig. 23. Details of the breakdown overhead are given in Table 5. 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 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

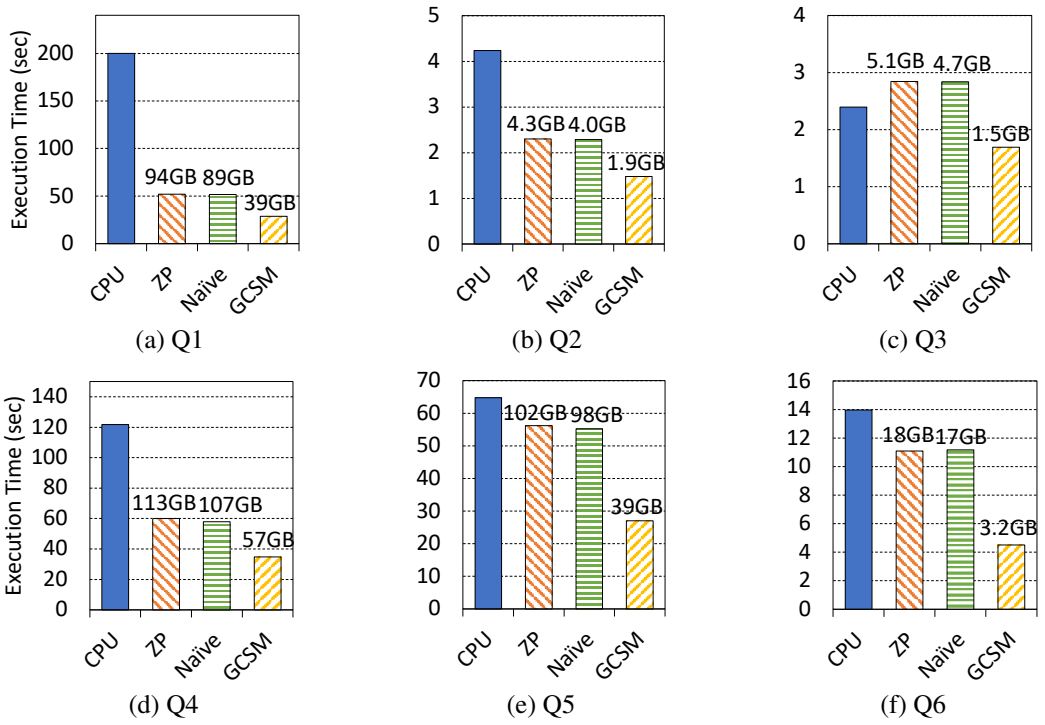


Figure 22. Execution time for matching different query patterns from a batch of 4096 edges on SF3K graph.

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. 24, 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. 47 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. 21 to Fig. 24, `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 26 shows the breakdown of execution time for processing one batch of edge

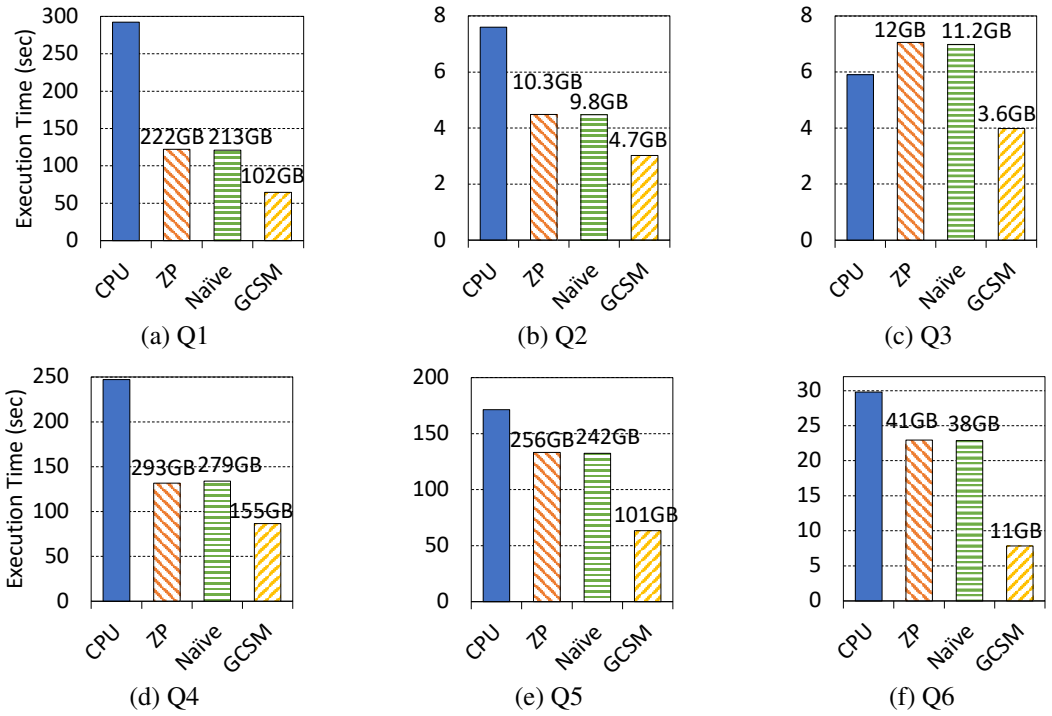


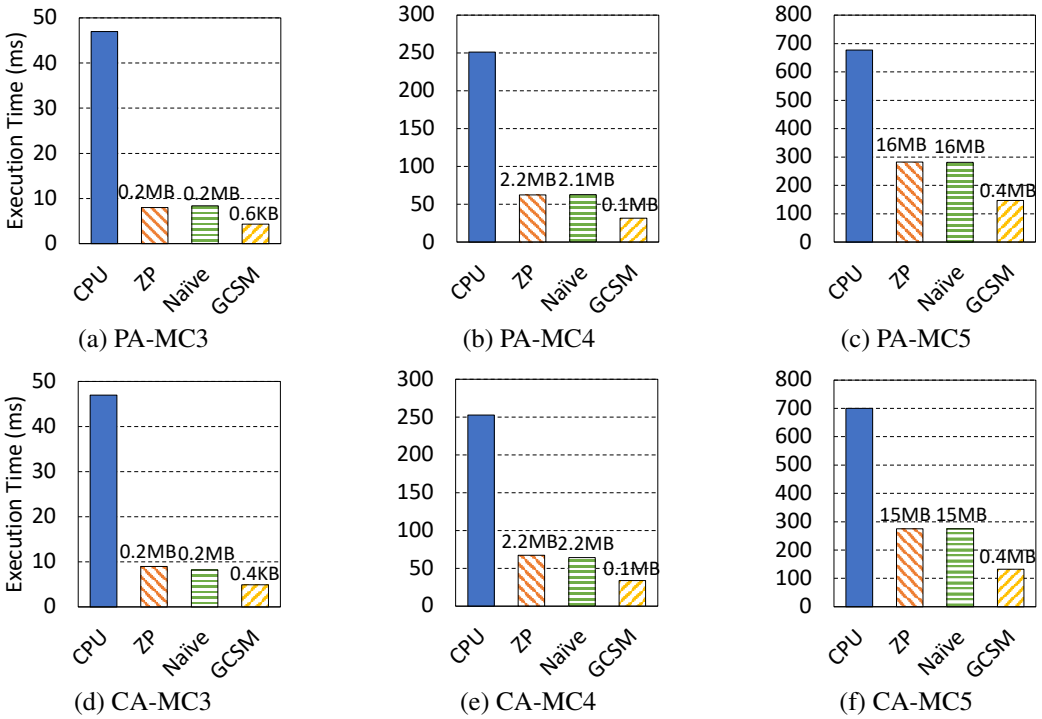
Figure 23. Execution time for matching different query patterns from a batch of 8192 edges on SF10K graph.

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. 25, 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.

2.6.3 Comparison With CPU Implementations

The execution time of our CPU implementation is included in Fig. 21 to Fig. 23. The CPU implementation is slower than the naive zero-copy implementation on GPU (for most cases),



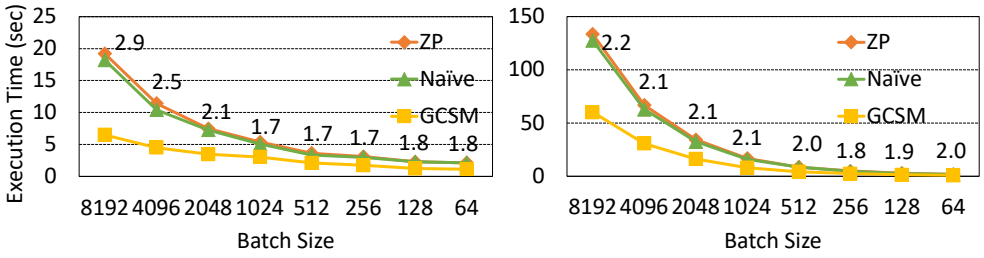
(a) PA-MC3 (b) PA-MC4 (c) PA-MC5

(d) CA-MC3 (e) CA-MC4 (f) CA-MC5

Figure 24. Execution time for counting size-3, 4, and 5 motifs on RoadNetPA and RoadNetCA. The batch size is set to 4096.

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 three large graphs. Therefore, we compare the performance with RapidFlow using two smaller graphs (AZ and LJ). The results are shown in Fig.27. 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.



(a) SF3K-Q6 (b) SF10K-Q5

Figure 25. 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.

Table 5. 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%

2.6.4 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 28b 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 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. 28a), the cached data on GPU can save most of the

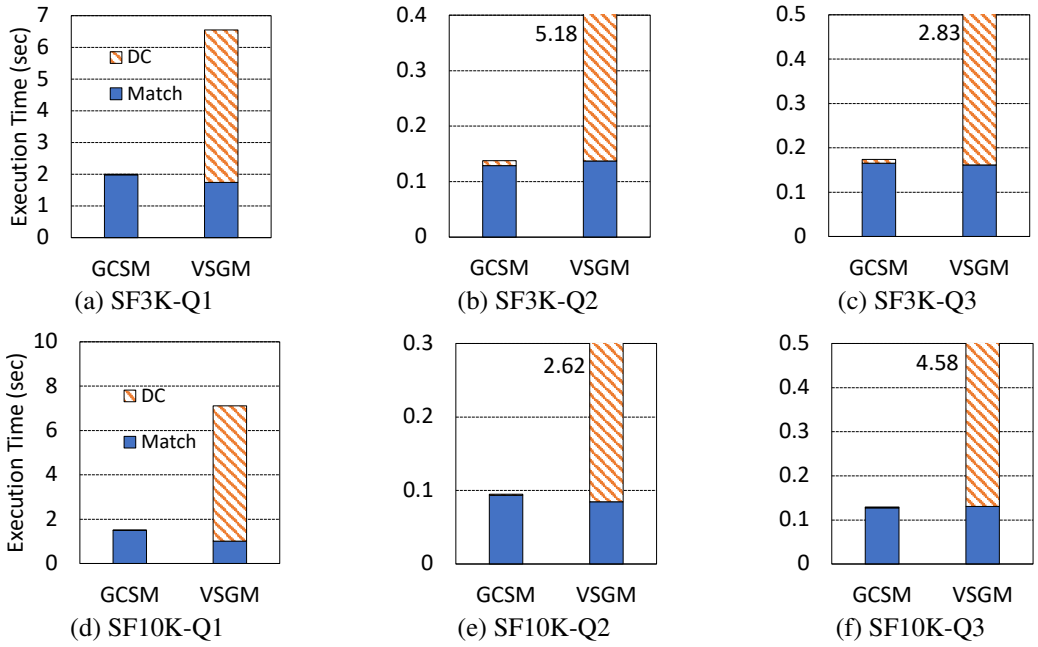


Figure 26. Breakdown execution time of VSGM and GCSM. 'DC' includes the time for identifying the caching vertices and copying their neighbor lists to GPU. 'Match' is the matching kernel execution time on GPU.

Table 6. Graph reorganization time (in milliseconds).

	$\|\Delta E\| = 4096$	$\|\Delta E\| = 8192$
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

accesses to CPU during the matching process.

