STMatch: Accelerating Graph Pattern Matching on GPU with Stack-Based Loop Optimizations

Yihua Wei  
University of Iowa  
Iowa City, USA  
yihua-wei@uiowa.edu

Peng Jiang  
University of Iowa  
Iowa City, USA  
peng-jiang@uiowa.edu

Abstract—Graph pattern matching is a fundamental task in many graph analytics and graph mining applications. As an NP-hard problem, it is often a performance bottleneck in these applications. Previous work has proposed to use GPU to accelerate the computation. However, we find that the existing GPU solutions fail to show a performance advantage over the state-of-the-art CPU implementation due to their subgraph-centric design. This work proposes a novel stack-based graph pattern matching system on GPU that avoids the synchronization and memory consumption issues of the previous subgraph-centric systems. We also propose a two-level work-stealing and a loop-unrolling technique to improve the inter-warp and intra-warp GPU resource utilization of our system. The experiments show that our system significantly advances the state-of-the-art for graph pattern matching on GPU.

Index Terms—Parallel programming, Backtracking

I. INTRODUCTION

Graph pattern matching is widely used for retrieving information from graph-structured data in many applications 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 II).

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...
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 [32] and Dryadic [16]. The experiments show that our system is compatible with the existing optimizations for backtracking-based graph pattern matching.

II. 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.

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

B. 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 \( \pi \) 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., \( \pi[0], ..., \pi[l] \)). 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 \( \text{Enumerate} \) 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 \( l \)th 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 \( \text{Enumerate} \) 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 removes \( v \) 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 \( u_0 \) but not in the neighbor list of \( v_1 \). Thus, we compute the candidate nodes for \( u_2 \) as \( N(u_0) - N(v_1) \). Similarly, the third loop computes the candidate nodes for \( u_3 \) as \( N(u_0) \cap N(v_1) \cap N(v_2) \) since \( u_3 \) is neighboring to \( u_0, u_1 \) and \( u_2 \).

C. 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.

III. 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 I, 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 \( N(u_0) \cap N(v_1) \) 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.

IV. 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 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 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.

V. 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.

A. 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

```c
// 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 < 0 || size) {
7:           if (Csize[l] == 0) {
8:               // if this subgraph has not been extended
9:               getCandidates(G, Q, l, C[l]);
10:              // algorithm stops if no subgraph can be extended
11:             if (l > 0) { l--; iter[l]++; }
14:         } else { // output subgraph at last level
15:             Output(C, iter); l--;
16:         } 
17:     } else { // if this subgraph has not been extended
18:         getCandidates(G, Q, l, C[l]);
19:         if (l > 0) { l--; iter[l]++; } }  
```
is_idle…………
Stealer Target
global_stks…………
mark itself as idlespin waitdetect idleclear idledivide and copy stack
Warp 0copies stack to local

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, \( \circ \) it marks itself as idle in the $is_idle$ array and \( \bullet \) spins 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, \( \square \) 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, \( \bigcirc \) it divides and copies its tasks to the $global_stk$ of that threadblock. The divide-and-copy
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 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 back track 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 set(set_2[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 Csize to zero in the stealer stack since the tasks in these iterations are not stolen from the target.

---

**Fig. 8:** Perform multiple set operations in one warp.

**Fig. 7:** An unrolled version of the loop in Fig. 3.
We also need to copy `uiter` from level zero to `target_level`.

VII. 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 $C_{size}$ 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 $C_2$ 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 second 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 $C_1 \cap N(v_1)$, 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 $C_{size}$ 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.
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 implementations 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.

### A. 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 I lists the data graphs used in our experiments. These graphs 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 graphs 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 system, 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: \(q_1 \sim q_6, q_9 \sim q_{16}\) 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 all systems.

#### Settings:
In all experiments, we set the \(StopLevel\) in Algorithm 2 to 2, and the \(DetectLevel\) for global work-stealing in Section V-B 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 \(NUM\_SETS\times UNROLL\times MAX\_DEGREE\times NUM\_WARP\). 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 GPU. Because real-world graphs are skewed and few nodes have degrees greater than 4096 (last column in Table I), 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.

### B. 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 II(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 III 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 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 II(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.

#### C. 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

![Table I: Graph datasets.](https://github.com/pkumod/GSI)

<table>
<thead>
<tr>
<th>Graph</th>
<th># nodes</th>
<th># edges</th>
<th>Max deg.</th>
<th>Med deg.</th>
<th>Deg. &gt; 4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal</td>
<td>24</td>
<td>9</td>
<td>1.1M</td>
<td>0.06%</td>
<td>117.2M</td>
</tr>
<tr>
<td>Orkut</td>
<td>3.5M</td>
<td>37K</td>
<td>9</td>
<td>0.12%</td>
<td>117.2M</td>
</tr>
<tr>
<td>MiCo</td>
<td>665M</td>
<td>33313</td>
<td>9</td>
<td>9.16x-9%</td>
<td>117.2M</td>
</tr>
</tbody>
</table>

\(^1\)https://doi.org/10.5281/zenodo.5154114
\(^2\)https://github.com/pkumod/GSI
TABLE II: 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.

