



# cuTS: Scaling Subgraph Isomorphism on Distributed Multi-GPU Systems Using Trie Based Data Structure

Lizhi Xiang
Washington State University
Pullman, Washington, USA
lizhi.xiang@wsu.edu

Arif Khan
Pacific Northwest National Lab
Richland, Washington, USA
ariful.khan@pnnl.gov

Edoardo Serra
Boise State University, PNNL
Boise, Idaho, USA
edoardoserra@boisestate.edu

Mahantesh Halappanavar
Pacific Northwest National Lab, WSU
Richland, Washington, USA
hala@pnnl.gov

Aravind Sukumaran-Rajam
Washington State University
Pullman, Washington, USA
aravind_sr@outlook.com

## ABSTRACT

Subgraph isomorphism is a pattern-matching algorithm widely used in many domains such as chem-informatics, bioinformatics, databases, and social network analysis. It is computationally expensive and is a proven NP-hard problem. The massive parallelism in GPUs is well suited for solving subgraph isomorphism. However, current GPU implementations are far from the achievable performance. Moreover, the enormous memory requirement of current approaches limits the problem size that can be handled. This work analyzes the fundamental challenges associated with processing subgraph isomorphism on GPUs and develops an efficient GPU implementation. We also develop a GPU-friendly trie-based data structure to drastically reduce the intermediate storage space requirement, enabling large benchmarks to be processed. We also develop the first distributed sub-graph isomorphism algorithm for GPUs. Our experimental evaluation demonstrates the efficacy of our approach by comparing the execution time and number of cases that can be handled against the state-of-the-art GPU implementations.

**ACM Reference Format:**
Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: Scaling Subgraph Isomorphism on Distributed Multi-GPU Systems Using Trie Based Data Structure. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3458817.3476214

## 1 INTRODUCTION

A graph, $G = (V, E)$ is a pair of vertex set $V$ representing unique entities and edge set $E$ representing binary relations on $V$. Two graph are said to be *isomorphic* or equivalent if there exists a bijection function on the vertices of the two graphs, such that any two vertices in the first graph are adjacent if and only if the bijected vertices are adjacent in the second graph. We study the *subgraph isomorphism* problem in this paper, which is an important primitive that is used to find patterns in graphs. Given two graphs, $\mathcal{G}$ (data graph) and $\mathcal{Q}$ (query graph), subgraph isomorphism can be defined as the task of finding all subgraphs in $\mathcal{G}$ that are isomorphic to $\mathcal{Q}$ (formal definitions are provided in §2). It is widely used in various domains such as bioinformatics[2], computer vision[4], social network analysis[5], chem-informatics[2]. For example, in [11], the authors use subgraph isomorphism to identify network motifs that can characterize common patterns occurring in biological networks such as protein-protein interactions and networks from other domains.

Even though the subgraph isomorphism problem is inherently parallel, the sheer amount of work and memory access required make it an expensive computational problem. The performance of many real-world applications is often limited by subgraph isomorphism, and the performance of state-of-the-art techniques is limited. Hence, optimizing it for modern architectures is an *essential and challenging problem.*

Due to its massive parallel processing power, high bandwidth, and low power requirements, GPUs are well suited for solving subgraph isomorphism. However, there are several challenges associated with achieving high-performance for subgraph isomorphism in GPUs: **i)** irregular memory access patterns; **ii)** huge memory requirement; **iii)** highly imbalanced workload; and **iv)** challenging memory placement. Graph isomorphism algorithms have to identify nodes in $\mathcal{Q}$ which can potentially match the nodes in $\mathcal{G}$. At each step of this process, we find common neighbors of multiple nodes of both $\mathcal{G}$ and $\mathcal{Q}$, all of which need not be laid out contiguously in memory. Depending on the processing strategy and data structure used, this can result in irregular memory access patterns. Moreover, since the number of neighbors of different nodes can vary significantly, assigning one thread to process each node will result in a huge load imbalance. This is especially true for GPUs, where the workload imbalance can result in intra-warp, inter-warp, inter thread-block, inter-grid and inter-node level load imbalance. Figuring out where to place the partial results computed by each thread is yet another important challenge. Since the number of intermediate results is not known apriori, the memory placement decisions have to be taken during the program execution (instead of predetermined memory placement), which requires expensive synchronization and/or atomic operations. Moreover, the intermediate



*SC '21, November 14–19, 2021, St. Louis, MO, USA*

ACM ISBN 978-1-4503-8442-1/21/11.
https://doi.org/10.1145/3458817.3476214

results should be kept in a format that minimizes the memory footprint and, at the same time, offers good access efficiency. Solving these challenges requires novel innovations in both data-structures and GPU algorithms.

In this paper we present a high-performance subgraph isomorphism algorithm and a new data-structure for GPUs. The contributions can be summarized as follows:

- A trie based data structure which allows to *reduce the required memory footprint* and simultaneously maintain *good memory access efficiency.*
- A *high throughput*, GPU architecture aware algorithm which provides *good load-balancing* and *good data reuse.*
- An efficient GPU mirco-kernel design for fast intersections
- A distributed GPU implementation which allows to *handle bigger data/query graphs* with *low intra node communication and synchronization overhead.* To the best of our knowledge this is the first distributed GPU implementation for subgraph isomorphism.
- A comprehensive evaluation of our approach against the state-of-the-art approaches: Our cuTS framework can handle 154 test cases out of 198(33 query graphs and 6 data graphs), whereas GSI succeeded for 99 test cases. For these 99 test cases, cuTS achieved a geomean speedup of 386 on the A100 machine and 312 on the V100 machine.

The rest of the paper is organized as follows. Section 2 presents the necessary background to understand the rest of the paper, and Section 3 gives an overview of existing subgraph isomorphism works. Section 4 gives an overview of our approach. Section 4.1 describes our single-node algorithm, and Section 4.2 describes our distributed GPU solution. Section 6 compares our implementation's performance with the state-of-the-art approaches, and Section 7 presents the conclusions.

# 2 BACKGROUND

## 2.1 Definitions

Definition 1 (Graph). *A graph is denoted by $\mathcal{G} = (V, E)$, where $V$ represents the set of vertices and $E \subseteq V \times V$ represents the set of edges.*

In this work, we consider directed graphs (edges are directed). A given undirected graph $\mathcal{G}_1(V, E_1)$ can be converted to directed graph $\mathcal{G}_2(V, E_2)$, by adding an edge $(v, u)$ for every edge $(u, v) \in E_1$.

Definition 2 (Isomorphism). *Two graphs $\mathcal{G}_1(V_1, E_1)$ and $\mathcal{G}_2(V_2, E_2)$ are isomorphic if there exists a bijective function $f : V_1 \mapsto V_2$ such that for any two vertices $u \in V_1$ and $v \in V_1$ are adjacent in $\mathcal{G}_1$ if and only if $f(u)$ and $f(v)$ are adjacent in $\mathcal{G}_2$. In otherwords, $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$ and $(u, v) \notin E_1 \Leftrightarrow (f(u), f(v)) \notin E_2$. Isomorphism is denoted by $\simeq$.*

In simple words, isomorphism shows the equivalence between two graphs. Subgraph isomorphism tries to find the equivalence between a subset of one graph and another graph.

Definition 3 (Subgraph Isomorphism). *Two graphs $\mathcal{G}_1(V_1, E_1)$ and $\mathcal{G}_2(V_2, E_2)$ are subgraph isomorphic if there exists a subgraph of $\mathcal{G}_1$, $\mathcal{G}_1^{'}(V_1^{'}, E_1^{'}) \mid V_1^{'} \subseteq V_1$ and $E_1^{'} \subseteq (E_1 \cap (V_1^{'} \times V_1^{'}))$ such that $\mathcal{G}_1^{'}$ and $\mathcal{G}_2$ are isomorphic. Subgraph isomorphism is denoted by $\cong$.*

In this paper, we solve the challenging task of subgraph isomorphism search on GPUs.

Definition 4 (Subgraph Isomorphism search). *Given two graphs $\mathcal{G}$ and $\mathcal{Q}$, find all subgraphs $\mathcal{G}^{'}$ of $\mathcal{G}$ such that $\mathcal{Q}$ and $\mathcal{G}^{'}$ are isomorphic. $\mathcal{G}^{'}$ is called as a match of $\mathcal{Q}$.*

Figure 1 (A) shows a sample data graph. (B) and (C) shows query graphs that are isomorphic to (A). A query graph can have multiple matches on the data graph. E.g., all triangles in (A) *match* (B). Graph (D) is not isomorphic to (A). Intuitively, since Vertex $d_6$ has a degree (in degree and out degree) of 5, which is higher than the maximum degree (in degree and out degree) all nodes of graph (A), the graphs are not isomorphic.