2.6.5 Graph Reorganization Overhead

So far we have evaluated Step-1 to Step-4 of our system shown in Fig. 16. 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 6, the average time of graph reorganization

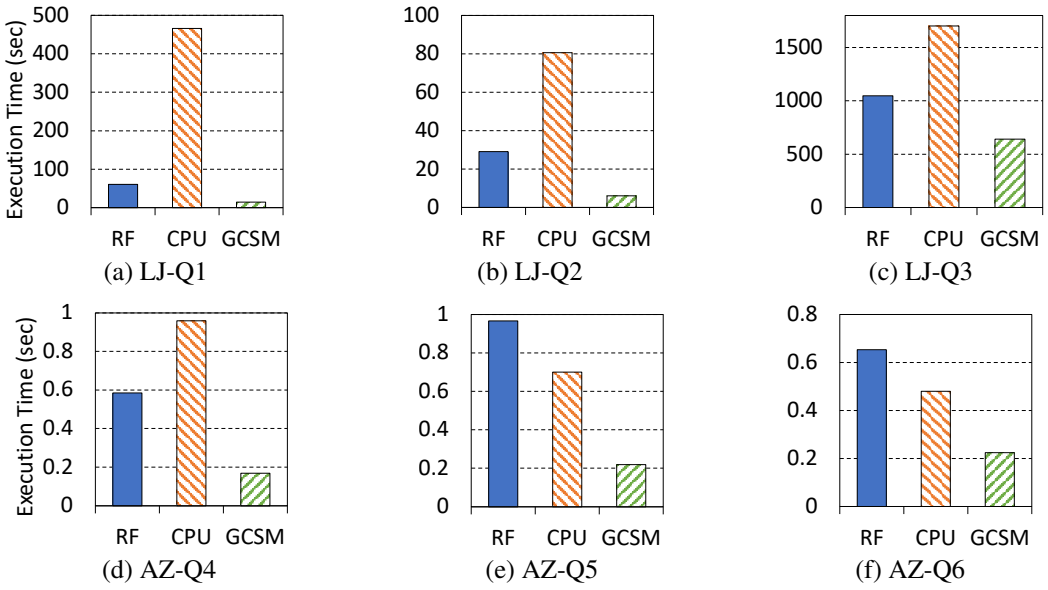


Figure 27. Performance comparison with RapidFlow.

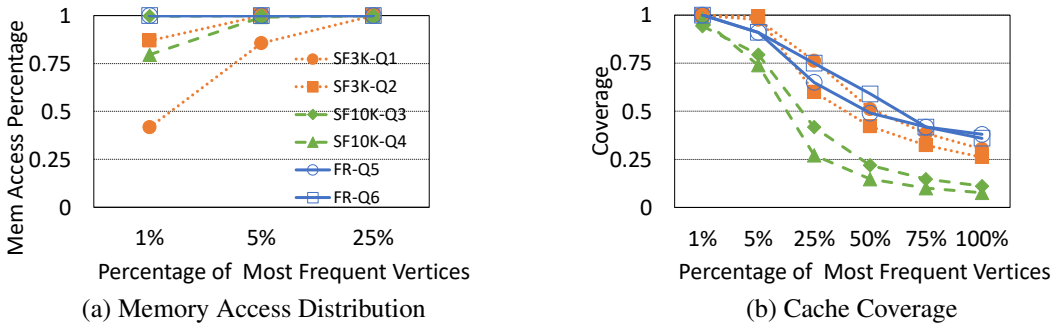


Figure 28. Memory access distribution and cache coverage of incremental matching.

for a batch of edge updates is no more than a few milliseconds, which is much smaller than the matching time in Fig. 21 to Fig. 24. 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.

2.7 Related Works

Static subgraph matching: The study of static subgraph matching dates back to Ullmann [82], which proposes the first backtracking algorithm for the task. Following Ullmann's work, many studies [23, 32, 73] 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 [55] 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 [3,40,57]. With the development of GPUs, many efforts have been made to utilize the massive parallelism of GPU to accelerate static subgraph matching [80,85,90,92,96]. CuTS [92] 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 [90] proposes a stack-based data structure to store intermediate subgraphs and employ work-stealing to balance the workloads among computing units on GPU. PBE [27] and VSGM [38] are two works that focus on subgraph matching in distributed GPU systems, allowing for matching on extremely large graphs.

Continuous subgraph matching: IncIsoMatch [24] 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 [40] avoids the repeated matching for each graph update and computes the incremental matching results by performing multi-way join operations. SJ-Tree [21] 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 [41] 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 [41], SymBi [59] proposes candidate vertices set pruning using query edges. Rapidflow [78] 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 [12].

ScaleMine [2] employs sampling to estimate the workload of different branches of the exploration process and uses the information to improve load balance. SampleMine [39] proposes a framework for applying random sampling to different graph mining tasks based on loop perforation. C-SAW [64] 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.

2.8 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.

3 MATCHA: A LANGUAGE AND COMPILER FOR BACKTRACKING-BASED SUBGRAPH MATCHING

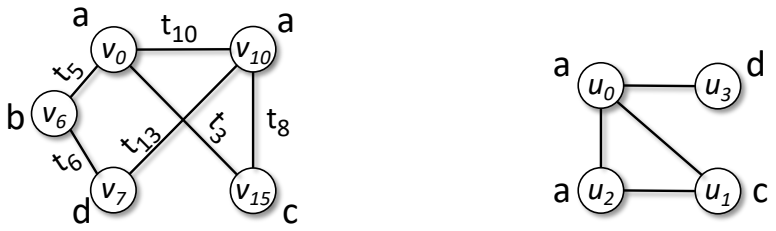
3.1 Introduction

Subgraph matching is the key building block of many graph analytics and learning applications. It aims to find subgraphs in a data graph conforming to the structural constraints of a query pattern. A subgraph matching algorithm can be used to extract information from graph-structured data in many domains, including bioinformatics [58], social network analysis [81], and cybersecurity [62]. Recently, subgraph matching also finds increasing use in graph-based machine learning applications, such as anomaly detection [4, 61], entity resolution [7], and community detection [72].

Fig. 29 shows an example of the most basic subgraph matching problem. Given a data graph G and a query pattern Q, the task is to find all subgraphs in G that match the pattern Q. Both G and Q here have labeled vertices. There are two subgraphs in G with the same structure as Q. One subgraph $(v_{10}, v_{15}, v_0, v_7)$ is a valid match, while the other $(v_0, v_{10}, v_{15}, v_6)$ is not due to mismatched labels.

As an NP-hard problem, subgraph matching can be time-consuming on large graphs. Following the basic idea of backtracking [82], many advanced algorithms have been developed [3, 11, 23, 28, 40]. Most of these algorithms are implemented on CPU, while some recent systems utilize GPU to accelerate computation [17, 28, 53, 55, 73, 77].

Table 7 compares the state-of-the-art subgraph matching systems regarding their supported tasks, algorithms, and optimizations. Since each system features unique algorithms and optimizations, it has been increasingly difficult to compare and combine different solutions, thus hindering progress in this research area. For example, RapidMatch [77] proposes a relation-filtering technique with join plan optimization. Its join-based algorithm performs better than many other subgraph matching systems on CPU. However, migrating the algorithm to GPU is nontrivial. It is unclear whether the algorithm can outperform an efficient implementation of more basic algo-



(a) Data Graph G (b) Query Q

Figure 29. Example data graph and query pattern.

rithms on a GPU (e.g., G2Miner [20]).

Although some previous systems aim to be general-purpose [16, 19, 20], they hard-code algorithms for different tasks and support limited computation patterns. For example, G2Miner [20] assumes the subgraph is always extended vertex by vertex using set operations. Users cannot express the join-based algorithm of RapidMatch [77] with its programming interface. It also does not support backtracking with edge extension as adopted by Everest [95].

To overcome the limitations of existing systems, we propose a domain-specific language called Matcha for implementing subgraph matching algorithms. The key language constructs in Matcha include a `List` data type and an `apply` operation that applies a user-defined function to items in the `Lists`. Our design is based on the observation that the backtracking procedure in subgraph matching algorithms can be modeled as nested data stream processing. A Matcha program stores intermediate subgraphs in a `List` and applies a function to each subgraph to generate a new `List` of larger subgraphs until the desired pattern size is reached. By allowing programmers to specify different exploration strategies with the `apply` operator, Matcha supports a broader range of subgraph matching algorithms than previous systems.

We implement a domain-specific compiler that generates efficient C++/CUDA code based on Matcha programs. Unlike previous systems that hard-code the optimizations, we decouple the optimizations from algorithm specification by implementing optimizations as transformations on the IR. The advantage of such a design is that it allows us to compose and configure optimizations for different subgraph matching tasks, achieving better performance and portability than previous systems. Finally, the optimized IR is translated into C++/CUDA code with a code generator.

In summary, the paper makes the following contributions:

- We propose a DSL called Matcha for expressing various backtracking-based subgraph matching algorithms.
- We implement a compiler that translates Matcha programs into C++/CUDA code that runs on CPU and GPU.
- We implement a number of optimization passes including operator fusion, code motion, and work stealing in our compiler to improve the performance of generated code.
- Our system provides a unified framework for developing and comparing subgraph matching algorithms.

The experiments show that 1) Matcha can be used to easily reimplement state-of-the-art subgraph systems, and the reimplementations perform comparably to the original systems when the same optimizations are applied; 2) Additional optimizations can be easily incorporated into Matcha programs, and the optimized re-implementations outperform the original systems by 23.7x on average; 3) Matcha simplifies the comparison of algorithms originally implemented on different platforms. For example, we find that the join-based algorithm proposed in RapidMatch does not have an obvious performance advantage over the basic vertex-extension backtracking algorithm on the GPU; 4) Developing new algorithms using Matcha is convenient. As an example, we combine a decomposition-based algorithm with the join-based algorithm of RapidMatch, which results in better performance than either algorithm alone.

3.2 Background

3.2.1 Subgraph Matching Problems

The data graph in many real applications can be abstracted as a 4-tuple $G = (V, E, L, T)$, where V represents a set of vertices, $E = \{(v_i, v_j) | v_i, v_j \in V\}$ is a set of edges, L is a labeling function that assigns labels to the vertices, and T is a function that assigns (label or timing) information

Table 7. Comparison of state-of-the-art subgraph matching systems: G2Miner (GM) [20], STMatch (STM) [90], DecoMine (DM) [17], RapidMatch (RM) [77], Everest (EV) [95], and our system Matcha (Mt), in terms of support for algorithms, performance optimizations, and matching tasks.

		GM	STM	DM	RM	EV	Mt
Algorithms	Vertex Ext.	✓	✓	✓	✓	x	✓
	Edge Ext.	x	x	x	✓	✓	✓
	Pattern Decomp.	✓	x	✓	x	x	✓
	Hash Join	x	x	x	✓	x	✓
Optimizations	Multi-threading	x	x	✓	x	x	✓
	GPU	✓	✓	x	x	✓	✓
	Code Motion	✓	✓	✓	x	x	✓
	Work Stealing	x	✓	x	x	✓	✓
	Dyn. Scheduling	✓	x	✓	x	x	✓
	Thread Group	x	x	x	x	x	✓
Tasks	Edge-Induced	✓	✓	✓	✓	✓	✓
	Vertex-Induced	✓	✓	✓	x	✓	✓
	Labeled	✓	✓	✓	✓	✓	✓
	Temporal	x	x	x	x	✓	✓

to the edges. A graph $G' = (V', E', L', T')$ is a *subgraph* of $G = (V, E, L, T)$ if $V' \subseteq V$, $E' \subseteq E$, $L'(v) = L(v), \forall v \in V'$, and $T'(e) = T(e), \forall e \in E'$. A subgraph G' is *vertex-induced* if all the edges in E that connect the vertices in V' are included E'. For example, $(v_{10}, v_{15}, v_0, v_7)$ is a vertex-induced subgraph of G in Fig. 29(a). A connected subgraph G' is *edge-induced* if it is not vertex-induced. All subgraph matching problems are defined based on the concept of graph isomorphism:

Definition 3 (Graph Isomorphism)**.** Two graphs $G_a = (V_a, E_a, L_a, T_a)$ and $G_b = (V_b, E_b, L_b, T_b)$ 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 $(f(v_i), f(v_j)) \in E_b$.

Depending on the usage of the label and edge information, there are various subgraph matching tasks in real applications. Some of the most common ones are:

- *k-motif counting*, which counts the number of all size-k unlabeled subgraphs in a data graph G.
- *Subgraph listing*, which lists all the subgraphs in a data graph G that are isomorphic to a given query graph Q. The query can be either labeled or unlabeled. For labeled queries, the subgraphs must match the labels of vertices in Q.

- *k-clique listing*, which lists all the fully-connected unlabeled subgraphs of size-k in a graph G.
- *Temporal subgraph listing*, which lists all the subgraphs in a graph $G = (V_g, E_g, L_g, T_g)$ that match a temporal query $Q = (V_q, E_q, L_q)$. Given an ordering of connected edges in E_q (i.e., $E_q = \{e_i\}_{i=1}^{l}$), $T_g(f(e_i))$ must not exceed $T_g(f(e_{i+1}))$. A subgraph in G matches Q iff it is an isomorphism f with $T_g(f(e_l)) - T_g(f(e_1)) \leq \delta$, and $T_g(f(e_{i+1})) - T_g(f(e_i)) \leq \delta_i, \forall i \in [1, l)$, where δ is the time constraint set for Q, and δ_i is the time constraint set for each edge in Q.

