Efficient GPU-Accelerated Subgraph Matching

XIBO SUN, The Hong Kong University of Science and Technology, Hong Kong SAR
QIONG LUO, The Hong Kong University of Science and Technology, Hong Kong SAR and The Hong Kong University of Science and Technology (Guangzhou), China

Subgraph matching is a basic operation in graph analytics, finding all occurrences of a query graph Q in a data graph G. A common approach is to first filter out non-candidate vertices in G, and then order the vertices in Q to enumerate results. Recent work has started to utilize the GPU to accelerate subgraph matching. However, the effectiveness of current GPU-based filtering and ordering methods is limited, and the result enumeration often runs out of memory quickly. To address these problems, we propose EGSM, an efficient approach to GPU-based subgraph matching. Specifically, we design a data structure Cuckoo trie to support dynamic maintenance of candidates for filtering, and order query vertices based on estimated numbers of candidate vertices on the fly. Furthermore, we perform a hybrid breadth-first and depth-first search with memory management for result enumeration. Consequently, EGSM significantly outperforms the state-of-the-art GPU-accelerated algorithms, including GSI and CuTS.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; • **Theory of computation → Parallel algorithms**.

Additional Key Words and Phrases: graph, subgraph matching, GPU

ACM Reference Format:
Xibo Sun and Qiong Luo. 2023. Efficient GPU-Accelerated Subgraph Matching. *Proc. ACM Manag. Data* 1, 2, Article 181 (June 2023), 26 pages. https://doi.org/10.1145/3589326

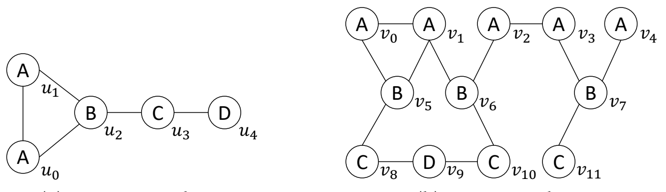
1 INTRODUCTION

Graphs, storing entities and relationships, are common in real-world applications, such as social media, financial transactions, computer networks, and knowledge representation. Subgraph matching is a basic operation that finds all occurrences of a small query graph Q in a large data graph G. For example, in Figure 1, $\{(u_0, v_0), (u_1, v_1), (u_2, v_5), (u_3, v_8), (u_4, v_9)\}$ and $\{(u_0, v_1), (u_1, v_0), (u_2, v_5), (u_3, v_8), (u_4, v_9)\}$ are both matches. This problem finds applications in identifying social positions [11], detecting credit-card fraud [40], finding related products [27], querying a knowledge graph [38], and so on. It is a well-known NP-hard problem [12], and a number of algorithms have been proposed, including both CPU-based [5, 6, 8, 16, 17, 20, 42, 43, 46, 57, 58] and recent GPU-based methods [26, 30, 30, 31, 50, 54, 55]. In this paper, we propose an efficient GPU-based algorithm, called EGSM, for subgraph matching.

Existing subgraph matching algorithms either perform joins on edge tuples or explore the data graph to match vertices. Both kinds of algorithms follow a *filtering-and-enumeration* paradigm. Specifically, latest algorithms on CPUs, such as RapidMatch [47] and VEQ [20], design filtering

Authors' addresses: Xibo Sun, xsunax@connect.ust.hk, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong SAR; Qiong Luo, luo@cse.ust.hk, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong SAR and The Hong Kong University of Science and Technology (Guangzhou), No.1 Du Xue Rd, Nansha District, Guangzhou, China.

2836-6573/2023/6-ART181 $15.00
https://doi.org/10.1145/3589326



(a) Query graph Q. (b) Data graph G.

Fig. 1. Example graphs.

rules based on the query graph structure to prune unpromising vertices (resp., edges) in the data graph, i.e., vertices (resp., edges) that will not be in the final result. Then, they develop various data structures to store the remaining vertices (resp., edges) as candidates. In the enumeration phase, the algorithms determine a matching order (i.e., a permutation of vertices in Q) based on the graphs and the statistics of candidates. The order may be either statically generated, the same for all results, or dynamically determined, adaptive to current candidates. These CPU-based algorithms employ the depth-first-search to find matches by extending vertices (resp. edges) along the matching order. Since only one result is processed at a time in DFS, the memory consumption is linear to the query graph size. Nevertheless, matching on large data graphs remains time-consuming.

To speed up subgraph matching, recent work has employed a variety of modern hardware[7, 18, 59, 60], including the GPU, a commodity parallel processor with thousands of cores and high-bandwidth memory. Latest GPU-based solutions of subgraph matching, such as GSI [55] and CuTS [54], adopt the same filtering-and-enumeration procedure, but they parallelize the execution. These GPU-based algorithms achieve significant speedups over CPU-based competitors; however, they often run out of GPU memory due to ineffective filtering and ordering, and their breadth-first-search enumeration.

Table 1. A comparison of representative subgraph matching algorithms.

Hardware Platform	Algorithm	Category	Filtering			Ordering		Enumeration		
			Factor	Index	Time Complexity	Method	Cost Model	Model	Target	Memory Cost
CPU	RapidMatch [47]	Join	Structure of Q	Encoded Tries	$O(\lvert E(Q)\rvert\lvert E(G)\rvert)$	Static	Density of Q	DFS	Index	$O(\lvert V(Q)\rvert)$
	VEQ [20]	Exploration	Structure of Q	Candidate Space	$O(\lvert E(Q)\rvert\lvert E(G)\rvert)$	Adaptive	Estimated Cardinality	DFS	Index	$O(\lvert V(Q)\rvert)$
GPU	GSI [55]	Exploration	Vertex Neighbors	Candidate Sets	$O(\lvert E(Q)\rvert\lvert E(G)\rvert)$	Static	Estimated Cardinality	BFS	G	$O(\lvert V(G)\rvert^{\lvert V(Q)\rvert})$
	CuTS [54]	Exploration	N/A	N/A	N/A	Static	Vertex Degrees in Q	BFS	G	$O(\lvert V(G)\rvert^{\lvert V(Q)\rvert})$
	ALFTJ [26]	Join	N/A	N/A	N/A	N/A	N/A	DFS	G	$O(\lvert E(Q)\rvert)$
	EGSM (This work)	Exploration	Structure of Q	Cuckoo Tries	$O(\lvert E(Q)\rvert\lvert E(G)\rvert \log\lvert V(G)\rvert)$	Adaptive	Estimated Cardinality	BFS-DFS	Index	$O(\lvert V(G)\rvert^{\lvert V(Q)\rvert})$*

* The memory cost of EGSM is always less than the space constraint.

Table 1 compares a few representative existing algorithms with our proposed algorithm EGSM. On the CPU, we include RapidMatch [47] and VEQ [20]; and on the GPU, we list GSI [55], CuTS [54], and ALFTJ [26], a multi-way join implementation that can be used for subgraph matching. As shown in the table, among the three GPU-based methods, only GSI employs a filtering rule, and the rule only considers the neighbors of a vertex. Because it is difficult to maintain dynamic data structures on the GPU, some steps of the GSI's filtering process execute on a CPU, which under-utilizes the GPU and introduces data transfer overhead. Also, even though GSI maintains candidate sets, the enumeration runs on the data graph, similar to that of CuTS and ALFTJ, leading to unnecessary visits of unpromising vertices or edges.

Both GSI and CuTS adopt BFS in their enumeration phase because BFS utilizes GPU parallelism better than DFS and can significantly reduce the query time. However, the space consumption of the BFS approach is exponential to the data graph size due to the storage of partial matches. Therefore, the method may easily fill up the memory space, even on graphs with a moderate size. When the algorithms run out of memory for further matching, the query will be unsolved.

Also, the BFS-based approach incurs a lot of memory accesses in the enumeration, which may be especially bad for the GPU memory, as the GPU memory has a much higher latency than the CPU memory. Specifically, in each round of extension, the algorithm loads partial results from memory, extends them, and then writes the extended partial results to the memory. Furthermore, with ineffective filtering and ordering, many partial results, not leading to full matches, are also written to and read back from the memory. Unfortunately, current GPU-based algorithms either have no matching order optimization or choose a fixed matching order for the entire data graph based on heuristics. Such static matching order may not be suitable in the presence of data skew. In comparison, ALFTJ utilizes DFS in the enumeration; however, it only supports query graphs of three to four vertices due to its usage of the small shared memory on the GPU.

In this paper, we seek to improve both the time and memory efficiency of GPU-based subgraph matching and propose EGSM (Efficient approach to GPU-accelerated Subgraph Matching). First, as both the ineffectiveness and inefficiency of the filtering method in previous studies are due to the lack of data structure support, we propose a new data structure Cuckoo trie based on Cuckoo hashing. For each edge e in Q, we use a set of Cuckoo tries to store all the candidate edges, i.e., the edges in G that can match e. Our Cuckoo tries support parallel edge insertion and deletion, which are the key operations in the filtering step. They also support random access to the neighbors of a vertex, which is frequently performed in the enumeration. Our EGSM enumerates results from the index rather than G, thus improving the efficiency. All these operations have theoretical guarantees on the time complexity and are practical on GPUs. Based on the Cuckoo tries, we design a three-step parallel pruning routine on GPUs, considering the structure of the entire Q, to reduce the number of candidates as much as possible. This pruning is effective and reduces the search space for subsequent enumeration.

To address the drawbacks of previous GPU-based enumeration, we propose a hybrid BFS-DFS matching strategy, which takes advantage of both the BFS- and DFS-based methods. Specifically, we divide all vertices in Q into several groups, and extend a group of vertices at a time (BFS). The grouping is determined by the query graph structures heuristically. Within each group, we perform the matching in the DFS order. When a group finishes, we write all new partial results to the global memory and then match the next group. Furthermore, we track the memory usage during the extension. If memory becomes insufficient, we dynamically put all remaining query vertices in one group and fall back to DFS to complete. As such, even though BFS has a memory cost of $O(|V(G)|^{|V(Q)|})$ to store partial results, the actual memory cost of EGSM is always less than the global memory size due to the DFS fall-back. Also, we design an adaptive ordering method inside a group based on the local properties of the data graph and implement the approach on GPUs with optimizations on the memory access. Finally, we perform dynamic load balancing to split heavy tasks. In summary, we make the following contributions in this paper.

- We propose a data structure Cuckoo trie for storing candidate edges and introduce the operations on the data structure with running time guarantee.
- We design a three-step parallel pruning routine on GPUs to filter out candidates with efficiency and effectiveness.
- We develop a BFS-DFS matching strategy with a memory management framework to keep memory cost under constraint. We design an adaptive ordering method for extending a partial match and dynamic load balancing methods to further improve the efficiency.
- We conduct extensive experiments on various datasets with different query graphs. Results show that EGSM outperforms previous approaches significantly.

2 PRELIMINARIES AND RELATED WORK

2.1 Definitions

In this paper, we study subgraph matching on undirected vertex-labeled graphs $g = (V, E)$, where V is the set of vertices and E is the set of edges. We use $V(g)$ and $E(g)$ to denote the vertex set and edge set of g. L is a function mapping from a vertex $v \in V(g)$ to a label l in the label set $\Sigma(g)$. $e(v, v')$ denotes the edge between vertices v and v'. Given $V' \subseteq V(G)$, $G[V']$ is an induced graph of G on V', where $V(G[V']) = V'$ and $E(G[V']) = \{e(u, u') \in E(G) | u \in V' \text{ and } u' \in V'\}$. We use Q and G to denote the query and data graphs, respectively. We call the vertices and edges in Q (resp., G) *query vertices* and *query edges* (resp., *data vertices* and *data edges*). The label sets of Q and G are identical, i.e., $\Sigma(Q) = \Sigma(G)$. Then, we define subgraph isomorphism as follows.