(a) Edge-induced

<table>
<thead>
<tr>
<th></th>
<th>WikiVote</th>
<th>Enron</th>
<th>MiCo</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1</td>
<td>492.0</td>
<td>433.4</td>
<td>1055.2</td>
</tr>
<tr>
<td>q2</td>
<td>1608.9</td>
<td>954.3</td>
<td>1139.2</td>
</tr>
<tr>
<td>q3</td>
<td>131.1</td>
<td>909.5</td>
<td>41077</td>
</tr>
<tr>
<td>q4</td>
<td>295.7</td>
<td>354.1</td>
<td>31679</td>
</tr>
<tr>
<td>q5</td>
<td>246.8</td>
<td>189.4</td>
<td>13812</td>
</tr>
<tr>
<td>q6</td>
<td>22.3</td>
<td>679.2</td>
<td>11845</td>
</tr>
<tr>
<td>q7</td>
<td>36.1</td>
<td>120.6</td>
<td>9002</td>
</tr>
<tr>
<td>q8</td>
<td>293.2</td>
<td>527.1</td>
<td>6680</td>
</tr>
<tr>
<td>q9</td>
<td>2799.8</td>
<td>215649</td>
<td>33514</td>
</tr>
<tr>
<td>q10</td>
<td>1858.7</td>
<td>41214</td>
<td>1641</td>
</tr>
<tr>
<td>q11</td>
<td>899.1</td>
<td>338166</td>
<td>8099</td>
</tr>
<tr>
<td>q12</td>
<td>7485.7</td>
<td>54704</td>
<td>5995</td>
</tr>
<tr>
<td>q13</td>
<td>653.2</td>
<td>1423</td>
<td>165745</td>
</tr>
<tr>
<td>q14</td>
<td>88.9</td>
<td>598</td>
<td>137365</td>
</tr>
<tr>
<td>q15</td>
<td>48.6</td>
<td>487.8</td>
<td>31268</td>
</tr>
<tr>
<td>q16</td>
<td>3599.6</td>
<td>9661</td>
<td>5228</td>
</tr>
<tr>
<td>q17</td>
<td>3917.8</td>
<td>253002</td>
<td>33005</td>
</tr>
<tr>
<td>q18</td>
<td>396.4</td>
<td>131216</td>
<td>13417</td>
</tr>
<tr>
<td>q19</td>
<td>3078.5</td>
<td>32567</td>
<td>3995</td>
</tr>
<tr>
<td>q20</td>
<td>2080.0</td>
<td>4301</td>
<td>4130</td>
</tr>
<tr>
<td>q21</td>
<td>2075.8</td>
<td>35991</td>
<td>2194</td>
</tr>
<tr>
<td>q22</td>
<td>265.6</td>
<td>1645</td>
<td>338</td>
</tr>
<tr>
<td>q23</td>
<td>139.3</td>
<td>531</td>
<td>129</td>
</tr>
</tbody>
</table>

(b) Vertex-induced

<table>
<thead>
<tr>
<th></th>
<th>WikiVote</th>
<th>Enron</th>
<th>MiCo</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1</td>
<td>296.0</td>
<td>3169</td>
<td>1055.2</td>
</tr>
<tr>
<td>q2</td>
<td>106.0</td>
<td>1429</td>
<td>101</td>
</tr>
<tr>
<td>q3</td>
<td>186.0</td>
<td>1471</td>
<td>206</td>
</tr>
<tr>
<td>q4</td>
<td>194.0</td>
<td>1781</td>
<td>133</td>
</tr>
<tr>
<td>q5</td>
<td>28.0</td>
<td>178</td>
<td>34</td>
</tr>
<tr>
<td>q6</td>
<td>36.0</td>
<td>123</td>
<td>82</td>
</tr>
<tr>
<td>q7</td>
<td>2088.0</td>
<td>19515</td>
<td>3405</td>
</tr>
<tr>
<td>q8</td>
<td>6380.0</td>
<td>63026</td>
<td>8871</td>
</tr>
<tr>
<td>q9</td>
<td>307.0</td>
<td>8921</td>
<td>388</td>
</tr>
<tr>
<td>q10</td>
<td>21584.0</td>
<td>20505</td>
<td>2091</td>
</tr>
<tr>
<td>q11</td>
<td>497.0</td>
<td>2279</td>
<td>414</td>
</tr>
<tr>
<td>q12</td>
<td>48.0</td>
<td>285</td>
<td>47</td>
</tr>
<tr>
<td>q13</td>
<td>2216.0</td>
<td>6661</td>
<td>1399</td>
</tr>
<tr>
<td>q14</td>
<td>86.0</td>
<td>579</td>
<td>110</td>
</tr>
<tr>
<td>q15</td>
<td>48.0</td>
<td>285</td>
<td>47</td>
</tr>
<tr>
<td>q16</td>
<td>2037.0</td>
<td>13433</td>
<td>1091</td>
</tr>
<tr>
<td>q17</td>
<td>2129.0</td>
<td>16506</td>
<td>1464</td>
</tr>
<tr>
<td>q18</td>
<td>2501.0</td>
<td>969</td>
<td>7091</td>
</tr>
<tr>
<td>q19</td>
<td>1957.0</td>
<td>4266</td>
<td>2732</td>
</tr>
<tr>
<td>q20</td>
<td>996.0</td>
<td>3940</td>
<td>902</td>
</tr>
<tr>
<td>q21</td>
<td>283.0</td>
<td>1089</td>
<td>356</td>
</tr>
<tr>
<td>q22</td>
<td>139.0</td>
<td>531</td>
<td>129</td>
</tr>
</tbody>
</table>