3.2.2 Subgraph Matching Algorithms

We focus on backtracking-based algorithms for subgraph matching. While some ML-based algorithms have emerged in recent years [47, 50, 84], they are not as accurate as traditional combinatorial algorithms and are beyond the scope of this paper.

The combinatorial algorithms considered in this work follow a backtracking idea [82] – starting from an empty subgraph, the algorithm gradually extends the subgraph to the size of the query pattern. The most basic backtracking algorithm extends the subgraph by a vertex at each step. Take the query pattern in Fig. 29(b) as an example. The algorithm first finds all the vertices in the data graph that have the same label as u_0. If the query is unlabeled, it will be all vertices in the data graph. Next, starting from the size-1 subgraphs, the algorithm finds all vertices that match u_1. Since u_1 is connected to u_0, the matching vertices must be neighbors of the first vertex. This gives us a set of size-2 subgraphs that match (u_0, u_1). The algorithm continues to find vertices that match u_2. Since u_2 is connected to both u_0 and u_1, the vertices that match u_2 should be neighbors of both the first two matching vertices, and they can be computed with a set intersection operation. Last, the algorithm finds all matching vertices for u_3 based on its connectivity with u_0, u_1, u_2.

Based on the basic backtracking algorithm, many algorithm variants and optimizations have been proposed to improve the performance of specific subgraph matching tasks. For example, backtracking can be executed with edge extensions, where an intermediate subgraph is extended by an edge instead of vertex at each step. Such edge-extension algorithms are most naturally used for querying graph databases where the graph edges are stored as relations and the subgraph extension

is achieved by joining the relations [3, 5, 40, 57]. More generally, an intermediate subgraph can be extended with another adjacent subgraph. This corresponds to algorithms that join two connecting subgraphs into a bigger subgraph [40, 57]. The pattern-decomposition algorithm [17, 66] can also be considered as extending a subgraph with one or more overlapping subgraphs. These algorithms also incorporate various optimizations to prune the subgraph exploration space as much as possible [10, 23, 28].

3.3 Limitations With Existing Subgraph Matching Systems

While numerous systems for subgraph matching have been proposed [3,17,53,55,66,77,90, 95], each of these systems features unique algorithms and optimizations. This makes it challenging to compare and combine existing techniques, and to develop new techniques based on prior work.

Restricted Algorithm Support. Previous subgraph matching systems implement algorithms that are best suited for specific tasks. For instance, DecoMine [17] employs a pattern-decomposition-based algorithm [66]; it is most efficient for counting subgraphs that can be divided into independent sub-patterns. RapidMatch [77] implements a join-based algorithm; it works best for labeled queries. AutoMine [55] supports both labeled and unlabeled queries, but it only runs the basic backtracking algorithm that matches the pattern vertices one by one. We summarize the algorithm support of state-of-the-art subgraph matching systems in Table 7.

Inconsistent Optimizations. The existing subgraph matching systems are inconsistent in performance optimizations. First, they are implemented on different platforms. Most of the systems are CPU-based [17, 28, 53, 55, 73, 77], while some recent systems utilize accelerators such as GPUs to speed up computation [20, 90, 92, 96]. This variety makes it difficult to compare and combine optimization techniques across different platforms. For instance, RapidMatch [77] is a CPU-only system. It is unclear whether its join-based algorithm can run efficiently on GPU and whether it can outperform the optimized GPU systems such as STMatch [90] and G2Miner [20] that use more basic vertex-extension algorithms. Even on the same hardware, different systems feature different optimizations, as summarized in Table 7.

Given the limitations of the existing systems, a unified framework that allows the development and integration of subgraph matching algorithms and optimization techniques across different platforms is desirable.

3.4 The Matcha Dsl

We propose a domain-specific language (DSL) called Matcha to express backtracking-based subgraph matching algorithms. The language supports four basic data types: `Var`, `Tuple`, `List`, and `Graph`. A `Var` represents a scalar variable, which can be an integer, floating-point, or boolean. A `Tuple` comprises one or multiple `Vars` of the same data type. A `List` is an iterable set of `Vars` or `Tuples`. A `Graph` is a list of edges with additional information associated with vertices and/or edges. Matcha provides an API that allows users to perform various operations on the four data types. In this section, we describe the API using a Python-like syntax.

3.4.1 Programming Interface

Matcha supports basic arithmetic $(+, -, *, /)$ and comparison operations on integer and floating-point `Vars`, as well as logical operations (and, or, not) on boolean `Vars`. To create a `Tuple`, users specify the `Vars` in it with the **mk_tuple**$(nvars, dtype, val)$ operator. Here, *nvars* is the size of the `Tuple`, *dtype* is the data type of the `Vars`, and *val* is an optional argument for initializing the `Tuple`. The user can obtain a `Var` from a specific position in a `Tuple` t with the **get**(t, pos) operator.

Users can create a `List` in Matcha with the operator: **mk_list**$(nitems, type, val)$. Here, the *nitems* is an integer indicating the maximum number of items that can be stored in the `List`. The *type* argument specifies the type of the items, which can be `Var` or `Tuple`. The *val* argument is for initializing the `List`.

To define a `Graph`, users need to specify its edge list with the **mk_graph**$(edges, label, einfo)$ operator. Here, the *edges* must be a `List` of two-integer `Tuples` representing the edges' source and destination. Optionally, the user can specify vertex and edge information on the graph. The

vlabel argument is also a `List` of two-integer `Tuples`, where the first number is the vertex ID and the second number is the label of that vertex. The argument *einfo* is a `List` of the same length as *edges*; it associates a `Var` to each edge in the edge list.

Matcha supports a variety of operations on `List` and `Graph`, which are essential for implementing subgraph matching algorithms:

- **neighbor**(*g*: `Graph`, *v*: `Var<int>`). This operator returns the neighboring vertices of a vertex *v* in a Graph *g*. The returned neighbors are stored as a `List<Var<int>>`.
- **edge_info**(g: Graph, v: `Var<int>`) returns the information on the neighboring edges of a vertex *v* as a `List<Var>`.
- **vertex_label**(*g*: Graph, *v*: `Var<int>`) returns the label of a vertex *v* as a `Var<int>`.
- **apply**(*func*: `Func(T1,...)`→`Tout`, *cond*: `Func(T1,...)` →`Var<bool>`, *l1*: `List<T1>`, ...)→`List<Tout>`. This is the most important operator in Matcha. It accepts one or multiple `Lists` as the input and iterates over items in all the `Lists` at the same time. It performs computation on each item with a function $func$, and returns a new `List` that contains the computed results. If the *cond* function is provided, the operator evaluates the *cond* function first and only computes $func$ on items where the condition is `True`.
- **intersect**(*l1*: `List<Var<T>>`, *l2*: `List<Var<T>>`) → `List<Var<T>>` computes the intersection of two `Lists` of `Vars`. An item *t* is in the resulting `List` if and only if it is in both *l*1 and *l*2. This operator is commonly used in subgraph matching algorithms to obtain nodes connected to two previously matched nodes.
- **diff**(*l1*: `List<Var<T>>`, *l2*: `List<Var<T>>`) → `List<Var<T>>`. Similar to `intersect`, this operator processes two `Lists` of `Vars`, but it computes the difference between the two `Lists`. An item *t* is in the result if it is in *l*1 but not in *l*2.
- **sum**(*l*: `List<Var<T>>`) → `Var<T>` sums up the `Var` items in a `List`. The operator is convenient for subgraph counting tasks.

```
1  def RapidMatch(L01, L02, L03, L12):
2    G02 = mk_graph(edges = L02)
3    G03 = mk_graph(edges = L03)
4    G12 = mk_graph(edges = L12)
5    def f1(e):
6      S0 = neighbor(G02, get(e, 0))
7      S1 = neighbor(G12, get(e, 1))
8      C2 = intersect(S0, S1)
9      def f2(v2):
10       C3 = diff(neighbor(G03, get(e, 0)), mk_list(val=[get(e, 1), v2]))
11       return size(C3)
12     return sum(apply(func=f2, cond=None, C2))
13   return sum(apply(func=f1, cond=None, L01))
```

Listing 1. Join-based algorithm implemented in Matcha for the query in Fig. 29(b).

- **size**(l: `List|Tuple`) $\rightarrow$ `Var<int>` returns the number of items in a `List` or `Tuple`.

Examples: Listing 1 shows a Matcha implementation of a join-based algorithm [77] for the query in Fig. 29(b). The algorithm takes as input the filtered edge `Lists` ($L01$, $L02$, $L03$, $L12$) that match the four edges of the query pattern. The `mk_graph` operations in line 2-4 build the index data structures for the edge lists corresponding to (u_0, u_2), (u_0, u_3) and (u_1, u_2). We will explain how the index data structure is constructed in Section 3.5.1. In line 13, the algorithm starts by iterating over the edges in $L01$. For each edge, it computes the set of vertices $C2$ that match u_2 by intersecting neighbors of v_0 and v_1 (line 8). Next, for each vertex in $C2$, it counts the vertices that match u_3 (line 9-11). The counts are summed up to obtain the final result.

Listing 2 shows another example of Matcha implementing a decomposition-based algorithm [17, 66] for the query in Fig 29(b). Since the algorithm takes the entire graph as input, it needs to first filter out the edges that do not match the labels of (u_0, u_1). This can be achieved by an `apply` operator with a filtering condition (line 15). For each matching edge, it computes the vertices that match u_2 by intersecting neighbors of v_0 and v_1 and filtering out the vertices with labels different from u_2 (line 10). The vertices that match u_3 are the neighbors of v_0 with the label of u_3 (line 11). Next, since u_2 and u_3 are isolated by (u_0, u_1) in the query pattern, we can directly calculate the number of matching subgraphs on an edge as `size(C2)` $*$ `size(S3)` (line 12). The counts from different edges are summed to obtain the final result.

```
def Decomine(edges):
  G = mk_graph(edges)

  # select vertices of a certain label from S
  def select(S, label)
    return apply(lambda v: v, cond=lambda v: vertex_label(G, v) == label,
        S)
  def f(e):
    S0 = neighbor(G, get(e, 0))
    S1 = neighbor(G, get(e, 1))
    C2 = select(intersect(S0, S1), 'a')
    S3 = select(S0, 'd')
    return size(C2) * size(S3)

  # iterate over edges that match (u0, u1)
  return sum(apply(func=f, cond=lambda e: vertex_label(G, get(e, 0)) ==
     'a' and vertex_label(G, get(e, 1)) == 'c', edges))
```

Listing 2. Decomposition-based algorithm implemented in Matcha for the query in Fig. 29(b).

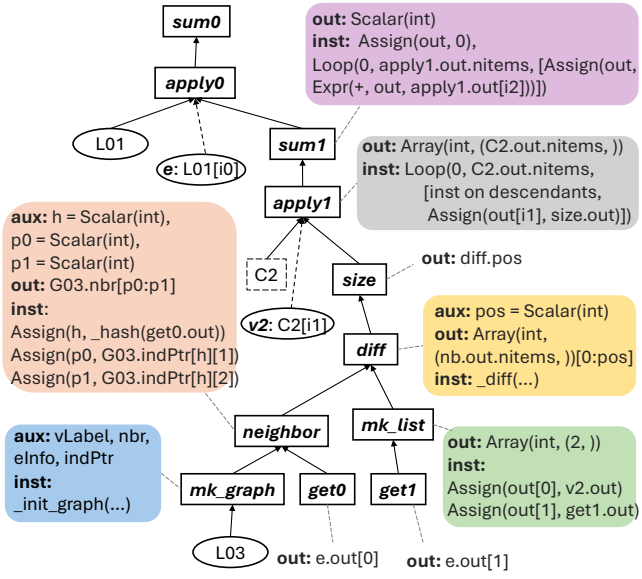
3.5 Matcha Compilation Pipeline

A Matcha program can be depicted as an Abstract Syntax Graph (ASG). In this graph, nodes represent data objects and operators, while edges connect each operator to its inputs. The Matcha compiler generates an intermediate representation (IR) from the ASG. The IR may go through several optimization stages before converting into C++/CUDA code.

Fig. 30 shows an example of the compilation pipeline. The program in Listing 1 is depicted as an ASG in Fig. 30(a). The `apply` operator on the top (`apply0`) has $f1$ as the computation function and $L01$ as the input `List`. The items in $L01$ are referenced by $f1$ through an operand e attached to the `apply` operator. Within the subtree of the function $f1$, the inner apply operator (`apply1`) has $f2$ as the computation function and $C2$ as the input `List`. The function $f2$ accesses items in $C2$ through an operand $v2$ attached to `apply1`. For a simple illustration, we omit the subtree for computing $C2$ in the figure.

3.5.1 Generating Ir On Asg

The compiler traverses the ASG in a depth-first order to generate the IR on each node. This ensures that the IR on any node is generated after the IR on all its input nodes has been generated.