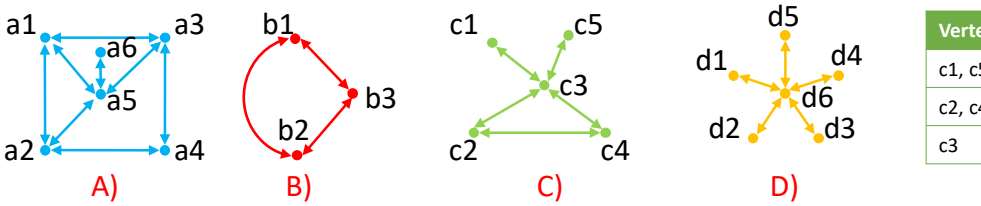
Definition 5 (Candidate set). *Given a data graph $\mathcal{G}(V_1, E_1)$ and a query graph $\mathcal{Q}(V_2, E_2)$, the set of all vertices $u \in V_1$ such that $degree(v) \leq degree(u)$, where $v \in V_2$, is called as the candidate set of $v$.*

Figure 1 E) shows the candidate sets for each vertex in (C) for the data graph (A). Since the degree of $c_1$ and $c_5$ is one, any node with a degree greater than or equal to one can be considered as a candidate. $c_2$ and $c_4$ can only match nodes with degree greater than or equal to two, and $c_3$ can only match nodes with degree greater than equal to 4.

## 2.2 GPU Architecture

*2.2.1 Thread hierarchy:* The threads in a GPU are grouped hierarchically. A group of 32 threads is called a warp. All threads in the same warp will execute the same instruction. If different threads in a warp follow different control flow paths, the non-active threads for a given path are masked during that path's execution (see performance factors for more details). Different warps are grouped together to form thread blocks. Each thread block is mapped to a streaming multiprocessor (SMP) which consists of a set of execution units, L1 cache/shared memory, and a register file. The CUDA API provides a collection of primitives to synchronize between different threads in a thread block efficiently. A group of thread blocks forms a grid. Each kernel launch is mapped to a grid, and multiple SMPs will collectively execute the work in a grid.

*2.2.2 Memory hierarchy:* Registers are the fastest memory units in the GPU memory hierarchy. Each thread owns a set of private registers. All threads in the same warp can efficiently exchange data contained in their registers using warp primitives. Different threads in a thread block share a common L1 cache/shared memory. Typically, the collective register capacity of a thread block exceeds the shared memory capacity. Hence it is best to keep the elements shared by the thread block in shared-memory and keep thread-local data in registers. The entire GPU shares one common L2 cache across all the threads. Global memory (DRAM) sits above this L2 cache. GPU threads cannot directly access the CPU memory. CUDA supports Unified Memory Access (UMA) which provides a single address space view of CPU memory and (multiple) GPU memory. In addition to the single address space view, on Nvidia Pascal and later architectures, UMA supports virtual addressing and on-demand paging, enabling the GPU to address the entire system memory. In other words, without UMA, the total size of memory




| Vertex | Candidate set |
|---|---|
| c1, c5 | a1, a2, a3, a4, a5, a6 |
| c2, c4 | a1, a2, a3, a4, a5 |
| c3 | a5 |

E)

**Figure 1: Examples of data graph along with query graphs and candidate set: A) A sample data graph; B and C) sample query graphs that are subgraph isomorphic to graph A; D) sample query graph which is not isomorphic to graph A; E) candidate set of query graph C for data in graph A.**

that can be allocated on the GPU is limited to the DRAM capacity of the GPU. In contrast, with UMA, we can allocate memory much higher than the GPU DRAM memory size. The GPU driver will automatically take care of migrating the pages between the CPU and the GPU.

*2.2.3 Performance factors: Data movement minimization* is one key consideration while designing GPU algorithms. Efficient implementations should judiciously distribute the data structures over the entire memory hierarchy and thereby achieve maximum data reuse. In addition to data reuse, coalescing is another factor that determines data movement. The memory accesses from different threads in a warp are aggregated into transactions. Typically, but not necessarily, contiguous threads accessing contiguous memory locations will result in a minimal number of possible transactions (coalesced access). Adjacent threads accessing non-contiguous data can result in uncoalesced memory accesses, increasing the number of required memory transactions. For subgraph isomorphism, the memory accesses are dominated by the neighborhood queries and storage of partial matches. Hence, it is vital to ensure that the memory accesses corresponding to these data structures are coalesced as much as possible.

*Load imbalance* is another critical factor that affects performance. In GPUs, load imbalance manifests in different forms. Intra warp divergence, or simply warp divergence, occurs when different threads in the same warp follow different control flow paths. In the latter case, each path's execution will be sequential (interleaving instructions from different paths in the same warp is allowed in modern GPUs). Warp divergence will result in resource underutilization. Inter-warp load imbalance occurs when the work distribution among different warps in the same thread block is not balanced. Since GPU resources are allocated at the thread block level, an SMP cannot replace a finished warp with a new one until all the warps in a thread block finish execution. If different thread blocks have a non-uniform amount of work, it may expose tail effects in GPUs, where only a few SMPs are active towards the end of the kernel execution. For a multi-GPU implementation, we also need to balance the workload between different nodes while *minimizing the synchronization* requirements to achieve the best performance.

*Occupancy* is another factor that determines performance. It is defined as the ratio of the active threads to the maximum number of threads that an SMP can support (1024 or 2048 in modern GPUs). A code with low occupancy typically exposes memory latency and thereby will only achieve limited performance. Occupancy is affected by shared-memory usage, register usage, and thread block size. Holding more data in shared memory, especially when tiling, allows better data reuse; however, this may reduce the occupancy. The challenge in developing an efficient sub-graph isomorphism GPU algorithm is to find the balance between all these performance factors.

## 3 RELATED

Prior subgraph isomorphism works can be classified into two categories. The first category uses the depth-first search-based strategy, and the second category uses the breadth-first search-based strategy. The depth-first search strategy keeps joining the next vertex to the current partial path until no match can be found. If the search did not result in a match, it would backtrack to a previous step. This strategy has a linear memory complexity with respect to the number of vertices of the query graph. Compared with the breadth-first strategy, reduced memory complexity is the most significant advantage of the depth-first strategy. However, depth-first search strategies are hard to parallelize since it is recursion-based and is not GPU-friendly. On the other hand, the Breadth-first strategy favors parallelism and can fully use GPU's massive threads. In breadth-first-based strategies, the massive parallelism will produce a volume of intermediate results, potentially hitting the memory limit. The latter is a significant challenge, especially for GPUs with limited memory capacity.

**CPU-based Work:** To the best of our knowledge, Ullmann[14] is considered as the first subgraph isomorphism framework and is based on the depth-first strategy. Other frameworks such as VF2[4], VF3[3], QuickSI[12], GraphQL[7], GADDI[18], and SPath[19] can be considered as extensions of the Ullmann algorithm. Like Ullman, VF2 also follows the filtering and join strategy but adds more pruning rules derived from the graph connectivity properties. QuickSI refines the query graph's searching order to access the vertex with the most infrequency label as fast as it can. GraphQL and GADDI further prune out the candidates by putting neighborhood information into consideration. Unlike other frameworks, SPath verifies each subpath instead of one vertex at each recursive call, which decreases the depth of the search tree.

**GPU-based Work:** GPSM[13], GUNROCK[15], GSI[17] are the most recent GPU frameworks for subgraph isomorphism. These three GPU works follow a breadth-first strategy that favors GPU architecture. The GPSM work tries to search for the next candidate by considering all the neighbors of a previously verified candidate. They use a single warp to read the neighbor list and thereby achieve good memory coalescing. However, for data graphs with a small

average degree, most of the threads will be idle and thereby underutilize the available resources (thread idling). Furthermore, their approach will result in a huge load imbalance for graphs with a lot of variance in the degree distribution. Compared to GPSM, we use a hybrid BFS-DFS strategy which enables a high degree of parallelism without increasing the overall memory required. Our trie-based data structure also offers additional compression of intermediate storage, which further reduces the memory requirement. In addition, our work virtual warp strategy helps to minimize thread idling. GSI is a recent work, and it is the fastest among these three. Similar to GPSM, GSI also assigns each candidate to a warp; hence it also suffers the same load imbalance issue. In addition, GSI doesn't have an efficient way to store the tons of intermediate results, which results in memory overflow. The advantages of our approach over GSI are similar to that of GPSM. In addition, we adaptively choose the intersection method, which enables higher performance. Gunrock uses optimized intermediate storage representation, which relies on encoding the path to a 64-bit integer. However, this approach is not scalable as it requires $|V_D|^{|V_Q|} < 2^{64}$, where $|V_D|$ and $|V_Q|$ are the number of nodes in the data graph and query graph respectively. As an example, consider a data graph with a million nodes ($10^6$). Gunrock can only support query graphs with a maximum of four vertices. More importantly, Gunrock relies on a pass-by-pass approach, where each pass will read (and possibly write) to global memory, which reduces performance. The cuTS approach uses trie to represent the data; hence it can support much bigger query graphs than Gunrock. Moreover, we use a fused single-phase approach which removes the need for repeatedly loading the same data from global memory.