Definition 2.1 (Subgraph Isomorphism). Given graphs g and g', a subgraph isomorphism is an injective function $M : V(g) \to V(g')$ such that 1) $\forall v \in V(g), L(v) = L(M(v))$; 2) $\forall e(v, v') \in E(g), e(M(v), M(v')) \in E(g')$.

Subgraph matching is to find the set $\mathcal{M}$ of all subgraphs in G isomorphic to Q. We use M to record the mapping from a query vertex u to a data vertex v, i.e., $M(u) = v$. Given a vertex $v \in V(g)$, we denote $N(v)$ as the set of v's neighbors and $d(v) = |N(v)|$ as the degree of v. We also define the *neighbor-label-count* (*NLC*) array of v as $NLC(v)$. $NLC(v)$ contains $|\Sigma(Q)|$ values, with each value recording the number of vertices $v' \in N(v)$ such that $L(v') = i$ (we map all vertex labels in $\Sigma(Q)$ to consecutive integers starting from 0). We say $NLC(v) \succcurlyeq NLC(v')$ if for any position i in $NLC(v)$, the value is greater than or equal to the value at the corresponding position in $NLC(v')$. We define a wedge as a graph containing two connected edges. In particular, wedge u'-u-u'' contains $e(u', u)$ and $e(u, u'')$. We call u the center of the wedge u'-u-u''.

Previous methods keep a candidate vertex set $C(u)$ for each query vertex u, which contains all data vertices v that can map to u. Alternatively, we maintain a relation $R(u, u')$ to store all the candidate edges of a query edge $e(u, u')$, which are more suitable than vertices for our proposed trie structure. We denote a tuple in $R(u, u')$ as $t(v, v')$. and $|R(u, u')|$ as the number of tuples in $R(u, u')$. A tuple $t(v, v')$ belongs to $R(u, u')$ if and only if $e(v, v') \in E(G)$ is in the candidate set of $e(u, u')$ such that v maps to u and v' maps to u'. We define the *neighbor set* of v in $R(u, u')$, $N(v, u, u')$, to be $\{v' | t(v, v') \in R(u, u')\}$. A *matching order* φ is a permutation of $V(Q)$. We denote the neighbors of a query vertex u that precede u in φ as $N_+(u)$, and those after u as $N_-(u)$. For an arbitrary array a, we denote $a[i]$ as the element at position i in a, $a[i : j)$ as the sub-array of a from the position i (included) to j (not included), and $|a|$ as the length of a. In particular, $\varphi[i]$ denotes the vertex at position i in φ.

2.2 GPU Architecture

In recent decades, graphic processing units, or GPUs, have been widely used for general-purpose computation tasks that can be accelerated by massive parallelism. A GPU device contains thousands of cores organized in a hierarchy. Specifically, a GPU has tens of stream multiprocessors (SMs), and each SM contains tens to hundreds of cores. Nvidia provides the Compute Unified Device Architecture model (CUDA) for programming on GPUs [36]. A GPU program is called a kernel function, which is executed by a CUDA thread grid. A grid contains a certain number of thread blocks, and each block contains multiple threads. A thread block runs within the same SM, and each thread runs on a core. The basic scheduling unit on a GPU is a warp, i.e., a set of 32 threads in a thread block executing the same instruction. If different threads in a warp follow different control flow paths, these flows are serialized with only one control flow executed at a time. Such a

```
Algorithm 1: SubgraphMatchingBFS on the GPU
  Input: a query graph $Q$, a data graph $G$
  Output: subgraph matches $\mathcal{M}$
1 generate candidate sets $C$s, determine a matching order $\varphi$;
2 $\mathcal{M} \leftarrow C(\varphi[0])$;
3 for $i$ from 1 to $V(Q)$ do
4     $u \leftarrow \varphi[i]$, $\mathcal{M}_{old} \leftarrow \mathcal{M}$, $\mathcal{M} \leftarrow \{\}$;
5     ExtendBFS($u$, $\mathcal{M}_{old}$, $\mathcal{M}$);
6 Kernel ExtendBFS($u$, $\mathcal{M}_{old}$, $\mathcal{M}$)
7     each warp gets an $M_{old}$ from $\mathcal{M}_{old}$;
8     $\mathbb{S} \leftarrow \{N(M_{old}(u')) | u' \in N_{+}(u)\} \cup \{C(u)\}$;
9     $s_{min} \leftarrow s \in \mathbb{S}$ with the smallest cardinality;
10    while $s_{min} \neq \phi$ do
11        each lane picks a vertex $v$ from $s_{min}$;
12        if $v \in s$ for all $s \in \mathbb{S}$ then
13            $M \leftarrow$ add $(u, v)$ to $M_{old}$;
14            write $M$ to $\mathcal{M}$;
```

situation is known as *warp divergence*, which degrades the performance. A thread in a warp is also called a *lane*, with lane ID from 0 to 31.

The GPU memory hierarchy contains global memory, transparent caches, and programmable shared memory. The global memory has a capability of several to tens of gigabytes, accessible by all threads in the grid. The global memory is organized into fixed-sized aligned segments, and each memory access is serviced by memory transactions on segments. If multiple lanes in a warp access data in the same segment, only one memory transaction is issued. The segment size is 128 bytes if the data are cached or 32 bytes otherwise. However, the access latency of the global memory is high. A GPU also has a constant memory space of several kilobytes. It has a similar latency to the global memory but a dedicated cache space. Therefore, access to the constant memory is fast if data is cached, or the total data size is small. The accesses from a warp to different locations in the constant memory are serialized. Each SM has a shared memory. The memory access time is much shorter than the global and constant memory, but it is only accessible by the threads within a thread block. Finally, each thread has its registers, only available to the thread itself. CUDA provides a rich set of primitives for threads in a warp to communicate their register values.

2.3 BFS-based enumeration on the GPU

Latest algorithms employ the BFS-based enumeration on GPUs, as illustrated in Algorithm 1. Line 1 performs the filtering of candidates and the ordering of query vertices. Line 2 sets the candidate set of $\varphi[0]$ as the partial results $\mathcal{M}$. After that, Lines 3-5 loop over each remaining vertex in φ to extend all partial matches one vertex at a time until all query vertices are processed. Specifically, Line 4 gets the next vertex u to match and initializes the new match set $\mathcal{M}$ to be empty. Then, u, $\mathcal{M}_{old}$, and $\mathcal{M}$ are passed to the kernel function ExtendBFS. In the function, each warp gets a partial match M_{old} from $\mathcal{M}_{old}$ on Line 7, and generates $\mathbb{S}$ by including the neighbor set of each vertex $M_{old}(u')$ where $u' \in N_{+}(u)$, and the candidate set $C(u)$. On Line 9, the warp finds the smallest set s_{min}. Then, each lane in the warp picks a vertex v from s_{min} and checks whether v appears in all of the remaining sets in $\mathbb{S}$ (Lines 10-11). If so, the lane gets a new M by adding the mapping (u, v) to M_{old}, and then writes M to $\mathcal{M}$ (Lines 12-14). The warp repeats this procedure until all elements in s_{min} are processed.

2.4 Related Work

Subgraph matching. Ullmann et al. [51] first proposed a graph exploration-based backtracking approach to subgraph matching. Subsequent methods utilized pruning strategies, designed query plans, and proposed auxiliary data structures to improve the matching performance [5, 6, 8, 16, 17, 20, 42, 43, 46, 57, 58]. Some methods further performed subgraph matching on dynamic graphs [19, 22, 33, 48, 49]. On multi-core CPUs, parallel subgraph matching follows the DFS-based enumeration [4, 23, 41]. In contrast, on GPUs, Lin et al. [31] first proposed to let each thread perform matching independently, leading to lots of warp divergence. Consequently, NEMO [30], GPSM [50], and GunRockSM [52] were proposed to find matches in the breadth-first search, which better utilizes the GPU's parallel execution. After that, GSI [55] developed an efficient data structure to represent the data graph and a preallocate-combine strategy to reduce unnecessary computation in the enumeration. CuTS [54] improved GSI by compressing partial results and adopting efficient intersection and load-balancing methods. Nevertheless, the BFS-based enumeration on the GPU may run out of memory quickly, and there is large room to improve the filtering and ordering effectiveness.

Subgraph Enumeration. Both the query and data graphs in subgraph matching are labeled. In comparison, subgraph matching on unlabeled graphs is called subgraph enumeration, and can be treated as a special case with all vertices having the same label. CPU-based solutions include PSgL [44], DUALSIM [21], and LIGHT [45]. TwinTwig [24], SEED [25], CRYSTAL [39], and BiGJoin [3] work in distributed environments. On GPUs, Guo et al. developed RPS [15] to reduce computation by reusing intermediate results, and PBE [14] was proposed to extend RPS to partitioned graphs and heterogeneous computing. Finally, Dodeja et al. designed PARSEC [9] with depth-first traversal and various optimizations to efficiently utilize the memory and compute resources of the GPUs. We do not compare our work with subgraph enumeration methods since their research focuses differ from ours.

Multi-way Join Algorithms. An alternative method of subgraph matching is the multi-way join. Traditionally, the multi-way join was performed by a sequence of pairwise joins [13, 34, 35]. Then, the worst-case optimal join algorithm was proposed with time complexity guarantee, and benefits many database systems [1, 32, 47]. LFTJ-GPU [53] was developed as a GPU-optimized join based on the breadth-first execution method. In comparison, Lai et al. [26] improved both hash join and LFTJ on GPUs with the DFS-based enumeration and various optimizations. However, these GPU-based multi-way join algorithms typically handle a small number of attributes (at most 4) due to GPU memory constraints, whereas the query graph often has more than 10 query vertices.

3 DATA STRUCTURE

We propose a new data structure on the GPU to store candidate edges of each query edge to support both the filtering and enumeration phases. This section introduces the background, describes the data structure, and illustrates the operations, including insertion, deletion, and search.

3.1 Background

The compressed sparse row (CSR) format is widely utilized to store data graphs and the index in CPU-based methods. Because data are consecutively stored in a CSR, each insertion and deletion requires rebuilding the entire structure, which is impractical in the massively parallel environment. Due to the difficulties of dynamically maintaining candidates, latest GPU-based algorithms only keep a candidate vertex set for each query vertex. However, such a simple design limits the adoption of advanced filtering rules and reduces the enumeration efficiency. For example, in the loop starting

u_0	u_1
v_0	v_1
v_1	v_0
v_2	v_3
v_3	v_2

u_0	u_2
v_0	v_5
v_1	v_5
v_1	v_6
v_2	v_6
v_3	v_7

u_1	u_2
v_0	v_5
v_1	v_5
v_1	v_6
v_2	v_6
v_3	v_7

u_2	u_3
v_5	v_8
v_6	v_{10}

u_3	u_4
v_8	v_9
v_{10}	v_9

Fig. 2. Example relations.

from Line 10 of Algorithm 1, if many vertices in $N(M(u'))$ are not in $C(u)$, much computation is wasted.

To improve filtering effectiveness, we develop a new data structure to store candidate edges rather than vertices. We adopt Cuckoo hashing [37] as a building block due to its theoretical guarantee on operation efficiency and its practical adaption to GPUs in previous studies [2, 10, 29, 56]. Cuckoo hashing maintains a group of m independent hash functions, each corresponding to a separate hash table. When inserting a key k, Cuckoo hashing computes the insertion position in each hash table based on the hash functions. Then, it inserts k to a hash table in which the position is not occupied. If all hash tables are occupied at the position, Cuckoo hashing will choose one of the hash tables to kick out the original value at that position as a victim, insert k at the position, and then insert the victim back into the hash tables with the same procedure. This process goes on until no victim is produced or a maximum number of iterations is reached. If it is the latter case, the hash functions will be re-chosen, and the hash tables rebuilt. Cuckoo hashing guarantees a constant time complexity for the search or deletion of a key in the worst case and an amortized constant time complexity for the insertion.