(a) ASG to IR

```
int RapidMatch(..., int L03[][2],
    int len3, ...) {
 // Memory Pre-allocation
 auto G03_nbr = new int[len3];
 auto G03_indPtr = new int[H][3];
 int nb_h, nb_p0, nb_p1;
 auto lst = new int[2];
 int df_pos;
 auto df_out = new int[_max_degree(L03)];
 auto apl1_out = new int[_max_degree(L01)];
 int sum1;
 ...
 for (int i0 = 0; i0 < len1; i0++) {
  ...
  for (int i1 = 0; i1 < C2_size; i1++) {
    _init_graph(L03, len3, G03_nbr,
                G03_indPtr);
    nb_h = _hash(L01[i0][0]);
    nb_p0 = G03_indPtr[nb_h][1];
    nb_p1 = G03_indPtr[nb_h][2];
    lst[0] = C2[i1];
    lst[1] = L01[i0][1];
    _diff(df_out, df_pos, G03_nbr,
          nb_p0, nb_p1, lst);
    apl1_out[i1] = df_pos; }
  sum1 = 0;
  for (int i2 = 0; i2 < C2_size; i2++) {
    sum1 = sum1 + apl1_out[i2];} }...}
```

(b) Generated C++ code

Figure 30. Compiling a Matcha program to C++ code. The IR of each ASG node contains an output (out), some auxiliary data (aux), and instructions (inst) for computing the output.

Upon visiting a node, the compiler determines the output of the operator as well as instructions for computing the output.

The IR has two data types: `Scalar` and `Array`. A `Scalar` can be integer, floating-point, or boolean. An `Array` can be one-dimensional or multi-dimensional and is defined by its data type and the size of each dimension. Each element in an `Array` is a `Scalar` and can be accessed through array `Indexing`. To represent arithmetic, comparison, and logical expressions, the IR has an `Expr(op, lhs, rhs)` construct where the *op* is the operation type, and the *lhs*/*rhs* can be a `Scalar`, an `Expr`, or a constant literal. The IR also has an `Assign(dest, src)` construct for representing assignments. Due to space limit, we only describe the generation rules for the `Graph` operators in this section. The rules for other operators are straightforward as illustrated in Fig. 30.

When creating a `Graph` object, we construct an auxiliary data structure based on the input *edges*, *vlabel*, and *einfo*. The data structure helps achieve efficient access to neighboring information in the graph. Specifically, upon visiting a `mk_graph` node on the ASG, the compiler generates

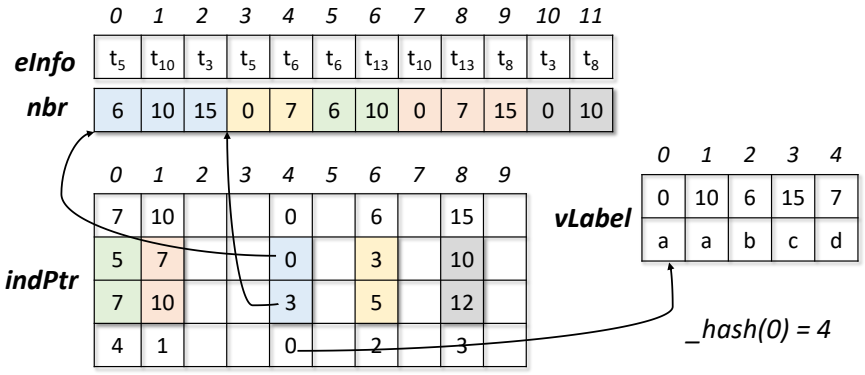


Figure 31. Graph G from Fig. 29(a) stored in four `Arrays`.

four auxiliary `Arrays` (*vLabel*, *nbr*, *eInfo*, *indPtr*). The *vLabel* is a copy of the input *vlabel* (if the information is provided); it is a 2-D array that pairs unique vertex IDs with their labels. The *nbr* is a 1-D array that holds the neighbors of vertices corresponding to the vertex IDs in *vLabel*. The neighbors of each vertex are stored contiguously in *nbr*. The *eInfo* Array has the same size as *nbr*, and it contains information on the neighboring edges corresponding to vertices in *nbr*. Fig. 31 shows an example of the data structure for the graph G in Fig. 29(a).

Depending on whether the vertex IDs are dense, we build different indirection arrays to achieve an efficient lookup of vertices' neighbors. If the vertex IDs are contiguous natural numbers, we use a 1-D Array (*indPtr*) to store the boundaries of their neighbors in *nbr*. The starting position of v's neighbors in *nbr* is $indPtr[v]$. This data structure is exactly the Compressed Sparse Row (CSR) format commonly used for storing graph data.

However, for edges stored as relational tables, which are commonly used in join-based algorithms [77], the vertex IDs are non-contiguous and sparse. To efficiently locate neighbors for the vertices, we store the neighbor boundaries in a hash table. Specifically, the *indPtr* is a 2-D Array of size ($H \times 4$) where H is the number of slots in the hash table. For a vertex v, its neighbor boundaries are stored in $indPtr[\_hash[v]][1]$ and $indPtr[\_hash[v]][2]$, and its position in *vLabel* is stored in $indPtr[\_hash[v]][3]$. In Fig. 31, the neighbors of vertex-0 are stored from position-0 ($indPtr[\_hash(0)][1]$) to position-3 ($indPtr[\_hash(0)][2]$) in *nbr*. The initialization of the data structure is performed by a built-in function (*_init_graph*), which is invoked on the `mk_graph` node.

The `neighbor`, `edge_info`, and `vertex_label` operators can be implemented efficiently with *indPtr*. The output of `neighbor/edge_info` is a slice of the graph's *nbr*/*eInfo* Array. The output of `vertex_label` operator is a `Scalar` read from *vLabel*. If the graph is stored in CSR format, the slicing boundaries for vertex v are $indPtr[v]$ and $indPtr[v+1]$; otherwise, the slicing boundaries are obtained from the hash table.

3.5.2 Translating Ir To C++ Code

The IR needs to undergo several optimization passes before being translated to the target code. We leave the details of code optimizations/parallelization in the next section and focus on generating sequential CPU code in this section.

After the IR is generated, the compiler takes another depth-first traversal on the ASG and generates a C++ function based on the IR. The leaf nodes (i.e., nodes without any input) are input data (e.g., $L01$, $L03$ in Fig. 30(a)). These data are initialized by the host code and are passed as arguments to the C++ function. In our implementation, the input data are initialized in Python, and the generated C++ code is compiled just-in-time into a pybind11 [36] module, which is invoked by the Python program.

Memory Pre-allocation. If an internal node on the ASG contains an `Array`, we need to allocate memory for the data. A straightforward approach is to allocate memory on the node. However, this involves dynamic memory allocation and may incur a large overhead, especially when it is in a loop. For example, the `intersect` operator in Listing 1 needs to allocate memory for the intersection results every time $f1$ is executed (i.e., for every item in $L01$). To avoid repeated memory allocation, we pre-allocate memory for `Arrays` on all ASG nodes at the beginning of the generated code. Specifically, the compiler checks the size of every `Array` in the internal nodes. If the size is constant or can be determined based on the data in the leaf nodes, we can pre-allocate the memory before execution of the internal nodes. If the size is unknown, the compiler reports an error and asks the user to provide a size.

Once memory is allocated, generating computing instructions is straightforward. The `Expr`,

`Loop`, and `Assign` constructs in the IR can be directly translated to expressions, for-loops, and assignments in C++. If a node is not a descendant of an `apply` operator, its *inst* is immediately translated when the node is visited. However, if it is a descendant of an `apply` operator, the node is skipped since its instructions should be included in the loop body of the `apply` operator. Fig. 30(b) shows the generated C++ code corresponding to the IR in Fig. 30(a).

3.6 Optimization And Parallelization

Matcha decouples the definition of subgraph matching algorithm from its execution strategy. This section explains how various optimization and parallelization strategies can be applied automatically to Matcha programs through IR transformations.

3.6.1 Operator Fusion

Our compiler fuses four types of operator pairs:

- `apply+sum`. Consider the loop of `apply1` (colored in grey) and the loop of `sum1` (colored in purple) in Fig. 30(b). The two loops share the same iteration space. The `apply1` loop writes data to $apl1\_out$, which is read by the `sum1` loop. We can fuse the two loops to avoid the storage of $apl1\_out$.

- `intersect/diff+sum`. In Fig. 30(a), we can also see that the output array of `diff` operator is never used; the algorithm is only interested in the number of items in the array. In this case, we can remove the allocation of df_out and invoke a more efficient version of $\_diff$ without storing the output data.

- `neighbor+apply` with boundary conditions. This pair of operators often appear in labeled subgraph matching algorithms, as they need to select neighboring vertices with a specific label (e.g., line 11 in Listing 2). By fusing the two operators, we can avoid the allocation of a new `Array` for the output of `apply` and replace it with a slice of the neighbor list.

- `intersect/diff+apply` with boundary conditions. An `intersect/diff` operator can also be fused with a conditional `apply` if the condition only affects the slicing boundaries of the output. This fusion enables the efficient implementation of the symmetry-breaking technique that eliminates redundant subgraphs [53].

The fusion is applied to the ASG nodes in a depth-first order. For each pair of child-parent nodes, we check if they match any fusible types and perform fusion on the IR accordingly.

3.6.2 Loop-invariant Code Motion

Consider the *_init_graph* function in the `apply1` loop in Fig. 30(b). The function's execution does not depend on the loop iterate $i1$, which means that it can be moved outside of the loop. In fact, the function does not depend on the loop iterate $i0$ either and can be moved outside of the `apply0` loop. This optimization is often referred to as loop-invariant code motion in compiler optimization literature.

Besides code motion at the ASG level, there might also be loop-invariant instructions at the IR level. For example, the `mk_list` node in Fig. 30(a) contains two `Assigns`, and the second `Assign` does not reference the items of $C2$. In such cases, we only include the first `Assign` in the loop body of `apply1` and leave the second `Assign` within the `mk_list`.

This optimization also achieves the code motion of set operations proposed in previous work [55]. For any `intersect` or `diff` operators independent from the input of the `apply` operator, the compiler automatically moves the code outside the loop. Listing 3 shows the optimized code after operator fusion and code motion for the example in Fig. 30.

3.6.3 Automatic Parallelization

Following previous work [20, 90], we apply two-level parallelization to the subgraph exploration procedure. The first parallelization is applied to the outermost loop that iterates over the initial vertices/edges. On a CPU, this corresponds to using multiple threads to explore subgraphs from different vertices/edges simultaneously. The second parallelization is applied to the compu-

```
int RapidMatch(..., int L03[][2]) {
 ...
 int sum1;
 ...
 //_init_graph is moved out of i0 loop after code motion.
 _init_graph(L03, len3, G03_nbr, G03_indPtr);
 for (int i0 = 0; i0 < len1; i0++) {
     ...
     //The four statements are moved out of i1 loop after code motion.
     nb_h = _hash(L01[i0][0]);
     nb_p0 = G03_indPtr[nb_h][1];
     nb_p1 = G03_indPtr[nb_h][2];
     lst[1] = L01[i0][1];
     sum1 = 0;
     //i1 loop is fused into i2 loop.
     for (int i2 = 0; i2 < C2_size; i2++) {
          lst[0] = C2[i2];
          _diff(df_out, df_pos, G03_nbr,
          nb_p0, nb_p1, lst);
          sum1 = sum1 + df_pos; }
     apl0_out[i0] = sum1; } }
```

Listing 3. Optimized code after operator fusion and code motion for the example in Fig 30.

tation of intermediate subgraphs at each exploration step. It exploits fine-grained parallelism (i.e., SIMD) to accelerate the `intersect/diff` operations.

GPU parallelization is similar. Unlike previous work that uses either a warp [20, 90] or a single thread [95] as the execution unit, we use cooperative thread groups [30] on Nvidia GPU to execute iterations of the outermost loop in parallel. This implementation allows us to configure the thread-group size and achieve better GPU utilization. For `intersect/diff`, we adapt the code from G2Miner [20] to use threads in each group for parallel execution.

Work Stealing. Subgraph matching on a GPU usually suffers from severe load imbalance [90]. Work-stealing is an effective technique that can mitigate this issue and significantly improve performance. We implement the work-stealing strategy from [90] in our compiler. Specifically, we place the *aux* in the IR of `neighbor`, `intersect` and `diff` operators, as well as the iterators of all `apply` loops in GPU shared memory. This allows an idle thread-group to check the amount of remaining work in another thread-group and steal work from it.

3.7 Evaluation

3.7.1 Experimental Setup

Platform. Our experiments are conducted on a dual-socket machine with Intel Xeon Gold 6226R 2.9GHz CPUs (32 cores in total), 512GB RAM, and an Nvidia A100 GPU. The generated CPU code was compiled using GCC 9.4.0, and the generated GPU code was compiled using NVCC 11.2.

