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 [19], social network analysis [26], and cybersecurity [20]. 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 [17].

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 [3, 5, 8, 9, 13, 18, 22, 27, 33]. However, as the problem is NP-hard [10], 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 [25, 28, 30, 32].

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 [25, 28], or they find a match for the next pattern node by performing set operations on the neighbor lists of the previous nodes [30, 32]. A common feature of these systems is that they need to store the partially matched subgraphs explicitly. For example, GSI [32] 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 [30], 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 [30], 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 [16]). As a result, the state-of-the-art GPU graph pattern matching system (cuTS [30]) can be even slower than a highly optimized CPU implementation (Dryadic [16]) 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 [16], 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 [16] 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 [30], GSI [32] and Dryadic [16]. 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;
```

```
// u0 can be mapped to any node in data graph
C0 = V;
for (i0 = 0; i0 < C0.size; i0++) {
  v0 = C0[i0];
  // u1 is a neighbor of u0
  C1 = N(v0);
  for (i1 = 0; i1 < C1.size; i1++) {
    v1 = C1[i1];
    // u2 is a neighbor of u0 but not a neighbor of u1
    C2 = N(v0) − N(v1);
    for (i2 = 0; i2 < C2.size; i2++) {
      v2 = C2[i2];
      // u3 is a neighbor of u0, u1 and u2
      C3 = N(v0) ∩ N(v1) ∩ N(v2);
      for (i3 = 0; i3 < C3.size; i3++) {
        v3 = C3[i3];
        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.

or edge-induced. If the subgraphs are edge-induced, the problem is equivalent to the subgraph isomorphism problem [27].

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 [5, 18, 22, 33]. 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 convenient for parallelization and optimization [1,6,16,17,23,24]. 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.

Figure 2. A query graph.

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 [16, 17]. 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 [2, 16, 23]. 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 1, 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 [30], 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 [16], 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 [16] 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.

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

```
   // 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.

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 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

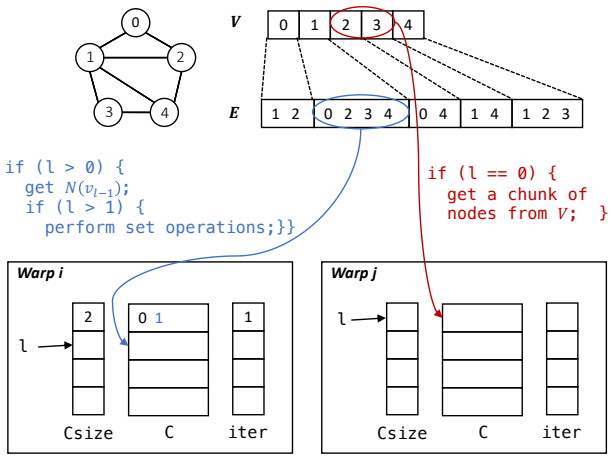


Figure 4. An example of *getCandidates* in two different warps.

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 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

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;
```

stealing technique.

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

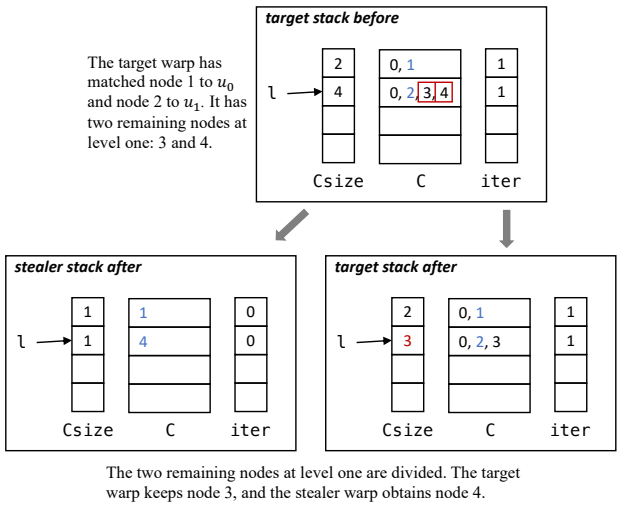


Figure 5. An example of dividing and copying tasks from a target warp.