# 4 cuTS

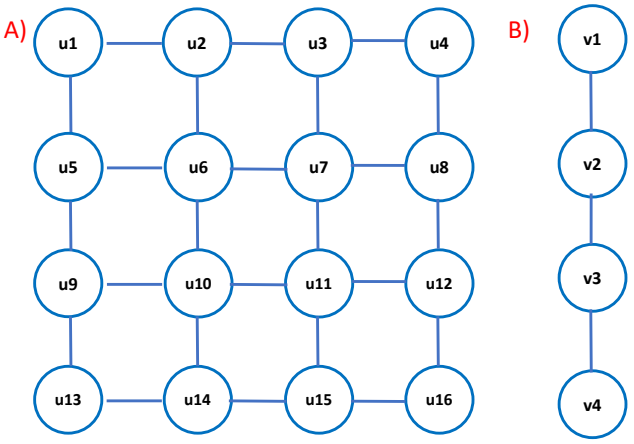
In this section, we present the design of our high-performance subgraph isomorphism algorithms for GPUs. Our design is motivated by one or more of the following limitations of existing approaches: **i)** inability to process massive data graphs and query graphs; **ii)** requires repeated computations – uses two-pass algorithms; **iii)** uses space inefficient data structures to keep the intermediate matches – a structure which is responsible for most of the data movement and memory requirement; **iv)** lack of distributed GPU support; and **v)** limited performance.

The overall design objective of cuTS is to support big datasets and query graphs without sacrificing performance. To this end, the algorithm design considers i) memory access efficiency, ii) space requirement, and iii) load-balancing. For simplicity, we explain the concepts using directed graphs. However, our approach works for both directed and undirected graphs. We use $\mathcal{D}(V_D,\ E_D)$ to denote the data graph and $\mathcal{Q}(V_Q,\ E_Q)$ to denote the query graph. Our overall approach is based on vertex matching. We construct each candidate by matching one node at a time (details in Section 4.1.2). Each such partially matched set, called *partial path*, is a part of a possible subgraph isomorphism match. We begin the search process by identifying the node with the maximum degree in $V_Q$. If there are multiple such nodes, the node with the minimum node id (a simple heuristic to break the tie) is selected. Let $q_0 \in V_Q$ denote this node. This node will serve as the root node for all our matches. Let $C(q_0)$ denote all the possible candidates of $q_0$ in $\mathcal{D}$, i.e., $C(q_0) = \{d_0 \mid d_0 \in V_D \ \wedge \ (degree(q_0) \leq degree\ (d_0))\}$. In the next step, we want to find a match for $q_1$, where $(q_0, q_1) \in E_Q$. If there are multiple candidates for $q_1$, we select one node with maximum degree (out degree in the case of directed graphs). For each $d_0$, we compute the candidate for $q_1$ as $C(q_1) = \{d_1 \mid d_1 \in V_D \ \wedge \ (degree(q_1) \leq degree(d_1)) \ \wedge \ (d_0, d_1) \in E_D\}$. This process is repeated till all the nodes of $\mathcal{Q}$ are matched. If next node $q \in \mathcal{Q}$ that we add to the partial path $P$ has a subset of neighbors which are already in $P$ ($\{p \mid p \in P \ \wedge \ (p, q) \in V_Q\}$), then the candidate of $q$ – ($C(q)$), should be connected to all the elements of the candidate set of $P$. This requires multiple intersection operations. The efficiency of maintaining the intermediate path data structures along with the intersection efficiency determines the overall performance of the implementation. In Section 4.1 we describe our data structure and the associated benefits and challenges for a single node system. We also compare it against existing approaches (the high-level differences are explained in Section 2). It also describes our hierarchical load balancing algorithm. In Section 4.2 we describe our distributed algorithm and our distributed load balancing algorithm.

For undirected graphs, we assume that the query graph $\mathcal{Q}$ and the data graph $\mathcal{D}$ are connected (for directed graphs, we assume that the graphs are weakly connected). If the query graph is not connected (weakly connected), but the data graph is, we split $\mathcal{Q}$ into a set of connected (weakly connected) components, compute sub-graph isomorphism on each of these components, and produce the final result as the cross product of individual solutions. If the data graph is not connected (weakly connected), but the query graph is, we will split $\mathcal{D}$ into a set of connected (weakly connected) components, compute sub-graph isomorphism on each of these components, and produce the final result as the union of individual solutions. A combination of the above approaches can be used for the case where both $\mathcal{Q}$ and $\mathcal{G}$ are not connected (not weakly connected).

## 4.1 Single Node

*4.1.1 Data Structure:* Data structure plays a key role in determining the memory footprint. Holding intermediate results in GPUs with limited memory is an arduous challenge. Each of the partial matches consists of a set of nodes the query graph $\mathcal{Q}$ and their corresponding candidates in the data graph $\mathcal{D}$. For query graphs with simple structures, such as a chain, the number of partial paths increases non-linearly as we increase the path length. For illustration, consider a simple mesh data graph and linear chain query graph as shown in Figure 2 (A) and (B). The first node of the query graph ($v_1$) can match any of the vertices of the data graph ($u_*$). In the second step, there are as many partial paths as two times the number of edges in $\mathcal{D}$. Intuitively, the total number of paths of length 1 is equal to the total number of edges (note that the edges are unidirectional). Each of these partial paths of length 1 has to keep track of two vertices (candidate vertices). In general, each partial path has to keep track of ||partial_path|| vertices. Figure 2(C) shows the storage requirement at each depth. In real-world graphs, the rate at which the number of partial paths (and thus the storage requirement) grows as a function of the degree of the graph. Table 1 shows the actual storage requirement for the Enron dataset for a fully connected network with five nodes as the query graph.




C)

| Partial path depth | Candidates | Storage size requirement (words) |
|---|---|---|
| 1 | 16 | 16 |
| 2 | 48 | 48×2 = 96 |
| 3 | 96 | 96×3 = 288 |
| 4 | 192 | 192×4 = 368 |

**Figure 2: Figure illustrating the storage requirement: A) data graph; B) query graph; C) number of candidates and storage size for each path depth.**

| partial path depth | naive storage (words) | our storage (words) | compression ratio |
|---|---|---|---|
| **1** | 16514 | 33028 | 0.5 |
| **2** | 631318 | 647832 | 0.974509 |
| **3** | 13485244 | 9217116 | 1.463065 |
| **4** | 237996028 | 121472508 | 1.959258 |
| **5** | 3723609628 | 1515717948 | 2.456664 |

**Table 1: Storage space comparison between the naive approach and our approach for the Enron dataset for a fully connected network with five nodes as the query graph.**

A careful reader might have observed that we could use a trie-based data structure, which can be implemented using Compressed Sparse Fibre (CSF) format (an extension of the CSR format) and thereby reduce the storage requirement considerably. Figure 3(b) summarizes CSF representation and our representation. Assuming that the nodes and edges do not hold any meta information such as labels, the CSF format can be implemented using two arrays per level (i) nodeid array – holds the id of each node (dark blue), and (ii) index array – holds the starting location (and ending location) of its neighbors (light blue). For example, at level 1, position 0, we have a node with id $u_3$, its neighbours will start at location $index\_array[0] == 0$ and will end at location $index\_array[1] == 3$ (end is non-inclusive). The challenge here is in determining the exact write location in parallel. Consider Figure 3 and assume that is the query graph is same as Figure 2 (B). We are trying to find candidates for $v_3$ for the partial path where $v_2$ is matched to $u_3$. Note that level 3 and level 4 of CSF are not yet constructed at this point in time. To construct level 3, we scan the neighbors of the node $u_3$ and detect the number of neighbors. The highlighted red polygon in Figure 3 shows the elements involved in this operation. Only after that can we determine the write location for level 3 for the partial path where $v_1$ is matched to $u_4$. One way to solve this problem is to use parallelism only within a partial path (the order of neighbors of $u_3$ at level 4 does not matter). However, this strategy introduces other problems: i) thread idling: Massive parallel architectures like GPUs requires massive parallelism to achieve high performance; restricting ourselves to process one vertex at a time will result in poor performance; ii) unnecessary synchronizations: after processing each vertex the entire threads in GPU have to synchronize which limits the performance. Some prior approaches have solved this problem using a two-stage algorithm where the first stage computes the intersection and simply counts the number of matches per node. The second stage recomputes the intersection and uses the count array to determine the exact write location. Thus the computations and, more importantly, the data-movement operations are performed twice.

