DGSM: A GPU-Based Subgraph Isomorphism framework with DFS exploration

Wei Han
Department of Computer Science
Colorado School of Mines
Golden CO, USA
whan@mines.edu

Connor Holmes
Department of Computer Science
Colorado School of Mines
Golden CO, USA
cholmes@mines.edu

Bo Wu
Department of Computer Science
Colorado School of Mines
Golden CO, USA
bwu@mines.edu

***Abstract*—Subgraph Isomorphism is a fundamental problem in graph analytics and it has been applied to many domains. It is well known that subgraph isomorphism is a NP-complete problem. Thus, it generally becomes bottle-neck of the applications to which it is applied. There has been a lot of efforts devoted to this problem in the past two decades. However, GPU-based subgraph isomorphism systems are relatively rare since the GPU memory is not big enough to hold all the instances during the matching process. Most current GPU subgraph isomorphism frameworks suffer from the limited GPU main memory and redundant computation. These issues restrict them on smaller patterns and graphs and limit their performance. To overcome these issues, we design a new GPU-based subgraph isomorphism system named DGSM. Our system also efficiently utilize special architecture features to improve data parallelism and memory bandwidth for matching. We validate our techniques by comparing with two state-of-the-art systems, CPU-based DAF and GPU-based GSI. Our experimental results show that our system achieve 2 orders of magnitude faster than DAF and GSI on both labeled and unlabeled graph.**

***Index Terms*—GPU, subgraph isomorphism, subgraph matching, backtracking, DFS, shared memory, VertexSet, motif, clique**

I. Introduction

The subgraph isomorphism (also called subgraph matching) is a fundamental problem in graph analytics and has a wide spectrum of applications including chemical engineer [1], cybersecurity [2], bioinformatics [3], [4], and artificial intelligence [5]. It has been well-known that the subgraph isomorphism is an NP-complete problem. So subgraph isomorphism is the bottleneck of any application where it presents. Therefore, an efficient subgraph isomorphism algorithm is demanding and attracts attention from many computer scientists.

Although the first subgraph matching algorithm [6] was first put forward by Ullmann in 1970s using backtracking approach, it is still an open question to discover an efficient algorithm for such a problem. In the past decade, many efforts [7]–[9] have been devoted to an efficient algorithm such that it will reduce the searching space and pruning branches as earlier as possible. Most of them focused on discovering an efficient matching order to achieve these goals. These systems include Turbo$_{\text{ISO}}$, CFL, and DAF. However, these works do not scale well with data graphs due to the data race of updating the auxiliary structures used in these works. Furthermore, these systems need to materialize all the results during the matching process and cannot handle some large real-world graphs because of the large amount of intermediate results. due to exponential size of path embeddings, DAF [9] is not able to enumerate all the matches and Turbo$_{\text{ISO}}$ [7] cannot handle large data graphs or query graphs.

As many other graph algorithms, there are numerous data parallelisms in subgraph isomorphism. To employ massive data parallelism in subgraph matching, a few works have implemented GPU-based algorithms to accelerate the matching process. To the best of our knowledge, almost all of the GPU-based subgraph matching systems adopt bulk synchronous parallel (BSP) model, which consists of two stages, filter and join. The start-of-the-art systems include Gunrock, GpSM, and GSI. In these systems, the device needs to shake hand with the host and then allocate memory on the device memory at each iteration since it is not possible to know how many subinstances will be generated by each execution unit before hand [10]–[12]. Gunrock and GpSM do the join computation twice to eliminate write conflicts in the join phase. But GSI optimally pre-allocates memory buffers to avoid such duplicate computation. All of these GPU-based systems extend subpatterns by one vertex at a time until the target size is reached. In other words, these systems grow the potential subpattern instances in the fashion of BFS exploration strategy.

We summarize the disadvantages of BFS exploration as follows. Firstly, all the subintances in the current iteration need to be discovered before we can move on to the next iteration. The number of subinstances at one iteration grows exponentially as the size of a pattern. But the global memory on a modern GPU device is a few tens GB. This fact results in that the above systems cannot handle subgraph matching with a larger pattern size on large real-world graphs. Secondly, multiple kernels are necessary for their implementations due to the usage of filter-join computation model. A couple of grid level synchronizations are needed in one iteration. For a query graph with N vertices, there will be $2 \times N$ grid synchronizations. So many grid synchronizations can degrade the performance significantly for a large query graphs. These downsides are not avoidable in BFS exploration strategy.

Including the common disadvantages in BFS exploration, there is another common issue with their systems. The problem is that the same embedding will be materialized multiple time when there are symmetries present in the target pattern. These duplicates not only waste memory but also lead to unnecessary computation. One possible way to remove these duplicates is through an automorphism test. The automorphism test is expensive and often leads to significant performance drop. Incorporating such a test in each iteration will incur a lot of overhead. Therefore, it is the reason why GSI retains these duplicates throughout the entire computation.

To overcome the above issues in GPU-based subgraph matching systems, we create a new framework that targets these issues. The contributions of our work are

1) We implement a new subgraph matching system that is able to do subgraph matching on large real-world graphs with a larger pattern size which cannot be done by other GPU-based systems.
2) We design a new data structure named VertexSet keepubg track of candidates and facilitating coalesced memory accesses.
3) We arrange the edge list of a data graph in a special way that the filter phase can be avoided.

The rest of the paper is organized as follows. We first give the definition of subgraph isomorphism in section II. The related works are discussed in section III. Section IV presents the challenges of implementing a GPU-based subgraph matching application and out solutions to these challenges. Section 5 proposes our GPU-based subgraph matching system and gives a detailed discussion about the system. The evaluations of our system are shown in the section VI and the conclusion is reached in the section VII.

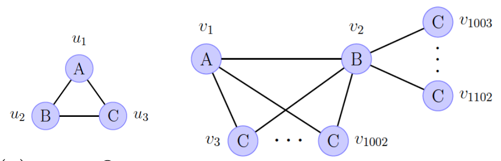
II. Subgraph Matching

Before we give the definition of subgraph isomorphism, we are going to first introduce the notations we use in the paper. Note that the terms, subgraph isomorphism and subgraph matching, are interchangeable. A vertex-labeled undirected graph is denoted by $G = (V, E, l, \Sigma)$ where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, l is a surjective mapping from V to Σ, and Σ is the set of labels which is finite and countable. We use $N_V(G)$ and $N_E(G)$ to stand for the number of vertices and edges in a graph G. The set of neighbors of a vertex v in graph G is represented by $N_G(v) = \{v' \in V(G) \,|\, (v, v') \in E(G)\}$. The degree or the size of the neighbor set of v is denoted by $d_G(v)$. Armed with these concepts, it is ready to provide the definition of subgraph isomorphism.