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 test cases. 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 work-steal 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 label 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.

IX. 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 [27] proposes the first backtracking algorithm that iteratively matches query nodes on data graph based on a certain order. Many studies follow...
TABLE III: Execution time (in milliseconds) of different systems for labeled edge-induced matching. GSI fails for all queries on MiCo, LiveJournal, Orkut and Friendster.

<table>
<thead>
<tr>
<th></th>
<th>WikiVote</th>
<th>Enron</th>
<th>Youtube</th>
<th>MiCo</th>
<th>LiveJournal</th>
<th>Orkut</th>
<th>Friendster</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>naïve</td>
<td>localsteal</td>
<td>local+globalsteal</td>
<td>unroll+local+globalsteal</td>
<td>naïve</td>
<td>localsteal</td>
<td>local+globalsteal</td>
</tr>
<tr>
<td>q1</td>
<td>51</td>
<td>48</td>
<td>45</td>
<td>52</td>
<td>60</td>
<td>59</td>
<td>43</td>
</tr>
<tr>
<td>q10</td>
<td>216</td>
<td>217</td>
<td>216</td>
<td>217</td>
<td>217</td>
<td>217</td>
<td>217</td>
</tr>
<tr>
<td>q11</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>q12</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>q13</td>
<td>401</td>
<td>401</td>
<td>401</td>
<td>401</td>
<td>401</td>
<td>401</td>
<td>401</td>
</tr>
<tr>
<td>q14</td>
<td>104</td>
<td>104</td>
<td>104</td>
<td>104</td>
<td>104</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td>q15</td>
<td>320</td>
<td>320</td>
<td>320</td>
<td>320</td>
<td>320</td>
<td>320</td>
<td>320</td>
</tr>
<tr>
<td>q16</td>
<td>3403</td>
<td>3403</td>
<td>3403</td>
<td>3403</td>
<td>3403</td>
<td>3403</td>
<td>3403</td>
</tr>
</tbody>
</table>

Due to the large intermediate exploration space, the partial subgraphs of a certain size before exploring larger subgraphs. There are a number of GPU systems extended with any of the previous matching order strategies. However, our system can be 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 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.

![Fig. 12: Speedups of labeled size-6 queries with and without work-stealing and loop unrolling.](image)

![Fig. 13: Thread utilization with different unrolling sizes.](image)
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.

X. 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.

ACKNOWLEDGEMENTS

This work was supported by NSF award CCF-2028825. The authors thank Bo Wu and Han Wei from Colorado School of Mines for helpful discussions and sharing the code of Dryadic.

REFERENCES


Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED
This artifact includes the code and scripts for reproducing the results of our system (STMatch) and compared systems (cuTS, GSI, Dryadic) in our experiments. The code is available on GitHub: https://github.com/HPC-Research-Lab/STMatch.git.

Hardware and Software Requirements. The code should be able to run on any Linux machine with Nvidia GPUs. To reproduce the results in the paper, the program should be run on the following platform:

- OS: Ubuntu 20.04.4 LTS
- Compiler: NVCC11.2, g++9.4.0
- GPU: NVIDIA GeForce RTX 3090
- CPU: a dual socket machine with Intel Xeon Gold 6226R 2.9GHz CPUs (32 cores in total) and 512GB RAM.

Experimental Workflow. There are mainly two steps to run the experiments:

- Follow the instructions in the ‘README’ of main directory to prepare the input data and reproduce results of STMatch.

Run with Docker. Our code does not depend on any external library, so the easiest way to run the code is to compile and run on the host machine. However, we also provide a podman/docker image that contains the source code and test data. The image can be launched by podman or docker. To use the host GPU within the container, one needs to do some setup on the host as instructed in the AD/AE.

We also provide bash scripts for preparing input data, compiling the source code, and reproducing results in the experiments. Details can be found in the GitHub repository.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1
Persistent ID: https://doi.org/10.5281/zenodo.6991445
Artifact name: STMatch

Reproduction of the artifact with container: A podman/docker image that contains the source code, software dependencies, and some test data is provided. The image can be launched by podman or docker. However, to use host GPU within the container, one needs to do some setup on the host as instructed in the AD/AE.

We also provide bash scripts for preparing input data, compiling the source code, and reproducing results in the experiments. Details can be found in the GitHub repository.