3.2 Cuckoo Tries

In this section, we describe our Cuckoo trie structure. For each query edge $e(u, u') \in E(Q)$, we maintain a relation $R(u, u')$. We utilize a set $CT(u, u')$ of Cuckoo tries to store $R(u, u')$, where all tuples $t(v, v') \in R(u, u')$ can be retrieved given v. Given our example graphs in Figure 1, some example relations are shown in Figure 2. Since in the enumeration, tuples in a relation may be searched by the value on either attribute, we generate both $CT(u, u')$ and $CT(u', u)$ for a relation $R(u, u')$. $CT(u, u')$ contains three levels – the hash tables, the offsets, and the neighbor sets. During the execution of subgraph matching, multiple threads in a warp search for a single vertex on the top level but different neighbors on the bottom level. We make the top level Cuckoo hash tables so that a vertex can be retrieved in $O(1)$ time. By contrast, we design the bottom level as sorted arrays where multiple threads in a warp can follow the same execution path of the binary search. The Cuckoo hash tables contain all unique values of attribute u in $R(u, u')$. We use bucketized Cuckoo hash tables in which multiple keys can be hashed into the same bucket, and all buckets are stored consecutively in an array. Therefore, threads in a warp access the elements consecutively in a bucket to search for the vertex, which takes advantage of the high memory bandwidth and massive parallelism of GPUs, and furthermore, involves no irregular access or divergence. We denote the ith hash table as $Cuc[i]$. Each $Cuc[i]$ corresponds to an $Off[i]$ on the second level, recording the positions of $N(v, u, u')$ for each $v \in Cuc[i]$. The size of $Off[i]$ is two times that of $Cuc[i]$, and the neighbors array of vertices $Cuc[i][j]$ resides on $Nbr[i][Off[2i] : Off[2i] + Off[2i+1])$. Each $Cuc[i]$ also corresponds to an array on the third level, denotes as $Nbr[i]$. Given any $v \in Cuc[i]$, we can find $N(v, u, u')$ sorted and consecutively stored in $Nbr[i]$.

Example 3.1. The third relation in Figure 2 shows $R(u_0, u_2)$, built upon our example query and data graphs (Figure 1). Figure 3a shows $CT(u_0, u_2)$, corresponding to $R(u_0, u_2)$, where a bucket contains 2 cells. On the first level, a bold rectangle represents a bucket. In $Cuc[0]$, suppose v_1 and v_2

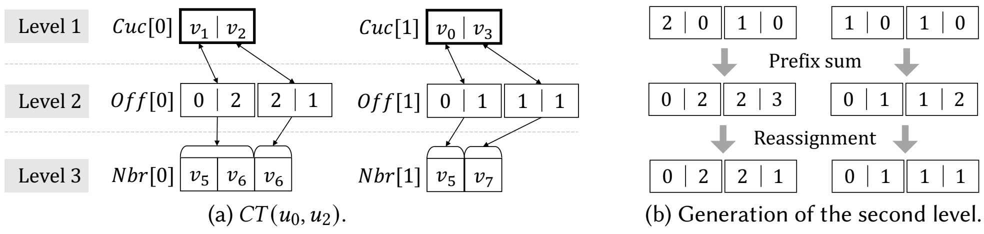


Fig. 3. Example Cuckoo tries.

are hashed into the first and second cell of the first bucket, respectively. Then, as shown in $Off[0]$, the neighbor array of v_1 resides at $Nbr[0][0 : 2)$, whereas that of v_2 resides at $Nbr[0][2 : 3)$. Therefore, the first Cuckoo trie represents three tuples, namely $t(v_1, v_5)$, $t(v_1, v_6)$, and $t(v_2, v_6)$. Similarly, the second Cuckoo trie represents two tuples $t(v_0, v_5)$ and $t(v_3, v_7)$. As a result, $CT(u_0, u_2)$ represents the same set of tuples as $R(u_0, u_2)$.

Parameters. Subgraph matching is a memory-intensive task [54], and most memory accesses are cache misses. For a GPU global memory access on a cache miss, the size of a memory transaction is 256 bits. A vertex ID is stored as a 32-bit unsigned integer, so we set the bucket size to 8 so that we can retrieve a bucket in one memory transaction. Another parameter is the number of hash tables m. A greater m leads to more memory accesses to locate a vertex in the hash tables. Specifically, for retrieving a vertex v, we check each hash table. If all hash tables have the same probability of containing v, the average number of memory accesses is $\frac{1+m}{2}$. Therefore, to minimize the number of memory accesses, we set m to 2, the minimum number of Cuckoo hash tables.

3.3 Operations

Cuckoo tries assist in both the filtering and enumeration phases. In the filtering phase, we first batch insert all data edges into a relation as the candidates. Then, some candidate edges are searched and filtered out as we perform pruning rules. In the enumeration, we search the neighbors of a particular vertex in the Cuckoo tries. In the following, we illustrate the operations on the set of Cuckoo tries $CT(u, u')$.

Batch Insertion. Given $R(u, u')$, we build $CT(u, u')$ level by level with batch processing. For $Cuc[0]$ and $Cuc[1]$, we group 8 consecutive threads on the GPU into a cooperative group, matching the size of a bucket. Each group of threads first gets a distinct value v with attribute u in $R(u, u')$, and then finds the bucket that v belongs to in $Cuc[0]$ by the hash function. After that, each thread in the group accesses one bucket cell, which can be achieved by one memory transaction. If there is an empty cell, the group will try to assign v to the empty cell atomically. If the insertion fails due to the write conflict between different groups, the current group will recheck the availability of the cells, get another empty cell, and assign the vertex again. If there is no empty cell, the group will choose a victim vertex from the bucket, replace the victim with v, and then insert the victim into another hash table. This Cuckoo hashing process goes on until the insertion succeeds or hash tables are rebuilt.

On the second level of our Cuckoo tries, all elements in $Off[0]$ are initialized to 0. Each warp on the GPU gets a non-empty cell in $Cuc[0]$. Suppose that the cell is located at position i and contains a vertex v. The warp will compute the size of the neighbor set of v in $R(u, u')$, $N(v, u, u')$, and store the size in $Off[0][2i]$. After that, we perform the exclusive prefix sum operator on the entire $Off[0]$ and then reassign $off[2i + 1]$ to $Off[0][2i + 1] - Off[0][2i]$ for each natural number $i < |Cuc[0]|$. $Off[1]$ is generated similarly. Figure 3b illustrates the computation of Offs for our example relation, and the resulting Offs is in Figure 3a.

Finally, each warp chooses a v at $Cuc[0][i]$ and gets the corresponding $N(v,u,u')$. Then, the warp writes $N(v,u,u')$ to $Nbr[0]$ starting from the position $Off[0][2i]$. Each warp has its own write areas in the array exclusive to others, so all warps can write in parallel without conflict. $Nbr[1]$ is generated the same way.

Search. Each GPU thread searches a vertex in the hash tables or a tuple in the Cuckoo tries. To search a vertex v, the thread applies the hash function of the first hash table to the ID of v, gets the target bucket, and checks whether any of the 8 cells records the ID. If so, the vertex is found. Otherwise, the thread checks the second hash table. To search a tuple $t(v,v')$, the thread locates v at the first level of the Cuckoo tries, gets the neighbor array of v, i.e., $N(v,u,u')$, at the third level, based on $Off[0]$ or $Off[1]$, and then performs the binary search in $N(v,u,u')$ to find v'.

Deletion. Each GPU thread also handles the deletion of a vertex or a tuple. Deleting a data vertex v that maps to query vertex u will also delete all tuples $t(v,v')$ in $R(u,u')$ where v' is a data vertex that maps to u'. The thread locates vertex v and removes it from the hash tables. To delete a tuple $e(v,v')$, the thread finds a vertex v and then performs binary search for v' in $N(v,u,u')$. Since data in $N(v,u,u')$ are consecutively stored, we should rebuild $N(v,u,u')$ after the removal. For efficiency, rather than removing v' physically, we mark v as `UPDATED` and v' as `REMOVED`. After the filtering phase, we rebuild all neighbor arrays $N(v,u,u')$ where v is `UPDATED`. Examples of the search and deletion are presented in Section 4, in the description of the filtering.

3.4 Analysis

The amortized cost of inserting a key into the Cuckoo hash tables is $O(1)$. Constructing the first level of the Cuckoo tries in parallel on GPU inserts at most $|V(G)|$ vertices, resulting in an $O(|V(G)|)$ time complexity. In the generation of the second level, each warp iterates over all neighbors of a vertex, and the total number of vertices to visit among all warps is at most $2|E(G)|$. The number of cells in Cuckoo hashing is linear to the number of vertices to insert, so the prefix sum operator takes $O(\log|V(G)|)$ time. Building the third level processes the same set of vertices as the second level. In summary, the amortized time complexity of constructing the Cuckoo tries for a relation is $O(|E(G)|)$. Subsequently, a search on our Cuckoo tries takes $O(1)$ time to locate a vertex v at the first level, and $O(\log|V(G)|)$ time, in the worst case, to further locate a neighbor v' of v in $N(v,u,u')$. A deletion, if we mark the deleted tuples without rebuilding, has the same time complexity as the search operation. Since we store at most $|E(G)|$ candidates edges for each query edge, the space cost of EGSM's index is $O(|E(Q)||E(G)|)$.

4 FILTERING PHASE

This section describes our filtering rules and their implementation details with our Cuckoo trie structure.

4.1 Filtering Rules

Initially, we build each relation $R(u,u')$ by inserting two tuples $t(v,v')$ and $t(v',v)$ for each $e(v,v') \in E(G)$. Then, we perform filtering to remove tuples $t(v,v')$ if v and v' cannot match u and u' simultaneously in any final result. Based on the injective constraint in Definition 2.1, we propose a three-step filtering procedure to reduce the number of tuples in a relation as much as possible. In the first filtering step, we perform a *label filter* by removing tuples $t(v,v')$ from $R(u,u')$ where v has a label different from u or v' has a label different from u'. In the second step, we execute a *neighbor-label counter filter*. Specifically, if $M(u) = v$ is in a match, each neighbor of u in Q will map to a distinct neighbor of v. Therefore, we compare the number of neighbors of u and v that have the same label, and remove tuples $t(v,v')$ in $R(u,u')$ if either $NLC(v) \succcurlyeq NLC(u)$ or $NLC(v') \succcurlyeq NLC(u')$ does not hold.

In the third step, we prune tuples by the wedge filter. We perform filtering based on the observation that, if $M(u) = v$ in a match and there is a wedge u'-u-u'' centered at u, then v maps to u in a match of u'-u-u'' in G, and consequently, v maps to u in a tuple in both $R(u', u)$ and $R(u, u'')$. Therefore, if v does not map to u in any tuple in $R(u', u)$, we can further remove all tuples $t(v, v'')$ from $R(u, u'')$. Taking our example relations $R(u_0, u_2)$ and $R(u_2, u_3)$ in Figure 2 as an example, Figure 4a shows the matches of wedge u_1-u_2-u_3 by joining the two input relations. v_7 does not map to u_2 in any match because v_7 does not map to u_2 in any tuple of $R(u_2, u_3)$. Therefore, the tuple $t(v_3, v_7)$ can be pruned from $R(u_1, u_2)$. We prune all relations adjacent to a query vertex at a time. Given $u \in V(Q)$, there are $|N(u)|(|N(u)|-1)$ distinct wedges. We optimize the computation by first sorting $N(u)$ by vertex ID into $\langle N(u)\rangle$. Then, we iterate over every two adjacent vertices u' and u'' in $\langle N(u)\rangle$ and process the wedge u'-u-u''. After that, we process $\langle N(u)\rangle$ again in the reversed order. This way, we can prune all unpromising tuples in relations adjacent to u by examining only $2(|N(u)| - 1)$ wedges.