Definition 1: Given a pattern graph $Q = (V(Q), E(Q), l, \Sigma)$ and a data graph $G = (V(G), E(G), l', \Sigma)$, Q is isomorphic to G if and only if there exists a injective mapping f from $V(Q)$ to $V(G)$, that is $f : V(Q) \to V(G)$, such that

1) $\forall u \in V(Q), l(u) = l'(f(u))$
2) $\forall (u, u') \in E(Q), (f(u), f(u')) \in E(G)$

Figure 1 shows an example of matching a triangle in an input graph. There are 1000 instances of the query triangle in the data graph with the mapping $\{u_1 \to v_1, u_2 \to v_2, u_3 \to v_i \text{ with } i \in \{3 \cdots 1002\}\}$ in this example. One instance is also called an embedding.



(a) Query Graph (b) Data Graph

Fig. 1: Triangle Matching

Embeddings of the pattern Q in the graph G can be induced either by a matching vertex subset $V_s(G)$ of $V(G)$ or a corresponding matching edge subset. We call these two kinds of embedding as vertex-induced matching and edge-induced matching. The vertex-induced embedding is represented by

$$\{V_v = \{f(u) \in V(G)\}, \\ E_v = \{\forall (f(u), f(u')) \in E(G)\}, l, \Sigma\},$$

while the edge-induced embedding is defined as

$$\{V_e = \{f(u) \in V(G)\}, \\ E_e = \{(f(u), f(u')) \in E(G), \forall (u, u') \in E(P)\}, l, \Sigma\}.$$

The difference between two embeddings can be seen in the case of searching a wedge pattern in a triangle data graph. For convenience, we assume there is only one element in Σ. According to the definition, there is no vertex-induced embedding in this case since the wedge requires that an embedding induced by the three mapping vertices should not contain the missing edge in the query. An embedding induced by three vertices in a triangle violates this condition. But there will be three edge induced embeddings because constraint of the edge set.

III. Related Work

The original Ullmann algorithm [6], [13] needs to try all the possible mapping for a vertex u of a query graph and figure out a complete mapping at the end. Therefore, it turns out to the slowest subgraph matching algorithm so far. VF2 [14] improves the performance by imposing more pruning constraints than the Ullmann algorithm . VF2 requires that the next query vertex to be matched has to be connected the query vertices which are already matched.

A significant progress in subgraph matching has been witnessed since the beginning of this century, especially the past decade. Several proposals were made to speed up the subgraph matching problem on CPUs. Boost-ISO [15] exploits vertex relationships in the data graph to reduce duplicate computation and speed up subgraph isomorphism. But most works strive to search for a better matching order limiting the search space. They can be classified into two categories, vertex order and path order. VF2 and QuickSI [16] are two typical systems in the former category. QuickSI selects the most infrequent query vertex as the next query vertex to be matched.

The well-recognized systems falling into the second category include Turbo$_{\text{ISO}}$, CFL, and DAF. Turbo$_{\text{ISO}}$ [7] builds a spanning tree from a query graph by running a BFS on the query graph and establish a matching order based upon the size of potential matches of a path. The edges in a query graph are classified into two classes depending on the condition whether an edge in a query graph is present in the corresponding spanning tree. The edges which are not present in the query tree are called non-tree edges. In Turbo$_{\text{ISO}}$, the non-tree edges will be taken into account at the end. But non-tree edges usually have more pruning power than normal edges. Therefore, such construction lead to poor pruning power in Turbo$_{\text{ISO}}$. To cope with the symmetries in a pattern graph, it rewrites the query graph to neighborhood equivalence class.

To overcome the issue with non-tree edges, CFL [8] decomposes a query graph into three parts which are core, forest and leaf. The core is the subgraph with the minimum vertices containing all the non-tree edges. CFL will match the core part first to apply the pruning power of non-tree edges as early as possible. After the decomposition, CFL designs an auxiliary data structure, compact path-index, to facilitate the matching.

DAF [9] takes a different route from the above two systems. It first builds a DAG from a query graph using BFS. The non-tree edges, whose directions depend on the the order of vertices, will appear in the DAG. Then DAF will construct an auxiliary data structure, CS, to enumerate the embeddings. DAF proposes an adaptive matching order based on the path size and it introduces failing set to reduce the searching space. Both DAF and Turbo$_{\text{ISO}}$ prefer path order [17] than vertex order.

Although these three algorithms prove their efficiencies to some extent, these algorithms are not easy to scale up to real-world graphs, have poor data parallelism and suffer severe load balance issue. To employ data parallelism in subgraph matching, a few research groups have worked on GPU-based implementations. Most of these GPU systems adopt two-phase filter-join model [16]. The state-of-the-art GPU systems include GpSM [11], Gunrock [10], and GSI [12]. These systems extend the subpatterns by one vertex at a time through BFS exploration. Such an approach is easier to utilize data parallelism and to take care of load balance in each phase. However, the biggest disadvantage of this model is a large amount of intermediate results need to be materialized in the global memory during matching. Due to the limitation of global memory, it makes these systems difficult to handle large query patterns and data graphs.

An automorphism test is needed at the end of each extension in order to remove redundant results. They do not incorporated automorphism test in the implementation because such a test is expensive and there is no efficient algorithm for the test. For clarity, let's take the pattern in Figure 2 as an example where swapping u_3 and u_4 leaves the pattern invariant. The outcome of this symmetry is that the same subgraph induced by vertices v_1, v_2, v_3 and v_4 appears two times in the results by exchanging the mapping of u_3 and u_4.

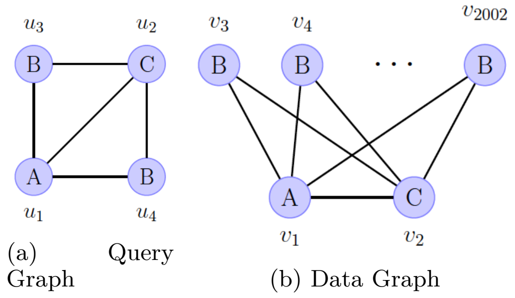


Fig. 2: Duplicate Computation in GSI

Moreover, GSI also introduces a data structure called Signature and PCSR to single out candidates and then do the join. Both data structures have memory overhead and PCSR wastes memory bandwidth to look up the candidates.

IV. Challenges

Challenge I: A typical size of GPU main memory is about a few tens of GB. However, the intermediate results on some large graphs can reach TBs. The GPU systems based on filtering-joining strategy need to obtain all the subembeddings at the the current iteration to proceed to the next iteration. This strategy is equivalent to the BFS exploration. The number of these subembeddings is exponentially proportional to the number of vertices in data graphs. This turns out to be the biggest challenge in a GPU-based subgraph matching system.

Challenge II: A massive amount of data are needed to be fed to these threads in order to guarantee higher hardware utilization and data parallelism. How to make the memory accesses as efficient as possible is critical to the matching performance. The best option is to render all the memory accesses coalesced because GPUs prefer such access pattern with relatively lower cycles. The irregularity of a graph data often makes coalesced memory accesses difficult.