We propose a data structure that is space-efficient and, at the same time, avoids re-computations. Our data structure is built on the realization that as the length (number of nodes) of the partial match path increases, the number of times the same prefix match path is used also increases. Our data structure is capable of reusing all prefix paths without the disadvantages of CSF. Figure 3 (C) shows the details of our data-structure. We first allocate two big arrays whose size equals half of the free space available in the GPU. We use the CUDA *cudaMemGetInfo* to determine the available free space. The first array is called parent array (PA) – orange –, is used to store the parent index of the current candidate, and the second array, called the candidate array (CA) – dark blue – is used to store the current candidate id. With our format, a thread processing the partial path where $v_1$ is matched to $u_4$ doesn't have to wait for the thread where $v_1$ is matched to $u_3$. This is mainly because we explicitly store the parent id; hence, the children of both paths can be written in an interleaved manner, which is not possible in the case of CSF data structure, which requires all children of the same node to be laid out contiguously in memory. The level 3 in Figure 3 (C) is based on the $[u_2, u_4, u_6, u_7, u_1, u_3]$ order; however any permutation of the previous list such as $[u_6, u_4, u_7, u_2, u_1, u_3]$ and $[u_2, u_7, u_4, u_1, u_6, u_3]$ is valid. Compared to CSF, our representation requires slightly more space. E.g., the highlighted red box can be represented using six words, whereas our representation requires seven words. *However, our strategy only requires an atomic operation to find the write location, offers a high degree of parallelism, and has good storage efficiency.*

The space-saving of our representation exploits the fact that many paths share common parents. If we represent paths using trie, the shared parents only need to be represented once. Figure 2 and Table 1 shows some examples of compression ratios. Before we jump into the actual data representation, we present a theoretical analysis to compute the estimated savings. If the query graph $Q$ has $N$ vertices, then the maximum length of partial paths is $N$. Let $P_l$ represent the set of partial paths at depth $l$ and let $|P_l|$ denote the

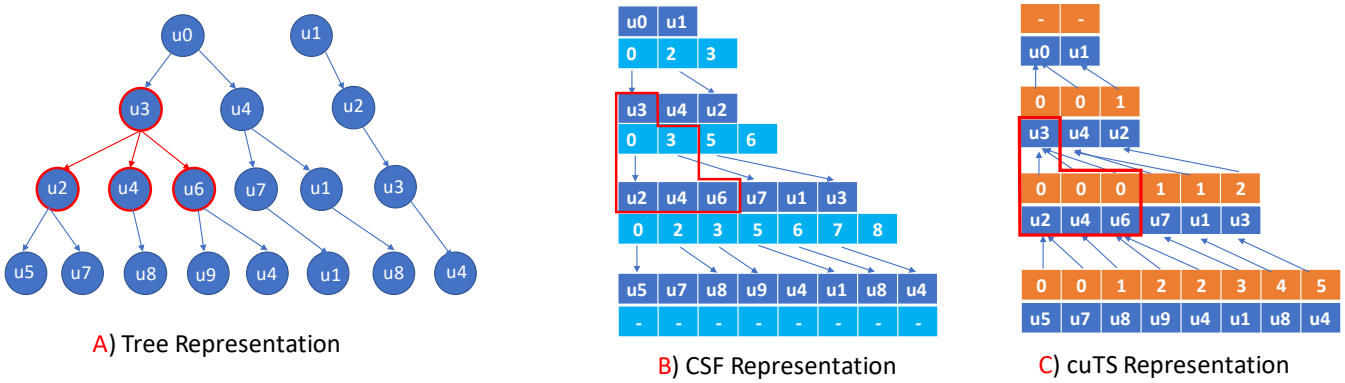



**Figure 3: Simplified view of multiple data representations**

number of partial paths at depth $l$. Let $\delta$ represent the maximum degree of the graph. The maximum number of total paths at depth $l+1$ can be estimated as $|P_l| \times \delta$. However, many of these paths may not satisfy the degree constraint and hence will be invalid. Let $\sigma_l$ denote the ratio of total paths to valid path at depth $l+1$ . $|P_{l+1}|$ can be bounded by $|P_l| \times \delta \times \sigma_l$. In other words, if we assume that there are there are $|P_1|$ initial candidates, the maximum number of partial path at depth 2 is $|P_2| = |P_1| \times \delta \times \sigma_1$; the maximum number of partial path at depth 3 is $|P_3| = |P_2| \times \delta \times \sigma_2$. The total number of paths at depth $l$ can be summarized as:

$$\begin{aligned} |P_l| &= |P_{l-1}| \times \delta \times \sigma_{l-1} = (|P_{l-2}| \times \delta \times \sigma_{l-2}) \times \delta \times \sigma_{l-1} \\ &= (|P_1| \times \delta \times \sigma_1) \times (\delta \times \sigma_2) \times (\delta \times \sigma_2) ... \times ... (\delta \times \sigma_{l-1}) \\ &= |P_1| \times \delta^{l-1} \times \prod_{i=1}^{l-1} (|\sigma_i|) \end{aligned} \tag{1}$$

For simplicity, assume that the fraction of valid paths at all levels is a constant ($\sigma = max(\sigma_i)$). Under this assumption and denoting $\delta \times \sigma$ as $ds$, Equation (1) can be re-written as:

$$|P_l| = |P_1| \times ds^{l-1} \tag{2}$$

Traditional representations for $\beta$ paths of depth $l$ requires $\beta \times l$ space. Using Equation (2), the space requirement for traditional method for paths of depth $l$ ($S\_trad_l$) can be summarized as:

$$S\_trad_l = |P_l| \times l = |P_1| \times ds^{l-1} \times l \tag{3}$$

Let $S\_cuTS_l$ represent the space requirement of our representation for paths of depth $l$. $S\_cuTS_1$ is equal to $|P_1|$ (the number of initial candidates). If there are $x$ partial paths at depth 2, we need $x$ storage for leaf nodes and $|P_1|$ space for its parent ($|P_1|$ + x). If there are $y$ partial paths at depth 3, we need $y$ storage for leaf nodes and $|P_2|$ space for its parent ($|P_2|$ + y). In general, the space requirement at level $l$ can be computed as:

$$\begin{aligned} S\_cuTS_l &= S\_cuTS_{l-1} + |P_l| \\ &= S\_cuTS_{l-2} + |P_{l-1}| + |P_l| \\ &= S\_cuTS_1 + |P_2| + |P_3| + ... + |P_l| \\ &= |P_1| + |P_2| + |P_3| + ... + |P_l| \\ &= (|P_1|) + (|P_1| \times ds^1) + (|P_1| \times ds^2) + (|P_1| \times ds^{l-1}) \\ &= (|P_1|)((ds^{l-1} - 1)/(ds - 1)) \end{aligned} \tag{4}$$

Assuming that the number of paths at depth $l$ is greater than number of paths at depth $l-1$, Equation (4) can be rewritten as:

$$\begin{aligned} S\_cuTS_l &< (|P_1|)(ds^{l-1})/(ds-1) \\ &< \frac{S\_trad_l}{l \times (ds-1)} \end{aligned} \tag{5}$$

Equation (4) shows that our representation has a space reduction factor of $l \times (ds-1)$. This matches our intuition. The average saving factor at each level is approximately equal to $ds$. There are l such levels. Hence the average savings factor is $l \times (ds-1)$. Also note that, as $l$ and $ds$, the number of computations also increases and so does the advantage of our representation. Hence our representation is ideal for large problem sizes which also matches the use case scenario for GPUs.

*4.1.2 Approach:* Let $S = \{q_0, q_1, q_2, ...\}$ be the order in which the query graph nodes are selected for matching. Let $P = \{p_1, p_2, p_3, ...\}$ be the set of partial paths. Let $n_i$ be the length of each partial path. Each $p_i$ can be thought about as a function which maps a node $v$ in the query graph to a node $u$ in the data graph ($pi(v_j) \mapsto (u_k)$). As mentioned in the middle of Section 4, we select $q_0$ as the node with the maximum out degree in the query graph (use node id to break the tie) and find the candidates for $q_0$. These set of candidates forms the initial set of partial paths. Note that the candidates for a lower degree query graph node includes all the candidates of a higher degree query graph node and some additional matches. By choosing to match the node with maximum out degree we reduce the set of initial matches. Each partial path ($p_i$) is then extended as follows : i) select the next node ($q_{(n_i+1)}$) in the query graph to be matched and ii) compute all neighbours ($next\_neigh$) of $q_{(n_i+1)}$ which are

already matched. Note that the next query node to be matched is selected independently of the partial path. Hence the *next_neigh* set will be the same for all partial paths of same length iii) the next set of matches (*matches*) can be computed by taking the intersection of all $p_i(next\_neigh_j)$; iv) append each of the *matches* to the current partial path and save it such that it can be processed in the next iteration; (v) repeat all the above steps until all elements of $S$ are matched or till the set of partial paths to be processed becomes empty. The costliest operation here is to compute the intersection. Since we use the CSR data structure to represent the data graph, finding the neighbors for performing the intersection can be done with $O(1)$ time cost.

Assigning one warp to process a single partial path can result in severe thread-idling. The latter is one of the major limitations of the prior works. The average degree of most graphs is less than 32. This means that some of the threads in a warp won't have any work to process. Hence compute cycles and resources allocated to them are wasted. We explored two ways to resolve this problem. One strategy is to group the paths based on the work into bins and use virtual warps to process the bins. For example, if the number of neighbors of each node to be intersected is eight, put this partial path in bin eight. In the next iteration, all paths in bin eight will be processed by a virtual warp of 8 threads; each warp will process four paths, significantly reducing thread idling. However, the challenging aspect of this approach is that we have to predict the amount of space assigned to each bin to store the intermediate results. A uniform partitioning of the buffer space is not a good option. Depending on the graph characteristics, most of the bins may be empty. The memory space assigned to empty bins is wasted. One could try an alternative strategy of splitting the buffer based on the number of paths in each bin; however, the number of paths is not a direct indicator of the buffer space required. More importantly, employing any of the latter two approaches may prevent us from processing big graphs that otherwise could have been compatible. There are additional challenges associated with binning. The binning code should itself be very efficient, and we have to append the results from different bins to compress the buffer space, which requires additional memory transactions.