Subgraph Matching Tasks. We evaluate the performance of four representative subgraph matching tasks: k-motif counting (k-MC), subgraph listing (SL), k-clique counting (k-CL), and temporal subgraph listing (TSL). A more detailed description of the tasks can be found in Section 3.2.1.

Datasets. Table 17 lists the data graphs used in our experiments. Six static graphs are used for k-MC, SL, and k-CL, and three temporal graphs are used for TSL. These graphs are commonly used to evaluate subgraph matching systems in previous work [16, 17]. For labeled SL, we randomly assign five labels to the vertices in the graphs.

Table 8. Graph datasets.

Graph	# nodes	# edges	Max degree
MiCo (mc)	96K	1.1M	1359
DBLP (db)	317K	1.0M	343
Amazon (az)	334K	0.9M	549
YouTube (yo)	1.1M	3.0M	28754
Patent (pt)	3.8M	16.5M	793
LiveJournal (lj)	4.0M	34.7M	14815
EmailEuCore (ee)	1K	332K	9782
wikitalk (wt)	1.1M	7.8M	264905
StackOverflow (st)	2.6M	63.5M	101663

Compared Systems. We compare our system against five state-of-the-art systems: STMatch (**STM**) [90], RapidMatch (**RM**) [77], DecoMine (**DM**) [17], G2Miner (**GM**) [20], and Everest (**EV**) [95]. These systems were chosen for their diverse algorithm and optimization supports. GM and STM implement the vertex-extension backtracking algorithm for general subgraph listing and motif counting tasks. DM employs a pattern-decomposition-based algorithm for subgraph counting. EV implements an edge-extension backtracking algorithm for temporal subgraph listing. RM

Table 9. Execution time of motif counting with vertex-extension algorithm. 'ms', 's', and 'm' stand for milliseconds, seconds, and minutes.

		mc	db	az	yo	pt	lj
3-MC	**GM**	2.6ms	1.2ms	1.0ms	18ms	23ms	142ms
	mGM	2.7ms	1.2ms	1.1ms	18ms	23ms	141ms
	mGM-o	1.7ms	0.6ms	0.4ms	13ms	16ms	136ms
4-MC	**GM**	1.58s	0.11s	0.04s	227s	1.93s	415s
	mGM	1.57s	0.13s	0.05s	227s	1.93s	415s
	mGM-o	1.02s	0.06s	0.02s	192s	1.78s	407s

Table 10. Execution time of motif counting with a decomposition-based algorithm.

		mc	db	az	yo	pt	lj
4-MC	**mDM-cpu**	0.39s	64ms	43ms	0.74s	1.6s	33s
	mDM-gpu	0.03s	3ms	2ms	0.06s	0.09s	2.3s
5-MC	**mDM-cpu**	128s	1.3s	0.17s	576s	21s	212m
	mDM-gpu	22s	0.3s	0.04s	17s	3.3s	26m

employs a join-based algorithm for labeled subgraph listing. GM, STM, and EV are GPU-based systems, while DM and RM only run on CPU.

To demonstrate Matcha's flexibility and efficiency, we reimplement the five systems in Matcha and compare performance with their original implementations. These reimplementations (referred to as **mGM**, **mDM**, **mSTM**, **mRM**, and **mEV**) use the same optimizations as original systems. Then, we improve the performance of the reimplementations by enabling various optimizations in our compiler. For the two CPU-only systems (DM and RM), we also parallelize the code for GPU execution.

3.7.2 Results for Motif Counting

We use DC and GM for motif counting as they reportedly achieve the best performance among existing systems for the task on CPU and GPU, respectively.

Table 9 lists the execution time of 3-MC and 4-MC with the vertex-extension algorithm of GM running on GPU. The results show that our reimplementation achieves almost the same performance as the original GM system, validating the efficiency of our generated code. We further optimize the baseline reimplementation by using a group of 8 threads to execute each iteration of

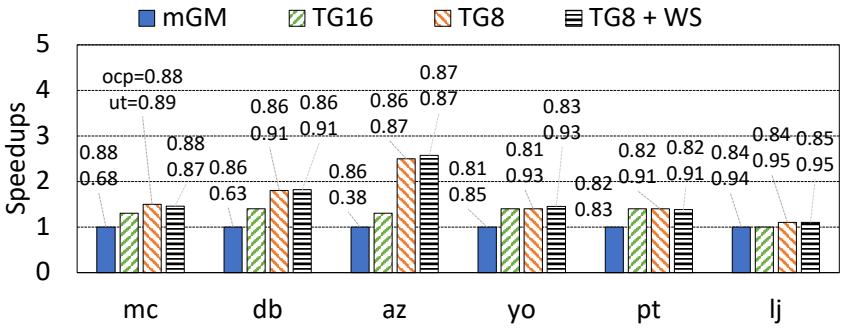
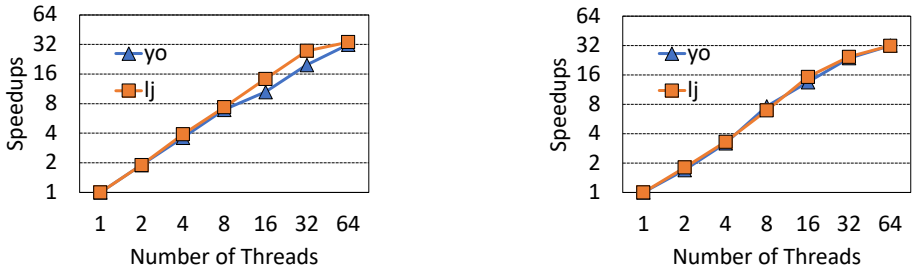


Figure 32. Effect of thread-grouping (TG) and work-stealing (WS) on mGM for 3-MC. 'ocp', 'ut' stand for SM occupancy and thread utilization.



(a) 4-MC (b) 5-MC

Figure 33. Scalability of multi-threaded mDM on CPU.

the outermost loop. We also enable work-stealing among the thread groups. (The original GM system uses warps for parallel execution without any work-stealing.) With the optimizations, our generated code (mGM-o) achieves 1.1x to 2.6x speedups (with an average of 1.5x) against the original GM.

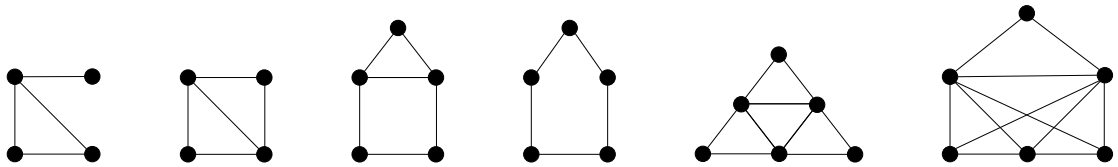
Table 10 lists the execution time of 4-MC and 5-MC with the pattern-decomposition algorithm of DM. Since DM is not open-sourced, we cannot directly compare performance with their original implementation. The execution time of our re-implementation on CPU is close to the numbers reported in their paper, affirming the efficiency of our generated code. While the original DM is CPU-only, our system automatically generates GPU code for the algorithm, easily accelerating the execution by 4x to 53x (with an average of 27x).

By activating different optimization passes during Matcha compilation, we study the effectiveness of each optimization technique. Fig. 32 shows how the performance of mGM is affected by thread-grouping and work-stealing. We profile the execution with Nvidia NSight [1] and mark the SM occupancy and thread utilization data in the figure. Occupancy is defined as the ratio of

active warps on an SM to the maximum number of active warps supported. Thread utilization is defined as the ratio of active threads in a warp to the total number of threads in the warp. As we can see from the figure, thread-grouping effectively improves thread utilization, bringing 1.1x to 2.5x speedups against the baseline mGM. On the other hand, work-stealing has little effect on thread utilization and SM occupancy. This is because the nested loop has only two levels for 3-MC, and the baseline mGM already achieves good load balance with dynamic scheduling of the outermost loop.

For DM, we study how the algorithm scales on a CPU with different numbers of threads. Fig. 33 shows that our generated code achieves near-linear speedups up to 32 threads. When the thread count increases to 64, performance still improves due to the CPU's hyper-threading.

3.7.3 Results for Subgraph Listing



(a) Q1 (b) Q2 (c) Q3 (d) Q4 (e) Q5 (f) Q6

Figure 34. Query patterns for subgraph listing.

GM, STM, and RM all support general subgraph listing. We test and compare their performance with our Matcha implementation using the query patterns in Fig. 34.

Performance of Unlabeled Queries. Table 11 shows the execution time of different systems for listing unlabeled subgraphs of the query patterns. In all test cases, our re-implementation of STM (mSTM) either matches or surpasses the performance of the original STM system. This is because the original STM needs to maintain a stack data structure to simulate the recursive execution of the backtracking algorithm, while Matcha directly translates the algorithm into a nested loop. The performance advantage of mSTM diminishes with large query patterns because the overhead of maintaining the stack data structure becomes small compared to the computation itself.

GM performs especially well on small queries but is slower than STM for larger queries. This is because GM launches many more thread-blocks (7000+) than STM (82) and achieves

Table 11. Execution time of unlabeled subgraph listing with vertex extension. 's' stands for seconds. All other numbers are in milliseconds. '−' indicates timeout after 4 hours.

Graph		Q1	Q2	Q3	Q4	Q5	Q6
az	**STM**	12	14	71	80	138	132
	mSTM	10	5.4	71	16	131	131
	GM	2.6	2.7	41	8.4	132	164
	mGM	2.5	2.7	41	8.4	132	164
	mGM-o	1.3	1.0	18	3.7	131	76
db	**STM**	312	147	903	368	14.6s	29.4s
	mSTM	110	39	901	367	14.6s	28.7s
	GM	26	12	1220	388	67.0s	95.6s
	mGM	26	14	1232	386	67.0s	95.6s
	mGM-o	15	5.3	457	138	12.7s	20.5s
mc	**STM**	445	235	122s	30.3s	–	–
	M-STM	449	229	121s	30.0s	–	–
	GM	769	315	178s	35.6s	–	–
	mGM	774	317	178s	36.6s	–	–
	mGM-o	407	227	121s	29.3s	–	–

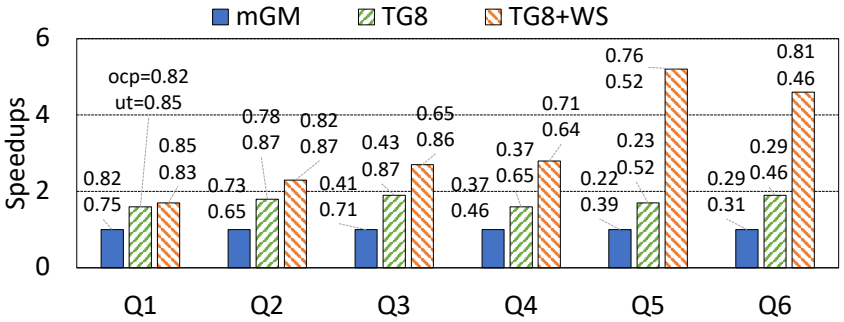


Figure 35. Effect of thread-grouping (TG) and work-stealing (WS) on mGM for subgraph listing on DBLP graph.

better GPU occupancy for small tasks, while STM achieves better load balance for larger tasks due to its work-stealing technique. Our re-implementation of GM (mGM) achieves almost the same performance as the original GM.

To further improve the performance, we apply thread-grouping and work-stealing to mGM. The optimized code (mGM-o) brings the best of both STM and GM, achieving up to 7.7x speedups against the original STM and 4.6x speedups against GM. Fig. 35 shows the effectiveness of each optimization. As expected, thread-grouping increases thread utilization, leading to 1.5x to 2.8x speedups against mGM. The benefit of work-stealing is small for the size-4 queries (Q1 and Q2), which is similar to the results for 3-MC in Fig. 32. However, as the pattern size grows (Q3∼Q6),

Table 12. Execution time (in milliseconds) of labeled subgraph listing with vertex extension.