Challenge III: In most GPU-based subgraph implementations, multiple threads will work collectively on a subembedding. The special features of GPU memory hierarchy provide us flexible choices for sharing data among threads. The registers have higher bandwidth and lower latency, buy they are scarce on GPU. Overuse of registers will lead to poor thread occupancy, data parallelism, and hardware utilization. In order to avoid warp divergence, we will normally let a warp extend instances. A warp needs to shared some candidate sets during the matching process. The minimum storage of these candidate sets should be no less than 16 Bytes. The address of a buffer consumes 8 bytes, while the source vertex ID and the number of elements in each buffer need 4 bytes each. If we naively cache the meta information of sets in register, we will not be able to achieve 100% thread occupancy for a pattern with 4 vertices on a graph like Mico. If we cache these set information into the global memory, the memory latency will possibly outweigh the thread occupancy. So how to maintain higher thread occupancy and lower memory latency for large patterns and data graphs is important to the performance.
Challenge IV: It has been shown in Turbo$_{\text{ISO}}$ [7] that backtracking is a successful approach for subgraph matching. Algorithms 1 shows the pseudo-code of naive the backtracking implementation similar to VF2. However, a GPU-based backtracking algorithm is quite challenging since the recursive kernel function is not supported on GPU devices. It would be impossible to directly transform such algorithm into a CUDA kernel.

Algorithm 1: Subgraph Isomorphism

```
   Input: $G(P), G(g)$
   Output: $Results$
1  Function Main($G(P)$, $G(D)$):
2      Initialization: $Results$, $M(p)$;
3      subMor($M(p), G(P), G(D), Results$);
4      return $Results$;
5  ;
6  Function subMor($M(p), G(P), G(g), Results$):
7      if $M(p)$ and $G(P)$ have the same number of
         vertices then
8          Append($M(p), Results$);
9      else
10         $CS$ = getNextCandidate($M(p), G(P)$);
           /* The candidate set for next vertex
              to be matched.                    */
11         for $u$ in $CS$ do
12             $M(p')$ =Extend($M(p), u$);
13             subMor($M(p'), G(P), G(g), Results$);
14             backTracking($M(p')$);
15         end
16 return
```

An alternative and easy solution to the backtracking is the BFS exploration strategy which is adopted by the other GPU-based systems. The advantage of BFS exploration is that the load balance is taken care of by a separate kernel with different launch configurations at each iteration. However, the BFS exploration suffers a lot issues mentioned in Section III which will significantly hurt the performance of a system.
Challenge V: The irregularity of graph data not only incurs uncoalesced memory accesses but also suffers the load balance issues. The load balance is always one of the main concerns in an application's performance. The load balance is more difficult in GPU-based frameworks than that in CPU-based frameworks because thread synchronization and memory sharing on GPUs are more complicated than CPUs.

V. DGSM

In this section, we will present our GPU-based subgraph isomorphism implementation in details and how we sort out the challenges in section IV on GPU devices.

A. Solution to Challenge I

As pointed out in Section IV, the root of large intermediate results is the BFS exploration strategy. Instead of the BFS exploration, we choose the recursive DFS exploration strategy to solve the large memory requirement issue. The benefits of such exploration over BFS are three folds. First, the number of potential instances are limited in DFS exploration. Not the entire subembeddings in the data graph are necessary to move on to the next iteration. Only the subembeddings already discovered by the warps needs to be materialized in the memory. Second, many grid level synchronizations are eliminated. A warp will continue to extends its own subembeddings until the last query vertex is finished. Therefore, no communications among warps or thread blocks are required in DFS exploration. The majority grid level synchronizations in GSI are not present in our implementation. Third, we fuse the kernels in the filter and join phase together. So no control transfer is needed between the host and the device in order to make dynamic memory allocations inside each iteration. With the help of DFS exploration, we are able to preallocate a memory buffer before the matching starts on GPUs.

B. Solution to Challenge II

We use a data structure VertexSet to store the information of a candidate set in order to achieve coalesced memory accesses for set operations. All the possible mappings of a query vertex u are stored in a continuous chunk of memory. Different warps will maintain their own VertexSets. There will be no communication among warps. This structure also makes the backtracking process and references to the VertexSets in previous iterations a lot easier. Most importantly, it supports us to reuse some of VertexSets to avoid duplicate connectivity check in GSI.

```
struct VertexSet{
  uint32_t *data;
  uint32_t vid;
  uint32_t size;
}
```

Listing 1: VertexSet

Therefore, there would be no extra memory overhead and no memory bandwidth will be wasted with the help of the VertexSet structure.

C. Solution to Challenge III

To minimize warp divergence, the threads in a warp work collectively on the embeddings emanated from a candidate vertex. Therefore, they are going to share and work on the same VertexSets. How to share these VertexSets will affect the performance. Caching the meta data of these VertexSets into global memory or registers is not the best solution. If they are stored in thread private registers, the lowest latency is achieved. But the thread occupancy might be reduced significantly for large patterns because each thread needs 16 bytes for one VertexSet. Algorithm 2 shows that 6 VertexSets are required for a pattern with 4 vertices. The computation resource will not be fully saturated without register spilling. There will be many more VertexSets for a larger pattern and it will lead to either register spilling or lower thread occupancy. Putting these VertexSets in the global memory will not reduce thread occupancy, while it incurs higher memory latency to access them. Both options will not help improve the performance for large query patterns. However, GPUs expose a programmable $L1$ cache, called shared memory, to us. The shared memory can be used to shared data among threads in a thread block. So we can resort to the shared memory to reduce the register pressure mentioned above. Caching VertexSets in the shared memory will help us get better occupancy without sacrificing too much latency.

D. Solution to Challenge IV

The backtracking or DFS approach cannot be naively ported to GPUs since recursive kernel is not supported on GPUs. However, we can simulate the backtracking process with the help of the data structure VertexSet and the matching order of a query pattern. DGSM matches a query vertex at a time. So the matching process can be abstracted as a nested for loop. The matching process of the pattern 2a can be written as Algorithm 2, where $N(m_0, C)$ stands for the neighbors of m_0 with label C, $\cap$ is the intersection operation of two sets, and $-$ denotes the difference of two sets. As seen in Algorithm 2, we enforce the partial order through a bound on the target VertexSet c_6. The VertexSet set c_4 will be used for matching both u_3 and u_4. So it should be computed in the nested for loop as earlier as possible to avoid redundant computation. In this case, it is computed inside the second for loop. Therefore, we don't waste memory and hardware for duplicate subembeddings and compute the set intersection twice like GSI for the pattern 2a. The VertexSet c_4, which is the intersection of the neighbor list of m_0 and m_1 with label B, has been calculated in the second for loop. In the third for loop, we will use the result by dereference the VertexSet c_4. It should be emphasized that the overhead of the analysis on a query graph is negligible since the query graph is usually small and the analysis is done on the host side.