Hence, we chose another strategy where we still use virtual warps to process the elements, but instead of having multiple bins, we only use one bin, and the size of the virtual warp is determined by the average degree of the node. Our experiments show that this strategy achieves better performance and has better compatibility with larger graphs. However, even with this strategy, we also observed considerable intra-warp and intra thread block load imbalance. Our analysis showed that, in most cases, this was an artifact of how the data graphs are created and due to the fact that we process the elements in the order of node ids. We randomized the partial path placement, and this simple strategy helped us achieve good intra-warp and intra thread block load balance.

Algorithm 1 shows the pseudo-code of our approach. Lines 4 to 14 represent the outer iteration. Lines 8 to 11 handle the first iteration, which forms the initial set of partial paths. For the rest of the iterations, we call the search kernel. In line 19, we declare a shared-memory buffer for performing intersections. For each partial path that this thread is responsible for, we first find the query graph nodes already present in the partial path, which are connected

```
1  //Input: Query graph Q(V_Q, E_Q) , Data graph D(V_D, E_D)
2  //Output: Matching table M
3
4  N = | Q |
5  V = V_Q ordered by decreasing order of degree
6  for n = 0 to N
7    v = V[n]
8    if n == 0:
9      for u in D :
10       if Degree(u) >= Degree(v)
11         append u to M
12   else:
13     search_kernel_GPU( M ,n, Q , D ,V)
14
15 func search_kernel_GPU ( M ,n, Q , D , V):
16   start = blockId/virtual_warp_size * blocks + warp_id;
17   workers = #virtual_warps_per_block * blocks;
18   //shared memory buffer used for intersection
19   shared intersection_buffer[#virtual_warps_per_block];
20   buffer = intersection_buffer[warp_id];
21   prev_M = extract_paths_that_this_thread_should_process( M )
22   for (m = start; m < |prev_M|; m += workers):
23     //extract one partial path
24     S = prev_M[m];
25     //global memory write loc
26     last_location = memory_location(S[n-1]);
27     for i = 0 to |S|:
28       u = S[i]
29       if (V[i], V[n]) ∈ E_Q :
30         if buffer[warp_id] = {}:
31           buffer[warp_id] = neighbors[u];
32         else:
33           warp_level_intersection(buffer[warp_id],neighbors[u
               ]);
34     for v` in buffer:
35       write(v`,last_location) to M
```

**Algorithm 1: Simplified psuedocode corresponding to our approach**

to the current query graph node. In lines lines 24 to 33, we then compute the intersection of the candidates of these nodes (details in Section 4.1.3). Lines 34 and 35 write the final output to global memory.

**Hybrid scanning:**As mentioned earlier, the DFS scanning strategy requires less space than the BFS; however, the DFS strategy has low parallelism and hence is not suitable for GPUs. Our approach uses a hybrid DFS-BFS strategy. We start with a BFS scan which gives us a lot of partial paths. These partial paths are then chunked, and the GPU will process one chunk at a time. This step corresponds to the DFS scan (for each chunk, we are going deeper). The GPU then expands each partial path corresponding to a given chunk using the BFS strategy. Once a chunk is fully processed, the resulting matches

```
1  //Input: vertices to be intersected (a_1, a_2, a_3...a_χ),
2           Graph D,
3           D.cs(x) denotes the children of x
4           D.ps(x) denotes the parent of x
5  //Output: Common children set of the intersecting vertexes
6
7  func scatter_vector_intersection():
8      interset[] = {}; sv[V] = {0}
9      for C ∈ (D.cs(a_1), D.cs(a_2), ...D.cs(a_χ))
10     //intersect
11         for v ∈ C:
12             sv[v]++
13     //collect
14     for v ∈ D.cs(a_1) :
15         if sv[v] == χ:
16             interset.append(v)
17     return interset
18
19 func c−intersection():
20     interset1[] = {}
21     //initial set
22     for v ∈ D.cs(a_1) :
23         interset1.append(v)
24     //intersect
25     for C ∈ (D.cs(a_2), D.cs(a_3), ...D.cs(a_χ))
26         interset2[] = {}
27         for v ∈ C:
28             if v ∈ interset1:
29                 interset2.append(v)
30         interset1 = interset2
31     return interset1
32
33 func p−intersection():
34     interset1[] = result[] = {}
35     //initial set
36     for v ∈ D.cs(a_1) :
37         interset1.append(v)
38     //intersect
39     for v ∈ interset1:
40         if (a_2, a_3...a_χ) ⊂ D.ps(a_v)
41             result.append(v)
42     return result
```

**Algorithm 2: Simplified pseudocode corresponding to the three intersection methods**

are written out, and the next chunk is loaded. This strategy allows us to process bigger graphs without sacrificing performance. A smaller chunk size enables our framework to process bigger graphs; however, if the chunk size is too small, there won't be enough work to occupy all the GPU cores. Hence we only use chunking for big graphs. We empirically found that that chunk size of 512 achieves a good performance.

*4.1.3 Micro kernel intersection design:* Efficient intersection of neighbors is arguably the most important computational part of subgraph isomorphism. In this subsection, we compare multiple intersection strategies (Algorithm 2) and describe our efficient GPU implementation and intersection selection mechanism. The first strategy is similar to the Scatter Vector (*SV* ) approach used in SpGEMM[6] (*SV* is not used for intersection). Assume that we want to intersect children of $\chi$ vertices and assume that the maximum out-degree is $\delta$. First, we initialize the *SV* array of $|V|$ elements to zero and the result array to the empty set (Line 8). Then for each node to be intersected, we traverse through its children and increment the corresponding element in the *SV* array by 1 (Lines 9-12). After processing all nodes, all the elements in *SV* whose corresponding count is $\chi$ represents the final result. Instead of scanning this entire array, we simply traverse through the children of the first node (a super-set of intersection results) and extract the elements whose count is $\chi$. This method is very efficient as as it only has a global memory data movement complexity of $O(\chi \times \delta)$, time complexity of $O(\chi \times \delta)$, and space complexity of (single-core) of $O(|V|)$ . However, this approach is not practical for GPUs as the space complexity in the parallel case is $O(|V| \times \xi)$ where $\xi$ is the number of parallel processors (warps in the case of GPUs – extremely high).

The next method (Lines 19 to 31) describes the *c-intersection*, which is better suited for GPUs. First, we load the children of $a_1$ to a buffer (Lines 22-23). In our GPU implementation, a warp will read all the children in a coalesced manner. Since the buffer is accessed multiple times, for access efficiency, we maintain it in shared memory. For each remaining node, we load its children and check whether they are present in the buffer(*interset*1); if yes, we add it to a temporary buffer(*interset*2). In our GPU implementation, the threads in a warp directly load the children from global memory to registers in a coalesced manner which are then reused multiple times while checking if the loaded elements are present in the current intersection array (Lines 27 to 29). At the end of each loop iteration in Line 25, we copy the temporary array to the intersection array (Line 30). The data movement complexity of this method is $O(\chi \times \delta)$, and the sequential time complexity is $O(\chi \times \delta^2)$; however the space complexity in the sequential case is $O(\delta)$ and parallel cases is $O(\delta \times \xi)$ which is much lower than the *SV* method.

Next, we explain our *p-intersection* method (Lines 33 to 42). The main idea here is similar to the *c-intersection* method, but instead of intersecting children, we intersect parents. First we load the children of $a_1$ to a buffer (*intersect*1) (Lines 36-37). Then for each element in the buffer, we check if its parent set includes all the $a_2$ to $a_\chi$. If yes, we append it to the result array and we repeat this process for all elements in (*intersect*1). The GPU implementation details are similar to that of *c-intersection* method. Assuming that the maximum in-degree is $\delta_{in}$, the data movement complexity of this method is $O(\delta + (\delta - 1) \times \delta_{in})$, and the sequential time complexity is $O(\delta + \delta \times \delta_{in} \times \chi)$; the space complexity in the sequential case is $O(\delta)$ and parallel cases is $O(\delta \times \xi)$ which is much lower than the *SV* method.