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.

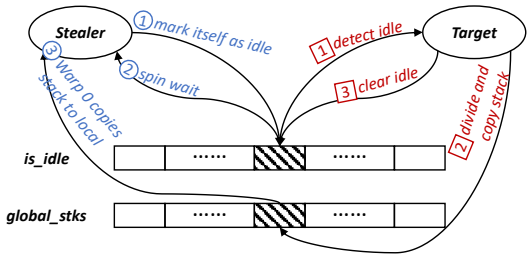


Figure 6. Work stealing across threadblocks.

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

```
   // 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.

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

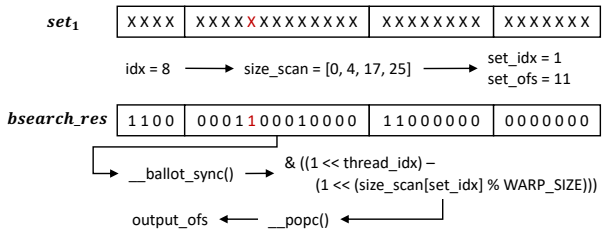


Figure 8. Perform multiple set operations in one warp.

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.

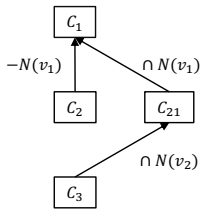
The divide-and-copy procedure in Fig. 5 needs to be slightly modified to enable working stealing for the unrolled loop. We use the same procedure for dividing and copying the tasks in the current unrolled iteration. But for the remaining unrolled iterations, we need to set the *Csize* to zero in the 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

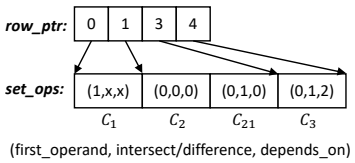
To show that our system is compatible with the existing optimizations for backtracking-based graph pattern matching, we implement the code motion technique proposed in [16] in our system. The idea is to lift the loop-invariant part of the set operations to upper levels so that they will not be computed 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 dimension of *C* and *Csize* from *PAT_SIZE* to the total number of sets of all levels. We also need

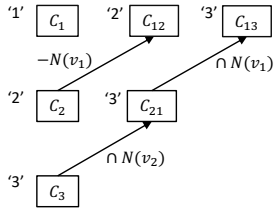


(a) Set dependence graph

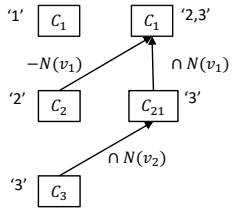


(b) A compact storage

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.

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 [16].

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 [16] 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, [16] 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.

1.8 Experimental Results

In this section, we compare our system with two state-of-the-art GPU graph pattern matching systems: cuTS [30] and GSI [32], and a state-of-the-art CPU system: Dryadic [16]. GSI achieves consistently better (or equal) performance than many previous CPU and GPU imple-

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.

mentations such as CFL-Match [3], CBWJ [18], VF3 [4], GPSM [25] and Gunrock [28]. CuTS has shown even better performance than GSI. However, because cuTS[1] does not support labeled queries, we also compare with GSI[2] 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 1 lists the data graphs used in our experiments. These graph are obtained from the SNAP [14] 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 [3, 16, 25, 32, 33]. 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 [16] 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