Algorithm 2: Backtracking Unrolling

```
1  VertexSet $c_0$ = getCandidate(A);
2  for auto $m_0 \in c_0$ do
3      VertexSet $c_1 = N(m_0, \text{C})$;
4      VertexSet $c_2 = N(m_0, \text{B})$;
5      for $m_1 \in c_1$ do
6          VertexSet $c_3 = N(m_1, \text{B})$;
7          VertexSet $c_4 = c_2 \cap c_3$ ;
8          for $m_3 \in c_4$ do
9              VertexSet $c_5 = N(m_3, \text{B})$;
10             VertexSet $c_6 = \{\text{m} \mid \text{m} \in c_4 - c_5$ and
                 $m < m_3$ };
11         end
12     end
13 end
```

Although recursive calls are not supported on GPUs, we can implement the DFS approach by viewing the backtracking process as a nested for loop and generate a plan based on the nested for loop. A nest for loop can be simulated with an array of counters on GPUs. The plan maintains the topology of subpatterns at each step, specifies the matching order and partial orders among vertices of a query graph, and has an one-to-one correspondence to a nested for loop. Once such a plan is generated, it will be moved to a GPU to guide the computation on the GPU.

E. Solution to Challenge V

The load balance in DFS exploration can be very complicated. A good understanding of the cause of load imbalance helps us find better solutions to this issue. DGSM targets the origin of load imbalance in subgraph matching: the skewness of vertex degrees. With the knowledge of source of load imbalance, we cope with the issue in subgraph matching with three techniques. The first solution is that we reorder the vertex ids according to the degrees. The more neighbors a vertex has, the less its id will be. A graph is stored in the most compacted CSR format. We further sort the neighbors of a vertex based on its label. The neighbors of a vertex with the same label are saved continuously in the edge list and are sorted in ascending order in that memory chunk. Please see Figure 3 for the details of how we organize an edge list. This organization of edges will not only improves the load balance issue, but also validates the design of the VertexSet struct when GPU memory access patterns are concerned. It is the reason why we don't need to preallocate the memory buffers for some VertexSets which are neighbors of vertices and we are able to assign them to addresses of the edge list on the fly. Note that vertex reordering can be done offline and it is done only once. Secondly, a naively parallel strategy is

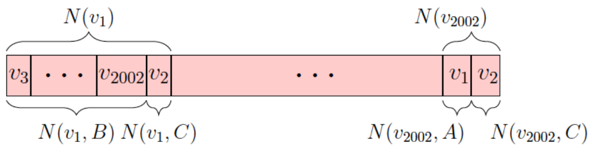


Fig. 3: Edge List

data parallelism on the the set c_0 in Algorithm 2. This strategy might suffer severe load imbalance due to the skewness of vertex degrees shown in the Figure 4. The red wavy arrows indicate the data parallelism on the vertices in the VertexSet c_0. The workload assigned to the thread T_1 is 100 times more than that of T_2; However, perfect load balance is achieved if each edge connecting labels A and C is assigned to a thread indicated by green wavy arrows. In such case, each thread will have the same amount of work. Thirdly, we adopt dynamic scheduling to deal with

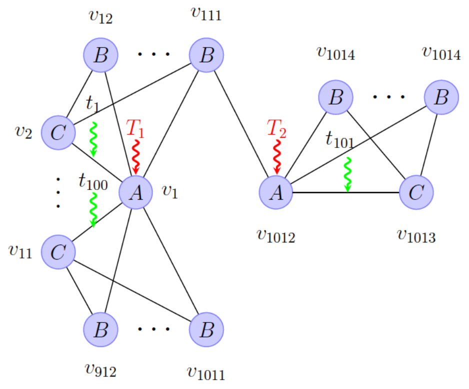


Fig. 4: Example of load imbalance

the potential load balance in deeper iteration.

Putting these solutions together, we are presenting DGSMin Algorithm 3.

Algorithm 3: GPU-based Subgraph Isomorphism

```
    Input: $G(P), G(g)$
    Output: $Results$
1   Function Main($G(P)$, $G(D)$):
2       Initialization: $Results$;
3       Load($G(g)$);
4       Preprocess($G(g)$);
5       allocateMemory($G(P)$);
6       $plan$ = generatePlan($G(P)$);
7       copyH2D($plan$);
8       kernel<<<
          grid, block >>>($plan, G(g), Results$);
9       copyD2H(Results);
10      return $Results$;
11  ;
12  Function
      kernel($plan, G(P), G(g), Results, buffer$):
13      wid = getWarpId();
14      tid = laneid();
15      initSharedMem;
16      init(VertexSet, buffer);
17      if $tid==0$ then
18          eid = atomicAdd(edge);
19      while $eid < num\_edges$ do
20          Extend($plan, eid, VertexSets$);
21          if $tid==0$ then
22              eid = atomicAdd(edge);
23      end
24  return
```

F. Other Optimizations

DGSM has two main components, plan generator on line 6 and matching kernel line 8 in Algorithm 3. The main purpose of plan generator is to devise a matching order which has more pruning power. The matching order is determined by the connectivity analysis on the patter graph only. Please see the paper [18] for the details about how the matching order is generated. This approach considers multiple paths through a query vertex at a time during the extension, while DAF proceeds with one path at a time. The benefit of our approach is that our system will backtrack immediately once one path containing the query vertex fails. DAF might waste computation on some full paths before the failing path. This ordering also takes care of the non-tree edges on the fly. Moreover, it serves to remove redundant computation. Note that redundant computation comes from three perspectives. First of all, there are duplicate instances when symmetries exist in a pattern graph. One way to get rid of these duplicates is to do an automorphism test once all the subembeddings in the current iteration are obtained. However, an automorphism test is very expensive. An efficient parallel automorphism test on both CPUs and GPUs is still an open question for now. In order to avoid the expensive automorphism test, we will do an analysis on the pattern graphs first to find all the vertices which are interchangeable. Then we impose a partial order on the IDs to each group of symmetrical query vertices to break the symmetries in the pattern. Each symmetrical query vertices is equivalent to the NEC class in Turbo$_{\text{ISO}}$. Take the query graph Figure 2a as an example. It is pointed out that two vertices u_3 and u_4 are interchangeable. Exchange the mapping of these two vertices will lead to the same subgraph. If we specify the partial order on the two vertices shown at the bottom of Figure 5 to the pattern, we will remove the duplicate embeddings of the query in the data graph Figure 2b. In this case, only the mapping $\{u_1 \to v_2, u_2 \to v_1, u_3 \to v_4, u_4 \to v_3\}$ is valid, while the other mapping $\{u_1 \to v_2, u_2 \to v_1, u_3 \to v_3, u_4 \to v_4\}$ violates the partial order requirement $u_3 < u_4$. Here we assume that the subscript in the data graph is the id for a vertex.