Graph		Q1	Q2	Q3	Q4	Q5	Q6
az	**STM**	12	12	12	10	9.5	6.9
	mGM-o	0.4	0.4	0.9	3.3	1.0	0.6
db	**STM**	12	12	14	19	61	83
	mGM-o	1.4	1.5	7.9	8.2	59	58
mc	**STM**	8.5	8.4	366	612	1371	1197
	mGM-o	5.3	5.1	272	383	1277	1183

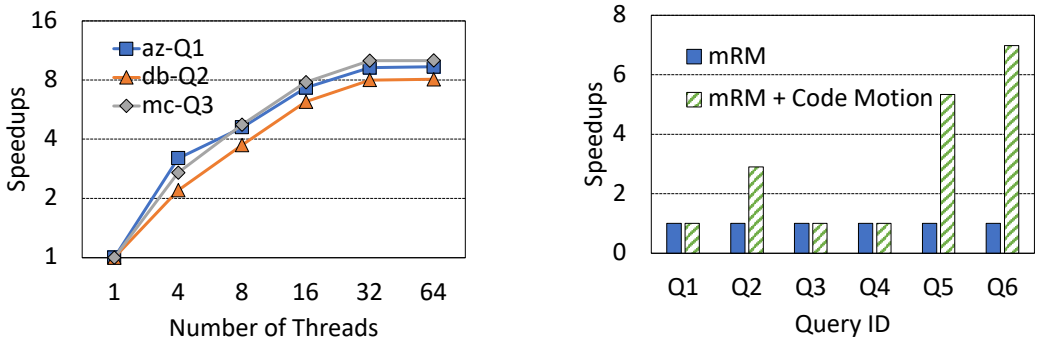
Table 13. Execution time of labeled subgraph listing with a join-based algorithm. 's' stands for seconds. All other numbers are in milliseconds.

Graph		Q1	Q2	Q3	Q4	Q5	Q6
az	**RM**	60	67	75	95	90	96
	mRM	12	14	12	18	18	17
	mRMo-cpu	1.3	1.6	1.6	2.9	1.6	1.6
	mRMo-gpu	0.1	0.1	0.3	1.2	0.2	0.2
db	**RM**	71	78	205	360	2262	1863
	mRM	13	17	66	130	971	1388
	mRMo-cpu	1.8	1.0	14	16	260	292
	mRMo-gpu	0.6	0.5	7.8	8.3	56	59
mc	**RM**	165	223	9688	18.1s	257s	207s
	mRM	62	136	7983	16.7s	239s	198s
	mRMo-cpu	8.5	18	797	1.1s	28s	28s
	mRMo-gpu	5.4	4.8	267	0.33s	1.25s	1.10s

the benefit becomes more noticeable. This performance is reflected by the SM occupancy. For Q1 and Q2, occupancy is already high without work-stealing. But for Q3∼Q6, work-stealing significantly increases occupancy.

Performance of Labeled Queries. Table 12 shows the performance of the backtracking algorithm for labeled subgraph listing. The queries are generated by randomly assigning four labels to the pattern vertices. Compared to STM, our code (mGM-o) is consistently faster for all queries, achieving an average speedup of 7.1x.

We also test the performance of the join-based algorithm of RM for the same set of labeled queries. The results are listed in Table 13. The original RM is a single-threaded CPU implementation. We reimplement it in Matcha and apply two optimizations, code motion and multi-threading, to it. The generated CPU code (mRMo-cpu) is on average 29.4x (up to 60x) faster than the original RM. Fig. 36 shows how each of the two optimizations affects the performance. For Q2, Q5, and



(a) Multi-threading (b) Code motion on DBLP graph

Figure 36. Effect of code motion and multi-threading on RM for labeled subgraph listing.

Table 14. Execution time (in milliseconds) of labeled subgraph counting with decomposition-based and join-based algorithms.

Graph		Q1	Q2	Q3	Q4	Q5	Q6
az	**mDM-cpu**	11	11	1	11	11	11
	mDM-gpu	0.3	0.3	0.5	3.2	0.4	0.3
	mDM-RM	0.1	0.1	0.1	2.9	0.3	0.4
db	**mDM-cpu**	16	16	16	18	15	16
	mDM-gpu	0.3	0.3	1.0	8.2	0.6	0.7
	mDM-RM	0.1	0.1	0.3	8.2	0.5	0.7
mc	**mDM-cpu**	12	13	30	1056	57	14
	mDM-gpu	0.4	0.3	8.4	263	2.7	4.5
	mDM-RM	0.1	0.1	3.3	259	2.4	4.3

Q6, code motion significantly reduces the redundant set operations, achieving up to 6.9x speedup. It is not effective for Q1, Q3, and Q4 because the three queries do not have redundant set operations. As more threads are used for execution, the program runs faster, but the speedups are small. This is because the total workloads in these labeled queries are small.

We further generate GPU code for RM (with code motion activated). The GPU code (mRMo-gpu) runs 1.6x to 16x faster than mRMo-cpu. Interestingly, when comparing data from Table 12 and Table 13, we note that the performance of mRMo-gpu closely matches that of mGM-o for most tasks. This observation casts doubt on the practical advantage of the join-based algorithm over the basic vertex-extension on GPU.

Performance of Subgraph Counting. In some cases, the user might be only interested in the count of subgraphs without listing them. The decomposition-based algorithm in DM is most suitable for

this task. For each sub-pattern, DM simply uses the basic vertex-extension backtracking algorithm to count the matching subgraphs. One potential way to improve the performance is to adopt RM to match the sub-patterns. While the idea is hard to implement in the original DM or RM system, the implementation is easy in Matcha. Table 14 shows the execution time of labeled subgraph counting with our re-implementation of DM on both CPU and GPU, and the combined algorithm (mDM-RM). We can see that Matcha easily accelerates DM by 2.2x to 53x with GPU parallelization. The combined algorithm further improves the performance on GPU by 1.8x on average.

3.7.4 Results for Clique Counting

Next, we compare GM and Matcha for 4, 5, 6-CL. We select GM as baseline because it accelerates the algorithm with an edge-pruning technique and reportedly achieves the best performance for the task among existing systems.

Table 15. Execution time of clique counting with the backtracking algorithm accelerated by edge pruning. 's' stands for seconds. All other numbers are in milliseconds.

		mc	**db**	**az**	**yo**	**pt**	**lj**
4-CL	**GM**	15	1.5	0.6	2.8	11	271
	mGM-o	11	0.8	0.2	1.4	8.9	271
5-CL	**GM**	592	11	0.7	5.5	13	9341
	mGM-o	551	11	0.3	4.5	10	9338
6-CL	**GM**	22.6s	235	0.7	10	17	466s
	mGM-o	20.3s	171	0.4	8.7	14	466s

Table 15 shows the execution time of the original GM system and our generated code. Our code (mGM-o) is optimized with thread-grouping and work-stealing, achieving 1.1x to 2.6x speedups with an average of 1.4x against the original GM. According to our profiling, the speedups are mainly attributed to the improved thread utilization by thread grouping. We omit the profiling data here due to the space limit.

3.7.5 Results for Temporal Subgraph Listing

We use Matcha to re-implement the edge-extension backtracking algorithm of EV and compare the performance with the original EV system.

(a) Q7 (b) Q8 (c) Q9 (d) Q10

Figure 37. Temporal Queries.

We use the four temporal queries in Fig. 37 for performance evaluation. The numbers on the edges indicate the matching order. We adopt the same setting as in the EV paper [95] and set the time constraint δ to one day (86400 seconds).

Table 16 shows the execution time of different queries with the original EV system and our generated code (mEV). Our code achieves almost the same performance as the original EV. This is because the original EV implementation already incorporates all the applicable optimizations listed in Table 7. Code motion is not helpful in this case because the edge-extension algorithm does not have any set operation, and symmetry breaking cannot be applied to temporal graphs. The results confirm the efficiency of our automatically generated code in comparison to the hand-optimized code.

Table 16. Execution time (in milliseconds) of temporal subgraph listing with edge-extension based backtracking.

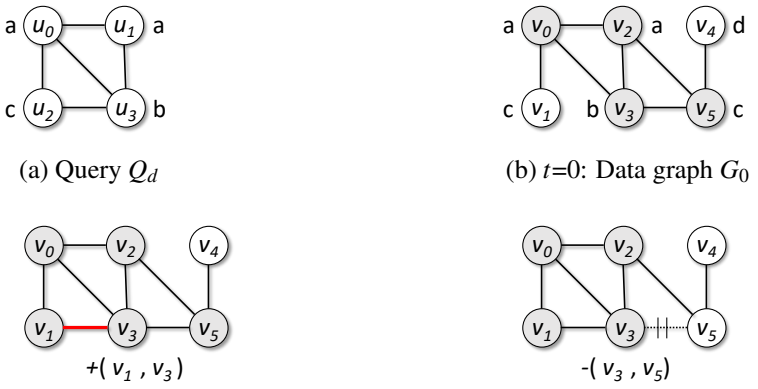
		Q7	**Q8**	**Q9**	**Q10**
eu	**EV**	1.8	4.8	4.6	3.8
	mEV	1.9	5.2	4.5	4
wk	**EV**	11	21	22	20
	mEV	11	22	23	19
st	**EV**	97	190	217	185
	mEV	97	185	193	188

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

4.1 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 [67]. It can also be applied to monitor money laundering in transaction networks [69], trace rumor propagation paths in social networks [87], and identify system anomalies in computer communication networks [51].

Figure 38 illustrates an example of CSM. Given an initial data graph G_0 at time $t = 0$ and a query pattern Q_d, G_0 contains one matching subgraph (v_0, v_2, v_3, v_5). At $t = 3$, an edge (v_1, v_3) is added to G_0, the updated graph G_1 produces an additional match (v_0, v_1, v_2, v_3) for the query. At $t = 5$, the edge (v_3, v_5) is deleted from the graph, which invalidates the previous match (v_0, v_2, v_3, v_5).



(a) Query Q_d

(b) t=0: Data graph G_0

(c) t=3: Data graph G_1

(d) t=5: Data graph G_2

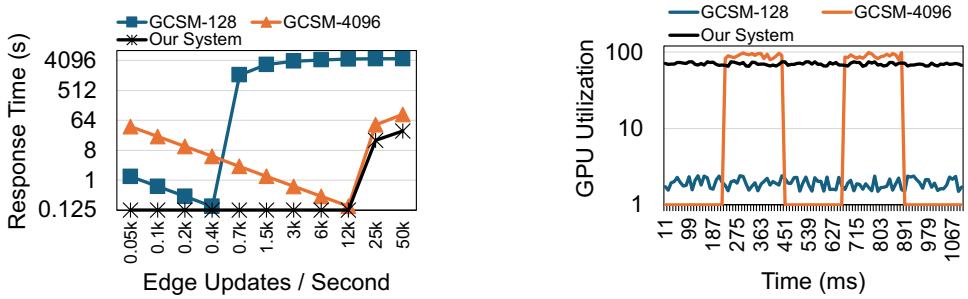
Figure 38. An example of continuous subgraph matching. G_{k+1} is the data graph after applyting update on G_k. Q_d is the query of "diamond" structure. *abcd* next to the vertices are the vertex labels.

Many efforts have been made to improve the performance of CSM on CPUs, primarily by reducing the search space during the matching process [41, 48, 59, 78]. However, due to the ex-

ponential time complexity of subgraph matching [82], CSM remains a computationally expensive procedure. To address this limitation, recent systems [68, 91] have begun exploiting the massive parallelism of GPUs to accelerate CSM.

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

To validate this point, we tested the state-of-the-art GPU-based CSM system (GCSM [91]) under different batch sizes. The performance results are presented in Figure 39a. 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 39b.



(a) Response Time (b) GPU Utilization

Figure 39. Average response time and GPU utilization of GCSM. GCSM-x denotes GCSM processing with batch size x. **Query:** Q4 (Figure 47). **Graph:** Unlabeled Netflow [13].

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 39, our system maintains short response times across a wide range of update rates and achieves higher average GPU utilization than GCSM.

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

We evaluate our system against state-of-the-art CPU and GPU systems. The experiments demonstrate that DCSM ensures optimal response time and throughput across different update rates. Compared to GCSM [91], 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 [78].

4.2 Background

4.2.1 A Formal Definition of Csm

Definition 4 (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 5 (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 6 (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 7 (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 8 (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 9 (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 $\Delta e_k \in \Delta G$.

4.2.2 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 [68, 91] 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 [91] stores G_k in the CPU's main memory, whereas GAMMA [68] stores G_k in the GPU's global memory.

4.3 Overview of Dcsm

Figure 40 provides an overview of our DCSM system, which consists of three modules: a graph updater (GU), an executor (EX), and a garbage collector (GC).