To complete filtering, we conduct this procedure on each $u \in V(Q)$. The order of processing the vertices, denoted as a *filtering order* δ, affects the filtering result. We want the removal of a tuple to trigger as many tuples as possible to be removed in other adjacent relations. Take a simple example, where the query graph is a path with n vertices, $v_0, v_1, \cdots, v_{n-1}$, we can process vertices along the order of the vertex ID and then the reverse order of the ID to remove all tuples not leading to a full match. However, for a query graph containing cycles, we cannot generate a δ that matches the order of vertices in each path in Q. Therefore, we generate δ in a heuristic way: We select an arbitrary query vertex and push all other vertices into δ in the breath-first order. Then, we filter by processing all vertices once in the order of δ and then another time in the reverse order.

4.2 Implementation Details

To minimize the write conflict among threads on the GPU, we store candidate information as a 2-dimensional bit matrix B with $|V(G)|$ columns and $|V(Q)|$ rows in the global memory $B[u][v] = 1$ indicates $v \in V(G)$ is a candidate of $u \in V(Q)$. We conduct the first two steps of our pruning procedure on B, and then build the Cuckoo tries for each relation based on B and perform the third pruning step.

Algorithm 2 shows our filtering procedure on the GPU. First, Line 1 initializes each element in B to 0. Then, Lines 2-3 get candidates for each query vertex. The kernel function GetCandidate performs the first two steps of our filtering procedure. In this kernel, each warp maintains a local bit array B_{shared} with 128 bits in the shared memory to process 128 consecutive data vertices. When the processing finishes, B_{shared} is copied to B with one memory transaction. Specifically, suppose the first vertex to process by a warp is v_{start}, the warp will check if the label of each vertex is the same as $L(u)$ and push those vertices that have passed the label filter into *Queue* in the shared memory (Lines 15-16). Then, the warp checks each $v \in Queue$ and computes $NLC(v)$ by looping over the neighbors of v in parallel. $NLC(v)$ is also stored in the shared memory. If $NLC(v) \geqslant NLC(u)$, the warp will set the corresponding bit in B_{shared} to 1 (Lines 17-19).

After candidates are recorded, Lines 4-5 build the Cuckoo tries based on B and G. The kernel function BuildCuckooTries performs the batch insertion operation described in Section 3.3. In building the first level of $CT(u, u')$, the algorithm inserts only those vertices v where $B[u][v] = 1$ into the hash tables. Then, at the second and third levels of $CT(u, u')$, only those neighbors v' of v where $B[u'][v'] = 1$ are considered.

Next, Lines 6-10 perform the third pruning step. For each $u \in \delta$, we sort $N(u)$ by vertex ID into $\langle N(u)\rangle$. Then, for each pair of adjacent vertices v' and v'' in $\langle N(u)\rangle$, we remove tuples $t(v, v'')$ in $R(u, u'')$ if v does not exist in $R(u, u')$ as attribute u. Since $R(u, u'')$ corresponds to two sets of Cuckoo tries $CT(u, u'')$ and $CT(u'', u)$, we remove tuples from both structures to keep them

Algorithm 2: Filtering on the GPU

```
   Input: a query graph $Q$, a data graph $G$
   Output: Cuckoo Tries $CT$s
 1 Create $B$ and initialize each bit to 0;
 2 for $u \in V(Q)$ do
 3     GetCandidate($u, Q, G, B$);
 4 for $e(u, u') \in E(Q)$ in both directions do
 5     $CT(u, u'), CT(u', u) \leftarrow$ BuildCuckooTries($B, G$);
 6 for $u \in \delta$ do
 7     foreach adjacent elements $u', u'' \in \langle N(u) \rangle$ do
 8         PruneCuckooTries($CT(u, u'), CT(u, u''), CT(u'', u)$);
 9     foreach adjacent element $u', u'' \in$ reversed $\langle N(u) \rangle$ do
10         PruneCuckooTries($CT(u, u'), CT(u, u''), CT(u'', u)$);
   /* process $u$ in the reversed order of $\delta$, omitted */
11 for $e(u, u') \in E(Q)$ do
12     CompactCuckooTries($CT(u, u')$);
13     CompactCuckooTries($CT(u', u)$);
14 Kernel GetCandidate($u, Q, G, B$)
15     each warp initializes $B_{shared}$ and gets 128 vertices starting from $v_{start}$;
16     a lane pushes $v$ to $Queue$ if $L(v) = L(u)$;
17     foreach $v \in Queue$ do
18         the warp loops over $N(v)$ to compute $NLC(v)$;
19         if $NLC(v) \geqslant NLC(u)$ then $B_{shared}[v - v_{start}] = 1$;
20     the warp copies $B_{shared}$ to $B[u][v_{start} : v_{start} + 128)$;
21 Kernel PruneCuckooTries($CT(u, u'), CT(u, u''), CT(u'', u)$)
22     each thread gets a vertex $v$ on the first level of $CT(u, u'')$;
23     the thread checks if $v$ is on the first level of $CT(u, u')$;
24     if not, remove $v$ from the first level of $CT(u, u'')$ and push $v$ to $Queue$;
25     each warp gets a $v$ from $Queue$ and the corresponding $N(v, u, u'')$;
26     each lane gets a $v'' \in N(v, u, u'')$ and finds $t(v'', v)$ in $CT(u'', u)$;
27     the lane marks $v$ as UPDATED and $v''$ as REMOVED;
```

consistent with $R(u, u'')$. After that, we do the same along the reversed order of $\langle N(u) \rangle$. This way, all relations containing u have the same set of vertices with attribute u at the end of the current round.

Specifically, in the function PruneCuckooTries, each thread gets a vertex v at the first level of $CT(u, u'')$ and checks with hash search if v exists at the first level of $CT(u, u')$ (Lines 22-23). If not, all tuples with v mapping to u will be removed from both $CT(u, u'')$ and $CT(u'', u)$. We directly remove v from the first level of $CT(u, u'')$, and then loop over each v'' that are previously in $N(v, u, u'')$ to remove $t(v'', v)$ from $CT(u'', u)$ (Lines 24-26). Rather than removing a tuple physically, we mark v'' at the first level as UPDATED, and v at the third level as REMOVED (Line 27). At the end of the filtering phase, the algorithm loops over each set of Cuckoo tries, and each warp selects a vertex v that is marked as UPDATED, compacts the $N(v, u, u')$, and updates the Off arrays.

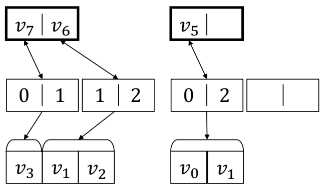
Example 4.1. Figure 4b shows the bit-matrix B constructed on our example graphs in Figure 1, where the cell on row u and column v is 0 indicates that v is pruned from the candidate set of u in the first two steps. At the beginning of the third step, we build the relations based on B. The resulting relations are shown in Figure 2. After that, the Cuckoo tries corresponding to the relations are also generated, among which $CT(u_0, u_2)$ and $CT(u_2, u_0)$ are illustrated in Figure 3a and Figure 4c, respectively. In PruneCuckooTries, since v_7 exists in $R(u_2, u_0)$ with attribute u_2

u_0	u_2	u_3
v_0	v_5	v_8
v_1	v_5	v_8
v_1	v_6	v_{10}
v_2	v_6	v_{10}

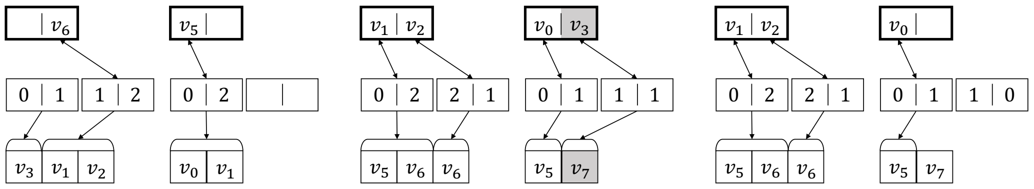
(a) Matches.

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}
u_0	1	1	1	1	0	0	0	0	0	0	0	0
u_1	1	1	1	1	0	0	0	0	0	0	0	0
u_2	0	0	0	0	0	1	1	1	0	0	0	0
u_3	0	0	0	0	0	0	0	0	1	0	1	0
u_4	0	0	0	0	0	0	0	0	0	1	0	0

(b) Example B.



(c) $CT(u_2, u_0)$ before pruning.



(d) $CT(u_2, u_0)$ after pruning. (e) $CT(u_0, u_2)$ in pruning. (f) $CT(u_0, u_2)$ after pruning.

Fig. 4. Example Cuckoo tries.

but does not exist in $R(u_2, u_3)$ with attribute u_2, we remove v_7 from the first level of $CT(u_2, u_0)$, resulting in Figure 4d. Then, to keep $CT(u_0, u_2)$ consistent with $CT(u_2, u_0)$, we loop over each vertex in $N(v_7, u_2, u_0)$ before the removal. There is only one vertex v_3 in $N(v_7, u_2, u_0)$. So we search the tuple $t(v_3, v_7)$ in $CT(u_0, u_2)$, and mark v_3 as `UPDATED` and v_7 as `REMOVED`, as shown in Figure 4e. After that, $N(v_3, u_0, u_2)$ becomes empty. So we set $Off[1][3]$ to 0 and remove v_3 from the first level, as shown in Figure 4f.

4.3 Analysis

Complexity. In Algorithm 2, Line 18 visits all neighbors of a vertex v to build $NLC(v)$ and Line 19 spends $O(|V(Q)|)$ time on comparing two NLCs. Therefore, the function GetCandidates costs $O(|E(G)|)$ time. Next, Line 5 takes amortized $O(|E(G)|)$ time to generate both $CT(u, u')$ and $CT(u', u)$. After that, in the function PruneCuckooTries, Lines 22-24 take constant time, and Line 26 takes $O(\log |V(G)|)$ time. Since Line 26 is processed at most $O(E(G))$ times, the overall time complexity of PruneCuckooTries is $O(|E(G)| \log |V(G)|)$. Finally, the function CompactCuckooTries costs $O(|E(G)|)$ time. In total, the amortized time complexity of Algorithm 2 is $O(|E(Q)||E(G)| \log |V(G)|)$. The complexity is higher than the filtering step of GSI. However, due to the parallel processing on our Cuckoo tries and GPU-optimized filtering procedure, the actual filtering time is generally shorter than GSI. We present the detailed experimental results and discussions in Section 6.

Pruning Power. We compare the pruning power of our method with that of GSI, the state-of-the-art GPU-based subgraph matching algorithm. GSI performs a filtering procedure similar to our first two steps. However, the pruning power of the first two steps of our method is greater than GSI for the following two reasons. First, we build an NLC array that contains the number of occurrences of each vertex label, whereas GSI hashes each label in Σ to a fixed range. If the number of labels is large, multiple labels may have the same hash value and are not distinguished in pruning. Second, GSI compares the existence of the neighbors with a certain label. In contrast, we compare the exact number of neighbors for each vertex label. Due to this difference, the candidate sets generated in GSI may contain some vertices that are pruned in EGSM.

5 ENUMERATION PHASE

5.1 Parallel DFS-based Enumeration

As described in Section 1, BFS-based enumeration on the GPU faces severe limitations in both time and memory efficiency. To seek an alternative solution, we naturally consider the DFS execution model in the extension of partial results. Algorithm 3 shows our DFS-based subgraph matching algorithm on the GPU.