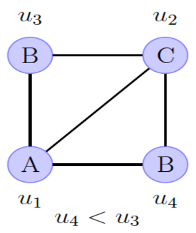


Fig. 5: Pattern with Partial Orders

Secondly, the DFS strategy allows us to preallocate memory buffesr for the VertexSets of each warp even for large query patterns and data graphs. Each VertexSet owns a continuous chunk in the memory buffer. This preallocation is only done once before the kernel starts. It will free us from the duplicate join computation in Gunrock and GpSM. Also it will not incur memory allocation overhead for each query vertex in GSI.

Thirdly, there exists duplicate connectivity computation in some patterns. The query vertices u_3 and u_4 in Figure 5 are both connected to the query vertices u_1 and u_2. GSI will do set intersections with matched vertices of u_1 and u_2 twice to get the candidate embeddings, once for u_3 and once for u_4. With the help of our VertexSet data structure, we only need to compute duplicate sets once [19]. We cache the candidate of u_3 and u_4 in the VertexSet c_4 and directly retrieve the results when we match u_4. Besides redundant computation, our plan already encodes the connectivity among query vertices. It leads to a unique subpattern at each matching step. Therefore, automorphism tests can be safely avoided.

After GPU receives the plan, the kernel will be launched right away. We assign a warp to collectively working on some embeddings to reduce the warp divergence as much as possible. It can be seen that we need to instantiate 7 VertexSets even for a pattern graph with 4 vertices. The register file size on our GPU is 256KB and each SM supports maximum 2048 threads. It leads to the fact that each thread can use up to 32 registers to achieve 100% occupancy. Obviously, registers will rapidly be consumed up by a larger query graph. The application will suffer either lower occupancy or register spills for large query graphs. Both cases are not what we expect. The former results in lower data parallelism, while the latter has higher memory latency. To overcome the register pressure, we use dynamic shared memory to cache the meta data of these VertexSets for fast accesses. The VertexSets will be visited millions of times, so it is perfect to cache the meta data in the shared memory. We also use the shared memory to store a counter for each VertexSet (c_1 and c_4 in this case) which we will iterate through. Figure 6 shows the layout of 6 VertexSets of the query Figure 2a for a thread block. Bearing the organization of the edges shown

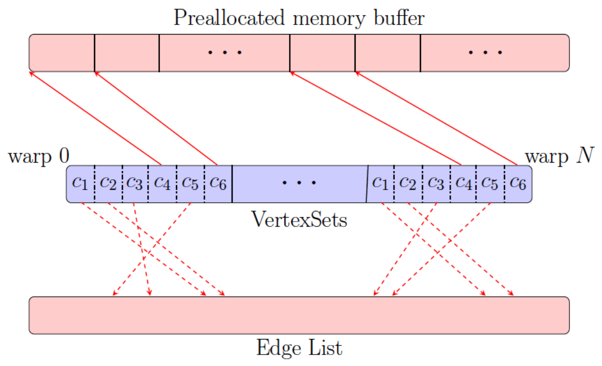


Fig. 6: Arrangement of VertexSets

in Figure 3 in your mind, it would be reasonable to see why we could use the neighbor list as the buffers for some VertexSets.

Incorporating all these techniques together into Algorithm 3, we implement our subgraph matching application, which is able to do both vertex-induced and edge-induced matching. Algorithm 2 gives the process of vertex-induced matching. For the edge-induced matching, the difference is that we only need to take the VertexSet c_4 bounded by m_3.

VI. Evaluation

We will present our experiment results in this section. Both vertex-induced matching and edge-induced matching are implemented in our system. We will show the performance of both matching schemes and their difference using some patterns in motifs and cliques. Then we will demonstrate the performance benefits after we incorporate the techniques we introduced in the section V. Finally, we will evaluate the performance of our system by comparing against with the-state-of-the-art CPU-based subgraph matching system DAF and GPU-based system GSI on some random query graphs.

A. Experiment Setup

We are going to evaluate our system on 7 real-world graphs in Table I. All of our experiments are conducted on a host with Intel(R) Xeon(R) CPU E7-4830v3 2.10GHz 24-core CPUs and 256GB DRAM equipped with an NVIDIA TITAN V GPU 12GB of RAM. The GPU has an array of 80 streaming processors and each SM can support up to 2048 threads. We use the NVCC compiler version 11.0.221 (g++ version 7.5.0) with O3 to compile all the programs. The operating system is Ubuntu Linux 18.04 with Linux kernel version 5.4.0. The GSI source code is available and can be obtained from its codebase on GitHub. DAF is not

Dataset (G)	$\lvert V(G) \rvert$	$\lvert E(G) \rvert$	$d_{max}(G)$
Enron	36692	367662	1383
Amazon	1134890	1851744	549
DBLP	317080	2099732	343
Mico	96638	2160312	1359
Patents	3774768	33037894	793
Livejournal	4846609	85702474	20333
wiki-Vote	7115	201524	1065

TABLE I: Statistics of Datasets

open-sourced but the executables are available on GitHub. The default number of threads is set as 24 when DAF is concerned.

B. *Effectiveness of Our Techniques*

We will first show the effectiveness of our techniques on unlabeled subgraph matching on some patterns since the an unlabeled query bears more the symmetries and the number of instances than a labeled one for a certain query graph. The baseline of our implementation is that all the VertexSet meta data are stored in registers, data parallelism is carried out on vertices, and static scheduling is chosen. Without using the shared memory to cache VertexSet meta data, only thread 50% occupancy can be achieved because of aggressive usage of registers in the baseline implementation. Table II presents the baseline performance of our system answering patterns in motif-4 on the graphs in Table I. The conventions are that PA, ST, RE, CR, TT, and CL stand for the patterns path-4, star-4, rectangle, chordal rectangle, tailed triangle, and clique-4 in motif-4.

Graph	PA	ST	RE	CR	TT	CL
Livejournal	34.9	1288.74	8.95	6.9	150.1	4.03
Patents	2.83	3.99	2.67	2.61	2.62	2.78
Mico	0.37	6.43	0.215	0.48	10.97	0.23
DBLP	0.2	0.41	0.2	0.19	0.83	0.21
Amazon	0.21	0.63	0.2	0.2	0.28	0.22
Enron	0.47	3.98	0.13	0.16	0.74	0.07
wiki-Vote	0.12	2.81	0.05	0.04	0.43	0.03

TABLE II: Baseline Performance(Second)