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

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

Figure 41 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 [91] with a

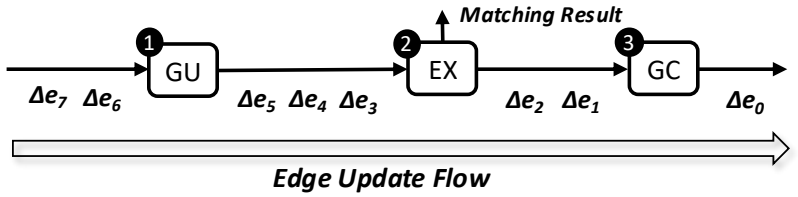


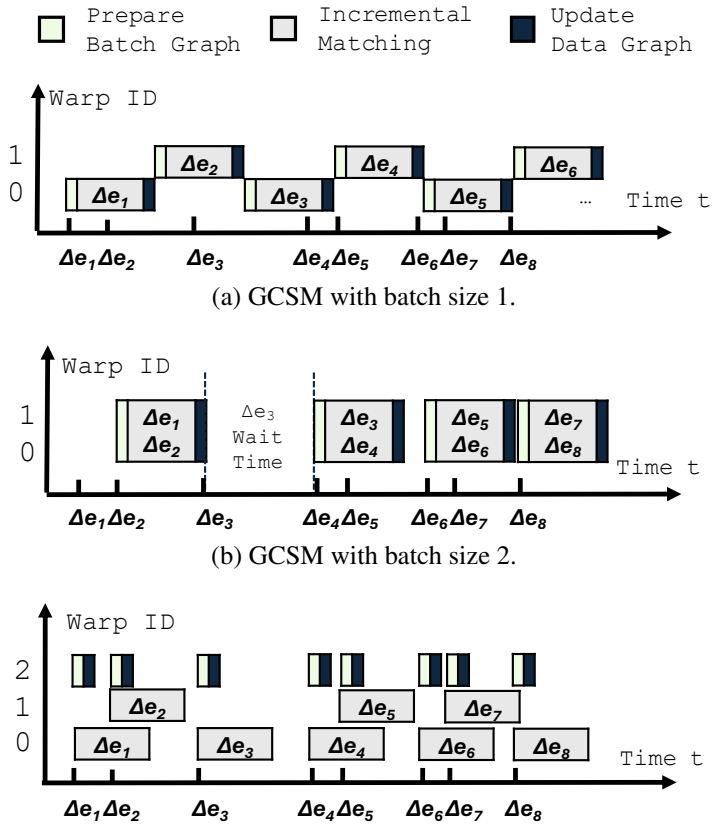
Figure 40. Overview of DCSM. DCSM consists of three functional modules: Graph Updater (GU), Executor (EX), and Garbage Collector (GC). The edge updates Δe_k flows through these modules sequentially.

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

The rest of the paper is organized as follows. Sections 4.4, 4.5, and 4.6 describe the three functional modules—GU, EX, and GC, respectively. Section 4.7 presents the evaluation results.

4.4 Graph Updater And Multi-version Graph

All existing CSM systems [68, 91] on GPUs are derived from static subgraph matching frameworks [20,90], and they fail to address data race problems that arise when processing multiple batches concurrently. As a result, these systems are forced to handle n batches (B_1, ..., B_n) in a strictly sequential manner: $P(B_1) \rightarrow M(B_1) \rightarrow U(B_1) \rightarrow \ldots \rightarrow P(B_n) \rightarrow M(B_n) \rightarrow U(B_n)$, where P, M, and U represent the Prepare, Match, and Update phases in Section 4.2.2. This serialization stems from the sort operation in the Update phase: executing $U(B_k)$ concurrently with $M(B_k)$ would introduce data races, and $M(B_{k+1})$ must be deferred until $U(B_k)$ finishes, because $M(B_{k+1})$ depends on the data graph state produced by $U(B_k)$. To overcome this limitation, we introduce a multi-version graph data structure for CSM.



(a) GCSM with batch size 1.

(b) GCSM with batch size 2.

(c) DCSM with inter-batch parallelism.

Figure 41. 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.

4.4.1 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 42 shows an example; its caption provides an explanation. This design fully considers the efficiency of the matching algorithm. To satisfy the matching efficiency, each neighbor array is sorted by vertex ID. To enable dynamic extension and shrinkage, slab objects of each vertex are linked together as a list. To avoid the data race problem mentioned above, we maintain multiple neighbor array versions for each vertex. Therefore, inserting a batch of edges in the data graph will not cause data races with the ongoing matching procedures since the newly inserted edges reside in newly created versions of the neighbor arrays.

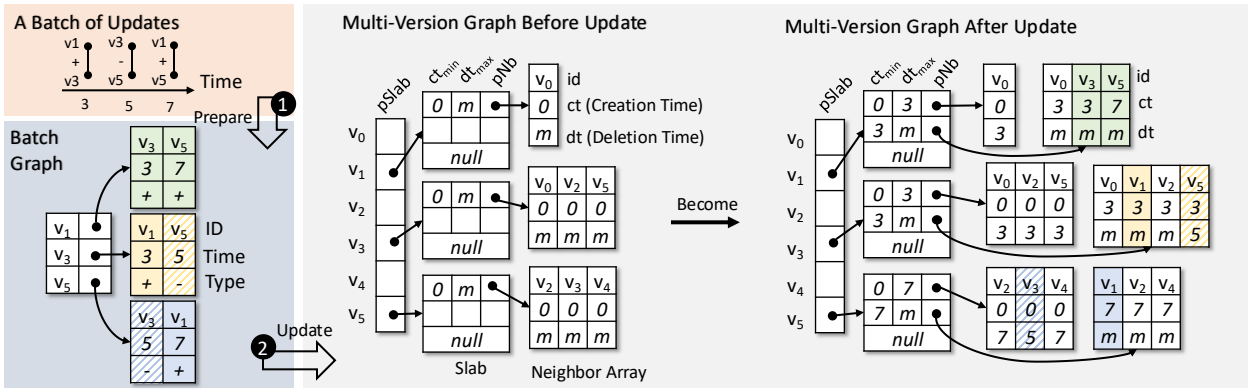


Figure 42. The multi-version graph (MVG) data structure and the procedure for updating it with a batch. The MVG before update represents G_0 in Figure 38. The vertex ID indexes the *pSlab*, pointing to a *Slab*. Each row of the *Slab* corresponds to a neighbor array version, storing its creation time (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.

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

4.4.2 Update Procedure

Figure 42 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 42. To update v_5's neighbor array in the MVG, we first update the deletion time of v_3 to 5, then create a new neighbor array version and insert v_1 into it.

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

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

4.4.3 Parallel Graph Updater

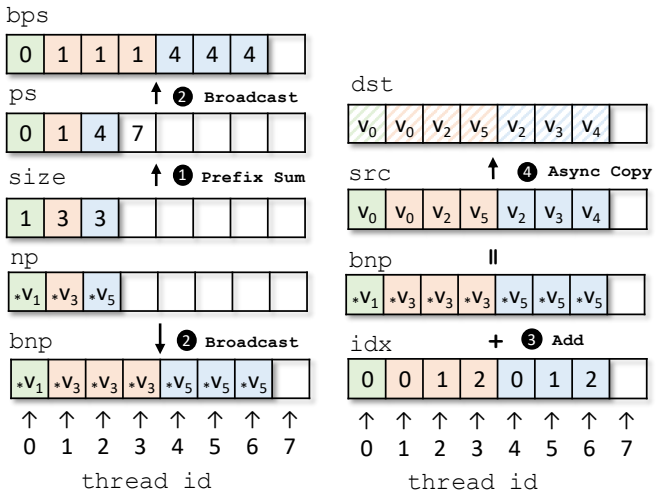


Figure 43. The process of copying three different-sized arrays in parallel using 8 threads of a warp. $*v_i$ represents the address of v_i's latest neighbor array version in the MVG.

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 43 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 42. 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.

4.5 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 [82], 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 [20, 90] for EX to ensure both correctness and efficiency in CSM.

4.5.1 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 4.2.2), 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.

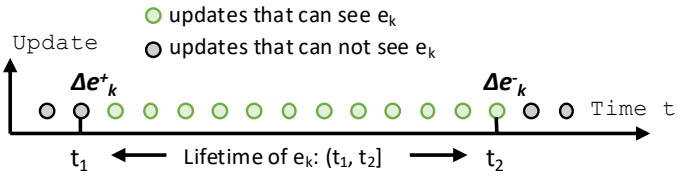


Figure 44. Visualization of edge e_k's lifetime and visibility across other updates.

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 44 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 42, each $(ct, dt]$ pair represents a segment of an edge's lifetime. For example, the lifetime of edge (v_1, v_0) consists of two segments, $(0, 3]$ and $(3, m]$, while edge (v_1, v_5) has a lifetime of $(7, m]$. The lifetime of a neighbor array covers the time ranges of edges in it. Therefore, according to Property 2, the valid neighbor edges must be in neighbor array versions whose lifetime $(ct_{min}, dt_{max}]$ contains $T(\Delta e_k)$.

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

4.5.2 Continuous Subgraph Matching-specific Optimizations

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

between 1 and 5, which is much smaller than 32. Therefore, compared with previous systems, our design does not increase the time complexity of neighbor array access.

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

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

4.6 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.

4.6.1 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.

4.6.2 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.

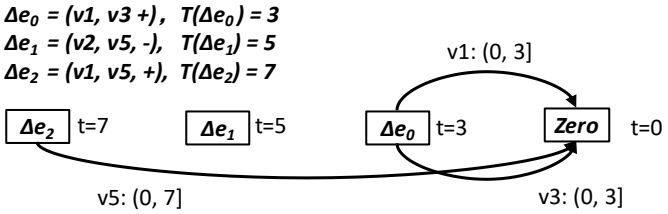
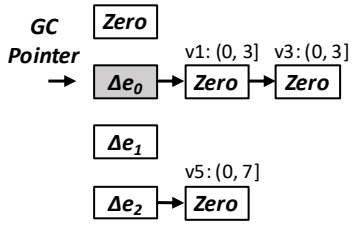


Figure 45. A garbage collection graph (GCG) built from the 3 updates Δe_0, Δe_1, Δe_2 in Figure 42.

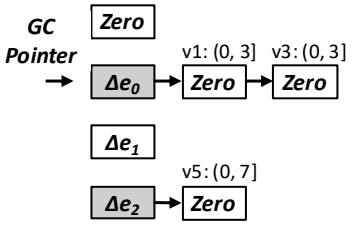
The GCG's vertices include a *Zero* vertex and all Δe_k that have been processed by the graph updater (GU). Each vertex in GCG is an edge update Δe_k timestamped with its arrival time, and the *Zero* vertex is a dummy edge update timestamped with 0. Each edge in GCG corresponds to a "deletable" neighbor 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 45 shows a GCG example; it has four vertices, three of which are the edge updates in Figure 42. Since GU has created v_1:$(3,m]$, v_3:$(3,m]$, and v_5:$(7,m]$, the arrays v_1:$(0,3]$, v_3:$(0,3]$, and v_5:$(0,7]$ become deletable. Therefore, three edges are added to the GCG and are marked with their corresponding deletable neighbor arrays.

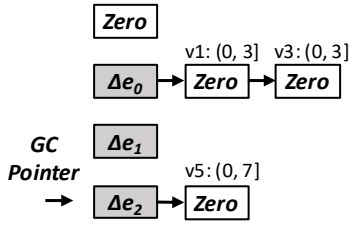
The GCG guides the release of neighbor arrays. An edge v_x:$(ct_{min}, dt_{max}]$ in the GCG corresponds to a neighbor array version, where the vertices (edge updates) below this edge represent updates Δe_k with $T(\Delta e_k) \in (ct_{min}, dt_{max}]$. According to Section 4.6.1, once all edge updates



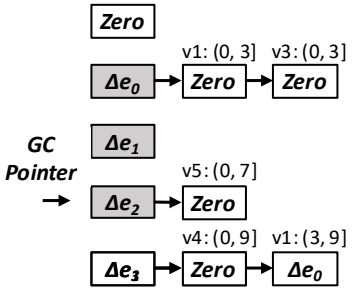
(a) The incremental matching of Δe_0 is finished, v_1:$(0, 3]$ and v_3:$(0, 3]$ can be released.



(b) The incremental matching of Δe_2 is finished.



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



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

Figure 46. (a)–(d) illustrate how GC releases neighbor arrays on the GCG in Figure 45. This GCG is stored in an adjacency-list format.

below a GCG edge have completed incremental matching, the corresponding neighbor array can be released. For example, in Figure 45, 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.

4.6.3 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 46 shows how GC releases the neighbor arrays on the GCG in Figure 45. 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 4.6.1.

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.

4.7 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.

4.7.1 Experimental Setup

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.

Table 17. 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

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 17 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 [46]. Netflow [14] is a graph of passive traffic traces. LSBench [44] 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 47. The experiments include labeled and unlabeled matching. For labeled matching, we randomly assign five labels to the data graph and query pattern.

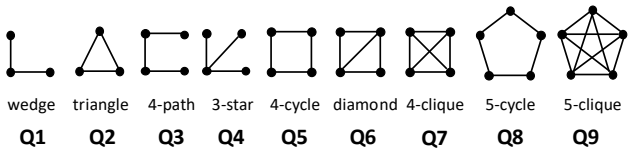


Figure 47. Query patterns for evaluation.