At Lines 1-2, the algorithm selects the first vertex u_0 and generates all partial matches of u_0, $\mathcal{M}_0$, by the kernel function GetFirstMatches. Each relation $R(u_0, u_0')$ ($u_0' \in N(u_0)$) corresponds to a candidate set of u_0, namely $\pi_{u_0}R(u_0, u_0')$. To reduce the number of invalid partial results, rather than selecting an arbitrary $\pi_{u_0}R(u_0, u_0')$, the algorithm gets $\mathcal{M}_0$, by intersecting $\pi_{u_0}R(u_0, u_0')$ for all $R(u_0, u_0')$ where $u_0' \in N(u_0)$. Specifically, the algorithm selects the smallest set s_{min} and assigns each vertex in s_{min} to a thread. Then, the thread checks whether the assigned vertex exists in all other sets. Since $\pi_{u_0}R(u_0, u_0')$ is stored at the first level of $CT(u_0, u_0')$ as hash tables, the thread performs the containment check with hash search. After that, each warp gets a partial result M (Line 3) and enumerates all full matches from M (Line 4).

In the kernel function ExtendDFS, we denote the set of query vertices that are not mapped, V^-. A warp selects a vertex u from V^- (Line 6) and finds the candidates of u by set intersection. Specifically, the warp generates a set of sets $\mathbb{S}$, including $N(M(u'), u', u)$ for each mapped $u' \in N(u)$ and $\pi_u R(u, u')$ for each unmapped $u' \in N(u)$ (Line 7), and then finds the smallest set s_{min} (Line 8). Each lane of the warp picks a vertex v from s_{min} and checks whether v exists in all other sets (Lines 10-11). We perform binary search on $N(M(u'), u', u)$ since a neighbor array is sorted, and hash search on $\pi_u R(u, u')$ since the first level of $CT(u, u')$ is a set of hash tables. If there are remaining vertices in V_- to extend, the lanes will push the vertices that pass the intersection check into a temporal array *temp* (Lines 12-13). Then, the warp loops over each vertex v_t in *temp*, adds a mapping (u, v_t) to M, and extends M recursively (Lines 14-16). After all vertices in *temp* are

Algorithm 3: SubgraphMatchingDFS on the GPU

```
   Input: a query graph $Q$, relations $R$s
   Output: subgraph matches $\mathcal{M}$
1  $u_0 \leftarrow$ SelectFirst($V(Q)$);
2  $\mathcal{M}_0 \leftarrow$ GetFirstMatches($R(u_0, u_0')$s), $\mathcal{M} \leftarrow \{\}$;
3  each warp gets an $M$ from $\mathcal{M}_0$;
4  ExtendDFS($V(Q) - \{u_0\}$, $M$, $\mathcal{M}$);
5  Kernel ExtendDFS($V^-$, $M$, $\mathcal{M}$)
6      $u \leftarrow$ SelectNext($V^-$);
7      $\mathbb{S} \leftarrow \{N(M(u'), u', u) | u' \in N_+(u)\} \cup \{\pi_u R(u, u') | u' \in N_-(u)\}$;
8      $s_{min} \leftarrow s \in \mathbb{S}$ with the smallest cardinality;
9      while $s_{min} \neq \phi$ do
10         each lane picks a vertex $v$ from $s_{min}$;
11         if $v \in s$ for all $s \in \mathbb{S}$ then
12             if $V^- - \{u\} \neq \phi$ then
13                 add $v$ to a temp array $temp$;
14                 foreach $v_t \in temp$ do
15                     add $(u, v_t)$ to $M$;
16                     ExtendDFS($V^- - \{u\}$, $M$, $\mathcal{M}$);
17             else
18                 add $(u, v)$ to $M$;
19                 write $M$ to $\mathcal{M}$
```

processed, the warp will check the other vertices in s_{min}. If all query vertices are mapped, the lanes with an intersection result will add a mapping (u, v) to M and report M (Lines 18-19).

In the parallel DFS extension, a warp enumerates all matches by extending from a partial match. We call a partial match to extend a *task*. Except for $\mathcal{M}_0$, Algorithm 3 does not materialize any partial result, but it is not time efficient due to the under-utilization of the GPU. Specifically, the number of partial matches to enumerate for extending a query vertex is not balanced between tasks, and the imbalance grows exponentially as the number of query vertices increases. Therefore, a warp may have an exponentially greater workload than another, leading to the idling of short-running warps. Furthermore, the number of vertices in $\mathcal{M}_0$ is likely smaller than the number of warps on the GPU, so some warps have no work to do during the entire matching process.

5.2 Parallel BFS-DFS Enumeration

Considering the advantages and drawbacks of both the BFS- and DFS-based enumeration, we propose BFS-DFS, a hybrid scheme containing both BFS and DFS. In this subsection, we use another example query graph shown in Figure 5 to better depict our enumeration method. Figure 6 shows the complete BFS-DFS matching procedure for the query graph. Specifically, we first partition the vertex set of Q into n groups, namely $V_0, V_1, \cdots, V_{n-1}$ ($n = 5$ in this example). Then, we generate $\mathcal{M}_0$ through GetFirstMatches in the same way as the DFS method. However, rather than extending a partial result in $\mathcal{M}_0$ all the way to full matches of Q, we only extend it to partial results matching the induced graph of Q on the first group of vertices, i.e., $Q[V_0]$. After that, we extend all those partial results to match $Q[V_0 \cup V_1]$. Figure 6 then illustrates the details of extending vertices in group V_1. All the N matches of $Q[V_0]$ are stored in the global memory. Each warp gets one $Q[V_0]$ match and enumerates the results by mapping each query vertex in V_1 along φ. In the zoom-in figure of Warp#N-1 on the right of Figure 6, the numbers enclosed by parentheses represent the step number. All lanes in the warp try to map the first vertex and generate at most 32 partial results (Step (1)). Then, the warp further extends the results one by one by mapping other remaining query vertices in the group (Steps (2)-(4)). When all the partial results are consumed, the lanes proceed to enumerate other partial results (Step (5)). Within DFS, a partial result with all vertices in V_1 mapped is written to the global memory. The extension step is performed recursively until all full matches are enumerated.

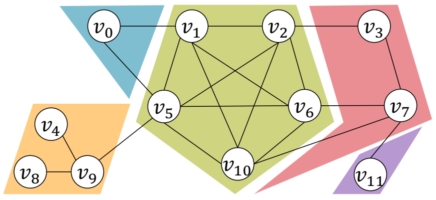


Fig. 5. Example of query vertex grouping.

To optimize the matching performance, it is crucial to design a good vertex grouping method as well as specify the order between groups. Even though one of our objectives is to alleviate load imbalance, we have limited control over the execution pattern of tasks on a GPU. In particular, we can neither assign a task to a specific warp nor decide the execution order between tasks. However, we can make each group small to reduce the workload of each task, thus alleviating load imbalance. We denote all matches of Q in G as $SM(Q, G)$, and the size of $SM(Q, G)$ as $|SM(Q, G)|$. Then, the total number of partial results written to (or read from) the global memory is $\sum_{i=0}^{n-1} |SM(Q[\cup_{j=0}^{i} V_i], G)|$. We want to minimize this expression because the decrease in partial results leads to (1) less memory consumption, (2) less global memory access, and (3) fewer invalid partial results. However, it is

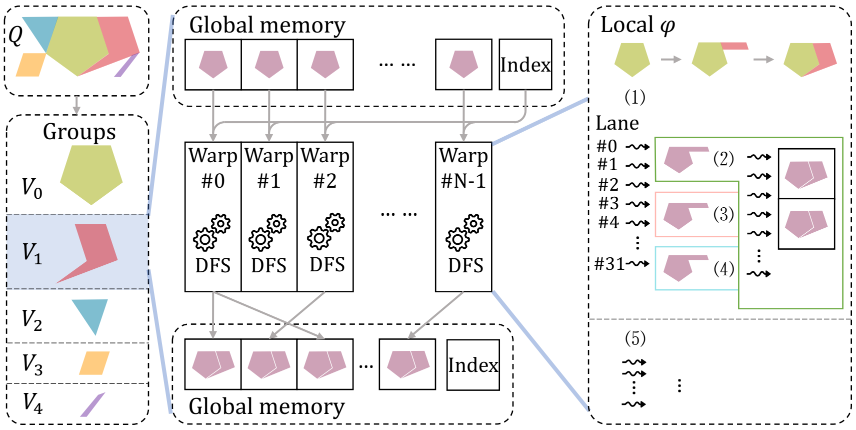


Fig. 6. The BFS-DFS matching strategy.

expensive to enumerate all possible grouping choices of Q, especially when Q is large. Furthermore, given a grouping result, computing the expression is NP-hard since the execution of each *SM* function is NP-hard. Therefore, we develop a heuristic approach to partition Q based on its structure.

First, we extract the core structure of the query graph, namely the largest subgraph of Q where each vertex has a degree of at least 2 in the subgraph. Then, we denote all vertices in the core structure *core vertices*, and other vertices *tree vertices*. If there is no core, we choose an edge $e(u, u')$ from Q with maximum $d(u) + d(u')$ and set u and u' as the core vertices. Then, the induced subgraph of Q on the tree vertices composes a set of trees. We match all the core vertices before the tree vertices since previous studies have shown that such an execution order can reduce much unnecessary enumeration [5]. After that, we assign vertices in each tree into a group and order the groups by the cardinality of the relations: For each tree, we can find a core vertex u_c connected to a vertex u_t in the tree, and we place the trees in the ascending order of $|R(u_c, u_t)|$.

After that, we group core vertices based on the local densities. Specifically, we extract all 4-cliques from the core structure since they are regarded as dense sub-structures of Q in previous work [47]. These extracted cliques may be overlapping, adjacent or unconnected, making it hard to group vertices. Therefore, we get the induced graph of all vertices in cliques, and select the connected component with the maximum number of vertices. If there is no 4-clique, we select a triangle in which the three endpoints are all core vertices, and the sum of the degrees of the three endpoints is the largest. If there is no triangle, we select an edge from the core where the sum of the degree of the two endpoints is the largest. All selected vertices are called *dense vertices*, forming the set V_d. V_d is the first group to match, because dense subgraphs generally appear less frequently than sparse ones in G. The remaining core vertices are called *sparse vertices*, forming the set V_s. To further group partition V_s, we get the induced graph of Q on V_s and organize vertices in each connected component into a group. These groups are matched after dense vertices but before tree vertices. Similarly, we order these groups by the cardinality of the relations: For each group, we can find a query edge $e(u_d, u_s)$, where u_d is a dense vertex and u_s is a sparse vertex in this group, with the minimum cardinality $|R(u_d, u_s)|$ in the group. We order the groups in the ascending order of this minimum cardinality.

Example 5.1. To partition the query vertices of the example graph in Figure 5 into groups, we first identify the core structure. Then, we generate the dense vertices $V_0 = V_d = \{v_1, v_2, v_5, v_6, v_{10}\}$ and the sparse vertices $V_s = \{v_0, v_3, v_7\}$. After that, we organize the sparse vertices into two groups

Algorithm 4: SubgraphMatchingBFS-DFS on the GPU

```
Input: a query graph $Q$, relations $R$s
Output: subgraph matches $\mathcal{M}$
1  $V_0, V_1, \cdots, V_{n-1} \leftarrow$ VertexGrouping(V(Q));
2  $u_0 \leftarrow$ SelectFirst($V_0$);
3  $\mathcal{M}_{old} \leftarrow$ GetFirstMatches($R(u_0, u'_0)$s), $\mathcal{M} \leftarrow \{\}$, $V_0 \leftarrow V_0 - \{u_0\}$, $V^- \leftarrow V(Q) - \{u_0\}$;
4  for $0 \leq i < n$ do
5      each warp gets an $M$ from $\mathcal{M}_{old}$;
6      ExtendDFS($V_i$, $M$, $\mathcal{M}$);
7      if not finished due to out-of-memory then break;
8      $\mathcal{M}_{old} \leftarrow \mathcal{M}$, $\mathcal{M} \leftarrow \{\}$, $V^- \leftarrow V^- - V_i$;
9  each warp gets an $M$ from $\mathcal{M}_{old}$;
10 ExtendDFS($V^-$, $M$, $\mathcal{M}$);
```

based on the connectivity, namely $V_1 = \{v_3, v_7\}$ and $V_2 = \{v_0\}$. Finally, we further divide the tree vertices into two groups $V_3 = \{v_4, v_8, v_9\}$ and $V_4 = \{v_{11}\}$.