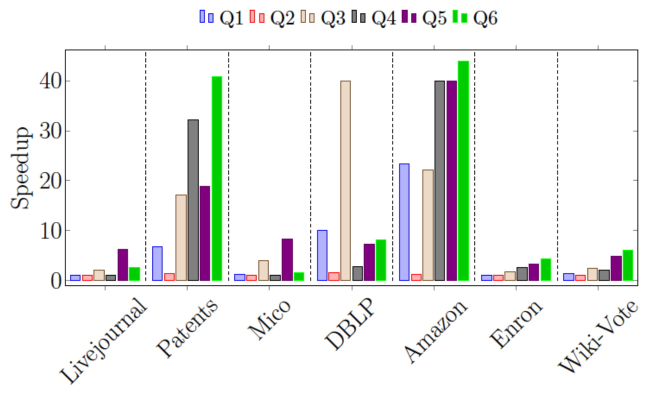


Fig. 7: Speedup utilizing shared memory

To show the benefit of each technique, we will present the speedup of implementations incorporating different techniques against the baseline. Fig. 7 shows speedup after implementing the shared memory optimization. There is improvement for each query pattern. Some queries can obtain a tremendous speedup on Patents, DBLP, Amazon because of higher thread occupancy. The speedup on patterns $Q1$ and $Q2$ are not as great as other patterns. The reason behind it is that the computation on hot vertices dominates the execution and leads to load imbalance. It can be seen from Fig. 7 that the improved shared memory implementation is an order of magnitude faster than the baseline on average.
Fig. 8 shows the speedup of the implementation of dynamic scheduling. It can be seen that the speedups has been improved with dynamic scheduling by looking at Fig.7 and Fig. 8. The skyline of the speedups in Fig. 8 is the similar to that in Fig. 7. It tells us that although dynamic scheduling is helpful for load imbalancing to a certain extent, dynamic scheduling alone cannot solve the load imbalance of hot vertices. Fig. 9 illustrates the

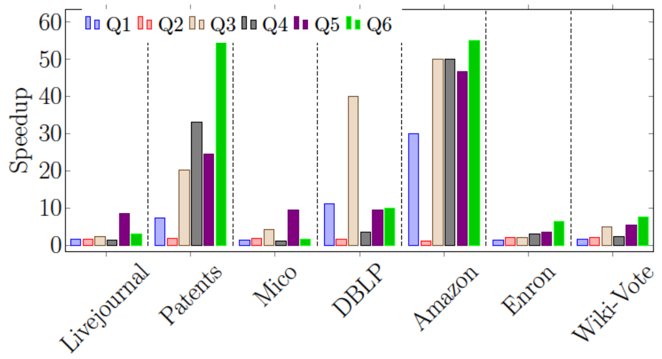


Fig. 8: Speedup of Dynamic Scheduling

final speedup after we adopt the following techniques including shared memory, dynamic scheduling and the edge mapping shown in Fig. 4. Table X lists the

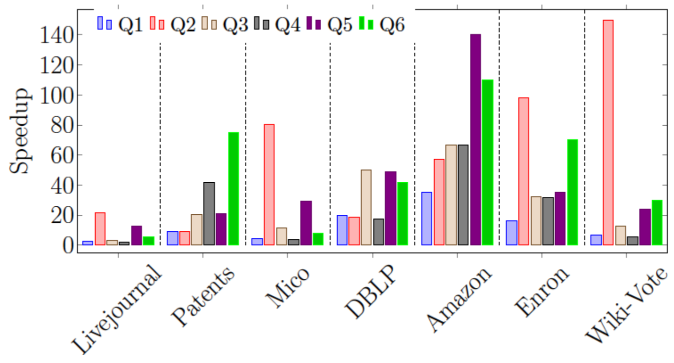


Fig. 9: Speedup of Edge Parallelism Mapping

performance of vertex-induced matching incorporating the above techniques for the same patterns in Table II. As mentioned earlier, our system can not only perform

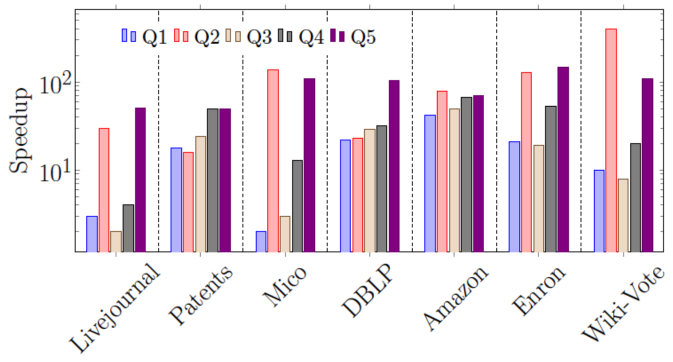


Fig. 10: Edge-induced Matching Speedup

vertex-induced matching, but also carry out edge-induced query answering.

The edge-induced matching results for the same patterns are presented in Table III. The difference between

Graph	Motif-3	Motif 4				
		PA	ST	RE	CR	TT
Livejournal	113	17494	65169	4471	1663	2959
Patents	24	153	256	113	53	52
Mico	2	202	91	83	38	101
DBLP	1	9	18	7	6	8
Amazon	0.9	5	8	4	3	4
Enron	0.3	31	64	7	3	5
wiki-Vote	0.2	17	14	6	2	4

TABLE III: Edge-induced Matching Performance(ms)

vertex-induced and edge-induced matching is whether the missing edges between vertices are verified or not. The vertex-induced matching needs to validate missing edges, while edge-induced does not need this process. Table III introduces the performance of edge-induced matching when the vertices of a query is less than 4. In general, the edge-induced matching outperforms the vertex-induced matching. This fact can been seen by compare Table X and Table III. However, the edge-induced matching will discover more patterns than the vertex-induced one. The performance discrepancy between two matching approaches depends on the balancing between the missing edge check and the difference in the number of discovered instance. There are only a few edge-induced queries slower than the corresponding vertex-induced queries. The reason is that those queries have a larger ratio of the number of edge-induced instances to the vertex-induced instances. Since the triangle pattern is trivial, we combine the wedge and triangle performance together in motif-3.

Fig. 10 illustrates the speedup of the edge-induced matching to the baseline and Fig. 11 shows the speedup of dynamic scheduling to the static scheduling for edge-induced matching. For clique-4, there are no missing edges. So its edge-induced performance is the same as vertex-induced matching. Therefore, we do not show the speedup for the clique-4 pattern. We cut the speedup of Q4 on Enron down to 2 in Fig. 11. Its actual value is 14.

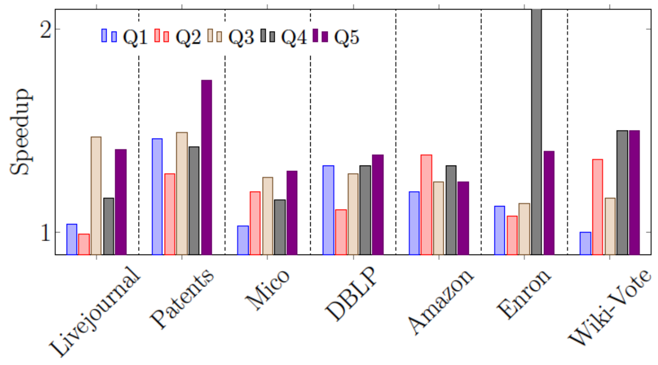


Fig. 11: Edge-induced Dynamic vs Static Speedup

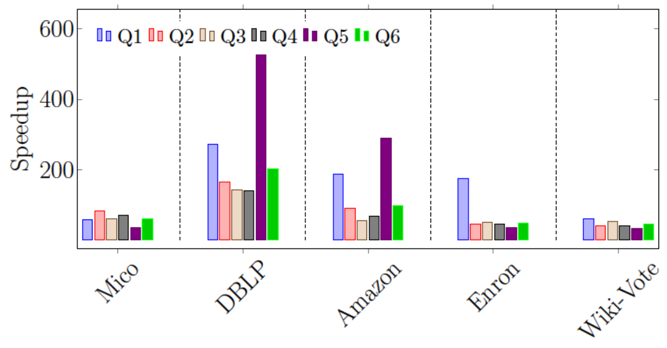


Fig. 12: Speedup with DAF

C. Comparison with DAF and GSI

DAF and GSI are not able to do the vertex-induced matching. For the fairness, we compare our edge-induced matching performance with theirs. DAF sets a threshold of the number of instances of a pattern in a data graph. The default value of that threshold is 10^4. Table IV presents the performance of our system once that threshold is reached. Fig. 12 shows the speedup compared against DAF. Our