## 4.2 Distributed Memory

**Algorithm 3:** Work distribution in the distributed implementation

```
Input: Q,D, rank, num_nodes
Let M denote the matching table
Let Q denote the query graph
let D denote the data graph
global_free_count = 0;
global_processes_count = N;
M = init_match(Q,D, rank)
depth = 1;
while !all_GPUs_free() do
    while depth<|Q| do
        split M to chunk[1],chunk[2],chunk[3]
        search_kernel(chunk[1], depth, Q, D)
        free_process = check_free_GPU()
        if free_process != -1 then
            send chunk[2] to free_process
            search_kernel(chunk[3],depth,Q,D)
        end
        else
            searching_kernel(chunk[2],depth,Q,D)
            searching_kernel(chunk[3],depth,Q,D)
        end
        depth++
    end
    broadcast_free(rank) if req = listen_process_request()
     then
        receive_work()
        //adjust depth and other parameters and begin
          processing of received work
    end
end
```

One of the major limitations of the single-node approach is the ability to handle large data graphs, especially when simple query graphs are used as the number of possible candidates is very large. We solve this problem using a distributed implementation. To the best of our knowledge, this is the first distributed implementation available for sub-graph isomorphism. The fundamental idea is to distribute the computation of partial paths across different nodes. The major challenge here is intra-node load balance. We explored two strategies to solve this. The first strategy is to synchronize all the compute nodes after each outer iteration (i.e., after finding all the partial paths with a given depth). Different compute nodes can then exchange the number of remaining partial paths (not the actual path data) and then distribute the partial paths evenly across each node. However, this strategy has two main disadvantages: i) wasted compute cycles and ii) incompatibility with the cuTS representation( without sacrificing the ability to handle large graphs and performance). Even though this strategy guarantees that the number of partial paths per node is the same in every iteration, the amount of total work in each iteration can vary drastically. Therefore nodes that have finished their work share have to wait for the other nodes to complete their work before proceeding. The second challenge comes from the fact that cuTS is based on the trie representation. Sending partial paths from one node to another requires us to extract the entire path from our data-structure, send it to the other node, and then integrate it to its own local trie. The latter is challenging and will require expensive copying. Another option is to send the entire trie to the other node. However, in this case, the receiving nodes should have enough space to hold their local trie and the incoming trie, which limits the ability to work with big graphs.

| Network | Vertices | Edges |
|---|---|---|
| enron | 36,692 | 367,662 |
| gowalla | 196,591 | 1,900,655 |
| roadNet-PA | 1,088,092 | 1,541,898 |
| roadNet-TX | 1,379,917 | 1,921,660 |
| roadNet-CA | 1,965,206 | 2,766,607 |
| wikiTalk | 2,394,385 | 5,021,410 |

**Table 2: Properties of data graphs used in our experiments.**

Our actual implementation solves the above problem using a different work distribution strategy. The main idea is to let each compute node complete its entire work without any synchronization. First, we split each outer iteration into several chunks. Each chunk is processed sequentially. At the end of each chunk, every busy node checks if any free node is available to share the workload. A node is considered as free if it has finished all its work (across all iterations). Once the node finishes all its work, it saves the final results and discards its local trie. Then it broadcasts a message to all other nodes to tell that it has finished its execution. A node that has not finished processing will then send a portion of its work to the free node along with the trie, and this whole process is repeated till the entire work is done. In order to ensure that required synchronizations are minimum, we developed a mini asynchronous protocol, built on top of the MPI framework, to exchange the required information. Using this protocol, we ensure that only one busy node sends data to a given free node, and a given busy node only sends data to one free node. Algorithm 3 describes the entire process. For simplicity we assume that the work at each iteration is only divided into three chunks.

## 5 TIME COMPLEXITY

**Sequential**: Let $V_D$ and $V_Q$ denote the number of vertices in the data graph and query graph, respectively. Let $\delta$ denote the maximum degree of the data graph. Let the query graph be fully connected (worst case). Let $\sigma_l$ denote the ratio of the number of total paths to the number of valid paths at depth $l+1$ and $\sigma_0$ be $|V_D|/number\_of\_initial\_candidates$. Let $P_l$ represent the set of partial paths at depth $l$ and let $|P_l|$ denote the number of partial paths at depth $l$. Let $P_l^i$ represent the $i^{th}$ path at level $l$.
Finding initial set of possible candidates requires scanning all data graph vertices; hence the complexity of this step is $O(|V_D|)$. $|P_1|$ can be estimated as $V_D \times \sigma_0$. To find all possible candidates at depth 2, we have to consider all the neighbours of depth 1 candidates; the complexity of this step is $O(|P_1| \times \delta)$. $|P_2|$ can be estimated as $|P_1| \times \delta \times \sigma_1$. A viable candidate at depth 3, for path

$P_l^i$ should be a neighbour of both $P_1^i$ and $P_2^i$, which requires intersecting neighbours of both $P_1^i$ and $P_2^i$. Initially we consider all neighbours of $P_2^i$ which can be estimated as $|P_2| \times \sigma_2$. Then we intersect this with the neighbours of $P_1^i$. The intersection of two sets of size $S_1$ and $S_2$ has complexity of $O(S_1 + S_2)$. Hence the total complexity at depth 3 is $O((|P_2| \times (\delta + \delta))$. $|P_3|$ can be estimated as $|P_2| \times \delta \times \sigma_2$. By generalization, complexity at depth $l$ can be computed as $O((|P_{l-1}| \times (l-1) \times \delta))$. $|P_l|$ can be estimated as $|P_{l-1}| \times \delta \times \sigma_{l-1}$. For simplicity, assume that the fraction of valid paths at all levels is a constant ($\sigma = max(\sigma_i)$). Under this assumption sequential complexity can be expressed as:

$$\begin{aligned} s\_complexity &= O(|V_D|) + O(|P_1| \times \delta) + O((|P_2| \times (\delta + \delta)) + \\ &\quad \ldots + O((|P_{l-1}| \times (l-1) \times \delta) \\ &= O(|V_D|) + O(|V_D| \times \sigma \times \delta) + \\ &\quad \sum_{l=3}^{|V_Q|} O(|V_D| \times \sigma^{l-1} \times \delta^{l-1} \times (l-1) \times \delta) \end{aligned} \tag{6}$$

Assuming $|V_Q|$ and ignoring lower order terms in Equation (6), the sequential time complexity can be estimated as $s_{complexity} = O(|V_D| \times \sigma^{|V_Q|-1} \times \delta^{|V_Q|-1} \times (|V_Q| - 1) \times \delta)$, which can be further simplified as $s_{complexity} = O(|V_D| \times |V_Q| \times \delta^{|V_Q|})$ since $\sigma <= 1$ and so $\sigma^{|V_Q|} << 1$ *or* $= 1$.

**Single GPU**: Assume that *nSMP* is the amount of parallelism available in a single GPU. For simplicity, assume that the work across GPU SMPs (streaming multiprocessors) is uniform. Note that this does not assume that the work across different thread blocks is uniform. Based on the state of the SM, the GPU can map multiple thread blocks to an SMP. Since the number of thread blocks launched is huge, it is reasonable to assume the scheduler will balance the workload. Under this assumption the single node complexity can be calculated as $p_{complexity} = s_{complexity}/nSMP = O(|V_D| \times \sigma^{|V_Q|-1} \times \delta^{|V_Q|-1} \times (|V_Q| - 1) \times \delta)/nSMP$.

**Multi GPU**: Let *nGPU* be the total number of (homogeneous) GPUs. The number of nodes processed by each node at level 1 is $|V_D|/nGPU$. Let $W_{min}$ and $W_{max}$ be the minimum and maximum workload across all GPUs. In the worst case, we can assume that all GPUs initially perform $W_{min}$ units of work in parallel, and then it has to process half of the remaining work of the processors which got $W_{max}$ units of work. So the total time complexity will be $O(W_{min} + (W_{max} - W_{min})/2)$. If we assume an all-to-all communication model, ignore the communication required for initialization and communication, and assume that the maximum space required per node is $S_{max}$ (see Equation (5)), the communication complexity will be $O(S_{max})$. If we assume that the load is perfectly balanced, we can compute multi-node complexity can be computed as $m_{complexity} = p_{complexity}/nGPUs = O(|V_D| \times \sigma^{|V_Q|-1} \times \delta^{|V_Q|-1} \times (|V_Q| - 1) \times \delta)/(nSMP * nGPU)$.

# 6 EXPERIMENTS

In this section, we compare the performance against the cutting edge subgraph isomorphism frameworks. In [17], Li Zeng et al. have shown that GSI consistently achieves better (or equal) performance than the state-of-the-art CPU and GPU implementations such as MAGiQ,[8], CFL-Match[1], CBWJ[10], Wukong+G[16], VF3[3], GPSM[13], GUNROCK[15]. Hence, we compare our approach against GSI[1].

## 6.1 Software and Hardware Configuration:

We evaluated our approach and GSI on two different machines: i) an Nvidia Volta V100 machine (84 SMs, 32GB global memory) paired with dual Intel Xeon E5-2680 v4 (128GB RAM) CPUs running Ubuntu 18.04.4 LTS, and ii) an Nvidia Ampere A100 machine(108 SMs, 40 GB global memory) paired with dual Intel Xeon Gold 6238R (384GB RAM) CPUs running Ubuntu 20.04 LTS and. We used CUDA 11.0 to compile codes on both devices. We evaluate the single node results on both the A100 machine and the V100 machine. We evaluated the distributed version by running our code on multiple V100 machines, each with a single V100 GPU.

## 6.2 Datasets and query graphs:

We evaluated our approach using six real-world data graph: enron(an email connection based graph), gowalla(location-based social), wikiTalk(Wikipedia communication network), roadNet-CA(road network of California), roadNet-PA(road network of Pennsylvania), and roadNet-TX(roadNet-TX). Table 2 shows the graph characteristics. These graphs were selected based on the prior works. We obtained these graphs from the SNAP [9] repository. Query graphs with lots of edges are the most difficult ones to solve efficiently. Hence we generated all possible five node graphs and then sorted them by the total number of edges in decreasing order and selected the top 11 as the query graphs. For graphs with the same number of edges, we broke the tie randomly. A similar procedure was carried out for six node and seven node query graphs.