5.3 Memory Management

Our BFS-DFS strategy needs to store all partial results when each group of query vertices finishes mapping, so the memory consumption is as large as the BFS-based approaches. Therefore, based on our grouping method, we propose a memory management framework to maximize the usage of the global memory as well as avoid running out of memory. Our matching strategy first finds all partial matches of $Q[V_0]$ and then recursively enumerates matches of $Q[\cup_{j=0}^{i} V_i]$ by extending those of $Q[\cup_{j=0}^{i} V_{i-1}]$. Considering the memory access pattern, we propose to store partial results in a cyclic queue in the GPU's global memory. When starting the execution of group i, we remove the partial results of group $i-2$ (if exists) from the front of the queue to save space. Then, as new partial results are enumerated, we write them to the back of the queue.

However, the memory consumption of the partial results generated within a group may exceed the free space in the queue. To avoid running out of memory, before matching each new group, we count the number of vertices that can be stored in the free space of the queue, denoted as *avl*. In round i, each partial result takes $|\cup_{j=0}^{i} V_i|$ space due to the materialization. Therefore, the remaining space in the cyclic queue can store at most $max\_res = \left\lfloor \frac{avl}{|\cup_{j=0}^{i} V_i|} \right\rfloor$ partial results. We use a variable num_res to record the number of partial matches generated in the current round. As a thread finds a partial result to write, it first adds num_res to 1 atomically and then writes the result to the back of the queue based on num_res without conflict. If, at a time, num_res is greater than max_res after the addition, this condition indicates there is no more space in the queue, and further enumeration cannot be processed. In this case, the kernel function exits directly, and the algorithm rolls back to the beginning of the loop, combines all the remaining vertices into a single group, and then processes the group by the DFS execution method.

Our matching framework is summarized as Algorithm 4. Line 1 divides query vertices into groups. Then, Lines 2-3 generate partial matches of the first query vertex, the same as Algorithm 3. After that, we perform the DFS method to match each group of vertices. If all partial results can be written to the memory, we update the partial results and continue matching. Otherwise, we exit the loop and conduct the DFS-based execution on V^-.

Compared with DFS, our parallel BFS-DFS enumeration reduces the number of query vertices to extend (the search path length), significantly alleviating load imbalance. Also, because the number of partial results is generally much greater than the size of $\mathcal{M}_0$, our strategy consumes more warps to compute and thus takes less time to finish than DFS. Our BFS-DFS is also faster than BFS (Algorithm 1) because the grouping can reduce both the number of invalid partial results and global memory accesses. Furthermore, the memory management framework ensures our method never runs out of memory.

Remarks. NEMO [30] proposed a memory management approach on the GPU based on pipelining to avoid running out of memory. A few main differences between NEMO and EGSM are as follows: (1) NEMO keeps all partial results, which are exponential, as long as the memory is not full, whereas EGSM deletes the partial results once they are not needed; (2) EGSM reuses a queue for storing partial results, but NEMO must allocate memory dynamically; and (3) In NEMO, splitting the partial results into pieces for pipelining incurs runtime overhead, and the parameter for the pipeline stage size may need to be adjusted based on datasets.

5.4 Matching Order

Section 5.2 describes the ordering method between groups. However, the exact order between query vertices in each group is not determined. In this section, we propose to determine the matching order within a group. Prior work on the GPU adopts a fixed matching order for all partial results. However, G may show different local properties. For a partial result M, we determine the matching order based on the sub-structure on G around M. For the remaining vertices in the group, we want to specify an order so that the number of partial results generated for extending M is the smallest. This ordering, again, is an NP-hard problem. Thus we adopt a greedy approach by choosing a query vertex with the minimum number of data vertices to map in each extension step. More specifically, we select an unmapped vertex u whose corresponding $\mathbb{S}$ leads to the fewest intersection result. However, most cardinality estimation methods of set intersections are heavy-weight, unsuitable to execute on each extension of the partial result. In contrast, latest CPU-based algorithms [16, 20] store the estimated cardinality in the index, instead of estimating on the fly. However, accessing the index stored in the global memory at each extension is costly, due to the high latency of the GPU global memory.

In our EGSM, we employ a lightweight cardinality estimation method based on the *min property*, which states that the size of the intersection result is bounded by the cardinality of the smallest set in $\mathbb{S}$. Furthermore, We store the cardinality of related sets in the shared memory on the GPU for fast access. We denote the set of all unmapped vertices in group i as V_i^-, and define an *extendable vertex* in group i as a member of V_i^- with at least one neighbor in Q already mapped. During the enumeration, a warp maintains a bitmap *mapped_vs* in the shared memory. The bit at position i of *mapped_vs* is set to 1 if vertex i is mapped, and 0 otherwise. In the GPU constant memory, we store a bitmap NBR_BITS_u for each $u \in V(Q)$ and a bitmap G_BITS_{gi} for each group gi. The bit at position i of NBR_BITS_u is 1 if the vertex i is a neighbor of u and the bit at position i of G_BITS_{gi} is 1 if gi contains vertex i. Furthermore, each warp maintains the sizes of $N(M(u), u, u')$ in the shared memory, where u is mapped, u' is unmapped, and $e(u, u') \in E(Q)$. We also store the cardinality of each attribute in a relation in the shared memory. In the function SelectNext, a warp owns a partial result, and each lane first checks whether a vertex is extendable in parallel with bitwise operations. Specifically, if we are processing group gi, the query vertex with ID i is extendable if (1) the position i of *mapped_vs* is 0, (2) the position i of G_BITS_{gi} is 1, and (3) the bitwise AND between NBR_BITS_u and *mapped_vs* is not zero. After that, lane i gets the cardinalities of all sets in $\mathbb{S}$ for extending vertex i in $V(Q)$ and finds the smallest cardinality $|s_{min}|$. Then, all lanes with an extendable vertex communicate through parallel reduction operations within the warp to get the vertex with minimum $|s_{min}|$.

Analysis. As in previous work, we assume Qs is small, with $|V(Q)| < 32$. We let each lane of warp process a query vertex u. The lane first checks the extendability with an $O(1)$ complexity.

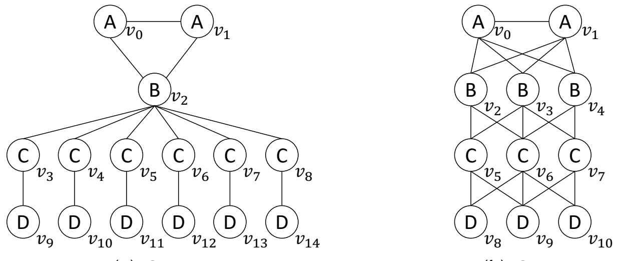
Then, in the worst case, it takes $O(V(Q))$ time to loop over all neighbors of u. The warp reduction operation takes $O(\log V(Q))$ time. So the overall time complexity of SelectNext is $O(V(Q))$. In previous studies, the function SelectNext only takes $O(1)$ time to retrieve the next vertex to extend from the fixed order. However, since the query graph is generally small and related variables are stored in the memory with low access latency, the actual computation time will be short. In our evaluation in Section 6, we show that the benefit of our adaptive ordering method outweighs its overhead.

5.5 Load Balancing

We separate the query vertices into groups, and load imbalance may exist between tasks when matching a group of vertices. Therefore, we propose load balancing methods within a group. Our main idea is to split a *heavy task* (a task that leads to many partial matches) into sub-tasks, each solved by a warp. In the following, we illustrate our load balancing methods with examples.

Suppose all query vertex in Q in Figure 1a are within the same group and the matching order is $\langle u_0, u_1, u_2, u_3, u_4 \rangle$. In Figure 7a, given a task $\{v_0, v_1, v_2\}$, the large number of partial results comes from the large number of candidates of u_3. Therefore, our first strategy is to split a large candidate set into smaller ones. Specifically, if we find $|s_{min}|$ is greater than a threshold TH_1 (4 in this example) before the intersection of $\mathbb{S}$, we will separate s_{min} into subsets and dynamically launch a kernel function to assign each subset to a warp. Then, each warp performs the intersection over the assigned subset and conducts further matching. For example, suppose each subset contains 3 vertices, then the algorithm launches two warps, one to process v_3, v_4, and v_5, and the other to process v_6, v_7, and v_8. Since we split a task before the enumeration of the task, we call this strategy *looking-forward* load balancing.

In the second example in Figure 7b, the task $\{v_0, v_1\}$ also has many partial results. However, no s_{min} has a length greater than 4 during the extension, and the large number of candidates in each step of the enumeration accumulates to the large number of partial results of $\{v_0, v_1\}$. To detect this kind of heavy tasks, we maintain the number of partial results for each task. If the number is greater than a threshold, we dynamically launch a kernel function to assign each of the following partial results on the current extension level to a warp. In this example, $\{v_0, v_1, v_2\}$ results in 7 partial results (full results are also counted as partial results). Suppose we set another threshold TH_2 to 6. As the warp finishes the extension of $\{v_0, v_1, v_2\}$ and backtracks to v_2, it finds that the number of partial results is greater than the threshold, so it will launch two warps to process the following tasks, namely $\{v_0, v_1, v_3\}$ and $\{v_0, v_1, v_4\}$. This way, the heavy task $\{v_0, v_1\}$ is executed by multiple warps in parallel. Since the heavy tasks are identified after their enumeration, we call this strategy *looking-backward* load balancing.


A v_0
A v_1
B v_2
C v_3 C v_4 C v_5 C v_6 C v_7 C v_8
D v_9 D v_{10} D v_{11} D v_{12} D v_{13} D v_{14}
A v_0
A v_1
B v_2 B v_3 B v_4
C v_5 C v_6 C v_7
D v_8 D v_9 D v_{10}

(a) Case 1. (b) Case 2.

Fig. 7. Example of load imbalance.

6 EXPERIMENTS

In this section, we introduce the experimental settings and then report the results.

6.1 Experimental Setup

Methods Under Study. We compare EGSM with the latest subgraph matching methods on GPUs, including GSI [55] and CuTS [54], which outperform previous subgraph matching algorithms on both CPUs and GPUs in their experiments. The code of both algorithms is publicly available on Github[1,2].

We also design several competitors for EGSM to study the individual techniques. First, we denote NoF3 as a variant of EGSM with the third filtering step disabled. In addition, to evaluate the parallel BFS-DFS scheme, we replace it with a parallel DFS algorithm in EGSM, denoted by DFS, by assigning all query vertices to a single group. Similarly, we develop a BFS-based approach in EGSM, denoted by BFS, by assigning each query vertex to a distinct group. DFS usually fits in the memory constraint, but BFS does not, due to the large memory consumption of partial results in the global memory. Therefore, we also integrate our memory management framework into BFS. Specifically, if, in an extension step, the memory space is not enough for storing all partial results, BFS performs DFS-based matching on all the remaining query vertices. Additionally, to study the efficiency of our adaptive ordering method, we replace EGSM's adaptive ordering with CuTS' static ordering method within a group, denoted by Static. Specifically, we set the next query vertex to extend on every extension as an unmapped vertex with the maximum number of mapped neighbors. Finally, we create a version EGSM-NoLB with load balancing disabled.

All competing algorithms are compiled with gcc 11.2 and nvcc 11.7. We run our experiments on a Linux machine with an Intel Xeon E5-2683 CPU and 256GB main memory. The machine is enquiped with an Nvidia GeForce RTX 2080 Ti GPU with 68 streaming multiprocessors and 12-GB global memory space.