Graph	PA	ST	RE	CR	TT	CL
Livejournal	2.26	0.29	1.81	1.87	2.37	1.52
Patents	0.55	0.23	0.6	0.62	0.8	0.72
Mico	0.34	0.22	0.28	0.32	0.37	0.28
DBLP	0.22	0.28	0.24	0.26	0.23	0.24
Amazon	0.3	0.29	0.4	0.34	0.31	0.59
Enron	0.3	0.3	0.29	0.32	0.3	0.33
wiki-Vote	0.27	0.23	0.27	0.29	0.28	0.25

TABLE IV: Performance with threshold (ms)

system is 108X faster than DAF on average. The minimum and maximum speedup is 32 and 527, respectively. The DAF suffers the load imbalancing issue presented in Fig. 4 since DAF maps each thread to a vertex in a candidate set.

As far as the unlabeled graphs and queries are concerned, GSI cannot process many patterns in motif-4 even for a

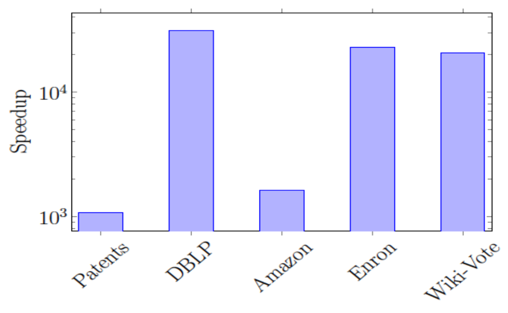


Fig. 13: Speedup of Clique 4 with GSI

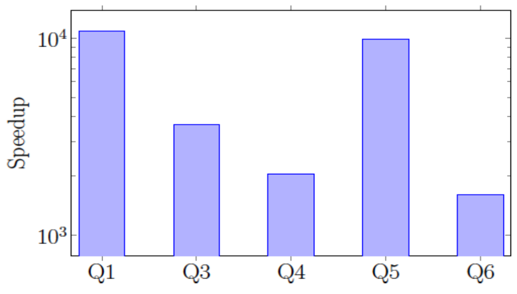


Fig. 14: Speedup of 5 queries in Motif-4

relatively small graph, wiki-Vote, because of the memory limitation. GSI is able to answer clique-4 query on most the graphs in the dataset except Mico and Livejournal. The query time of clique-4 on Patents, DBLP, Amazon, Enron, and wiki-Vote is 39.5, 153.3, 3.2, 23, and 20.6 in second. Fig. 13 presents the speedup of our system with GSI. GSI can answer 5 queries out 6 in motif-4. The corresponding speedup is shown in Fig. 14. Our performance is at least 3 orders of magnitude faster than GSI for motif-4 query graphs. We profile GSI for these

Graph	PA	ST	RE	CR	TT	CL
Livejournal	23920	92253	2633	3129	12716	768
Patents	311	427	130	81	123	68
Mico	106	153	19	121	371	28
DBLP	10	22	5	11	17	5
Amazon	6	11	4	5	7	4
Enron	40	84	4	5	21	1
wiki-Vote	24	37	4	7	18	1

TABLE V: Vertex-induced Matching Performance(ms)

query graphs and find there are 42 kernel calls on average in GSI on data graph Amazon ranging from 39 to 47. So many grid level synchronizations and three memory allocations hurt the performance significantly. It is the reason why our implementation is much faster than GSI. Beside patterns in motif-4, we also measure the performance of our system on dense queries clique 5 through 10. Their performance is reported in Table VI and Table VII. The cliques 8 through 10 take over a day to answer the queries for data graphs Livejournal and Mico. Therefore, we don't report their performance in the paper.

Graph	C5	C6	C7	C8	C9	C10
Patents	0.06	0.08	0.14	0.23	0.29	0.32
Amazon	0.004	0.004	0.004	0.004	0.004	0.004
DBLP	0.035	0.7	15.62	331.13	5632.65	78513.3
Enron	0.006	0.037	0.131	0.343	0.704	1.255
wiki-Vote	0.004	0.013	0.043	0.117	0.234	0.398

TABLE VI: Clique Performance(s)

Graph	C5	C6	C7
Livejournal	22.79	1075.96	44205.7
Mico	0.96	34.6	1255.98

TABLE VII: Clique Performance(s)

Since our system is able to find all the instances of a query pattern in a data graph, we will know the count of those instances. Therefore, we can compared our performance with mining frameworks, for instance the CPU-based AutoMine and GPU-based Pangolin.

D. Labeled Graph Performance

For labeled queries, we randomly assign a unique label out of 10 labels to a vertex of a data graph in Table I. We generate 6 sparse query patterns through a random walk on the labeled data graphs. Table VIII and Table IX shows the performance of these 6 queries on the data graphs. The performance of Q6 on Livejournal and Mico is excluded because it takes more than 1 day to answer these two queries. Table X shows the performance of 6

Graph	Q1	Q2	Q3	Q4	Q5	Q6
Patents	13.9	44.65	72.01	55.3	562.91	12908.8
DBLP	1.98	4.08	31.5	305.13	4425.91	46942.5
Amazon	0.68	0.6	1.3	1.8	0.8	1.72
Enron	0.94	7.43	58.95	125.03	727.63	22009.4
wiki-Vote	10.94	53.88	164.01	674.96	5500.78	98658.9

TABLE VIII: Performance of 6 Labeled Patterns(ms)

Graph	Q1	Q2	Q3	Q4	Q5
Livejournal	87.44	6460.88	51320.1	5988970	113077000
Mico	45.85	131.1	3370.69	67786.6	1521.49