## 6.3 Evaluation

**Evaluation Metric:** We use the execution time of GPU kernel(s) as the evaluation metric. The time to transfer the data graph was not accounted in both approaches. As explained in Section 4, in order to support bigger datasets and complex query graphs, our frameworks use both BFS-DFS traversals. We have included this overhead in all our measurements. In the distributed version, we have included all the CPU-GPU transfer times, inter-node communication time, and synchronization time. To account for the difference in performance between the two frameworks, we use the Nvidia Nsight Compute to collect different hardware metrics such as occupancy, data movement, and thread divergence.

**Single node results:** Table 3 shows our single node results. The GSI runtime (in milliseconds) and cuTS runtime (in milliseconds) are encoded in the following format "GSI ; cuTS". On the A100 machine, we successfully ran 164 cases, whereas GSI could only run 99 cases. Similarly, on the V100 machine, we were able to successfully run 154 cases, whereas GSI was only able to run 99 cases. A100 machine has a higher memory capacity than the V100 machine. Hence, we are able to run more cases on the A100 machine. The geomean speedup of cuTS over GSI for the roadNet-PA, roadNet-TX, and roadNet-CA was 329, 430, and 407, respectively, on the Nvidia A100 machine. The geomean speedup of cuTS over GSI for the

[1] https://github.com/pkumod/GSI – commit id c8d631236983

| | enron | | | | gowalla | | | | roadNet-PA | | | | roadNet-TX | | | | roadNet-CA | | | | wikiTalk | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A100 | | V100 | | A100 | | V100 | | A100 | | V100 | | A100 | | V100 | | A100 | | V100 | | A100 | | V100 | |
| | GSI | cuTS | GSI | cuTS | GSI | cuTS | GSI | cuTS | GSI | cuTS | GSI | cuTS | GSI | cuTS | GSI | cuTS | GSI | cuTS | GSI | cuTS | GSI | cuTS | GSI | cuTS |
| q1 | - | 8851 | - | 6607 | - | 33206 | - | 49332 | 687 | 7 | 431 | 4 | 930 | 7 | 491 | 4 | 1245 | 11 | 730 | 6 | - | 82588 | - | 114504 |
| q2 | - | 8837 | - | 6573 | - | 33086 | - | 54928 | 688 | 7 | 409 | 4 | 892 | 7 | 516 | 4 | 1255 | 11 | 740 | 6 | - | 86639 | - | 143173 |
| q3 | - | 9327 | - | 6970 | - | 34342 | - | 52048 | 684 | 7 | 419 | 4 | 923 | 7 | 529 | 4 | 1259 | 11 | 718 | 6 | - | 71255 | - | 108136 |
| q4 | - | 10573 | - | 8385 | - | 43982 | - | 55546 | 653 | 8 | 393 | 4 | 858 | 8 | 565 | 4 | 1186 | 13 | 683 | 6 | - | 93539 | - | 151134 |
| q5 | - | 9293 | - | 6919 | - | 34021 | - | 49820 | 657 | 7 | 389 | 4 | 861 | 7 | 495 | 4 | 1162 | 11 | 685 | 6 | - | 85291 | - | 138199 |
| q6 | - | 10564 | - | 8307 | - | 44083 | - | 57746 | 644 | 8 | 385 | 4 | 839 | 8 | 496 | 4 | 1182 | 13 | 693 | 6 | - | 96772 | - | 158326 |
| q7 | - | 9298 | - | 6926 | - | 34205 | - | 49473 | 654 | 7 | 405 | 4 | 856 | 7 | 503 | 4 | 1163 | 11 | 693 | 6 | - | 88887 | - | 150911 |
| q8 | - | 8151 | - | 6090 | - | 30070 | - | 48026 | 653 | 7 | 388 | 4 | 872 | 7 | 493 | 4 | 1188 | 11 | 689 | 6 | - | 103818 | - | 176300 |
| q9 | - | 8164 | - | 6112 | - | 30281 | - | 47956 | 654 | 8 | 372 | 4 | 880 | 7 | 487 | 4 | 1183 | 11 | 689 | 6 | - | 88341 | - | 153688 |
| q10 | - | 8268 | - | 6160 | - | 30614 | - | 42881 | 620 | 8 | 370 | 4 | 825 | 7 | 461 | 4 | 1081 | 11 | 651 | 6 | - | 90141 | - | 150503 |
| q11 | - | 8259 | - | 6180 | - | 30295 | - | 43017 | 617 | 9 | 364 | 4 | 834 | 8 | 458 | 4 | 1107 | 13 | 648 | 7 | - | 104175 | - | 168898 |
| q12 | - | 152560 | - | 139398 | - | 580505 | - | - | 653 | 1 | 407 | 1 | 874 | 1 | 489 | 1 | 1187 | 2 | 690 | 1 | - | 1155800 | - | 2063400 |
| q13 | - | 255168 | - | 325684 | - | 1082000 | - | - | 688 | 1 | 423 | 1 | 892 | 1 | 511 | 1 | 1242 | 2 | 728 | 1 | - | 1399930 | - | 2308120 |
| q14 | - | 167811 | - | 232490 | - | 624750 | - | - | 659 | 1 | 400 | 1 | 879 | 1 | 493 | 1 | 1181 | 2 | 680 | 1 | - | 1120940 | - | 1915960 |
| q15 | - | 162114 | - | 225663 | - | 600496 | - | - | 651 | 1 | 406 | 1 | 862 | 1 | 485 | 1 | 1143 | 2 | 688 | 1 | - | 1320980 | - | 2426190 |
| q16 | - | 150660 | - | 215953 | - | 558794 | - | - | 617 | 1 | 375 | 1 | 824 | 1 | 463 | 1 | 1115 | 2 | 645 | 1 | - | 1160250 | - | 2032980 |
| q17 | - | 149425 | - | 213845 | - | 553420 | - | - | 656 | 1 | 411 | 1 | 869 | 1 | 493 | 1 | 1195 | 2 | 689 | 1 | - | 1210250 | - | 2083800 |
| q18 | - | 149923 | - | 219232 | - | 554430 | - | - | 657 | 2 | 386 | 1 | 883 | 2 | 469 | 1 | 1150 | 2 | 684 | 1 | - | 1247810 | - | 1967790 |
| q19 | - | 165271 | - | 242986 | - | 621429 | - | - | 644 | 1 | 390 | 1 | 855 | 1 | 498 | 1 | 1155 | 2 | 685 | 1 | - | 1389050 | - | 2243470 |
| q20 | - | 153147 | - | 224247 | - | 545104 | - | - | 656 | 1 | 395 | 1 | 850 | 1 | 494 | 1 | 1167 | 2 | 687 | 1 | - | 1980750 | - | 3463900 |
| q21 | - | 152539 | - | 220596 | - | 554440 | - | - | 642 | 1 | 395 | 1 | 837 | 2 | 491 | 1 | 1178 | 2 | 693 | 1 | - | 1057240 | - | 1687480 |
| q22 | - | 182688 | - | 214690 | - | - | - | - | 692 | 1 | 416 | 1 | 883 | 1 | 513 | 1 | 1214 | 2 | 733 | 1 | - | 1609290 | - | 2502260 |
| q23 | - | - | - | - | - | - | - | - | 717 | 1 | 450 | 1 | 938 | 1 | 551 | 1 | 1304 | 1 | 765 | 1 | - | - | - | - |
| q24 | - | - | - | - | - | - | - | - | 715 | 1 | 443 | 1 | 921 | 1 | 577 | 1 | 1306 | 1 | 771 | 1 | - | - | - | - |
| q25 | - | - | - | - | - | - | - | - | 740 | 1 | 432 | 1 | 938 | 1 | 555 | 1 | 1282 | 1 | 921 | 1 | - | - | - | - |
| q26 | - | - | - | - | - | - | - | - | 605 | 1 | 371 | 1 | 815 | 1 | 451 | 1 | 1122 | 2 | 655 | 1 | - | - | - | - |
| q27 | - | - | - | - | - | - | - | - | 650 | 1 | 387 | 1 | 870 | 1 | 510 | 1 | 1164 | 1 | 696 | 1 | - | - | - | - |
| q28 | - | - | - | - | - | - | - | - | 692 | 1 | 406 | 1 | 891 | 1 | 520 | 1 | 1252 | 1 | 842 | 1 | - | - | - | - |
| q29 | - | - | - | - | - | - | - | - | 684 | 1 | 421 | 1 | 917 | 1 | 524 | 1 | 1255 | 1 | 725 | 1 | - | - | - | - |
| q30 | - | - | - | - | - | - | - | - | 652 | 1 | 398 | 1 | 837 | 1 | 491 | 1 | 1150 | 1 | 801 | 1 | - | - | - | - |
| q31 | - | - | - | - | - | - | - | - | 649 | 1 | 388 | 1 | 860 | 1 | 487 | 1 | 1174 | 1 | 698 | 1 | - | - | - | - |
| q32 | - | - | - | - | - | - | - | - | 570 | 1 | 344 | 1 | 780 | 1 | 439 | 1 | 1037 | 1 | 610 | 1 | - | - | - | - |
| q33 | - | - | - | - | - | - | - | - | 655 | 1 | 400 | 1 | 867 | 1 | 496 | 1 | 1166 | 1 | 695 | 1 | - | - | - | - |