Datasets. We use six data graphs, including DBLP, Enron, Github, Gowalla, Patents, and Wikitalk, following previous studies. DBLP maintains the co-authorships between authors; Enron records email communication links within a set of users; Github represents the social network of Github developers; Gowalla is a location-based online social network; Patents contains the citation relationships among US Patents; and Wikitalk stores the communication networks between users in Wikipedia. All the datasets are downloaded from the website of the SNAP group [28]. The Patents dataset contains labels originally, but there are more than 400 labels, making the queries trivial to solve. The other datasets do not contain any label. Therefore, we uniformly attach labels to all datasets as in related work [55]. In particular, we randomly assign a label to each vertex in the graph following the power-law distribution to simulate such real-world graphs. We also vary the number of distinct labels to evaluate the scalability of competing algorithms. Table 2 lists the properties of the datasets, including $|V|$, $|E|$, $|\Sigma|$, the average degree d_{avg}, the maximum degree d_{max}, and the maximum core number c_{max}.

To generate a query graph, we extract a subgraph of G by random walk. Specifically, given the desired number of vertices n, we first randomly select a vertex from G and put it into the subgraph Q_s. Then, we extend Q_s by adding a neighbor v' of a vertex $v \in V(Q_s)$ and the edges between v' and vertices in $V(Q_s)$, until $|V(Q_s)|$ reaches n. For all datasets, we use query graphs with 12 vertices by default. We vary the number of query vertices to conduct the scalability study. For each graph size in each dataset, we generate 100 query graphs.

[1]https://github.com/pkumod/GSI - commit id b16484e
[2]https://github.com/appl-lab/CuTS - commit id d090d4f

Table 2. Datasets.

Datasets	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\Sigma\rvert$	d_{avg}	d_{max}	c_{max}
DBLP	317K	1.0M	16	6.6	343	113
Enron	36K	184K	16	10.0	1K	43
Github	37K	289K	16	15.3	9K	34
Gowalla	196K	1.0M	16	9.7	14K	51
Patents	3.8M	17M	16	8.7	793	64
Wikitalk	2.4M	4.7M	64	3.9	100K	131

Metrics. To evaluate an algorithm, we use the query time as a metric, which is the time starting from both Q and G are loaded in the GPU memory and ending when all matches are found. To complete the experiments in a reasonable amount of time, we set the time limit for processing a query to one hour. We call a query with all matches found within the time limit, a *solved query*, or otherwise an *unsolved query*. A query is unsolved if the GPU memory space is insufficient for execution (OM) or the query is not completed within the time limit (OT). We report the number of solved queries in a query set as well as the average query time of a set of solved queries by default. We also examine the query time of individual queries in some experiments. Additionally, similar to previous work, we report the speedup average of algorithm $\mathcal{A}$ over $\mathcal{B}$, namely $\frac{1}{|\mathbb{Q}|}\sum_{Q\in\mathbb{Q}}\frac{t_{\mathcal{A}}(Q)}{t_{\mathcal{B}}(Q)}$, where $\mathbb{Q}$ is a set of queries, and $t(Q)$ is the query time of Q.

6.2 Overall Comparison

In this experiment, we run GSI, CuTS, and EGSM on the six datasets. To gain an overview of the performance of competing algorithms, we first show the number of solved queries on each dataset in Figure 8. As we can see, all methods fail on some queries, but the number of solved queries of our method is 2.7X and 11.6X greater than that of CuTS and GSI on average, respectively. We note that all unsolved queries of GSI and CuTS are OM, whereas those of EGSM are OT. This result is consistent with our observation of previous approaches. Specifically, even though GPUs favor the BFS extension model, BFS requires a large memory space to store partial results. However, the GPU memory space is limited, and if the memory is filled up, no further matching can be performed. Compared with GSI, CuTS has more solved queries because it designs a data structure to store partial results, which occupies less memory. In contrast, EGSM's BFS-DFS fully utilizes the global memory to store partial results under the memory capacity.

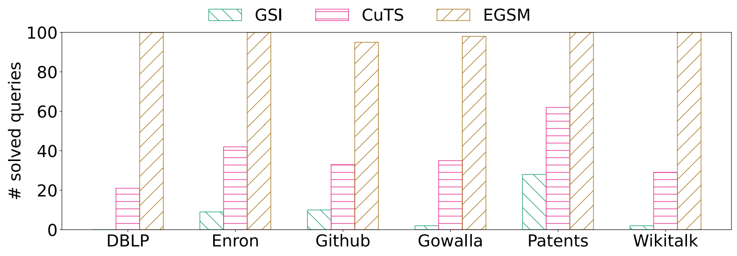


Fig. 8. Number of solved queries.

Next, we select three datasets, including Enron, Github, and Patents, whose total number of solved queries over all algorithms is the greatest. For each query solved by EGSM and CuTS (resp., GSI), we plot a point with the query time of the two algorithms as the X (CuTS or GSI) and Y (EGSM) coordinates in Figure 9. Therefore, a point under the diagonal indicates that EGSM outperforms the other algorithm on a single solved query. As shown in the figure, on almost all queries, our method is faster than both CuTS and GSI. In particular, EGSM achieves speedups of 4.8X and 702.3X on average over CuTS and GSI, respectively. The running time of GSI is dominated by the enumeration

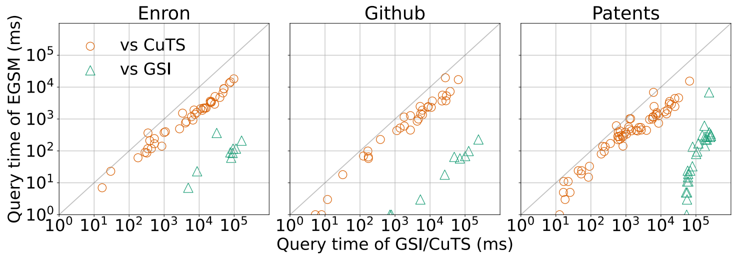


Fig. 9. Comparison on query time.

phase, caused by both the ineffective filtering and the under-optimized GPU-based implementation. Specifically, on matching a new vertex, GSI inccurs many GPU memory allocations and memory transfers between different memory spaces. Also, GSI's intersections are done on two arrays at a time, which leads to many internal synchronizations, prefix sum operations, and global memory accesses. CuTS is much faster than GSI due to the reduction in operations as well as data allocation and movement.

6.3 Evaluation of Individual Techniques

Results in the previous section have demonstrated the improvements of EGSM over existing approaches on the overall performance. In this experiment, we further examine the impact of each individual technique of EGSM.

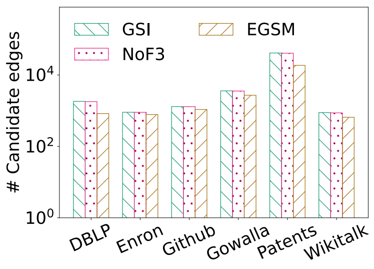
Space cost of the index. Table 3 compares the GPU memory space for storing candidates in GSI and EGSM on the six datasets. EGSM spends 12 to 25 times more memory than GSI due to the storage of candidate edges. However, the total space does not exceed tens of Megabytes, much smaller than the total space of current GPUs with typically tens of Gigabytes.

Table 3. Space cost of candidates.

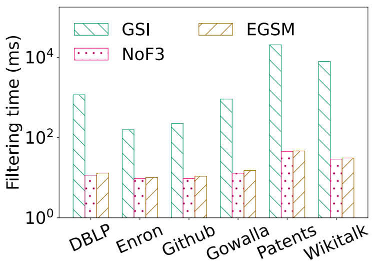
Method	DBLP	Enron	Github	Gowalla	Patents	Wikitalk
GSI	132KB	19KB	28KB	93KB	3MB	35KB
EGSM	2MB	424KB	702KB	2MB	32MB	536KB

Filtering. We first evaluate our filtering technique on the Cuckoo trie data structure and compare the filtering method with that of NoF3 and GSI. We run the filter step on each query and record the average number of edges in each relation to compare the pruning power. Since GSI stores candidate vertices instead of candidate edges, we build a relation $R(u, u')$, corresponding to each query edge $e(u, u')$, for GSI by adding all data edges $e(v, v')$ where $v \in C(u)$ and $v' \in C(u')$. The results are shown in Figure 10a. Specifically, GSI's index contains 1.7X more candidate edges than our method on average. The pruning power of the first two steps is also slightly stronger than that of GSI.

In the following, we compare the filtering time between the three methods. For GSI, we record the total time spent on vertex signature building on the CPU and candidate filtering on the GPU. In EGSM (resp., NoF3), we sum up the overall time of the three-step (resp., two-step) filtering process and the time for Cuckoo hash trie generation. Compared with GSI's filtering technique, EGSM involves an additional $\log |V(G)|$ factor in the time complexity as shown in Table 1. However, as shown in Figure 10b, EGSM spends much less time than GSI. This performance advantage is because our Cuckoo trie structure can support the insertion and deletion of candidate edges in the massively parallel environment with both theoretical guarantee and good performance in practice. Additionally, the optimized parallel filtering procedure can fully take advantage of GPU's computation power. In contrast, even though GSI performs parallel filtering on the GPU, the vertex signatures are built on the CPU only , which chould be ported on the GPU. In GSI's current version, the filtering time is 153.8X slower than EGSM on average. The filtering procedure of NoF3 is slightly faster than EGSM due to the absence of the third filtering step. However, as shown in Table 4, NoF3 is 1.09X slower than EGSM on average, likely due to the low pruning power. Consequently, the benefit of the third filtering step significantly outweighs its overhead.



(a) Number of candidates.



(b) Filtering time.

Fig. 10. Evaluation of the filtering step.

On the Enron dataset, we have tried our method with filtering orders starting from different query vertices but they resulted in the same set of candidate edges. This result indicates that our filtering method can prune most unpromising candidates and is stable with different starting vertices.

BFS-DFS matching strategy. Our enumeration approach takes advantage of both the BFS and DFS execution models. The second row in Table 4 compares the speedup average of EGSM over DFS on each dataset, which is between 78X and 179X. Due to the exponential search space of the subgraph matching, the load imbalance between warps increases dramatically as the number of vertices to match increases in the DFS approach. Therefore, some heavy tasks take much longer time than others, dominating the overall query time. We also compare EGSM with the BFS execution approach. As shown in the third row in Table 4, our execution strategy is 1.1-1.6X faster than BFS on average because our vertex grouping method reduces the invalid partial results and global memory accesses.

Adaptive Ordering. Within a group, we design an adaptive ordering method on the GPU to dynamically choose the next vertex to extend in the enumeration. In this experiment, we study the advantage of our adaptive ordering method by comparing EGSM with Static. The fourth row of Table 4 reports the speedup average over all queries in each dataset. Our ordering method achieves a speedup of up to 1.64X over the static ordering method in CuTS. Therefore, even though the time complexity for determining the next vertex on each extension is higher than Static, as illustrated in Section 5.4, the benefit offsets its overhead because (1) We take advantage of the low-latency shared memory on the GPU; and (2) A good matching order can reduce the number of invalid partial results thus reducing the overall query time.

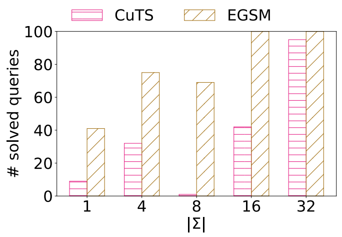
Table 4. Average speedup of EGSM on the set of queries.