Baselines: We select three recent work, RapidFlow [78], CaLiG [93], and NewSP [48] as our CPU baselines. They are three state-of-the-art open-sourced continuous subgraph matching systems on the CPU, and have reported better performance than other CPU systems, including Graphflow [40], TurboFlux [41], SJ-Tree [21], IEDyn [33], and Symbi [59]. Each of them features different optimizations.

There are two previous GPU systems: GCSM [91] and GAMMA [68]. We summarize their performance and differences in Table 18. 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 [20] 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 [90, 92] 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 18. 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.

4.7.2 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.

Comparison with the CPU Systems

Table 19 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

Table 19. 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

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.

Comparison with the GPU Naive Methods

As introduced in Section 4.7.1, 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.

From Table 19, 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.

Comparison with GCSM

Table 20 shows the performance comparison between GCSM and DCSM for unlabeled and labeled continuous subgraph matching. As introduced in Section 4.1, 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.

Table 20. 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

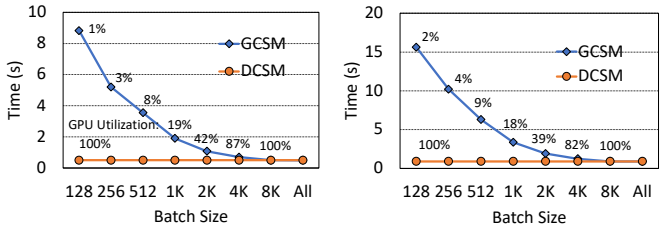
We observe that the numbers in Table 20 roughly correspond to the number of active warps on the GPU. GCSM achieves the same performance as DCSM when processing with a batch size large enough to saturate GPU resources. For some small labeled data graphs, such as en, az, and db, GCSM requires a larger batch size, because a smaller batch is processed too quickly to effectively hide the kernel launch time for the next batch.

Figure 48 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.

4.7.3 Response Time

In this section, we compared the response times of DCSM, GCSM, and RapidFlow over different update rates (updates / second). Figure 49 shows the experimental results.

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



(a) (b)

Figure 48. (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.

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 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 49 represents update rate.

The experimental results confirm the effectiveness of the DCSM. As shown in Figure 49, 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 49; the GCSM-128 curve appears as a nearly flat line above RapidMatch. In Figures 49a to 49f, 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 50 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

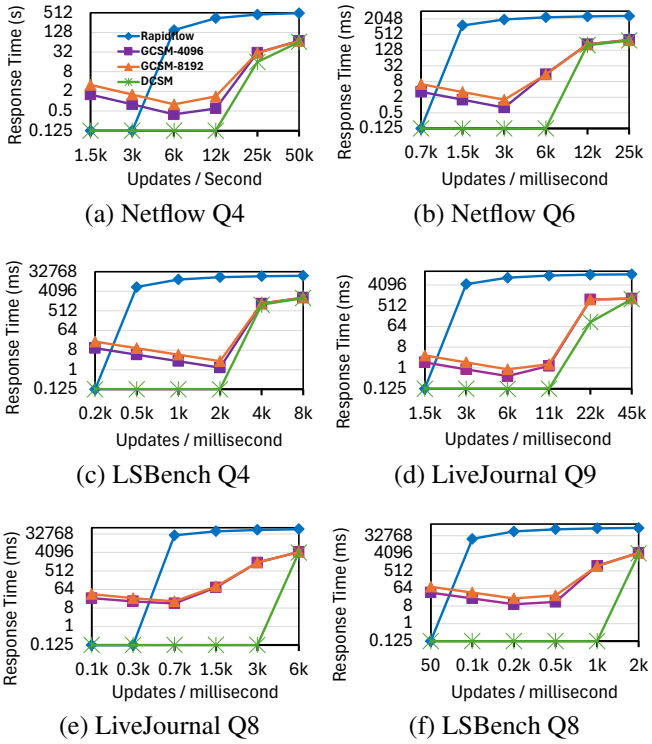


Figure 49. Response time over different update rates. The time unit is seconds for Netflow Q4 and milliseconds for all others. GCSM-4096 refers to GCSM configured with a batch size of 4096.

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%.

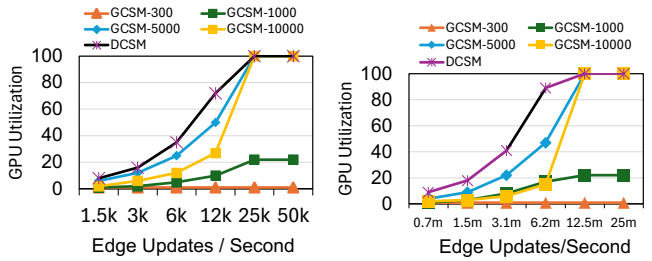


Figure 50. GPU utilization vs. update rate (k: thousand, m: million) for different systems

4.7.4 Overhead Analysis

Table 21. 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

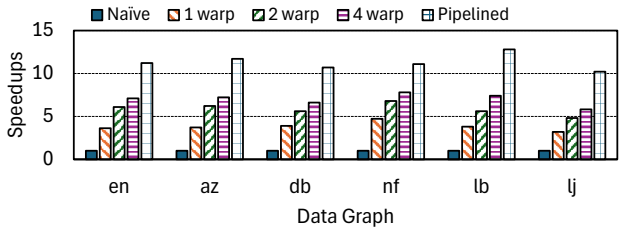
The MVG data structure causes DCSM to perform more data copying than other systems, which increases memory access overhead.

Table 21 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.

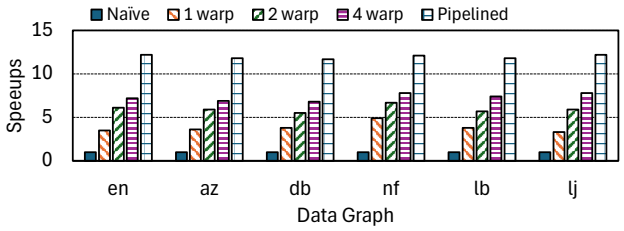
By comparing Table 21 and Table 19, it can be observed that, in most cases, the time required by GU is significantly lower than that for incremental matching. This indicates that GU can produce updates faster than the executor (EX) can consume them, even when updating the MVG with a batch size of 1. 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 19, 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



(a) Batch Size = 64



(b) Batch Size = 128

Figure 51. 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.

rates equal to or lower than the processing speed of EX, GPU memory consumption remains stable over time.

4.7.5 Ablation Study

Figure 51 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 51a, 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.

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.

4.8 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 [52], Dryadic [53], and GraphPi [74], Peregrine [37], Automine [55], G2Miner [20], Sandslash [18], DecoMine [17], and RapidMatch [77] These also include GPU systems, such as Gunrock [85], GPSM [80], GSI [96], cuTS [92], PBE [27], and STMatch [90]. These systems introduce various program optimization methods to improve hardware utilization.

Continuous subgraph matching has been studied for decades on CPU. IncIsoMatch [24] 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 [40] 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 [21] 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 [41] 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 [59] is the successive work of TurboFlux and proposes a pruning method for candidate vertices with query edges. Rapidflow [78] 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 [93] 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 [48] 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 [68, 91] have started using GPUs to accelerate continuous subgraph matching. GCSM [91] 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 [68] 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.

4.9 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] Nsight compute. `https://developer.nvidia.com/nsight-compute`.

[2] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 716–727. IEEE, 2016.

[3] 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.

[4] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery*, 29:626–688, 2015.

[5] 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.

[6] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, et al. Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 282–297, 2021.

[7] Indrajit Bhattacharya and Lise Getoor. Entity resolution in graphs. *Mining graph data*, 311, 2006.

[8] Bibek Bhattarai, Hang Liu, and H Howie Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1447–1462, 2019.

[9] 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*, SIGMOD '16, page 1199–1214, New York, NY, USA, 2016. Association for Computing Machinery.

[10] 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.

[11] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 14:1–13, 2013.

[12] M. Bressan, Stefano Leucci, and A. Panconesi. Motivo: Fast motif counting via succinct color coding and adaptive sampling. *Proc. VLDB Endow.*, 12:1651–1663, 2019.

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

[14] JE CAIDA. The caida ucsd anonymized internet traces, 2019.

[15] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):804–818, 2017.

[16] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–12, 2018.

[17] Jingji Chen and Xuehai Qian. Decomine: A compilation-based graph pattern mining system with pattern decomposition. In *Proceedings of the 28th ACM International Conference on*

Architectural Support for Programming Languages and Operating Systems, Volume 1, pages 47–61, 2022.

[18] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Sandslash: a two-level framework for efficient graph pattern mining. In *Proceedings of the ACM International Conference on Supercomputing*, pages 378–391, 2021.

[19] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proc. VLDB Endow.*, 13(8):1190–1205, April 2020.

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

[21] 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.

[22] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[23] 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.

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

[25] 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.

[26] Chuangyi Gui, Xiaofei Liao, Long Zheng, Pengcheng Yao, Qinggang Wang, and Hai Jin. Sumpa: Efficient pattern-centric graph mining with pattern abstraction. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 318–330. IEEE, 2021.

[27] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1067–1082, 2020.

[28] 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.

[29] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 337–348, New York, NY, USA, 2013. Association for Computing Machinery.

[30] Mark Harris and Kyrylo Perelygin. Cooperative groups: Flexible cuda thread programming. `https://developer.nvidia.com/blog/cooperative-groups/`, 2017.

[31] Juris Hartmanis. Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). *Siam Review*, 24(1):90, 1982.

[32] Huahai He and Ambuj K Singh. Query language and access methods for graph databases. In *Managing and mining graph data*, pages 125–160. Springer, 2010.

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

[34] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, et al. Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment*, 9(13):1317–1328, 2016.

[35] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. {ASAP}: Fast, approximate graph pattern mining at scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 745–761, 2018.

[36] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 — seamless operability between c++11 and python. `https://github.com/pybind/pybind11`, 2016.

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

[38] 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.

[39] Peng Jiang, Yihua Wei, Jiya Su, Rujia Wang, and Bo Wu. Samplemine: A framework for applying random sampling to subgraph pattern mining through loop perforation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 185–197, 2022.

[40] 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.

[41] 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.

[42] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce: a cost-oriented approach. *The VLDB Journal*, 26(3):421–446, 2017.

[43] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 10(3):217–228, 2016.

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

[45] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment*, 6(2):133–144, 2012.

[46] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[47] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*, pages 3835–3845. PMLR, 2019.

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

[49] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiao-Li Li. Network motif discovery: A gpu approach. *IEEE transactions on knowledge and data engineering*, 29(3):513–528, 2016.

[50] Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, Jure Leskovec, et al. Neural subgraph matching. *arXiv preprint arXiv:2007.03092*, 2020.

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

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

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

[54] Daniel Mawhirter and Bo Wu. Automine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 509–523, New York, NY, USA, 2019. Association for Computing Machinery.

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

[56] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, July 2019.

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

[58] 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.

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

[60] 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.

[61] Caleb C Noble and Diane J Cook. Graph-based anomaly detection. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, 2003.

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

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

[64] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. C-saw: A framework for graph sampling and random walk on gpus. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[65] Aduri Pavan, Srikanta Tirthapura, et al. Counting and sampling triangles from a graph stream. 2013.

[66] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th international conference on world wide web*, pages 1431–1440, 2017.

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

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

[69] 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.

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

[71] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. Fast and robust distributed subgraph enumeration. *arXiv preprint arXiv:1901.07747*, 2019.

[72] Satu Elisa Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[73] 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.

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

[75] 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.

[76] Jiya Su, Linfeng He, Peng Jiang, and Rujia Wang. Exploring pim architecture for high-performance graph pattern mining. *IEEE Computer Architecture Letters*, 20(2):114–117, 2021.

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

[78] 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.

[79] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. An in-depth study of continuous subgraph matching. *Proceedings of the VLDB Endowment*, 15(7):1403–1416, 2022.

[80] 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.

[81] Johan Ugander, Lars Backstrom, and Jon Kleinberg. Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, page 1307–1318, New York, NY, USA, 2013. Association for Computing Machinery.

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

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

[84] Hanchen Wang, Ying Zhang, Lu Qin, Wei Wang, Wenjie Zhang, and Xuemin Lin. Reinforcement learning based query vertex ordering model for subgraph matching. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 245–258. IEEE, 2022.

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

[86] Leyuan Wang, Yangzihao Wang, and John D Owens. Fast parallel subgraph matching on the gpu. In *HPDC*, 2016.

[87] 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.

[88] Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, and Yihua Huang. Benu: Distributed subgraph enumeration with backtracking-based framework. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 136–147. IEEE, 2019.

[89] Larry Wasserman. *All of statistics : a concise course in statistical inference*. Springer, New York, 2010.

[90] 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.

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

[92] 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.

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

[94] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. Huge: An efficient and scalable subgraph enumeration system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2049–2062, 2021.

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

[96] 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.

[97] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3(1–2):340–351, September 2010.