**Table 3: Single node results (times in milliseconds). "-" indicates that the execution did not complete successfully.**

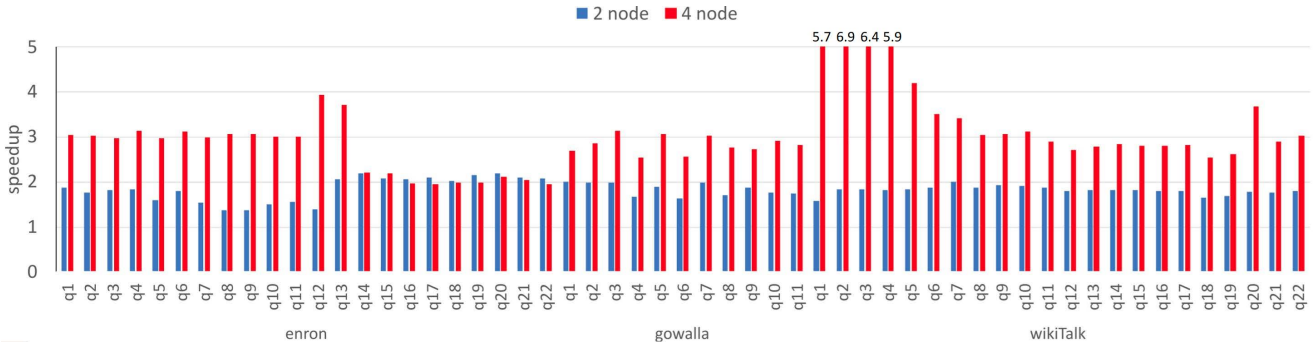



**Figure 4: Speedup against single node**

roadNet-PA, roadNet-TX, and roadNet-CA was 250, 314, and 387, respectively, on the Nvidia V100 machine.

Our high speedup can be attributed to (i) reduced number of operations, (ii) significantly lower data movement. The total number of operations in cuTS is much less than that of GSI. This is due to our superior query node ordering process. We select the query graph node with the highest degree ($q_{max}$) as the root node to match. Let there be $x$ candidates with this strategy. Choosing any other query graph node ($q_*$), whose degree is less than $q_{max}$ as the root node to match, will increase the search space. Any such node will at least have these $x$ candidates, as these nodes are guaranteed to have a higher degree than or equal to $q_*$ by construction and hence are viable candidates. In addition to these $x$ candidates, since the degree of $q_*$ is less than $q_{max}$, there will be additional candidates. Note that filtering efficiency, especially at lower depths, is crucial as reducing the total number of candidates at lower levels exponentially reduces the total number of higher levels and achieves exponential speedup. There are cases where cuTS has more than 785x fewer candidates than GSI at depth 1 and 26,000x lower candidates at depth 2. Moreover, when the query graph is not present in the data graph, our approach can eliminate all candidates at a lower depth than GSI. This is also evident from the hardware metrics as cuTS only executes significantly fewer instructions (the reduction factor is 1000x SASS (assembly) instructions).

Data movement is one of the critical factors that determine performance. When compared to GSI, cuTS has upto 200x lower DRAM read traffic. Subgraph isomorphism is a memory-bound problem. Hence a 200x reduction in data movement will drastically improve the performance. The total number of write operations is also much lower for cuTS. In addition to these factors, compared to GSI, we have 34x lower shared-memory (programmable cache) writes and 7x lower reads. We also have 2x lower atomic operations, and we execute 7x lower instructions. The reduction in instructions is a direct result of query node matching order prioritization.

**Distributed results:** One of the most significant advantages of using our distributed version is the ability to run larger datasets and complex query graphs. While our single-node implementations only ran 154 cases on the V100 machine, our distributed implementation was able to run 158 cases using two nodes and 164 cases using four nodes. Figure 4 shows the scalability of our approach on the three biggest datasets. Since we are showing speed up, we are only showing cases where the single node implementation was completed successfully. The multi-node results are only collected for big datasets such as enron, gowalla, and wikiTalk.
We achieve close to 2x speedup over two nodes for the big graphs and close to 3.1x on four nodes for the big data graphs. There are cases where we achieve superlinear speedup, which can be attributed to better cache hits.

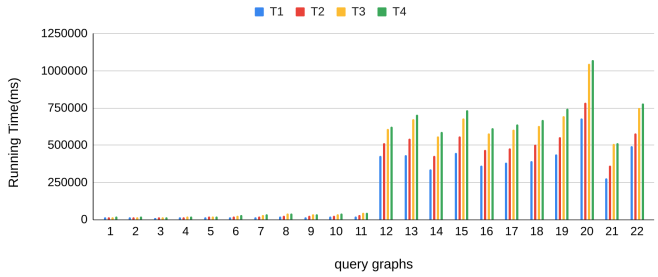



**Figure 5: Load balancing on 4 node V100 system with wikiTalk dataset. T1 is the running time for the first node, T2 is the running time for the second node and so on.**

Figure 5 shows the efficacy of our load balancing scheme on the wikiTalk dataset. Our node to node runtime variation is very low. Others follow the same pattern. Thanks to our work distribution strategy all nodes gets similar amount of work and does not suffer from synchronization overheads.

## 7 CONCLUSION

This paper develops a novel trie-based data structure that achieves good compression without sacrificing performance. Our new data structure is well suited for massive parallelism in GPUs and avoids the need for two-pass algorithms. We developed the first (to the best of our knowledge) distributed algorithm for subgraph isomorphism on GPUs. Our experimental evaluation sections show the efficacy and scalability of the proposed approach. We demonstrated that cuTS is able to handle bigger data graphs and complex query graphs. Our single-node implementations significantly outperform the state-of-the-art competitors, and our multi-node implementation achieves close to linear speedup on big data graphs.

## ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through awards 1816793 and 1919211 at the Washington State University, and the U.S. Department of Energy (DOE) through the Exascale Computing Project (17-SC-20-SC) (ExaGraph) at the Pacific Northwest National Laboratory. We also thank the IT department at Washington State University, especially Vollmer, Simon for help with setting up the machines used for experiments.

## REFERENCES

[1] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.
[2] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14, 7 (2013), 1–13.
[3] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. 2017. Introducing VF3: A new algorithm for subgraph isomorphism. In *International Workshop on Graph-Based Representations in Pattern Recognition*. Springer, 128–139.
[4] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.
[5] Wenfei Fan. 2012. Graph Pattern Matching Revised for Social Network Analysis. (2012).
[6] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (1978), 250–269. https://doi.org/10.1145/355791.355796
[7] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 405–418.
[8] Fuad Jamour, Ibrahim Abdelaziz, Yuanzhao Chen, and Panos Kalnis. 2019. Matrix algebra framework for portable, scalable and efficient query engines for rdf graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–15.
[9] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[10] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *arXiv preprint arXiv:1903.02076* (2019).
[11] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. 2002. Network Motifs: Simple Building Blocks of Complex Networks. *Science* (2002). https://doi.org/10.1126/science.298.5594.824
[12] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (2008), 364–375.
[13] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*. Springer, 299–315.
[14] JR ULLMANN. 1976. An Algorithm for Subgraph Isomorphism. (1976).
[15] Leyuan Wang and John D Owens. 2020. Fast Gunrock Subgraph Matching (GSM) on GPUs. *arXiv preprint arXiv:2003.01527* (2020).
[16] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. 2018. Fast and concurrent {RDF} queries using RDMA-assisted {GPU} graph exploration. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 651–664.
[17] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang. 2020. GSI: GPU-friendly Subgraph Isomorphism. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1249–1260. https://doi.org/10.1109/ICDE48307.2020.00112
[18] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. 192–203.
[19] Peixiang Zhao and Jiawei Han. 2010. On graph query optimization in large networks. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 340–351.

# Appendix: Artifact Description/Artifact Evaluation

## SUMMARY OF THE EXPERIMENTS REPORTED

We ran our experiments on Nvidia A100 and Nvidia V100. We used NVCC 11.0 with the "-O3" flag to compile our code. We ran our distributed code using OpenMPI 3.0.1. The results presented for cuts in the paper can be reproduced by running the automated script 'cuts.py', '2nodes_exe.sh' and '4nodes_exe.sh' which will generate the experimental data corresponding to our code. For GSI, we have provided the github link in our 'README' file. We have also provided a sample python script (convert_ours_to_gsi.py) to convert our graph format to theirs. For other detailed information, refer to the 'README' file inside the repo.

*Author-Created or Modified Artifacts:*

```
Persistent ID: https://doi.org/10.5281/zenodo.5154114
Artifact name: CuTS
```

## BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

*Relevant hardware details:* Nvidia A100, Nvidia V100

*Operating systems and versions:* 18.04.5 LTS

*Compilers and versions:* nvcc/11.0

*Libraries and versions:* OpenMpi 3.0.1

*Key algorithms:* subgraph isomorphism