TABLE IX: Performance of 5 Labeled Patterns(ms)

labeled queries when the threshold is specified. Fig. 15

Graph	Q1	Q2	Q3	Q4	Q5	Q6
Livejournal	0.42	0.96	3.92	1.1	1.2	1
Patents	1	0.58	0.84	5.4	5.27	5.28
Mico	0.39	0.43	0.44	0.44	0.5	0.49
DBLP	0.38	0.39	0.42	0.41	0.41	0.42
Amazon	0.35	0.4	0.39	0.4	0.4	0.46
Enron	0.12	0.12	0.13	0.13	0.14	0.13
wiki-Vote	0.08	0.08	0.09	0.09	0.1	0.1

TABLE X: Labeled Performance with Threshold(ms)

shows the speedup of these 6 queries to DAF. DAF cannot answer queries on Livejournal and Patents. Therefore, their speedups are not displayed in the Fig. 15. We achieve 57X speedup with DAF on average. The speedup ranges

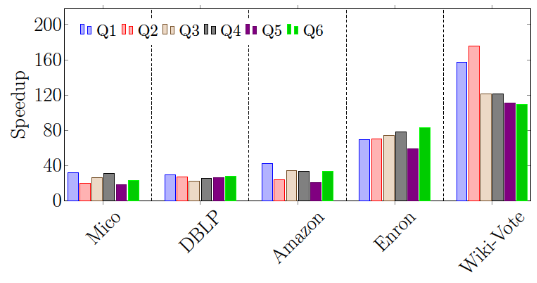


Fig. 15: Speedup with DAF for Labeled Queries

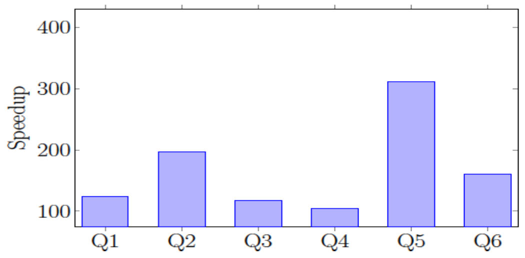


Fig. 16: Speedup of Labeled Queries on Amazon with GSI

from 18X to 795X. For GSI, no query can be done on Livejournal. The first 2 queries can be answered on most of the rest data graphs. All the 6 queries can be answered on Amazon. The average speedup is 160X. The range is from 104X to 311X. Fig 16 shows the speedup on these 6 queries. The average speedup of the two queries is about 95X. Fig. 17 presents the speedup of the first two queries where the missing bar denote the query cannot be answered.

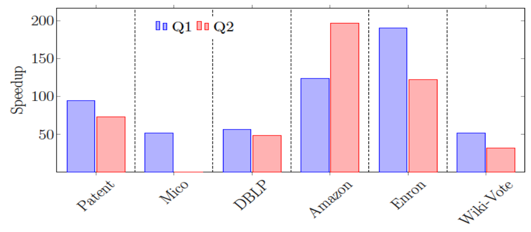


Fig. 17: Speedup of Two Queries with GSI

VII. Conclusion

In this paper, we introduce our new GPU-based subgraph matching system. The system is able to answer both vertex-induced and edge-induced query. The DFS exploration grants our system the ability to handle large data graphs and queries. After adopting the shared memory and load imbalance optimizations, our system is boosted significantly. Our system outperforms the two state-of-the-arts systems including both CPU-based and GPU-based application. Our system is about 2 and 3 orders of magnitude faster than DAF and GSI for unlabeled queries, while it is 2 orders of magnitude faster on labeled queries.

References

[1] J. Raymond and P. Willett, "Maximum common subgraph isomorphism algorithms for the matching of chemical structures," *Journal of computer-aided molecular design*, vol. 16, pp. 521–33, 08 2002.

[2] S. Noel, "A review of graph approaches to network security analytics," in *From Database to Cyber Security - Essays Dedicated to Sushil Jajodia on the Occasion of His 70th Birthday*, ser. Lecture Notes in Computer Science, P. Samarati, I. Ray, and I. Ray, Eds., vol. 11170. Springer, 2018, pp. 300–323. [Online]. Available: https://doi.org/10.1007/978-3-030-04834-1_16

[3] A. Messina, A. Fiannaca, L. La Paglia, M. La Rosa, and A. Urso, "Biograph: a web application and a graph database for querying and analyzing bioinformatics resources," *BMC Systems Biology*, vol. 12, no. 5, p. 98, 2018. [Online]. Available: https://doi.org/10.1186/s12918-018-0616-4

[4] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data," *BMC Bioinformatics*, vol. 14, 2013. [Online]. Available: http://doi.acm.org/10.1145/2749469.2749475

[5] G. Bouritsas, F. Frasca, S. Zafeiriou, and M. M. Bronstein, "Improving graph neural network expressivity via subgraph isomorphism counting," 2021.

[6] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976. [Online]. Available: http://doi.acm.org/10.1145/321921.321925

[7] W. Han, J. Lee, and J. Lee, "Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. ACM, 2013, pp. 337–348. [Online]. Available: https://doi.org/10.1145/2463676.2465300

[8] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 1199–1214. [Online]. Available: https://doi.org/10.1145/2882903.2915236

[9] M. Han, H. Kim, G. Gu, K. Park, and W. 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 Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 1429–1446. [Online]. Available: https://doi.org/10.1145/3299869.3319880

[10] L. Wang and J. D. Owens, "Fast gunrock subgraph matching (gsm) on gpus," 2020.

[11] H.-N. Tran, J.-j. Kim, and B. He, "Fast subgraph matching on large graphs using graphics processors," in *Database Systems for Advanced Applications*, M. Renz, C. Shahabi, X. Zhou, and M. A. Cheema, Eds. Cham: Springer International Publishing, 2015, pp. 299–315.

[12] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang, "Gsi: Gpu-friendly subgraph isomorphism," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1249–1260.

[13] J. Lee, W. Han, R. Kasperovics, and J. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *Proc. VLDB Endow.*, vol. 6, no. 2, pp. 133–144, 2012. [Online]. Available: http://www.vldb.org/pvldb/vol6/p133-han.pdf

[14] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 10, pp. 1367–1372, 2004. [Online]. Available: https://doi.org/10.1109/TPAMI.2004.75

[15] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proc.*

VLDB Endow., vol. 8, no. 5, p. 617–628, Jan. 2015. [Online]. Available: https://doi.org/10.14778/2735479.2735493

[16] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, “Taming verification hardness: an efficient algorithm for testing subgraph isomorphism,” *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 364–375, 2008. [Online]. Available: http://www.vldb.org/pvldb/vol1/1453899.pdf

[17] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu, “Graph homomorphism revisited for graph matching,” *Proc. VLDB Endow.*, vol. 3, no. 1–2, p. 1161–1172, Sep. 2010. [Online]. Available: https://doi.org/10.14778/1920841.1920986

[18] D. Mawhirter, S. Reinehr, W. Han, N. Fields, M. Claver, C. Holmes, J. McClurg, T. Liu, and B. Wu, “Dryadic: Flexible and fast graph pattern matching at scale,” in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021, pp. 289–303.

[19] W. Han, “Optimizing graph analyses on gpus,” PhD dissertation, Colorado School of Mines, 2021.