[1] `https://doi.org/10.5281/zenodo.5154114`
[2] `https://github.com/pkumod/GSI`

	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.

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 1), 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 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.

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`).

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 match-

ing systems.

CPU-based systems: The study of subgraph isomorphism problem dates back to 1970s. Ullmann [27] 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 [5, 18, 22, 33]. 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 [3,8,9,13]. A more recent work, Dryadic [16], 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 [25, 28, 32] 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 [15,30]. 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 [30] 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 [7,30]. PBE [7] 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 [2, 11, 12, 21, 23, 29, 31]. The main challenge is to balance the workload among machines. CECI [2] 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 [21] proposes a region-grouped multi-round expand technique to reduce communication and minimize intermediate result storage. BENU [29] proposes a task splitting technique based on node degree to optimize the load balance among machines. GraphPi [23] 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.

REFERENCES

[1] 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.

[2] 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.

[3] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1199–1214, New York, NY, USA, 2016. Association for Computing Machinery.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1429–1446, New York, NY, USA, 2019. Association for Computing Machinery.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

	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.

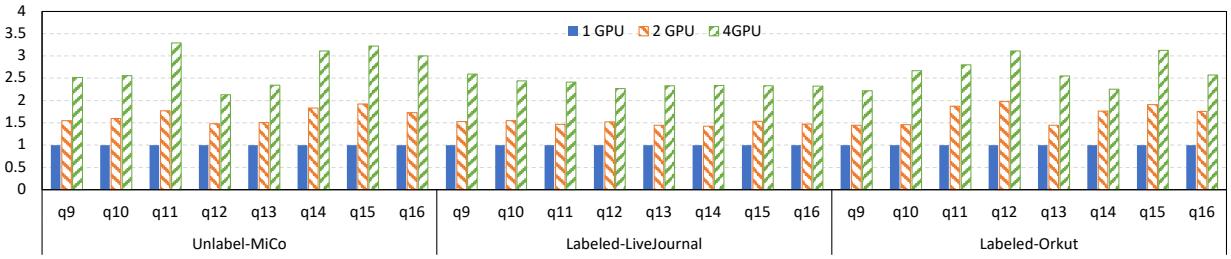
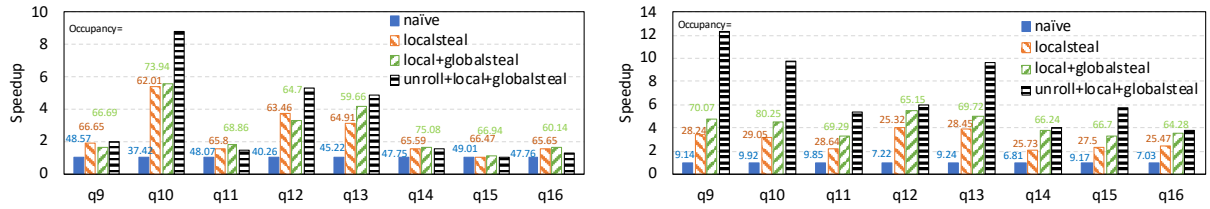
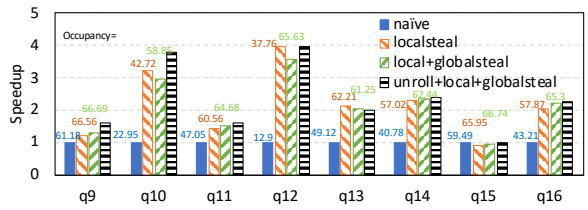


Figure 11. Speedups of labeled and unlabeled size-6 queries across multiple GPUs.

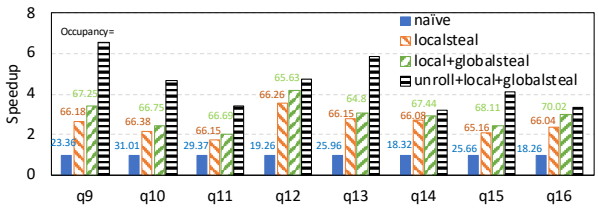


(a) Enron

(b) MiCo



(c) YouTube



(d) LiveJournal

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

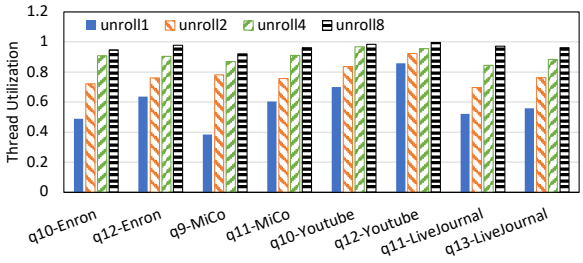


Figure 13. Thread utilization with different unrolling sizes.