Competitor	DBLP	Enron	Github	Gowalla	Patents	Wikitalk
NoF3	1.08X	1.06X	1.11X	1.02X	1.27X	1.02X
DFS	78.63X	179.21X	145.02X	114.45X	135.68X	90.84X
BFS	1.41X	1.40X	1.46X	1.27X	1.63X	1.11X
Static	1.16X	1.05X	1.35X	1.12X	1.64X	1.07X
NoLB	1.00X	1.85X	2.59X	1.66X	1.14X	1.01X

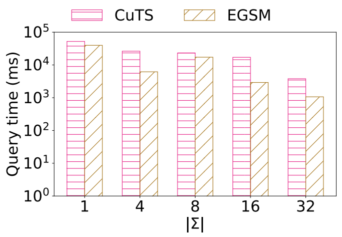
Load Balancing. In our EGSM, we propose two load-balancing methods to alleviate the inefficiency caused by heavy tasks. To demonstrate the benefit of our load balancing methods, we compare EGSM with and without load balancing. As shown in the fixth row of Table 4, the improvement in query time of EGSM over NoLB is up to 2.59X.

6.4 Scalability Evaluation

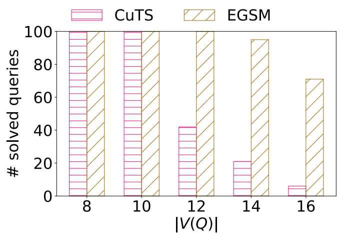
Previous experiments ran with our default settings. In this experiment, we compare CuTS and EGSM with different numbers of labels and query graph sizes to study their scalability. Figure 11a and 11b (resp., 11c and 11d) shows the number of solved queries and average query time over all solved queries on each dataset with various $|\Sigma|$ (resp., $|V(Q)|$). In particular, since CuTS cannot finish any query in the dataset with 1 or 4 labels given query graphs of size 12, we use query graphs of size 6 and 8 when $|\Sigma| = 0$ and 4, respectively. As shown in the figures, the performance generally worsens with the decrease of $|\Sigma|$ or increase of $|V(Q)|$. After that, we gather all solved queries of EGSM with different sizes, sorted by the number of matches. Then, we plot the query time of both methods in Figure 11g. All unsolved queries of CuTS are OM, labeled by a $\times$ on the top. As we can see, (1) The query time of competing algorithms is highly related to the number of matches; (2) EGSM consistently outperforms CuTS; and (3) CuTS runs out of memory on the queries with a large result size. Finally, we compare CuTS and EGSM in Figure 11e and 11f given various query graph shapes, such as paths, trees, sparse graphs ($d_{avg} < 3$), cycles, and dense graph ($d_{avg} \geq 3$). Cycle and dense queries are generally easy to solve due to the small result size. Nevertheless, the relative performance between CuTS and EGSM regarding all shapes is similar to that in the default settings.



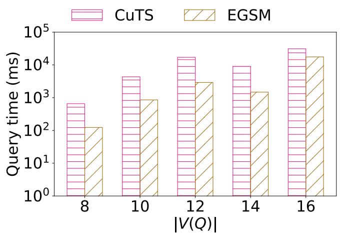
(a) Number of solved queries.



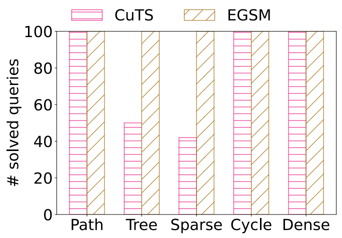
(b) Query time.



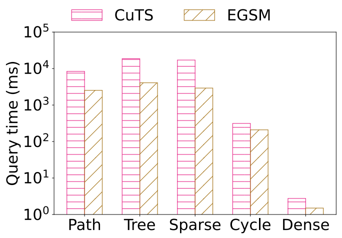
(c) Number of solved queries.



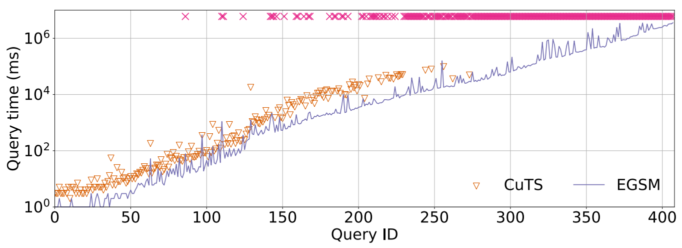
(d) Query time.



(e) Number of solved queries.



(f) Query time.



(g) Query Time.

Fig. 11. Scalability Evaluation.

7 CONCLUSION

In this paper, we study the problem of subgraph matching and propose an efficient GPU-accelerated approach, EGSM. We develop a data structure Cuckoo trie to maintain candidates and propose a three-step filtering procedure on the GPU. Furthermore, we design a BFS-DFS execution model and a memory management framework to reduce memory access and keep memory usage under capacity. We also present an adaptive ordering method and two kinds of load balancing approaches to further reduce the query time. Extensive experimental results show that EGSM significantly outperforms latest subgraph matching algorithms on GPUs.

ACKNOWLEDGMENTS

This work was partially supported by grant 16209821 from the Hong Kong Research Grants Council.

REFERENCES

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 431–446.

[2] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. 2009. Real-time parallel hashing on the GPU. *ACM Trans. Graph.* 28, 5 (2009), 154.

[3] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *Proc. VLDB Endow.* 11, 6 (2018), 691–704.

[4] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1447–1462.

[5] 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, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 1199–1214.

[6] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis E. Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinform.* 14, S-7 (2013), S13.

[7] Linchuan Chen, Xin Huo, Bin Ren, Surabhi Jain, and Gagan Agrawal. 2015. Efficient and Simplified Parallel Graph Processing over CPU and MIC. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*. IEEE Computer Society, 819–828.

[8] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 26, 10 (2004), 1367–1372. https://doi.org/10.1109/TPAMI.2004.75

[9] Vibhor Dodeja, Mohammad Almasri, Rakesh Nagi, Jinjun Xiong, and Wen-Mei Hwu. 2022. PARSEC: PARallel Subgraph Enumeration in CUDA. In *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022*. IEEE, 168–178.

[10] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*. USENIX Association, 371–384.

[11] Wenfei Fan. 2012. Graph pattern matching revised for social network analysis. In *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, Alin Deutsch (Ed.). ACM, 8–21.

[12] M. R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman.

[13] The PostgreSQL Global Development Group. 2022. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org. (2022), (Accessed Date: Sept 30th, 2022).

[14] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 1067–1082.

[15] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2020. Exploiting Reuse for GPU Subgraph Enumeration. *IEEE Trans. Knowl. Data Eng.* 34, 9 (2020), 4231–4244.

[16] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. 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. ACM, 1429–1446.

[17] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. 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*. ACM, 337–348.

[18] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*. IEEE Computer Society, 78–88.

[19] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1695–1698.

[20] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 925–937.

[21] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, Jeong-Hoon Lee, Seongyun Ko, and Moath H. A. Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 1231–1245.

[22] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 411–426.

[23] Raphael Kimmig, Henning Meyerhenke, and Darren Strash. 2017. Shared Memory Parallel Subgraph Enumeration. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 519–529.

[24] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable Subgraph Enumeration in MapReduce. *Proc. VLDB Endow.* 8, 10 (2015), 974–985.

[25] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2016. Scalable Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 10, 3 (2016), 217–228.

[26] Zhuohang Lai, Xibo Sun, Qiong Luo, and Xiaolong Xie. 2022. Accelerating multi-way joins on the GPU. *VLDB J.* 31, 3 (2022), 529–553.

[27] Sune Lehmann, Martin Schwartz, and Lars Kai Hansen. 2008. Biclique communities. *Physical review E* 78, 1 (2008), 016108.

[28] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (2014), (Accessed Date: Sept 30th, 2022).

[29] Yuchen Li, Qiwei Zhu, Zheng Lyu, Zhongdong Huang, and Jianling Sun. 2021. DyCuckoo: Dynamic Hash Tables on GPUs. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 744–755.

[30] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiaoli Li. 2017. Network Motif Discovery: A GPU Approach. *IEEE Trans. Knowl. Data Eng.* 29, 3 (2017), 513–528.

[31] Xiaojie Lin, Rui Zhang, Zeyi Wen, Hongzhi Wang, and Jianzhong Qi. 2014. Efficient Subgraph Matching Using GPUs. In *Databases Theory and Applications - 25th Australasian Database Conference, ADC 2014, Brisbane, QLD, Australia, July 14-16, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8506)*. Springer, 74–85.

[32] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704.

[33] Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F. Italiano, and Wook-Shin Han. 2021. Symmetric Continuous Subgraph Matching with Bidirectional Dynamic Programming. *Proc. VLDB Endow.* 14, 8 (2021), 1298–1310.

[34] MonetDB. 2022. MonetDB. https://www.monetdb.org. (2022), (Accessed Date: Sept 30th, 2022).

[35] Inc. Neo4j. 2022. Neo4j Graph Data Platform. https://neo4j.com. (2022), (Accessed Date: Sept 30th, 2022).

[36] Nvidia. 2022. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. (2022), (Accessed Date: Sept 30th, 2022).

[37] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algorithms* 51, 2 (2004), 122–144.

[38] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.

[39] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph Matching: on Compression and Computation. *Proc. VLDB Endow.* 11, 2 (2017), 176–188.

[40] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11, 12 (2018), 1876–1888.

[41] Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. 2014. PGX.ISO: Parallel and Efficient In-Memory Engine for Subgraph Isomorphism. In *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*. CWI/ACM, 5:1–5:6.

[42] Carlos R. Rivero and Hasan M. Jamil. 2017. Efficient and scalable labeled subgraph matching using SGMatch. *Knowl. Inf. Syst.* 51, 1 (2017), 61–87.

[43] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.* 1, 1 (2008), 364–375.

[44] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 625–636.

[45] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. 2019. Efficient Parallel Subgraph Enumeration on a Single Machine. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 232–243.

[46] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 1083–1098.

[47] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A Holistic Approach to Subgraph Query Processing. *Proc. VLDB Endow.* 14, 2 (2020), 176–188.

[48] Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. 2022. RapidFlow: An Efficient Approach to Continuous Subgraph Matching. *Proc. VLDB Endow.* 15, 11 (2022), 2415–2427.

[49] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. 2022. An In-Depth Study of Continuous Subgraph Matching. *Proc. VLDB Endow.* 15, 7 (2022), 1403–1416.

[50] Ha Nguyen Tran, Jung-Jae Kim, and Bingsheng He. 2015. Fast Subgraph Matching on Large Graphs using Graphics Processors. In *Database Systems for Advanced Applications - 20th International Conference, DASFAA 2015, Hanoi, Vietnam, April 20-23, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9049)*. Springer, 299–315.

[51] Julian R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (1976), 31–42.

[52] Leyuan Wang and John D. Owens. 2020. Fast Gunrock Subgraph Matching (GSM) on GPUs. *CoRR* abs/2003.01527 (2020). arXiv:2003.01527 https://arxiv.org/abs/2003.01527

[53] Haicheng Wu, Daniel Zinn, Molham Aref, and Sudhakar Yalamanchili. 2014. Multipredicate Join Algorithms for Accelerating Relational Graph Processing on GPUs. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2014, Hangzhou, China, September 1, 2014*. 1–12.

[54] 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 *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*. ACM, 69:1–69:14.

[55] Li Zeng, Lei Zou, M. Tamer Özsu, Lin Hu, and Fan Zhang. 2020. GSI: GPU-friendly Subgraph Isomorphism. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1249–1260.

[56] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. *Proc. VLDB Endow.* 8, 11 (2015), 1226–1237.

[57] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings (ACM International Conference Proceeding Series, Vol. 360)*. ACM, 192–203.

[58] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.* 3, 1 (2010), 340–351.

[59] Shijie Zhou, Rajgopal Kannan, Viktor K. Prasanna, Guna Seetharaman, and Qing Wu. 2019. HitGraph: High-throughput Graph Processing Framework on FPGA. *IEEE Trans. Parallel Distributed Syst.* 30, 10 (2019), 2249–2264.

[60] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 301–316.

Received October 2022; revised January 2023; accepted February 2023